Saarland University
Department of Computer Science

Systematic Characterization, Exploitation, and Protection of Microarchitectural Features

Dissertation
zur Erlangung des Grades
des Doktors der Ingenieurwissenschaften
der Fakultät für Mathematik und Informatik
der Universität des Saarlandes

von
Till Schlüter

Saarbrücken, 2025

Tag des Kolloquiums:	10. Februar 2026
Dekan:	Prof. Dr. Roland Speicher

Prüfungsausschuss:

Vorsitzender:	Prof. Dr. Jan Reineke
Berichterstattende:	Dr. Nils Ole Tippenhauer
	Prof. Dr. Andreas Zeller
	Dr. Michael Schwarz
Akademischer Mitarbeiter:	Dr. Rafael Dutra

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet. Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form in einem Verfahren zur Erlangung eines akademischen Grades vorgelegt.

Declaration of original authorship

I hereby declare that this dissertation is my own original work except where otherwise indicated. All data or concepts drawn directly or indirectly from other sources have been correctly acknowledged. This dissertation has not been submitted in its present or similar form to any other academic institution either in Germany or abroad for the award of any other degree.

Saarbrücken, October 2025

gez. / signed

(Till Schlüter)

Zusammenfassung

Moderne digitale Infrastrukturen benötigen schnelle und effiziente Prozessoren. Um hohe Leistungen zu erzielen, nutzen CPUs proprietäre mikroarchitekturelle Optimierungsmechanismen. Diese können unbeabsichtigt Sicherheitslücken verursachen, die grundlegende Plattformgarantien aushöhlen.

Diese Dissertation untersucht solche Optimierungen aus drei Perspektiven: Charakterisierung, Angreifbarkeit und Verteidigung. Zur Charakterisierung entwerfen wir das Konzept der *Leakage Templates*, um Mikroarchitektur-Seitenkanäle abstrakt darstellen zu können. Wir zeigen, wie solche Templates erstellt und genutzt werden können, um konkrete Seitenkanäle in Programmen aufzudecken. Außerdem entwerfen wir *FetchBench*, ein Framework zur Charakterisierung von Hardware-Prefetchern. Damit decken wir eine bislang unbekannte Prefetcher-Implementierung in einem kommerziellen Prozessor auf und demonstrieren, wie diese zum Ausleiten von Daten über Berechtigungsebenen hinweg ausgenutzt werden kann. Zur Verteidigung entwickeln wir *PreFence*, eine effiziente Gegenmaßnahme gegen Prefetching-Angriffe, die Prefetcher während der Ausführung sicherheitskritischen Codes selektiv deaktiviert. Zuletzt systematisieren wir Abwehrmaßnahmen gegen Mikroarchitektur-Angriffe anhand akademischer und nicht-akademischer Literatur. Wir erkennen eine Tendenz zur x86_64-Architektur und identifizieren Defizite, wo keine Gegenmaßnahmen gegen die von uns betrachteten Angriffe vorhanden sind.

Abstract

Modern digital infrastructure relies on fast, efficient processors. To achieve high performance, CPUs incorporate proprietary microarchitectural optimization features. These features, however, can inadvertently introduce security vulnerabilities that compromise fundamental platform guarantees.

This thesis examines microarchitectural features from three perspectives: characterization, exploitation, and protection. We develop novel methods to characterize security-critical microarchitectural properties, including the *Leakage Template*, an abstract representation of a side channel. We propose how Leakage Templates can be created and utilized to find instances of the channel they describe in real programs. We also design *FetchBench*, a framework for characterizing hardware prefetching mechanisms. Using FetchBench, we uncover an unknown Spatial Memory Streaming (SMS) prefetcher implementation in a commercial processor and show how it can be exploited to leak secrets across privilege domains. On the defensive side, we design and implement *PreFence*, an efficient mitigation against prefetching-based attacks that selectively disables prefetchers during security-critical code execution. Finally, we systematize defenses against attacks on microarchitectural features by analyzing academic and non-academic literature. We point out a bias towards the x86_64 architecture and identify gaps where no defenses are proposed against the attacks we consider.

Background of this Dissertation

This dissertation is based on the papers listed below. The content of these papers was not used in any other thesis submission.

Chapter 3 is based on research paper [P1], resulting from a collaboration with Ahmad Ibrahim, Hamed Nemati, Nils Ole Tippenhauer, and Christian Rossow. The general idea for reverse engineering microarchitectural behavior was initially proposed by Ahmad Ibrahim, Hamed Nemati, and Christian Rossow. The abstract concept of Leakage Templates was proposed by Nils Ole Tippenhauer. Plumber, our systematic framework to discover and evaluate Leakage Templates, was designed by Till Schlüter, Ahmad Ibrahim, and Hamed Nemati. The detailed implementation and evaluation of Plumber was contributed by Till Schlüter and Ahmad Ibrahim. Till Schlüter also added the design, implementation, and evaluation of Leakage Template matching in third-party program binaries. All authors participated in the writing and reviewing of the paper. Till Schlüter presented the work at ACM CCS 2022.

Chapter 4 is based on research paper [P2], which is a result of joint work with Amit Choudhari, Lorenz Hetterich, Leon Trampert, Hamed Nemati, Ahmad Ibrahim, Michael Schwarz, Christian Rossow, and Nils Ole Tippenhauer. The initial idea of the FetchBench framework was contributed by Till Schlüter and refined by Nils Ole Tippenhauer. The responsibility of designing, running, and evaluating identification and characterization tests as well as describing them in the paper was split up as follows: Till Schlüter was responsible for the stride and SMS prefetchers, while Amit Choudhari analyzed stream prefetchers and region-unbounded prefetchers. Lorenz Hetterich was responsible for the pointer array prefetcher and Leon Trampert took care of the pointer chasing prefetcher. Michael Schwarz contributed the test for the adjacent cache line prefetcher. Till Schlüter designed and implemented the common testing framework. Till Schlüter further designed, implemented, and evaluated the first case study (AES attack) in consultation with Nils Ole Tippenhauer. The second case study (TrustZone covert channel) was jointly designed by Till Schlüter and Amit Choudhari in coordination with Nils Ole Tippenhauer and Christian Rossow. It was implemented and evaluated by Amit Choudhari. Ahmad Ibrahim, Hamed Nemati, Michael Schwarz, Christian Rossow, and Nils Ole Tippenhauer actively provided feedback and participated in writing and reviewing the text. Till Schlüter presented the work at ACM CCS 2023.

Chapter 5 is based on research paper [P3], which is the outcome of a collaboration with Nils Ole Tippenhauer. The systematic analysis of prior attack and defense schemes was performed by Till Schlüter. Till Schlüter also proposed the initial design of the countermeasure, which was refined in dialog with Nils Ole Tippenhauer. While Till Schlüter was primarily responsible for implementation, evaluation, and paper writing, Nils Ole Tippenhauer actively provided feedback and participated in writing and reviewing the text. Till Schlüter presented the work at IEEE EuroS&P 2025.

Chapter 6 is based on [P4], which is a collaborative project with Nils Ole Tippenhauer that is currently in preparation. The initial idea of the SoK paper was contributed by Till Schlüter and Nils Ole Tippenhauer. Till Schlüter performed the systematic literature review, built the dataset that is the foundation of the work, and aggregated and evaluated the collected data to answer the research questions. While Till Schlüter

was primarily responsible for paper writing, Nils Ole Tippenhauer actively provided feedback and participated in reviewing the text.

[P1] Ibrahim, A., Nemati, H., Schlüter, T., Tippenhauer, N. O., and Rossow, C. "Microarchitectural Leakage Templates and Their Application to Cache-Based Side Channels." In: *Proceedings of the ACM Conference on Computer and Communications Security (CCS).* 2022.

[P2] Schlüter, T., Choudhari, A., Hetterich, L., Trampert, L., Nemati, H., Ibrahim, A., Schwarz, M., Rossow, C., and Tippenhauer, N. O. "FetchBench: Systematic Identification and Characterization of Proprietary Prefetchers." In: *Proceedings of the ACM Conference on Computer and Communications Security (CCS).* 2023.

[P3] Schlüter, T. and Tippenhauer, N. O. "PreFence: A Fine-Grained and Scheduling-Aware Defense Against Prefetching-Based Attacks." In: *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P).* 2025.

[P4] Schlüter, T. and Tippenhauer, N. O. "SoK: Countermeasures Against Attacks on Microarchitectural Features." In preparation.

Further Contributions of the Author

During my doctoral research, I also contributed to the following papers. I did not include paper [S1] in this thesis as it overlaps with the content of my master's thesis.

[S1] Schlüter, T. and Lemke-Rust, K. "Differential Analysis and Fingerprinting of ZombieLoads on Block Ciphers." In: *Proceedings of the International Conference on Smart Card Research and Advanced Applications (CARDIS).* 2020.

[S2] Lin, Y.-D., Schlüter, T., Narimani, P., Abbasi, A., and Tippenhauer, N. O. "PowerFuzz: Efficient Power Side-Channel Guided Fuzzing of Proprietary Embedded Systems with Execution Divergence Graphs." In preparation.

Acknowledgments

First and foremost, I would like to thank my advisor, Nils Ole Tippenhauer, for giving me the opportunity to pursue my doctorate in his research group. I deeply appreciate the freedom and guidance he provided, as well as his constant support, constructive feedback, and valuable advice that I could always rely on. What I have learned from him will continue to shape my work and thinking long after the PhD. Thank you, Nils!

I would also like to thank the examiners of this thesis, Andreas Zeller and Michael Schwarz. Thank you for your time.

In addition, I want to thank all my co-authors, listed in alphabetical order: Ahmad Ibrahim, Amit Choudhari, Christian Rossow, Hamed Nemati, Kerstin Lemke-Rust, Leon Trampert, Lorenz Hetterich, and Michael Schwarz. I have truly enjoyed our collaborations. Thank you for your excellent work and dedication to our joint projects.

I also want to express my gratitude to the CISPA Helmholtz Center for Information Security, Saarland University, and the Saarbrücken Graduate School of Computer Science. They have shaped an exceptional research environment that I could benefit from. It has been a privilege to contribute to the growth of this young research center, and I look forward to seeing its continued development.

My doctoral journey would not have been the same without the current and former members of Nils' group, again listed in alphabetical order: Alessandro, Burak, Felix, John, Joshua, Sahil, Simeon, and Yu-De. I will fondly remember our discussions, travels to conferences and other places, and the many experiences we shared. I also greatly enjoyed my time with my fellow members of the saarsec CTF team. I will miss our meetings and events on campus.

Finally, I owe my deepest gratitude to my parents, Gabi and Frank, for their unwavering support, trust, and the example they set for me. Thank you so much.

Contents

1 Introduction **1**
1.1 Research Questions and Contributions 3
1.2 Artifacts 5

2 Background **7**
2.1 Processor Design and Microarchitecture 9
2.2 Memory Hierarchy and Caching 9
2.3 Prefetching 11
2.3.1 Software Prefetching 11
2.3.2 Hardware Prefetching 11
2.4 Cache-Timing Side Channels 12
2.5 Simultaneous Multithreading (SMT) 13

3 Microarchitectural Leakage Templates and Their Application to Cache-Based Side Channels **15**
3.1 Introduction to Microarchitectural Leakage Templates 17
3.2 Preliminaries 18
3.2.1 Notations 18
3.2.2 Side Channel Abstract Model Validator (Scam-V) 19
3.2.3 Prevction 20
3.3 Leakage Templates 20
3.3.1 Goal and Motivation for Leakage Templates 20
3.3.2 Motivating Example I: Caching 22
3.3.3 Motivating Example II: Prevction 22
3.3.4 Definition of Leakage Templates 23
3.4 Deriving Leakage Templates: The Plumber Framework 23
3.4.1 High-Level Framework Design 24
3.4.2 Definition of the Generative Testcase Specification (GTS) 25
3.4.3 Generative Testcase Specification (GTS) Language 25
3.4.4 Framework Components 27
3.5 Application of Microarchitectural Leakage Templates 31
3.5.1 Framework Implementation and Experimental Setup 31
3.5.2 Case Study: Cache Eviction Strategy 31
3.5.3 Case Study: Prevction 34
3.5.4 Case Study: Prefetching 36
3.6 Matching Leakage Templates in Binaries 38

3.7 Novel Leakage Primitives . . . 40
3.7.1 Previction With Shared Memory (PR_FR) . . . 41
3.7.2 Previction Without Shared Memory (PR_PP) . . . 43
3.7.3 Prefetching Control-Flow Leakage (PRF_CF) . . . 43
3.7.4 Prefetching on an Interrupted Sequence (PRF_IS) . . . 43
3.7.5 Prefetching on an Outstanding Sequence (PRF_OS) . . . 43
3.8 Related Work to Leakage Templates and Plumber . . . 44
3.8.1 Side-Channel Discovery and Characterization . . . 44
3.8.2 Identifying Side Channels in Existing Software . . . 45
3.8.3 Related Side Channels . . . 45
3.9 Discussion of Microarchitectural Leakage Templates . . . 46
3.10 Conclusion on Microarchitectural Leakage Templates . . . 47

4 FetchBench: Systematic Identification and Characterization of Proprietary Prefetchers 49
4.1 Introduction to FetchBench . . . 51
4.2 Systematic Classification of Hardware Prefetchers . . . 53
4.2.1 Hardware Prefetcher Taxonomy . . . 53
4.2.2 FetchBench: Identifying and Characterizing Prefetchers . . . 55
4.3 Characterizing ARMv8 and x86_64 Prefetchers . . . 63
4.3.1 Implementation . . . 63
4.3.2 Experimental Setup . . . 63
4.3.3 Experimental Results . . . 65
4.4 Exploiting Prefetchers . . . 66
4.4.1 Case Study: AES Attack . . . 67
4.4.2 Case Study: ARM TrustZone Covert Channel . . . 71
4.5 Related Work on Prefetcher Reverse Engineering and Exploitation . . . 73
4.5.1 Prefetcher Reverse Engineering . . . 73
4.5.2 Prefetcher Exploitation . . . 74
4.6 Discussion of FetchBench . . . 74
4.7 Conclusion on FetchBench . . . 75

5 PreFence: A Fine-Grained and Scheduling-Aware Defense Against Prefetching-Based Attacks 77
5.1 Introduction to PreFence . . . 79
5.2 Defending Against Prefetching-Based Attacks . . . 80
5.2.1 System and Attacker Model . . . 80
5.2.2 Research Questions and Challenges . . . 80
5.3 Systematization of Attacks . . . 81
5.3.1 Stages of Prefetching-Based Attacks . . . 81
5.3.2 Scopes of Prefetching-Based Attacks . . . 82
5.3.3 Prefetch Attack Systematization . . . 84
5.4 PreFence: Design and Implementation . . . 89
5.4.1 Design Considerations . . . 89
5.4.2 PreFence Design . . . 90

5.4.3 PreFence Implementation . . . 92
5.5 Evaluation of PreFence . . . 93
5.5.1 Evaluation Environments . . . 93
5.5.2 Prerequisite: Disabled Prefetcher Behavior . . . 94
5.5.3 Efficacy: Protecting OpenSSL . . . 95
5.5.4 Efficacy: Protecting MbedTLS . . . 96
5.5.5 Efficiency: Overhead on Context Switch and System Call . . . 98
5.5.6 Efficiency: Non-Critical Workloads (Scenarios 1 and 2) . . . 100
5.5.7 Efficiency: Security-Critical Workloads (Scenario 3) . . . 100
5.5.8 Summary of Results . . . 104
5.6 Discussion on PreFence . . . 104
5.6.1 Applicability to Hardware Prefetcher Types . . . 104
5.6.2 Automating System Call Placement . . . 105
5.6.3 Applicability to Different Scopes . . . 106
5.6.4 Applicability to Other Processors and Flags . . . 106
5.6.5 Applicability to Other Prefetching Types . . . 107
5.6.6 Covert Channels . . . 107
5.7 Related Work on Defenses Against Prefetching-Based Attacks . . . 108
5.7.1 Related Work on Software Defenses . . . 108
5.7.2 Related Work on Hardware Defenses . . . 110
5.8 Conclusion on PreFence . . . 111

6 Systematization of Knowledge: Countermeasures Against Attacks on Microarchitectural Features 113
6.1 Introduction on Countermeasures . . . 115
6.2 Systematization Scope . . . 116
6.3 Systematization Methodology . . . 118
6.4 Systematization Results . . . 120
6.4.1 Architecture Gaps . . . 120
6.4.2 Mitigation Gaps . . . 121
6.4.3 Implementation Gaps . . . 123
6.5 Related Systematization Work . . . 125
6.6 Discussion and Conclusion on the Systematization . . . 126

7 Conclusion 129
7.1 Summary of Contributions . . . 131
7.2 Future Directions . . . 132

Bibliography 135

A Appendix 151
A.1 Appendix to Chapter 4 (FetchBench) . . . 153
A.1.1 Processors Under Evaluation . . . 153
A.1.2 Experiment Success Rates . . . 153
A.2 Appendix to Chapter 5 (PreFence) . . . 156
A.2.1 Prefetch Disable Flags on Various Processors . . . 156

A.3 Appendix to Chapter 6 (Systematization of Knowledge) 160
A.3.1 Mitigation Gaps: Scope Mapping Analysis Tables 160
A.3.2 Implementation Gaps: Implementation Table 164

List of Figures

2.1 Example of a memory hierarchy with a 4-way set-associative cache . . . 10

3.1 Physical address mapping . . . 19
3.2 Counterexample for previction on ARM Cortex-A53 . . . 20
3.3 Leakage Templates: Proposed approach . . . 21
3.4 Leakage Template for caching . . . 23
3.5 Plumber's components . . . 24
3.6 Flow chart of the analyzer function for previction . . . 30
3.7 Case study Leakage Templates . . . 33
3.8 Code examples for covert channels . . . 41
3.9 Histograms comparing the load latency of covert channels . . . 42

4.1 FetchBench: Proposed approach . . . 52
4.2 Prefetcher taxonomy . . . 54
4.3 FetchBench framework . . . 56
4.4 Stride prefetcher tests . . . 57
4.5 Stream prefetcher tests . . . 58
4.6 Pointer array test . . . 59
4.7 SMS prefetcher tests . . . 61
4.8 Region-unbounded prefetcher identification test . . . 62
4.9 Loads and prefetches in victim/attacker process . . . 68
4.10 Synchronization of attacker and victim processes . . . 70
4.11 Covert channel based on the SMS prefetcher . . . 72

5.1 Stages of prefetching-based side channels . . . 81
5.2 Overview of the sequence of activities in prefetching-based attacks . . . 83
5.3 Integration of PreFence with the scheduler . . . 91
5.4 Prefetcher behavior when trained while disabled . . . 94
5.5 Results of the OpenSSL efficacy experiment . . . 96
5.6 Results of strengthened attack on MbedTLS AES . . . 97
5.7 Context switch overhead on the stock kernel and on our patched kernel 99
5.8 Overhead of a system call that sets or clears the *prefetch_disable* bit . . 99
5.9 SPEC CPU 2017 benchmark results . . . 101
5.10 Lighttpd benchmark results . . . 103

6.1 Distinct processors and processor vendors per paper . . . 121
6.2 Scope matching analysis example . . . 122

A.1 Prefetcher identification and characterization experiment success rates . 155

List of Tables

3.1 Summary of notations . . . 19
3.2 Bit tables for previction testcases . . . 29
3.3 Approximate total experiment execution times . . . 32
3.4 Example permutation outcome . . . 35
3.5 Examples of Generative Testcase Specifications . . . 35
3.6 Template matching confusion matrix . . . 40
3.7 Transmission and error rates of covert channels . . . 42

4.1 Prefetcher identification and characterization results . . . 64

5.1 Prefetching-based attacks in prior work . . . 84

6.1 Corpus of documents . . . 117
6.2 Results of the scope matching analysis . . . 124

A.1 Hardware platforms under evaluation in Chapter 4, identified prefetchers, and test runtime . . . 154
A.2 Prefetch disable flags on Intel processors . . . 157
A.3 Prefetch disable flags on AMD processors . . . 158
A.4 Prefetch disable flags on ARM processors . . . 159
A.5 Attacks and attack scopes . . . 160
A.6 Defenses and defense scopes . . . 161
A.7 Defense implementations . . . 164

List of Algorithms

3.1 Algorithm to find an eviction strategy 32

1
Introduction

Computer systems are an integral part of everyday life in our modern society, be it in the form of personal computers, phones, embedded devices, or cloud systems. We entrust these systems with our most personal information, such as private messages, pictures, and health metrics, expecting that this data is stored and processed securely.

The foundation of every such computer system is a microprocessor, which can conceptually be considered a simple building block that takes data and instructions as inputs, performs the requested computations, and returns the result as an output. In reality, however, modern processors are much more complex. Internally, on the microarchitectural level, they perform optimizations that aim to speed up computations in various ways. For instance, processors may execute instructions out of order, speculatively predict the next instructions to execute, or predict the next pieces of data to fetch from memory (so-called *prefetching*). All these optimizations, however, are supposed to be invisible from the outside and especially transparent to the software running on the processor. The processor attempts to maintain the appearance that all instructions and memory operations are being processed in-order and following the intended control flow path.

Unfortunately, recent work has revealed that processors sometimes do not fully succeed in keeping up this illusion. In particular, many such optimizations also introduce side-channel vulnerabilities, putting the processed data at risk. For instance, out-of-order execution can lead to (transiently) skipping permission checks and leaking the content of internal buffers [88, 133, 148, 149, 151], speculative execution can lead to the unintended (transient) execution of code with malicious side-effects [22, 57, 78, 79, 81, 93, 129, 157, 158], and side-effects of prefetching may reveal previous program activity or data [28, 29, 30, 54, 131, 137, 160]. Thus, insecure microarchitectural performance optimizations in processors introduce a security risk at the foundation of a computer system and break assumptions that higher-level security mechanisms rely on.

In order to be able to fully trust a computer system, it is therefore imperative to understand its microarchitectural features and the security characteristics resulting from them. Yet, processor designers typically do not publish such detailed information, making any security assessment challenging.

1.1 Research Questions and Contributions

In this thesis, we approach this fundamental problem from three different angles. First, we present novel methodologies that allow us to systematically *characterize* security-relevant properties of microarchitectural features of a given processor. Second, we apply our methodologies in case studies and find that some microarchitectural features can be *exploited* in novel ways. Finally, we take on the defensive role and explore how vulnerable microarchitectural features can be *protected*, and we systematize existing protective mechanisms to assess their completeness.

Leakage Templates. A common approach to discover vulnerabilities in microarchitectural features is searching for code examples that behave unexpectedly, for example with respect to the execution time of individual instructions in different scenarios. Such unexpected behavior hints towards the existence of a microarchitectural side channel, i.e., a hidden information flow channel that may be exploitable to leak information.

However, it remains challenging to abstract from the single conspicuous code example and determine the general preconditions and characteristics that are causing this behavior. This generalization step is necessary to fully understand the side channel, its security implications, and determine its prevalence in existing software. In this context, we ask our first research question:

> **RQ1:** How can we systematically abstract from a concrete example of side-channel behavior in a program to learn its generalized characteristics, and how can we find other instances of the same side channel in other programs?

In Chapter 3, we answer this question by introducing the novel concept of *Leakage Templates*, which allow us to describe such behavior in an abstract and generalized way. We further introduce a methodology to derive Leakage Templates, along with our novel framework *Plumber*, which facilitates this process. These foundations enable us to investigate microarchitectural side channels caused by previction and hardware prefetching. Finally, we show how a Leakage Template can be used to find other instances of the same side channel in other programs.

FetchBench: Prefetcher Characterization. Inspired by our initial investigation of hardware prefetching, we then continue with a broader analysis of this feature. Given that many different prefetching strategies exist and processor manufacturers typically do not disclose security-relevant implementation details, we ask:

> **RQ2:** How can we systematically identify and characterize hardware prefetchers and their security implications in real-world processors?

We answer this question in Chapter 4. First, we introduce a novel taxonomy of prefetching strategies based on literature, patents, and known implementations. We then design and implement *FetchBench*, a framework to systematically identify and characterize prefetcher implementations on a given processor, and apply our approach to 19 commercial CPUs. As one notable result, we identify a previously unknown implementation of a Spatial Memory Streaming (SMS) prefetcher in an ARM processor. We uncover that this implementation does not separate its internal state between privilege domains, allowing for information leakage across processes as well as from trusted execution to the normal operating system.

PreFence: Prefetch Attack Countermeasure. Alarmed by this finding, we proceed to systematically investigate countermeasures against prefetching-based attacks. In Chapter 5, we ask:

> **RQ3:** How can we systematically defend against prefetching-based attacks?

In particular, we are interested in solutions that go beyond the trivial ones: Simply disabling a vulnerable microarchitectural feature is surely effective, but also expensive in terms of performance. Thus, we systematically survey existing attacks, identify similarities, and design, implement, and evaluate *PreFence*, a novel software-based countermeasure that features low performance overhead and can be applied to vulnerable

systems without requiring expensive hardware changes.

SoK on Countermeasures. Finally, in Chapter 6, we broaden our scope again. In a preliminary *Systematization of Knowledge* work, we ask:

> **RQ4:** How can we systematically identify gaps in the academic and non-academic consideration of defenses against microarchitectural attacks?

Specifically, we systematize academic and non-academic literature discussing microarchitectural predictors, including branch prediction and prefetchers, in offensive and defensive contexts. We extract information on the attacks and defenses discussed in those documents, as well as relations between them, to build a comprehensive dataset for further analysis. We identify a bias towards the x86_64 architecture and especially Intel processors. In addition, we find that almost all attacks in our dataset can be fully mitigated by proposed defenses.

1.2 Artifacts

For every research paper represented in this thesis, we also provide an accompanying artifact repository that eases reproduction and future extension of our work. All artifacts are available online:

- Chapter 3: `https://github.com/scy-phy/plumber`
- Chapter 4: `https://github.com/scy-phy/FetchBench`
- Chapter 5: `https://github.com/scy-phy/PreFence`
- Chapter 6: `https://github.com/scy-phy/sok-defenses`

2
Background

In this chapter, we provide the essential background for the remainder of this thesis. We first explain how modern processors are specified and how their behavior is determined by their microarchitecture. We then go into more detail on the memory hierarchy in modern computer systems, prefetching, cache-timing attacks, and simultaneous multithreading.

2.1 Processor Design and Microarchitecture

ISA. To the outside, every computer processor provides a well-defined interface that describes how software is supposed to interact with it. This interface is known as the *instruction set architecture (ISA)*. For example, the ISA specifies the available instructions, how they are encoded, and the number and size of available registers [53].

Microarchitecture. The inner workings of the processor, however, are not as well-defined. As long as the processor maintains its ISA-defined outside interface, the internal implementation can be designed as the manufacturer sees fit. This internal implementation is typically referred to as the *microarchitecture* of a processor [52].

Different processors may implement the same ISA using different microarchitectures. For example, both Intel and AMD offer processors that implement the x86_64 ISA, with probably vastly different internal implementations [52]. Similarly, the internal design of an Intel processor from the early 2000s will differ from a more recent Intel microarchitecture, even though the ISA is still (largely) the same.

Microarchitectural Features. The microarchitecture determines how instructions are processed and executed to compute the results requested by software. To accelerate this process, processor manufacturers introduce various microarchitectural features for performance optimization. For instance, they use caches and hardware prefetching to reduce the effective memory latency, or perform branch prediction to speculatively resolve situations where one instruction depends on the result of a slower earlier one [52]. We describe caches and hardware prefetching in more detail in Sections 2.2 and 2.3, respectively.

2.2 Memory Hierarchy and Caching

Many instructions that a processor executes involve memory interactions. From a software's perspective, the memory interface is straightforward: The software specifies an address and instructs the processor to perform a load or store operation. However, as is customary in computing, the real complexity of the memory subsystem is hidden under layers of abstraction that obscure low-level implementation details.

Memory Hierarchy. Data is stored in Dynamic Random Access Memory (DRAM), which is volatile and relatively inexpensive. Unfortunately, at the scale of the processor's clock speed, operations on data in DRAM are slow. For this reason, modern computer systems implement a hierarchy of cache stages that buffer parts of that data, making them available for faster access to the processor and thus reducing memory latency. With every level in this hierarchy, caches become faster, but also more expensive and thus smaller [52, 53].

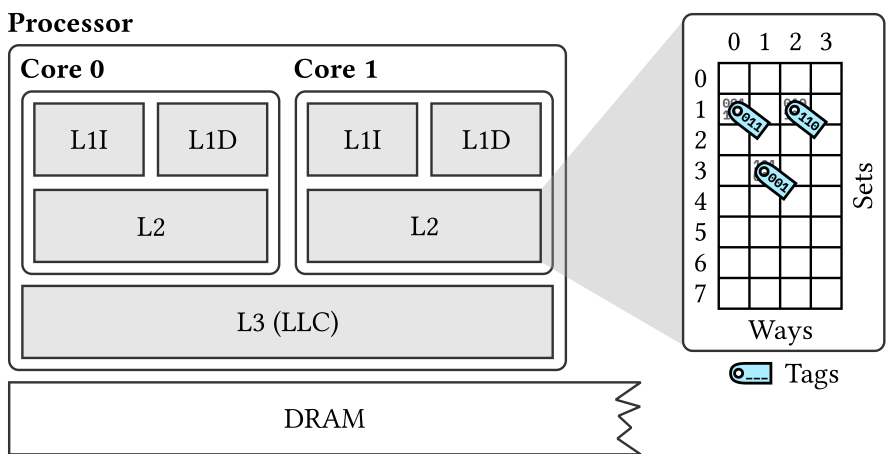

Figure 2.1: Example of a memory hierarchy with a 4-way set-associative cache

Figure 2.1 shows an example of a memory hierarchy with three cache levels. Each processor core features its own level-1 cache, which is split into a data cache (L1D) and an instruction cache (L1I). The next layer comprises per-core unified level-2 (L2) caches, storing both data and instructions. Finally, there is a single unified level-3 (L3) cache that is shared among all cores. This cache is also known as last-level cache (LLC). Most modern Intel x86_64 processors follow this general structure [69]. In contrast, many ARM processor designs (such as Cortex-A53 and Cortex-A72, which we will use later in this thesis) only feature two levels of caches [8, 9]. These designs omit the layer of unified per-core caches (L2 in Figure 2.1).

When a memory operation can be served from a cache, this is referred to as a *cache hit*. A memory operation on data that is not cached has to be served from DRAM. This event is called a *cache miss*.

Cache Organization. Caches typically store data in blocks of fixed size. These blocks are also referred to as *cache lines*. A typical cache-line size is 64 bytes. In addition, caches are usually set-associative. This means that every memory location can only be stored in one particular *cache set* within the whole cache. Every cache set may contain one or more cache lines at the same time. The number of lines per set is commonly referred to as *ways*. As a concrete example, Figure 2.1 shows a 4-way set-associative cache with 8 sets. Each cache line in a cache is identified by a *tag*. In some microarchitectures, the set index and the tag can directly be derived from particular bits of the physical address of a memory location.

Exploiting Temporal and Spatial Locality. An ideal cache would always contain the most relevant data for the processor at any given moment, resulting in a 100% hit rate. Although this goal cannot be achieved in practice, there are several common strategies that aim to keep the hit rate as high as possible. Many such strategies rely on the assumptions of temporal and spatial locality [52].

The assumption of temporal locality states that many programs tend to access the same data multiple times in succession. Thus, it is usually beneficial to keep recently accessed data in the cache. However, whenever a new cache line is to be inserted into an already full cache set, one of the present cache lines must be selected for eviction.

The selection is governed by a *replacement policy*. A common example of a replacement policy that follows the temporal locality assumption is *least recently used (LRU)*, which evicts the cache line from a set that has not been accessed for the longest time [52].

The assumption of spatial locality states that, once a certain address has been accessed, a program will likely access nearby addresses as well. This is one reason why caches operate on blocks of data (i.e., cache lines) and effectively always fetch multiple words of data at a time [52]. Another optimization that exploits spatial locality is hardware prefetching, which we discuss in Section 2.3.2.

2.3 Prefetching

A prefetching mechanism brings data that is likely to be accessed in the near future from DRAM into a cache. We distinguish between software prefetching and hardware prefetching. In this section, we discuss software prefetching only for completeness, as we only discuss hardware prefetching in the following chapters. Whenever we refer to *prefetching* as a general term, we typically mean hardware prefetching.

2.3.1 Software Prefetching

Software prefetching is based on instructions that can be placed in application code by a programmer or a compiler. These instructions indicate which memory addresses will soon be accessed, so that the CPU can bring them into the cache before they are used. For example, the x86_64 instruction set provides the `PREFETCH` instruction to prefetch data from a specific address into specific cache levels [70]. The ARM A64 instruction set contains a similar instruction (`PRFM`) [14].

Exploitation. As prior work has shown, software prefetching instructions can be exploited in different ways. Gruss et al. [42] present an attack that exploits the timing behavior of software prefetch instructions on Intel CPUs to infer whether an inaccessible page is physically backed. Lipp, Gruss, and Schwarz [85] show that the software prefetch instruction on AMD additionally leaks the state of the prefetched page in the translation look-aside buffer (TLB). Guo et al. [49] exploit the x86_64 prefetch instruction `PREFETCHW` to leak cryptographic keys through the data-dependent access pattern of an application.

2.3.2 Hardware Prefetching

Hardware prefetching is a microarchitectural feature of the memory subsystem that automatically determines blocks of memory in DRAM that are likely to be accessed soon. These blocks are then brought into cache lines in the cache ahead of time. To do so, a hardware prefetcher observes the processor's memory operations at runtime, detects patterns, and extrapolates from them to predict future target addresses. This ideally increases the cache hit rate and reduces overall memory latency [53]. From an application's point of view, hardware prefetching is a completely transparent mechanism and requires no active interaction with the prefetcher.

Address-Based Prefetching. Address-based prefetchers monitor the accessed memory *addresses* and attempt to identify specific access patterns to predict the next

addresses to be accessed. The data at these predicted addresses is then fetched into the cache. This form of hardware prefetching is based on the assumption of spatial locality: It assumes that regular memory access patterns exist [35].

A common address-based prefetcher is the stride prefetcher [21, 35]. It detects access sequences whose addresses are separated by a fixed distance (the *stride*) from each other. We call such sequences *prefetch streams*. Once a stream is detected, the prefetcher extrapolates from the observed pattern and fills the cache with subsequent cache lines that are multiples of the stride away from the initial access locations.

When software performs secret-dependent memory accesses, those accesses may leave traces in the prefetcher's internal state [30]. Afterwards, the prefetcher's predictions may correlate with the secret. An attacker can then observe traces of the prefetcher's activities in the cache and infer the secret.

Data Memory-Dependent Prefetching (DMP). Instead of evaluating patterns in memory addresses, the class of data memory-dependent prefetchers (DMPs) evaluates the *values* that are loaded from memory to predict future target addresses.

For example, a DMP could inspect all data that is loaded from DRAM, scan it for pointers, and dereference those pointers ahead of time [28, 131].

DMPs cause leakage when they observe secret-dependent values and process them [28, 131]. Again, attackers can infer the secrets by observing the prefetcher's behavior and the resulting modifications to the cache state.

Implementation. CPUs often implement multiple hardware prefetchers that use different prefetching strategies and complement each other. However, implementation details on most prefetchers are not publicly available. We discuss various prediction strategies and a novel methodology to identify and characterize them in real-world CPUs in Chapter 4.

2.4 Cache-Timing Side Channels

Side Channels. Over the past decades, researchers have devised various means to exfiltrate secrets from computing devices through indirect communication channels, so-called *side channels*. Side channels occur when a measurable parameter of a processor, for example its power consumption [50, 80, 87, 94], electromagnetic radiation [19, 24, 37, 51, 91], or timing behavior [44, 117, 145, 161] becomes dependent on the control flow or data that is processed.

Cache-Timing Side Channels. When a program loads data from a memory address, the processor first checks whether the data is present in a cache (cache hit) or not (cache miss). In case of a hit, the load instruction is significantly faster. Otherwise, the data needs to be fetched from DRAM, which takes more time. This difference in timing behavior is the foundation of cache-timing side channels: An attacker measures the execution time of a load instruction to learn whether a cache line is currently present in cache or not. Common attack primitives that exploit cache-timing side channels are Flush+Reload [161] and Prime+Probe [114].

Flush+Reload. The Flush+Reload side channel [161] relies on shared memory. As a simple example, we consider the case where an attacker process and a victim process share a single cache line, and the victim process performs secret-dependent accesses to

that cache line. The attacker process first flushes the shared cache lines from the cache. Second, the victim process executes and, depending on a secret bit, either loads the flushed cache lines or not. Finally, the attacker reloads these cache lines and measures the execution time of the load instruction. Depending on the reload latency, the attacker decides whether a cache hit or miss occurred, which tells them the value of the secret bit.

Prime+Probe. The Prime+Probe side channel [114] does not rely on shared memory. Instead, the attacker process first fills the cache with its own data. Next, the victim process executes and, depending on a secret bit, either loads a specific cache line or not. If the cache line is loaded, one of the attacker's lines is evicted. Finally, the attacker uses a timed load instruction to test whether or not their own data is still present in the cache, allowing them to infer the secret bit value.

2.5 Simultaneous Multithreading (SMT)

Traditionally, every processor core executes exactly one program thread at a time. On a system that supports multithreading, multiple threads take turns in using the core. To switch from one thread to another, the state of the current thread needs to be stored in memory, and the state of the next thread needs to be restored to the processor registers. This procedure is known as *context switching* and handled by the scheduler, a component of the operating system [142].

Simultaneous Multithreading (SMT) [146] is a concept that aims to better utilize the resources of a processor core. It schedules multiple threads on a single processor core at the same time, based on the insight that a single thread often cannot utilize all the processing units available in a core. SMT has been adopted by major processor vendors such as Intel (branded *HyperThreading*) [96], AMD [2], and ARM [11]. Their implementations expose one physical processor core as multiple independent logical cores to the operating system. We refer to logical cores that are backed by the same physical core as *sibling cores*. The operating system schedules threads on logical cores in the same way as it would on a physical core [96].

On a non-SMT system, a thread has exclusive access to the resources of a processor core while it is scheduled. In contrast, on an SMT system, instructions from parallel threads that are scheduled on sibling cores are processed concurrently and share processor resources, potentially also the prefetcher. We emphasize that, as a result of SMT, instructions issued by multiple processes can run on the same physical core without requiring a context switch.

3

Microarchitectural Leakage Templates and Their Application to Cache-Based Side Channels

3.1 Introduction to Microarchitectural Leakage Templates

In recent years, many side-channel vulnerabilities have been discovered that exploit underspecified or undocumented microarchitectural features [1, 45, 79, 88, 108, 144]. These findings have fueled the development of techniques that discover new instances of side-channel leakage in a given processor [41, 103, 105, 109, 122, 153]. For example, Weber et al. [153] introduce the Osiris fuzzing framework, which randomly generates and measures instruction sequences to identify those that constitute timing-based side channels. Nemati et al. [105] propose Scam-V, which compares a processor's behavior at program runtime with the expected behavior according to a model. Any instances of unexpected behavior indicate potential side channels on the real hardware. Moghimi et al. [103] present Transynther, an approach to synthesize Meltdown-type attacks. They vary known attack patterns to create candidate attack code snippets and then evaluate whether these snippets leak data.

All of these approaches have in common that they automate the search for new (variants of) side channels. However, they generate singular vulnerable code examples instead of finding a generic description of the underlying side channel. Unfortunately, a single code example is insufficient for understanding a side channel in detail, as it does not describe the channel's general prerequisites and activation criteria. A more general understanding of the channel is desirable though, as it allows identifying and eliminating related side-channel vulnerabilities in real-world applications. However, gaining this understanding is a hard problem, as it requires detailed understanding of the microarchitectural feature causing the side channel.

Leakage Templates. In this chapter, we address this problem by introducing *Leakage Templates*, an abstract description of side channels. Leakage Templates consist of a generalized code sequence and a set of relations on input parameters (e.g., instruction operands) that describe the prerequisites for a specific leakage behavior to occur.

Research Questions. Using our concept of Leakage Templates, we address the high-level research question of this chapter (RQ1, as introduced in Chapter 1): *How can we systematically abstract from a concrete example of side-channel behavior in a program to learn its generalized characteristics, and how can we find other instances of the same side channel in other programs?* We break this question down into the following two parts:

RQ1.1: How can we learn generic Leakage Templates for largely-undocumented leakage behavior?

RQ1.2: How can we use Leakage Templates to find side-channel vulnerabilities in existing programs?

Proposed Approach. We answer RQ1.1 by designing and implementing *Plumber*, our novel framework to facilitate deriving Leakage Templates, leveraging instruction generation, instruction operand mutation, and statistical analysis. Plumber explores the architectural space through the execution of program-input pairs and analyzes the resulting microarchitectural states (focusing on traces in the cache). We design a

domain-specific language that simplifies the generation of a large number of instruction sequences (i.e., programs) and mutating their operands. Further, we use a statistical analysis approach to classify microarchitectural states and to extract relations on inputs.

To validate our approach, we use it to investigate cache-related behavior of two ARM processors. In particular, we study cache eviction strategies for the ARM Cortex-A72 and two microarchitectural features of the ARM Cortex-A53 processor: *previction* [105] and *prefetching*. Leveraging Plumber, we derive Leakage Templates for these case studies. For the eviction experiment, the derived Template establishes eviction strategies for the L1 data cache. In the case of previction, the extracted Templates reveal conditions under which address bits of memory loads can be leaked. For prefetching, we leverage Plumber to discover parameters such as the minimal number of loads to trigger prefetching, the impact of intermediate instructions, the impact of page boundaries on prefetching, and the impact of cache hits. In addition, we show that the resulting Leakage Template expresses prefetching side channels that allow us to infer the control flow of a program.

To address RQ1.2, we use static and dynamic binary analysis techniques to find code sections in real-world binaries that match a particular Leakage Template.

Although our focus is on the ARM architecture and cache-based side channels in this chapter, Plumber's design is generic and can be used to detect Leakage Templates for other hardware features and on other architectures.

Contributions. Our main contributions are as follows:

- We introduce the concept of *Leakage Templates*, which allow us to identify code sequences (and required values) that expose a specific side-channel behavior in a binary application executed on a specific architecture (Section 3.3).
- We design and implement *Plumber*, a framework that facilitates the generation of Leakage Templates, and allows us to obtain a deeper understanding of hidden behavior of microarchitectures (Sections 3.4 and 3.5).
- We show Plumber's efficacy by investigating previction and prefetching behaviors of the ARM Cortex-A53 (Section 3.5). Based on this, we identify five novel side channels (Section 3.7).
- We demonstrate how a Leakage Template can be used to identify a side channel in a binary application (Section 3.6).

3.2 Preliminaries

In this section, we introduce some central notations for this chapter, describe the information flow analysis tool *Scam-V*, and present the microarchitectural behavior of *previction*.

3.2.1 Notations

We use the notations listed in Table 3.1 throughout this chapter.

In particular, we use a_i to refer to physical addresses, and ${a_i}^{m-n}$ to refer to certain bits of a physical address. In addition, we introduce notations for specific parts of an

Table 3.1: Summary of notations. Physical address bit offsets refer to the L1D cache mapping of the Broadcom BCM2837 (Cortex-A53).

Notation	Description
a_i	A physical address
${a_i}^{m-n}$	Bits m through n of a_i (both inclusive)
$\texttt{word}(a_i) = {a_i}^{2-5}$	Word offset of address a_i
$\texttt{bus}(a_i) = {a_i}^{4-5}$	Bus index of address a_i
$\texttt{set}(a_i) = {a_i}^{6-12}$	Cache set index of address a_i
$\texttt{tag}(a_i) = {a_i}^{13-31}$	Cache tag of address a_i
$\texttt{page}(a_i) = {a_i}^{12-31}$	Page index of address a_i
$\texttt{sameSet}(a_i, a_j, \ldots)$	Cache set index equality predicate
$\texttt{sameTag}(a_i, a_j, \ldots)$	Cache tag equality predicate
$\texttt{samePage}(a_i, a_j, \ldots)$	Page index equality predicate

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
physical address
tag | set index | word offset
page index | bus index

Figure 3.1: Physical address mapping for the Broadcom BCM2837's L1D cache

address, such as cache tags and set indices (as discussed in Section 2.2). Since the concrete bit offsets are microarchitecture-dependent, we report the relevant values for the first-level data cache by example of the Broadcom BCM2837, which is based on four ARM Cortex-A53 cores. This processor is one of our main targets in this chapter. This cache has a size of 32 KiB and is 4-way set-associative with 128 sets. Figure 3.1 shows the address mapping. In addition, we introduce three predicates, `sameTag`, `sameSet`, and `samePage`, to check for equality of cache tag, set indices, or page indices of addresses, respectively.

3.2.2 Side Channel Abstract Model Validator (Scam-V)

The *Side Channel Abstract Model Validator (Scam-V)* [105] is an information flow analysis tool. It simulates the processor's behavior for a given program based on an observational model. In particular, this model describes the architecture of the Cortex-A53 processor as well as its L1 data cache.

A main use case of Scam-V is to discover unknown side channels. To this end, the tool generates random binary programs, which we denote by $\mathcal{P}$. It then constructs pairs of initial states $\mathcal{I}$ (e.g., initial values in registers) such that the execution of $\mathcal{P}$ starting from these states is observationally indistinguishable on the model. Scam-V then tests if the two states are also indistinguishable on a real processor, i.e., in the presence of undocumented microarchitectural features. If the two states are distinguishable on hardware, the generated program and the pair of initial states constitute a *counterexample*

Code Template	Input 1 (Tag/Set)	Input 2 (Tag/Set)
`ldr x3, [x0]`	x0: **0x80100000**(T_1/S_1)	**0x80100020**(T_1/S_1)
`ldr x24, [x1]`	x1: 0x80100010(T_1/S_1)	0x80100010(T_1/S_1)
`ldr x9, [x2]`	x2: 0x80100000(T_1/S_1)	0x80100000(T_1/S_1)
`ldr x2, [x3]`	x3: 0x8013e000(T_2/S_1)	0x8013e000(T_2/S_1)
`ldr x15, [x4]`	x4: 0x80104000(T_3/S_1)	0x80104000(T_3/S_1)

Figure 3.2: Counterexample for previction on ARM Cortex-A53

to the soundness of the observational model, indicating a potential side channel.

3.2.3 Previction

Previction is an undocumented behavior of the ARM Cortex-A53 processor that invalidates cache-related observational models. We will use this behavior as an example throughout this chapter. In short, previction causes a cache line to be evicted before the corresponding cache set is full. Nemati et al. discover a handful of instances of this behavior using Scam-V [105]. The real root cause of previction is unknown, but the authors conjecture that the processor detects a short sequence of loads to the same cache set and anticipates more loads to the same set with no reuse of previously loaded values. It evicts the valid cache line in order to make space for more colliding lines.

Previction Counterexample. Figure 3.2 depicts a counterexample (as found by Scam-V) for previction. It consists of a code template and two initial states. The code template comprises five load instructions. Each initial state consists of five input values, or, more precisely, five register values that represent memory addresses. In both sets of inputs, the first three addresses map to the same cache line in the cache (cache set and cache tag are identical), while the remaining two addresses map to different cache lines within the same cache set. The only difference between both sets of inputs is the inner-cache-line offset of address `x0`, which affects the address used for the first load. Since `x0` maps to the same cache line in both cases, both sets of inputs are observationally equivalent with respect to the model and should lead to identical microarchitectural behavior. In particular, all addresses map to the same cache set. In a 4-way associative cache that is initially empty, one expects no eviction to occur, as only three different cache lines need to be stored. Executing the program with the given sets of inputs on real hardware, however, results in two different cache states. In one case, `x0` is present in the cache after executing the code template, while in the other case it is not, indicating that previction occurred.

3.3 Leakage Templates

We now introduce *Leakage Templates* and motivate their utility.

3.3.1 Goal and Motivation for Leakage Templates

Goal of the Analyst. The high-level setting is depicted in Figure 3.3. We assume that an analyst already knows a concrete code example that behaves distinctively on

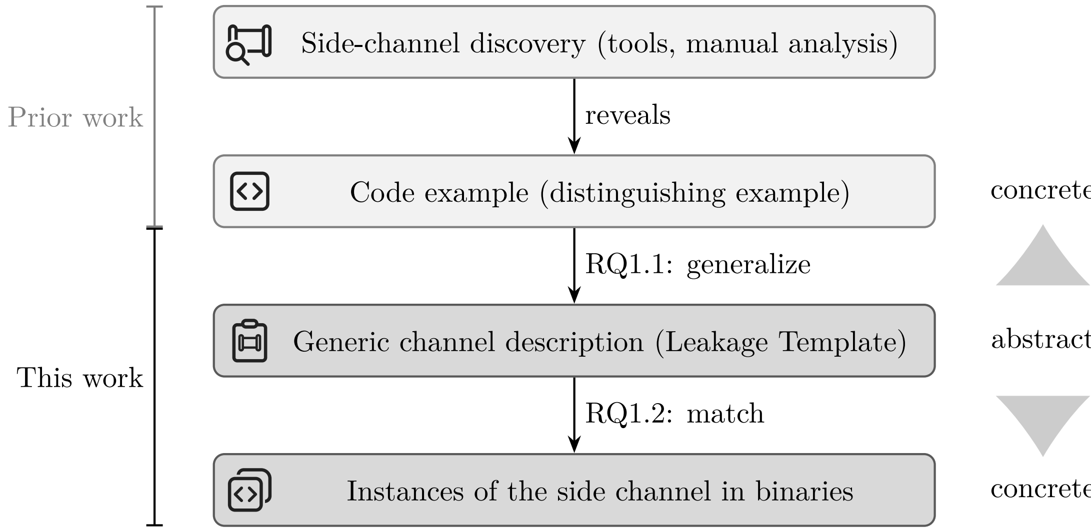

Figure 3.3: Proposed approach. We first abstract from concrete code examples with side-channel leakage to a generic Leakage Template, which we then use to identify further instances of the channel in other binaries.

the microarchitectural level for different inputs on a specific processor, thus exposing undocumented behavior. We call such a code example a *distinguishing example*. The acquisition of distinguishing examples is not the main focus of this work; nevertheless, we outline the two most common ways: First, distinguishing examples can be extracted from concrete code traces that expose the intended side channel, e.g., from counterexamples identified by tools like Scam-V. Second, they can be derived from abstract, natural-language descriptions of the architecture's behavior, e.g., in reference manuals.

