NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

BSc THESIS

Enhanced Techniques for Testing x86 and RISC-V CPUs Against Speculation Contracts

Grigorios E. Moulkiotis

Supervisor: **Karakostas Vasileios,** Assistant Professor

ATHENS

June 2024

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Βελτιωμένες Τεχνικές Δοκιμών x86 και RISC-V Επεξεργαστών Έναντι Συμβολαίων Υποθετικής Εκτέλεσης

Γρηγόριος Ε. Μουλκιώτης

Επιβλέπων: **Καρακώστας Βασίλειος,** Επίκουρος Καθηγητής

ΑΘΗΝΑ

June 2024

BSc THESIS

Enhanced Techniques for Testing x86 and RISC-V CPUs Against Speculation Contracts

Grigorios E. Moulkiotis
S.N.: 1115201900117

SUPERVISOR: **Karakostas Vasileios,** Assistant Professor

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Βελτιωμένες Τεχνικές Δοκιμών x86 και RISC-V Επεξεργαστών Έναντι Συμβολαίων Υποθετικής Εκτέλεσης

Γρηγόριος Ε. Μουλκιώτης
Α.Μ.: 1115201900117

ΕΠΙΒΛΕΠΩΝ: **Καρακώστας Βασίλειος,** Επίκουρος Καθηγητής

ABSTRACT

Transient execution attacks exploit the speculative execution of modern CPUs to leak information regarding the execution of victim programs through the use of microarchitectural side-channels, e.g., caches. Due to the severe security threats that such microarchitectural attacks impose, prior work has focused on developing various testing techniques and frameworks that aim at automatically identifying security vulnerabilities on modern processors. Revizor is a recently proposed model-based relational testing framework that produces random instruction sequences, i.e., test cases, and inputs to detect existing and novel security vulnerabilities leveraging the concept of speculation contracts. In this thesis we enhance Revizor with additional functionality targeting both x86 and RISC-V processors. On the x86 front, we extend Revizor to use timing measurements, instead of hardware performance counters, for collecting hardware traces. This extension allows processor testing in more realistic system setups in which the use of hardware performance counters may be restricted to the attacker. We also enhance Revizor to use the Flush+Flush data cache side-channel attack, which has been shown to be a fast and stealthy attack. Furthermore, we extend Revizor to use the instruction cache as a side-channel for collecting hardware traces by implementing instruction cache attacks. In this way, we increase the testing coverage of the processor's microarchitectural components beyond the data cache. On the RISC-V front, we port the executor component of Revizor to be able to test real implementations of RISC-V processors, as the current version supports only simulation-based testing. Finally, we reuse some of the Revizor's components to automatically identify the CycleDrift RISC-V architectural vulnerability. Our experimental evaluation shows the capabilities and performance of our enhanced testing techniques.

SUBJECT AREA: Computer architecture

KEYWORDS: Hardware security, microarchitectural side-channel attacks, transient execution attacks, cache attacks, fuzzing, x86, RISC-V

ΠΕΡΙΛΗΨΗ

Οι επιθέσεις υποθετικής εκτέλεσης εκμεταλλεύονται την υποθετική εκτέλεση των νεότερων επεξεργαστών για να διαρεύσουν πληροφορίες σχετικά με την εκτέλεση των προγραμμάτων των θυμάτων κάνοντας χρήση μικροαρχιτεκτονικών πλευρικών καναλιών, όπως οι κρυφές μνήμες. Λόγω των σοβαρών απειλών ασφάλειας που επιβάλλουν οι μικροαρχιτεκτονικές επιθέσεις, προηγούμενες εργασίες επικεντρώθηκαν στην ανάπτυξη διάφορων μεθοδολογιών δοκιμών που στοχεύουν στον αυτόματο εντοπισμό τρωτών σημείων ασφάλειας σε σύγχρονους επεξεργαστές. Το Revizor είναι ένα πλαίσιο (framework) δοκιμών σχεσιακού μοντέλου που προτάθηκε πρόσφατα. Παράγει τυχαίες ακολουθίες εντολών και εισόδους για τον εντοπισμό υφιστάμενων και νέων τρωτών σημείων ασφαλείας αξιοποιώντας την έννοια του υποθετικών συμβολαίων. Σε αυτή την πτυχιακή επεκτείνουμε το Revizor με επιπλέον λειτουργικότητα στοχεύοντας επεξεργαστές x86 και RISC-V. Στο κομμάτι του x86, επεκτείνουμε το Revizor ώστε να χρησιμοποιεί χρονικές μετρήσεις για την συλλογή ιχνών υλικού, αντί για μετρητές απόδοσης υλικού. Αυτή η επέκταση επιτρέπει την δοκιμή των επεξεργαστών σε ρεαλιστικές ρυθμίσεις συστήματος στις οποίες η χρήση μετρητών απόδοσης υλικού μπορεί να είναι περιορισμένη στον επιτιθέμενο. Ενισχύουμε επίσης το Revizor ώστε να χρησιμοποιεί την επίθεση πλευρικού καναλιού κρυφής μνήμης Flush+Flush, που έχει προταθεί ως μια γρήγορη και κρυφή επίθεση. Επιπλέον, επεκτείνουμε τον Revizor για να χρησιμοποιεί την κρυφή μνήμη εντολών ως πλευρικό κανάλι για τη συλλογή ιχνών υλικού με την εφαρμογή επιθέσεων κρυφής μνήμης εντολών. Με αυτόν τον τρόπο, αυξάνουμε την κάλυψη δοκιμών των μικροαρχιτεκτονικών στοιχείων των επεξεγαστών πέρα από την κρυφή μνήμη δεδομένων. Στο κομμάτι του RISC-V, επεκτείνουμε το τμήμα εκτέλεσης δοκιμών του Revizor ώστε να υποστηρίζει τη δοκιμή πραγματικών υλοποιήσεων των RISC-V επεξεργαστών, καθώς η τρέχουσα έκδοση υποστηρίζει μόνο δοκιμές που βασίζονται σε προσομοίωση. Τέλος, επαναχρησιμοποιούμε μερικά από τα στοιχεία του Revizor για την αυτόματη αναγνώριση της αρχιτεκτονικής ευπάθειας CycleDrift σε RISC-V επεξεργαστές. Η πειραματική μας αξιολόγηση δείχνει τις δυνατότητες και την απόδοση των βελτιωμένων τεχνικών ελέγχου.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Αρχιτεκτονική υπολογιστών

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: Ασφάλεια υλικού, μικροαρχιτεκτονικές επιθέσεις πλευρικών καναλιών, επιθέσεις υποθετικής εκτέλεσης εντολών, επιθέσεις κρυφών μνημών, fuzzing, x86, RISC-V

Στην οικογένειά μου και τους φίλους μου που ηταν πάντα δίπλα μου.

ACKNOWLEDGEMENTS

I would like to thank my advisor Vasileios Karkostas who introduced me to the world of microarchitectural security and guided me through my thesis. I would also like to thank Asst. Prof. Marco Guarnieri who provided me with access to the RISC-V front-end of Revizor.

CONTENTS

1. INTRODUCTION 15

1.1 Goal & Approach . . . 16

1.2 Thesis Contributions . . . 16

1.3 Organization . . . 17

2. BACKGROUND 18

2.1 Microarchitectural Components and Optimization Techniques . . . 18
2.1.1 Caches . . . 18
2.1.1.1 Cache maintenance instructions . . . 18
2.1.2 Out-of-order execution . . . 19
2.1.3 Speculative execution . . . 19
2.1.3.1 Branch prediction . . . 19

2.2 Microarchitectural side-channel attacks . . . 19
2.2.1 Data cache attacks . . . 20
2.2.1.1 Flush+Reload . . . 20
2.2.1.2 Prime+Probe . . . 20
2.2.1.3 Flush+Flush . . . 20
2.2.1.4 Evict+Reload . . . 21
2.2.2 Instruction cache attacks . . . 21
2.2.3 Transient execution attacks . . . 21
2.2.3.1 Spectre . . . 21
2.2.3.2 Meltdown . . . 22
2.2.3.3 Other attacks . . . 22
2.2.4 Side-channel attacks in RISC-V processors . . . 22
2.2.4.1 CycleDrift . . . 22
2.2.4.2 Cache+Time . . . 23
2.2.4.3 Flush+Fault . . . 23

2.3 Automatic Detection of Speculative Vulnerabilities in Black-Box CPUs . . . 23
2.3.1 Speculation contracts . . . 23
2.3.2 Revizor . . . 24

3. Enhancing Revizor for testing x86 CPUs 26

3.1 Collecting Hardware Traces Based on Timing Measurements . . . 26
3.1.1 Measuring Cache Timing Differences . . . 26
3.1.1.1 Flush+Reload . . . 26
3.1.1.2 Prime+Probe . . . 26
3.1.1.3 Flush+Flush . . . 27
3.1.2 Generating Hardware Traces based on Timing Measurements . . . 27
3.1.2.1 Flush+Reload . . . 28
3.1.2.2 Prime+Probe . . . 29