Based on the concrete distinguishing example, the analyst aims to generalize to a more abstract description of the underlying side channel (RQ1.1). This generalization should depict the conditions that need to be fulfilled to cause the distinguishable behavior. Based on this description, the analyst finally aims to identify code segments in other binaries that, under certain conditions, may trigger the undocumented microarchitectural behavior (RQ1.2).

Motivation for Leakage Templates. The analyst starts with concrete code sequences (in form of distinguishing examples). It is unlikely that the exact same code sequences will appear identically in another target application. Therefore, we need to abstract from the concrete code sequences and find: (i) a generalized code sequence, (ii) a set of relevant attributes, and (iii) relations between those attributes that expose the specific side channel. In composition, we call such information a *Leakage Template*. A Leakage Template abstractly defines the conditions for code segments under which the leakage is observed. Given a Leakage Template, an analyst can identify code segments in a target application that expose a side channel. We demonstrate in Section 3.6 how binaries can be scanned for code sections that match the code pattern of a Leakage Template. In addition, we show how dynamic binary analysis techniques can be used to analyze a matching code section for the presence of side-channel leakage based on the

relations of a Leakage Template.

3.3.2 Motivating Example I: Caching

For demonstration purposes, we now show how a basic cache-timing side channel can be expressed as a Leakage Template. Assume that simple caching behavior of a CPU was insufficiently documented, and the analyst tries to understand in which parts of a target application and under which conditions caching occurs.

Distinguishing Example. An initial distinguishing example could consist of a code sequence comprising two load instructions that load from variable addresses, as well as two initial states that show different microarchitectural behavior. Each initial state defines two concrete addresses that the code sequence loads from. With the first initial state, both load instructions load from the same address, which causes the second load to execute faster due to caching. With the second initial state, two addresses on different cache lines are accessed, and the second load is not accelerated by the cache.

Outline of a Leakage Template. A Leakage Template that describes this side channel would specify the side channel in a more abstract way:

- An abstract code template: At least two loads from symbolic addresses with potentially other instructions in between
- The two possible behaviors: Fast and slow execution time of the second load
- The relations over the loaded addresses which lead to the respective behaviors: Accesses are fast if both addresses have identical tags and set indices, and are slow otherwise

We will present a more formal notation of this Leakage Template in Section 3.3.4.

Benefit of the Leakage Template. The initial distinguishing example just provides one concrete code instance that leads to the leakage behavior, while the Leakage Template ideally covers every possible sequence of instructions in which two loads constitute a similar side channel. Consequently, one would expect that the counterexample alone will not enable the analyst to identify instances of the side-channel leakage in an arbitrary target program, while the Leakage Template is expected to have a much higher chance of discovering related code fragments.

3.3.3 Motivating Example II: Prevision

For previction, the leakage behavior requires a more complex code sequence, as outlined in Section 3.2.3.

Distinguishing Example. The previction counterexample shown in Figure 3.2 could serve as an initial distinguishing example. It contains a concrete code sequence of five loads along with two concrete sets of addresses that lead to different behaviors.

Outline of a Leakage Template. A Leakage Template that describes this side channel would specify:

- An abstract code template: At least five loads from symbolic addresses with potentially a number of other instructions in between

Code $\mathcal{P}(A)$	Behavior and Relations	
`ldr x0, [x1]`	$\mathcal{B}$	$\mathcal{R}(A, b)$
`; 0-n NIIs`	(•) fast	$\texttt{sameTag}(x_1, x_2) \wedge \texttt{sameSet}(x_1, x_2)$
`ldr x0, [x2]`	(∘) slow	$\neg\texttt{sameTag}(x_1, x_2) \vee \neg\texttt{sameSet}(x_1, x_2)$

Figure 3.4: Leakage Template for caching. *fast* and *slow* are distinguishing behaviors, referring to the memory latency of the second load. *NIIs* are Non-Interfering Instructions.

- The two possible behaviors: Previction or no previction
- The relations over the loaded addresses which lead to the respective behaviors, which are not immediately obvious for previction. We will derive these relations using our novel methodology in a case study in Section 3.5.3.

3.3.4 Definition of Leakage Templates

Based on our informal motivating examples of Leakage Templates above, we now define the term more formally. A Leakage Template is a triple $(\mathcal{P}(A), \mathcal{B}, \mathcal{R}(A, b))$ with $\mathcal{P}(A)$ being a *code template* with a set of attributes $A \subseteq \mathbb{A}$, $\mathcal{B}$ a set of observed distinct behaviors, and $\mathcal{R} : \mathbb{A} \times \mathcal{B} \rightarrow 2^{\mathbb{P}_A}$ maps a behavior $b \in \mathcal{B}$ to a (set of) predicate(s) on attributes, where $\mathbb{P}_A$ is the set of predicates on A.

Attributes are variables in the code template that can be filled with arbitrary values. In practice, attributes are often placeholders for concrete values or addresses that are stored in the registers that the code template operates on.

We note that our definition of behavior is generic. For instance, it may refer to temporal (e.g., execution time) or spatial (e.g., cache content) behavior. For spatial behavior, we typically refer to the difference between the initial and final state. For example, this could be the cache content before and after the code template is executed.

Given this description, we provide the resulting Leakage Template for our caching example from Section 3.3.2 in Figure 3.4. We derive a Leakage Template for the previction example in Section 3.5.3.

3.4 Deriving Leakage Templates: The Plumber Framework

We now address RQ1.1 and present our design of Plumber, a framework to semi-automatically derive Leakage Templates (focusing on caching behavior). On a high level, Plumber generates testcases that are then executed on real hardware to systematically analyze the processor's behavior. Our goal is to derive relations that describe the conditions under which a specific behavior occurs. We address the following challenges in our design:

- **C1**: We need to construct efficient specifications that steer testcase generation towards generating inputs that are likely to trigger a specific microarchitectural behavior.

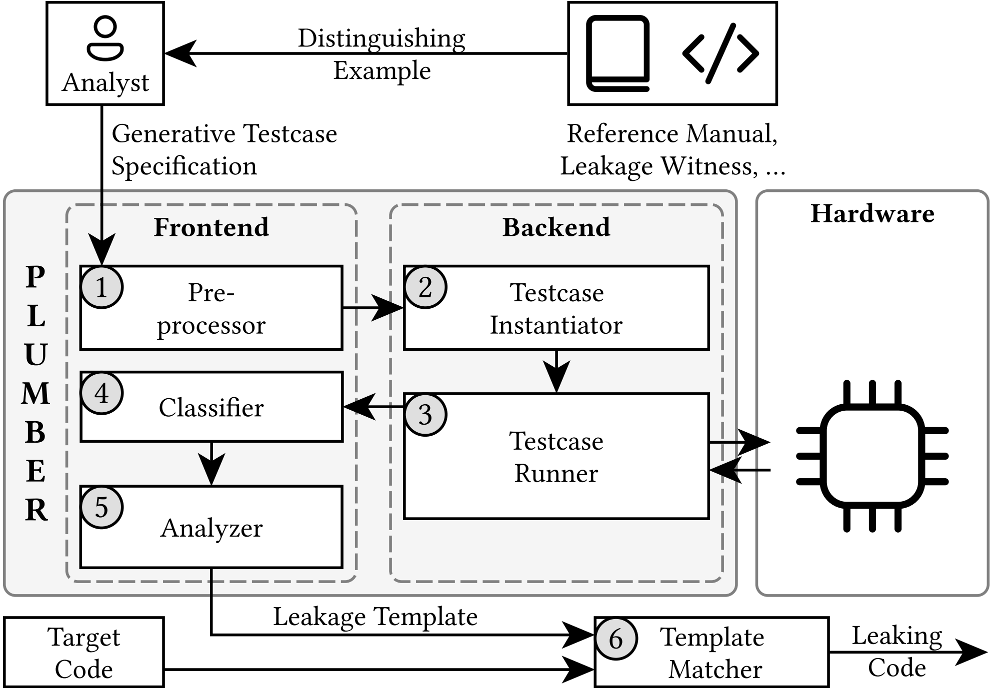

Figure 3.5: Plumber's components

- **C2**: The size of the input space (i.e., the number of possible code sequences and input values) is prohibitively large. Thus, we need an approach to explore the input space efficiently while still ensuring accuracy of the derived relations.

- **C3**: We need to get accurate measurements with minimal noise, as close as possible to the "ground truth" of the respective channel.

- **C4**: The relations between attributes in code sequences and triggered behaviors can be very complex and counter-intuitive, and manual derivation of such relations is highly error-prone (if not impossible).

3.4.1 High-Level Framework Design

Figure 3.5 gives an overview of the components of our framework. The input to Plumber is a *Generative Testcase Specification (GTS)*, which abstractly describes programs to analyze for their leakage behavior in a domain-specific language. The GTS is written by an analyst based on distinguishing examples. We provide the detailed description of the GTS, the language, and how it addresses **C1** in Sections 3.4.2 and 3.4.3. The final result of the analysis is a Leakage Template, which includes the behavior of the monitored microarchitectural components and the relations between attributes and behaviors.

The framework itself consists of two parts, the *Backend* and the *Frontend*. The Frontend first preprocesses a given GTS by parsing it and expanding some of the operators that the language provides. The Backend then systematically translates the preprocessed GTS into one or more concrete testcases (addressing **C2**), where each

testcase consists of program code and concrete inputs. These testcases are then executed in a controlled environment (addressing **C3**) while recording microarchitectural behavior. Next, the Frontend takes over again and classifies the testcases based on the observed behavior. Finally, the Frontend analyzes the classified behaviors (addressing **C4**) and returns relations on attributes that trigger specific microarchitectural behavior (i.e., $\mathcal{B}$ and $\mathcal{R}(A, b)$ of the Leakage Template).

Once a Leakage Template has been produced, one possible application of it is to identify other instances of the described behavior in other binaries (addressing RQ1.2). We propose a Template Matcher that uses static and dynamic binary analysis techniques to find code sections matching the code pattern of the Leakage Template as well as its relations. We discuss this approach in detail in Section 3.6.

3.4.2 Definition of the Generative Testcase Specification (GTS)

The GTS is the input to Plumber. Intuitively, it defines sequence(s) of instructions to be executed over (mutated) operands, and microarchitectural component(s) to be monitored (e.g., content of the cache). The GTS can also specify the initial state of a component before the execution of its main instruction sequence.

To address **C1**, we introduce a domain-specific generative language. The language provides three main features:

1. It allows us to abstractly specify possible mutations of the program (i.e., enabling generic code templates).
2. It allows us to specify which input values should be mutated (i.e., defining domains for attributes).
3. It allows us to specify relations to apply to non-mutated values in the program.

3.4.3 Generative Testcase Specification (GTS) Language

We now introduce our domain-specific language for Generative Testcase Specifications.

Directives. A GTS is formed of a sequence of directives that specify different operations, e.g., arithmetic operations or `nop` instructions. The proposed language defines an extensible set of directives, including `M`, `A`, and `N`. The directive `M` denotes memory loads, `A` denotes arithmetic or logical operations, and `N` denotes `nop` instructions.

The addresses of memory loads (`M`) are mapped to certain cache sets and have specific cache tags. These attributes can be specified through abstract references to a cache tag and a cache set index. We use $\mathtt{M}_{\mathtt{t}_1,\mathtt{s}_1}$ to refer to a load from an address with tag $\mathtt{t}_1$ and cache set $\mathtt{s}_1$. This allows us to express the relation between tags or sets of different loads. When omitted, these attributes assume default values which are identical for the same request. The language allows defining arithmetic relations between the tag and set attributes of different memory directives.

Example. The GTS $\mathtt{M\ M\ M}_{\mathtt{t}_1,\mathtt{s}_1}\ \mathtt{M}_{\mathtt{t}_1,\mathtt{s}_1}\ \mathtt{M}_{\mathtt{t}_1+1,\mathtt{s}_1+5}\ \mathtt{A\ A\ N}$ represents two loads of memory addresses with the same set and the same tag (default), followed by two other loads with a different tag $\mathtt{t}_1$ and a different set $\mathtt{s}_1$ and a fifth load with tag ($\mathtt{t}_2 = \mathtt{t}_1 + 1$) and set ($\mathtt{s}_2 = \mathtt{s}_1 + 5$). These loads are followed by two arithmetic or logical operations

and a `nop` instruction. The values of the operands of other instructions, e.g., arithmetic operations, can be defined in the same manner, e.g., $\mathtt{A}_{\mathtt{v}_1,\mathtt{v}_2}$ represents an arithmetic operation with two operands $\mathtt{v}_1$and $\mathtt{v}_2$.

Operators. Our language further provides a number of operators that make it easier to construct complex GTSs. Most operators expand a singular testcase to a set of testcases that differ by specific attributes and are then executed in sequence. Some operators enable special features. For example, a precondition operator allows us to define the initial state of a hardware component. We now describe the basic operators that the language provides. We note that the language is extensible and new operators can be defined as needed to investigate various processor components.

$[\cdot]\mathtt{dir}_{\mathtt{attr}}, \mathsf{n}, \mathsf{i}$ **(Power):** This operator allows repeating directive(s) n times, while incrementing the attribute `attr` of the directive `dir` by a value of i. For example, the GTS $[\mathtt{M}_{\mathtt{t}_1,\mathtt{s}_1}]\mathtt{M}_{\mathtt{s}}, 2, 1$ can be used to refer to the GTS: $\mathtt{M}_{\mathtt{t}_1,\mathtt{s}_1}\ \mathtt{M}_{\mathtt{t}_1,\mathtt{s}_1+1}$. The power operator can also be used with a single input n. In this case the directive(s) are repeated n times. For instance, the example GTS in the previous paragraph can also be expressed as: $[\mathtt{M}]2\ [\mathtt{M}_{\mathtt{t}_1,\mathtt{s}_1}]2\ \mathtt{M}_{\mathtt{t}_1+1,\mathtt{s}_1+5}\ [\mathtt{A}]2\ \mathtt{N}$.

$\#\mathsf{n}$ **(Wildcard):** This operator expands to n arbitrary directives that do not perform memory operations. For example, one possible expansion of $\mathtt{M}\ \#3\ \mathtt{M}$ is $\mathtt{M\ N\ A\ N\ M}$.

$(\cdot)!$ **(Shuffle):** This operator generates all possible permutations of a GTS while omitting those with similar directives. For example, $([\mathtt{M}]2\ \mathtt{M}_{\mathtt{t}_1,\mathtt{s}_1})!$ expands to the set $\{\mathtt{M\ M\ M}_{\mathtt{t}_1,\mathtt{s}_1}; \mathtt{M\ M}_{\mathtt{t}_1,\mathtt{s}_1}\ \mathtt{M}; \mathtt{M}_{\mathtt{t}_1,\mathtt{s}_1}\ \mathtt{M\ M}\}$.

$(\cdot)\subset$ **(Subset):** This operator generates all possible subsets of a GTS while omitting those with similar directives. For example, $(\mathtt{M\ M\ M}_{\mathtt{t}_1,\mathtt{s}_1})\subset$ expands to the set $\{\mathtt{M}; \mathtt{M\ M}; \mathtt{M\ M}_{\mathtt{t}_1,\mathtt{s}_1}; \mathtt{M}_{\mathtt{t}_1,\mathtt{s}_1}\}$.

$(\cdot)\mathsf{n}$ **(Slide):** For a given GTS, this operator shifts all loaded addresses one set at a time up to n times. For example, $(\mathtt{M}_{\mathtt{t}_1,\mathtt{s}_1}\ \mathtt{M}_{\mathtt{t}_2,\mathtt{s}_2})3$ expands to the set:

$$\{\mathtt{M}_{\mathtt{t}_1,\mathtt{s}_1}\ \mathtt{M}_{\mathtt{t}_1,\mathtt{s}_1}; \mathtt{M}_{\mathtt{t}_1,\mathtt{s}_1+1}\ \mathtt{M}_{\mathtt{t}_1,\mathtt{s}_1+1}; \mathtt{M}_{\mathtt{t}_1,\mathtt{s}_1+2}\ \mathtt{M}_{\mathtt{t}_1,\mathtt{s}_1+2}\}$$

$(\cdot : \cdot)+$ **(Merge):** This operator merges two requests by sliding the directives of the first over the second. For example, the GTS $(\mathtt{M}_{\mathtt{t}_1,\mathtt{s}_1}\ \mathtt{M}_{\mathtt{t}_2,\mathtt{s}_2} : \mathtt{M}_{\mathtt{t}_3,\mathtt{s}_3}\ \mathtt{M}_{\mathtt{t}_4,\mathtt{s}_4})+$ expands to:

$$\{\mathtt{M}_{\mathtt{t}_1,\mathtt{s}_1}\ \mathtt{M}_{\mathtt{t}_2,\mathtt{s}_2}\ \mathtt{M}_{\mathtt{t}_3,\mathtt{s}_3}\ \mathtt{M}_{\mathtt{t}_4,\mathtt{s}_4}; \mathtt{M}_{\mathtt{t}_1,\mathtt{s}_1}\ \mathtt{M}_{\mathtt{t}_3,\mathtt{s}_3}\ \mathtt{M}_{\mathtt{t}_2,\mathtt{s}_2}\ \mathtt{M}_{\mathtt{t}_4,\mathtt{s}_4}; \mathtt{M}_{\mathtt{t}_3,\mathtt{s}_3}\ \mathtt{M}_{\mathtt{t}_1,\mathtt{s}_1}\ \mathtt{M}_{\mathtt{t}_4,\mathtt{s}_4}\ \mathtt{M}_{\mathtt{t}_2,\mathtt{s}_2}; \mathtt{M}_{\mathtt{t}_3,\mathtt{s}_3}\ \mathtt{M}_{\mathtt{t}_4,\mathtt{s}_4}\ \mathtt{M}_{\mathtt{t}_1,\mathtt{s}_1}\ \mathtt{M}_{\mathtt{t}_2,\mathtt{s}_2}\}$$

$\langle\cdot\rangle@$ **(Load Offset Mutation):** For every load instruction, this operator signals generation of a testcase for all possible addresses with the indicated tag and set, i.e., it brute forces word offsets. For example, $\langle\mathtt{M\ M}\rangle@$ generates a set formed of all two loads having the same tag and set with all possible combinations of word offsets.

$\langle\cdot\rangle\$$ **(Cache Line Mutation):** For every load instruction, this operator signals the generation of a testcase for every possible memory address having the indicated tag and word offset, i.e., it brute forces all possible sets. For example, $\langle \mathtt{M\ M} \rangle\$$ generates a set formed of all two loads that have the same tag for all possible combinations of sets, i.e., $\mathtt{M}_{t_1,s_1}\ \mathtt{M}_{t_1,s_2}$ for every set s_1 and s_2.

$|\cdot|n$ **(Repetition):** This operator repeats the GTS n times, e.g., the GTS $|\mathtt{M}\ \mathtt{M}_{t_1,s_1}|3$ corresponds to: $\{\mathtt{M}\ \mathtt{M}_{t_1,s_1}; \mathtt{M}\ \mathtt{M}_{t_1,s_1}; \mathtt{M}\ \mathtt{M}_{t_1,s_1}\}$.

$P(\cdot)$ **(Precondition):** This operator allows setting up the state of a hardware component before the testcase is executed. For instance, the GTS $P(\mathtt{M}_{t_1,s_1}\ \mathtt{M}_{t_2,s_1})\ \langle \mathtt{M\ M} \rangle\$$ generates cache line mutation testcases where two lines in s_1 are already occupied.

3.4.4 Framework Components

In the following, we provide more details on Plumber's components depicted in Figure 3.5. On a high level, the framework consists of a Frontend and a Backend. The Frontend handles user interaction and comprises three components, the Preprocessor, Classifier, and Analyzer. The Backend is responsible for generating concrete testcases (a set of programs and their inputs) from a GTS, setting up microarchitectural components, executing the programs over their respective inputs on real hardware, and returning the microarchitectural behavior of this execution. The Backend is formed of two components: The Testcase Instantiator and the Testcase Runner.

We now proceed to discuss the individual components of Plumber in the order of execution when processing a GTS.

3.4.4.1 Preprocessor

The Preprocessor interprets a given GTS. It expands repetition, power, and wildcard operators. Wildcards are expanded by randomly picking directives from the set of defined operations. The Preprocessor is also responsible for expanding the shuffle, subset, slide, and merge operators, i.e., it generates permutations of a given GTS by applying the respective operations. Finally, it forwards the expanded GTS to the Testcase Instantiator in the Backend.

3.4.4.2 Testcase Instantiator

The Instantiator receives an expanded GTS from the Preprocessor and generates concrete testcases, i.e., assembly programs. It performs instruction generation and instruction operand mutation. Specifically, the Instantiator uses an instruction store to pick instructions for each operation when generating programs. It further uses an address store to generate inputs: For the first occurrence of every tag and/or set attribute value, the generator picks random values from the store and queries their corresponding address. For every later occurrence of a tag or set, the generator uses the previously selected tag and/or set from the first occurrence. In the case of an arithmetic relation between these attributes, the generator searches for addresses that satisfy this relation. Alternatively, the Instantiator can generate new addresses and add them to the address

store. Finally, the Instantiator generates and/or mutates inputs to instructions (i.e., their operands) as requested. For example, when the GTS contains an offset mutation operator, the generator generates inputs with all possible address combinations by brute-forcing their word offset. When the cache line mutation operator is used, the Instantiator generates inputs with all possible address combinations by brute forcing the set index. The generated testcases are then forwarded to the Testcase Runner.

3.4.4.3 Testcase Runner

The Runner receives a testcase from the Instantiator to be executed on the hardware. It connects to the hardware and refreshes the microarchitectural state, e.g., clears the cache. The Runner then sets up microarchitectural components by executing the precondition part of the testcase. Finally, the program is executed in ARM TrustZone, which provides the highest level of privilege and allows the execution of all possible instructions as well as the inspection of some microarchitectural state. Most importantly, TrustZone provides direct access to the cache state through privileged debug instructions (RAMINDEX). However, Plumber may also leverage other techniques to infer the microarchitectural state (e.g., Reload+Time typically used in cache side-channel attacks [117, 161] to infer the content of the cache) if such special debug instructions are unavailable.

We run all experiments in a bare-metal environment. There are no background processes or interrupts which could induce noise in terms of cache content or timing. We still experience a low amount of noise due to the memory subsystem being shared with the GPU and because our experiments are not synchronized with the memory controller. We found, however, that this noise could be safely ignored.

The result of the testcase execution is the final microarchitectural state (e.g., cache state), which is communicated to the Classifier.

3.4.4.4 Classifier

The Classifier classifies the executed testcases based on the behavior observed at runtime. For previction, for example, the output is classified based on the occurrence of previction (or, more precisely, based on whether potentially previcted addresses are present in the final cache state or not). For every behavior, the Classifier generates a *bit table* containing the binary representation of the mutated attribute values (e.g., instruction operands) that cause this behavior. For previction, these are the five addresses that are accessed (see Figure 3.2). Table 3.2 shows an example of such bit tables for previction. Each testcase is represented by one row in the table, each column represents an attribute. In this example, the attributes determine the values in the load address registers, which are instantiated with different concrete addresses in every testcase. The bit tables serve as input to the Analyzer.

3.4.4.5 Analyzer

The Analyzer evaluates bit tables to extract relations between inputs and their effect on the monitored microarchitectural behavior. It is based on a set of primitive operations

Table 3.2: Bit tables for prevction testcases. Shaded columns denote the attributes responsible for prevction. Mutated bits are in bold and the relevant bits for causing prevction are underlined.

Testcase	Attributes (here: address registers)				
	`x0`	`x1`	`x2`	`x3`	`x4`
	Bit table for behavior "Prevction" (●)				
1	$1\cdots000000000\underline{\mathbf{00}}\mathbf{00}00$	$1\cdots000000000\underline{\mathbf{00}}\mathbf{00}00$	$\cdots$	$\cdots$	$1\cdots100000000\mathbf{0000}00$
2	$1\cdots000000000\underline{\mathbf{00}}\mathbf{01}00$	$1\cdots000000000\underline{\mathbf{00}}\mathbf{00}00$	$\cdots$	$\cdots$	$1\cdots100000000\mathbf{0000}00$
3	$1\cdots000000000\underline{\mathbf{00}}\mathbf{10}00$	$1\cdots000000000\underline{\mathbf{00}}\mathbf{00}00$	$\cdots$	$\cdots$	$1\cdots100000000\mathbf{0000}00$
4	$1\cdots000000000\underline{\mathbf{00}}\mathbf{11}00$	$1\cdots000000000\underline{\mathbf{00}}\mathbf{00}00$	$\cdots$	$\cdots$	$1\cdots100000000\mathbf{0000}00$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
	Bit table for behavior "No Prevction" (○)				
5	$1\cdots000000000\underline{\mathbf{01}}\mathbf{00}00$	$1\cdots000000000\underline{\mathbf{00}}\mathbf{00}00$	$\cdots$	$\cdots$	$1\cdots100000000\mathbf{0000}00$
6	$1\cdots000000000\underline{\mathbf{01}}\mathbf{01}00$	$1\cdots000000000\underline{\mathbf{00}}\mathbf{00}00$	$\cdots$	$\cdots$	$1\cdots100000000\mathbf{0000}00$
7	$1\cdots000000000\underline{\mathbf{01}}\mathbf{10}00$	$1\cdots000000000\underline{\mathbf{00}}\mathbf{00}00$	$\cdots$	$\cdots$	$1\cdots100000000\mathbf{0000}00$
8	$1\cdots000000000\underline{\mathbf{01}}\mathbf{11}00$	$1\cdots000000000\underline{\mathbf{00}}\mathbf{00}00$	$\cdots$	$\cdots$	$1\cdots100000000\mathbf{0000}00$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

over bit tables to generate and validate such relations. These primitives are comparable to SQL statements. For example, they include:

- The function *count*() counts the number of rows in a table;
- The variadic function $select(cond(\mathbf{x}_i[m_i \ldots n_i], \ldots), \mathbf{x}_i[m_i \ldots n_i], \ldots)$ returns all rows whose fields are in the relation *cond*, e.g., $select(\mathbf{x}_i[j] = 1, \mathbf{x}_i)$ returns rows where the jth bit of the register $\mathbf{x}_i$ is 1;
- The variadic function $relation(\mathbf{x}_i[m_i \ldots n_i], \mathbf{x}_j[m_j \ldots n_j], \ldots)$ takes two or more inputs and returns a linear relation over specific bits of the inputs (if present).

Based on these primitives, we define an example analyzer function for prevction. Other analyzer functions typically follow the same strategy. As shown in Figure 3.6, this function has three phases: candidate selection, relation extraction, and relation validation.

Step 1: Candidate Selection. The goal of the first step is to pinpoint which parts of the mutated inputs correlate with the observed behavior. In the case of prevction, we want to find the specific bits in every address that cause prevction to appear (or not appear).

To do so, we first define an auxiliary function occ (short for *number of occurrences*). Let $\circ$ denote function composition. We define occ as the composition of *count* and *select*: occ $= count \circ select$. occ retrieves and filters the rows in a bit table based on a condition (just like *select*), and then *count*s the number of remaining rows.

We use this function to determine *candidates*. To this end, we first investigate whether specific bits in an address must be set (or cleared) in order to cause a specific

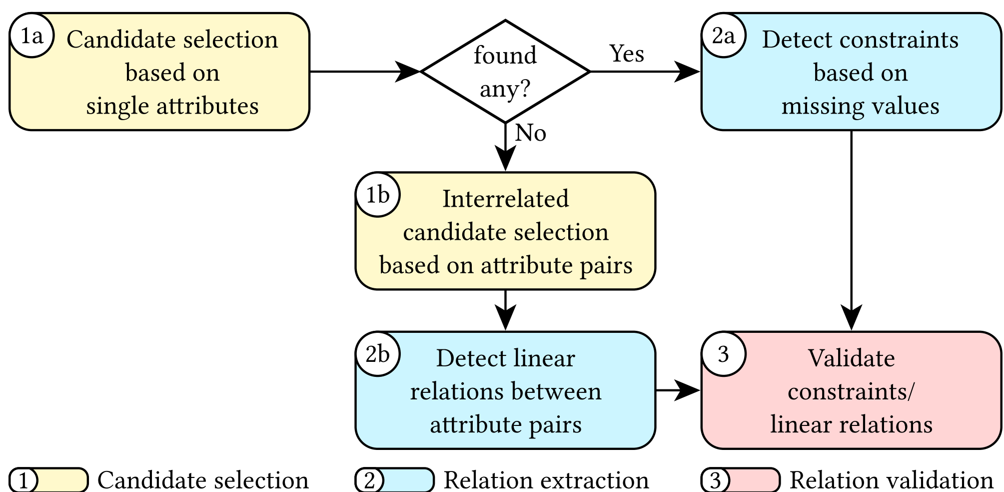

Figure 3.6: Flow chart of the analyzer function for previction

behavior (step 1a). If this is the case, we would expect that all testcases with those bits set only appear in one of the bit tables and are absent from the other. Otherwise, for unrelated bits, all possible values should appear equally often within each bit table. Formally, we perform the following test: For every attribute (here: address register) $\mathbf{x}_i$, we use $\mathsf{occ}(\mathbf{x}_i[m \dots n] = x, \mathbf{x}_i)$ to find the number of occurrences of each possible bit sequence $x \in \{0,1\}^{(m-n)}$ for the *non-constant bit sequence* indexed by m and n, i.e., the mutated bits, in the bit tables. The Analyzer compares the number of actual occurrences to the expected number of occurrences of this value, i.e., it checks whether $\mathsf{occ}(\mathbf{x}_i[m \dots n] = x, \mathbf{x}_i) = \frac{\mathsf{Total}}{\mathsf{n_{pos}}}$; where Total denotes the size of the bit table and $\mathsf{n_{pos}} = 2^{(m-n)}$ is the number of different possible values for the bit sequence indexed by m and n. Every attribute where the number of occurrences of (some or all) values deviates from the expectation is marked as a *candidate attribute*.

If all tests result in the same number of occurrences equal to $\frac{\mathsf{Total}}{\mathsf{n_{pos}}}$, i.e., no candidate attributes were found, the Analyzer proceeds to step 1b. In this step, we extend the previous check to pairs of attributes. In this way, we check whether a specific combination of bit sequences across two attributes (address registers) is required to cause previction. Formally, we test as follows: For every pair of attributes $\mathbf{x}_i$ and $\mathbf{x}_j$ the Analyzer checks whether $\mathsf{occ}(\mathbf{x}_i[m \dots n] = x \land \mathbf{x}_j[m \dots n] = y, \mathbf{x}_i, \mathbf{x}_j) = \frac{\mathsf{Total}}{\mathsf{n'_{pos}}}$, for all possible values $x, y \in \{0,1\}^{(m-n)}$, where $\mathsf{n'_{pos}} = 2^{2\times(m-n)}$ is the number of possible values for the two bit sequences. Every pair of attributes for which the number of occurrences of (some or all) values deviates from the expectation is marked as a pair of *candidate interrelated attributes*.

Step 2: Relation Extraction. Next, the Analyzer detects the constraints and relations, i.e., the rules that certain bits must follow to cause the studied behavior.

In step 2a, we use occ to determine constraints on address bits. More precisely, the Analyzer checks which bit values are missing from a bit table. In simple terms, a resulting constraint could be phrased like this: *Whenever we see the studied behavior, bits $[m \dots n]$ of attribute* ***x0*** *are set.*

In step 2b, we determine whether there is a linear relation between the addresses in candidate interrelated attribute pairs that must be fulfilled to show the studied behavior. This leads to constraints like this: *Whenever we see the studied behavior, bits* $\texttt{x0}[m \dots n]$ *are equal to bits* $\texttt{x1}[m \dots n] + 1$. Formally, we use the function $relation(\mathtt{x}_i[m \dots n], \mathtt{x}_j[m \dots n])$ to find such relations between the non-constant bits of interrelated candidate attribute values. It finds a and b in the equation $y = ax + b \mod (m-n)$, where x and y represent the interrelated bits in the two values, respectively.

Step 3: Relation Validation. In the final step, the Analyzer validates the generated constraints and relations. A constraint is validated by using the occ function to check whether all value combinations of unrelated bits occur in the bit table. A relation is validated by using occ to check whether (i) every row in the bit table satisfies the extracted relation and (ii) every value combination of unrelated bits occurs in the bit table.

3.5 Application of Microarchitectural Leakage Templates

We now proceed to show the utility of Leakage Templates and the effectiveness of Plumber through three case studies: finding a cache eviction strategy, analyzing previction, and analyzing prefetching. For each case study, we design experiments in the form of GTSs, run them, and use the results to iteratively build a Leakage Template. We note that the final Templates presented in Figure 3.7 omit repetitive cases for brevity as well as some corner cases that produce inconclusive outcomes, i.e., cases that generate random behavior and may not be reliably exploited.

3.5.1 Framework Implementation and Experimental Setup

Plumber currently targets the ARM architecture. It is implemented in C and Python as well as ARM assembly. We use ARM assembly to implement testcases and to set up and (accurately) read architectural components. However, our design is general and applicable to other architectures as well.

We perform the experiments described in this section on Raspberry Pi 3 and 4 computers, which are widely used ARMv8 platforms and use Broadcom BCM2837 (ARM Cortex-A53) and Broadcom BCM2711 (Cortex-A72) processors, respectively. On these processors, data is transferred between memory and cache in cache lines of 64 bytes and the L1 data cache is 4-way set-associative. The Cortex-A53 cores show the previction behavior. As far as prefetching is concerned, the *exact* behavior of the prefetcher on these cores is unknown.

To perform our experiments, we use a cluster of five Raspberry Pi 3 boards and one Pi 4 board. Table 3.3 shows the approximate total execution time to perform each experiment. We note that each experiment is a one-time effort required once per side channel and architecture.

3.5.2 Case Study: Cache Eviction Strategy

To demonstrate the utility of Plumber, we first use it to find a *cache eviction strategy* for the ARM Cortex-A72. Consider the following scenario: An attacker wants to evict a

Table 3.3: Approximate total experiment execution times. d, m, and s indicate *day(s)*, *minute(s)*, and *second(s)*, respectively.

		Execution time				
Case study	Eviction	$62m$				
	Previction	**E1**	**E2**	**E3**	**E4**	**E5**
		$0.7d$	$6s$	$7m$	$1.5d$	$> 1s$
	Prefetching	**E6**	**E7**	**E8**	**E9**	**E10**
		$4d$	$6m$	$> 1s$	$> 1s$	$0.2d$

target cache line from the cache. To this end, they first build an eviction set, i.e., a list of addresses mapping to the same cache set as the target line. Next, the attacker has to access the addresses in the eviction set. This is where the eviction strategy comes in: It determines in which order the addresses in the eviction set should be accessed to evict the target cache line with high probability.

To find viable eviction strategies, we implement the loop-based Algorithm 3.1, which was proposed by Gruss, Maurice, and Mangard [43] and has been adopted to ARM CPUs by Lipp et al. [86]. In this algorithm, the list a contains the addresses of the eviction set. The parameter S is the size of the eviction set. D determines how many consecutive addresses are accessed per iteration, while C determines how often those accesses are repeated. The parameter L allows overlapping accesses across rounds (e.g., accessing $a[0]$, $a[1]$ in the first round and $a[1]$ and $a[2]$ in the second round). The goal of the attacker is to find "good" values for the parameters D, C, L, and S, i.e., values that lead to eviction of the target cache line with high probability.

Algorithm 3.1: Algorithm to find an eviction strategy (from [43])

1 **for** $(s = 0; s \leq S - D; s \mathrel{+}= L)$ **do**
2   **for** $(c = 0; c \leq C; c \mathrel{+}= 1)$ **do**
3   **for** $(d = 0; d \leq D; d \mathrel{+}= 1)$ **do**
4             $*a[s + d]$;

To carry out this experiment with Plumber, we first transform it into a GTS:

$$|\mathsf{P}\ (\mathtt{M}_{\mathtt{t}_2,\mathtt{s}_1}\ \ [[[\mathtt{M}_{\mathtt{t}_1,\mathtt{s}_1}]\mathtt{M}_{\mathtt{t}}, \mathsf{D}, 1]\mathsf{C}]\mathtt{M}_{\mathtt{t}}, \mathsf{S}, \mathsf{L})\ \mathtt{M}_{\mathtt{t}_2,\mathtt{s}_1}|1000$$

We use a precondition to place a target cache line in cache and run the loops from the eviction strategy algorithm. Afterwards, we probe the cache state of the target cache line. Each experiment is repeated 1000 times.

We classify the results based on the latency of the probing load, indicating whether eviction occurred or not. Based on the results, we formulate the Leakage Template in Figure 3.7 (a), which shows viable corridors for the algorithm's parameters. We opt for an error tolerance of 5%, i.e., testcases that lead to eviction in more than 950 out of 1000 cases were classified as triggering (•) behaviors.

Code $\mathcal{P}(A)$

```
let pa = x1+s+d in
 (
  ((ldr x0,[pa])^D)^C
 )^(S-D/L)
Step size:
    d += 1;  s += L
Stop condition:
    D ≤ d;  S - D ≤ s
```

Behavior and Relations

$\mathcal{B}$	$\mathcal{R}(A,b)$
•	$\mathtt{sameSet}(x_i) \wedge$ $(D = 3 \wedge 3 \leq C \leq 4 \wedge L = 1 \wedge 22 \leq S \leq 23$ $\vee \cdots$ $D = 4 \wedge 4 \leq C \leq 6 \wedge L = 1 \wedge 19 \leq S \leq 22$ $\vee \cdots$ $D = 5 \wedge 3 \leq C \leq 6 \wedge L = 1 \wedge 18 \leq S \leq 22$ $\vee \cdots$ $D = 6 \wedge C = 6 \wedge L = 1 \wedge 18 \leq S \leq 22)$
∘	Negation of the condition •

(a) Cache Eviction

Code $\mathcal{P}(A)$

```
ldr x0, [x1]
#n1 instructions
ldr x0, [x2]
#n2 instructions
ldr x0, [x3]
#n3 instructions
ldr x0, [x4]
#n4 instructions
ldr x0, [x5]
```

Behavior and Relations

$\mathcal{B}$	$\mathcal{R}(A,b)$
•	$(n_1 + n_2 + n_3 + n_4 < 12 \wedge n_4 < 7)$ $\wedge$ ((sameTag(x1, x2, x3) $\wedge$ sameSet(x1, x2, x3, x4, x5) $\wedge$ bus(x1) $\neq$ bus(x2) + 1 mod 4) $\vee$ (sameTag(x2, x3, x4) $\wedge$ sameSet(x1, x2, x3, x4, x5) $\wedge$ bus(x2) $\neq$ bus(x3) + 1 mod 4) $\vee \cdots$)
∘	Negation of the condition •

(b) Previction

Code $\mathcal{P}(A)$

```
ldr x8,[x0,x1]
#n1 instructions
ldr x8,[x0,x2]
#n2 instructions
ldr x8,[x0,x3]
```

Behavior and Relations

Let $d_1 = \mathtt{set}(\mathtt{x0}+\mathtt{x2}) - \mathtt{set}(\mathtt{x0}+\mathtt{x1})$,
$d_2 = \mathtt{set}(\mathtt{x0}+\mathtt{x3}) - \mathtt{set}(\mathtt{x0}+\mathtt{x2})$,
$a_{\min} = \mathtt{x0}+\mathtt{x1}$, $a_{\max} = \mathtt{x0}+\mathtt{x3}$,
$\mathrm{N}_3 := (n_1 < 3 \wedge n_2 < 3) \vee (n_1 = 5 \wedge n_2 = 0) \vee (n_1 = 0 \wedge n_2 = 5)$,
$\mathrm{N}4 := (n_1 = 3 \wedge n_2 = 0) \vee (n_1 = 0 \wedge n_2 = 3)$,
$\mathrm{N}7 := (n_1 = 4 \wedge n_2 = 0) \vee (n_1 = 0 \wedge n_2 = 4)$ in

$\mathcal{B}$	$\mathcal{R}(A,b)$
P_0	$d_1 \neq d_2 \vee d_1 = 0 \vee \lvert d_1 \rvert > \delta_{\max} \vee$ $\neg\mathtt{samePage}(a_{\min}, a_{\max}) \vee$ $\neg\mathtt{samePage}(a_{\max}, a_{\max} + d_1)$
P_1	$(\mathrm{N}_3 \vee \mathrm{N}_4 \vee \mathrm{N}_7) \wedge \neg\mathtt{samePage}(a_{\max}, a_{\max} + 2d_1)$
P_2	$(\mathrm{N}_3 \vee \mathrm{N}_4 \vee \mathrm{N}_7) \wedge \neg\mathtt{samePage}(a_{\max}, a_{\max} + 3d_1)$
P_3	$\mathrm{N}_3 \vee ((\mathrm{N}_4 \vee \mathrm{N}_7) \wedge \neg\mathtt{samePage}(a_{\max}, a_{\max} + 4d_1))$
P_4	$\mathrm{N}_4 \vee (\mathrm{N}_7 \wedge \neg\mathtt{samePage}(a_{\max}, a_{\max} + 5d_1))$
P_5	$\mathrm{N}_7 \wedge \neg\mathtt{samePage}(a_{\max}, a_{\max} + 6d_1)$
P_6	$\mathrm{N}_7 \wedge \neg\mathtt{samePage}(a_{\max}, a_{\max} + 7d_1)$
P_7	N_7

(c) Prefetching

Figure 3.7: Case study Leakage Templates with their most relevant relations. In (a), a^b means b times inlining repetition of instruction a. In (b) and (c), $\#n_i$ means inlining n_i simple arithmetic, logical or `nop` instructions. For (a) and (b), triggering and not triggering behavior are denoted by • and ∘, respectively. In (c), P_l denotes distinguishing behaviors and l is the number of prefetched lines. Relations must be checked in order.

We note that the eviction strategy experiment is not a typical use case for Plumber, as we did not use a counterexample as the starting point. However, it is suitable for illustrating Leakage Templates with a simple example.

3.5.3 Case Study: Prevction

We now proceed to use Plumber to analyze a processor's behavior starting from corresponding counterexamples. In particular, we used Scam-V [105] to obtain a set of programs that cause previction on the ARM Cortex-A53 (see Section 3.2.3). All the obtained counterexamples were formed of exactly five load instructions. Moreover, looking at the cache content, it was evident that all counterexamples loaded three different tags to the same cache set (and optionally a fourth tag to a different set), i.e., two to three loads targeted the same cache line. We exploit this knowledge to construct initial GTSs. We use these GTSs in five experiments (**E1**–**E5**) to iteratively refine a Leakage Template for previction.

E1: Minimal Code. First, we want to check whether the obtained counterexamples contain minimal programs that trigger previction. We create a GTS which generates testcases containing all possible subsets (without repetition) of all instructions of each counterexample. The GTS further mutates the word offsets for each of the loaded addresses. None of the generated testcases triggers previction. Thus, the counterexamples contain minimal previction programs.

Performing **E1** requires 4 sets of experiments, each consisting of $5!/(5-x)! \times 16^x$ testcases, where $1 \leq x \leq 4$. Table 3.3 shows the total execution time of this experiment.

E2: Order of Instructions. Next, we check whether the order of instructions affects previction. We create GTSs that use the shuffle operator to generate testcases containing all possible permutations of the load sequence in our distinct counterexamples, for example:

$$(\mathtt{M}_{\mathtt{t}_1,\mathtt{s}_1}\ \ \mathtt{M}_{\mathtt{t}_1,\mathtt{s}_1}\ \ \mathtt{M}_{\mathtt{t}_1,\mathtt{s}_1}\ \ \mathtt{M}_{\mathtt{t}_2,\mathtt{s}_1}\ \ \mathtt{M}_{\mathtt{t}_3,\mathtt{s}_1})!$$

The first three loads, which we denote as $\underline{1}$, $\underline{2}$, and $\underline{3}$ in the following, all target the same cache line (but at different offsets). The outcome of the experiment is shown in Table 3.4. We draw three conclusions from this test:

- *Consecutive same-tag loads required:* The relation between the location of tags affects previction. More precisely, only programs with three consecutive load instructions with the same tag cause previction.
- *Tag value is irrelevant:* The exact tag value and location of the three consecutive loads within the five loads does not matter. The same holds for the non-consecutive load instructions.
- *Word offset is relevant:* Previction behavior differs based on the byte (word) offset of the loaded addresses. In our example, multiple programs with the same order of cache sets and tags are in both PR and nPR, e.g., 2-1-3-5-4 (PR) and 2-3-1-5-4 (nPR). These permutations have instructions 1 and 3 swapped, i.e., two load instructions that only differ in their word offset.

Table 3.4: Example permutation outcome. Each number represents an instruction from the initial testcase. Underlined numbers are loads from addresses that have the same tag.

Behavior	**Observation**
Previction (PR)	$\underline{1}$-$\underline{2}$-$\underline{3}$-4-5, $\underline{1}$-$\underline{2}$-$\underline{3}$-5-4, $\underline{2}$-$\underline{1}$-$\underline{3}$-4-5, $\underline{2}$-$\underline{1}$-$\underline{3}$-5-4, 4-$\underline{1}$-$\underline{2}$-$\underline{3}$-5, 4-$\underline{2}$-$\underline{1}$-$\underline{3}$-5, 4-$\underline{3}$-$\underline{1}$-$\underline{2}$-5, 4-$\underline{1}$-$\underline{3}$-$\underline{2}$-5, 5-$\underline{1}$-$\underline{2}$-$\underline{3}$-4, 5-$\underline{2}$-$\underline{1}$-$\underline{3}$-4, 5-$\underline{3}$-$\underline{1}$-$\underline{2}$-4, 5-$\underline{1}$-$\underline{3}$-$\underline{2}$-4
No previction (nPR)	$\underline{2}$-$\underline{3}$-$\underline{1}$-5-4, $\cdots$, $\underline{2}$-$\underline{3}$-$\underline{1}$-4-5, $\cdots$, $\underline{3}$-4-$\underline{1}$-5-$\underline{2}$, $\cdots$

Table 3.5: Examples of Generative Testcase Specifications used as input to Plumber in experiment **E4**

	GTS	**Description**
(1)	$\langle[\mathtt{M}]5\rangle@$	1 Tag, 1 Set
(2)	$\langle[\mathtt{M}_{t_1,s_1}]4\ \mathtt{M}_{t_2,s_1}\rangle@$	2 Tags, 1 Set
(3)	$\langle[\mathtt{M}_{t_1,s_1}]3\ \ [\mathtt{M}_{t_2,s_1}]2\rangle@$	2 Tags, 1 Set
⋮	⋮	⋮
(12)	$\langle\mathtt{M}_{t_1,s_1}\ \mathtt{M}_{t_2,s_1}\ \mathtt{M}_{t_3,s_1}\ \mathtt{M}_{t_4,s_1}\ \mathtt{M}_{t_5,s_2}\rangle@$	5 Tags, 2 Sets

E3: Tags and Sets. We further check the effect of *exact tag and set values* on previction. We create GTSs which generate testcases that preserve the relations between tags and/or sets of loaded addresses, while randomly changing these addresses:

1. $|[\mathtt{M}_{t_1,s_1}]3\ \mathtt{M}_{t_2,s_1}\ \ \mathtt{M}_{t_3,s_1}|10000$
2. $|\mathtt{M}_{t_2,s_1}\ \ [\mathtt{M}_{t_1,s_1}]3\ \mathtt{M}_{t_3,s_1}|10000$
3. $|\mathtt{M}_{t_2,s_1}\ \ [\mathtt{M}_{t_1,s_1}]2\ \mathtt{M}_{t_4,s_2}\ \ \mathtt{M}_{t_3,s_1}|10000$

For every GTS, all 10,000 generated testcases show the same behavior. Thus, the exact values of tags and sets do not matter.

E4: Word Offset Behavior. In **E2**, we observed that the word offsets of loaded addresses affect previction. To broaden our understanding, we now leverage GTSs as shown in Table 3.5. They use the offset mutation operator to generate testcases for 5-load programs with all possible combinations of tags and sets (for loads targeting up to two cache sets) while mutating the word offset. On Cortex-A53, each cache line (64 bytes) is divided into four disjoint “buses” of 16 bytes (i.e., a cache line is loaded in 4 bus rounds). The two most significant bits of the word offset determine the bus index (see Figure 3.1). We find that, for three tag-identical loads in a sequence of five set-identical loads, previction occurs if the bus index of the first load is not the direct successor to the bus index targeted by the second load.