3.1.2.3 Evict+Reload . . . 30

3.2 Introducing the Flush+Flush data cache attack . . . 30

3.3 Introducing instruction cache attacks . . . 31

4. Enhancing Revizor for testing RISC-V CPUs **34**

4.1 Front-end . . . 34

4.2 Executor . . . 34
4.2.1 Data cache side channel attacks . . . 35
4.2.1.1 Prime+Probe . . . 36
4.2.1.2 Evict+Reload . . . 37
4.2.2 Instruction cache side channel attacks . . . 38

4.3 CycleDrift vulnerability detection . . . 39

4.4 Architectural Fuzzer . . . 40

5. Evaluation **42**

5.1 Methodology . . . 42

5.2 x86 Results . . . 42
5.2.1 Using Timing Measurements for Generating Hardware Traces . . . 42
5.2.1.1 Flush+Reload . . . 43
5.2.1.2 Prime+Probe . . . 43
5.2.1.3 Flush+Flush . . . 43
5.2.2 Using Instruction Cache for Generating Hardware Traces . . . 44

5.3 RISC-V Results . . . 44
5.3.1 CycleDrift . . . 46
5.3.2 Instruction and Data cache attacks . . . 46

6. Related Work **47**

7. Conclusions and future work **48**

ABBREVIATIONS - ACRONYMS **49**

APPENDICES **49**

A. Artifact appendix for timing measurements enhancement **50**

A.1 Calibration . . . 50

A.2 Execution . . . 50

A.3 Analysis of results . . . 50

A.4 Noise reduction . . . 51

B. Artifact appendix for the RISC-V CycleDrift vulnerability **52**

B.1 Installation and configuration 52

B.2 Analyzing the results 52

C. Artifact appendix for instruction cache side channel leakage **55**

REFERENCES **59**

LIST OF FIGURES

2.1 Workflow of Revizor [40]. 24

3.1 Histogram of the latency of memory accesses for the Flush+Reload attack. 27
3.2 Histogram of the latency of memory accesses for the Flush+Flush attack. . 27
3.3 Histogram of the latency of accesses in the instruction cache. 33

4.1 Workflow of the Executor component. 35

5.1 Spectre V1 detection time for (1) Flush+Reload, (2) Flush+Reload with timing differences, (3) Prime and Probe, (4) Flush and Flush, (5) Prime and Probe with timing differences, and (6) Instruction cache side channel attack. 44
5.2 Spectre V1-VAR detection time diagram for (1) Flush+Reload, (2) Flush+Reload with timing differences, (3) Prime and Probe, (4) Flush+Flush, and (5) Instruction cache side channel. 45
5.3 MDS detection time diagram for (1) Flush+Reload, (2) Flush+Reload with timing differences, (3) Prime and Probe, and (4) Flush+Flush. 45
5.4 Spectre V4 detection time diagram for (1) Flush+Reload, (2) Flush+Reload with timing differences, (3) Prime and Probe, and (4) Flush+Flush. 46

LIST OF TABLES

5.1 Ability to detect or not various known vulnerabilities through different side-channel attacks. 42
5.2 Detection times for each vulnerability through side channels. 43

PREFACE

I would like to thank the personnel of Computer Architecture Laboratory, especially Prof. Dimitrios Gizopoulos and Dr. George Papadimitriou for their suggestions during our meetings.

1. INTRODUCTION

To keep up with the performance gains predicted by Moore's Law, high-performance processors use sophisticated optimization techniques at the microarchitecture level. However, these optimization techniques create an important attack surface through microarchitectural side-channels that may jeopardize the security properties of a processor. Given the rising importance of computing in all aspects of everyday life, identifying security vulnerabilities and protecting computing systems from them is of utmost importance, including microarchitectural side-channel attacks.

Microarchitectural side-channel attacks exploit the performance optimizations of processors in order to leak in an indirect way sensitive information about the execution of a victim process. Cache side-channel attacks were among the first microarchitectural side-channel attacks that were introduced. These attacks use either the data cache or the instruction cache to infer which cache lines the victim process touched during its execution. More recently, a new family of microarchitectural attacks was introduced that relies on speculative execution. These attacks are called speculation attacks or transient execution attacks. The famous Spectre [27] and Meltdown [30] attacks were the first ones of this family of attacks, while several other attacks followed next [45, 47, 46]. These attacks aim at exploiting the speculative execution that high-performance processors use, e.g., branch prediction, to leave traces in microarchitectural components, e.g., caches, and extract them using side-channel attacks [54, 42, 21]. In this thesis we focus on transient execution attacks and the automated discovery of speculation vulnerabilities.