This experiment took approximately 1.5 days to complete. The most time-consuming part of this experiment is case number 12 in Table 3.5 which consists of 12×16^5 testcases.

E5: Priming the Cache. We further check if previction also affects data cached

before the execution of a testcase. We created GTSs which generate and execute previction testcases while between 1 and 4 lines of a cache set are occupied:

1. $\mathsf{P}(\mathtt{M}_{t_4,s_1})$ $[\mathtt{M}_{t_1,s_1}]3$ $\mathtt{M}_{t_2,s_1}$ $\mathtt{M}_{t_3,s_1}$
2. $\mathsf{P}(\mathtt{M}_{t_4,s_1}\ \mathtt{M}_{t_5,s_1})$ $[\mathtt{M}_{t_1,s_1}]3$ $\mathtt{M}_{t_2,s_1}$ $\mathtt{M}_{t_3,s_1}$
3. $\mathsf{P}(\mathtt{M}_{t_4,s_1}\ \mathtt{M}_{t_5,s_1}\ \mathtt{M}_{t_6,s_1})$ $[\mathtt{M}_{t_1,s_1}]3$ $\mathtt{M}_{t_2,s_1}$ $\mathtt{M}_{t_3,s_1}$
4. $\mathsf{P}(\mathtt{M}_{t_4,s_1}\ \mathtt{M}_{t_5,s_1}\ \mathtt{M}_{t_6,s_1}\ \mathtt{M}_{t_7,s_1})$ $[\mathtt{M}_{t_1,s_1}]3$ $\mathtt{M}_{t_2,s_1}$ $\mathtt{M}_{t_3,s_1}$

The results show that when the targeted set is half full (the second GTS), one of the two preloaded cache lines is evicted. As we show in Section 3.7, this insight enables a side-channel attack.

Previction Leakage Template. Our experiments resulted in Leakage Templates that allow us to identify previction side channels in applications. Figure 3.7 (b) illustrates an example Template. Each of the experiments substantially refined this template. **E1** dictated the general structure of five loads; **E2** contributed to the set and tag affinity; **E3** showed that we do not need to constrain certain tag/set values; **E4** revealed the bus relationship; and **E5** gave auxiliary information about previction behavior when caches are primed.

3.5.4 Case Study: Prefetching

Analyzing the ARM Cortex-A53's reference manual [9], we find that prefetching could leak the number of loads, the relation between loaded addresses (*stride*), the occurrence of cache misses, and the end of a page. We aim to validate this behavior and also examine whether undocumented behavior could leak sensitive information. For example, the ARM manuals do not answer any of the following questions:

- **Q1**: How much data is prefetched?
- **Q2**: Do non-memory operations influence the prefetching behavior?
- **Q3**: Do load operations in one page affect prefetching due to memory loads from a different page?

To this end, we again design five experiments (**E6**–**E10**).

E6: Prerequisites for Prefetching. First, we want to devise necessary conditions for prefetching and determine the number of prefetched cache lines (**Q1**). We create GTSs which generate testcases for all possible programs consisting of three to five loads while mutating their set index. We chose this range as one or two loads do not trigger prefetching and more than 5 loads would create too many testcases. This results in the following three GTSs:

$\langle[\mathtt{M}_{t_1,s_1}]3\rangle\$;\quad \langle[\mathtt{M}_{t_1,s_1}]4\rangle\$;\quad \langle[\mathtt{M}_{t_1,s_1}]5\rangle\$$

The main outcome of this test is a Leakage Template describing the relations between loaded cache lines and the number of prefetched addresses. Consider the GTS $\mathtt{M}_{t_1,s_1}\ \mathtt{M}_{t_1,s_2}\ \mathtt{M}_{t_1,s_3}$. Prefetching occurs when the distance between the three accesses

is constant and less than 4 cache lines ($\mathtt{s}_3 - \mathtt{s}_2 = \mathtt{s}_2 - \mathtt{s}_1 \leq \delta_{max} = 4$). Moreover, programs with 3–4 consecutive loads trigger the prefetching of 3 additional cache lines, while streams with 5 loads lead to 4 prefetched addresses.

For prefetching, **E6** is the most time-consuming case: It consists of 3 sets of experiments, each containing $2^{(7\times x)}$ testcases of x-loads programs, where $x \in \{3, 4, 5\}$. For the $x = 3$ case we perform the full experiment but for the two other cases we fix a few bits (1 and 2 bits respectively) of the set indices to make the experiment manageable. Overall, this experiment took 4 days to complete: 0.2d for $x = 3$, 1d for $x = 4$ and 2.8d for $x = 5$.

E7: Intermediate Instructions. Next, we want to check the effect of intervening instructions on prefetching (**Q2**). For this, we have created GTSs which generate testcases containing programs with a fixed stride and a varying number of intermediate arithmetic instructions. For $0 < n \leq 100$ and $0 < m \leq 30$ we create:

1. $\mathtt{M}_{\mathtt{t}_1,\mathtt{s}_1}$ $[\mathtt{A}]\mathsf{n}$ $\mathtt{M}_{\mathtt{t}_1,\mathtt{s}_1+1}$ $\mathtt{M}_{\mathtt{t}_2,\mathtt{s}_1+2}$ $\mathtt{M}_{\mathtt{t}_1,\mathtt{s}_1+3}$
2. $\mathtt{M}_{\mathtt{t}_1,\mathtt{s}_1}$ $\mathtt{M}_{\mathtt{t}_1,\mathtt{s}_1+1}$ $[\mathtt{A}]\mathsf{n}$ $\mathtt{M}_{\mathtt{t}_2,\mathtt{s}_1+2}$ $\mathtt{M}_{\mathtt{t}_1,\mathtt{s}_1+3}$
3. $\mathtt{M}_{\mathtt{t}_1,\mathtt{s}_1}$ $\mathtt{M}_{\mathtt{t}_1,\mathtt{s}_1+1}$ $\mathtt{M}_{\mathtt{t}_2,\mathtt{s}_1+2}$ $[\mathtt{A}]\mathsf{n}$ $\mathtt{M}_{\mathtt{t}_1,\mathtt{s}_1+3}$
4. $\mathtt{M}_{\mathtt{t}_1,\mathtt{s}_1}$ $[\mathtt{A}]\mathsf{m}$ $\mathtt{M}_{\mathtt{t}_1,\mathtt{s}_1+1}$ $[\mathtt{A}]\mathsf{m}$ $\mathtt{M}_{\mathtt{t}_2,\mathtt{s}_1+2}$ $[\mathtt{A}]\mathsf{m}$ $\mathtt{M}_{\mathtt{t}_1,\mathtt{s}_1+3}$

We create similar GTSs for 3- and 6-load streams. Our results show that adding instructions between consecutive loads can alter the number of prefetched addresses, e.g., adding 3 arithmetic instructions between two consecutive loads can increase the number of prefetched addresses from 3 to 4. Similarly, adding 4 arithmetic instructions increases this number to 7 and adding 5 instructions reduces it again to 3. Thus, the prefetcher may leak the control flow at the granularity of one instruction, a new insight which may lead to potential side channels (see Section 3.7).

E8: Respecting Page Boundaries. We also check whether prefetching respects page boundaries (as stated in the manual), i.e., if the processor prefetches addresses past the end of a page (**Q3**). To this end, we create GTSs to generate testcases containing programs with fixed strides while gradually shifting the loaded addresses towards the next page, i.e., up to one page ($64 * 64 = 4096$). For $0 < n \leq 5$ we create:

1. $(\mathtt{M}_{\mathtt{t}_1,\mathtt{s}_1}$ $\mathtt{M}_{\mathtt{t}_1,\mathtt{s}_1+n}$ $\mathtt{M}_{\mathtt{t}_1,\mathtt{s}_1+2n})\mathsf{64}$
2. $(\mathtt{M}_{\mathtt{t}_1,\mathtt{s}_1}$ $\mathtt{M}_{\mathtt{t}_1,\mathtt{s}_1+n}$ $\mathtt{M}_{\mathtt{t}_1,\mathtt{s}_1+2n}$ $\mathtt{M}_{\mathtt{t}_1,\mathtt{s}_1+3n})\mathsf{64}$
3. $(\mathtt{M}_{\mathtt{t}_1,\mathtt{s}_1}$ $\mathtt{M}_{\mathtt{t}_1,\mathtt{s}_1+n}$ $\mathtt{M}_{\mathtt{t}_1,\mathtt{s}_1+2n}$ $\mathtt{M}_{\mathtt{t}_1,\mathtt{s}_1+3n}$ $\mathtt{M}_{\mathtt{t}_1,\mathtt{s}_1+4n})\mathsf{64}$

The results show that testcases at the end of the page had fewer prefetched cache lines, indicating that prefetching does not cross page boundaries. Testcases with loads spread across different pages did not cause prefetching.

E9: Multiple Prefetching Sequences. We now explore how the prefetcher handles multiple, possibly interleaving sequences. To this end, we specify a GTS that merges three 3-load sequences with distinct tags and sets, i.e., from different memory pages.

$((\mathtt{M}_{t_1,s_1}\ \mathtt{M}_{t_1,s_1+1}\ \mathtt{M}_{t_1,s_1+2} : \mathtt{M}_{t_2,s_2}\ \mathtt{M}_{t_2,s_2+1}\ \mathtt{M}_{t_2,s_2+2})+ : \mathtt{M}_{t_3,s_3}\ \mathtt{M}_{t_3,s_3+1}\ \mathtt{M}_{t_3,s_3+2})+$

Our results show that the prefetcher becomes active for only the first two sequences; any additional sequence will *not* be prefetched. To decide which one is first, the prefetcher picks the first sequences of three consecutive loads (two strides). This means that multiple independent sequences can cause interference, even if they are on different pages (**Q3**). Again, this novel observation can lead to potential side channels to leak information (see Section 3.7).

E10: Cache Hits. Finally, we test the prefetcher's behavior for cache hits. We create GTSs that generate and execute prefetching testcases while one of the loaded addresses is cached:

1. $\mathsf{P}(\mathtt{M}_{t_1,s_1})\ \langle \mathtt{M}_{t_1,s_1}\ \mathtt{M}_{t_1,s_1+1}\ \mathtt{M}_{t_1,s_1+2}\rangle\$$
2. $\mathsf{P}(\mathtt{M}_{t_2,s_2})\ \mathtt{M}_{t_1,s_1}\ \mathtt{M}_{t_2,s_2}\ \mathtt{M}_{t_1,s_1+1}\ \mathtt{M}_{t_1,s_1+2}$

The first GTS does not trigger prefetching for all generated testcases, while the second GTS induces a behavior similar to that of $\mathtt{M}_{t_1,s_1}\ \mathtt{M}_{t_1,s_1+1}\ \mathtt{M}_{t_1,s_1+2}$. Thus, the prefetcher only monitors cache misses, i.e., preloaded data may destroy sequences that would have otherwise been prefetched. This is problematic if prefetched sequences from different pages (and contexts) interfere with each other.

Prefetching Leakage Template. Figure 3.7 (c) illustrates a Leakage Template for a 3-load stream, allowing us to identify prefetching side channels in applications. We construct this Template from the experiments above. In particular, **E6** identified constraints on the cache sets; **E7** derived bounds on intermediate instructions and their effect on the prefetching behavior; **E8** refined constraints on cache sets based on page boundaries; **E9** revealed the interference between interleaving prefetching sequences; and **E10** gave auxiliary information about the effect of cache hits on prefetching.

3.6 Matching Leakage Templates in Binaries

We now demonstrate how a Leakage Template can be used to identify an instance of the side channel it describes in a target binary. This addresses RQ1.2.

Target Binary and Side Channel. As a proof of concept, we re-identify a known prefetching-based side channel in OpenSSL 1.1.0g. This channel was first identified and exploited on an Intel CPU by Shin et al. [137]. The side channel exploits that data-dependent accesses to a lookup table may or may not trigger the prefetcher, depending on the memory access pattern that occurs. If the accesses to the table (accidentally) form a regular pattern, the prefetcher is activated and nearby cache lines are brought into the cache. These cache lines can therefore be exploited as an indicator of whether a relation exists between the accessed lookup table elements or not. By extension, this reveals information about the processed data. Shin et al. exploit such relations to leak the scalar of a scalar point multiplication on an elliptic curve. In Elliptic Curve Diffie-Hellman (ECDH), a scalar represents the private key. The attack recovers the key incrementally. The same computation is applied to both the target scalar and a

candidate scalar. By changing the candidate scalar such that the prefetching behavior matches, both scalars converge. Even though this vulnerability is no longer present in recent OpenSSL versions, we still consider it a reasonable case study to demonstrate that Leakage Templates can be used to identify real-world vulnerabilities in binaries.

Approach: Combining Static and Dynamic Analysis. Shin et al. [137] limit the scope of their search to a specific cryptographic operation. In contrast, our starting point is the whole OpenSSL binary. We combine static and dynamic binary analysis techniques to search it for instances of the prefetching Leakage Template (see Figure 3.7 (c)). First, we scan the binary for code sections that match the code pattern $\mathcal{P}(A)$ of the Template. This results in a list of candidate code sections that potentially contain a prefetching side channel. Second, we need to check whether a candidate section satisfies different relations $\mathcal{R}(A, b)$ for different input values. If this is the case, we expect the section to show input-dependent behavior, indicating a side channel. Not all relations can be resolved statically, especially if they refer to addresses in instruction operands. To overcome this, we dynamically analyze the target code to learn its concrete addresses.

Performing Static Analysis. We use *asmregex* [18] to statically scan the target binary for the code pattern $\mathcal{P}(A)$ of the prefetching Leakage Template. Asmregex searches binaries for code sections that match a specified pattern. We extended the tool by approx. 200 LoC to support a subset of the ARM instruction set and added support for backreferences to the pattern language. Backreferences allow us to express simple relations between instructions. For instance, two subsequent load instructions can be required to use the same base address register. To identify code sections matching $\mathcal{P}(A)$ in OpenSSL, we convert $\mathcal{P}(A)$ into an asmregex pattern. This pattern matches 429 3-load sequences across 18 OpenSSL modules. By briefly inspecting the matching candidate sections, we identify accesses to lookup tables in 11 of these modules. The remaining matches are predominantly caused by operations on complex data structures. Most importantly, we identify the code section exploited in [137] among the candidates in the module `crypto/bn/bn_gf2m.o`.

Performing Dynamic Analysis. We proceed with the dynamic analysis step to check a candidate code section for input-dependent behavior using the relations $\mathcal{R}(A, b)$ from the Leakage Template. We create a simple wrapper program that calls the matching library function with varying input values. This program can be used to log all (input-dependent) loads from the relevant lookup table `SQR_tb`, which spans across three cache lines in memory. We record two different traces for each input value. First, we use Valgrind [107] and GDB [38] to record an *access trace*, a list of all loads from `SQR_tb` during program execution. This trace can be used to determine the *expected* prefetching behavior based on the relations $\mathcal{R}(A, b)$ from the Leakage Template. Second, we use a Flush+Reload side channel to record a *cache trace*. This trace contains the cache state of the memory lines around `SQR_tb` after execution. It is captured for evaluation purposes and indicates the *actual* prefetching behavior of the CPU.

In order to show that the Leakage Template accurately represents the prefetching behavior, we recorded traces for 100 random input values to the library function. For each input value, we determined the expected prefetching behavior using the access trace and compared it with the actual behavior using the corresponding cache trace. As

Table 3.6: Template matching confusion matrix, comparing relation-based prefetching behavior classification with the actual behavior

		Classification		
		P_0	P_3	undecidable
Actual behavior	P_0	66	0	0
	P_3	0	6	28

we found in experiment **E10** that the prefetcher only operates on cache misses, the load instructions relevant to the prefetcher are not necessarily the first three load instructions in the matching code section. Therefore, we perform our analysis based on the first three loads in each access trace that target different cache lines.

Evaluation. Table 3.6 illustrates the classification performance. For all 66 cases where the load instructions satisfy the relations for the behavior P_0, the cache traces show that no prefetching occurred. In six cases, the relations for behavior P_3 are satisfied. The three relevant load instructions load data from three consecutive cache lines and the number of instructions between the load instructions (n_1 and n_2) is within the specified bounds. In all six cases, the cache trace shows that prefetching of three additional cache lines occurred. In the remaining 28 cases, the relations for none of the behaviors from the Leakage Template are satisfied. The reason is that the distances n_1 and n_2 between the relevant load instructions are outside the parameter range we tested when the Template was created. We denote these cases as *undecidable* cases. We note that no misclassifications occurred.

Conclusion. We successfully demonstrated that the prefetcher of the Cortex-A53 CPU shows input-dependent behavior for the library function under investigation. This is the base requirement for the differential attack in [137]. The Leakage Template helped us to re-identify this vulnerability known from the Intel architecture in ARM binary code. In contrast to prior work, our starting point was the whole OpenSSL code base. For code sections that closely match the Template (i.e., they closely correspond to code and relations that Plumber encountered during creation of the Leakage Template), the behavior classification based on the relations is accurate. When unknown relations occur, undecidable cases are more likely to appear. In our example, undecidable cases occur due to higher values of n_1 and n_2 than we used when creating the Template (to keep the number of testcases within a reasonable range). However, these cases can be detected and the analyst may use them to design further experiments in order to refine the Template in a targeted manner. This highlights again that a Leakage Template, which can hardly ever be complete, can be developed in an iterative fashion.

3.7 Novel Leakage Primitives

Our experiments in Section 3.5 also helped us to identify five novel prefetching-based leakage primitives. In this section, we transform them into covert channels and evaluate the performance of four of them in practice (we skip the `PR_PP` covert channel as it is not applicable in our shared-memory covert-channel setup).

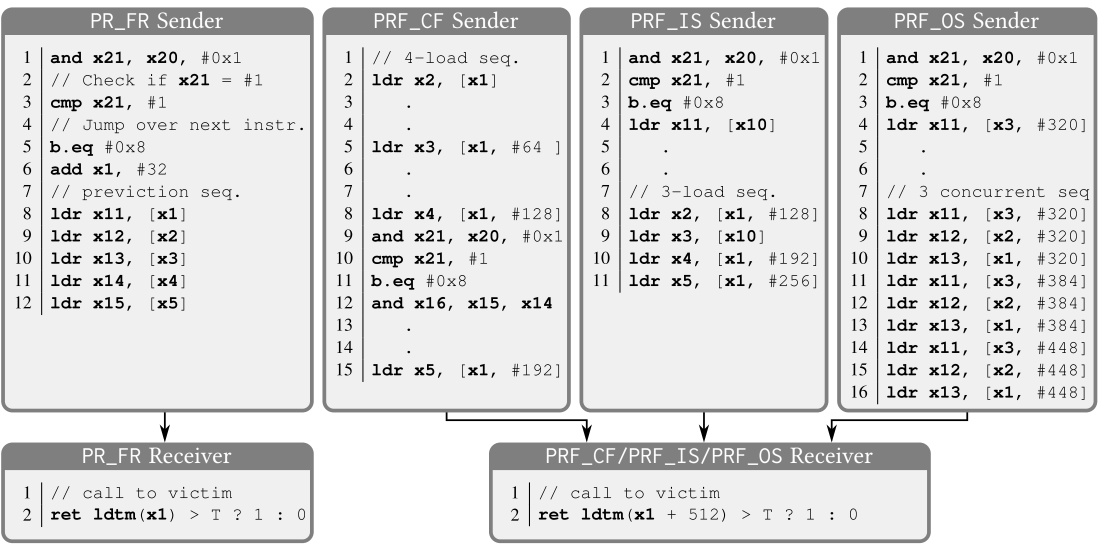

Figure 3.8: Code examples for covert channels based on our leakage primitives. `ldtm` measures the `ldr` execution time for the given address, `T` is a threshold value.

For each channel, we present a minimal code example for a sender and a receiver function in Figure 3.8, which we explain in detail in the following subsections.

As Figure 3.9 illustrates, we can clearly distinguish transmitted bits in the majority of the cases for all evaluated covert channels. We further report transmission and error rates of our channels in Table 3.7 and place them in the context of similar channels in related work.

3.7.1 Prevíction With Shared Memory (`PR_FR`)

Our prevíction-based Flush+Reload primitive `PR_FR` is based on our insights from **E4** in Section 3.5.3. Unlike traditional Flush+Reload primitives, `PR_FR` allows leaking information based on *bus* granularity rather than cache line granularity. The idea is to make the (observable) occurrence of prevíction dependent on a secret bit (the leak target) by changing bus relations.

The primitive `PR_FR` in Figure 3.8 leverages the strong bus dependency between the consecutive load instructions in a valid prevíction sequence. Let the lines 8–12 be a valid prevíction sequence following our Leakage Template in Figure 3.7 (b), i.e., `x1`–`x3` being consecutive loads with a “valid” bus relation, and `x4`/`x5` arbitrary other loads from the same set. The idea is to use a secret-dependent conditional change to the byte offset of the first load (`x1`) to cause or prevent prevíction (lines 1 through 6). More precisely, the word offset of the first address loaded (in `x1`) depends on the value of the least significant bit of the data stored in `x20`. According to the Template, the relation between the offsets in the addresses in `x1` and `x2` determines if `x1` will be prevícted from the cache. By measuring the time required to load from the address stored in `x1` and comparing the result to an empirically determined threshold `T`, the receiver can test whether this address resides in cache or not, thus learn whether prevíction happened or not, and consequently deduce the secret bit.

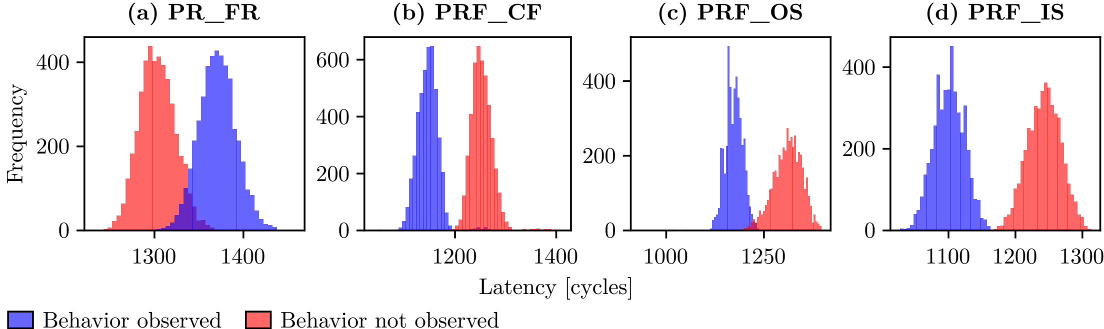

Figure 3.9: Histograms comparing the load latency of address `x1` at the receiving end of the covert channels. Blue bars indicate that the respective behavior (previction or prefetching) was observed, red bars indicate the opposite case. Each experiment was repeated 4500 times per case. For all four channels, both cases are clearly distinguishable, with only small overlaps.

Table 3.7: Transmission and error rates of our covert channels in the context of related work

Covert channel (Element)	Transmission rate	Error rate
Liu et al. [90] (L3)	600 kbit/s	1%
Pessl et al. [119] (DRAM)	411 kbit/s	4.11%
Maurice et al. [99] (L3)	362 kbit/s	0%
PRF_IS	276 kbit/s	0.05%
PRF_OS	206 kbit/s	2.1%
PRF_CF	76 kbit/s	0.7%
PR_FR	73 kbit/s	1.2%
Maurice et al. [98] (L3)	751 bit/s	5.7%
Wu, Xu, and Wang [159] (memory bus)	747 bit/s	0.09%
Semal et al. [136] (memory bus)	480 bit/s	5.46%
Schwarz et al. [134] (DRAM)	11 bit/s	0%

3.7.2 Previction Without Shared Memory (`PR_PP`)

Based on experiment **E5** in Section 3.5.3, previction may target preloaded memory addresses and leak information in the absence of shared memory, e.g., through Prime+Probe. The sender code of our previction-based Prime+Probe primitive `PR_PP` is similar to that of `PR_FR`. However, in `PR_PP`, the receiver first loads two memory lines into the targeted cache set before the execution of the sender code. The receiver then probes the lines to determine the leaked bits.

3.7.3 Prefetching Control-Flow Leakage (`PRF_CF`)

`PRF_CF` allows leaking the control flow of a program based on prefetching. It is based on the results of **E7** in Section 3.5.4. Figure 3.8 shows a code example code for `PRF_CF`. The sender code executes a 4-load prefetching sequence with a fixed stride (lines 2, 5, 8, and 15). The loads are separated by a number of arithmetic instructions. The instruction at line 12 is conditionally executed depending on one bit of a secret that is stored in `x20` (lines 9 through 12). According to **E7**, the number of executed instructions within a prefetching sequence affects the number of prefetched cache lines. By measuring the time required to reload a (possibly prefetched) address `x1`+512, the receiver can determine whether the conditional load instruction was executed and consequently learn the secret bit.

3.7.4 Prefetching on an Interrupted Sequence (`PRF_IS`)

Inspired by **E7**, we analyze and exploit the effect of intermediate memory operations on prefetching. We observe that an intermediate load from a different page leads to prefetching of additional cache lines by a 3-load stream. `PRF_IS` is based on this outcome. It also allows leaking accesses to non-shared memory through shared pages.

As shown in Figure 3.8, the sender code contains a 3-load prefetching sequence with a fixed stride and an interleaving load from a different page, i.e., [`x10`] (lines 8 through 11). Since the processor ignores cache hits, the number of prefetched lines will depend on whether `x10` is cached. Consequently, by measuring the time required to load from `x1`+512, the receiver can determine whether `x10` has been cached and consequently learn the secret bit.

In contrast to the prefetching experiment in Section 3.5.4, `PRF_IS` checks how the prefetcher's behavior changes when it observes interleaved loads from different pages (i.e., across page boundaries). Note that in `PRF_IS` all predicted addresses are within bounds of the same page (in accordance with **E8**).

3.7.5 Prefetching on an Outstanding Sequence (`PRF_OS`)

`PRF_OS` exploits competing prefetching sequences to leak accesses to non-shared memory through shared pages. In other words, it allows leaking secrets through Flush+Reload even when secret-dependent memory accesses are from non-shared memory. `PRF_OS` is based on the outcome of experiments **E9** and **E10** in Section 3.5.4.

As shown in Figure 3.8, the sender has three interleaving 3-load streams with fixed strides (lines 8, 11, 14; lines 9, 12, 15; and lines 10, 13, 16). These streams are preceded

by a load from the address `x3`, whose execution depends on one bit of a secret stored in `x20` (lines 1 through 4). According to **E9**, memory addresses are prefetched for sequences whose strides are detected first. Additionally, according to **E9**, the processor ignores cache hits when detecting prefetching sequences. Consequently, depending on whether `x3` is cached, prefetching would be triggered for either the first or the second two prefetching sequences. By measuring the latency of loading from `x1`+512, the receiver can determine whether prefetching occurred for the third stream and consequently learn the secret bit.

3.8 Related Work to Leakage Templates and Plumber

We now place Leakage Templates and our framework Plumber within the context of related work. Specifically, we discuss approaches that can be used to generate counterexamples for analysis with Plumber, highlight related (but different) approaches for identifying side channels in existing software, and identify side channels similar to those in our case studies (previction and prefetching).

3.8.1 Side-Channel Discovery and Characterization

Besides Scam-V [105], which we introduced in Section 3.2.2, other approaches have been proposed that can be used to find counterexamples, i.e., starting points for an analysis with Plumber and Leakage Templates.

Timing-Based Side Channels. Weber et al. [153] present Osiris, which detects instances of side-channel leakage by randomly generating three-part instruction sequences and evaluating their timing behavior. In particular, each sequence consists of *reset*, *trigger*, and *measurement* sub-sequences. For both cases, the execution time of the measurement sequence is measured. If it differs, a code example showing side-channel leakage has been discovered, which could serve as input to Plumber.

Differential Fuzzing. Various differential fuzzing approaches have been proposed to perform side-channel analysis. Nilizadeh, Noller, and Pasareanu [109] execute a given program with two different secret-related inputs and measure the program's execution time and memory usage as metrics to guide a fuzzer. Specifically, the fuzzer is instructed to mutate the inputs such that the difference in execution time or memory consumption is maximized, thus exposing potential side-channel instances. Thomas et al. [143] run instruction sequences on different processors and emulators and compare the observed behavior. Any difference may indicate a CPU bug, and could serve as a starting point to derive a Leakage Template.

Spectre and Meltdown Variants. Other works propose approaches to discover and characterize Spectre- and Meltdown-type vulnerabilities. Moghimi et al. [103] focus on discovering Meltdown-style side channels: They introduce Transynther, a tool that randomly mutates building blocks of known Meltdown attacks, aiming to find new variants. Oleksenko et al. [110, 111] focus on finding Spectre-style vulnerabilities, applying a similar approach to Scam-V. Their tool, Revizor, tests real-world CPUs for compliance with *speculation contracts*. Those contracts describe the expected speculative behavior of the ISA [47]. On a high level, Revizor generates and executes test programs

on the hardware with different sets of input data. If two sets of input data are predicted to behave the same according to the speculation contracts but behave differently on the hardware, this is considered a counterexample. Hofmann et al. [55] extend Revizor and the underlying leakage model to support Meltdown-style vulnerabilities as well. In contrast to our work, neither Transynther nor Revizor identify a generic pattern to identify instances of those channels in existing code. While Plumber and our domain-specific language currently lack the required directives to perform Spectre- or Meltdown-style experiments (such as a branch directive or a way to cause a faulting load from a location with specific page table attributes), the extensible nature of our work allows for adding such features in the future.

Contention-Based Side Channels. Gras et al. [41] investigate contention-based side channels across SMT cores (backed by the same physical core). Their tool, ABSynthe, finds contention side channels in existing programs. ABSynthe instruments the target program with spy code in order to allow for an attacker to iteratively refine a contention-based channel that can then be used to attack the equivalent non-instrumented binary. Notably, ABSynthe does not produce a generic representation of the channel and relies on re-training for every software new under investigation.

Prefetcher-Based Side Channels. In Chapter 4, we design and implement a methodology to identify and characterize hardware prefetchers on a given CPU. In contrast to Plumber's bare-metal approach, we perform this analysis through Linux-based software tests, allowing for simpler scaling across different architectures.

3.8.2 Identifying Side Channels in Existing Software

Recent work on the identification of side-channel vulnerabilities in existing software has mainly focused on finding code that is vulnerable to Spectre-type attacks. SpecFuzz [112] instruments a program to simulate speculative execution. In particular, it simulates mispredictions by explicitly executing speculative paths. Vulnerable paths are detected by identifying out-of-bounds memory accesses during simulated speculative execution. We note that this detection approach cannot be generalized to all cache-based side channels, as many of them do not perform transient out-of-bounds accesses. Spectector [46] and InSpectre Gadget [156] utilize symbolic execution to explore speculative program paths and identify Spectre-style transmission channels. SpecTaint [121] identifies Spectre gadgets in a given binary by searching for occurrences of abstract patterns of Spectre gadgets. In addition, it uses taint tracking to track attacker-controlled variables. Gadgets that contain attacker-controlled variables are considered more likely to be exploitable.

All these works discover side channels in existing software, but do not provide a generic methodology to derive a high-level characterization of the underlying side channel.

3.8.3 Related Side Channels

Cache-Timing Primitives. The basis of most cache-based side channels is a primitive such as Prime+Probe [114], Flush+Reload [161], as described in Section 2.4. Typically, those primitives use a timed load instruction to test whether or not a particular cache

line is present in cache. In contrast, since we take the role of an analyst who has full control over the system, we use the privileged RAMINDEX debug interface [9] for this purpose. This interface increases the accuracy of our measurements and reduces the noise level.

Previction. Similarly to our previction side channel on an ARM processor, Yarom, Genkin, and Heninger [162] find that cache bank conflicts on Intel Sandy Bridge processors cause measurable timing inconsistencies on Intel processors. They exploit this behavior to leak the targets of memory accesses at sub-cache-line granularity, allowing them to break a scatter-gather implementation of modular exponentiation that was previously considered constant-time.

Prefetching. Our findings on prefetch-based side channels lay the foundation for a deeper analysis of the characteristics of prefetching in modern processors. We explore this topic from an offensive perspective in Chapter 4 and take the defensive role in Chapter 5. In particular, we refer the reader to Section 4.2.1 for a discussion of various prefetching approaches and to Section 5.3 for an in-depth systematization of prefetching-based attacks in literature.

3.9 Discussion of Microarchitectural Leakage Templates

Extendability. Depending on the target architecture and the concrete component under investigation, basing experiments only on the attributes of cache set indices and cache tags might be too limiting. In this case, our Leakage Templates and the Plumber framework can be extended to support the missing attributes.

For example, modern processors may further sub-divide caches into *slices*, introducing an additional variable that is derived from the physical address and might impact microarchitectural behavior. The slice mapping is often based on proprietary hash functions. To consider such mappings with Plumber, the hash function first needs to be reverse-engineered [74, 97, 123]. Once the hash function is known, the notion of a cache slice can be added to the memory directive (`M`) in our GTS language.

Limitations. Due to the complexity of microarchitectural features and the variety of potential side channels, the current implementation of Plumber focuses on cache-based side channels. The implementation of some components is limited to ARM architectures. However, the concept of Leakage Templates and the design of Plumber are generic. They are not constrained to cache-related channels and could also be ported to other architectures. The main challenge of such an adaptation is porting Plumber's code generation and cache inspection modules. Moving to a new architecture would require replacing the RAMINDEX inspection module with a less precise probing mechanism such as Flush+Reload [161] or Flush+Flush [44].

Plumber vs. Scam-V. Plumber's goal is to facilitate understanding microarchitectural behavior, e.g., triggers and effects, via easily constructed queries. The approach of Scam-V [105] (and similar tools, e.g., [41, 153]) is complementary to that of Plumber. This relation is better expressed as a two-step approach: First, Scam-V or similar tools are used to detect a possible channel by specifying the monitored component, e.g., the cache state, which often is dictated by the vulnerability discovery tool (e.g., port contention in [41], or execution time in [153]). Whenever the tool discovers yet-unsupported

types of side channels on monitored components, Plumber can be used to learn the correlation between (attacker-controlled) inputs and the channel (see Figure 3.5). The challenging aspect of such an integration is the generalization of concrete code examples, which can be provided by the analyst.

Scam-V finds a channel by executing randomly generated program-input pairs. Using an SMT solver to generate inputs requires careful engineering of the queries sent to the solver, otherwise generated inputs and thus counterexamples will be too similar which makes them not suitable for statistical analysis. Conversely, Plumber makes input generation more efficient by using GTS and allows learning the correlation between inputs and the channel.

3.10 Conclusion on Microarchitectural Leakage Templates

In this chapter, we addressed the question: *How can we systematically abstract from a concrete example of side-channel behavior in a program to learn its generalized characteristics, and how can we find other instances of the same side channel in other programs?* Specifically, we introduced the concept of Leakage Templates to abstractly describe a code pattern and the relations between input parameters that, when satisfied, can trigger specific microarchitectural behavior. As details on microarchitectural features, such as caches, previction, and prefetching, are scarce, derivation of Leakage Templates is challenging. Expressive specifications for testcases are needed, and a large set of inputs has to be explored. Also, techniques for the discovery of relations between code, data, and leakage behavior are needed.

To address those challenges, we proposed Plumber, our novel framework that leverages instruction generation, instruction operand mutation, and statistical analysis to explore under-specified behavior of microarchitectural features. Plumber's high-level goal is to facilitate the understanding of microarchitectural behavior, and thus, the creation of Leakage Templates.

We showed the utility of Leakage Templates by identifying five novel side-channel primitives for an ARM Cortex-A53 core. In addition, we demonstrated that Leakage Templates enable automated identification of code sections in existing application binaries that are vulnerable to side-channel attacks. We showed how static and dynamic binary analysis can be used to find instances of the side channel specified in a Leakage Template in a target binary.

Acknowledgment

The work presented in this chapter was supported in part by the German Federal Ministry of Education and Research (BMBF) through funding for the CISPA-Stanford Center for Cybersecurity (FKZ: 13N1S0762).

4

FetchBench: Systematic Identification and Characterization of Proprietary Prefetchers

4.1 Introduction to FetchBench

In this chapter, we investigate the characterization of hardware prefetchers in more detail. The problem of detecting and accelerating memory access patterns with prefetchers can be approached in many different ways, and a multitude of prefetcher design options is available to the chip designers [35]. Because prefetch operations are transparent to software applications, CPU vendors usually do not disclose any detailed information on the types of prefetchers they use, characteristics of these prefetchers, and their implementation. For example, Intel has confirmed the existence of certain types of prefetchers by disclosing documentation on how to disable them [70] but limits public documentation of their characteristics to superficial recommendations for software developers [69]. ARM is even more restrictive: For instance, the technical reference manual for the ARM Cortex-A72 confirms the existence of a prefetcher for data but does not even specify what kind of memory access patterns it detects [8].

This secrecy is especially problematic since security researchers identified several vulnerabilities that arise out of specific prefetcher implementations. For example, prior work characterized the stride prefetcher in certain Intel CPUs and exploited the reverse-engineered characteristics to build covert channels [30, 32] or to leak private keys from ECDH [137] or RSA [30] computations. Researchers also identified and characterized a prefetcher that dereferences pointers on the Apple M1 [131]. They exploited this prefetcher as an oracle for the validity of virtual addresses and thus to circumvent Address Space Layout Randomization (ASLR). These examples show that prefetchers are a security-relevant component of a CPU that requires systematic investigation. However, all prior work focused on a narrow set of prefetcher designs and CPUs; so far, no systematic analysis of a wide range of prefetcher architectures and their security implications has been provided.

Research Questions and Challenges. In this chapter, we close this gap by addressing the high-level research question (RQ2): *How can we systematically identify and characterize hardware prefetchers and their security implications in real-world processors?* We break this problem down into the following three questions:

RQ2.1: How can we systematically identify and characterize under-specified prefetching mechanisms in proprietary processors?

RQ2.2: What are the main types and characteristics of prefetchers in modern CPUs?

RQ2.3: What are the security implications of so-far unexplored prefetcher types?

We identified the following three main research challenges when addressing these questions: First, we need to collect and systematize possible prefetcher designs that CPU vendors are likely to use, despite having only limited documentation available. In addition, we have to identify the prefetcher's relevant implementation variables (*characteristics*). Second, as a CPU does not directly expose the prefetcher's internals, the identification and characterization of the prefetcher have to be performed based on

Figure 4.1: Proposed approach

its behavior during operation. While prior work mainly focused on characterizing a specific type of prefetcher [30, 32, 131, 137], we need solutions that allow identifying multiple prefetcher designs and characterizing each of them appropriately. Third, we want to verify that security-relevant inner workings of prefetcher designs enable new vulnerabilities. Such demonstrations will also require means of synchronization between victim and attacker code.

Proposed Approach. We first classify different prefetcher designs and create a taxonomy (see Figure 4.1), allowing us to relate prefetcher designs to each other and to identify common properties as well as fundamental differences. We then design and implement FetchBench, a modular framework that uses our taxonomy to identify a prefetcher design on a given CPU. In addition, our framework characterizes the prefetcher w.r.t. the most relevant characteristics (such as the trigger method or prefetch depth) for this particular design. We apply our framework to 19 processors of seven different vendors and two architectures (ARMv8 and x86_64).

With FetchBench, we are able to identify a variant of the replay-based Spatial-Memory-Streaming (SMS) prefetcher [139] in three ARM Cortex-A-series processors. We investigate the security properties of this prefetcher, which has received little attention so far. We demonstrate use cases for selected vulnerabilities. In particular, we exploit the SMS prefetcher to leak half of an AES key across process boundaries and show that the prefetcher's state is exempted from the usual separation between secure and non-secure world on TrustZone-enabled ARM CPUs. We build a covert channel that has a data rate of 1245 Bytes/s at 5.02% error rate.

Contributions. Our main contributions are as follows:

- We compile and classify seven hardware data prefetcher designs based on a novel taxonomy.
- We design and implement FetchBench, a modular framework to identify and characterize prefetchers on real CPUs, and apply those tests to 19 CPUs of seven different vendors and two different architectures. FetchBench uncovers and characterizes a previously unknown, replay-based prefetcher on a series of ARM Cortex-A processors.
- We present a novel attack on the replay-based prefetcher of the ARM Cortex-A72,

leaking 64 bits of an AES-128 key through the prefetcher. As part of the attack, we address the non-trivial problem of synchronizing the attacker process with the victim process to make the attack practical on Linux in presence of a scheduler.

- We show that the prefetcher's internal state is not separated between normal and secure world, leaking metadata across the TrustZone privilege boundary.

Disclosure. ARM acknowledged our findings on shared prefetcher state across hardware contexts. They initially reserved CVE-2023-33936, but later decided to not assign the CVE. ARM published an advisory [13] (initially released in 2023 and updated in 2025) and updated the architecture documentation [15] to reflect that leakage due to hardware prefetching is possible. In particular, the advisory explains: *"Arm has relaxed FEAT_CSV2 to only apply to virtual address-based cache prefetch predictions based on, or causing dereference of, data values from memory, which encompasses the more advanced class of prefetchers with a higher security risk. As such, predictive leakage from other hardware prefetchers is not considered to violate the security guarantees of FEAT_CSV2."* [13]

4.2 Systematic Classification of Hardware Prefetchers

In this section, we systematize and identify prefetcher designs commonly found in CPUs and in academic literature, answering RQ2.1. In Section 4.2.1, we propose a two-level taxonomy based on the strategy used to decide what to prefetch and on the information source the prefetcher is trained on. Based on our taxonomy, we present a framework to automatically detect and characterize prefetchers (Section 4.2.2).

4.2.1 Hardware Prefetcher Taxonomy

Prefetcher Design Selection. A plethora of prefetcher designs have been discussed in the past, either in form of academic proposals, reverse-engineering results, or hardware documentation. With our goal of identifying prefetchers *in real-world hardware* in mind, we focus on designs that we consider likely to be implemented. We use the *Primer on Hardware Prefetching* by Falsafi and Wenisch [35] as a starting point. From their selection, we focus mainly on designs that (i) are known to be present in real-world hardware or (ii) do not require large (off-chip) data structures to keep their state. We follow the conjecture of Ayers et al. [20] that prefetchers with high memory requirements have a higher "implementation burden" and are less likely to be implemented. We further include designs that are known to be present in real-world hardware but not covered by Falsafi and Wenisch.

Figure 4.2 summarizes our prefetcher classification with respect to two dimensions that we describe in the following: (i) the prediction strategy and (ii) the source of prediction knowledge.

Prediction Strategy. We distinguish two types of strategies used for the prediction, *extrapolation* and *replay*:

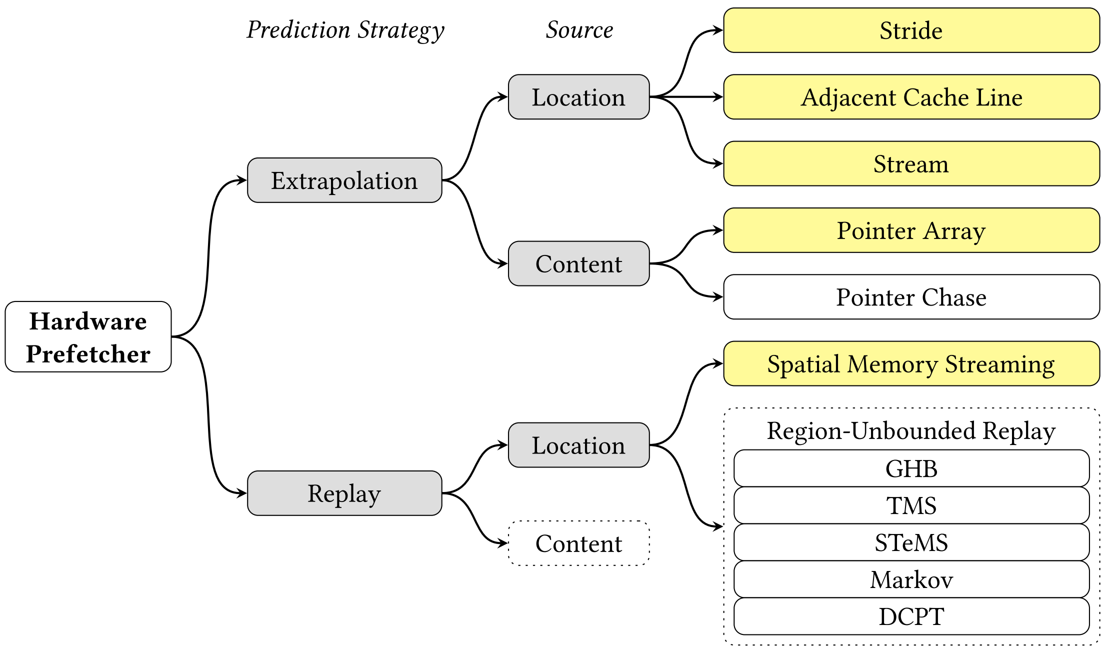

Figure 4.2: Prefetcher taxonomy. Yellow leaf nodes identify prefetcher designs that we experimentally observed in CPUs.

- *Extrapolation*-based prefetchers detect sequences in recent memory accesses and extend these by prefetching likely future accesses—notably irrespective of (and agnostic to) past accesses to these prefetched locations.
- *Replay*-based prefetchers repeat known access patterns and thereby prefetch previously-observed accesses (only)—without trying to extend access sequences.

Extrapolation-based prefetchers aim to identify *and extend* memory access patterns. Once identified, the prefetcher uses these patterns to compute ("extrapolate") which memory locations are *likely* to be accessed next. This way, those prefetchers can boost performance even if a memory region is not accessed repeatedly. Usually, extrapolation-based prefetchers do not need to maintain an extensive access history, and—depending on the prefetcher—can even be fully stateless. Consequently, such prefetchers typically have a more lightweight internal state. A disadvantage of these prefetchers is that they cannot predict more complex access patterns.

Replay-based prefetchers *repeat* past memory accesses patterns. Unlike extrapolating prefetchers, they are not bound to simple patterns (e.g., monotonic access sequences). Instead, they rely on more involved data structures to capture even complex access patterns. The prefetched access patterns are thereby identical to (or a subset of) previously-observed accesses. Replay prefetchers do *not* extend access patterns but may replay accesses in other contexts (e.g., at different locations). Designs in this domain differ in the way how the history is stored and filtered and how the prediction is triggered. Some prefetchers operate on fixed-size memory regions [139], others are region-unbounded. Moreover, there are different trade-offs between memory consumption and accuracy, e.g., by deciding to only store longer and more frequent patterns [35, 154].

Prediction Source. For the second level of our classification, we use the source of the prediction, i.e., what property the hardware prefetcher uses to decide on the address to prefetch. We distinguish between *location*-based and *content*-based sources.

- *Location*: Only the location, i.e., address, of the memory load is used to decide what to prefetch.
- *Content*: The content of the memory load is used to decide what to prefetch.

The location of memory loads is a common source for many hardware prefetchers, both for extrapolating as well as for replaying prefetchers. Typically, these prefetchers detect common patterns found in many applications [20] or previously-observed access patterns. Simple patterns include strides, i.e., the linear access to memory with a constant offset between the accessed memory addresses. Based on the strategy of the prefetcher, a detected pattern either leads to an extrapolated completion or a history-based completion.

Prefetchers using the content of memory loads as a prediction source can prefetch indirect memory loads. The prefetcher predicts future accesses based on the observed value, not on the address where a value is stored. Such prefetchers can (heuristically) detect pointers and start to prefetch the target memory location, e.g., to improve the performance of linked lists or trees. These data structures are usually not contiguous, as the next item that is accessed during a list traversal is only referenced via a pointer.

4.2.2 FetchBench: Identifying and Characterizing Prefetchers

In this section, we propose FetchBench, an automated modular software framework to identify and characterize hardware prefetchers. While knowledge about the existence and characteristics of prefetchers can, in principle, be acquired manually, such tedious efforts are typically bound to single CPU models. Time-consuming manual attempts thereby cannot generalize the findings to the by-now large landscape of CPU models.

Deep knowledge of prefetchers and their implementations is critical for security. First, the sole existence of a certain prefetcher determines if a given system is vulnerable to prefetcher-based side-channel attacks [30, 137] *in principle.* Second, only the precise knowledge of the exact characteristics of a given prefetcher implementation—which is normally not or only sparsely documented by CPU vendors—helps to understand if a particular defense is effective. Third, new insights into prefetcher implementations may reveal novel prefetcher-based side-channel vulnerabilities.

System Model. An analyst aims to identify and characterize prefetcher implementations that exist on a given CPU. We assume that the analyst has full software access to a computer system with the CPU under investigation. Specifically, they are free to install any software or operating system of their choice and have full root access to the system. They are permitted to read and modify arbitrary values in model-specific registers, especially those that are related to prefetching, even if they require privileged access. They have access to a precise timing source and a cache line flushing primitive. The analyst is limited to the publicly available documentation on the CPU under investigation and has no insider knowledge of its internals. They are unable to modify the CPU physically in any way.