Due to the criticality of transient execution attacks, many prior works have focused on methodologies and tools that identify vulnerabilities due to side-channel attacks. These methodologies can be classified into white-box and black-box approaches. White-box approaches [10, 44, 7, 14, 34] target the RTL design of the processor or are based on known microarchitectural details regarding the processor design to discover microarchitectural vulnerabilities. On the other hand, black-box approaches [37, 52, 16, 51, 32, 40] do not leverage any information regarding the processor design and treat the processor as a black box.

Revizor [40] is a recently proposed black-box framework for detecting speculative vulnerabilities that relies on model-based relational testing. The main difference between Revizor and other black-box tools is the use of speculation contracts [22]. Speculation contracts aim at making visible the microarchitectural changes that the ISA abstracts, but the attacker can exploit, through side-channel attacks during transient execution. The processor complies with a speculation contract when the processor exposes as much information as defined by the contract. Revizor generates random instruction sequences, i.e., test cases, and inputs to find counterexamples, i.e., cases in which the processor is not complying with the contract. To identify such counterexamples, Revizor executes the test cases in an emulator and in the real CPU that is under test using different inputs, and collects the contract and the hardware traces, respectively. If the same contract trace but different hardware traces are produced for a given test case with different inputs, Revizor reports the identified speculation vulnerability.

1.1 Goal & Approach

Our goal in this thesis is to understand in depth various microarchitectural side-channel attacks and develop techniques that improve the automated testing of black-box x86 and RISC-V processors against speculative vulnerabilities and detection of flaws related to microarchitectural attacks. To meet that goal, we build on top of the Revizor framework and we enhance it in several aspects.

To improve the testing of x86 processors, we extend Revizor to use timing measurements, instead of hardware performance counters, for collecting hardware traces. This extension allows processor testing in more realistic system setups in which the use of hardware performance counters may be restricted to the attacker. We also enhance Revizor to use the Flush+Flush data cache side-channel attack [20], which has been shown to be a fast and stealthy attack that only relies on timing measurements. Furthermore, we extend Revizor to use the instruction cache as side-channel for collecting hardware traces by implementing instruction cache attacks [5, 6, 13]. In this way, we increase the testing coverage of the processor's microarchitectural components beyond the data cache.

Our results on an Intel Core i7-8750H CPU (Coffee Lake) processor show that the use of timing measurements for collecting hardware traces with the Flush+Reload cache side-channel attack brings similar results compared to using the same attack with hardware performance counters, in terms of vulnerabilities detection (same vulnerabilities are detected) and execution time (in the same order of magnitude). The results are similar for the Flush+Flush attack as well. Regarding the use of instruction cache as a side-channel to collect hardware traces, we observe that this side-channel suffers from much more noise than the data cache, and hence we are able to observe only some speculation vulnerabilities while requiring longer execution time.

To improve the testing of RISC-V processors, we port the executor component, i.e., a Linux kernel module, of Revizor to be able to test real implementations of RISC-V processors, as the current version [12] supports only testing of processor in the gem5 simulator. This porting effort includes, among other parts, the integration of various RISC-V cache side-channel attacks in Revizor. We also reuse some of the Revizor's components to automatically identify the CycleDrift architectural vulnerability [13] that is specific to the RISC-V ISA.

Our results using a RISC-V virtual machine based on QEMU verify that the ported components are able to compile and run. Our experiments also verify the functionality of the our approach to successfully detect the CycleDrift architectural vulnerability.

1.2 Thesis Contributions

In summary, the main contributions of this thesis are:

- We extend Revizor to use timing measurements, instead of hardware performance counters, for collecting hardware traces.
- We introduce the Flush+Flush data cache side-channel attack, which has been shown to be a fast and stealthy attack.
- We introduce an instruction cache attack that allows using the L1 instruction cache as side-channel for collecting hardware traces, instead of using the L1 data cache,

increasing the coverage of side-channels.

- We port the executor component of Revizor to the RISC-V ISA, in order to be able to test real implementations of RISC-V processors.
- We define a methodology that automatically identifies the CycleDrift architectural vulnerability that is specific to the RISC-V ISA.

1.3 Organization

The rest of this document is organized as follows. In Section 2 we provide background information regarding processor optimizations, microarchitectural attacks, and the Revizor framework. We present the extensions that we developed in Revizor for testing x86 and RISC-V processors in Sections 3 and 4, respectively. In Section 5 we evaluate the developed enhancements. We summarize prior works that focused on automatically identifying security vulnerabilities due to microarchitectural attacks in Section 6. Finally, in Section 7 we conclude this work and provide suggestions for future work.