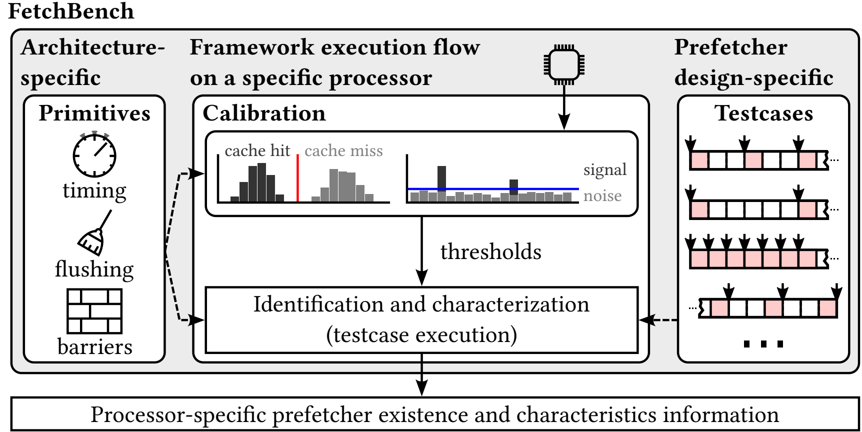

Figure 4.3: FetchBench framework

Overall Design. Figure 4.3 provides a high-level overview of FetchBench. The framework consists of one module for each potential prefetcher design. Each module implements testcases that first detect whether a prefetcher is implemented and, if so, what its characteristics are. We focus especially on characteristics that we consider security-relevant, i.e., those that may enable a side channel or impact the quality or throughput of a potential side channel. Modules essentially provide memory access patterns and queries for the cache state of memory locations. Based on a sequence of tests, a module infers the inner workings of the prefetcher. Tests are architecture-agnostic, as only access sequences and expected cache states are defined. This modular design allows adding new tests for different prefetchers. The functionality to access memory and query the cache state of specific cache lines is common for all modules but specific to the architecture that the framework runs on. On a given CPU, the framework first calibrates the primitives used to measure cache states. It then runs the testcases specified by the modules for the different prefetcher designs. In the end, the experimental results indicate the existence and characteristics of the tested prefetcher designs on the CPU. For our FetchBench prototype, we implement modules for seven prefetchers. In the following, we describe the design of these modules and how they detect and characterize hardware prefetchers.

4.2.2.1 Stride

Stride prefetching [21, 35] aims to detect sequences of accesses that are spaced a constant offset (*stride*) apart from each other. Once the prefetcher has detected such a pattern, a trigger causes it to extrapolate: The prefetcher predicts data locations by adding multiples of the detected stride to the last accessed address. Figure 4.4 (a) shows a sequence of n memory accesses (in red) that form a pattern with a stride of $d = 3$ cache lines, along with potential prefetch locations (in blue).

Characteristics. We consider the following stride prefetcher characteristics: (i) Does the prefetcher also learn *backward* strides? (ii) Which are the *smallest/largest strides* that are supported? (iii) What is the *minimal training size*, how large is the *prefetched set*? (iv) Does the prefetcher *cross the page boundary*? (v) On which *granu-*

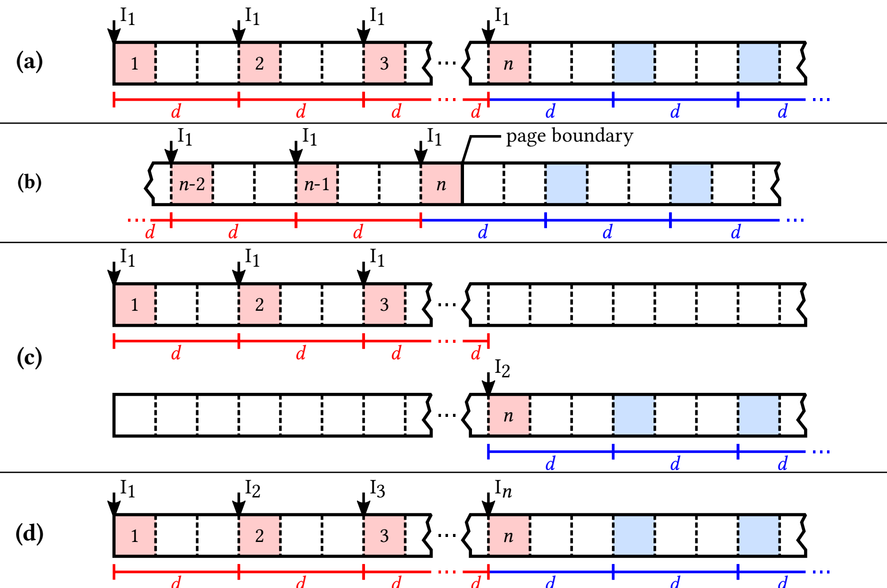

Figure 4.4: Stride prefetcher tests. Dashed boxes represent cache lines within a memory page. Arrows indicate loads and are labeled with the load instruction (I_x). Red CLs are cached due to architectural loads, blue CLs are potential prefetch locations. Boxes are numbered by the order of access.

larity, i.e., minimal data unit (e.g., cache line), does the prefetcher operate? (vi) What *triggers* the prefetching phase, e.g., a previously-trained instruction or an accessed memory address, and can there be trigger collisions?

Identification and Characterization. Figure 4.4 (a) illustrates how we test for the presence of a stride prefetcher. In our base test, we train a cache-line-aligned stride pattern repetitively (in red) and measure if the CPU caches memory past our accesses (in blue). We then expand our base test to measure each of the characteristics individually. To reveal the direction, supported strides, and training size, we change the direction, offsets, and number of data accesses, respectively, and test whether prefetching occurs. The number of prefetched cache lines may depend on the selected stride and the number of preceding training accesses, so we analyze the number of prefetches for different strides and after different numbers of training steps. To test whether the prefetcher crosses the page boundary, we align the sequence of training accesses to the end of a page, as illustrated in Figure 4.4 (b). If we see any prefetching on the following page, the prefetcher can cross the page boundary. To test the granularity, we use known-good parameters for the stride and the number of training steps but add a small random offset between 0 and $(\textit{cache line size} - 1)$ to each loaded address. If the prefetcher detects this pattern and completes it, we conclude that the prefetcher operates on addresses at cache-line granularity.

To identify the trigger, we perform three different tests. The most likely triggers are the instruction address of a load instruction, the data address accessed, or fractions or combinations of these. To verify this assumption, we start with a negative test. We train the prefetcher on a first memory page using a first load instruction (I_1) and try to

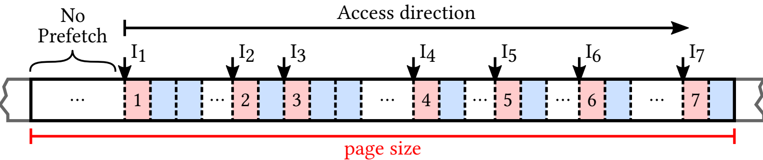

Figure 4.5: Stream prefetcher tests. Load instructions (I_x) are performed in one direction.

trigger it on a second memory page using a second load instruction (I_2), as illustrated in Figure 4.4 (c). If the prefetcher is triggered by a combination of instruction and memory addresses, we expect no prefetching, since we changed significant portions of both. Next, we test whether the prefetcher can successfully be trained by different load instructions loading from the same memory page (Figure 4.4 (d)). Lastly, we test whether the prefetcher can be trained by a single load instruction that is executed with different operands multiple times (for example in a loop) across different pages. This is similar to Figure 4.4 (c), but using I_1 throughout all accesses. If we observe prefetching for the instruction address test but not for the memory page test, we conclude that the instruction address is the only trigger. In the opposite case, the memory address is the trigger. If both tests show prefetching, the prefetcher uses either the instruction address or the memory address as a trigger. For prefetchers that use an instruction address as a trigger, we further test whether only a fraction of that address is considered. We re-use the experimental setup of Figure 4.4 (c) but align the addresses of I_1 and I_2 such that their c least-significant bits (LSBs) match. We increase c and stop as soon as we start seeing prefetching, indicating a collision. If a collision occurs, we conclude that the prefetcher only stores the c LSBs of the instruction address as a trigger.

4.2.2.2 Adjacent Cache Line

When accessing a cache line, the adjacent-cache-line prefetcher [70] automatically prefetches one of the neighboring lines.

Characteristics. The prefetcher can either be forward-fetching or block-fetching. A forward-fetching prefetcher always loads the line after the accessed one. In contrast, a block-fetching prefetcher divides memory into blocks of 2 cache lines, starting at an address that is a multiple of 2 cache line sizes. When a memory address is accessed, this prefetcher always loads the sibling cache line from the same block, i.e., preceding *or* subsequent cache line.

Identification and Characterization. First, we access a cache line at an address that is a multiple of 2 cache line sizes. If the following line is prefetched, we identified an adjacent-cache-line prefetcher. We further access a cache line that would be the second line in a 2-cache-line block. If this access prefetches one cache line in backward direction, the prefetcher is block-fetching. Otherwise, we characterize it as forward-fetching.

4.2.2.3 Stream

A stream prefetcher [69, 128] is designed to prefetch subsequent locations for an irregular access pattern that runs in a constant direction, using extrapolation. Although its behavior may appear similar to that of an adjacent-cache-line prefetcher, there are three

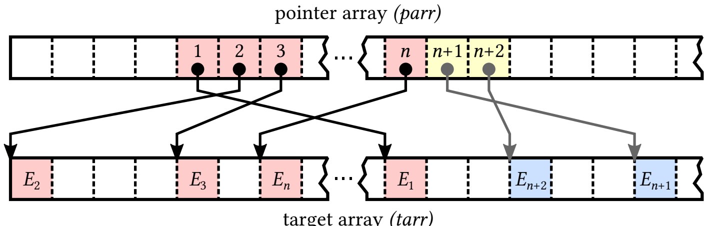

Figure 4.6: Pointer array test. Dashed boxes represent cache lines within a memory page. Red CLs are cached due to architectural loads during training, yellow and blue CLs are potential prefetch locations. Boxes are numbered by the order of access.

key differences. First, the stream prefetcher starts prefetching only when the direction is known, which requires at least two accesses. Second, it can prefetch multiple subsequent cache lines. Third, it prefetches cache lines in the same direction as the access, whereas an adjacent-cache-line prefetcher exhibits block or constant prefetching. Compared to the stride prefetcher, which is also directional and extrapolation based, the stream prefetcher does not need a regular pattern to initiate prefetching.

Identification. We develop an identification test to detect stream prefetchers by generalizing previously reverse-engineered properties of the Intel Kaby Lake CPU [128]. As illustrated in Figure 4.5, the test involves initiating a load instruction I_1 at a specific cache line (in this example, the 10th line). Thereafter the test continues to access cache lines in an irregular pattern using load instructions I_2, I_3, and I_4, on cache lines 16, 18, and 32, respectively, in that order. These irregular loads stop at the end of a page, and we measure the prefetched lines. Our test ensures that no two consecutive pairs of accesses have the same stride to prevent false positives from other types of prefetchers, such as the stride. Additionally, we use distinct load instructions for every load to prevent false prefetches from replay-based prefetchers that trigger based on the Program Counter (PC). In order to not restrict the test to just Intel's implementation [128], we perform more than four directional loads before observing the prefetches. We consider the test successful and identify a stream prefetcher when all prefetches run in the same direction, there is at least one instance of consecutive prefetches adjacent to load (e.g., cache lines 11 and 12), and no prefetching is observed before the first access (I_1).

4.2.2.4 Pointer Array

Arrays of pointers are common data structures that linearly store addresses of relevant data locations. The target addresses do not necessarily point to linear memory locations. Pointer array prefetchers [131, 163] analyze memory content to identify arrays of pointers and then prefetch their targets.

Characteristics. We consider the following prefetcher characteristics: (i) Does the prefetcher also detect *backward* iterations? (ii) How many *training accesses* are required? (iii) What is the maximum amount of data that is prefetched? (iv) How many *pointers* are prefetched? (v) Is the trigger instruction pointer dependent?

Identification and Characterization. Figure 4.6 shows how we test for the presence and features of a pointer array prefetcher. In our test, we set up an array of

pointers (*parr*), each referring to random entries within a sparse target array (*tarr*). First, we evict all entries of *parr* and *tarr* from the cache. Second, we ensure the address translation is cached for each page of *tarr* by accessing one cache line on each page. After that, we iterate over *parr* and dereference the first n training pointers. Then, we measure the time it takes to dereference pointer $n + 1$ and access a predetermined random entry of *tarr* that was not accessed during training. If a pointer array prefetcher is present, we observe a cache hit for pointer $n + 1$ but a cache miss for the random entry.

To unveil parameters of the prefetcher, we vary the number of training pointers n in *parr*, the offset of the measurement pointer from the last accessed *parr* pointer, and the number of bytes we measure past the start of a possibly prefetched *tarr* entry. To identify whether the prefetcher also activates on backward iteration, we start accessing and dereferencing the last entry of *parr* and decrement the index with each iteration. To determine whether the prefetching behavior depends on the memory location of the instruction accessing *parr* or dereferencing the pointer, we unroll the loop accessing and dereferencing pointers in *parr*. In an additional test, we spread these instructions across different memory pages and align them to different page offsets.

4.2.2.5 Pointer Chasing

In a chain of pointers, each pointer depends on its predecessor in the access sequence. The next address is stored where the previous pointer points to, as for example in a linked list. A pointer-chasing prefetcher [20, 35] loads subsequent pointers, reducing the stalling times when traversing such data structures. Note that related work [131] uses the term “pointer chase” to describe any behavior where data is directly used as an address. Our work, however, only uses the term in scenarios where the address of the next element of the data structure is unknown until it can be read from memory (e.g., a linked list). This effectively serializes memory accesses and prefetching, preventing parallelization.

Identification. As, to our knowledge, this type of prefetcher cannot be found in commodity hardware, we do not distinguish any characteristics for this type and only present an existence test. In our test, we set up a linked list where each node of the list resides in a different cache line. The order of the list is randomized to prevent other extrapolation-based or replay-based prefetchers from interfering. First, we evict all list entries from the cache. Next, we iterate over the pointer chase and dereference the first n training pointers. Then, we measure the access time to pointer *n+1* and a predetermined random entry that was not previously accessed. If the prefetcher is present, we detect a cache hit for pointer *n+1* but a cache miss for the random entry.

4.2.2.6 Spatial Memory Streaming (SMS)

Load instructions often access structured elements and exhibit repetitive access patterns in spatially-nearby memory relative to initial memory access. An SMS prefetcher [139] targets such use cases, which are common for a repetitive data structure such as socket buffers (`sk_buff`) in Linux. The prefetcher operates in two stages: training and prediction, which can be triggered by a particular load instruction (PC-triggered) or

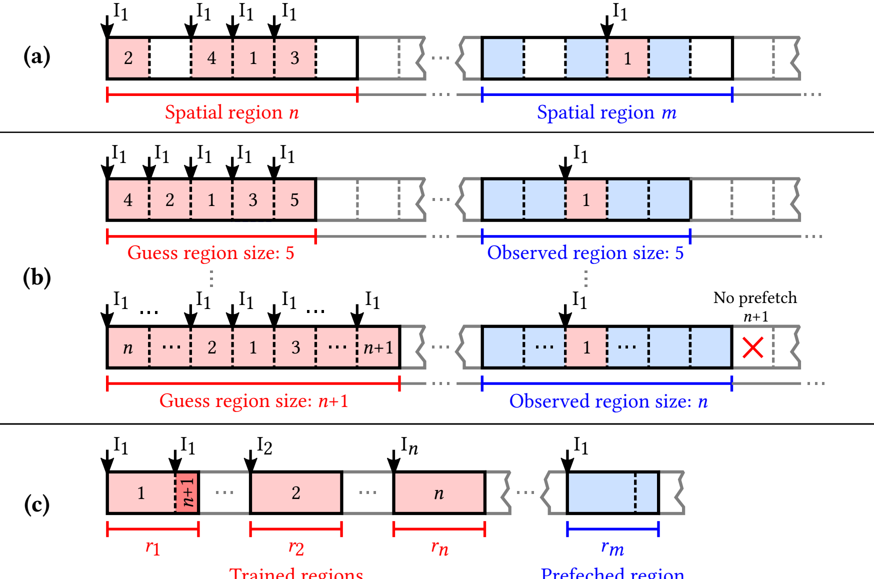

Figure 4.7: SMS prefetcher tests. Dashed boxes represent cache lines, bold boxes represent the spatial region. The red cross represents a miss for an expected prefetch.

memory location (Mem-triggered). For instance, in the training stage, a PC-triggered prefetcher learns all the memory accesses over a given load instruction in spatial memory. Eventually, when a different spatial region is accessed with the same load instruction, it initiates the prediction stage to prefetch learned memory access patterns—speculating that the load instruction will show similar access patterns in other regions.

Characteristics. We identify the following characteristics of an SMS prefetcher. (i) What triggers the prefetcher to start learning memory access patterns? (ii) What spatial boundary does the prefetcher use for learning memory accesses? (iii) Is there a difference in the prefetching behavior in forward or backward direction from the first load from a region? (iv) Can the training continue when another load instruction's address partially collides with the trigger instruction? (v) How many spatial regions can the prefetcher learn simultaneously in the training stage?

Identification and Characterization. We leverage the two stages of the SMS prefetchers to design our tests. As depicted in Figure 4.7, we demonstrate the PC-triggered SMS prefetcher tests. First, we induce a pattern in the prefetcher's training stage. Second, we trigger the prefetching stage to measure the prefetched cache lines in a different spatial region. Figure 4.7 (a) shows the base testcase where we induce a pattern by accessing spatially-nearby cache lines in one region, followed by trigger access in a new region (in red). We then compare the cache lines prefetched in the new spatial region (in blue) with the memory accesses in the trained region. If they are identical, the test confirms the presence of the SMS prefetcher. Extending the base test, we develop tests to identify the relevant factors for successful training and prediction stages. These factors comprise the address of the load instruction and the address offset of the first memory access. Figure 4.7 (b) illustrates the technique used to determine the spatial boundary for the prefetcher. We gradually increase the spatial size and stop

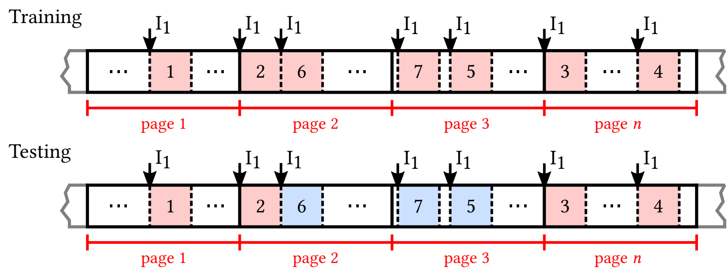

Figure 4.8: Region-unbounded prefetcher identification test

when the prefetched cache lines do not match the trained pattern. Knowing the spatial boundary, we also investigate the maximum size of the induced pattern for training when the consecutive memory accesses are in the same direction (forward or backward).

We further designed tests to identify the number of independent spatial patterns that can be learned simultaneously in the training stage. As illustrated in Figure 4.7 (c), we train the subject region r_1 with a few loads (I_1). We continue training $n-1$ additional regions r_2, r_3, until r_n. The training phase completes with one additional $(n+1)$ load I_1 in region r_1. For the load (I_1), we measure the prefetch corresponding to the last access $(n+1)$ in region r_m. If cache line $(n+1)$ is prefetched, the spatial region r_1 was not evicted from the training stage. The probability of eviction of the region r_1 increases with the number of simultaneously trained regions. However, due to unknown replacement policies, this test only provides an estimate.

4.2.2.7 Generalized Region-Unbounded Replay Prefetchers

As explained in Section 4.2.1, replay-based prefetchers aim to prefetch irregular but repetitive memory access patterns. The SMS prefetcher (Section 4.2.2.6) covers accesses that occur in spatially-nearby locations, but cannot cope with spatially-distant accesses. This is where region-unbounded replay prefetchers come into play. For example, when scanning a large unclustered database, although the database is logically sequential, the memory allocation is fragmented, making region-unbounded replay prefetchers more effective.

These prefetchers typically require more space to store the metadata necessary to detect unbounded and irregular patterns. To balance the accuracy and space trade-off, researchers have proposed innovative designs such as circular FIFO buffers, as implemented in the Global History buffer (GHB) [106], table-based as in Markov [76], storing a history in off-chip memory as in Temporal Memory Streaming (TMS) [155] and Spatio-Temporal Memory Streaming (STeMS) [138], and storing relative distance instead of complete addresses as in Delta Correlating Prediction Tables (DCPT) [40]. Despite these differences in design, the fundamental property to learn a sparse access pattern and replay it when detected remains common to all region-unbounded replay prefetchers.

Identification. As illustrated in Figure 4.8, we train the prefetcher by repeatedly accessing a sequence of a few memory locations at spatially-distant locations, where every memory access results in a cache miss. Subsequently, we flush the page from the

cache and access only the initial few locations of the sequence to test the prefetching behavior. If the prefetcher correctly prefetches the subsequent memory locations, a region-unbounded prefetcher is present.

Classification. There are many academic variations of region-unbounded replay prefetchers but no known instances on real CPUs. Existing tests in FetchBench, such as tests for the trigger or the number of prefetches, can serve as a reference for more detailed identification tests. For instance, a variant of the GHB prefetcher (GHB PC/AC) utilizes the PC to localize a cache miss and subsequently employs address correlation (AC) to detect the access pattern. Using the memory correlation primitive in FetchBench is sufficient to identify such a prefetcher, as it will prefetch for the same memory address but not for a different address.

4.3 Characterizing ARMv8 and x86_64 Prefetchers

In this section, we address RQ2.2 by using FetchBench to analyze 19 different CPUs for their hardware prefetching implementations.

4.3.1 Implementation

Our framework is implemented in C/C++ and Assembly, consisting of approximately 8,100 lines of code. It currently supports the ARMv8 and x86_64 architectures.

To implement the tests outlined in Section 4.2.2, we require three primitives on each platform: a precise timing source, a method to flush cache lines from the cache, and memory barriers. The flushing primitive allows training or triggering the prefetcher from a clean cache state. The timing source and the barrier enable us to determine whether a memory line is cached or not through a Flush+Reload cache side channel [161]. This allows us to observe prefetching behavior. On x86_64 CPUs, we use the `clflush` instruction to flush cache lines. On ARM-based platforms, we flush using the `DC CIVAC` instruction. On the Apple M1, we enable this feature through the privileged model-specific register `S3_0_C15_C4_X` first. To collect precise timestamps, we either call `clock_gettime`, use a counter thread, or use architecture-specific means like the `rdtscp` instruction on x86_64 or the *Performance Monitors Cycle Count Register (PMCCNTR)* on ARM (if available). We observe different noise characteristics for these methods across different platforms, so we choose the option that produces the least noise for each of them.

To ensure that the memory activities of the Flush+Reload side channel do not impact the results, we always probe one cache line at a time. More precisely, we run an experiment, probe the cache state of one memory location, repeat the experiment, probe a different location, and so on. This ensures that our probe accesses do not form patterns that could impact the prefetcher's behavior.

4.3.2 Experimental Setup

We run FetchBench on 19 different processors in total, comprising six ARMv8 SoCs, nine Intel x86_64 CPUs, and four AMD Ryzen CPUs. We provide a list of all testing environments in Table A.1 in the appendix, where we also assign them short IDs to

Table 4.1: Prefetcher identification and characterization results. Tests marked in bold print are existence tests.

(a) Stride Prefetcher (4.2.2.1)

Processor → / ↓ Characteristics	A53	A55	A72	A73	A76	M1i	M1f	i7SB, i5HW, XeSL, XeCL, i7CL	i3IL, i7TL, XeIL	i9ALp	R5Z, R5Z+	R7Z3, R9Z3+
Pos./neg. direction	●/●	●/●	●/●	●/●	●/●	●/●	●/●	●/●	●/●	●/●	●/●	●/●
Min./max. stride (B)	±64/ ±256	±64/ ±2048	±64/ ±4096	±64/ ±2048	±64/ ±8192	±128/ ±256	±128/ ±8192	±64/ ±1024	±64/ ±8192	±64/ ±16384	±64/ ±8192	±64/ > ±16384
Min./max. prefetches	3/5	1/28	1/16	1/31	1/18	8/20	8/16	1/2-5	1/2-6	1/8	1/5-7	5/15
Trigger	Mem	PC/Mem	PC/Mem	PC/Mem	PC/Mem	Mem	Mem	PC	PC	PC	PC	Mem
PC collision (bits)	N/A	—	12	—	15	N/A	N/A	8	10	10	12	N/A
Cross page boundary?	○	●	●	●	●	●	●	○	●	●	●	●
Strides < 1 CL?	○	○	○	○	○	●	●	●	○	●	○	○
Strides with random inner-CL offsets?	●	●	●	●	●	●	●	●	●	●	●	●

Not identified on i9ALe.

(b) Ptr. Array Prefetcher (4.2.2.4)

Processor → / ↓ Characteristics	M1f
Existence	●
Trigger	Mem
Pos./neg. direction	●/●
Max. prefetch size	256
Max. prefetch amount	16 ptrs.
No. training pointers	2

Not identified on all other processors.

(c) SMS Prefetcher (4.2.2.6)

Processor → / ↓ Characteristics	A72	A73	A76
Trigger	PC	PC	Mem
Region size (B)	1024	1024	1024
PC collision (bits)	12	—	—
Pos./neg. direction	12/9	16/11	12/9
No. of entries (est.)	5	9	10

Not identified on all other processors.

(d) Other Prefetchers

Processor → / ↓ Prefetcher	i7SB, i5HW, XeSL, XeCL, i7CL	i3IL, i7TL, XeIL	i9ALe
Adjacent CL (4.2.2.2)	●[B]	●[F]	●[F]
Stream (4.2.2.3)	●	●	○
Pointer chase (4.2.2.5)	○	○	○
Region-unbounded replay (4.2.2.7)	○	○	○

●[B] Block ●[F] Forward; None identified on all others.

refer to them throughout this chapter. Our selection of ARM-based platforms comprises five Cortex-A-series designs, ranging from the Cortex-A53 to the Cortex-A76, as well as the low-energy and performance cores of the Apple M1 Max SoC (dubbed *Icestorm* and *Firestorm*). The selected Intel CPUs comprise *Core* and *Xeon* models from the Sandy Bridge to the Alder Lake generation. We further test four AMD Ryzen CPUs from the Zen to the Zen 3+ microarchitecture. We run all our tests on each processor. We assume the analyst has root access to the systems under evaluation. We use a Linux distribution that is provided or recommended by the respective vendor for each of the ARM boards, as listed in Table A.1.

4.3.3 Experimental Results

According to our tests, all processors we examined implement at least one prefetcher design, most even two or more. We present the detailed results of our identification and characterization in Table 4.1 and summarize the most significant findings.

SMS Prefetchers. To the best of our knowledge, we are the first to identify and characterize a replay-based prefetcher in real-world hardware. As shown in Table 4.1 (c), our tests uncover previously unknown SMS prefetchers in the ARM Cortex-A72, -A73, and -A76 processors. All use a region size of 1 KiB. The prefetchers in `A72` and `A73` use the Program Counter (PC) as a trigger, i.e., they map the instruction address of a load instruction to a spatially-bounded memory access pattern. The `A72`'s prefetcher cannot distinguish trigger instruction addresses with 12 or more identical least-significant bits. This enables address collisions, causing the prefetcher to apply spatial access patterns learned in one region to another. As we show in Section 4.3.3, such collisions pose a security risk, as they leak memory access patterns across privilege domains. The SMS prefetcher on `A76` is memory-triggered, i.e., it stores an access pattern for a particular spatial region in memory. In addition, we discover that the SMS prefetchers on both `A72` and `A73` store the accessed memory locations within a region as distances relative to the first accessed memory line of a region. We suspect that the prefetcher stores the cache state of a limited number of memory lines around the initial access as a bit vector. Consequently, these prefetchers miss accesses at one end of the region when the initial access occurs close to a region boundary. Due to unknown replacement policies, we only estimate for the number of entries in the SMS prefetchers: 5 entries on the `A72`, 9 entries on the `A73`, and 10 entries on `A76`.

Stride Prefetchers. As revealed in Table 4.1 (a), the most commonly identified prefetcher in our test is the stride prefetcher, implemented in all of the tested processors except the Alder Lake efficiency core (`i9ALe`). The ARM-based CPUs in our test show considerably different stride-prefetch behavior. The prefetcher on `A53` is the least aggressive: The maximum stride is low (256 B), and at most 5 cache lines are fetched in advance. In contrast, the stride prefetcher on `A76` detects strides up to 8 KiB. More aggressive prefetching can also be observed on `A55` and `A73`, loading up to 28 and 31 cache lines upfront, respectively. While stride prefetching can be triggered by a matching load instruction address or an access to a nearby memory address for most Cortex-A-cores, the `A53` and the Apple M1 (`M1i`, `M1f`) rely only on memory addresses as a trigger. The stride prefetchers on the Intel CPUs from the Sandy Bridge to the Comet Lake

generations (`i7SB`, `i5HW`, `XeSL`, `XeCL`, `i7CL`) behave almost identically in our test, only differing in the maximum number of prefetched cache lines. We confirm the finding of [30] that this prefetcher uses only the 8 least-significant bits of a load instruction to reference a stride pattern internally [30]. In line with [160], we see prefetching for strides smaller than the cache line size [160]. In the Ice Lake and Tiger Lake generations (`i3IL`, `i7TL`, `XeIL`), the stride prefetcher is enhanced to detect larger strides of up to 8 KiB. The number of relevant instruction address bits increases from 8 to 10. Our Alder Lake CPU features a hybrid design. On its performance cores (`i9ALp`), strides up to 16 KiB are detected. We did not identify a stride prefetcher on the efficient cores (`i9ALe`). On the AMD CPUs based on the Zen and Zen+ microarchitectures (`R5Z`, `R5Z+`), we find a PC-triggered prefetcher that detects strides up to 8 KiB. In contrast, the prefetcher in the Zen3 and Zen3+ generations (`R7Z3`, `R9Z3+`) is memory-triggered and detects strides up to a size that exceeds our testing capabilities. All stride prefetchers in our test can be triggered by a pattern that forms a regular stride at cache-line granularity but has random inner-cache-line offsets.

Other Prefetchers. In our test, all Intel CPUs from Sandy Bridge to Comet Lake complement the stride prefetcher by a block-fetching adjacent-cache-line prefetcher and a stream prefetcher. Starting from Ice Lake, the adjacent-cache-line prefetcher is forward-fetching instead. The Alder Lake CPU has no stream prefetcher and an adjacent-cache-line prefetcher only on the efficient core.

Matching Sanchez Vicarte et al.'s results [131], we identify a pointer array prefetcher on the Firestorm cores of the Apple M1 CPU (`M1f`). We confirm that the trigger does not depend on the location of the instruction and examine additional features such as the size of prefetched entries. We detect prefetching starting with 2 training pointers in contrast to 3 as reported in prior work [131].

None of our CPUs feature a pointer-chasing prefetcher or a region-unbounded prefetcher. We suspect the primary reason for this is storage limitations. Region-unbounded prefetchers tend to require significant amounts of memory to store an access history, making their implementation expensive in practice. To verify that our test for region-unbounded prefetchers works in general, we execute it on the implementation of the DCPT (Delta Correlating Prediction Table) prefetcher [40] in the gem5 microarchitecture simulator [23]. The DCPT prefetcher uses a PC-indexed table pointing to a circular buffer of deltas. Our testcase successfully detects the region-unbounded prefetcher in this simulation.

Execution Time and Success Rates. The execution time of FetchBench mainly depends on the CPU performance and the identified prefetchers. In our tests, the runtime ranged from 27 minutes (`R9Z3+`) to 6.2 hours (`A53`). We illustrate the success rates of our experiments in Section A.1.2 in the appendix.

4.4 Exploiting Prefetchers

In this section, we address RQ2.3 by investigating the security properties of a prefetcher that we identified in hardware for the first time: The SMS prefetcher. We know from FetchBench's results (see Table 4.1 (c)) that the implementation of this prefetcher on the ARM Cortex-A72 is PC-triggered. Further, trigger collisions may occur when the

addresses of two load instructions share the 12 least-significant bits. This combination of characteristics is problematic, as it allows us to train a prefetcher in one context and observe its prefetching behavior in another. We present two case studies to illustrate this finding. First, we show that we can exploit the prefetcher's characteristics to leak secret-dependent memory accesses from one process to another. Second, we demonstrate that the prefetcher does not properly separate patterns from normal world and secure world, allowing for side-channel leakage from TrustZone.

4.4.1 Case Study: AES Attack

We exploit the SMS prefetcher on the ARM Cortex-A72 to leak parts of an AES key from one user-space process to another. Lookup-table-based AES implementations have been target of side-channel attacks in the past [44, 144]. Some attempts relied on tables being shared between attacker and victim process, enabling techniques such as Flush+Reload and Flush+Flush [44]. Approaches not based on shared memory faced the non-trivial challenge of profiling tables without knowing their addresses and cache-set mappings [144].

Our attack targets an implementation where lookup tables are *not* shared, making Flush+Flush and Flush+Reload infeasible. In contrast to prior approaches, we avoid the need to learn the table's memory addresses by exploiting the prefetcher. The prefetcher uses partial *instruction* addresses as a trigger, making it susceptible to address collisions—an insight gained from FetchBench. This allows us to leak memory access patterns in a targeted way based on the addresses of load instructions that access the tables. Prior attacks considered the prefetcher rather a source of noise [144], while we use it as a source of leakage.

We further address the problem of extracting a secret-dependent prefetcher state before it is overwritten or evicted by subsequent memory loads. While prior work modified the victim code to work around this problem [137], we synchronize the attacker process with the victim and interrupt it soon after the first AES key addition. We implement this synchronization primitive by combining Flush+Reload on code [161] with inter-processor interrupts [83].

System and Attacker Model. We perform the attack on our `A72` system, a Raspberry Pi 4 with four ARM Cortex-A72 cores. We attack the default AES-128 implementation of Mbed TLS 3.3.0 (released December 2022), which is based on T-tables. We assume a chosen-plaintext scenario: The victim process encrypts an attacker-specified plaintext using a fixed 128-bit key targeted by the attacker.

The attacker can execute native code in user space on the same CPU core as the victim and uses the PMCCNTR register as a timing source. We assume that the attacker can execute the victim program repeatedly, each time providing an arbitrary plaintext. Both the attacker and the victim process have access to the same (shared) Mbed TLS instance in memory. However, we emphasize that only the library code is shared in memory, not the T-tables. The tables are generated in victim memory during library initialization. The attacker can thus apply Flush+Reload only to code (to synchronize with the victim process) but *not* to the T-tables (as prior attacks did to infer access patterns; we exploit the prefetcher instead).

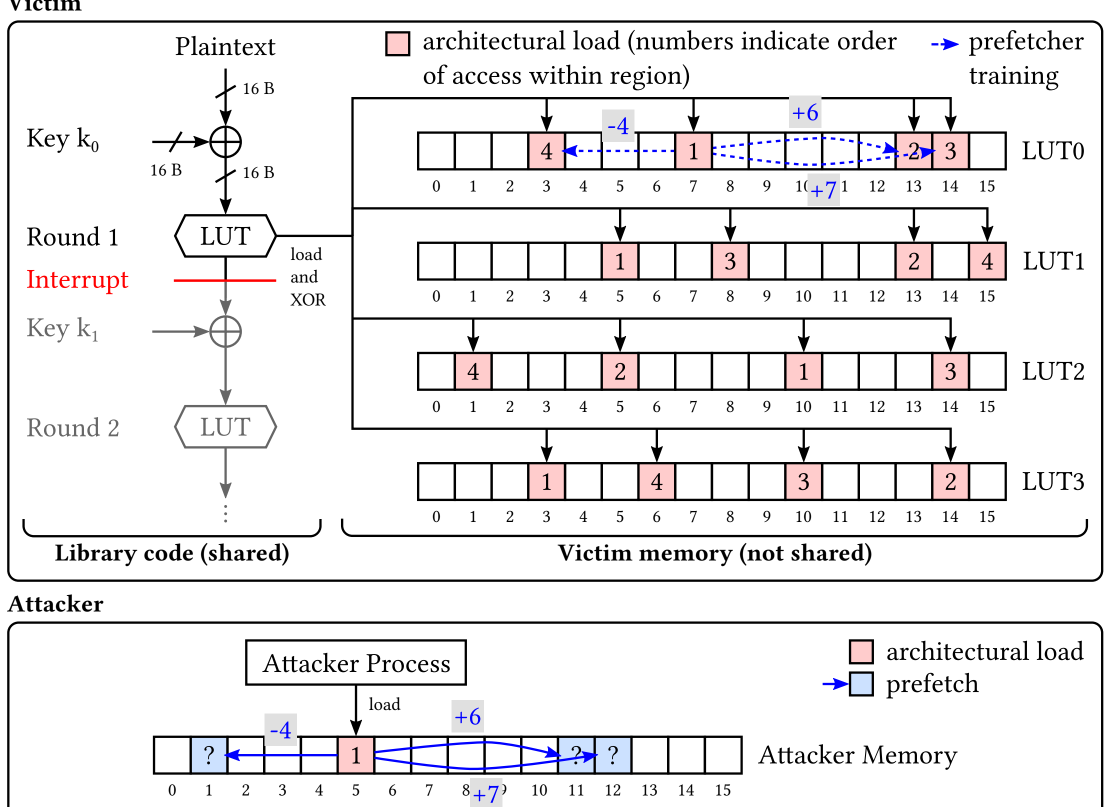

Figure 4.9: Loads and prefetches in victim/attacker process

Victim Program Procedure. The victim process receives a plaintext as a command line parameter and encrypts it using the Mbed-TLS library. During library initialization, four lookup tables of 256 4-byte integers each are generated. The generated tables are local to the victim process and thus inaccessible to the attacker. The default table size of 1 KiB (or 16 cache lines) matches the region size of the A72's prefetcher. As illustrated in Figure 4.9, the attacker-provided plaintext bytes are XORed with the AES key k_0 in the first encryption round. The 16 resulting bytes are secret-dependent and are used as indices to load elements from the lookup tables—the source for the prefetcher-based leak.

Cache-Line Granularity. Each cache line of the lookup tables contains 16 table elements and we expect the same prefetch behavior regardless of which of those elements is accessed. Thus, our attack cannot distinguish offsets within a cache line and can thus only infer the upper 4 bits of each access location and key byte. This limits us to leaking only half of the AES key. We emphasize however that knowing 64 key bits reduces the brute-force effort significantly, as an attacker knows which key bits are missing. To illustrate this, we refer to past approaches to brute-force block cipher keys. *Deep Crack*, a machine built in 1998 to brute-force a DES key, tried 92 billion keys per second [34]. At this pace, brute-forcing a full AES key would take $5.9 \cdot 10^{19}$ years on average, while 64 key bits could be found in only 3.2 years on average. As brute-forcing

can be parallelized, for example in the cloud, a modern solution could be even faster.

Prefetcher Activation. When a lookup table is first accessed in the victim process, the prefetcher is activated, sets up a new region, and associates it with the address of the initial load instruction. For all subsequent loads from the same lookup table, the prefetcher stores the distances (in cache lines) between the initial memory address and the memory addresses of the three following accesses. For instance, in Figure 4.9, the initial load from LUT0 targets cache line 7, and the next three loads target lines 13, 14, and 3. The prefetcher stores the secret-dependent distances relative to the first access, i.e., $13 - 7 = 6$, $14 - 7 = 7$, and $3 - 7 = -4$.

Attack Procedure. Our attack has 3 steps. First, we determine the distances between the initial access and the three later accesses to a lookup table. Second, we find the offset of the initial access from the beginning of the lookup table. Finally, we decide which of the three leaking distances is influenced by which key byte. Since the procedure is the same for all tables, we describe it for only one.

Initial leakage. The attacker provides an arbitrary plaintext to the victim process, interrupts it after the first AES round, and extracts the prefetcher's state. We explain the interrupt mechanism in the next section. To extract the state, the attacker executes a load instruction that is located at a colliding address to the first load in the library accessing the table. This instruction address is public and can be extracted from the shared library binary. The address is also unaffected from ASLR, as the relevant lower address bits are usually not randomized. Executing the colliding load instruction activates the prefetcher in the attacker's memory space, as shown in Figure 4.9. The prefetcher erroneously associates the load with earlier memory activity in the victim process and prefetches three more addresses according to the previously recorded distances, starting from the initial load location in the attacker's memory. The attacker recovers the distances from the cache state of their own memory using a Flush+Flush cache side channel. However, the three distances are insufficient to recover the corresponding key bits. Additionally, (i) the offset of the initial load from the beginning of the table, and (ii) the order of the three later accesses are required.

Leaking the initial offset. To identify the offset of the initial access from the beginning of the table, the attacker needs to determine the 4 most-significant bits (MSBs) of the byte that influences the initial load from the table. To this end, the attacker flips one of the 4 MSBs of the corresponding plaintext byte and repeats the experiment. As a result, all three leaking distances move by a constant offset in either a positive or negative direction. The direction reveals the bit. The attacker repeats this process for each of the 4 MSBs.

Leaking the remaining offsets. The attacker already knows the three distances between the initial and the later loads but does not know which input bytes influence which of the distances. To find this mapping, the attacker flips a bit in one of the plaintext bytes that control the three later table accesses and repeats the experiment. As a result, only one of the three distances changes. The attacker maps this distance to the mutated plaintext byte and uses this knowledge to compute the four key bits for the corresponding key byte.

After this process is completed for all four lookup tables, the attacker successfully leaked 64 bits of the AES-128 key.

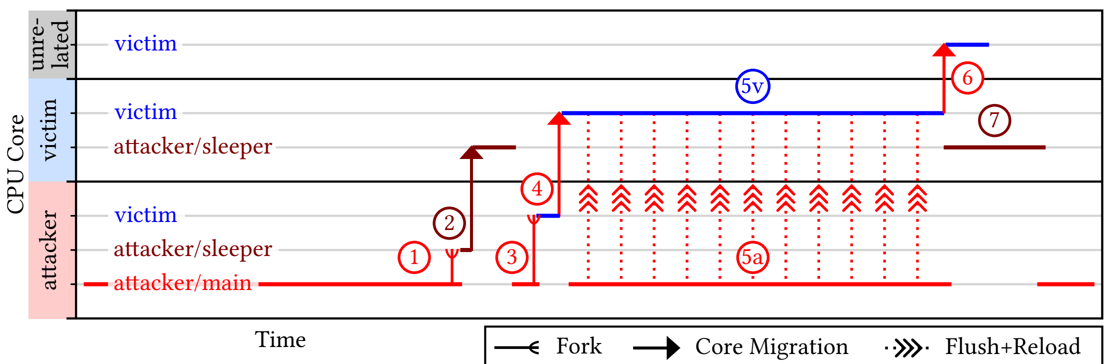

Figure 4.10: Synchronization of attacker and victim processes

Synchronization. The attacker faces two challenges when synchronizing their process with the victim. First, they need to interrupt the victim process when the prefetcher's state is secret-dependent, as later memory accesses could alter that state. Second, they have to schedule attacker code that extracts the prefetcher's state as soon as possible after the interrupt on the victim's CPU core. We approach the challenge of synchronizing attacker and victim with a Flush+Reload attack on the library code [161] to identify the right moment for an interrupt and an inter-processor interrupt [83] to schedule attacker code.

The detailed procedure we use is illustrated in Figure 4.10. We are interested in the prefetcher's state as soon as possible after the first round of accesses to the AES lookup tables is completed (highlighted in red in Figure 4.9). After starting the main attacker process, we fork a child process (1), which we refer to as the *sleeper* process, and move it to the core where we later run the victim on. We block the sleeper process, for instance by making it wait for a semaphore controlled by the main attacker process. Next, the main attacker process starts the victim process (3) and moves it to the victim core (4). While the victim runs and performs the encryption (5v), the main attacker process performs a Flush+Reload attack on the shared library code (5a). More precisely, we select the address of a function that is called just before the encryption (`mbedtls_aes_setkey_enc`) as a trigger. We flush this address in a loop and reload it while measuring the memory latency. As soon as we notice a low latency, we know that the victim called the function. At this point, the main attacker process wakes the blocking sleeper process through the semaphore and triggers an inter-processor interrupt, e.g., by moving the victim process to another core (6) or calling the `membarrier` system call [83]. The scheduler interrupts the victim process on the victim core and schedules the sleeper process (7). The sleeper can then perform a memory load from an instruction at a colliding code address and inspect the cache state of its own memory to recover the prefetcher's state.

The synchronization may interrupt the victim process slightly too early or too late. However, we can easily filter the recovered prefetcher states based on the number of accesses: If we interrupt the victim process at the right moment, we expect to leak three distances. We ignore all other samples for our attack.

Evaluation. We repeated the attack with 5 random keys on a Raspberry Pi 4 and

successfully recovered the 4 MSBs of 12.4 AES key bytes on average (maximum: 16, minimum: 8). The average runtime of our attack is 30 hours. We use 256 different plaintexts for each execution run and repeat each plaintext encryption between 10,000 and 15,000 times. Our synchronization mechanism provides us with the expected cache state in 0.19% of these cases. While this hit rate may seem low, almost all unexpected samples are easy to detect and filter out based on the number of cache hits, as we primarily expect samples containing three distances.

Edge Cases. The attack as described above assumes that the T-tables are 1 KiB-aligned (i.e., aligned to prefetch regions) in memory. The table alignment is determined at compile time and can change between compiler versions. If the tables are misaligned and span across two regions each, the attack can be adapted by tracking accesses in both regions or by mutating the plaintexts such that all accesses are moved into one of the regions. Similarly, if two memory accesses hit the same memory table element, less than three distances are leaked. This problem can be solved by mutating the plaintext until this is no longer the case.

4.4.2 Case Study: ARM TrustZone Covert Channel

Our second case study demonstrates a covert channel based on the SMS prefetcher characteristics uncovered by FetchBench. Our prefetcher-based covert channel uses the prefetcher's state containing the access pattern—as encoded by the sender and retrieved by the receiver—to implicitly send information between processes. We demonstrate that such covert channels are possible even if the two processes run in vastly different privilege domains. To this end, we leverage FetchBench's characterization of the SMS prefetcher implementation on ARM Cortex-A72. Using ARM TrustZone [17], we create a covert channel between a user space application in a non-secure world and a trusted app in a secure world. The covert channel thereby (a) demonstrates missing state separation between secure and non-secure world in a prefetcher, (b) approximates the upper bound of the data transmission rate.

The general idea of the covert channel is as follows. Given a spatial region comprising N cache lines, the sender uses one of these lines as a trigger and encodes the bits to send into the state of the remaining $N - 1$ cache lines. The receiver uses a colliding load instruction to trigger the prefetcher. Immediately thereafter, the receiver derives the transferred bits from the prefetcher's prediction.

Implementation. To run our attack on a fully TrustZone-enabled platform, we prototype our attack on a Rock 4 SE board instead of the Raspberry Pi 4 (`A72`). This board is based on a Rockchip RK3399-T SoC, which is TrustZone-supported [127] and contains four Cortex-A53 and two Cortex-A72 processor cores. We only use the A72 cores for the covert channel. We run Linux in a non-secure world and OP-TEE, an implementation of ARM TrustZone, in a secure world.

The concrete implementation of this covert channel requires a detailed understanding of the underlying prefetcher internals. We make use of the characterization obtained by FetchBench as reported in Section 4.2.2.6: The spatial region size is 16 cache lines. The prefetcher is PC-triggered and uses the 12 least-significant bits of the PC to map it to a region pattern. This implies that the sender and receiver applications only need to

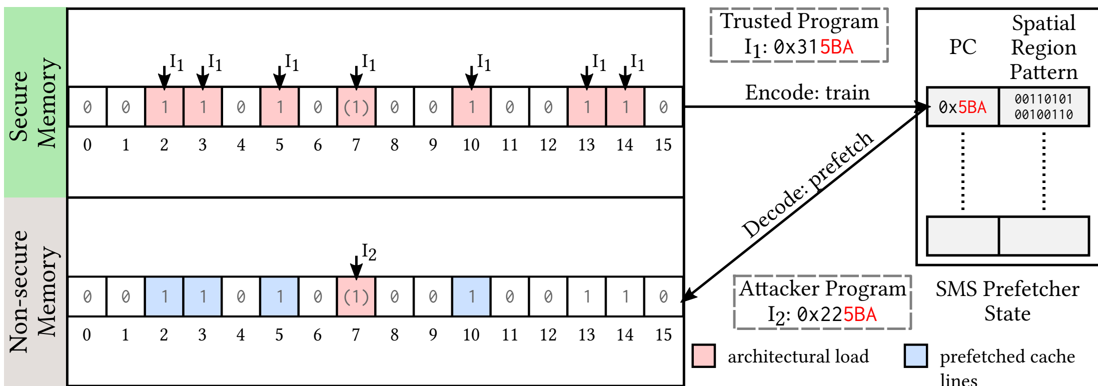

Figure 4.11: Covert channel based on the SMS prefetcher. Top: The sender encodes a bit vector in the SMS region (in red). Bottom: The receiver triggers a PC-colliding load to reveal this bit vector via the prefetcher regions (in blue).

align 12 bits of their load instruction addresses to map them to the same pattern in the prefetcher's state. FetchBench also identifies a limitation on the A72 CPU: While a spatial region has a size of 16 cache lines, the prefetcher is unable to capture the state of some of those lines in some situations. More precisely, the prefetcher can only record the state of up to 9 cache lines before and 12 cache lines after the initial (trigger) access to a region. Hence we choose the trigger location as the middle of the region, allowing us to make use of all cache lines of a region. Furthermore, FetchBench finds that at least five region patterns can be stored simultaneously in the SMS prefetcher. Thus, we optimize the data rate by training *multiple* region patterns in each iteration; we then empirically validated that training four region patterns in parallel hits the sweet spot between low error rate and high throughput.