2. BACKGROUND

In this section we provide background information regarding some critical microarchitectural components and optimization techniques that are usually involved in microarchitectural side-channel attacks. We then present various microarchitectural side-channel attacks, i.e., cache attacks and transient execution attacks. Furthermore, we discuss some side-channel attacks that have focused on the RISC-V architecture. Finally, we analyze speculation contracts and Revizor that is an automatic testing framework for identifying existing and novel speculative execution vulnerabilities that we extend in this thesis.

2.1 Microarchitectural Components and Optimization Techniques

In this section we briefly present the architecture of caches in modern systems, and then we describe some optimization techniques of processors which can be used from attackers to leak information through side-channel attacks.

2.1.1 Caches

Caches offer fast access to the stored data, in contrast to the slow access time that the main physical memory (i.e., DRAM) provides, by leveraging the properties of spatial and temporal locality. In modern CPUs, there are separate caches for instruction (i.e., instruction cache) and data accesses (i.e., data cache), as well as unified caches that hold both instructions and data. In addition, these caches are set associative, i.e., the cache is divided into sets and every set includes multiple cache lines depending on the associativity [20]. Furthermore, modern CPUs include a hierarchy of caches. For example, Intel CPUs use three level of caches in which the L1 and L2 caches are private per core, while the L3 cache is shared among the cores. The L3 cache is also inclusive meaning that data from L1 and L2 must also be present in L3. The L3 cache is organized in slices, and the number of slides is typically equal to the number of the cores. The map function that maps physical addresses to slices and sets is undocumented. However, prior works [33, 13] have reverse engineered the map and indexing function of caches, as well as other microarchitectural details.

2.1.1.1 Cache maintenance instructions

ISAs usually provide cache maintenance instructions for managing directly the processor caches. More specifically, the x86 ISA provides: (i) the `clflush` instruction that flushes a specific memory address and is available in user space, and (ii) the `wbinv` instruction which writes back all modified cache lines to main memory and invalidates (flushes) the caches.

The RISC-V ISA does not specify any instruction that flushes the data cache. However, some vendors have implemented data cache flush instructions [13]. On the other hand, the RISC-V ISA includes the `fence.i` instruction that flushes the instruction cache in user space.

2.1.2 Out-of-order execution

Out-of-order execution is an optimization technique that increases the CPU performance by executing instructions based on the availability of input data and execution units, rather than based on the original order in the program. Because data dependent instructions introduce stalls in the processor pipeline, the early execution of non data dependent instructions that appear later in the program's instruction stream may accelerate the execution. Modern CISC (e.g., x86) out-of-order CPUs decode instructions into micro-ops and if all the micro-ops of an instruction are finished, the instruction is retired and the changes are committed in the architectural state; in this way, the instructions retire in order [27]. RISC (e.g., RISC-V) out-of-order CPUs operate in similar fashion directly at the instruction level, without using micro-ops.

2.1.3 Speculative execution

During the program execution, the CPU is not always sure about the execution of the next instruction sequence due to data and control dependencies. Speculative execution allows the processor to avoid stalls due to such dependencies and proceed with executing instruction. During those scenarios the processor predicts and executes the next instruction(s) in speculative mode without affecting the architectural state. If this guess is correct, then the instructions that have been executed in speculation mode, are committed in the architectural state, and the performance is improved by avoiding stalls in the processor pipeline. In contrast, if the prediction is wrong, then the processor stops the speculative execution, flushes the pipeline, falls back into the previously saved architectural state and starts again executing instructions from the correct path now. Examples of speculative execution include: branch prediction, prediction of addresses of memory loads and stores, and prediction of values for executing instructions, among others.

The instructions executed during wrong speculative execution are called transient instructions. Even though the architectural state is not affected after wrong speculative execution, the microarchitectural state, e.g., caches, may change during the execution of such instructions, leaving microarchitectural traces in the processor's components.

2.1.3.1 Branch prediction

Branch prediction is an optimization technique that aims to accelerate the execution time based on speculative execution, by improving the utilization of the processor pipeline. The CPU during branches does not know which path should be followed, so it will attempt to deduce if the branch is taken or not based on recent history. The component that is responsible for predicting branches is called branch predictor or branch prediction unit. If the prediction is correct, the execution will continue. In different case, the processor flushes the pipeline and starts executing instructions from the correct path.

2.2 Microarchitectural side-channel attacks

Given the organization of modern processors from the previous section we examine how the aforementioned optimization mechanisms can be used against information security through cache and transient execution attacks.