Figure 4.11 illustrates the covert channel in detail. First, the attacker aligns the user-space code such that the least-significant 12 bits of the receiving load instruction collide with those of the sending load instruction in the secure world. Second, the sender encodes data into the prefetcher's state as discussed before—we use the cache state of each cache line within a region to encode one bit of information. Every accessed cache line (in red) corresponds to a set bit, while cache lines corresponding to unset bits are not accessed. Because one cache line (here: the 7th) serves as the trigger which must always be accessed, we cannot use this cache line to encode a data bit. Consequently, we encode 15 bits of data per region pattern into the SMS prefetcher's state. Finally, the receiver triggers prefetching (in blue) in user space by accessing the 7th cache line in a region in the receiver's memory space using a colliding load instruction. By simultaneously training four region patterns in this way, we can transfer 60 bits per iteration.

Evaluation. To evaluate the performance of the covert channel, we transfer 1 MiB of test data from secure to non-secure memory. The accuracy is computed as the percentage of error bits while transferring the test data, and the data rate is computed as the total time required for transferring the test data. After decoding the data, we observed a bit error rate of 5.02% and a data rate of 1245 Bytes/s.

Applicability to Other Prefetchers. Stride- and stream-based covert channels

encode data in strides. Existing approaches [30] use two different stride widths to represent 0 or 1; i.e., 1-bit information per prefetcher entry. In comparison, the region-based prefetcher allowed us to encode more bits and achieve a significantly higher data rate, ten times higher than existing approaches. Furthermore, using FetchBench's insights into prefetcher characteristics allowed us to minimize the error rate and improve the data rate. We believe that our new insights into various prefetcher types can also improve related covert channels. For example, one could use FetchBench to characterize a stride prefetcher and encode more bits per pattern by also considering larger stride lengths.

4.5 Related Work on Prefetcher Reverse Engineering and Exploitation

4.5.1 Prefetcher Reverse Engineering

Most prior work focuses on specific prefetchers in x86 processors. Rohan, Panda, and Agarwal [128] reverse-engineer the Intel Kaby Lake stream prefetcher and find the stream table size and the direction of its entries, the trigger condition of the prefetcher, and if the prefetcher is shared between different cores of the processor. Chen, Pei, and Carlson [30] reverse-engineer and characterize the Intel IP-stride prefetcher in the Haswell and Coffee Lake microarchitectures, while Xiao, Tang, and Guilley [160] investigate the behavior of the Intel Comet Lake IP-stride prefetcher for small strides. Didier et al. [33] characterize the L2 stream prefetcher and the adjacent-cache-line prefetcher on the Intel Whiskey and Coffee Lake microarchitectures. Hetterich et al. [54] present *StrideRE*, a reverse engineering tool that characterizes stride prefetchers in depth, and apply it to AMD and Intel CPUs.

Only few publications investigate prefetchers on ARM architectures. Sanchez Vicarte et al. [131] and Chen et al. [28] characterize the pointer array prefetcher in the Apple M1 and later processors.

While the aforementioned works focus on data prefetching, Zhang et al. [166] reverse-engineer the instruction prefetcher on a number of Intel CPUs and find that it follows branch predictions from the branch target buffer (BTB).

Another related prediction mechanism is load value prediction (LVP), which predicts the contents of memory directly and uses them during speculative execution. Sanchez Vicarte et al. [132] and Holtryd, Manivannan, and Stenström [56] discuss the perils of LVP theoretically. Kim et al. [77] are the first to identify, reverse-engineer, and exploit this mechanism in a real processor. They find that the Apple M3 and related models perform LVP on constant load values of up to 4 bytes.

In contrast to prior work, our prefetcher characterization approach covers a multitude of data prefetcher designs and is applicable to x86 and ARM processors. We are the first to identify and characterize a replay-based prefetcher design, in particular, the SMS prefetcher on the ARM Cortex-A72 architecture. Our approach enables us to better evaluate the security of a high number of CPUs across the x86 and ARM architectures.

4.5.2 Prefetcher Exploitation

Prefetchers have been exploited to leak data from program memory either through side channels [28, 30, 131, 137, 160] or by constructing covert channels [30, 32, 128]. We discuss prior-work prefetcher side-channel attacks in detail in Section 5.3.3, discuss prior-work defense approaches in Section 5.7, and introduce our own countermeasure in Section 5.4.

Compared to these works, we are the first to exploit a replay-based data prefetcher design. Similar to AfterImage [30], we cross the boundary between the operating system and the trusted execution environment and show that the prefetcher's internal data structure can be exploited to leak information across these domains.

4.6 Discussion of FetchBench

Limitations. Our approach enables us to identify and characterize prefetcher designs we implemented testcases for, but not prefetchers that operate or are triggered in a different way. However, both our taxonomy and framework are extensible and can be adapted to support further contemporary or future designs.

In addition to data prefetchers, modern CPUs also utilize instruction prefetchers that allow predictive fetching of instructions before they are requested for execution. Data and instruction prefetchers share a common goal but are fundamentally different in the way they operate internally. For example, instruction prefetchers can be coupled with the branch predictor [166]. Characterizing and exploiting these prefetchers is an orthogonal research problem [39].

Countermeasures. A straightforward approach to mitigate prefetcher-based attacks is to permanently disable prefetching, which is possible on most recent processors through model-specific registers. However, this might impact performance. Several other countermeasures provide trade-offs between security, performance overhead, and implementation effort. For example, prefetching could be disabled selectively for security-relevant code. While causing less overhead than disabling prefetching completely, this countermeasure requires modifications to both the kernel and compilers, which need to expose and utilize this functionality, respectively. CPU vendors could adapt the following countermeasures: (i) partitioning the prefetcher state across processes and/or security domains; (ii) implementing a special load instruction that does not impact the prefetcher's state, which can be used in security-relevant code; (iii) implementing an instruction that flushes the prefetcher's state, which can be triggered upon context/domain switches; (iv) storing complete instruction addresses to correlate load instructions based on the program counter. Most of these countermeasures also induce non-negligible performance overhead, require kernel modifications, and may be incomplete. A software-based countermeasure is *constant-time programming*, i.e., writing code with secret-independent resource usage. More precisely, developers should ensure that runtime, code access patterns, and data access patterns do not depend on secret values [64]. This approach could mitigate our attack on the AES implementation.

4.7 Conclusion on FetchBench

Due to their proprietary nature, the types and characteristics of specific prefetcher implementations are not known to the public for many common CPU models. This hinders systematic assessment of security threats posed by prefetchers. In this chapter, we addressed the following question: *How can we systematically identify and characterize hardware prefetchers and their security implications in real-world processors?* We proposed a taxonomy for prefetcher designs that differentiates based on prediction strategy and information source, leading to seven different types of prefetchers that can be expected to be in use in modern CPUs. For each type, we provided a detailed description and cross-platform testcases, which are used by our FetchBench framework to identify and characterize implemented prefetchers. We implemented and used this framework to characterize 19 different ARM and x86_64 CPUs, and we found that all differed in the specific behavior of their prefetchers. Our analysis and framework also allowed identifying and characterizing the previously unknown replay-based SMS prefetcher on the ARM Cortex-A72 CPU. To illustrate how accurate knowledge of prefetcher characteristics impacts prefetcher security, we demonstrated two side-channel attacks using the Cortex-A72's SMS prefetcher to leak secret information, even from the secure TrustZone into normal world.

5

PreFence: A Fine-Grained and Scheduling-Aware Defense Against Prefetching-Based Attacks

5.1 Introduction to PreFence

We have shown in Section 4.4.1 that prefetching-based side channels can be exploited to compromise cryptographic secrets, such as AES keys. Similarly, prior work has reported on prefetching-based attacks on Diffie-Hellman keys [28, 137] and RSA private keys [28, 30]. Covert channels were presented in prior work [30, 32, 128] and in Section 4.4.2, in some cases bypassing security guarantees of process isolation and trusted execution. No defense has been presented to date that protects against such attacks effectively and efficiently. While most platforms allow hardware prefetchers to be completely disabled, this drastic measure has a significant performance impact, as non-security-relevant code can no longer benefit from prefetching.

In this chapter, we address the high-level research question (RQ3): *How can we systematically defend against prefetching-based attacks?* We propose *PreFence*, our novel countermeasure that allows user-space code to defend itself against the perils of hardware prefetching on affected CPUs with minimal overhead. With PreFence, processes gain fine-grained control over the prefetcher. Processes indicate to the scheduler when security-relevant operations are executed, so that the kernel can disable prefetchers temporarily. This signaling requires only minimal code changes and can be integrated at different levels. For example, when integrated at library level, all applications that use the respective library benefit from PreFence automatically. Alternatively, signaling can be integrated into application code or even be performed manually on a per-process basis by the user. Our solution has negligible performance overhead on non-security-critical workloads. For an application containing security-critical code sections, we find that our countermeasure allows the application to run up to 15.8% (Intel) and 7.2% (ARM) faster on average compared to disabling prefetching permanently.

We systematically analyze existing side-channel attacks that exploit hardware prefetching, identify their differences and similarities, and find suitable entry points for defenses. Based on these insights, we design, implement, and evaluate PreFence, an enhancement to the process scheduler to make it security-aware. Our approach enables processes to ask the kernel to disable the prefetcher temporarily during security-critical operations. We also address the challenges arising from process scheduling and related to multi-core processing and Simultaneous Multithreading (SMT). As a software-based mitigation, PreFence leverages the widespread support of processors to control the prefetcher at runtime and does not require further hardware adaptations. We focus on defending processes against side-channel attacks based on hardware prefetching, as those attacks directly expose secrets from the victim's context; we exclude covert channels, as those merely transfer information that is already accessible to the attacker.

Contributions. We make the following contributions:

- We systematize existing prefetching-based side-channel attacks and investigate their similarities. We identify five main stages and map each attack's flow to those stages, demonstrating that there are core components required by all attacks.

- We design *PreFence*, which enables the mitigation of prefetching-based attacks in a fine-grained manner. Our solution enhances the scheduler to be security-aware, managing prefetcher state for all processes.

- We implement and evaluate PreFence for an x86_64 and an ARM processor. We demonstrate that PreFence prevents two prior-work attacks, that its performance impact is negligible for non-security-critical workloads, and that a security-critical workload performs significantly better with PreFence compared to prefetching being permanently disabled.

5.2 Defending Against Prefetching-Based Attacks

In this section, we introduce our system and attacker model and pose detailed research questions for this chapter.

5.2.1 System and Attacker Model

We assume a system with a processor that performs hardware prefetching. We further assume that the defender and attacker know the type of the deployed prefetcher as well as its security-relevant characteristics (e.g., obtained by the attacker with a copy of the target hardware and a suitable testbench, such as our FetchBench framework from Chapter 4). The defender is able to modify the software running on the CPU, including the operating system kernel. The hardware provides an interface to control (i.e., enable or disable) the prefetcher from the kernel. The attacker is able to execute arbitrary code in user space. In Section 5.6.3, we extend the attacker model to attackers at higher privilege levels.

5.2.2 Research Questions and Challenges

We break down the high-level research question of this chapter into the following questions:

RQ3.1: What kind of side-channel vulnerabilities in prefetchers have been exploited in prior work? Is there a core set of vulnerabilities that are critical for all known attacks?

RQ3.2: Is there a software-only countermeasure to mitigate all known prefetching-based side-channel vulnerabilities effectively and efficiently?

Challenges. To answer these research questions, we need to overcome the following challenges:

- Prefetcher side channels have been exploited in different settings in prior work. We need to work out similarities and differences between those approaches to be able to identify common patterns.

- Countermeasures require trustworthy arguments on why they can be expected to prevent current and future attacks. So far, mitigations have been discussed briefly and in the context of specific attacks as part of offensive papers, but never considering the class of prefetching-based side channels as a whole.

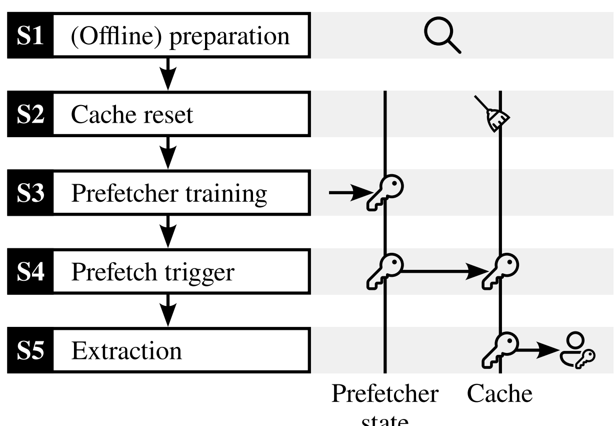

Figure 5.1: Stages of prefetching-based side channels

- Any countermeasure will cause a performance impact, which needs to be quantified and minimized.

Proposed Approach. To overcome these challenges, we pursue the following approach. First, we systematize known prefetching-based side-channel attacks and the vulnerabilities they exploit. We identify a minimal set of vulnerabilities that are required for any attack to work. Based on that, we design and implement a solution to prevent exploitation of this minimal set of vulnerabilities, leading to a countermeasure effective against all prefetcher side-channel attacks from user space. We then evaluate the implemented solution on real-world hardware.

5.3 Systematization of Attacks

To protect against prefetching-based side channels, we first need to understand the attack vectors in detail. To this end, we systematize all attacks exploiting hardware-based data prefetchers that we could find in academic literature (13 attacks across 7 papers). Inspired by prior works on mitigating other microarchitectural side channels [25, 102, 135], we break down prefetching-based attacks into stages (Section 5.3.1). We further define the scopes in which those attacks operate (Section 5.3.2). Finally, we visualize our systematization by plotting the attack sequences, deduce similarities and differences, and expose where software-based mitigations can effectively be applied (Section 5.3.3).

5.3.1 Stages of Prefetching-Based Attacks

We observe that prefetching side channels can be split into the following five stages, as illustrated in Figure 5.1:

- **S1: (Offline) Preparation.** For some attacks, an attacker must take extra steps before the actual attack, such as reverse engineering or setting up data structures.
- **S2: Cache Reset.** To start from a clean state, some attacks include actions to reset the initial cache state.

- **S3: Prefetcher Training.** Most prefetchers keep an internal state that governs their behavior. The prefetcher continuously observes memory activity to identify patterns in the addresses or data being accessed. When a new pattern is detected or existing patterns are altered or interrupted, the prefetcher's internal state changes. Thus, the prefetcher's future behavior changes as well. We refer to this state change as *prefetcher training.* From the attacker's perspective, this step can be seen as *encoding information into the prefetcher's state.* In the context of an attack, this information is secret-dependent.

- **S4: Prefetch Trigger.** Upon a trigger event, such as another memory access that matches certain criteria, a prefetcher may bring additional memory lines into the cache. Those memory lines are selected based on the prefetcher's internal state. We refer to this process as *prefetch trigger.* From the attacker's perspective, this step can be seen as *extracting information from the prefetcher's state into the cache.*

- **S5: Extraction.** The cache state is inspected to extract information about the prefetcher's internal state. This step can be seen as *extracting information from the cache into the attacker's context.*

Attacks may skip some of these stages, as we show in Section 5.3.3.

5.3.2 Scopes of Prefetching-Based Attacks

Prefetch attacks often operate across privilege domains. In this respect, we classify attacks based on the following scopes:

- **SP: Same-process.** Leaking information within the same process.

- **CP: Cross-process.** Leaking information from one user-space process to another.

- **KU: Kernel to user.** Leaking information from kernel space to user space.

- **TO: TEE to OS.** Leaking information from a trusted execution environment (TEE), such as Intel SGX or ARM TrustZone, to the (untrusted) operating system.

If victim and attacker operate in different contexts (e.g., in different user-space processes), the attacker has to ensure that the prefetcher keeps its state across the context switch. Especially when leaking between two user-space processes, the attacker faces the problem that the scheduler manages the process runtime, making it non-trivial to interrupt the victim process at a specific point in time (when the prefetcher's state is secret-dependent) and schedule the attacker process (to extract the state). Some attacks assume shared memory, either data memory or shared libraries, between both processes to address this issue [30]; others use additional side channels for synchronization (such as our own attack in Section 4.4.1).

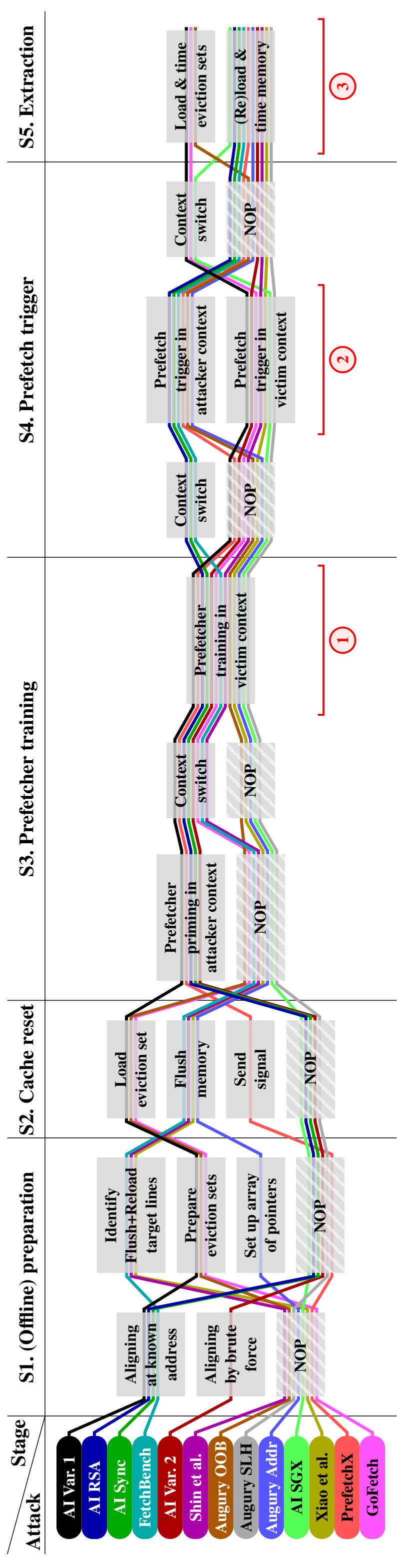

Figure 5.2: Overview of the sequence of activities in prefetching-based attacks. Core activities required by all are highlighted in red.

Table 5.1: Prefetching-based attacks in prior work. Prefetchers marked with † are DMPs, all others are address-based. Scopes are explained in Section 5.3.2.

Attack	Prefetcher	Scope	Target
Shin et al. [137]	Intel IP stride	CP	OpenSSL ECDH
Augury [131] OOB	Apple DMP†	SP	Custom
Augury [131] SLH	Apple DMP†	SP	Custom
Augury [131] Addr.	Apple DMP†	SP	—
AfterImage [30] Var. 1	Intel IP stride	SP/CP	Custom
AfterImage [30] Var. 2	Intel IP stride	KU	Custom
AfterImage [30] SGX	Intel IP stride	TO	Custom
AfterImage [30] RSA	Intel IP stride	SP	MbedTLS RSA
AfterImage [30] Sync	Intel IP stride	CP	OpenSSL RSA
Xiao et al. [160]	Intel IP stride	SP	AES
FetchBench [P2] AES	ARM SMS	CP	MbedTLS AES
PrefetchX [29]	Intel XPT	CP	MbedTLS RSA, GnuPG RSA
GoFetch [28]	Apple DMP†	CP	Go RSA, OpenSSL DHKE, CRYSTALS

5.3.3 Prefetch Attack Systematization

We now systematize 13 attacks from prior work to answer RQ3.1. We list these attacks in Table 5.1 and denote the prefetcher types they exploit, the scope the attacks operate in, and the attack targets. In addition, we map all attack procedures to the five stages introduced above. We summarize our results in Figure 5.2, which shows the sequence of activities per attack. If an attack does not perform any of the alternative activities at some point in the sequence, we record a transition through a *no-operation (NOP)* block.

5.3.3.1 Systematizing Prior Work on Prefetching-Based Attacks

Shin et al. [137]. This paper exploits the Intel IP stride prefetcher and targets the ECDH implementation in OpenSSL. In the preparation phase (S1), the attacker identifies memory lines in the OpenSSL shared library code that are cached only conditionally depending on the supplied input. These cache lines only appear in the cache when an internal lookup table is accessed such that a sequence of accesses forms a regular stride pattern. In the reset phase (S2), these memory locations are flushed from the cache by the attacker. Then, the attacker calls the library. The library trains (S3) and triggers (S4) the prefetcher only if the supplied input leads to memory loads that form a stride pattern. The attacker probes the cache lines through a cache-timing side channel (S5) to decide whether the input triggered the prefetcher or not. This prefetching-based primitive is then embedded in a differential attack to recover the secret.

Augury [131]. This paper is the first to investigate the data memory-dependent prefetcher (DMP) of the Apple M1 SoC. It describes the prefetcher's behavior when

multiple pointers, which are stored sequentially in memory (e.g., in an array), are loaded and dereferenced: The DMP fetches subsequent pointers and dereferences them. The authors present multiple approaches to exploit this behavior.

The first approach ("Out-of-bounds reads", *Augury OOB*) assumes that a user secretly selects a pointer from a finite list of candidate pointers. This pointer is stored just behind an array of pointers in the victim's memory. The goal of the attacker is to find the chosen pointer without accessing it architecturally. The attack is described with Flush+Reload and Prime+Probe; we discuss the more complex Prime+Probe variant. The attacker sets up an eviction set for each of the candidate pointers (S1) and loads them (S2). Next, all the pointers in the pointer array are dereferenced to train the prefetcher (S3). When the end of the array is reached, the prefetcher is triggered to prefetch past the array bound (S4) and to dereference the user-chosen pointer. By timing the access latency to all candidate pointers using the eviction sets (S5), the attacker decides which of the pointers was chosen.

In a second approach (*Augury SLH*), the authors discuss the impact of the DMP on code that uses speculative load hardening (SLH) [92] to protect against Spectre attacks [79]. The core idea of SLH is to verify untrusted (e.g., user-supplied) memory offsets during speculative execution to prevent speculative out-of-bounds accesses. The compiler adds branchless code that replaces out-of-bounds offsets with a safe value (often 0) using binary arithmetic. In Augury's example, a code snippet trains the prefetcher by iterating over a pointer array (S3). While SLH prevents *speculative* out-of-bounds reads, the prefetcher is still able to prefetch past the array bound when triggered by an access to the last array element (S4). Thus, a pointer that is stored just behind the pointer array can be fetched and dereferenced by the prefetcher, leaving traces in the cache that can be recovered (S5).

A third approach (*Augury Addr*) describes how the DMP can be used to determine whether an address is a valid (mapped) virtual memory address or not. The attacker sets up an array of 3 pointers, where the third pointer is the address to test (S1). The attacker ensures that the array is not cached (S2). Next, the attacker traverses the array in speculative execution and within the context where the mapping is to be checked. This trains the prefetcher (S3). As the DMP requires at least three valid addresses to be triggered for the first time (S4), prefetching will only happen if the address is valid. The attacker tests the cache state of the first out-of-bounds element after the array of pointers (S5). If this element is cached, the tested address is valid.

AfterImage [30]. This work exploits the Intel IP stride prefetcher in five different ways. Generally, these approaches exploit collisions on the instruction pointer (IP) address. The exploited prefetcher identifies patterns stored in its internal state based on the instruction address of the load instruction that caused the load. However, this instruction address is internally truncated to the 8 least significant bits. Consequently, the attacker can cause a collision by aligning a load instruction in their own code such that its 8 least significant bits match the respective bits of the instruction address in the victim code. The prefetcher is then unable to distinguish those two instructions from different contexts.

In *AfterImage variant 1*, the prefetcher is used to leak the control flow of a victim process. This attack is described with same-process and cross-process scope as well as

using Flush+Reload and Prime+Probe for extraction. We focus on the more complex cross-process, Prime+Probe variant. The attacker's goal is to determine whether a branch in a victim process is taken or not taken. To this end, the attacker selects one load instruction from each of the two potential code flows in the victim process and aligns two load instructions in their own process to them (S1). The attacker further prepares (S1) and loads (S2) eviction sets on the load targets in the victim process. The attacker then primes the prefetcher (S3) by training it in the attacker's process. Then, a context switch to the victim process happens, where the branch is either taken or not taken and the respective load instruction is executed, further (mis)training the prefetcher (S3). As the prefetcher was pre-trained, this load will further trigger prefetching after the load target of the victim instruction (S4). The attacker extracts those prefetching effects from the cache by reloading the eviction sets (S5).

AfterImage variant 2 exploits the stride prefetcher to determine whether a branch is taken in the kernel space. The authors attack a system call handler that operates on a memory buffer passed in from user space. The idea is similar to variant 1. However, aligning to a kernel instruction is more difficult, as the instruction address is unknown. For this reason, the attacker first determines the offset by testing all $2^8 = 256$ possibilities in a process called *IP matching* (S1). Then, the prefetcher is primed in the attacker process (S3). The system call is issued. If the branch is taken, the prefetcher will be further (mis)trained (S3) and triggered (S4) to prefetch memory from the buffer. After returning from the system call, the attacker probes the cache state of the respective locations in shared memory (S5).

In addition, AfterImage describes an attack on *SGX*. This attack does not exploit a collision. The goal is to leak control flow from an enclave. In the described setting, a load instruction is executed in a loop within the victim enclave. The instruction loads from a shared buffer that is passed from user space into the enclave. The stride of the loads is secret-dependent. This memory activity trains (S3) and triggers (S4) the prefetcher. After returning to user space, the attacker process recovers the stride by inspecting the cache state of the shared buffer (S5).

The paper further presents an attack on the *RSA* implementation of MbedTLS. The victim code contains secret-dependent branches that the attacker wants to monitor. To this end, the attacker first identifies suitable load instructions to align to by reverse engineering the victim binary (S1). Next, the attacker primes the prefetcher in their own memory to a high confidence level (S3) and switches to the victim code. The victim executes a colliding load instruction and re-trains the prefetcher (S3). As the loaded address will likely not match the previously trained stride, the confidence will be lowered. After switching back to the attacker process, the attacker tries to trigger the prefetcher in their own memory again (S4) and extract the prefetcher's behavior from the cache state (S5). The attacker will only observe prefetching effects if the confidence was not lowered by the victim, i.e., if the monitored branch was not taken.

Finally, the authors present a prefetching-based *synchronization primitive* that operates similar to the RSA attack. They envision this primitive could be used as a trigger for a power-based side-channel attack, for example to detect the beginning of a cryptographic operation. Again, the attacker begins by identifying a load instruction to align to in the victim code (S1). The prefetcher is then primed on a colliding load

instruction in the attacker's process to a high confidence level (S3). The attacker now switches frequently between victim and attacker code. As soon as the victim executes the target instruction, the prefetcher will be trained (S3) and the confidence will be lowered. The attacker tries to trigger the prefetcher (S4) and inspects the prefetcher's activity in the cache (S5). Once the prefetcher can no longer be triggered, the attacker knows that the victim executed the target instruction. In that case, the attacker raises a trigger signal.

Xiao et al. [160]. This paper uses the Intel IP stride prefetcher to attack an (undisclosed) AES implementation. To this end, the attacker monitors the cache state of the two memory lines just before and after the S-box. Depending on the access pattern to the S-box, which depends on plaintext and key, different prefetching activity in those cache lines is to be expected. After identifying the cache lines to monitor (S1), the attacker flushes them (S2). Then, the encryption is performed, potentially training (S3) and triggering (S4) the prefetcher. Finally, the cache state is inspected using a timing-based side channel (S5).

FetchBench [P2] (see Chapter 4). In our own previous work, we exploit the Spatial Memory Streaming (SMS) prefetcher in the ARM Cortex-A72 processor to attack the T-table-based AES implementation of MbedTLS. The attacker's goal is to extract the encryption key from the victim process by exploiting an instruction address collision. We first align load instructions in the attacker process with those in the victim process that load secret-dependent values (S1). We further identify a cache line that is accessed shortly before the victim process processes the secret (S1). This line, as well as a local probe array in the attacker process, are then flushed (S2). Next, the victim process encrypts an attacker-supplied plaintext using its own secret key. During encryption, the victim accesses multiple elements of a lookup table. The accesses to the lookup table train the prefetcher (S3). The attacker then switches into their own process using an inter-processor interrupt and triggers the prefetcher there (S4), making the prefetcher transfer the pattern learned in the victim context into the attacker's context. Finally, the attacker deduces the prefetcher's activity from the cache state (S5).

PrefetchX [29]. This paper uses the Intel eXtended Prediction Table (XPT) prefetcher to exploit the RSA implementations of MbedTLS and GnuPG. The XPT prefetcher is the only known prefetcher that is attached to the last-level cache (LLC) and thus shared across cores. It keeps a list of recently accessed pages and counts the number of cache misses per page. Once such a miss counter surpasses a fixed threshold for a page, the prefetcher effectively bypasses the LLC for future loads from that page.

The attacker and victim processes run on different cores. To start from a clean cache state, the attacker sends a signal to the victim process (S2). This enforces a context switch into the kernel, and the resulting overhead evicts the target cache line. Next, the attacker fills the prefetcher's state with pages that they control (S3). Then, the victim executes a code section containing a secret-dependent load. If the load is performed, the prefetcher's state is updated and one of the attacker's pages is evicted from the state (S3). The attacker then checks the prefetcher's state by triggering it on their own pages (S4) and measures whether the prefetcher still triggers or not (S5).

GoFetch [28]. This paper revisits the DMP prefetcher of the Apple M1 SoC and its successors and discovers that the DMP is more than a pointer array prefetcher: It can be

triggered by merely loading a single address from memory, even without dereferencing it. The paper presents attacks on four real-world targets: The Go implementation of the RSA cryptosystem, the OpenSSL implementation of the Diffie-Hellman key exchange, and implementations of the post-quantum algorithms CRYSTALS-Kyber and CRYSTALS-Dilithium. All attacks are based on the same idea. The attacker crafts malicious inputs that, when combined with secrets during computation of the target algorithm, result in intermediate values that form a valid pointer if and only if the secret fulfills a certain condition. The prefetcher is only activated if the condition is fulfilled, leaking information about the secret. On a high level, those attacks perform the following steps. First, an attacker process prepares (S1) and loads (S2) eviction sets that evict the pointer's anticipated location and target address. Then, the input is crafted and supplied to the victim process. If the condition is fulfilled, an intermediate value in the victim context forms a pointer. Once this pointer is loaded, the prefetcher is trained (S3, it updates its history) and triggered (S4, it dereferences the pointer). Finally, the attacker process re-loads the eviction sets (S5) to determine whether the condition was fulfilled or not.

Notably, these attacks apply even to constant-time implementations, which only guarantee constant execution time and (architectural) memory access locations, but do not constrain intermediate values.

5.3.3.2 Summary of Findings

Based on our systematization and the analysis in Figure 5.2, we make four core findings:

Finding: There Are Mandatory Stages. Our systematization shows that prefetcher training ①, prefetcher triggering ②, and cache extraction ③ (highlighted in red in Figure 5.2) are activities that are common to all attacks, as there are no *NOP* alternatives.

Finding: Prefetch Attacks Are Cache Attacks. We further note that all prefetching-based side channels are cache-based attacks: All attacks are built upon techniques to probe the cache state of certain cache lines ③, typically akin to Flush+Reload [161] and Prime+Probe [114].

Finding: Prefetching Is Triggered by Victim or Attacker. We find that many attacks rely on the victim context to trigger the prefetcher to extract information from the prefetcher's state and transfer it into the cache ②. However, recent works have shown that this step can be moved into the attacker's context in some cases, even though this involves more complex synchronization steps [29, 30, P2].

Finding: The Victim Trains The Prefetcher. Most importantly, we emphasize that all attacks rely on prefetcher training within the victim context ①. This is plausible, as the victim necessarily needs to work with the secret (e.g., perform secret-dependent memory accesses) to encode it into the prefetcher's state. Notably, from a defender's perspective, this means that the victim is able to protect itself using mitigations applied to its own code.

We conclude that a general mitigation approach against prefetching-based side-channel attacks is to *ensure that no training occurs in the victim context.* Fortunately, code running in victim contexts is easier to control than attacker code, which is (as the name suggests) constituted by the attacker.

5.4 PreFence: Design and Implementation

We now address RQ3.2 by presenting PreFence, our software-only countermeasure against prefetching-based side-channel attacks. It is based on the insight that the prefetcher is trained by the victim process in all attacks. We discuss alternative software-based defense approaches and why we consider them infeasible in Section 5.7.1.

5.4.1 Design Considerations

Design Goals. We want our countermeasure to fulfill the following design goals:

- **DG1:** It mitigates all prior prefetching-based attacks conducted from user space.
- **DG2:** It is simple to use for application developers and end users.
- **DG3:** It has minimal runtime overhead.
- **DG4:** It is functional when the prefetcher is shared across physical or SMT sibling cores.

Idea: Disabling the Prefetcher Temporarily. As we have shown in Section 5.3.3.2, stage S3 is a critical phase in all attacks. In this stage, the victim process encodes secrets into the prefetcher's state. This is where our countermeasure intervenes: Our idea is to allow user-space applications to disable the prefetcher temporarily in a fine-grained manner, for example while security-critical code is executed. We exploit (and verify in Section 5.5.2) that the prefetcher does not update its state while it is disabled, i.e., it cannot be trained in this state. In this way we fulfill DG1, since we avoid secrets being encoded into the prefetcher's state. At the same time, we limit the performance impact of the prefetcher being disabled to the security-critical code only, thus enabling DG3.

Naive Design: Giving User-Space Applications Direct Control. The following naive defense idea sounds straightforward at first: We could directly expose the processor's model-specific registers (MSRs) for prefetch control to user-space applications, giving them full control over the prefetcher's activation state. However, upon closer inspection, we identify four problems.

Problems With the Naive Design. First, we note that exposing the relevant MSRs directly to user-space applications requires application developers to write processor-specific code, since MSRs are not standardized (see Section 5.6.4). This violates DG2.

Second, we identify two issues that may occur when a security-critical process (which disabled the prefetcher) is descheduled and a non-critical process is scheduled instead: (i) the interrupting process cannot benefit from the prefetcher's performance improvements, violating DG3, and (ii) the interrupting process could maliciously re-enable the prefetcher, potentially violating DG1. Because scheduling events are transparent to applications, applications cannot easily address these issues. Thus, our solution needs to ensure that the prefetcher is re-enabled while other non-critical processes are running and disabled again when the security-critical process is re-scheduled.

Third, we identify a problem with prefetchers that are shared across physical or SMT sibling cores. A co-located malicious process with full control over the prefetcher could re-enable the prefetcher while the security-critical process is running, violating DG4. Consequently, we need to ensure that a prefetcher cannot be enabled if any of the co-located processes sharing that prefetcher requested it to be disabled.

Fourth, we note that a security-critical process may be migrated from one processor core to another. If the prefetcher operates on a per-core basis, it needs to be re-enabled on the original core and disabled on the destination core to fulfill DG1 and DG3.

The above problems clearly show that we have to dismiss the naive design: We cannot provide user-space applications with direct access to the prefetcher's activation state. In the next section, we propose PreFence, our improved mitigation design that addresses these problems.

5.4.2 PreFence Design

Solution: Scheduling-Aware Prefetch Control. Our proposed design, PreFence, addresses the problems with the naive design. It provides user-space applications with reliable deactivation of the prefetcher, regardless of the state of parallel processes and scheduling. We achieve this by tightly integrating our solution with the operating system kernel and the scheduler. The kernel can provide a unified interface to the user-space applications, abstracting away any processor-specific implementation details, thus enabling DG2. In addition, by integrating our defense into the scheduler, we can better address DG1, DG3, and DG4 and ensure that the prefetcher is disabled exactly as long as required, regardless of intermittent scheduling events.

Final Design. PreFence empowers user-space code to protect itself from prefetching-based side-channel attacks by requesting the prefetcher to be disabled temporarily. More precisely, a process signals to the operating system kernel when it enters or leaves a security-critical code section, such as an encryption function. This signaling requires only minimal code changes and can be integrated at various levels, from very fine-grained to more coarse-grained. If it is integrated at library level (a fine-grained way of applying PreFence), all programs that use the respective library benefit from it automatically. Alternatively, PreFence can also be integrated at application level, either to protect application-specific security-critical code or to wrap calls to security-critical legacy libraries that do not yet use PreFence themselves. Finally, PreFence also allows the user to decide that prefetching should be disabled for an entire process (the most coarse-grained way of applying PreFence). The implementation effort for library and application developers is low compared to complex rewrites required by other countermeasures (such as constant-time programming, see Section 5.7.1). In this way, PreFence achieves DG2.

Handling Scheduling. The kernel has to keep track of whether a process is currently in a security-critical code section and ensure that the prefetcher is disabled during this period—even if the process is interrupted by the scheduler. Once the prefetcher has been disabled, it remains disabled until the requesting process no longer executes security-relevant code, or until a different process not running security-relevant code is scheduled. If the next-to-be-scheduled process also requested to have the prefetcher disabled, the prefetcher remains disabled until non-critical code is reached.

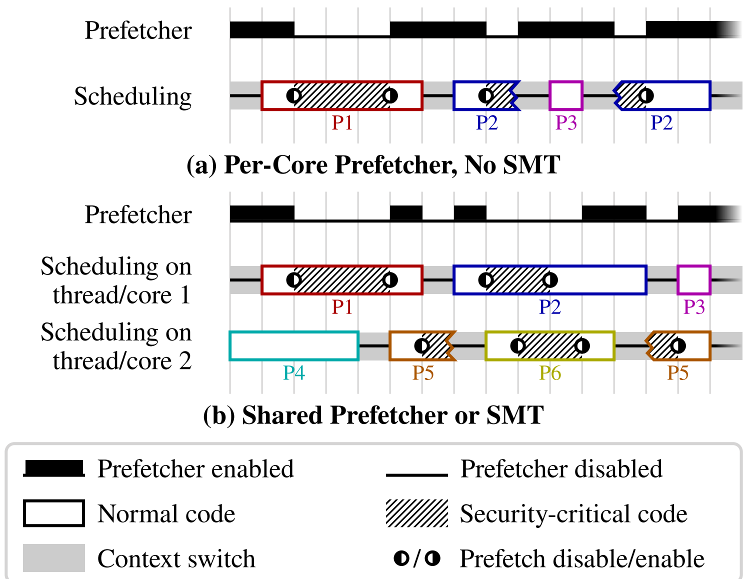

Figure 5.3: Integration of PreFence with the scheduler. The prefetcher is disabled temporarily while security-critical code runs. On SMT-capable cores and for shared prefetchers, the scheduler considers requested activation states of the relevant parallel processes.

This is enforced by the scheduler, which changes the prefetcher's activation state based on the security state of the next process to be executed.

The Simple Case: Per-Core Prefetcher, No SMT. We use the example in Figure 5.3 to illustrate our countermeasure, starting with the simple case in the upper half (a): a prefetcher that is exclusive to one core on a CPU without SMT. When a process is started, prefetching is enabled by default. As process P1 shows, the process can then request prefetching to be disabled, perform a security-critical operation, and request prefetching to be re-enabled. Process P2 illustrates the case where a process is interrupted by the scheduler during a security-critical code section. The scheduler deschedules P2 and checks whether the next process requested prefetching to be disabled or not. In this example, the next process (the new process P3) did not request prefetching to be disabled. Consequently, the scheduler re-enables prefetching while P3 is running. Once P3 is finished, the scheduler disables the prefetcher again and switches back to P2.

The Special Case: Shared Prefetcher or SMT. If the processor uses SMT and sibling cores share the same prefetcher, we need to pay attention to implicit context switches in order to fulfill DG4. With SMT, multiple processes can be executed on the same physical processor core simultaneously without operating-system-controlled context switches between them. In the worst case, when an attacker and a victim process are scheduled on sibling cores, the victim could request the prefetcher to be disabled, while the attacker requests it to be re-enabled. Similarly, if the prefetcher's state is shared across cores, an attack could be run simultaneously from a different core. For this reason, it is insufficient to update the prefetcher's activation state on OS-controlled context switches in these cases. Instead, we keep prefetching disabled on all (logical)

cores that share a prefetcher as soon as and as long as any of the scheduled processes requests it.

We provide an example in Figure 5.3 (b). First, process P1 requests prefetching to be disabled while the parallel process P4 does not request it. To protect process P1, prefetching is disabled until P1 leaves the security-critical code section. Next, process P5 enters a security-critical section but is soon interrupted by the scheduler. Since no more processes request the prefetcher to be disabled after P5 is descheduled, the prefetcher is re-enabled. Next, processes P2 and P6 both run security-critical code in parallel. The prefetcher is disabled when the first process (P2) enters the security-critical section and is re-enabled when the last process (P6) leaves it. Finally, P5 is re-scheduled. As it was interrupted in a security-critical section, the prefetcher is disabled at context switch and re-enabled once the security-critical section is completed.

Core Migrations. We note that our methodology also handles the case of a core migration. On a context switch, the scheduler adjusts the prefetcher's activation state based on the request of the next process. When a core migration occurs, the prefetcher's activation state on the original core is determined by the next process running on that core. On the target core, the migrating process is the next process, so it determines the prefetcher's activation state.

5.4.3 PreFence Implementation

Kernel Patch. We implement our countermeasure as a Linux kernel patch. Our prototype is currently able to control the prefetchers of Intel x86_64 (tested on Comet Lake) and ARM Cortex-A72 CPUs. Excluding comments, our patch adds only 91 (Intel) / 62 (ARM) lines of code to the Linux kernel code base. We extend the `task_struct`, the place where the scheduler keeps all information related to a process, with a boolean *prefetch_disable* flag. The flag is initialized to `false` for new processes, i.e., prefetching is enabled by default and can be disabled on request.

To allow processes to control this flag from user space, we add options to set, clear, or query the flag to the `prctl` system call. When the flag is changed through the system call, the kernel updates the `task_struct` of the calling process accordingly. In addition, the kernel changes the prefetcher's activation state on the respective CPU immediately by writing to the corresponding MSRs (bits 1-4 in MSR `0x1A4` on our Intel CPU [70], bits 21, 32, 42, 56 in MSR `CPUACTLR_EL1` on the ARM A72 [8]) before returning to user space. These MSRs can only be modified from kernel space. We provide an overview of prefetch-related MSR flags for various microarchitectures in Section 5.6.4. We further extend the scheduler's `context_switch` function to update the activation state of the prefetcher on context switches based on the *prefetch_disable* flag of the next process.

To deal with prefetchers shared across physical or SMT sibling cores, we keep a global bit vector that indicates for each CPU whether it currently runs a process with the *prefetch_disable* flag set. We check this bit vector before enabling the prefetcher on any core. Only if none of the cores sharing the same prefetcher currently runs a process with the *prefetch_disable* flag set, the prefetcher can be enabled; otherwise, it remains disabled.

Using PreFence in Applications. To make use of PreFence, the *prefetch_disable* flag needs to be set before entering security-critical code sections and cleared afterwards through system calls. These system calls can be issued in a fine-grained manner in library code or application code. PreFence can even be made available to users who want to protect legacy software: Users can invoke a wrapper program similar to `taskset` that just sets the *prefetch_disable* flag and executes the target application, which then inherits the flag. This effectively sets the *prefetch_disable* flag in a more coarse-grained way, affecting the entire codebase of the target application.

5.5 Evaluation of PreFence

In this section, we evaluate PreFence for efficacy and efficiency, based on our implementation. First, as a prerequisite, we evaluate the behavior of a disabled prefetcher to verify that disabling the prefetcher has the expected effects (especially, no further training), and thus PreFence is applicable. Next, we demonstrate the efficacy of our countermeasure: It prevents the prefetching-based side channel presented by Shin et al. [137] as well as our end-to-end attack proposed in Section 4.4.1. Finally, we show that PreFence is also efficient. We first evaluate the fixed overhead introduced on every context switch and the one-off overhead of a system call whenever a *prefetch_disable* flag is modified in isolation. Then, we investigate three scenarios:

- **Scenario 1: Stock kernel.** For an unmodified (“stock”) kernel, we measure the performance impact when the prefetcher is disabled for the whole execution time of an application (using SPEC benchmarks). These measurements represent the currently most viable general countermeasure and serve as a baseline for the following experiments.

- **Scenario 2: Patched kernel with non-critical workload.** For a patched kernel, we measure the performance impact (on SPEC benchmarks) when no process requests prefetching to be disabled. These measurements show the fixed performance overhead on a process introduced by our countermeasure.

- **Scenario 3: Patched kernel with critical workload.** As an end-to-end example, we evaluate the performance of a web server application running on a patched kernel. Using our system call, we disable the prefetcher during execution of TLS-related code. In addition to cryptographic code, each HTTP request to our server also triggers application code that can still benefit from the prefetcher (e.g., a file upload).

5.5.1 Evaluation Environments

We use the following two platforms throughout the evaluation.

x86_64. Our x86_64 platform is an Intel Core i7-10510U (Comet Lake) CPU running Alpine Linux 3.19 with kernel `6.6.14-r0-lts`. Depending on the experiment, we either use the original kernel from the Alpine repositories or our patched kernel derived from it. We use the `rdtscp` instruction to measure time.

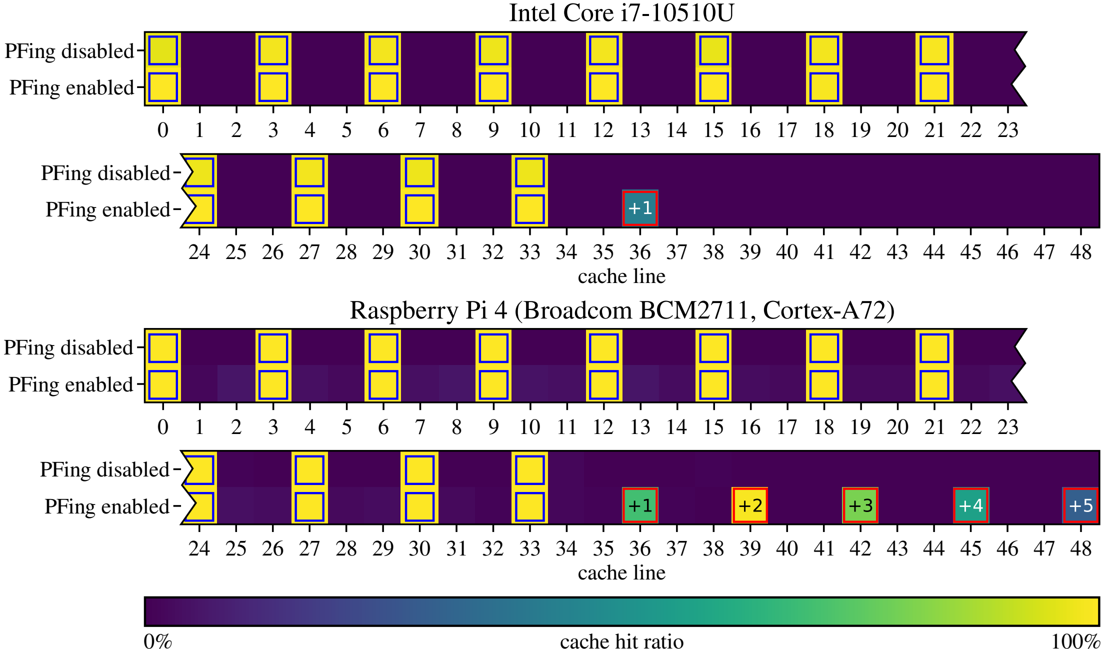

Figure 5.4: Prefetcher behavior when trained while disabled (compared to the behavior when trained while enabled). Brighter colors denote cache hits, darker colors denote cache misses. Blue frames indicate training access locations, red frames indicate prefetch locations. The tested prefetchers cannot be trained while disabled.

ARM. Our ARM platform is a Raspberry Pi 4 using a Broadcom BCM2711 SoC with four Cortex-A72 cores. It runs Raspberry Pi OS 12 64-bit with Linux kernel `6.6.22-v8`. We either use a kernel that we compiled from the official sources [125] without any changes or a kernel derived from it using our kernel patch. We use the cycle count register (`PMCCNTR_EL0`) to measure time.

5.5.2 Prerequisite: Disabled Prefetcher Behavior

PreFence requires that a disabled prefetcher cannot be trained, i.e., it does not update its state while it is disabled.

Experiment. To test the prefetcher for this behavior, we implement a corresponding testcase for stride prefetchers, the most common type of prefetchers, in the FetchBench framework (see Chapter 4). We first disable the prefetcher, access a sequence of memory locations with constant distance between them, re-enable the prefetcher, and perform one more memory access matching the pattern. If the prefetcher keeps learning while disabled, we expect it to be triggered by that last access and bring more elements into the cache. Otherwise, no prefetching effects should appear in cache. As a baseline, we repeat the same experiment with the prefetcher being enabled. In that case, we expect prefetching effects in the cache.

Results. We run the testcase on the Intel Core i7-10510U and BCM2711 processors. As illustrated in Figure 5.4, we find that the prefetchers cannot be trained while disabled. We conclude that PreFence can be used with these prefetcher implementations.

5.5.3 Efficacy: Protecting OpenSSL

In this experiment, we evaluate the efficacy of our countermeasure using the attack by Shin et al. [137] on the ECDH implementation in OpenSSL 1.1.0g as an example. Instead of re-implementing the end-to-end attack, we focus on reproducing the underlying prefetching side channel in both evaluation environments and show that PreFence prevents the leakage successfully.