2.2.1 Data cache attacks

Caching introduces timing differences between cached data and data that reside only in physical memory. The attacker can use that timing difference or delta in order to implement a side-channel attack against a victim process by inferring which cache lines the victim process touched during its execution. Next we briefly describe the most well-known data cache attacks.

2.2.1.1 Flush+Reload

Flush+Reload [54] is a side-channel attack that utilizes cache maintenance instructions in order to identify victim accesses into the cache lines. More precisely, the attacker first flushes the cache line using cache flush instructions (e.g., the `clflush` in x86). Then the attacker lets the victim process execute. Finally, the attacker reloads the data. Based on the timing difference between cached data, the attacker can infer whether the victim used particular data or not. This attack can be used to compromise cryptographic implementations. In order to initiate the attack, there must be shared memory between the attacker and the victim and the attacker has to be able to execute cache maintenance instructions. The countermeasures of the attack are mainly limiting cache flush instructions to only privileged users and preventing shared memory between processes (as it is typically happening in the cloud between virtual machines).

2.2.1.2 Prime+Probe

Prime+Probe [42] is side-channel cache attack that is used in more restricted environments where memory is not shared and no cache maintenance instructions are available. In this attack, the attacker first probes either the entire cache or specific sets and fills it with its own data. Then the attacker lets the victim process execute. Finally, the attacker probes the cache to determine either which sets were accessed or whether a specific set was accessed by the victim. To successfully mount such an attack, the attacker must use an eviction strategy in order to fill the cache line which requires reverse engineering the indexing and replacement policy of the cache [33]. Prior work [41] has shown that the attack can be used across virtual machines and sandboxed JavaScript environments. Possible countermeasures for mitigating this attack include developing software that is not leaking information in the caches [15] and disallowing the co-execution of VMs on the same processor [26, 29]. Further countermeasures include reducing the accuracy of timing measurements [24, 48], partitioning of caches [1, 49], and randomizing memory and cache mappings [49, 50].

2.2.1.3 Flush+Flush

Flush+Flush [20] is a side-channel attack that utilizes the timing difference of cache flush maintenance instruction. More specifically, the time it takes in order to flush already cached data is slower than flushing cache lines without any data in it. In this attack, the attacker first flushes the cache line and then lets then victim execute. Finally, the attacker flushes the cache again and measures the latency of the flush instruction. The entire attack can be conducted in a loop with flush instructions in it. The attack is faster than the previous ones as it is just flushing the cache lines, and stealthier because it can

not be detected using hardware performance counters as the flush instruction does not perform any memory accesses. The countermeasures for this attack are similar to those used against Flush+Reload. An additional hardware countermeasure for mitigating the Flush+Flush attack is to execute the flush instruction in constant time. The impact of that countermeasure is almost negligible as applications use the flush instruction very rarely.

2.2.1.4 Evict+Reload

Evict+Reload [21] is a hybrid side-channel attack based on the Prime+Probe and Flush+Reload attacks. In more detail, in the first part of the attack the attacker evicts the cache line by performing the prime phase of Prime+Probe. After that, the victim executes. In the final part of the attack, the attacker will reload the data like the second part of the Flush+Reload attack. The Evict+Reload attack does not require the use of cache maintenance instructions, but it requires the use of shared memory between the attacker and the victim.

2.2.2 Instruction cache attacks

Instruction cache attacks [5, 6] are not very famous among the side-channel attacks because the instruction caches are private per core and thus the attacker must be running on the same core as the victim to perform such attacks. The concept of the instruction cache attacks is very close to that of the Prime+Probe data cache attack. The attacker first fills the instruction cache with its own dummy instructions and then lets the victim execute. Then, the attacker tries to reload those instructions by jumping to the corresponding label and measuring the time of each reload. As we mention later, RISC-V processors are additionally vulnerable to the instruction cache side-channel through the Flush+Fault side-channel attack that is close to the Flush+Reload data cache side-channel attack.

2.2.3 Transient execution attacks

Transient execution attacks are based on the microarchitectural changes that occur due to the execution of instructions in the mispredicted path. The Spectre and Meltdown vulnerabilities are representative examples of this kind of attacks.

2.2.3.1 Spectre

Spectre [27] leverages speculative execution and branch prediction in order to trick the victim program leak information through a microarchitectural side-channel that would not have been otherwise available without the speculative execution of instructions. More specifically, the spectre attack consists of three phases. First, in the prepare phase the attacker mistrains the branch prediction unit and prepares the microarchitectural state of the side-channel (e.g., the data cache). Then, the victim program runs leaking information through the microarchitectural side-channel during the speculation phase. Finally, the attacker reads the microarchitectural state using a side-channel attack.The attack can be implemented targeting the branch predictor or the return stack buffer. .The attack can target operating systems and compilers. The attack can be mitigated in numerous ways, such as employing software countermeasures [8] and redesigning hardware [53, 25, 43].