Vulnerability. The leakage is caused by memory accesses to a lookup table when a point on an elliptic curve is squared. If those accesses (by chance) form a regular pattern, the prefetcher is activated and fetches memory lines before and/or after the lookup table. This prefetcher activity leaves traces in the cache state of shared memory, leaking relations between different portions of the point on the curve. Depending on the context where this operation is used, the point may be secret information.

Experiment. We identify the OpenSSL library function `BN_GF2m_mod_sqr_arr` as the function that operates on the lookup table. Our test program calls this function with a value that produces a regular access pattern and thus triggers the prefetcher (if enabled). It then accesses the potentially prefetched location, in our case the first cache line after the lookup table, and measures the memory latency to determine its cache state.

We repeat the experiment in two configurations. In the first configuration, we call the function without any countermeasure against prefetching-based side channels enabled. This experiment serves as a baseline and shows that the library function actually leaks information when called with certain inputs. In the second configuration, we set the *prefetch_disable* flag before calling the library function and clear it after returning from the library function. If PreFence is effective, we expect no more prefetching leakage.

Results. We run both configurations in both evaluation environments and present the results in Figure 5.5. We repeat each configuration 1,000,000 times on the Intel CPU and 10,000,000 times on the ARM CPU. When the *prefetch_disable* flag is cleared on the Intel CPU, we observe a significantly lower latency when loading from the memory line right after the lookup table (median: 96 units). This indicates that the prefetcher loaded this memory line into the cache (i.e., unwanted leakage). In contrast, when PreFence is activated on the Intel CPU by setting the *prefetch_disable* flag, the observed memory latency is above typical values for cache hits (median: 340 units), indicating a cache miss and absence of leakage. On the ARM CPU, we observe a weaker leakage signal (possibly indicating that the prior-work attack would not perform as well there). Without a countermeasure, the prefetching leakage appears at around 250 cycles. When we set the *prefetch_disable* flag before calling the target function, we observe that the leakage disappears reliably. We conclude that PreFence successfully prevents the prefetching-based side channel on both CPUs.

Execution Time Evaluation. For completeness, we also evaluate the temporal overhead of the prefetcher being temporarily disabled while the vulnerable library function is executed. We use the same experimental setup as before, but we additionally measure the execution time of the `BN_GF2m_mod_sqr_arr` function with and without the *prefetch_disable* flag set.

On the ARM CPU, we measure a slowdown of 2.6% when the prefetcher is disabled (the median execution time increases from 550 to 564 cycles). On the Intel CPU,

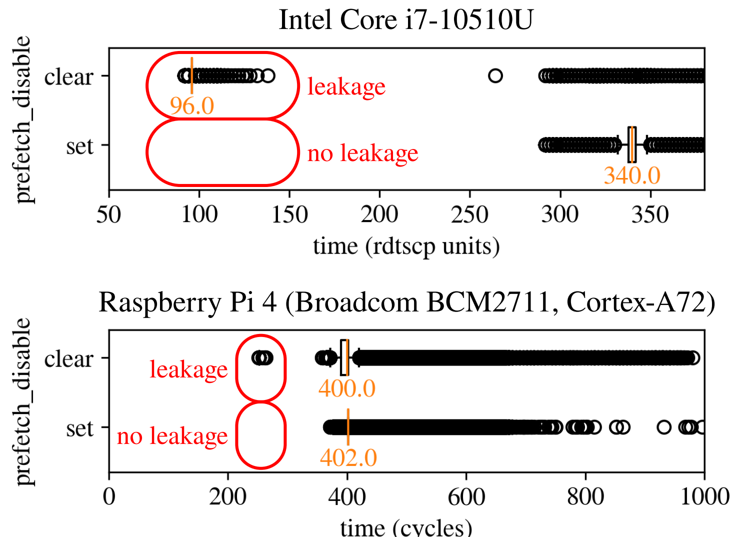

Figure 5.5: Results of the OpenSSL efficacy experiment. The plot shows the latency of accessing the prefetch location after calling the vulnerable OpenSSL function with the PreFence countermeasure not applied (*prefetch_disable* flag cleared) and applied (flag set). Short access latency indicates unwanted leakage, which is prevented by activating our countermeasure.

we find that the median execution time even decreases slightly when prefetching is disabled (from 374 to 368 units, a speedup of 1.8%), which we attribute to the prefetcher interfering with non-ideal predictions when it is enabled.

However, these measurements only reflect the performance of PreFence in an artificial individual case. Thus, we conduct an in-depth efficiency evaluation based on more complex and realistic workloads in Sections 5.5.6 and 5.5.7.

5.5.4 Efficacy: Protecting MbedTLS

Next, we show that PreFence successfully prevents an end-to-end attack from prior work, namely our attack on MbedTLS AES (see Section 4.4.1). As this attack exploits ARM's Spatial Memory Streaming (SMS) prefetcher, we can only reproduce it on our ARM-based platform.

Vulnerability. We recall that the SMS prefetcher divides memory into fixed-size regions of 1 KiB each. When a load instruction accesses multiple cache lines within the same region (e.g., in a loop), the prefetcher records this access pattern in its internal state. As the vulnerable AES-128 implementation issues key-dependent accesses to lookup tables (which span multiple such regions) during encryption, key-dependent information is encoded into the prefetcher's state. An attacker can extract this state and recover up to half of the secret key bits (i.e., 64 bits) using a properly aligned (aliasing) load instruction in their own code running on the same CPU core.

Experiment. We run two experiments: First, as a baseline, we run the end-to-end attack on our patched kernel, but without making any PreFence system calls in the victim code. This configuration is expected to show leakage. We record how many secret

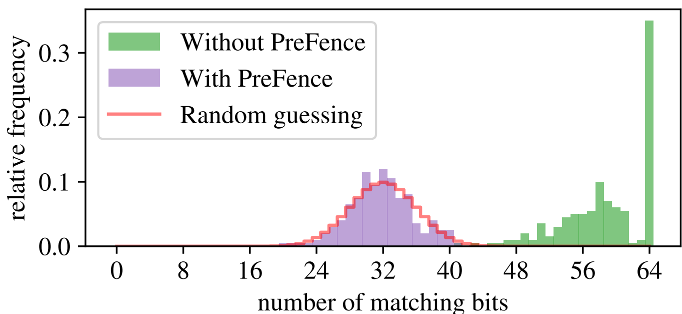

Figure 5.6: Results of strengthened attack on MbedTLS AES with 200 repetitions per configuration. The histogram shows how many key bits the attacker is able to extract correctly. When PreFence is not applied (green), all 64 key bits can be extracted in 35% of the cases. When PreFence is applied (purple), the attack is mitigated and the attacker's success rate drops to the level of random guessing.

bits can be recovered successfully. Second, we repeat the attack, but with PreFence applied. We set the *prefetch_disable* flag in the victim code before calling the AES encryption function and clear it afterwards. Again, we record the leakage.

Implementation. We build upon the FetchBench proof-of-concept code (see Section 1.2). Due to the complex synchronization between the attacker and victim processes, the original attack has a low success rate. This is not ideal for our experiment, because a failed attack (due to failed synchronization) is hard to distinguish from a prevented attack. Thus, we strengthen the attack to increase its success rate. We eliminate the need for synchronization through side channels by merging victim and attacker code into a single process and returning from the vulnerable library function right after the secret-dependent accesses have occurred. We further replace the cache inspection mechanism: Instead of Flush+Flush [44], we use the privileged RAMINDEX interface [8]. We emphasize that these changes only strengthen the attacker, not the victim. We will now show that PreFence can protect even against this stronger attacker.

Results. We repeat both experiments 200 times each and generate a fresh random key for every run. One execution of the end-to-end attack takes approx. 24.2 minutes on average. Figure 5.6 illustrates our results. Without protection through PreFence (green), we are able to leak 58.8 out of 64 recoverable key bits successfully on average. We leak all recoverable key bits in 35% of the attacks. With PreFence applied (purple), we observe a success rate that is essentially equivalent to random guessing, with an average success rate of 31.8 correct key bits per attack. The red line indicates the expected distribution for random guessing, more precisely, a binomial distribution with $n = 64$ independent guesses, where each bit guess is correct with a probability of $p = 0.5$. This expected distribution closely matches the observed distribution with PreFence applied. We conclude that PreFence successfully mitigates this attack.

Execution Time Evaluation. Finally, we also measure the temporal overhead on the vulnerable library function caused by the lack of prefetching. To this end, we call the function 10,000,000 times with and without the *prefetch_disable* flag set and measure its execution time. We find that the median execution time increases by approx. 2.7% when prefetching is temporarily disabled (from 903 to 927 cycles).

5.5.5 Efficiency: Overhead on Context Switch and System Call

We now measure the fixed overhead caused by the additional kernel code that needs to be executed on every context switch. In addition, we measure the overhead of performing a system call in order to set or clear the *prefetch_disable* bit of a process. Both experiments were only performed on the Intel CPU.

5.5.5.1 Fixed Overhead on Context Switch

Experiment. We evaluate the fixed overhead that our PreFence implementation adds to every context switch. To this end, we measure the execution time of a context switch in the stock kernel and compare it to the execution time in our patched kernel.

We implement two user-space processes that share a memory page. Both processes are pinned to the same CPU core. The first process constantly writes the current value of a high-resolution timer (retrieved from the `rdtscp` instruction) to shared memory. The second process reads the timer value from shared memory and computes the difference to the current timestamp. When the first process is scheduled, it keeps incrementing the timestamp written to memory until it is descheduled. Next, the second process is scheduled, computes the timestamp difference and logs the result. We filter out “zero samples” caused by the second process being re-scheduled before the first one, i.e., where the timestamp in memory has not been incremented compared to the last execution of the second process. We run this experiment on an idle system to maximize the probability of a context switch between our two processes.

Results. We measure the execution time of a context switch in three scenarios on our Intel CPU: (i) with the stock kernel, (ii) with our patched kernel, switching between two processes with the *prefetch_disable* bit cleared, (iii) with our patched kernel, switching from a process with the *prefetch_disable* bit cleared into a process with the bit set. We repeat each experiment until 10,000 non-zero samples have been collected.

We present our results in Figure 5.7. Unsurprisingly, the stock kernel has the smallest median context switch execution time of 3766 time units. Switching between two normal processes on our patched kernel requires 3842 units (median), a negligible increase of 76 units or 2%. When the prefetcher’s state needs to be changed on context switch, the median execution time is 4158 units, an increase of 392 units or 10% compared to the stock kernel.

5.5.5.2 One-off Overhead of the System Call

Experiment. We set up a user-space process that performs the `prctl` system call twice, once to set the *prefetch_disable* bit and once to clear it again. Before and after each of the system calls, we use the `rdtscp` instruction to get high-precision timestamps. Finally, we compute the difference between the timestamps.

Results. We repeat the experiment 10,000 times on our Intel CPU. Figure 5.8 shows the results. We note that the median duration of both system calls is around 430 units. The median duration of the system call to set the flag is negligibly longer than the median duration of the system call to clear it. For comparison, the overhead of the

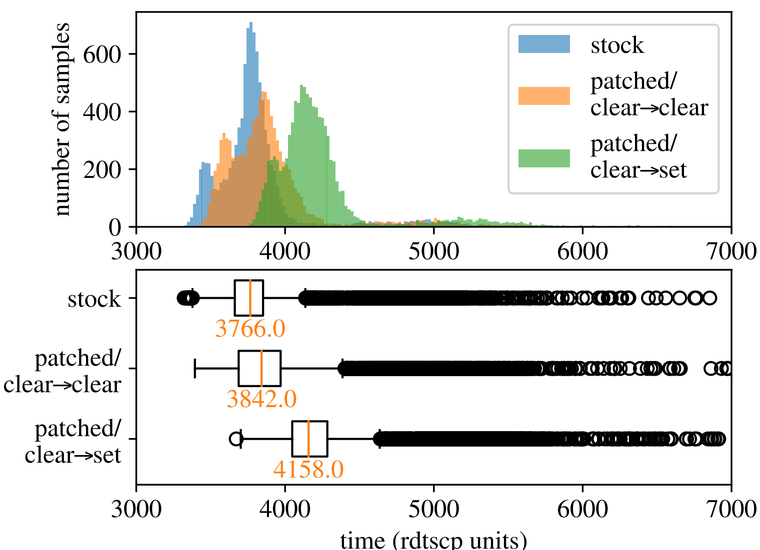

Figure 5.7: Context switch overhead on the stock kernel and on our patched kernel. For the patched kernel, we evaluate transitions between normal processes (*prefetch_-disable* bits cleared) and the transition into a security-critical process (from bit cleared to bit set). The added overhead is negligible.

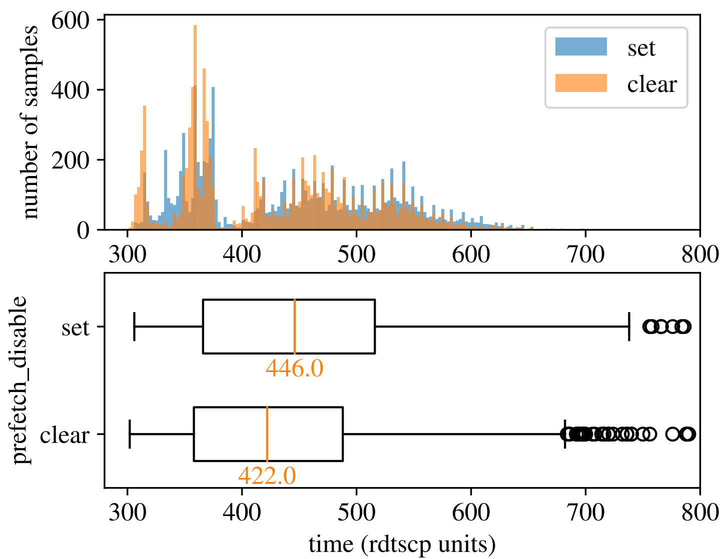

Figure 5.8: Overhead of a system call that sets or clears the *prefetch_disable* bit

system call is roughly in the same order of magnitude as a memory load that misses the cache (in the OpenSSL experiment in Section 5.5.3, we observed that a cache miss takes around 340 units on the same system).

5.5.6 Efficiency: Non-Critical Workloads (Scenarios 1 and 2)

We now investigate the efficiency of PreFence for complex workloads, starting with the performance impact on workloads that are not security-critical.

Experiment. We run *SPEC CPU 2017* benchmarks [141] on three different system configurations. As baselines, we measure the performance of the SPEC workloads on a stock kernel while prefetching is either enabled or disabled permanently (scenario 1). These measurements show us how much different workloads benefit from prefetching at all and how expensive the radical-but-simple defense of disabling the prefetcher permanently would be. Afterwards, we measure the performance of the same workloads on a patched kernel with prefetching enabled and without setting the *prefetch_disable* flag (scenario 2). This allows us to rate the performance impact on non-security-critical workloads caused by the added code that is executed on every context switch.

Benchmark Parameters. We run the *SPEC CPU 2017 Integer Rate* set of benchmarks and report the execution time of the individual benchmarks as a metric for their performance. We run each benchmark three times (the maximum number of iterations in a "reportable run" [140]).

Results. Figure 5.9 shows the benchmark results in both evaluation environments. The bars represent the median runtime of the individual benchmarks across the three iterations, while the black error bars indicate the runtime of the other two iterations.

Comparing the two stock kernel configurations (orange and blue bars), we find that the prefetcher especially speeds up the benchmarks 502–523. At a maximum, the prefetcher improves performance by 43% (benchmark 505 on the Intel CPU) and 37% (benchmark 502 on the Raspberry Pi), respectively. In most other workloads, both configurations performed similarly. In one exceptional case, we see a slowdown by 5% caused by the prefetcher (557 on the Raspberry Pi). Nevertheless, we conclude that disabling the prefetcher permanently can lead to a significant performance drop on both tested systems.

When we compare the stock kernel and the patched kernel, both with prefetching enabled (blue and green bars), we observe only small differences in execution time. For most benchmarks, the absolute difference is around 1%. We conclude that our kernel patch has negligible impact on non-critical workloads.

5.5.7 Efficiency: Security-Critical Workloads (Scenario 3)

Next, we evaluate the performance impact of PreFence on a security-critical workload that uses our protection mechanism.

Experiment. To evaluate the efficiency of PreFence in a realistic end-to-end scenario, we now apply it to real-world software. We use the web server *lighttpd 1.4.75* [84] (released in March 2024) as an example. Lighttpd ships with plugins for various cryptographic libraries that can act as backends to provide HTTPS support. In the following experiments, we use the OpenSSL plugin for this purpose. Our goal is

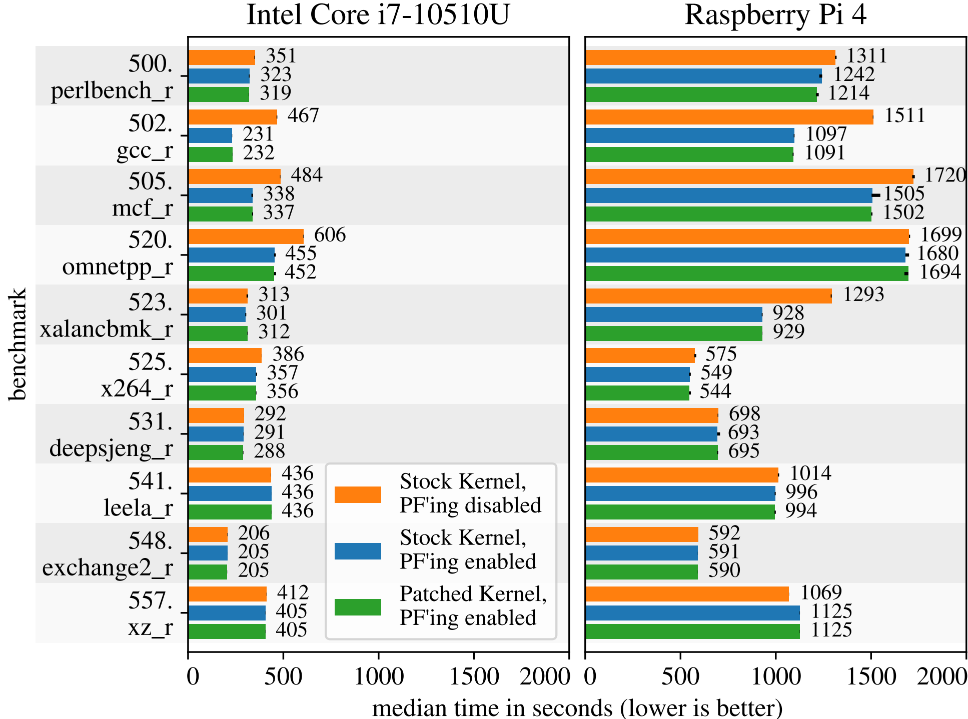

Figure 5.9: SPEC CPU 2017 benchmark results. Disabling the prefetcher permanently causes significant performance overhead in benchmarks 502 to 523. The performance overhead introduced by our patched kernel is negligible for non-security-critical workloads.

to protect the key material processed by OpenSSL from prefetch-related side-channel leakage. We compare two approaches of applying PreFence to the web server: fine-grained (at plugin level) and coarse-grained (at application level).

Fine-Grained PreFence. In this approach, we apply PreFence in a more fine-grained manner, i.e., at plugin level. We modify lighttpd's OpenSSL plugin such that the *prefetch_disable* flag is set whenever the control flow enters any function in the plugin code (which then calls OpenSSL) and cleared before the control flow returns from the plugin code. This approach allows the majority of the web server code base and the hosted web application to benefit from prefetching but causes frequent system calls to enable or disable the prefetcher.

Coarse-Grained PreFence. In this approach, we apply PreFence in a more coarse-grained way, i.e., at application level. We use a wrapper program to set the *prefetch_disable* flag immediately when lighttpd is started. In other words, the prefetcher is disabled for the whole lighttpd application, including all critical and non-critical server code and any hosted web application that is executed in child processes of the web server. This means that the server application cannot benefit from prefetching at all, but fewer system calls are required.

Web Application. In this case study, we use lighttpd to host a web application that we expect to benefit from prefetching in a scalable way. Our example application

is written in PHP and attached to lighttpd via CGI. It implements a simple file sharing service that allows the user to upload a file that is then stored on the server. During the HTTPS-encrypted upload, both the web server and the web application will process the file contents from beginning to end. We anticipate that these operations benefit from prefetching, especially for larger files.

Benchmarking Approach. We configure lighttpd to serve our web application via HTTPS over TLSv1.3 and access it through a benchmarking script. The script communicates with the web server via the loopback interface and uploads files of different sizes containing random data. We use the duration of the POST request that performs the upload as a performance measure. We repeat each experiment (i.e., each file upload) 300 times. To evaluate the performance of this setup when protected with PreFence, we evaluate it in scenario 3, i.e., with a patched kernel and prefetching temporarily disabled through our PreFence system calls. For completeness and as a baseline to compare with, we also include performance measurements of the same setup in scenario 1 (stock kernel) and scenario 2 (patched kernel with prefetching permanently enabled).

Results. We present the results in Figure 5.10. The x-axis indicates the size of the uploaded file. The y-axis indicates the duration of the upload POST request, relative to a baseline. This baseline is the median duration of the request in scenario 1 (stock kernel) with prefetching permanently disabled, normalized to 1, as shown by the black line. We represent the distribution of measurement values as boxplots. For a clearer presentation, we exclude the top and bottom 2% of values from each boxplot in this figure. The horizontal lines connect the boxplot's medians.

Finding: Application Benefits From Prefetching. Our first finding is that our example application generally benefits from prefetching on both CPUs. With prefetching permanently enabled (i.e., without any protection; blue/green lines), the upload performs 20.3% faster on average on the Intel CPU and 10.7% faster on average on our ARM processor compared to the baseline of disabling prefetching permanently. Thus, disabling prefetching permanently would be a costly measure.

Finding: PreFence Always Faster Than Baseline. All PreFence variants reduce this cost significantly; they always perform better than the baseline (permanently disabling prefetching).

Finding: Fine-Grained PreFence is Faster. We observe that the fine-grained use of PreFence at plugin level (purple) performs better than the more coarse-grained use of PreFence at application level (orange) in this experiment. The lower number of system calls in the application-level scenario does not compensate for the slowdown of the non-critical parts of the web server and the web application. The better-performing variant, PreFence at plugin level (purple), is not as fast as the insecure default configuration (i.e., having prefetching permanently enabled; blue/green), as the OpenSSL code can no longer fully benefit from prefetching. However, with fine-grained use of PreFence, we are able to reclaim a performance improvement of 15.8% on average over the baseline on the Intel CPU and 7.2% on the ARM processor. For coarse-grained use of PreFence, the average performance improvement is still 7.8% (Intel) and 3.1% (ARM) over the baseline.

Finding: Negligible Performance Impact of Scheduler Patch. We observe again that the overhead of our scheduler patch is negligible for non-critical code, as the blue and

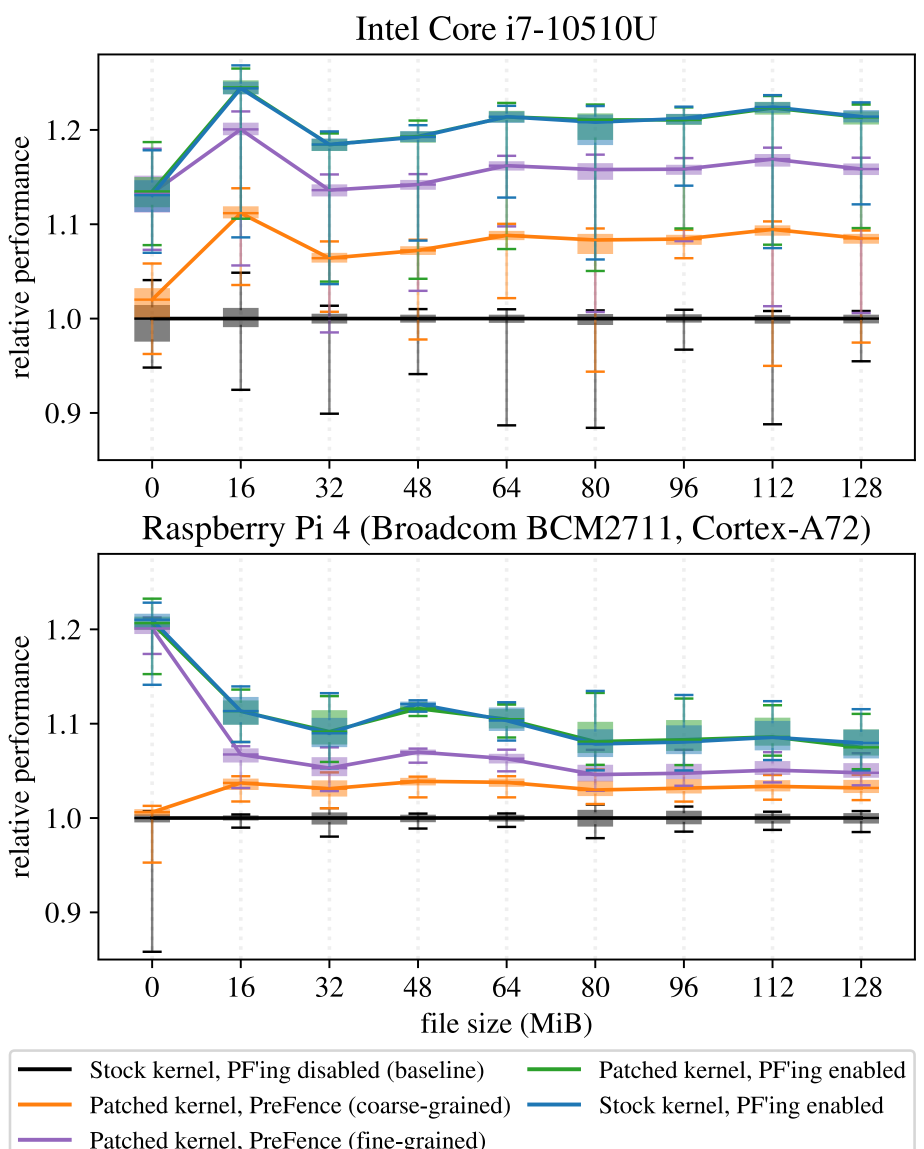

Figure 5.10: Lighttpd benchmark results, showing the performance benefit of PreFence compared to prefetching being permanently disabled for a security-critical workload (black). Prefetching generally speeds up the process, and once a significant file size is reached, the relative performance benefit is approximately constant. Any form of PreFence performs better than disabling the prefetcher permanently. Fine-grained use of PreFence (purple) improves performance by 15.8% (Intel) and 7.2% (ARM) over the baseline on average, approaching the performance of prefetching being enabled permanently (blue/green).

green lines overlap in both plots. This confirms our result from Section 5.5.6 that programs that do not use PreFence do not suffer a noticeable performance drop.

5.5.8 Summary of Results

We have demonstrated that PreFence is applicable to our Intel and ARM processors, as their prefetchers cannot be trained while they are disabled (Section 5.5.2). We also verified that PreFence is effective, as it eliminates leakage caused by the prefetcher in a known-vulnerable library function on both architectures (Section 5.5.3) and mitigates an end-to-end attack from prior work successfully (Section 5.5.4). We further demonstrated that PreFence is efficient: The additional overhead at context switch is negligible and the cost of the system call is small. For workloads that are not security-critical, such as SPEC benchmarks, we observed a performance overhead of less than 1% for the majority of the benchmarks (Section 5.5.6). For our example of a security-critical workload, a lighttpd web server serving a web application that benefits from prefetching via HTTPS, we found that PreFence always performs better than disabling the prefetcher permanently. More specifically, when PreFence is applied in a fine-grained manner at the level of the OpenSSL plugin, our example application runs up to 15.8% (Intel) and 7.2% (ARM) faster on average compared to the prefetcher being permanently disabled (Section 5.5.7).

5.6 Discussion on PreFence

Next, we discuss the applicability of PreFence to different types of hardware prefetchers (Section 5.6.1) and an approach to automate the insertion of its system calls. In addition, we discuss the applicability of PreFence to different scopes, different processors and processor flags, and other types of prefetching. Finally, we discuss PreFence in the context of prefetching-based covert channels.

5.6.1 Applicability to Hardware Prefetcher Types

When PreFence is to be applied to real-world programs, the question arises which code sections should be wrapped in PreFence system calls. Naturally, applying PreFence to a whole process (coarse-grained) is an easy-to-implement and trivially secure option. However, it is not the most performant one. For more fine-grained use of PreFence, the leaking code sections need to be identified and wrapped in PreFence system calls. As we found in Section 5.5.7, this additional effort at development time is rewarded with better performance at runtime.

When PreFence is to be used in a more fine-grained manner, the system call locations depend on the prefetcher types that are expected to be present in the target system. More precisely, we need to distinguish between systems with only address-based prefetchers and systems also featuring DMPs.

Address-Based Prefetchers. If only address-based prefetchers are present, it is sufficient to apply PreFence to code regions that operate on secret-dependent addresses. We envision that automated taint tracking can facilitate this process, as we discuss in Section 5.6.2.

DMPs. To also protect against side channels introduced by DMPs, PreFence system calls need to be placed more conservatively to account for data-dependent predictions as well. It is intuitive to place system calls around code that operates on secret-dependent values directly. Otherwise, a DMP may activate and operate on the secret-dependent values, potentially leaking them to an attacker. In addition, however, we need to consider cases where a DMP is triggered to operate on secret-dependent values by a memory operation on non-secret values. We consider two such scenarios: (i) adjacent buffers and (ii) multiple levels of pointer indirection.

The first scenario illustrates the problem of adjacent buffers: Consider a non-secret buffer and a secret buffer, located consecutively in memory. When non-critical code traverses the non-secret buffer, the DMP may be activated and continue prefetching beyond the buffer's bounds into the secret buffer. The DMP may then interpret secret-dependent data as a pointer, dereference it, and encode secret-dependent information into the cache state. Thus, memory operations on adjacent buffers should also be wrapped in PreFence system calls. While this is a non-trivial task to perform manually, we discuss how this step could be automated in Section 5.6.2.

The second scenario occurs with DMP implementations that prefetch pointers multiple levels of indirection deep. In this case, an attacker could craft a pointer into security-critical memory and dereference it to trigger the DMP. If the DMP then proceeds to dereference a secret value as well, this action may leave traces in the cache. In this scenario, any pointer dereference in the application code could be exploited to trigger the prefetcher on a security-critical buffer. Thus, PreFence should be applied to the whole security-critical process in this case. However, to the best of our knowledge, no current DMP implementation (neither Apple nor Intel) follows pointers multiple layers of indirection deep.

5.6.2 Automating System Call Placement

Our approach depends on system calls being placed before and after security-critical code sections. While the manual effort required to apply PreFence to a whole process (coarse-grained use) is low, placing system calls at finer granularity is expected to cause higher effort.

To reduce the manual effort required for fine-grained use of PreFence, we envision that prior research on taint tracking can be leveraged to place system calls automatically. For example, the approach of CryptoMPK [75] could be adapted. CryptoMPK aims to automatically identify code that operates on buffers containing secret or secret-dependent values. Such code is then surrounded with instructions that switch memory protection keys (MPKs). These additional instructions could also be our PreFence system calls.

With CryptoMPK, developers annotate buffers that contain *initial secrets* (e.g., keys), which are easier to identify than security-critical code. CryptoMPK then tracks the use of those buffers. It identifies additional secret-dependent buffers as well as all memory operations operating on secret or secret-dependent buffers.

To protect against leaks from address-based prefetchers, CryptoMPK could place the PreFence system calls before and after such memory operations. In addition, Crypto-MPK could be extended to facilitate fine-grained system call placement for DMPs:

To eliminate the problem of a DMP being triggered on adjacent buffers (as discussed in Section 5.6.1), such adjacent buffers could be treated as security-critical as well. Consequently, operations on those buffers would also be protected with PreFence system calls, preventing prefetcher activation.

5.6.3 Applicability to Different Scopes

Our current implementation of PreFence, as presented in Section 5.4.3, protects a user-space process from attacks by other user-space processes (scopes SP and CP, as introduced in Section 5.3.2). This covers 11 out of the 13 attacks that we discussed in Section 5.3. The general principle of PreFence can also be used to protect kernel code from user-space attacks (scope KU): The kernel could disable the prefetcher temporarily during security-critical operations. To prevent attacks from an untrusted operating system on a trusted execution environment (scope TO), the prefetcher could be disabled temporarily while security-critical code is executed in trusted execution. However, because the untrusted operating system can usually interrupt trusted execution [82, 130, 150], the prefetcher's activation state needs to be saved when an interrupt causes a context switch to the untrusted operating system and restored when switching back. For Intel SGX, which is interrupt-unaware [100, 150], this requires additional hardware support.

5.6.4 Applicability to Other Processors and Flags

Other Processors. We implement and evaluate PreFence for two specific CPUs. These CPUs represent popular attack targets (such as the Intel IP stride prefetcher [30, 137, 160]) and cover two popular architectures (x86_64 and ARMv8). However, as we have shown in Chapter 4, prefetcher implementations can differ widely, even across processors of the same brand or architecture. We are confident that the design of PreFence is general enough to be transferred to other processors as well, as long as those fulfill two basic requirements. First, the processor must expose a way to disable the relevant prefetchers dynamically at runtime, e.g., by setting a bit in an MSR. Second, the prefetcher must not update its state while disabled.

To get an overview of how widespread processor support for MSRs controlling the hardware prefetchers is, we examined a large corpus of technical reference manuals from Intel, AMD and ARM. We report our detailed findings in Section A.2.1 in the appendix. Our results indicate that most processors provide a way to control hardware prefetchers, but the relevant MSRs are not unified, neither across vendors nor for processors of the same vendor. This strengthens our position that it is infeasible for user-space applications to interact with the prefetcher directly; instead, the operating system kernel should abstract from implementation-dependent interfaces and provide a unified and easy-to-use way for user-space applications to control hardware prefetchers. The design of PreFence allows for this architecture. The Intel XPT prefetcher is an example for a prefetcher that cannot be controlled dynamically, as Intel does not publicly disclose any way to control it [67]. In such cases, PreFence cannot be applied.

Other Flags. We emphasize that PreFence's general design is not limited to managing prefetching-related flags. PreFence could be adapted to control other mi-

croarchitectural components during the execution of security-critical code as well. For instance, ARM and Intel recently introduced the *Data-Independent Timing (DIT) flag* [7, 12] and the *Data Operand Independent Timing Mode (DOITM) flag* [72], respectively. The processors concerned only guarantee data-independent timing behavior for certain instructions when these flags are set. These flags are meant to be set while cryptographic operations implemented in constant-time code are executed. They temporarily disable a range of optimizations that impact timing behavior (including, but not limited to, certain prefetchers [28, 72]). PreFence could be adapted to dynamically set and clear these (or other) flags as well.

5.6.5 Applicability to Other Prefetching Types

We focus on mitigating vulnerabilities caused by hardware prefetching *on data*. As prior work has shown, prefetching *on instructions* can also be exploited to leak information [166]. In principle, PreFence can also be applied to this kind of prefetching, as long as the prefetcher can be controlled. In fact, our implementations for both architectures also disable the instruction prefetchers when the *prefetch_disable* flag is set. However, when PreFence is applied in a very fine-grained manner, i.e., only wrapping carefully selected code sections, the selection of code sections may need to be reassessed with instruction prefetching in mind to make PreFence fully effective against instruction prefetch attacks. This is because code sections vulnerable to attacks on instruction prefetching might be different from those that are vulnerable to data prefetch attacks.

Another kind of prefetching is *software prefetching*. While software prefetching may also introduce vulnerabilities [42, 85], those vulnerabilities are not caused by automated prediction mechanisms. Instead, they concern explicit prefetching instructions. Thus, we consider software prefetching vulnerabilities an orthogonal problem that is out of the scope of this thesis.

5.6.6 Covert Channels

In this chapter, we focus on mitigating prefetching side channels and exclude covert channels from the scope. Prefetching-based covert channels have been proposed in recent work. Cronin and Yang [32] implement a covert channel using the Intel IP stride prefetcher. They prime the prefetcher from the receiver's end, either evict or keep the primed patterns from the sender process, and finally probe for the existence of the primed patterns in the receiver process. This probing either triggers the prefetcher or not, indicating either a 0-bit or a 1-bit, respectively. Chen, Pei, and Carlson [30] exploit the same prefetcher but encode the information to transmit into the stride. Rohan, Panda, and Agarwal [128] exploit the Intel stream prefetcher. The sender triggers the prefetcher on shared memory. The direction of prefetch (forward or backward) is interpreted as a 1-bit value by the receiver. Chen et al. [29] implement two covert channels based on the Intel XPT prefetcher. For the first channel, the receiver primes the prefetcher's state and the sender either idles or changes the state by evicting one of the primed entries. For the second channel, the sender either trains or resets the prefetcher for a shared page. In our own prior work (see Chapter 4), we encoded bit

vectors into the region-based ARM SMS prefetcher to transfer information from trusted execution to the untrusted OS.

All of these attacks have in common that it is the attacker, not the victim, who controls the training stage. In fact, there is no victim process at all in a typical covert-channel setting: Covert channels are incapable of leaking secret information on their own. Rather, they are a means of exfiltrating secret information that the attacker has obtained in another way beforehand. As these attacks do not leak information out of a victim process directly, a victim process has no incentive to defend against them. Thus, PreFence is not applicable in this case.

5.7 Related Work on Defenses Against Prefetching-Based Attacks

In this section, we provide an overview of software- and hardware-based mitigations against prefetching-based attacks that were proposed in prior work, and we relate them to our approach.

5.7.1 Related Work on Software Defenses

We start by discussing mitigations from prior work that can be applied via (modified) software and argue that none of them prevent attacks in an easy and efficient way.

Constant-Time Programming [30, 137, 160, P2]. One way to prevent most prefetching-based side channels, and also other cache-timing side channels, is the programming technique of *constant-time programming*. Despite the name, it not only refers to writing code that executes in the same time regardless of the (potentially secret) information processed, but it also mandates that no secret-dependent control flow or memory access patterns occur [64]. Avoiding secret-dependent memory access patterns prevents address-based prefetchers from transferring secrets into their internal state during training. Moreover, avoiding secret-dependent branches prevents prefetching-based attacks that infer the victim's control flow based on conditionally executed load instructions. However, constant-time programming is ineffective against attacks on DMP prefetchers that exploit secret-dependent *values* instead of addresses or branches [28]. In addition, making code constant-time requires complex rewrites and results in significantly reduced performance [26, 118].

Clearing the Prefetcher's State on Context Switches [30, 32, P2]. Some prefetching-based attacks train the prefetcher in a victim context, then switch into the attacker's context and trigger it there. Relevant context switches are transitions between two user-space processes, transitions between kernel and user space, and returns from trusted execution. To mitigate such attacks, the prefetcher's state could be cleared on context switches. This is straightforward to implement if the CPU provides a suitable instruction, such as `CPP RCTX` on some ARM CPUs [12, 13]. Otherwise, all stored patterns need to be evicted, which is computationally expensive and requires knowledge of implementation details such as the number of patterns stored and the replacement policy.

We note that clearing the prefetcher's state on context switches is an incomplete countermeasure in three cases. First, it is not applicable to attacks that trigger the prefetcher in the victim process. Second, this countermeasure assumes that the prefetcher's state is not shared across physical or SMT sibling cores. Otherwise, the attack could be executed from a different core before the context switch resets the state. Third, in the case of trusted execution, prior work has shown that an attacker is able to interrupt trusted execution before it completes [82, 130, 150]. If this is possible, clearing the prefetcher's state only at the end of a trusted execution procedure is insufficient.

Mitigating Cache-Timing Side Channels [30, 137]. We found in Section 5.3.3.2 that all prefetch attacks rely on cache-timing side channels. To mitigate those, access to timer interfaces can be restricted or their resolution can be reduced. This mitigation has especially been applied to browsers in the past [31, 152]. However, attackers may fall back to alternatives such as a counter thread as a timer replacement [86]. General countermeasures against cache-based side channels have been discussed in prior work extensively [89, 114, 161, 165], but none were implemented on a large scale. Consequently, we consider it next to impossible to reliably block an attacker from all possible ways to generate precise timestamps.

Anomaly Detection [30]. During a prefetching-based side-channel attack, the attacker may execute code that results in unusual values of performance counters. For instance, counters related to cache activity or prefetch-related events may increase at a higher rate than usual. These counters can thus be observed to detect unusual activities [48, 116, 164]. However, such a heuristic detection system will produce false-positive alerts, miss some malicious events (false negatives), and introduce a constant runtime overhead affecting all workloads. In addition, intrusion detection systems generally do not prevent attacks but merely detect them after they have started. Thus, we consider this mitigation strategy incomplete.

Security-Aware Core Assignment [28, 131]. For per-core prefetchers, another possible mitigation is more advanced core assignment. On heterogeneous processors, a vulnerable prefetcher may only be present on some of the cores. In this case, security-critical or untrusted workloads could be assigned to invulnerable cores. Similarly, a vulnerable per-core prefetcher could be disabled on one of the cores, which could then be reserved for critical workloads. In practice, however, it is not trivial to decide which processes should be assigned to which core. In addition, reserving a core for critical operations is likely to reduce overall system performance: If critical workloads are frequent or long-running, assigning them to a single core will limit their throughput. If critical workloads are rare or short-running, reserving one core for them will result in the core idling most of the time.

Oblivious Execution [30]. Oblivious execution [124] eliminates the side-channel effect of a secret-dependent conditional. It executes both branches but persists only the result of the correct branch in memory. Thus, the attacker can no longer distinguish those branches: The resulting timings and memory access patterns are always the same. However, this approach comes with a significant performance overhead. Rane et al. [124] report a mean overhead of 16.1×.

Blinding [28]. A DMP may be triggered by a data value that has a specific property, e.g., that looks like a pointer. To ensure that such a prefetcher is not triggered

by an untrusted value, a mask can be added to the value before it is stored in memory and removed after it is loaded again. However, implementing this countermeasure is not trivial and introduces computational and memory overhead.

Disabling the Prefetcher [28, 29, 30, 32, 137, P2]. If the CPU allows for controlling the prefetcher, the most straightforward way to prevent any leakage from the prefetcher is to disable it permanently. However, this countermeasure comes with a significant performance decline for workloads that benefit from prefetching, as we show in Sections 5.5.6 and 5.5.7. The general idea of disabling the prefetcher temporarily has been briefly mentioned in the context of offensive works before [28, P2], but it has not been elaborated further. No detailed design, implementation, or evaluation has been provided. In particular, prior work did not make any considerations beyond the "naive approach", which we dismissed as incomplete in Section 5.4.1. Notably, no integration with the scheduler was proposed, which is vital to make the defense complete and easy to use. In addition, the special case of SMT was not considered. PreFence fills this gap.

Conclusion: Prior-Work Countermeasures Are Costly or Incomplete. In summary, every prior-work countermeasure violates one of our design goals stated in Section 5.4.1. We consider *constant-time programming* complex to implement, expensive at runtime, and ineffective against DMP-based side channels; *clearing the prefetcher's state on context switches* specific to attacks that trigger the prefetcher in the attacker's context, expensive at runtime, and incomplete when the state is shared across physical or SMT cores; *mitigation of timing sources* and *anomaly detection* inherently incomplete approaches; *security-aware scheduling* hard to implement in an efficient way; *blinding* complex to implement, expensive at runtime, and specific to DMP-based side channels; *oblivious execution* and *disabling the prefetcher permanently* expensive at runtime.

5.7.2 Related Work on Hardware Defenses

We generally focus on software-based mitigations in this chapter, as those can easily be applied and evaluated on current hardware. However, for completeness, we also briefly discuss defense proposals that require hardware modifications.

Choosing a different trigger [P2]. Some attacks rely on collisions on the prefetch trigger, e.g., the (partial or complete) address of a load instruction. To mitigate such attacks, the instruction address must not be used as the only trigger. For example, a process identifier could be added. Only if the process identifier *and* the instruction address match, a prefetch can be triggered.

Partitioning the Prefetcher's State [29, 30, 32, P2]. Partitioning protects against accidentally triggering a prefetch pattern in a wrong context. To avoid leakage between privilege levels, the prefetcher could keep track of the privilege level that a pattern belongs to. However, this approach does not protect against attacks within the same privilege level, e.g., between two user-space processes. To distinguish between those, an additional process identifier must be stored. Depending on the implementation, attackers may still be able to prime the prefetcher with attacker-controlled patterns that are then potentially evicted by victim activity, leaking control flow information.

Extending the Instruction Set [131, P2]. An instruction could be added that flushes the prefetcher's state, such as the `CPP RCTX` instruction available on some

ARM CPUs [12, 13]. This instruction could then be called by the operating system on context switches or when switching between privilege domains. However, this approach only works when the prefetcher is not shared among multiple cores or SMT threads. Alternatively, a special load instruction to be used in security-critical code sections could be introduced that does not influence the prefetcher's state.

5.8 Conclusion on PreFence

In this chapter, we answered the question: *How can we systematically defend against prefetching-based attacks?* We started by providing the first systematic analysis of 13 existing related attacks from literature, and we showed that all rely on three main stages: prefetcher training, prefetcher triggering, and cache extraction.

Our proposed countermeasure, PreFence, allows vulnerable user-space programs to ensure that they do not train the prefetcher. More precisely, it enables processes to selectively disable the prefetcher from user space, while ensuring that parallel processes sharing the same prefetcher on other cores or SMT siblings are considered as well. In addition, issues in process re-scheduling are considered and handled transparently for the vulnerable process. PreFence enables fine-grained control to minimize the time the prefetcher is unavailable, while ensuring that the critical *prefetcher training* attack stage cannot target the victim process any longer. Our prototype implementation of PreFence for an Intel x86_64 and an ARM Cortex-A72 processor is a Linux kernel patch to the scheduler that automatically ensures correct prefetcher activation state on process re-scheduling. In addition, it provides processes with a system call to request the prefetcher to be disabled. Our PreFence implementation is open-source software (see Section 1.2).

We demonstrated the efficacy of our approach by successfully mitigating two prior-work side-channel vulnerabilities with low overhead (less than 3%) on the vulnerable functions. Our performance evaluation showed that the performance impact of our solution on non-security-critical code is around 1%. For security-critical workloads, its performance impact depends on the way PreFence is applied to the code; for a real-world web server application, we showed that security-critical code runs 15.8% (Intel) and 7.2% (ARM) faster on average compared to prefetching being permanently disabled when PreFence is applied in a relatively fine-grained manner.

In conclusion, we presented an easy-to-use and efficient scheduling-aware countermeasure to protect victim processes against prefetcher side channels, founded on a systematic analysis of prior work on attacks and countermeasures. We expect our countermeasure could be extended to the general signaling of security-relevant code to the kernel to allow for coordinated application of countermeasures (e.g., DIT flags).

6
Systematization of Knowledge: Countermeasures Against Attacks on Microarchitectural Features

6.1 Introduction on Countermeasures

In recent years, researchers have uncovered and discussed many novel attacks on microarchitectural features in academic literature. By contrast, academic discussion of the respective defenses is limited and often only occurs marginally as part of offensive papers.

One potential reason for this is that researchers lack the proprietary knowledge and tools necessary to fully understand and modify a processor's internals, which is necessary in order to develop effective and comprehensive defenses. Therefore, the responsibility of fixing microarchitectural side-channel vulnerabilities primarily falls to processor manufacturers, followed by developers of operating systems, compilers, libraries, and applications. Unfortunately, their work is rarely the subject of further public or academic discussion. Consequently, seven years after the initial publication of Spectre, the landscape of microarchitectural defenses remains opaque and confusing. Public information on many of these defenses is scarce, scattered across various sources, and lacks detail.

In this chapter, we ask the high-level research question (RQ4): *How can we systematically identify gaps in the academic and non-academic consideration of defenses against microarchitectural attacks?* We address this question in three parts, examining three specific gaps:

RQ4.1 (Architecture Gaps): Which processor models and architectures do academic research papers on microarchitectural attacks and defenses consider, and are there architectures that are under-covered?

RQ4.2 (Mitigation Gaps): Can the proposed defenses against microarchitectural attacks fully mitigate the proposed attacks?

RQ4.3 (Implementation Gaps): Are the defenses referenced in literature actually implemented?

Research question RQ4.1 allows us to identify potential biases towards specific processor architectures and vendors. When answering RQ4.2 and RQ4.3, we distinguish between *theoretical* and *practical* completeness of defenses. We first assess whether the attacks we consider can be mitigated by the applicable defenses in theory, i.e., based on their textual description in literature. We then proceed to determine for which of the defenses one or more implementations are referenced, indicating that the defense also exists in practice.

We collect relevant literature, including academic and non-academic sources, and compile a dataset that we use for further analysis. In this dataset, we collect information about the documents we consider, the attacks, defenses, and defense implementations they discuss, the relations between those, and the processor models used for evaluation.

With the help of this dataset, we find that a majority of the research papers in our corpus focuses on the x86_64 architecture and Intel processors in particular. Our systematic analysis of attacks and defenses yields that most attacks can theoretically be

defended against, with the exception of *Retbleed on AMD*. As far as implementations are concerned, we only find references to concrete implementations for about half of the defenses that are discussed in the documents in our dataset. Many of them are closed-source implementations. Finally, we also discuss limitations of our current dataset that may prevent it from being representative, especially with respect to implementations. We give directions for future expansion to overcome this issue.