2.2.3.2 Meltdown

The Meltdown [30] vulnerability abuses the out of order optimization with the aim of breaking the isolation of user space and kernel space, and exfiltrate data from the kernel space which are inaccessible to the user space. The mitigation for that security vulnerability is the use of KAISER [18], a software countermeasure that ensures better isolation by making the pages of physical memory space and kernel space invalid in user space.

2.2.3.3 Other attacks

After the initial discovery of the Spectre and Meltdown vulnerabilities, numerous additional transient execution attacks have been published over the years. An interesting transient execution attack is Foreshadow [45] which exploits the speculative execution of processors and leak cryptographic keys from SGX enclaves that are supported in Intel processors. RIDL [47] is a class of speculative execution attacks (also known as Microarchitectural Data Sampling or MDS) that exploits the functionality of CPU internal micro-optimizations, such as the line buffers. Finally, LVI [46] aims to inject speculatively information into victim execution.

2.2.4 Side-channel attacks in RISC-V processors

The topic of security for RISC-V processors has received a lot of attention recently, with researchers investigating both hardware and software attacks and countermeasures. Recent works have demonstrated software attacks [9] and Rowhammer-based attacks [35]. A recent survey [31] provides an overview of the security field for RISC-V processors, suggesting that the RISC-V ISA must standardize cryptographic instructions and that security software that is available in other architectures must be ported to RISC-V as well.

A very recent work [13] shows that RISC-V even though is a newer ISA, it still suffers from cache side-channel attacks similar to the x86 and ARM ISAs. More specifically attacks like Prime+Probe and Evict+Reload, that do not require special cache maintenance instructions such as cache flushing, can be implemented in RISC-V processors. Furthermore, while the RISC-V ISA does not provide a data cache flushing instruction in user space, RISC-V processor vendors tend to add cache maintenance instructions like cache flush which are also available in user space and make it possible for the attacker to initiate attacks such as Flush+Flush and Flush+Reload. In addition, the RISC-V ISA has also the following design flaws from the security perspective: it includes the `rdinstret` and `rdcycle` instructions that return the number of retired instructions and the number of cycles since the boot of the core, respectively, and the `fence.i` instruction that flushes the entire instruction cache. Using those instructions one can perform the following attacks.

2.2.4.1 CycleDrift

This instruction side-channel attack is based on the combined use of the `rdinstret` and `rdcycle` instructions for detecting the multi-cycle instructions executed and determining which instructions may have been executed during the execution of a program. This side-channel attack affects software that requires constant execution time where even if constant time is guaranteed there must also be constant number of executed instructions, i.e.,

two paths in a program must exhibit the same amount of cycles and instructions in order to be secure against this attack. Attacks involving this side-channel can be used to break AES, KASLR, and compiler countermeasures such as Zigzagger.

2.2.4.2 Cache+Time

This type of instruction side-channel does not require shared memory, has cache line granularity, and is based on the `fence.i` instruction. More specifically, the attacker first flushes the entire instruction cache with `fence.i` and mistrains the branch predictor. Then, the attacker lets the victim execute measuring its execution time. If the victim accesses the preloaded branch by the attacker, it will execute faster; otherwise, it will execute slower. The attacker exploits such timing difference to compromise the security of the victim.

2.2.4.3 Flush+Fault

In this instruction cache side-channel attack, the attacker first flushes the instruction cache using the `fence.i` instruction. Afterwards, the attacker reads the cycles timestamp and jumps into the victim's addresses With that jump the attacker will try to cause a fault, e.g., by setting registers to 0 and inside the victim's code exists a jump into register address. After the fault, there exists a fault handler in order to catch the fault and read the cycles timestamp for the second time. By comparing the difference of the timestamps, the attacker can infer whether the victim has used that instruction cache line (i.e., executed the corresponding instructions) by observing fast execution, or not. The attack is similar to the logic of the Flush+Reload attack.

2.3 Automatic Detection of Speculative Vulnerabilities in Black-Box CPUs

2.3.1 Speculation contracts

Speculation contracts have been recently introduced [22] in order to make visible the microarchitectural changes that the ISA abstracts, but the attacker can exploit through side-channel attacks during speculative or transient execution. More specifically, a speculation contract has two parts: the observation clause and the execution clause. The observation clause describes what information is disclosed during the execution of the program. There are several observation clauses: PC describes the changes in the program counter, CT the addresses of loads and stores, ARCH the values of loads and stores. The execution clause describes how the programs are executed (sequentially or speculatively into the wrong predicted path). Summing up the different clauses, the contracts can be ordered based on the security they guarantee. For example, the SPEC-ARCH contract (i.e., the execution is performed speculatively in the wrong path and one can observe the values of loads and stores) is weaker than SEQ-ARCH (i.e., the execution is sequential and we one observe the same values).