Contributions. In summary, we make the following contributions:

- We systematize academic and non-academic literature on microarchitectural attacks and defenses into a novel dataset, linking attacks with defenses and allowing data-based reasoning about them. We publish this dataset to foster future research (see Section 1.2).
- We use our dataset to identify gaps of research papers on microarchitectural attacks and defenses with respect to the architectures that they consider.
- We further use our dataset to identify gaps in the theoretical and practical completeness of defenses, indicating where vulnerabilities remain undefended.

6.2 Systematization Scope

Microarchitectural Features in Scope. We limit our systematization efforts to works that discuss attacks and defenses concerning microarchitectural features speculating on future memory accesses or control flow. More specifically, we consider hardware prefetching attacks (as discussed in previous chapters) as well as variants of Spectre [79]. We exclude attacks on trusted execution environments (such as Intel SGX or ARM TrustZone) from our scope. We also exclude defense approaches based on formal verification.

Academic and Non-Academic Documents. We find that defenses, especially, are often not discussed in depth in academic literature. For this reason, we also include relevant documentation and similar publications from processor vendors, operating system maintainers, compiler developers, and browser developers. Throughout this chapter, we refer to all these information sources as *documents*. We started from the prefetching works discussed in Chapter 5 and both foundational as well as recent contributions to the field of Spectre attacks and defenses [22, 79, 129, 157]. We identified further relevant academic and non-academic works through recursive exploration of the relevant references. While our goal was comprehensive coverage, the corpus reflects the current subset of works that could be analyzed within the available time frame. Table 6.1 lists the corpus of documents that we consider for our analysis. We classify each document by its main perspective (attack-focused or defense-focused) and denote its publication type (academic or non-academic).

Table 6.1: Corpus of documents

Ref.	Title	Year	Perspective	Publication Type
[129]	Branch Privilege Injection: Compromising Spectre v2 Hardware Mitigations by Exploiting Branch Predictor Race Conditions	2025	Attack	🎓
[P3]	PreFence: A Fine-Grained and Scheduling-Aware Defense Against Prefetching-Based Attacks	2025	Defense	🎓
[54]	ShadowLoad: Injecting State into Hardware Prefetchers	2025	Attack	🎓
[28]	GoFetch: Breaking Constant-Time Cryptographic Implementations Using Data Memory-Dependent Prefetchers	2024	Attack	🎓
[156]	InSpectre Gadget: Inspecting the Residual Attack Surface of Cross-privilege Spectre v2	2024	Defense	🎓
[29]	PREFETCHX: Cross-Core Cache-Agnostic Prefetcher-Based Side-Channel Attacks	2024	Attack	🎓
[30]	AfterImage: Leaking Control Flow Data and Tracking Load Operations via the Hardware Prefetcher	2023	Attack	🎓
[P2]	FetchBench: Systematic Identification and Characterization of Proprietary Prefetchers	2023	Attack	🎓
[36]	FineIBT: Fine-grain Control-flow Enforcement with Indirect Branch Tracking	2023	Defense	🎓
[131]	Augury: Using Data Memory-Dependent Prefetchers to Leak Data at Rest	2022	Attack	🎓
[22]	Branch History Injection: On the Effectiveness of Hardware Mitigations Against Cross-Privilege Spectre-v2 Attacks	2022	Attack	🎓
[157]	RETBLEED: Arbitrary Speculative Code Execution with Return Instructions	2022	Attack	🎓
[6]	JumpSwitches: Restoring the Performance of Indirect Branches In the Era of Spectre	2019	Defense	🎓
[79]	Spectre Attacks: Exploiting Speculative Execution	2019	Attack	🎓
[93]	ret2spec: Speculative Execution Using Return Stack Buffers	2018	Attack	🎓
[81]	Spectre Returns! Speculation Attacks using the Return Stack Buffer	2018	Attack	🎓
[78]	Speculative Buffer Overflows: Attacks and Defenses	2018	Attack	🎓
[137]	Unveiling Hardware-Based Data Prefetcher, a Hidden Source of Information Leakage	2018	Attack	🎓
[124]	Raccoon: Closing Digital Side-Channels through Obfuscated Execution	2015	Defense	🎓
[71]	Branch History Injection and Intra-mode Branch Target Injection / CVE-2022-0001, CVE-2022-0002 / INTEL-SA-00598	2025	Defense	🏭
[16]	Spectre-BHB: Speculative Target Reuse Attacks	2025	Defense	🏭
[70]	Intel® 64 and IA-32 Architectures Software Developer's Manual, Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4	2024	Defense	🏭
[113]	Reptar	2024	Attack	🏭
[5]	Software Techniques for Managing Speculation on AMD Processors	2023	Defense	🏭

(continued on next page)

🎓: Academic paper
🏭: Non-academic document, such as documentation by CPU vendors or software developers

Table 6.1: Corpus of documents (continued)

Ref.	Title	Year	Perspective	Publication Type
[92]	Speculative Load Hardening — LLVM 21.0.0git documentation	2025	Defense	🏭
[4]	LFENCE/JMP Mitigation Update for CVE-2017-5715	2022	Defense	🏭
[65]	Retpoline: A Branch Target Injection Mitigation	2022	Defense	🏭
[66]	Return Stack Buffer Underflow / CVE-2022-29901, CVE-2022-28693 / INTEL-SA-00702	2022	Defense	🏭
[10]	Cache Speculation Side-channels	2020	Defense	🏭
[3]	AMD64 Technology Indirect Branch Control Extension	2018	Defense	🏭
[58]	Branch Target Injection / CVE-2017-5715 / INTEL-SA-00088	2018	Defense	🏭
[59]	Indirect Branch Predictor Barrier	2018	Defense	🏭
[60]	Indirect Branch Restricted Speculation	2018	Defense	🏭
[126]	Mitigating Spectre with Site Isolation in Chrome	2018	Defense	🏭
[101]	Mitigating speculative execution side-channel attacks in Microsoft Edge and Internet Explorer	2018	Defense	🏭
[152]	Mitigations Landing for New Class of Timing Attack	2018	Defense	🏭
[147]	Retpoline: a software construct for preventing branch-target-injection	2018	Defense	🏭
[61]	Single Thread Indirect Branch Predictors	2018	Defense	🏭
[115]	Spectre mitigations in MSVC	2018	Defense	🏭
[57]	speculative execution, variant 4: speculative store bypass	2018	Attack	🏭
[62]	Speculative Store Bypass / CVE-2018-3639 / INTEL-SA-00115	2018	Defense	🏭
[120]	What Spectre and Meltdown Mean For WebKit	2018	Defense	🏭

🎓: Academic paper
🏭: Non-academic document, such as documentation by CPU vendors or software developers

6.3 Systematization Methodology

To answer the research questions of this chapter, we systematically analyze the documents in our corpus. In particular, we collect the following information for each document:

- Metadata about the document, such as the title, authors, publisher, and the publication date
- The attacks discussed in or referenced by the document
- The defenses discussed in or referenced by the document
- The defense implementations discussed in or referenced by the document
- The processor models used for evaluation
- Mitigation relations: Whether a defense mitigates (or does not mitigate) an attack (according to the document)

- Vulnerability relations: Whether a specific processor model is vulnerable (or not vulnerable) to an attack (according to the document)
- Whether the document discusses reverse-engineering of microarchitectural features

Scopes of Attacks and Defenses. For each attack, we further determine in which of the following scopes they are able to operate:

- **Same-address-space:** The attack can be carried out within an address space (e.g., within a user-space process).
- **Cross-address-space:** The attack can be carried out across address spaces (e.g., across user-space processes)
- **Cross-privilege-level:** The attack can be carried out across privilege levels (e.g., attacking the kernel from user space)
- **Cross-SMT:** The attack can be carried out across SMT sibling cores on the same physical core
- **Cross-Core:** The attack can be carried out across physical processor cores

We use the same scopes to classify defenses. For example, a defense is assigned the *same-address-space* scope when it mitigates same-address-space attacks.

In comparison to our previous discussion on the scopes of prefetching-based attacks in Section 5.3.2, we add the cross-SMT and cross-core scopes, which allow us to include the dimension of the processor-core privilege boundaries in this analysis. We remove the trusted-execution scope because we excluded such attacks from this study (see Section 6.2).

We will use the scopes of attacks and defenses to assess whether an attack can (theoretically) be fully mitigated by the applicable defenses. An attack is considered mitigated if at least one defense exists in every scope that the attack is applicable to. We perform this analysis in Section 6.4.2.

Properties of Defense Implementations. For each defense implementation, we also register its maturity by distinguishing between prototypes and production-ready implementations. In addition, we record whether the source code of an implementation is available or not. This information allows us to better assess the practical relevance of a defense implementation.

Data-Driven Approach. We store our dataset in a relational database to make it easily accessible for further analysis. In particular, we set up a MariaDB database [95] with 13 tables that represent the objects we deal with (such as *documents*, *attacks*, *defenses*, *implementations*, and *processors*), as well as relevant *n*:*m* relations between them (such as *document describes defense*, *defense mitigates attack*, or *processor vulnerable to attack*). We use SQL queries to aggregate and filter the data in those tables, allowing us to reason about it. We provide this database and all queries used to generate the figures and tables in this chapter as an artifact along with this work (see Section 1.2).

6.4 Systematization Results

We now answer our research questions with the help of our dataset derived from the document corpus, which contains 42 documents, 19 from academia and 23 from non-academic sources. Of the 19 papers from academia, we classify 14 documents as attack-focused, and 5 as defense-focused. 13 academic papers perform reverse-engineering of microarchitectural features before making their offensive or defensive contribution. We identify 17 distinct attacks (listed in Table A.5 in the appendix) and 61 distinct defenses (listed in Table A.6 in the appendix) that are proposed in or referenced by the documents in our corpus.

6.4.1 Architecture Gaps

To answer our first research question (RQ4.1), we focus on the subset of academic papers in our document corpus and examine which processor models and architectures they consider. For each research paper, we count the number of distinct processor models and manufacturers they use in their evaluation. We report the results in Figure 6.1.

Results. We find that most papers only consider one or two distinct processor models. Only 26% of the papers evaluate their results on more than four different models. 16% do not specify a processor model at all. Most papers consider only a single processor vendor, only 37% consider two or more. The most commonly considered vendor, by a large margin, is Intel, followed by AMD.

Discussion. Our results indicate a bias towards x86_64 platforms in the academic papers in our dataset. In addition, we find that the results of research papers are often not scaled up to cover many different microarchitectures, despite the fact that the number of different microarchitectures per processor vendor is large and grows with every new product generation. For instance, Intel's *Intel Affected Processor List* [73] lists 72 microarchitectures as of October 2025. 50 of those microarchitectures are not covered by any of the papers in our document corpus. Thus, every paper can be seen as a spotlight on a particular microarchitecture, and in total, those spotlights cover only a fraction of the processors on the market, leaving a great fraction unexplored.

In practice, the task of scaling the tests for vulnerabilities in microarchitectural features is left to the processor manufacturers, as only they have the capabilities to do so. On the one hand, this is reasonable, as the designers are in the best position to assess and ensure the security of their own products. On the other hand, manufacturers are naturally biased and incentivized to keep public discussion of security vulnerabilities to a minimum.

Therefore, independent analysis of microarchitectural security is necessary to ensure the security of the foundation of our computer systems, but neither manufacturers nor researchers can fully provide it. One solution to this problem could be an independent third party that acts as a *processor observatory*, which is equipped with generic and scalable analysis tools and responsible for scaling research results. Developing the tooling for such an institution can be subject of future research.

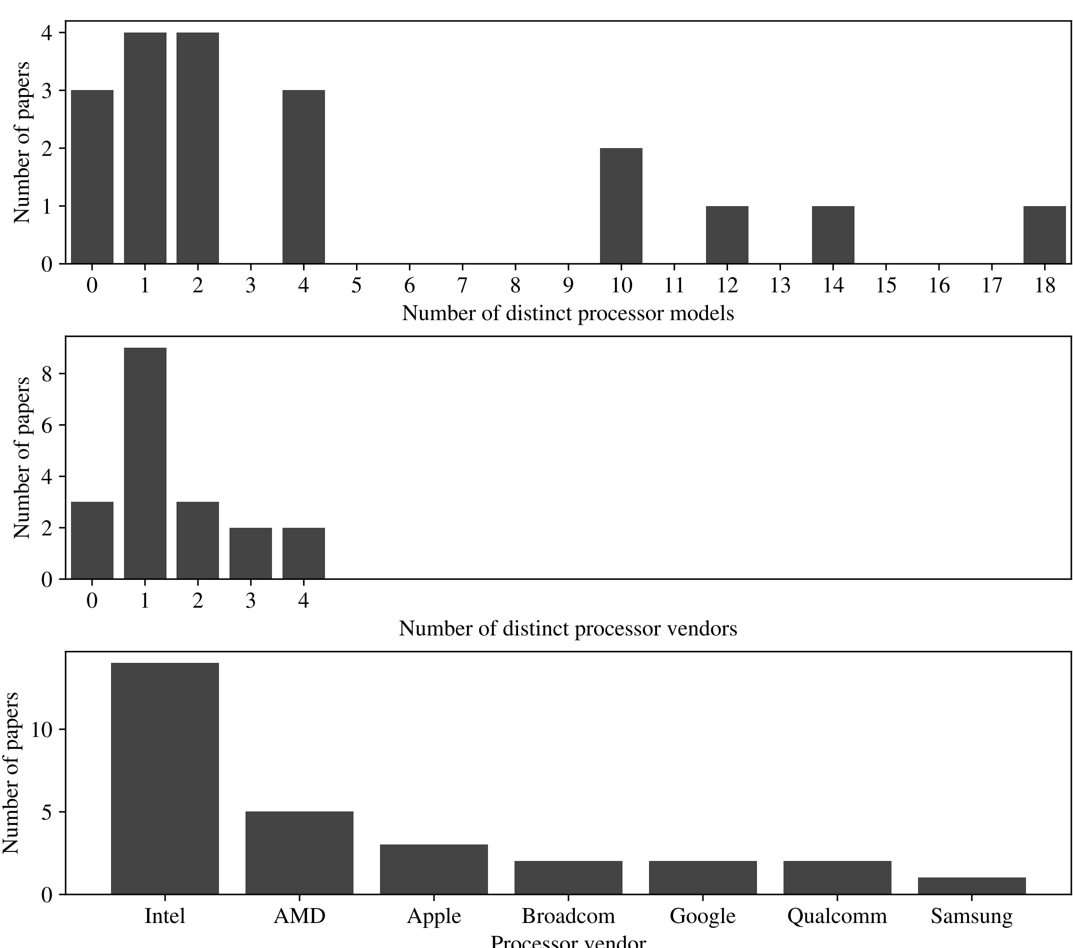

Figure 6.1: Distinct processors and processor vendors per paper. Most papers evaluate less than 3 processor models, and most focus on a single processor vendor. Intel processors are considered the most.

6.4.2 Mitigation Gaps

In this section, we answer research question RQ4.2 and evaluate whether each of the attacks in our dataset can be fully mitigated in theory.

Approach: Scope Matching Analysis. In order to be considered *fully mitigated in theory*, an attack must be mitigated by at least one applicable defense in all of the scopes that it is vulnerable in (see Section 6.3 for an explanation of the scopes). Our goal is to find whether all attacks in our dataset fulfill this requirement. To perform this analysis, we first compile a list of all attacks and assign them the scopes they can be carried out in. We report this list in Table A.5 in the appendix. We also assign scopes to each of the defenses, as listed in Table A.6 in the appendix.

We then use the mitigation relations in our dataset that we extracted from the documents in our corpus. These relations express which attacks can be mitigated by which defenses (according to the literature) and whether there are any limitations (e.g.,

Attack	**Scopes**				
	same-addr.-space	cross-addr.-space	cross-privilege	cross-SMT	cross-core
Instruction address aliasing causes prefetcher state confusion	●	●	●	○	○

●: Attack applicable to scope ○: Attack not applicable to scope

1. Match defenses that mitigate the attack ↓

Defense	**Scopes**				
Cache/State Partitioning along privilege levels	○	○	●	○	○
Cache/State Partitioning along process IDs	○	●	●	○	○
Clear prefetcher state on context switch	○	●	○	○	○
Disable prefetcher permanently	●	●	●	●	●
Disable prefetcher selectively (PreFence)	○	●	○	●	●

●: Defense mitigates attacks in scope ○: Defense does not mitigate attacks in scope

2. Count defenses in applicable scopes ↓Σ ↓Σ ↓Σ ↓× ↓×

Attack	**Scopes**				
Instruction address aliasing causes prefetcher state confusion	● 1	● 4	● 3	—	—

● N: Attack mitigated (by N defenses) ○ 0: Attack not mitigated by any defense
—: Attack not applicable to scope

Figure 6.2: Scope matching analysis example. For each attack, the applicable defenses in each scope are counted to find whether or not the attack can be mitigated in all relevant scopes.

being an incomplete countermeasure that makes attacks harder, but not impossible). For each scope that an attack is vulnerable in, we count the number of applicable defenses that mitigate the attack in that scope without any limitations. If we find at least one defense in each scope, we consider the attack fully mitigated in theory.

Example. We illustrate this approach using one attack as an example. Specifically, we consider the prefetching-based attack from Section 4.4.1, which we classify as *Instruction address aliasing causes prefetcher state confusion* in our dataset. As we show in Figure 6.2, this attack is applicable in all scopes except *cross-SMT* and *cross-core*. This is based on the observation that the known prefetching-based attacks only work on the same logical core, as also reported by Hetterich et al. [54]. Cross-address-space attacks and cross-privilege attacks are possible though, as we have shown in Section 4.4.1 and Section 4.4.2, respectively.

Using our dataset, we identify the defenses listed in the middle table of Figure 6.2 as matching mitigations. We then count the available defenses in each scope and report the final counts in the bottom table of Figure 6.2. Because we are able to identify at

least one defense per scope, we conclude that this attack can, theoretically, be fully mitigated with the defenses in our dataset.

Results. We perform this analysis for all attacks and present the results in Table 6.2. We find only one attack to be insufficiently mitigated: *Retbleed on AMD*. This attack exploits that some AMD processors may fall back to using the Branch Target Buffer (BTB) to predict the target of a return (`ret`) instruction (instead of the Return Stack Buffer (RSB)). More precisely, when both the BTB and the RSB contain a valid entry for a given `ret` instruction, the BTB entry takes precedence. This makes `ret` instructions susceptible to malicious mistraining. The assumption that `ret` instructions are not predicted through the BTB is also fundamental for the *retpoline* defense. Thus, this behavior invalidates retpolines on these processors [157].

The proposed defenses against this vulnerability are:

- *Single Thread Indirect Branch Predictors (STIBP)*, which only prevents the attack in a cross-SMT scenario,
- *Disable SMT*, which also only mitigates cross-SMT attacks, and
- *jmp2ret*, which is a defense proposal of AMD to prevent Retbleed attacks on the kernel from user space.

Specifically, none of those defenses protect against the attack in the same-address-space scope, leaving the affected processors vulnerable.

We note that the identified mitigation gap does not apply to the similar attack *Retbleed on Intel*. The reason is that this vulnerability is harder to exploit on vulnerable Intel processors, as the BTB is only consulted when an RSB underflow occurs. This condition can be prevented by the defense *RSB stuffing when deep call stacks occur*, which can, in principle, also be applied in a same-address-space scenario.

Discussion. We are encouraged by the fact that almost all attacks in our dataset can theoretically be mitigated. Furthermore, the only attack with incomplete defenses is vulnerable only in the same-address-space scope, which is often hard to exploit in practice.

However, it is worth considering that many of the highly effective defenses in our dataset, which are applicable to many scopes, also tend to be radical and performance-intensive. This is already apparent from the example in Figure 6.2, where the most widely applicable defense is the trivial solution of disabling the microarchitectural feature permanently. We therefore propose to extend this analysis in the future and also evaluate defenses in the dimension of performance cost. Assessing the cost of a defense in generality might be difficult, as it depends on many context-specific factors. However, for the purpose of the analysis, it might be sufficient to rank defense on a coarse-grained qualitative cost scale (such as *low*, *moderate*, *high*, *very high*) based on the expected performance impact in a typical scenario.

6.4.3 Implementation Gaps

To evaluate also the *practical completeness* of defenses and address RQ4.3, we now take implementations of defenses into account. We perform this analysis again based on

Table 6.2: Results of the scope matching analysis

Attack	**Scopes**				
	same-addr.-space	cross-addr.-space	cross-privilege	cross-SMT	cross-core
Branch Privilege Injection (BPI, Branch Predictor Race Conditions, BPRC)	● 4	● 4	● 4	● 3	—
Branch-dependent cache attack	● 1	● 1	● 1	● 1	● 1
DMP (array-of-)pointer prefetcher exploitation	● 4	● 6	—	—	—
Instruction address aliasing causes prefetcher state confusion	● 1	● 4	● 3	—	—
Leaking single loads through prefetcher state (FetchProbe)	● 3	● 5	● 5	—	—
Prefetcher attack on shared memory/library	● 1	● 2	● 1	● 2	● 2
Prime+Probe-like prefetcher attack (e.g., PrefetchX-evict)	● 1	● 3	● 2	● 3	● 2
Reptar (Instruction Decoding Bug on Intel CPUs)	● 1	● 1	● 1	● 1	—
Retbleed on AMD	○ 0	● 1	● 1	● 2	—
Retbleed on Intel (Return Stack Buffer Underflow)	● 1	● 1	● 2	—	—
Spectre-BHB (Branch History Injection (BHI), Intra-Mode Branch Target Injection (Intra-Mode BTI))	● 3	● 5	● 6	● 2	—
Spectre-BTB (variant 2, branch target injection, Spectre-BTI)	● 5	● 6	● 5	● 5	—
Spectre-PHT (variant 1, input validation bypass)	● 5	● 4	● 4	● 4	—
Spectre-PHT variant 1.1 (bounds check bypass on stores, BCBS, speculative buffer overflow)	● 4	● 4	● 4	● 3	—
Spectre-RSB	● 7	● 7	● 7	—	—
Spectre-STL (variant 4, Spectre-SSB, speculative store bypass)	● 5	—	—	—	—
Use prefetcher to trigger attacker-controlled prefetch in victim context (ShadowLoad)	● 2	● 4	● 4	—	—

● N: Attack mitigated (by N defenses) ○ 0: Attack not mitigated by any defense
—: Attack not applicable to scope

our dataset over the document corpus. Whenever a document in the corpus references a concrete implementation of a defense, we add it to our dataset and link it to the respective defense. This allows us to investigate which defenses are implemented in practice and to reason about their properties. Our dataset includes hardware-based, software-based, and microcode-based implementations.

Results. We report the implementations we identified, along with the defenses they implement, in Table A.7 in the appendix. Specifically, we found references to 40 implementations in total. Out of the 61 defenses, only 30 defenses have at least one implementation assigned to them in our dataset. 31 defenses have no implementation assigned, and 8 defenses have more then one implementation assigned. We find that 3 of the implementations are private research prototypes and therefore not publicly available. 17 implementations are provided as open-source software.

Discussion. It is positive to see a relatively low number of private research prototypes, which is likely connected to the academic security conferences pushing for the publication of artifacts. The number of closed-source implementations is still high; a significant reason for this is that many of the implementations rely on modifications to processor hardware or microcode, which are traditionally kept private.

At first glance, the results suggest that approximately half of the defenses in our dataset have not been implemented. However, we may have reached the limits of our methodology here. We note that some defenses are not implemented in one central place that could be referenced by a document; rather, they are abstract defensive concepts. This applies to defenses such as *constant-time programming* or *blinding*: Developers need to apply them to many different code bases, such as security-critical libraries and application software. Therefore, documents are more likely to mention the mere concept without referencing any implementations. Related to this, we note that many research papers discuss defenses on a high level and therefore tend to discuss concepts rather than concrete implementations anyway.

We therefore conclude that we need to further extend our dataset in order to perform a representative analysis of the prevalence and adoption of defenses in real-world systems. Specifically, we need to add more information sources that are more likely to discuss implementations in the first place. For example, we could systematically search for references to the defenses in our dataset on open-source code sharing platforms like GitHub. Similarly, we could explicitly scan mailing-list archives and documentation pages of open-source projects for mentions of defense approaches, and in general try to identify more non-academic sources that reference defense implementations. Finally, expert interviews could be conducted. Once we reached a more accurate representation of defense implementations in the real world, further analyses can be performed. For instance, we can attempt to identify properties that make defenses more or less likely to be implemented. However, due to the limited data basis, we have to leave further analysis to future work.

6.5 Related Systematization Work

We are not the first to systematize microarchitectural side-channel attacks and defenses; however, prior work approached the topic from different angles.

Canella et al. [25] give a concise overview of the state of Spectre- and Meltdown-style attacks in 2019. They introduce a taxonomy, classifying Spectre attacks based on the exploited microarchitectural feature and a mistraining strategy that partially overlaps with our notion of attack and defense scopes. Meltdown attacks are classified based on the exception type that causes transient execution. Based on their classification, the authors identify blind spots and uncover previously unknown attack variants.

Holtryd, Manivannan, and Stenström [56] systematize attacks on microarchitectural features and corresponding defenses. Besides variants of Spectre and Meltdown, they also consider prefetching and computational simplification. For their systematization, they divide microarchitectural attacks into five generic steps and identify four common root causes for the microarchitectural vulnerabilities under investigation. The authors further use those root causes to classify defenses. Finally, they extrapolate from their results and propose defenses against the perils of *value prediction*, a predictive mechanism that had not yet been identified in a commercial processor at the time this paper was published. In contrast to our work, no implementations are considered.

Cauligi et al. [27] take a more formal approach compared to our work. They classify and compare different formal modeling approaches that have been proposed to verify that software is not vulnerable to Spectre attacks, which we exclude from the scope of our analysis.

Naseredini et al. [104] focus on the prevalence of defensive mechanisms in execution environments of programming languages and analyze whether such environments implement defenses against Spectre-type vulnerabilities. They find that out of 42 compilers, interpreters, and runtime environments, only 16 implement mitigations against at least one Spectre variant.

6.6 Discussion and Conclusion on the Systematization

Discussion: On the Use of LLMs. We experimented with using large language models (LLMs) to automatically extract information from the documents and populate our dataset, but ultimately decided against this approach. Specifically, we ran experiments with several OpenAI models, including GPT-4o, GPT-4.1, and GPT-5. Our prompts contained the schema and the current contents of our database in JSON format, along with instructions to produce similar records representing the content of a new paper (provided in PDF format). However, we found the results to be unreliable due to hallucinations. For example, the models frequently mapped new attacks to defenses that were not mentioned in the provided paper. Because we could not rely on the model's output, we continued to fill the database manually.

Conclusion. In this chapter, we addressed research question RQ4: *How can we systematically identify gaps in the academic and non-academic consideration of defenses against microarchitectural attacks?* Specifically, we conducted a data-driven analysis based on 42 academic and non-academic documents to analyze three different gaps: the architecture, mitigation, and implementation gaps.

As far as architectures are concerned, we identified a bias towards the x86_64 architecture and specifically Intel processors in the research papers in our document corpus. We further extracted information on attacks on microarchitectural features

and corresponding defenses from the documents in our corpus. We assigned attacks and defenses to the scopes that they operate in and used those scopes to evaluate the theoretical completeness of defenses against each of the attacks under investigation. We found that all but one of the attacks can be mitigated, in theory, with the defenses we consider. Finally, we considered defense implementations that are referenced in the documents in our corpus. We found references to implementations of about half of the defenses in our dataset. Of those implementations, about 43% are provided as open-source software and about 8% are private research prototypes that are not available to the public. However, we also note the need for future extension of our dataset, especially with respect to defense implementations. Because our data sources are based on documents that are either academic documents or referenced in academic documents, we hypothesize that especially implementations of abstract defense approaches are not yet sufficiently represented in our dataset. To improve upon this situation, we propose to add more diverse data sources in the future.

7
Conclusion

In this thesis, we examined the security of microarchitectural features from three different angles: We *characterized* features by generalizing from suspicious code samples or through targeted software-based tests; we *exploited* microarchitectural side channels in case studies to show that control flow or data can be leaked; and we proposed and systematized *protective measures* that mitigate certain (classes of) side-channel attacks.

7.1 Summary of Contributions

We started this thesis by tackling a side-channel characterization problem. Specifically, in Chapter 3, we addressed the challenge of generalizing from a single code example that shows microarchitectural side-channel leakage to an abstract characterization of that side channel (RQ1). We introduced *Leakage Templates*, a way to abstractly describe side-channel leakage in program code, and presented *Plumber*, our framework to facilitate the derivation of such Templates. In case studies, we derived Leakage Templates for previction and prefetching behavior of an ARM Cortex-A53 core and demonstrated how these behaviors can be exploited as leakage channels. In addition, we showed that Leakage Templates are actually useful: They can be used to identify instances of the side channel they describe in other programs. We demonstrated this by re-identifying a prefetching-based side-channel vulnerability in the OpenSSL cryptography library by means of static and dynamic program analysis.

In Chapter 4, we introduced a novel methodology to identify and characterize prefetcher implementations and their security properties in real-world processors (RQ2). To this end, we first developed a taxonomy that systematizes relevant prefetching approaches. Then, we designed and implemented *FetchBench*, our extensible framework to identify and characterize implementations of those prefetching approaches on real hardware using software-based and prefetcher-specific tests. We analyzed 19 CPUs from seven vendors with our tool and reported the results. Most notably, we identified, for the first time, an implementation of the Spatial Memory Streaming (SMS) prefetcher in an ARM Cortex-A72 core. In subsequent case studies, we showed that this prefetcher can be exploited to leak AES key material across the process isolation boundary. In addition, we showed that the underlying side channel even exists between secure world (TrustZone) and normal world, breaking fundamental security guarantees of the platform.

Next, in Chapter 5, we shifted to the defensive perspective and studied how we can defend against prefetching-based side channels (RQ3). We started by analyzing existing prefetching attacks for common building blocks and, most notably, identified that all attacks rely on some form of prefetcher training within the victim process. Based on this insight, we designed and implemented *PreFence*, our novel defense against user-space prefetching-based attacks. Our general idea is to empower the victim process to stop the prefetcher from learning secret-dependent patterns. To this end, we introduced a new system call that processes can use to flexibly disable the prefetcher while performing security-critical operations. We evaluated PreFence on two different processor architectures (x86_64 and ARMv8-A 64-bit) and found that it successfully prevents prior-work attacks. The performance overhead on non-security-critical processes is low, and processes that use our countermeasure run faster compared to the trivially secure solution of disabling the prefetcher permanently.

Finally, in Chapter 6, we systematically identified gaps in the consideration of defenses against microarchitectural attacks in academic and non-academic literature (RQ4). We started by analyzing said literature and creating a dataset of microarchitectural attacks, defenses, their relations, and additional properties, as represented in those documents. Based on this dataset, we investigated the architecture, mitigation, and implementation gaps. We identified a bias of the academic papers in our corpus towards the x86_64 architecture and specifically Intel processors and found that all but one of the discussed attacks can, in theory, be fully prevented by the discussed defenses. As far as implementations are concerned, the examined documents provide limited information: We only found references to concrete implementations for about half of the defenses in the dataset. This result calls for future expansion of our dataset.

7.2 Future Directions

The discovery, characterization, and exploitation of side channels caused by microarchitectural features will remain a relevant topic in cybersecurity research. While our systematic characterization approaches ease the process for some classes of side channels and uncovered new side-channel vulnerabilities, they still suffer from inherent limitations: As long as researchers have to treat processors as black boxes and have to reverse-engineer their internals bit by bit, the search space for side channels is huge, and every approach trying to identify and characterize them has to specialize on a sub-space to keep the problem manageable. We anticipate, however, that our methodologies could be extended to characterize other microarchitectural components and thus cover at least a greater fraction of the search space. For instance, the Plumber and FetchBench frameworks could be extended to characterize branch predictors, internal buffers, or other features. Determining the required extensions to Plumber's domain-specific language or designing corresponding tests for FetchBench can be subject of future work.

From a defensive perspective, we anticipate that a desirable future path is the generalization of defenses. The current landscape of defenses against microarchitectural attacks is a convoluted patchwork resulting from years of exchanges between the offensive and the defensive side. Whenever new side channels were discovered, existing defenses had to be adapted or new ones developed. The constant influx of new findings meant that the goalposts for defensive measures were constantly shifting, resulting in a confusing array of spot fixes.

Recently, Intel and ARM introduced the DOITM and DIT bits, respectively, to their architectures [12, 72]. These bits are to be set whenever application code relies on the constant-time behavior of certain instructions. This step marks an open shift by processor manufacturers towards a *fast-by-default, secure-on-demand* paradigm—the same approach we take with our PreFence defense, albeit in the prefetching domain.

Currently, PreFence enables processes to disable prefetching during security-critical computations, thereby preventing prefetcher training. This concept can be extended to other microarchitectural features. Ideally, a generalized version would disable all microarchitectural features that compromise security, such as any predictors and any variable timing behavior. Applications requiring the highest security standard could even consider disabling memory caching altogether in certain (minimal) code paths.

However, a follow-up research project would be required to determine where minimal security-critical code regions begin and end in that case. This analysis needs to take the timing behavior of instructions (in normal and DOIT/DIT mode) and the impact of instructions on various predictors into account. More coarse-grained system call placement might be required.

As indicated above, our systematization work can be extended in various ways. Besides adding more academic and non-academic data sources, enriching the dataset with more attributes can lead to additional insights. For instance, defenses could be rated based on their performance cost or the manual effort involved. Having such data available would allow us, for example, to derive recommendations for the *most convenient* or *most performant* defenses against specific attacks, which could guide developers and users of affected systems.

Bibliography

Author's Papers for This Thesis

[P1] Ibrahim, A., Nemati, H., Schlüter, T., Tippenhauer, N. O., and Rossow, C. "Microarchitectural Leakage Templates and Their Application to Cache-Based Side Channels." In: *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. 2022.

[P2] Schlüter, T., Choudhari, A., Hetterich, L., Trampert, L., Nemati, H., Ibrahim, A., Schwarz, M., Rossow, C., and Tippenhauer, N. O. "FetchBench: Systematic Identification and Characterization of Proprietary Prefetchers." In: *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. 2023.

[P3] Schlüter, T. and Tippenhauer, N. O. "PreFence: A Fine-Grained and Scheduling-Aware Defense Against Prefetching-Based Attacks." In: *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P)*. 2025.

[P4] Schlüter, T. and Tippenhauer, N. O. "SoK: Countermeasures Against Attacks on Microarchitectural Features." In preparation.

Other Papers of the Author

[S1] Schlüter, T. and Lemke-Rust, K. "Differential Analysis and Fingerprinting of ZombieLoads on Block Ciphers." In: *Proceedings of the International Conference on Smart Card Research and Advanced Applications (CARDIS)*. 2020.

[S2] Lin, Y.-D., Schlüter, T., Narimani, P., Abbasi, A., and Tippenhauer, N. O. "PowerFuzz: Efficient Power Side-Channel Guided Fuzzing of Proprietary Embedded Systems with Execution Divergence Graphs." In preparation.

Other References

[1] Acıiçmez, O. and Koç, Ç. K. "Trace-Driven Cache Attacks on AES." In: *Proceedings of the International Conference on Information and Communications Security (ICICS)*. 2006.

[2] AMD Inc. “A New x86 Core Architecture for the Next Generation of Computing.” HotChips 28. Aug. 2016. URL: https://old.hotchips.org/wp-content/uploads/hc_archives/hc28/HC28.23-Tuesday-Epub/HC28.23.90-High-Perform-Epub/HC28.23.930-X86-core-MikeClark-AMD-final_v2-28.pdf (visited on 2025/09/18).

[3] AMD Inc. “AMD64 Technology Indirect Branch Control Extension.” 2018. URL: https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/white-papers/111006-architecture-guidelines-update-amd64-technology-indirect-branch-control-extension.pdf (visited on 2025/09/30).

[4] AMD Inc. “LFENCE/JMP Mitigation Update for CVE-2017-5715.” 2022. URL: https://www.amd.com/en/resources/product-security/bulletin/amd-sb-1036.html (visited on 2025/09/30).

[5] AMD Inc. “Software Techniques for Managing Speculation on AMD Processors.” 2023. URL: https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/software-techniques-for-managing-speculation.pdf (visited on 2025/09/30).

[6] Amit, N., Jacobs, F., and Wei, M. “JumpSwitches: Restoring the Performance of Indirect Branches In the Era of Spectre.” In: *Proceedings of the USENIX Annual Technical Conference*. 2019.

[7] Apple Inc. “Writing ARM64 Code for Apple Platforms.” 2025. URL: https://developer.apple.com/documentation/xcode/writing-arm64-code-for-apple-platforms (visited on 2025/09/18).

[8] Arm Ltd. “ARM Cortex-A72 MPCore Processor Technical Reference Manual.” Dec. 2016. URL: https://developer.arm.com/documentation/100095/0003/ (visited on 2025/09/18).

[9] Arm Ltd. “Arm Cortex-A53 MPCore Processor Technical Reference Manual.” June 2018. URL: https://developer.arm.com/documentation/ddi0500/j/ (visited on 2025/09/18).

[10] Arm Ltd. “Cache Speculation Side-channels.” 2020. URL: https://developer.arm.com/documentation/102816/0205/ (visited on 2025/09/30).

[11] Arm Ltd. “Arm® Cortex®-A65AE Core Technical Reference Manual.” Revision r1p1. Mar. 2022. URL: https://documentation-service.arm.com/static/624ff3decaabfd7b3c13e9c7 (visited on 2025/10/12).

[12] Arm Ltd. “Arm® Architecture Registers for A-profile Architecture.” 2024. URL: https://developer.arm.com/documentation/ddi0601/2024-03 (visited on 2025/09/18).

[13] Arm Ltd. “Architecture Security Advisory: Prefetcher Side Channels.” v1.1. 2025. URL: https://developer.arm.com/documentation/109504/0101/ (visited on 2025/09/18).

[14] Arm Ltd. “Arm® A64 Instruction Set for A-profile Architecture.” June 2025. URL: `https://developer.arm.com/documentation/ddi0602/2025-06/` (visited on 2025/09/19).

[15] Arm Ltd. “Arm® Architecture Reference Manual for A-profile Architecture.” Revision L.b. Apr. 2025. URL: `https://developer.arm.com/documentation/ddi0487/latest/` (visited on 2025/09/18).

[16] Arm Ltd. “Spectre-BHB: Speculative Target Reuse Attacks.” 2025. URL: `https://developer.arm.com/documentation/102898/0108/` (visited on 2025/09/30).

[17] Arm Ltd. “TrustZone for Cortex-A.” 2025. URL: `https://www.arm.com/technologies/trustzone-for-cortex-a` (visited on 2025/09/18).

[18] asmregex developers. “asmregex.” URL: `https://github.com/Usibre/asmregex/` (visited on 2025/09/18).

[19] Awal, M. S. and Rahman, M. T. “Disassembling Software Instruction Types through Impedance Side-channel Analysis.” In: *Proceedings of the IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*. 2023.

[20] Ayers, G., Litz, H., Kozyrakis, C., and Ranganathan, P. “Classifying Memory Access Patterns for Prefetching.” In: *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2020.

[21] Baer, J.-L. and Chen, T.-F. “An Effective On-Chip Preloading Scheme to Reduce Data Access Penalty.” In: *Proceedings of the ACM/IEEE Conference on Supercomputing (SC)*. 1991.

[22] Barberis, E., Frigo, P., Muench, M., Bos, H., and Giuffrida, C. “Branch History Injection: On the Effectiveness of Hardware Mitigations Against Cross-Privilege Spectre-v2 Attacks.” In: *Proceedings of the USENIX Security Symposium*. 2022.

[23] Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., Hestness, J., Hower, D. R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M. D., and Wood, D. A. “The Gem5 Simulator.” *ACM SIGARCH Computer Architecture News* 39, 2 (Aug. 2011), 1–7.

[24] Camurati, G., Poeplau, S., Muench, M., Hayes, T., and Francillon, A. “Screaming Channels: When Electromagnetic Side Channels Meet Radio Transceivers.” In: *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. 2018.

[25] Canella, C., Van Bulck, J., Schwarz, M., Lipp, M., Berg, B. von, Ortner, P., Piessens, F., Evtyushkin, D., and Gruss, D. “A Systematic Evaluation of Transient Execution Attacks and Defenses.” In: *Proceedings of the USENIX Security Symposium*. 2019.

[26] Cauligi, S., Disselkoen, C., Gleissenthall, K. V., Tullsen, D., Stefan, D., Rezk, T., and Barthe, G. “Constant-Time Foundations for the New Spectre Era.” In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2020.

[27] Cauligi, S., Disselkoen, C., Moghimi, D., Barthe, G., and Stefan, D. "SoK: Practical Foundations for Software Spectre Defenses." In: *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. 2022.

[28] Chen, B., Wang, Y., Shome, P., Fletcher, C. W., Kohlbrenner, D., Paccagnella, R., and Genkin, D. "GoFetch: Breaking Constant-Time Cryptographic Implementations Using Data Memory-Dependent Prefetchers." In: *Proceedings of the USENIX Security Symposium*. 2024.

[29] Chen, Y., Hajiabadi, A., Pei, L., and Carlson, T. E. "PREFETCHX: Cross-Core Cache-Agnostic Prefetcher-Based Side-Channel Attacks." In: *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2024.

[30] Chen, Y., Pei, L., and Carlson, T. E. "AfterImage: Leaking Control Flow Data and Tracking Load Operations via the Hardware Prefetcher." In: *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2023.

[31] Chromium Project. "Mitigating Side-Channel Attacks." 2018. URL: `https://www.chromium.org/Home/chromium-security/ssca/` (visited on 2025/09/18).

[32] Cronin, P. and Yang, C. "A Fetching Tale: Covert Communication with the Hardware Prefetcher." In: *Proceedings of the IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*. 2019.

[33] Didier, G., Maurice, C., Geimer, A., and Ghandour, W. J. "Characterizing Prefetchers using CacheObserver." In: *Proceedings of the IEEE International Symposium on Computer Architecture and High-Performance Computing (SBAC-PAD)*. 2022.

[34] Electronic Frontier Foundation. "Frequently Asked Questions (FAQ) About the Electronic Frontier Foundation's 'DES Cracker' Machine." 1998. URL: `https://w2.eff.org/Privacy/Crypto/Crypto_misc/DESCracker/HTML/19980716_eff_des_faq.html` (visited on 2025/09/18).

[35] Falsafi, B. and Wenisch, T. F. *A Primer on Hardware Prefetching*. Synthesis Lectures Computer Architecture 28. Morgan & Claypool, 2014.

[36] Gaidis, A. J., Moreira, J., Sun, K., Milburn, A., Atlidakis, V., and Kemerlis, V. P. "FineIBT: Fine-grain Control-flow Enforcement with Indirect Branch Tracking." In: *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. 2023.

[37] Gandolfi, K., Mourtel, C., and Olivier, F. "Electromagnetic Analysis: Concrete Results." In: *Proceedings of the Conference on Cryptographic Hardware and Embedded Systems (CHES)*. 2001.

[38] GDB developers. "GDB: The GNU Project Debugger." URL: `https://sourceware.org/gdb/` (visited on 2025/09/18).

[39] Gerlach, L., Weber, D., Zhang, R., and Schwarz, M. "A Security RISC: Microarchitectural Attacks on Hardware RISC-V CPUs." In: *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. 2023.

[40] Grannaes, M., Jahre, M., and Natvig, L. "Multi-Level Hardware Prefetching Using Low Complexity Delta Correlating Prediction Tables with Partial Matching." In: *Proceedings of the International Conference on High Performance Embedded Architectures and Compilers (HiPEAC)*. 2010.

[41] Gras, B., Giuffrida, C., Kurth, M., Bos, H., and Razavi, K. "ABSynthe: Automatic Blackbox Side-Channel Synthesis on Commodity Microarchitectures." In: *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. 2020.

[42] Gruss, D., Maurice, C., Fogh, A., Lipp, M., and Mangard, S. "Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR." In: *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. 2016.

[43] Gruss, D., Maurice, C., and Mangard, S. "Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript." In: *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. 2016.

[44] Gruss, D., Maurice, C., Wagner, K., and Mangard, S. "Flush+Flush: A Fast and Stealthy Cache Attack." In: *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. 2016.

[45] Guanciale, R., Nemati, H., Baumann, C., and Dam, M. "Cache Storage Channels: Alias-Driven Attacks and Verified Countermeasures." In: *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. 2016.

[46] Guarnieri, M., Köpf, B., Morales, J. F., Reineke, J., and Sánchez, A. "Spectector: Principled Detection of Speculative Information Flows." In: *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. 2020.

[47] Guarnieri, M., Köpf, B., Reineke, J., and Vila, P. "Hardware-Software Contracts for Secure Speculation." In: *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. 2021.

[48] Gulmezoglu, B., Moghimi, A., Eisenbarth, T., and Sunar, B. "FortuneTeller: Predicting Microarchitectural Attacks via Unsupervised Deep Learning." 2019. arXiv: `1907.03651 [cs.CR]`.

[49] Guo, Y., Zigerelli, A., Zhang, Y., and Yang, J. "Adversarial Prefetch: New Cross-Core Cache Side Channel Attacks." In: *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. 2022.

[50] Han, Y., Chan, M., Tippenhauer, N. O., Aref, Z., and Zonouz, S. "Hiding in Plain Sight? On the Efficacy of Power Side Channel-Based Control Flow Monitoring." In: *Proceedings of the USENIX Security Symposium*. 2022.

[51] Han, Y., Etigowni, S., Liu, H., Zonouz, S., and Petropulu, A. "Watch Me, but Don't Touch Me! Contactless Control Flow Monitoring via Electromagnetic Emanations." In: *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. 2017.

[52] Harris, S. L. and Harris, D. M. *Digital Design and Computer Architecture: ARM® Edition*. Morgan Kaufmann, 2016.

[53] Hennessy, J. L. and Patterson, D. A. *Computer Architecture: A Quantitative Approach*. 6th ed. Morgan Kaufmann, 2017.

[54] Hetterich, L., Thomas, F., Gerlach, L., Zhang, R., Bernsdorf, N., Ebert, E., and Schwarz, M. "ShadowLoad: Injecting State into Hardware Prefetchers." In: *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2025.

[55] Hofmann, J., Vannacci, E., Fournet, C., Köpf, B., and Oleksenko, O. "Speculation at Fault: Modeling and Testing Microarchitectural Leakage of CPU Exceptions." In: *Proceedings of the USENIX Security Symposium*. 2023.

[56] Holtryd, N. R., Manivannan, M., and Stenström, P. "SoK: Analysis of Root Causes and Defense Strategies for Attacks on Microarchitectural Optimizations." In: *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P)*. 2023.

[57] Horn, J. "speculative execution, variant 4: speculative store bypass." 2018. URL: https://project-zero.issues.chromium.org/issues/42450580 (visited on 2025/09/18).

[58] Intel Corp. "Branch Target Injection / CVE-2017-5715 / INTEL-SA-00088." 2018. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/branch-target-injection.html (visited on 2025/09/30).

[59] Intel Corp. "Indirect Branch Predictor Barrier." 2018. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/indirect-branch-predictor-barrier.html (visited on 2025/09/30).

[60] Intel Corp. "Indirect Branch Restricted Speculation." 2018. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/indirect-branch-restricted-speculation.html (visited on 2025/09/30).

[61] Intel Corp. "Single Thread Indirect Branch Predictors." 2018. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/single-thread-indirect-branch-predictors.html (visited on 2025/09/30).

[62] Intel Corp. “Speculative Store Bypass / CVE-2018-3639 / INTEL-SA-00115.” 2018. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/speculative-store-bypass.html (visited on 2025/09/30).

[63] Intel Corp. “Data Dependent Prefetcher.” Nov. 2022. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/data-dependent-prefetcher.html (visited on 2025/09/18).

[64] Intel Corp. “Guidelines for Mitigating Timing Side Channels Against Cryptographic Implementations.” June 2022. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/secure-coding/mitigate-timing-side-channel-crypto-implementation.html (visited on 2025/09/18).

[65] Intel Corp. “Retpoline: A Branch Target Injection Mitigation.” 2022. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/retpoline-branch-target-injection-mitigation.html (visited on 2025/09/30).

[66] Intel Corp. “Return Stack Buffer Underflow / CVE-2022-29901, CVE-2022-28693 / INTEL-SA-00702.” 2022. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/return-stack-buffer-underflow.html (visited on 2025/09/30).

[67] Intel Corp. “Hardware LLC Prefetch Feature on 4th Gen Intel® Xeon® Scalable Processor (Codename Sapphire Rapids).” June 2023. URL: https://www.intel.com/content/www/us/en/content-details/780991/ (visited on 2025/09/18).

[68] Intel Corp. “Hardware Prefetch Control for Intel® Atom® Cores.” Whitepaper. Dec. 2023. URL: https://cdrdv2-public.intel.com/795247/357930-Hardware-Prefetch-Controls-for-Intel-Atom-Cores.pdf (visited on 2025/09/18).

[69] Intel Corp. “Intel 64 and IA-32 Architectures Optimization Reference Manual: Volume 1.” Apr. 2024. URL: https://cdrdv2-public.intel.com/821612/248966-Optimization-Reference-Manual-V1-050.pdf (visited on 2025/09/18).

[70] Intel Corp. “Intel® 64 and IA-32 Architectures Software Developer’s Manual, Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4.” June 2024. URL: https://cdrdv2-public.intel.com/825743/325462-sdm-vol-1-2abcd-3abcd-4.pdf (visited on 2025/10/01).

[71] Intel Corp. “Branch History Injection and Intra-mode Branch Target Injection / CVE-2022-0001, CVE-2022-0002 / INTEL-SA-00598.” 2025. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/branch-history-injection.html (visited on 2025/09/30).

[72] Intel Corp. “Data Operand Independent Timing Instruction Set Architecture (ISA) Guidance.” Sept. 2025. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/data-operand-independent-timing-isa-guidance.html (visited on 2025/09/18).

[73] Intel Corp. “Intel Affected Processor List.” 2025. URL: https://github.com/intel/Intel-affected-processor-list/blob/3a84f776ea6ea013ee1d982735df4426a50554c4/Intel_affected_processor_list.csv (visited on 2025/09/30).

[74] Irazoqui, G., Eisenbarth, T., and Sunar, B. “Systematic Reverse Engineering of Cache Slice Selection in Intel Processors.” In: *Proceedings of the Euromicro Conference on Digital System Design (DSD)*. 2015.

[75] Jin, X., Xiao, X., Jia, S., Gao, W., Gu, D., Zhang, H., Ma, S., Qian, Z., and Li, J. “Annotating, Tracking, and Protecting Cryptographic Secrets with CryptoMPK.” In: *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. 2022.

[76] Joseph, D. and Grunwald, D. “Prefetching Using Markov Predictors.” In: *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 1997.

[77] Kim, J., Chuang, J., Genkin, D., and Yarom, Y. “FLOP: Breaking the Apple M3 CPU via False Load Output Predictions.” In: *Proceedings of the USENIX Security Symposium*. 2025.

[78] Kiriansky, V. and Waldspurger, C. “Speculative Buffer Overflows: Attacks and Defenses.” 2018. arXiv: 1807.03757 [cs.CR].

[79] Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., and Yarom, Y. “Spectre Attacks: Exploiting Speculative Execution.” In: *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. 2019.

[80] Kocher, P., Jaffe, J., and Jun, B. “Differential Power Analysis.” In: *Proceedings of the Annual International Cryptology Conference (CRYPTO)*. 1999.

[81] Koruyeh, E. M., Khasawneh, K. N., Song, C., and Abu-Ghazaleh, N. “Spectre Returns! Speculation Attacks using the Return Stack Buffer.” In: *Proceedings of the USENIX WOOT Conference on Offensive Technologies*. 2018.

[82] Kou, Z., He, W., Sinha, S., and Zhang, W. “Load-Step: A Precise TrustZone Execution Control Framework for Exploring New Side-channel Attacks Like Flush+Evict.” In: *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*. 2021.

[83] Lee, Y., Min, C., and Lee, B. “ExpRace: Exploiting Kernel Races through Raising Interrupts.” In: *Proceedings of the USENIX Security Symposium*. 2021.

[84] Lighttpd Developers. "1.4.75 - Lighttpd." 2024. URL: https://www.lighttpd.net/2024/3/13/1.4.75/ (visited on 2025/09/18).

[85] Lipp, M., Gruss, D., and Schwarz, M. "AMD Prefetch Attacks through Power and Time." In: *Proceedings of the USENIX Security Symposium*. 2022.

[86] Lipp, M., Gruss, D., Spreitzer, R., Maurice, C., and Mangard, S. "ARMageddon: Cache Attacks on Mobile Devices." In: *Proceedings of the USENIX Security Symposium*. 2016.

[87] Lipp, M., Kogler, A., Oswald, D., Schwarz, M., Easdon, C., Canella, C., and Gruss, D. "PLATYPUS: Software-based Power Side-Channel Attacks on x86." In: *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. 2021.

[88] Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., and Hamburg, M. "Meltdown: Reading Kernel Memory from User Space." In: *Proceedings of the USENIX Security Symposium*. 2018.

[89] Liu, F., Ge, Q., Yarom, Y., McKeen, F., Rozas, C., Heiser, G., and Lee, R. B. "CATalyst: Defeating Last-Level Cache Side Channel Attacks in Cloud Computing." In: *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2016.

[90] Liu, F., Yarom, Y., Ge, Q., Heiser, G., and Lee, R. B. "Last-Level Cache Side-Channel Attacks are Practical." In: *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. 2015.

[91] Liu, Z., Samwel, N., Weissbart, L., Zhao, Z., Lauret, D., Batina, L., and Larson, M. "Screen Gleaning: A Screen Reading TEMPEST Attack on Mobile Devices Exploiting an Electromagnetic Side Channel." In: *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. 2021.

[92] LLVM Project. "Speculative Load Hardening — LLVM 21.0.0git documentation." 2025. URL: https://llvm.org/docs/SpeculativeLoadHardening.html (visited on 2025/09/18).

[93] Maisuradze, G. and Rossow, C. "ret2spec: Speculative Execution Using Return Stack Buffers." In: *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. 2018.

[94] Mangard, S., Oswald, E., and Popp, T. *Power Analysis Attacks*. Springer US, Boston, 2007.

[95] MariaDB Foundation. "About MariaDB Server." 2025. URL: https://mariadb.org/about/ (visited on 2025/10/12).

[96] Marr, D. T., Binns, F., Hill, D. L., Hinton, G., Koufaty, D. A., Miller, J. A., and Upton, M. "Hyper-Threading Technology Architecture and Microarchitecture." *Intel Technology Journal* 6, 1 (Feb. 2002).

[97] Maurice, C., Le Scouarnec, N., Neumann, C., Heen, O., and Francillon, A. "Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters." In: *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. 2015.

[98] Maurice, C., Neumann, C., Heen, O., and Francillon, A. "C5: Cross-Cores Cache Covert Channel." In: *Proceedings of the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. 2015.

[99] Maurice, C., Weber, M., Schwarz, M., Giner, L., Gruss, D., Boano, C. A., Mangard, S., and Römer, K. "Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud." In: *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. 2017.

[100] McKeen, F., Alexandrovich, I., Berenzon, A., Rozas, C. V., Shafi, H., Shanbhogue, V., and Savagaonkar, U. R. "Innovative Instructions and Software Model for Isolated Execution." In: *Proceedings of the International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*. 2013.

[101] Microsoft Edge Team. "Mitigating speculative execution side-channel attacks in Microsoft Edge and Internet Explorer." 2018. URL: `https://blogs.windows.com/msedgedev/2018/01/03/speculative-execution-mitigations-microsoft-edge-internet-explorer/` (visited on 2025/09/30).

[102] Miller, M. "Mitigating Speculative Execution Side Channel Hardware Vulnerabilities." Microsoft Security Response Center (MSRC) Blog. Mar. 2018. URL: `https://msrc.microsoft.com/blog/2018/03/mitigating-speculative-execution-side-channel-hardware-vulnerabilities/` (visited on 2025/09/18).

[103] Moghimi, D., Lipp, M., Sunar, B., and Schwarz, M. "Medusa: Microarchitectural Data Leakage via Automated Attack Synthesis." In: *Proceedings of the USENIX Security Symposium*. 2020.

[104] Naseredini, A., Gast, S., Schwarzl, M., Bernardo, P. M. S., Smajic, A., Canella, C., Berger, M., and Gruss, D. "Systematic Analysis of Programming Languages and Their Execution Environments for Spectre Attacks." In: *Proceedings of the International Conference on Information Systems Security and Privacy (ICISSP)*. 2022.

[105] Nemati, H., Buiras, P., Lindner, A., Guanciale, R., and Jacobs, S. "Validation of Abstract Side-Channel Models for Computer Architectures." In: *Proceedings of the International Conference on Computer Aided Verification (CAV)*. 2020.

[106] Nesbit, K. J. and Smith, J. E. "Data Cache Prefetching Using a Global History Buffer." In: *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2004.

[107] Nethercote, N. and Seward, J. "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation." *ACM SIGPLAN Notices* 42, 6 (June 2007).

[108] Neve, M. and Seifert, J.-P. "Advances on Access-Driven Cache Attacks on AES." In: *Proceedings of the International Conference on Selected Areas in Cryptography (SAC)*. 2007.

[109] Nilizadeh, S., Noller, Y., and Pasareanu, C. S. "DifFuzz: Differential Fuzzing for Side-Channel Analysis." In: *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*. 2019.

[110] Oleksenko, O., Fetzer, C., Köpf, B., and Silberstein, M. "Revizor: Testing Black-Box CPUs against Speculation Contracts." In: *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2022.

[111] Oleksenko, O., Guarnieri, M., Köpf, B., and Silberstein, M. "Hide and Seek with Spectres: Efficient Discovery of Speculative Information Leaks with Random Testing." In: *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. 2023.

[112] Oleksenko, O., Trach, B., Silberstein, M., and Fetzer, C. "SpecFuzz: Bringing Spectre-type Vulnerabilities to the Surface." In: *Proceedings of the USENIX Security Symposium*. 2020.

[113] Ormandy, T. "Reptar." 2024. URL: `https://lock.cmpxchg8b.com/reptar.html` (visited on 2025/09/30).

[114] Osvik, D. A., Shamir, A., and Tromer, E. "Cache Attacks and Countermeasures: The Case of AES." In: *Proceedings of the Cryptographers' Track at the RSA Conference*. 2006.

[115] Pardoe, A. "Spectre mitigations in MSVC." 2018. URL: `https://devblogs.microsoft.com/cppblog/spectre-mitigations-in-msvc/` (visited on 2025/09/30).

[116] Payer, M. "HexPADS: A Platform to Detect 'Stealth' Attacks." In: *Proceedings of the International Symposium on Engineering Secure Software and Systems (ESSoS)*. 2016.

[117] Percival, C. "Cache Missing for Fun and Profit." In: *BSDCan*. 2005.

[118] Pereida García, C., Brumley, B. B., and Yarom, Y. "'Make Sure DSA Signing Exponentiations Really Are Constant-Time'." In: *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. 2016.

[119] Pessl, P., Gruss, D., Maurice, C., Schwarz, M., and Mangard, S. "DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks." In: *Proceedings of the USENIX Security Symposium*. 2016.

[120] Pizlo, F. "What Spectre and Meltdown Mean For WebKit." 2018. URL: `https://webkit.org/blog/8048/what-spectre-and-meltdown-mean-for-webkit/` (visited on 2025/09/30).

[121] Qi, Z., Feng, Q., Cheng, Y., Yan, M., Li, P., Yin, H., and Wei, T. "SpecTaint: Speculative Taint Analysis for Discovering Spectre Gadgets." In: *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. 2021.

[122] Ragab, H., Milburn, A., Razavi, K., Bos, H., and Giuffrida, C. "CrossTalk: Speculative Data Leaks Across Cores Are Real." In: *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. 2021.

[123] Rainer, M., Hetterich, L., Thomas, F., Hornetz, T., Trampert, L., Gerlach, L., and Schwarz, M. "Rapid Reversing of Non-Linear CPU Cache Slice Functions: Unlocking Physical Address Leakage." In: *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. 2025.

[124] Rane, A., Lin, C., and Tiwari, M. "Raccoon: Closing Digital Side-Channels through Obfuscated Execution." In: *Proceedings of the USENIX Security Symposium*. 2015.

[125] Raspberry Pi Ltd. "Raspberry Pi Documentation - The Linux Kernel." 2024. URL: `https://www.raspberrypi.com/documentation/computers/linux_kernel.html` (visited on 2025/09/18).

[126] Reis, C. "Mitigating Spectre with Site Isolation in Chrome." 2018. URL: `https://security.googleblog.com/2018/07/mitigating-spectre-with-site-isolation.html` (visited on 2025/09/30).

[127] Rockchip Electronics Co., Ltd. "Rockchip RK3399-T Datasheet." Revision 1.0. Aug. 2021. URL: `https://dl.radxa.com/rockpi4/docs/hw/datasheets/Rockchip%5C%20RK3399-T%5C%20Datasheet%5C%20V1.0-20210818.pdf` (visited on 2025/09/18).

[128] Rohan, A., Panda, B., and Agarwal, P. "Reverse Engineering the Stream Prefetcher for Profit." In: *Proceedings of the IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. 2020.

[129] Rüegge, S., Wikner, J., and Razavi, K. "Branch Privilege Injection: Compromising Spectre v2 Hardware Mitigations by Exploiting Branch Predictor Race Conditions." In: *Proceedings of the USENIX Security Symposium*. 2025.

[130] Ryan, K. "Hardware-Backed Heist: Extracting ECDSA Keys from Qualcomm's TrustZone." In: *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. 2019.

[131] Sanchez Vicarte, J. R., Flanders, M., Paccagnella, R., Garrett-Grossman, G., Morrison, A., Fletcher, C. W., and Kohlbrenner, D. "Augury: Using Data Memory-Dependent Prefetchers to Leak Data at Rest." In: *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. 2022.

[132] Sanchez Vicarte, J. R., Shome, P., Nayak, N., Trippel, C., Morrison, A., Kohlbrenner, D., and Fletcher, C. W. "Opening Pandora's Box: A Systematic Study of New Ways Microarchitecture Can Leak Private Data." In: *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 2021.

[133] Schwarz, M., Lipp, M., Moghimi, D., Van Bulck, J., Stecklina, J., Prescher, T., and Gruss, D. "ZombieLoad: Cross-Privilege-Boundary Data Sampling." In: *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. 2019.

[134] Schwarz, M., Maurice, C., Gruss, D., and Mangard, S. "Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript." In: *Proceedings of the Financial Cryptography and Data Security Conference (FC)*. 2017.

[135] Schwarzl, M., Canella, C., Gruss, D., and Schwarz, M. "Specfuscator: Evaluating Branch Removal as a Spectre Mitigation." In: *Proceedings of the Financial Cryptography and Data Security Conference (FC)*. 2021.

[136] Semal, B., Markantonakis, K., Mayes, K., and Kalbantner, J. "One Covert Channel to Rule Them All: A Practical Approach to Data Exfiltration in the Cloud." In: *Proceedings of the IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. 2020.

[137] Shin, Y., Kim, H. C., Kwon, D., Jeong, J. H., and Hur, J. "Unveiling Hardware-Based Data Prefetcher, a Hidden Source of Information Leakage." In: *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. 2018.

[138] Somogyi, S., Wenisch, T. F., Ailamaki, A., and Falsafi, B. "Spatio-Temporal Memory Streaming." In: *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 2009.

[139] Somogyi, S., Wenisch, T. F., Ailamaki, A., Falsafi, B., and Moshovos, A. "Spatial Memory Streaming." In: *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 2006.

[140] Standard Performance Evaluation Corporation (SPEC). "runcpu - Using CPU 2017." 2019. URL: `https://www.spec.org/cpu2017/Docs/runcpu.html#iter` (visited on 2025/09/18).

[141] Standard Performance Evaluation Corporation (SPEC). "SPEC CPU® 2017." 2022. URL: `https://www.spec.org/cpu2017/` (visited on 2025/09/18).

[142] Tanenbaum, A. S. and Bos, H. *Modern Operating Systems*. 4th. Pearson, Boston, 2015.

[143] Thomas, F., García Arribas, E., Hetterich, L., Weber, D., Gerlach, L., Zhang, R., and Schwarz, M. "RISCover: Automatic Discovery of User-exploitable Architectural Security Vulnerabilities in Closed-Source RISC-V CPUs." In: *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. 2025.

[144] Tromer, E., Osvik, D. A., and Shamir, A. "Efficient Cache Attacks on AES, and Countermeasures." *Journal of Cryptology* 23, 2 (Jan. 2010).

[145] Tsunoo, Y., Saito, T., Suzaki, T., and Shigeri, M. "Cryptanalysis of DES Implemented on Computers with Cache." In: *Proceedings of the Conference on Cryptographic Hardware and Embedded Systems (CHES)*. 2003.

[146] Tullsen, D. M., Eggers, S. J., and Levy, H. M. "Simultaneous Multithreading: Maximizing On-Chip Parallelism." In: *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 1995.

[147] Turner, P. "Retpoline: a software construct for preventing branch-target-injection." 2018. URL: `https://support.google.com/faqs/answer/7625886` (visited on 2025/09/30).

[148] Van Bulck, J., Minkin, M., Weisse, O., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Wenisch, T. F., Yarom, Y., and Strackx, R. "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient out-of-Order Execution." In: *Proceedings of the USENIX Security Symposium*. 2018.

[149] Van Bulck, J., Moghimi, D., Schwarz, M., Lipp, M., Minkin, M., Genkin, D., Yarom, Y., Sunar, B., Gruss, D., and Piessens, F. "LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection." In: *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. 2020.

[150] Van Bulck, J., Piessens, F., and Strackx, R. "SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control." In: *Proceedings of the Workshop on System Software for Trusted Execution (SysTEX)*. 2017.

[151] van Schaik, S., Milburn, A., Österlund, S., Frigo, P., Maisuradze, G., Razavi, K., Bos, H., and Giuffrida, C. "RIDL: Rogue In-Flight Data Load." In: *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. 2019.

[152] Wagner, L. "Mitigations Landing for New Class of Timing Attack." Mozilla Security Blog. Jan. 2018. URL: `https://blog.mozilla.org/security/2018/01/03/mitigations-landing-new-class-timing-attack` (visited on 2025/09/18).

[153] Weber, D., Ibrahim, A., Nemati, H., Schwarz, M., and Rossow, C. "Osiris: Automated Discovery of Microarchitectural Side Channels." In: *Proceedings of the USENIX Security Symposium*. 2021.

[154] Wenisch, T. F., Ferdman, M., Ailamaki, A., Falsafi, B., and Moshovos, A. "Practical Off-Chip Meta-Data for Temporal Memory Streaming." In: *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2009.

[155] Wenisch, T. F., Somogyi, S., Hardavellas, N., Kim, J., Ailamaki, A., and Falsafi, B. "Temporal Streaming of Shared Memory." In: *Proceedings of the International Symposium on Computer Architecture (ISCA)*. 2005.

[156] Wiebing, S., Faveri Tron, A. de, Bos, H., and Giuffrida, C. "InSpectre Gadget: Inspecting the Residual Attack Surface of Cross-privilege Spectre v2." In: *Proceedings of the USENIX Security Symposium*. 2024.

[157] Wikner, J. and Razavi, K. "RETBLEED: Arbitrary Speculative Code Execution with Return Instructions." In: *Proceedings of the USENIX Security Symposium*. 2022.

[158] Wikner, J. and Razavi, K. "Breaking the Barrier: Post-Barrier Spectre Attacks." In: *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*. 2025.

[159] Wu, Z., Xu, Z., and Wang, H. "Whispers in the Hyper-Space: High-Speed Covert Channel Attacks in the Cloud." In: *Proceedings of the USENIX Security Symposium*. 2012.

[160] Xiao, C., Tang, M., and Guilley, S. "Exploiting the Microarchitectural Leakage of Prefetching Activities for Side-Channel Attacks." *Journal of Systems Architecture* 139 (2023), 102877.

[161] Yarom, Y. and Falkner, K. "Flush+Reload: A High Resolution, Low Noise, L3 Cache Side-Channel Attack." In: *Proceedings of the USENIX Security Symposium*. 2014.

[162] Yarom, Y., Genkin, D., and Heninger, N. "CacheBleed: a timing attack on OpenSSL constant-time RSA." In: *Proceedings of the Conference on Cryptographic Hardware and Embedded Systems (CHES)*. 2016.

[163] Yu, X., Hughes, C. J., Satish, N., and Devadas, S. "IMP: Indirect Memory Prefetcher." In: *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2015.

[164] Zhang, T., Zhang, Y., and Lee, R. B. "CloudRadar: A Real-Time Side-Channel Attack Detection System in Clouds." In: *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. 2016.

[165] Zhang, Y. and Reiter, M. K. "Düppel: Retrofitting Commodity Operating Systems to Mitigate Cache Side Channels in the Cloud." In: *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. 2013.

[166] Zhang, Z., Tao, M., O'Connell, S., Chuengsatiansup, C., Genkin, D., and Yarom, Y. "BunnyHop: Exploiting the Instruction Prefetcher." In: *Proceedings of the USENIX Security Symposium*. 2023.

A
Appendix

A.1 Appendix to Chapter 4 (FetchBench)

A.1.1 Processors Under Evaluation

We provide a complete list of the processors under evaluation in Table A.1.

A.1.2 Experiment Success Rates

In this section, we illustrate the success rates of our identification and characterization experiments from Section 4.3. Since most of our CPUs implement a stride prefetcher, we use the results of the stride prefetcher existence test in positive direction as a representative example.

Generally, we repeat each testcase 40,000 times per CPU. Figure A.1 visualizes the cache state of the cache lines in our testing memory region after the experiment. The brighter the color, the more often did we observe the respective cache line in cache after completing the experiment. At the beginning of the memory area, we usually observe a strong signal for the loads that we performed to train and trigger the prefetcher (marked with blue dots, •). For many CPUs, we observed prefetching (red crosses, ×) almost as strong as the architectural loads without any false positives. For the `A55`, we see a long sequence of strong prefetch signals that finally fades out until the signal drops below the noise threshold (magenta crosses, +). For the `A72`, most prefetches produce a strong signal. However, some prefetches have a significantly lower probability to occur. For the `A73`, we observe some very strong prefetches accompanied by some false positives at later cache lines. For the `A76`, we observe an overall lower signal for prefetching. For the Ryzen CPUs (`R5Z`, `R5Z+`, `R7Z3`, `R9Z3+`), we observe prefetching with a lower signal and some noise at later cache lines.

Table A.1: List of hardware platforms under evaluation in Chapter 4, identified prefetchers, and test runtime. When multiple different cores are combined in a single package, we highlight the tested cores in bold type.

ID	System Information					Prefetcher Existence							Test Runtime (min)
	Vendor/Model	OS	Arch.	CPU/SoC	CPU/SoC Release	Stride	SMS	Adj. CL	Stream	R.-U. Replay	Ptr. Array	Ptr. Chase	
A53	Raspberry Pi 3	Raspberry Pi OS 11	ARMv8	Broadcom BCM2837 (Cortex-A53)	2016	●	○	○	○	○	○	○	376.0
A55	HardKernel Odroid C4	Ubuntu 20.04	ARMv8	Amlogic S905X3 (Cortex-A55)	2019	●	○	○	○	○	○	○	54.9
A72	Raspberry Pi 4	Raspberry Pi OS 11	ARMv8	Broadcom BCM2711 (Cortex-A72)	2019	●	●	○	○	○	○	○	78.4
A73	96Boards HiKey 960	Debian 9	ARMv8	HiSilicon Kirin 960 (Cortex-A53, **-A73**)	2016	●	●	○	○	○	○	○	79.7
A76	NanoPi R6S	Ubuntu 22.04	ARMv8	Rockchip RK3588S (Cortex-A55, **-A76**)	2021	●	●	○	○	○	○	○	56.6
M1i	Apple Mac Studio	Asahi Linux	ARMv8	Apple M1 Max *Icestorm core*	2021	●	○	○	○	○	○	○	269.8
M1f	— " —	— " —	— " —	*Firestorm core*	— " —	●	○	○	○	○	●	○	236.5
i7SB	HP EliteBook 2760p	Fedora 37	x86_64	Intel Core i7-2620M (Sandy Bridge)	2011	●	○	●	●	○	○	○	36.2
i5HW	Lenovo ThinkPad T440p	Debian 11	x86_64	Intel Core i5-4300M (Haswell)	2013	●	○	●	●	○	○	○	32.7
XeSL	Mini PC	Ubuntu 20.04	x86_64	Intel Xeon E3-1505Mv5 (Skylake)	2015	●	○	●	●	○	○	○	49.0
XeCL	Custom PC	Ubuntu 20.04	x86_64	Intel Xeon E-2176M (Coffee Lake)	2018	●	○	●	●	○	○	○	217.2
i7CL	Lenovo ThinkPad X1 Carbon Gen 8	Ubuntu 22.04	x86_64	Intel Core i7-10510U (Comet Lake)	2019	●	○	●	●	○	○	○	62.3
i3IL	Mini PC	Ubuntu 22.04	x86_64	Intel Core i3-1005G1 (Ice Lake)	2019	●	○	●	●	○	○	○	115.4
i7TL	Lenovo ThinkPad X1 Carbon Gen 9	Ubuntu 22.04	x86_64	Intel Core i7-1165G7 (Tiger Lake)	2020	●	○	●	●	○	○	○	47.7
XeIL	Custom PC	Ubuntu 22.04	x86_64	Intel Xeon Gold 6346 (Ice Lake)	2021	●	○	●	●	○	○	○	363.0
i9ALp	Custom PC	Ubuntu 22.04	x86_64	Intel Core i9-12900K (Alder Lake) *P-core*	2021	●	○	○	○	○	○	○	63.4
i9ALe	— " —	— " —	— " —	*E-core*	— " —	○	○	●	○	○	○	○	340.1
R5Z	Mini PC	Ubuntu 22.04	x86_64	AMD Ryzen 5 2500U (Zen)	2017	●	○	○	○	○	○	○	59.5
R5Z+	Mini PC	Ubuntu 22.04	x86_64	AMD Ryzen 5 3550H (Zen+)	2019	●	○	○	○	○	○	○	58.4
R7Z3	Mini PC	Ubuntu 22.04	x86_64	AMD Ryzen 7 5700G (Zen 3)	2021	●	○	○	○	○	○	○	40.4
R9Z3+	Mini PC	Ubuntu 22.04	x86_64	AMD Ryzen 9 6900HX (Zen 3+)	2022	●	○	○	○	○	○	○	27.4

● Prefetcher identified ○ Prefetcher not identified

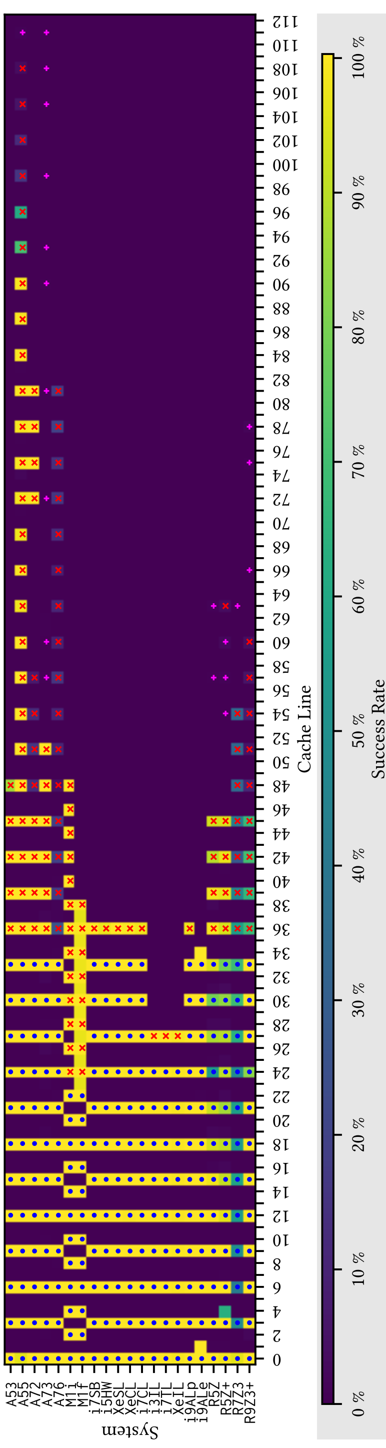

Figure A.1: Visualization of experiment success rates. Blue dots (•) indicate the locations of architectural loads that train and trigger the prefetcher. Red crosses (×) indicate potential prefetch locations that were frequently observed in cache. Magenta crosses (+) indicate potential prefetch locations that were observed only rarely (below the noise threshold).

A.2 Appendix to Chapter 5 (PreFence)

A.2.1 Prefetch Disable Flags on Various Processors

In Tables A.2, A.3, and A.4, we list all MSR flags related to disabling hardware prefetchers that we could find in processor documentation provided by Intel, AMD and ARM, respectively. For Intel, we considered the *Software Developer's Manual* (June 2024) [70] as well as additional documentation [63, 72, 68]. For AMD, we considered all *BIOS and Kernel Developer's Guides* and *Processor Programming References* available from the AMD Documentation Hub[1] (as of July 2024). For ARM, we considered all *Technical Reference Manuals* available from ARM's documentation page[2] starting from the Cortex-A53 (as of July 2024). We note that we only include officially documented disable flags here, as those are the flags that operating system developers could safely rely on; processors may implement additional (undocumented) flags.

We find that the interfaces for controlling hardware prefetchers differ widely between processors, even for models from the same vendor. Thus, abstraction at the operating system kernel level is required to provide a unified interface to user-space processes.

[1] https://www.amd.com/en/search/documentation/hub.html
[2] https://developer.arm.com/documentation/

Table A.2: Prefetch disable flags on Intel processors

Microarchitecture	Model-specific registers, bits
NetBurst	IA32_MISC_ENABLE[9,19]
Core	IA32_MISC_ENABLE[9,19,37,39]
Nehalem	MSR_MISC_FEATURE_CONTROL[0:3]
Sandy Bridge	MSR_MISC_FEATURE_CONTROL[0:3]
Ivy Bridge	MSR_MISC_FEATURE_CONTROL[0:3]
Haswell, Haswell-E	MSR_MISC_FEATURE_CONTROL[0:3]
Broadwell	MSR_MISC_FEATURE_CONTROL[0:3]
Skylake	MSR_MISC_FEATURE_CONTROL[0:3]
Cascade Lake	MSR_MISC_FEATURE_CONTROL[0:3]
Copper Lake	MSR_MISC_FEATURE_CONTROL[0:3]
Caby Lake	MSR_MISC_FEATURE_CONTROL[0:3]
Coffee Lake	MSR_MISC_FEATURE_CONTROL[0:3]
Cannon Lake	MSR_MISC_FEATURE_CONTROL[0:3]
Comet Lake	MSR_MISC_FEATURE_CONTROL[0:3]
Ice Lake	MSR_MISC_FEATURE_CONTROL[0:3]
Tiger Lake	MSR_MISC_FEATURE_CONTROL[0:3]
Alder Lake P-Core	MSR_PREFETCH_CONTROL[0:3,5]
Alder Lake E-Core	MSR_0x1A4[0,2:5], MSR_0x1320[43], MSR_0x1321[43]
Raptor Lake P-Core	MSR_PREFETCH_CONTROL[0:3,5]
Raptor Lake E-Core	MSR_0x1A4[0,2:5], MSR_0x1320[43], MSR_0x1321[43]
Meteor Lake (Core 7 Ultra)	MSR_PREFETCH_CONTROL[0:8]
Xeon Phi Processors 06_57H/06_85H	MSR_PREFETCH_CONTROL[0:1]
Silvermont (Atom)	MSR_MISC_FEATURE_CONTROL[0,2]
Goldmont (Atom)	MSR_MISC_FEATURE_CONTROL[0,2]
Goldmont Plus (Atom)	MSR_MISC_FEATURE_CONTROL[0,2]
Tremont (Atom)	MSR_MISC_FEATURE_CONTROL[0,2]
CPUs with CPUID.(EAX=0x07,ECX=2):EDX[3]=1	IA32_SPEC_CTRL[8] (DDPD_U)
CPUs with CPUID.(EAX=0x07,ECX=0):EDX[29]=1	IA32_UARCH_MISC_CTL[12] (DOITM)

Table A.3: Prefetch disable flags on AMD processors

Processor family	Model-specific registers, bits
Family 10h (K10)	MSRC001_1022[13]
Family 11h (K8/K10)	MSRC001_1022[13]
Family 12h (K10)	MSRC001_1022[13]
Family 14h Models 00h-0Fh (Bobcat)	MSRC001_1022[13]
Family 15h Models 00h-0Fh (Bulldozer/Piledriver)	MSRC001_1022[13]
Family 15h Models 10h-1Fh (Piledriver)	MSRC001_1022[13]
Family 15h Models 30h-3Fh (Steamroller)	MSRC001_1022[13]
Family 15h Models 60h-6Fh (Excavator)	MSRC001_101C[23], MSRC001_102B[18]
Family 15h Models 70h-7Fh (Excavator)	MSRC001_101C[23], MSRC001_102B[18]
Family 16h Models 00h-0Fh (Jaguar)	MSRC001_1022[13], MSRC001_10A0[7]
Family 16h Models 30h-3Fh (Puma)	MSRC001_1022[13], MSRC001_10A0[7]
Family 17h Models 01h, 08h, Revision B2 (Zen/Zen+)	—
Family 17h Model 18h, Revision B1 (Zen+)	—
Family 17h Model 20h, Revision A1 (Zen)	—
Family 17h Model 31h, Revision B0 (Zen 2)	—
Family 17h Model 60h, Revision A1 (Zen 2)	—
Family 17h Model 71h, Revision B0 (Zen 2)	—
Family 17h Model A0h, Revision A0 (Zen 2)	—
Family 19h Model 01h, Revision B1 (Zen 3)	MSRC000_0108[0:3,5]
Family 19h Model 11h, Revision B1 (Zen 4)	MSRC000_0108[0:3,5]
Family 19h Model 21h, Revision B0 (Zen 3)	—
Family 19h Model 51h, Revision A1 (?)	—
Family 19h Model 61h, Revision B1 (Zen 4)	MSRC000_0108[0:3,5]
Family 19h Model 70h, Revision A0 (Zen 4)	MSRC000_0108[0:3,5]
	—: No documented prefetch disable flags.

Table A.4: Prefetch disable flags on ARM processors

Processor	Model-specific registers, bits
Cortex-A53	CPUACTLR_EL1[13:15,22]
Cortex-A57	CPUACTLR_EL1[21,32,56]
Cortex-A72	CPUACTLR_EL1[21,32,42,56]
Cortex-A73	L2CTLR_EL1[21], ECTLR[7,8,10]
Cortex-A75	CPUECTLR[6:10]
Cortex-A76	CPUECTLR_EL1[5,7,8,15,51]
Cortex-A76AE	CPUECTLR_EL1[5,7,8,15,51]
Cortex-A77	CPUECTLR_EL1[5,7,8,15,51]
Cortex-A78	CPUECTLR_EL1[4:5,8,9,15,51,61:63]
Cortex-A78C	CPUECTLR_EL1[4:5,8,9,15,51,61:63]
Cortex-A510	AArch64_imp_cpuectlr_el1[13:14], AArch64_imp_cmpxectlr_el1[26]
Cortex-A520	AArch64_imp_cpuectlr_el1[13:14], AArch64_imp_cmpxectlr_el1[26]
Cortex-A710	CPUECTLR_EL1[4:5,8,9,15,51,61:63]
Cortex-A715	AArch64_imp_cpuectlr_el1[12,41:52], IMP_CPUECTLR2_EL1[8:9]
Cortex-A720	AArch64_imp_cpuectlr_el1[12,41:52,56:57], AArch64_imp_cpuectlr2_el1[8:9,21]
Cortex-X1	CPUECTLR_EL1[4:5,8,9,15,51,61:63]
Cortex-X1C	CPUECTLR_EL1[4:5,8,9,15,51,61:63]
Cortex-X2	IMP_CPUECTLR_EL1[4:5,8,9,15,51,61:63]
Cortex-X3	AArch64_imp_cpuectlr_el1[4:5,8,9,15,51,61:63]
Cortex-X4	AArch64_imp_cpuectlr_el1[4:5,8,9,15,51,61:63]
Cortex-X725	AArch64_imp_cpuectlr_el1[12,41:52,56:57], AArch64_imp_cpuectlr2_el1[8:9,21]
Cortex-X925	AArch64_imp_cpuectlr_el1[4:5,8,9,15,51,61:63]
Neoverse E1	CPUACTLR_EL1[10:15,22]
Neoverse N1	CPUECTLR_EL1[5,8,15,51]
Neoverse N2	AArch64_imp_cpuectlr_el1[4:5,8,9,15,51,61:63]
Neoverse N3	AArch64_imp_cpuectlr_el1[12,41:52,56:57], AArch64_imp_cpuectlr2_el1[8:9,21]
Neoverse V1	CPUECTLR_EL1[4:5,8,9,15,51,61:63]
Neoverse V2	CPUECTLR_EL1[4:5,8,9,15,51,61:63]
Neoverse V3	CPUECTLR_EL1[4:5,8,9,15,51,61:63]

A.3 Appendix to Chapter 6 (Systematization of Knowledge)

A.3.1 Mitigation Gaps: Scope Mapping Analysis Tables

In Section 6.4.2, we map all attacks and defenses in our dataset to the scopes introduced in Section 6.3. Table A.5 shows the result of this mapping for the attacks in our dataset, and Table A.6 shows the mapping result for the defenses.

Table A.5: Attacks and attack scopes

Attack	**Scopes**					**References**
	same-addr.-space	cross-addr.-space	cross-privilege	cross-SMT	cross-core	
Branch Privilege Injection (BPI, Branch Predictor Race Conditions, BPRC)	●	●	●	●	○	[129]
Branch-dependent cache attack	●	●	●	●	●	[124]
DMP (array-of-)pointer prefetcher exploitation	●	●	○	○	○	[131, 28, P3]
Instruction address aliasing causes prefetcher state confusion	●	●	●	○	○	[30, P2, P3]
Leaking single loads through prefetcher state (FetchProbe)	●	●	●	○	○	[54]
Prefetcher attack on shared memory/library	●	●	●	●	●	[137, P3]
Prime+Probe-like prefetcher attack (e.g., PrefetchX-evict)	●	●	●	●	●	[29, P3]
Reptar (Instruction Decoding Bug on Intel CPUs)	●	●	●	●	○	[113]
Retbleed on AMD	●	●	●	●	○	[157]
Retbleed on Intel (Return Stack Buffer Underflow)	●	●	●	○	○	[65, 157, 66]
Spectre-BHB (Branch History Injection (BHI), Intra-Mode Branch Target Injection (Intra-Mode BTI))	●	●	●	●	○	[22, 71, 66, 16, 156]
Spectre-BTB (variant 2, branch target injection, Spectre-BTI)	●	●	●	●	○	[79, 60, 101, 152, 147, 65, 59, 61, 3, 5, 4, 22, 157, 71, 58, 10, 92, 129, 6, 36]
Spectre-PHT (variant 1, input validation bypass)	●	●	●	●	○	[79, 120, 101, 152, 115, 126, 5, 10, 92]
Spectre-PHT variant 1.1 (bounds check bypass on stores, BCBS, speculative buffer overflow)	●	●	●	●	○	[10, 78, 92]
Spectre-RSB	●	●	●	○	○	[10, 81, 93]

(continued on next page)

●: Attack applicable to scope ○: Attack not applicable to scope

Table A.5: Attacks and attack scopes (continued)

Attack	Scopes					References
	same-addr.-space	cross-addr.-space	cross-privilege	cross-SMT	cross-core	
Spectre-STL (variant 4, Spectre-SSB, speculative store bypass)	●	○	○	○	○	[79, 57, 60, 10, 62]
Use prefetcher to trigger attacker-controlled prefetch in victim context (ShadowLoad)	●	●	●	○	○	[54]

●: Attack applicable to scope ○: Attack not applicable to scope

Table A.6: Defenses and defense scopes

Defense	Scopes					References
	same-addr.-space	cross-addr.-space	cross-privilege	cross-SMT	cross-core	
Add data dependency on the outcome of a compare to avoid speculation	●	●	●	●	●	[5]
Arm-recommended compiler defense against Spectre-RSB	●	○	○	○	○	[10]
Avoid indirect jumps (e.g., use JumpSwitches)	●	●	●	●	●	[129, 6]
Blinding	●	●	●	●	●	[28]
Cache/State Partitioning along privilege levels	○	○	●	○	○	[P2]
Cache/State Partitioning along process IDs	○	●	●	○	○	[30, P2, 29, 54, 16, 22]
Cache/State Partitioning along SMT threads	○	○	○	●	○	[29]
Clear BHB on privilege domain switch	○	●	●	○	○	[71, 66, 16, 156]
Clear prefetcher state on context switch	○	●	○	○	○	[30, P2, 54]
Clear registers before return instruction	●	○	○	○	○	[66]
Co-Scheduling	○	●	●	●	○	[28]
Constant-time programming	●	●	●	●	●	[137, 30, P2, 54]
Constrain prefetcher to specific memory regions	●	●	●	●	●	[28]
Disable branch prediction permanently	●	●	●	●	●	[10]
Disable indirect branch prediction in privileged code	○	○	●	○	○	[22]
Disable prefetcher permanently	●	●	●	●	●	[137, 30, P2, 29, 28, 54]

(continued on next page)

●: Defense mitigates attacks in scope ○: Defense does not mitigate attacks in scope

Table A.6: Defenses and defense scopes (continued)

Defense	Scopes					References
	same-addr.-space	cross-addr.-space	cross-privilege	cross-SMT	cross-core	
Disable prefetcher selectively (PreFence)	○	●	○	●	●	[P3, P2, 28]
Disable SMT	○	○	○	●	○	[157, 22]
Disable speculative execution permanently	●	●	●	●	●	[79, 93]
Do not share shared libraries in memory	○	●	●	●	●	[137]
Exploiting prefetcher implementation details as defense	●	●	●	●	●	[131]
FEAT_ECBHB hardware fix	○	○	●	○	○	[16]
FineIBT	●	○	○	○	○	[156, 36]
Flush RSB on context switch	○	●	○	○	○	[93]
Frozen Store-to-Load Forwarding ("Arctic SLoth")	●	●	●	●	○	[78]
Index masking as additional bounds check	●	●	●	●	●	[79, 120, 5, 78]
Indirect Branch Predictor Barrier (IBPB)	●	●	○	○	○	[79, 59, 3, 5, 22, 71, 58, 6]
Indirect Branch Restricted Speculation (IBRS) (incl. eIBRS, AutoIBRS)	○	○	●	○	○	[79, 60, 3, 157, 66, 5, 22, 71, 58, 129, 6]
Insert serializing instruction (such as lfence) to block speculative execution	●	●	●	●	○	[79, 115, 5, 71, 10, 93, 78, 92, 156, 62]
Intel BPI Microcode Update	●	●	●	●	○	[129]
Intel CET-IBT (Control-Flow Enforcement Technology - Indirect Branch Tracking)	●	○	○	○	○	[22, 71, 6, 156]
Intel Indirect Branch Predictor Controls	●	●	●	○	○	[71, 129, 156]
Invalidate branch predictor state on context switch	○	●	○	○	○	[10]
Invalidate branch predictor state on privilege mode switch	○	○	●	○	○	[22, 10]
Isolate secrets into separate address spaces	●	○	○	○	○	[131, 79, 126, 10, 93, 62]
jmp2ret (AMD Retbleed defense proposal)	○	●	●	○	○	[157]
Lazy Store-to-Load Forwarding ("SLoth")	●	●	●	○	○	[78]
Mark specific memory regions non-prefetchable	●	●	●	●	●	[54]
Oblivious Execution	●	●	●	●	●	[124, 30]
Pointer Poisoning	●	●	●	●	●	[79, 120]
Prevent use of speculatively fetched data in subsequent operations	●	●	●	●	●	[79, 93]

(continued on next page)

●: Defense mitigates attacks in scope ○: Defense does not mitigate attacks in scope

Table A.6: Defenses and defense scopes (continued)

Defense	Scopes					References
	same-addr.-space	cross-addr.-space	cross-privilege	cross-SMT	cross-core	
Reduce timer resolution	●	●	●	●	●	[30, 79, 120, 101, 152, 93]
Remove code gadgets that activate a microarchitectural component	●	●	●	●	●	[131]
Reptar Microcode Fix	●	●	●	●	○	[113]
Restrict unprivileged access to interfaces	○	○	●	○	○	[22, 71, 156]
Retpoline ("Generic" variant)	●	●	●	●	○	[79, 147, 65, 5, 157, 22, 71, 58, 92, 129, 6, 156]
Retpoline (AMD's LFENCE/JMP variant)	●	●	●	●	○	[5, 4, 157, 22]
Retpoline variant against Spectre-RSB	●	●	●	●	○	[93]
RSB stuffing on entering privileged mode	○	○	●	○	○	[5, 81]
RSB stuffing when deep call stacks occur	●	●	●	○	○	[157, 66]
Schedule security-critical code on cores where the vulnerable component is not present	●	●	●	●	●	[131, 28]
Single Thread Indirect Branch Predictors (STIBP)	○	○	○	●	○	[79, 61, 3, 5, 22, 58, 6, 157]
Special load instruction without side effects	●	●	●	●	●	[P2]
Speculative Load Hardening (SLH)	●	●	●	●	○	[78, 92]
Speculative Store Bypass Disable (SSBD)	●	○	○	○	○	[62]
Store-to-Load Blocking ("SLoth Bear")	●	●	●	●	○	[78]
Supervisor Mode Access Protection (SMAP)	○	○	●	○	○	[5, 70]
Supervisor Mode Execution Protection (SMEP)	○	○	●	○	○	[5, 71, 81, 70]
Update microcode to add privilege level separation to prefetchers (like IBRS)	○	○	●	○	○	[54]
Use full instead of partial addresses as indices into microarchitectural buffers	○	●	●	●	●	[P2]
Use Speculative Store Bypass Barriers	●	○	○	○	○	[10]

●: Defense mitigates attacks in scope ○: Defense does not mitigate attacks in scope

A.3.2 Implementation Gaps: Implementation Table

In Section 6.4.3, we investigate defense implementations. The table below lists all defense implementations we identified along with with defenses they implement. In addition, we indicate whether the defense is provided as open-source software and the maturity level of the defense.

Table A.7: Defense implementations

Implementation	Defense	Open Source	Maturity
AMD IBPB implementation	Indirect Branch Predictor Barrier (IBPB)	○	🌍
AMD IBRS/AutoIBRS implementation	Indirect Branch Restricted Speculation (IBRS) (incl. eIBRS, AutoIBRS)	○	🌍
AMD SMAP implementation	Supervisor Mode Access Protection (SMAP)	○	🌍
AMD SMEP implementation	Supervisor Mode Execution Protection (SMEP)	○	🌍
AMD STIBP implementation	Single Thread Indirect Branch Predictors (STIBP)	○	🌍
Arm CLRBHB instruction	Clear BHB on privilege domain switch	○	🌍
Arm FEAT_CSV2	Cache/State Partitioning along process IDs	○	🌍
Arm FEAT_ECBHB	FEAT_ECBHB hardware fix	○	🌍
Arm SMCCC_ARCH_WORKAROUND_1	Invalidate branch predictor state on privilege mode switch	●	🌍
Arm SSBB/PSSBB instructions	Use Speculative Store Bypass Barriers	○	🌍
Chromium: Site Isolation	Isolate secrets into separate address spaces	●	🌍
clear-ip-prefetcher instruction (to be called on context switches)	Clear prefetcher state on context switch	○	Private research prototype
Conditional RSB refill linux kernel patch	RSB stuffing when deep call stacks occur	●	🌍
FineIBT Artifact Implementation	FineIBT	●	⚗
FineIBT in Linux Kernel	FineIBT	●	🌍
GCC -fzero-call-used-regs flag	Clear registers before return instruction	●	🌍
GCC retpoline implementation	Retpoline ("Generic" variant)	●	🌍
Intel BPI Microcode Update	Intel BPI Microcode Update	○	🌍
Intel CET-IBT Implementation	Intel CET-IBT (Control-Flow Enforcement Technology - Indirect Branch Tracking)	○	🌍
Intel IBPB implementation	Indirect Branch Predictor Barrier (IBPB)	○	🌍
Intel IBRS/eIBRS implementation	Indirect Branch Restricted Speculation (IBRS) (incl. eIBRS, AutoIBRS)	○	🌍

(continued on next page)

●: Open-source implementation ○: Closed-source implementation
🌍: In production ⚗: Public research prototype Private research prototype (crossed-eye symbol)

Table A.7: Defense implementations (continued)

Implementation	Defense	Open Source	Maturity
Intel Indirect Branch Predictor Controls Implementation	Intel Indirect Branch Predictor Controls	○	🌍
Intel SMAP implementation	Supervisor Mode Access Protection (SMAP)	○	🌍
Intel SMEP implementation	Supervisor Mode Execution Protection (SMEP)	○	🌍
Intel STIBP implementation	Single Thread Indirect Branch Predictors (STIBP)	○	🌍
JumpSwitch Implementation	Avoid indirect jumps (e.g., use JumpSwitches)	○	👁‍🗨
Linux Kernel Retpolines	Retpoline ("Generic" variant)	●	🌍
Linux Kernel SSBD implementation	Speculative Store Bypass Disable (SSBD)	●	🌍
Linux kernel: Disallow unprivileged eBPF by default	Restrict unprivileged access to interfaces	●	🌍
LLVM retpoline implementation	Retpoline ("Generic" variant)	●	🌍
LLVM Speculative Load Hardening Implementation	Speculative Load Hardening (SLH)	●	🌍
Microsoft Edge, Internet Explorer: Reduced timer precision	Reduce timer resolution	○	🌍
Mozilla Firefox: Reduced timer precision	Reduce timer resolution	●	🌍
MSVC: Inserting barriers against Spectre v1 automatically (x86 and ARM)	Insert serializing instruction (such as lfence) to block speculative execution	○	🌍
PreFence prototype	Disable prefetcher selectively (PreFence)	●	⚗
Raccoon prototype	Oblivious Execution	○	👁‍🗨
Reptar Microcode Fix Implementation	Reptar Microcode Fix	○	🌍
WebKit: Index masking	Index masking as additional bounds check	●	🌍
WebKit: Pointer poisoning	Pointer Poisoning	●	🌍
WebKit: Reduced timer precision	Reduce timer resolution	●	🌍

●: Open-source implementation ○: Closed-source implementation
🌍: In production ⚗: Public research prototype 👁‍🗨: Private research prototype