Furthermore, hardware traces are defined as the sum of observations made by a side-channel and are used in order to model a side-channel adversary. An example can be the cache set indexes that were used during the execution of the victim process.

More formally, a contract can be written as:

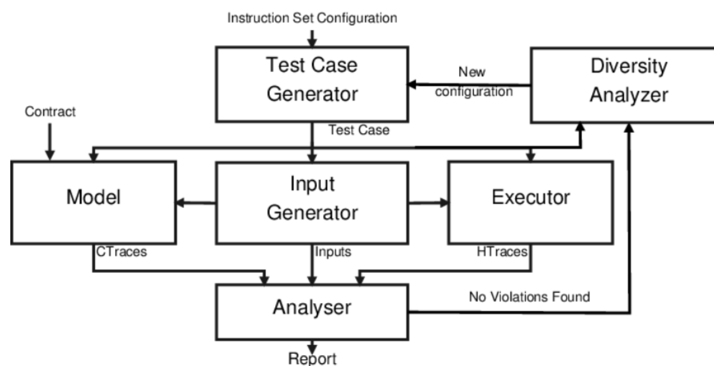

Figure 2.1: Workflow of Revizor [40].

$$[p](\sigma) \Longrightarrow \tau_1\tau_2\tau_3...\tau_n$$

where p express the program that is being executed and σ is the microarchitectural state of the system. States in the assembly code are connected with observations τ that leak security microarchitectural events.

And a hardware trace as:

$$\{p\}(\sigma) \Longrightarrow \mu_1\mu_2\mu_3...\mu_n$$

where p and σ are the same as in the contract traces and $\mu_1, \mu_2, \mu_3, ..., \mu_n$ indicate the hardware traces that the attacker can assume using side-channel attacks.

One can combine the contract traces and hardware traces during the execution of a program in order to find out if the contract complies or violates the security properties of the processor. Next we describe how Revizor uses speculative contracts to detect speculative vulnerabilities.

2.3.2 Revizor

Fuzzing has been originally proposed and extensively used as a software technique [36] in order to find bugs automatically by providing random generated inputs. More recently, fuzzing has been also used for security bugs in hardware. The same approach is taken by Revizor [40], which is a black box framework aiming at finding speculation vulnerabilities based on fuzzing techniques. Revizor uses the speculative contracts making them executable and generates random instruction sequences, i.e., test cases, in assembly language. Then, for each test case Revizor generates various inputs in order to trigger or not speculation. Afterwards, it executes the produced test case in emulation and in real system and compares their results using the formal definition of speculation contracts that was described in the previous section. The real system execution collects the hardware traces using the executor, a kernel module that minimizes the noise from external software and hardware performance counters (i.e., L1D cache misses for D-cache attacks) and also minimizes the noise of counting cycles and using timing differences. If different hardware traces are observed during the execution of the same instruction sequence with different inputs, i.e., different information is leaked, then Revizor reports a violation. The workflow of Revizor is summarized in Figure 2.1 (as published in the original Revizor paper [40]).

Subsequent enhancements to Revizor have incorporated additional optimizations to detect more vulnerabilities and to accelerate performance [39] through faster identification

and detection of vulnerabilities. More specifically, the enhanced version of Revizor is able to detect two new vulnerabilities in x86 processors; one that is based on speculative division and another that is based on speculative string comparison. In addition, the enhanced version of Revizor is faster by introducing the speculative and the observation filters. The speculative filter checks whether the generated test program produces speculation during its execution by using performance counters, filtering out programs with no speculation. The observation filter excludes generated program tests that do not produce observable difference between the serialized execution (that uses `lfence` after every instruction) and the speculative one.

Finally, another recent enhancement [23] to Revizor has been the introduction of fault handling in order to deal with vulnerabilities that rely on exceptions, such as Meltdown, MDS, and Foreshadow. The addition of formal verification contracts to model CPU exceptions aid in observing three new behaviours: the execution of an exception, the transient execution after an exception, and finally the transient execution after exception and the value of the faulting instruction that was determined speculatively.

3. ENHANCING REVIZOR FOR TESTING X86 CPUS

In this thesis we extend the x86 infrastructure of Revizor to use timing measurements instead of hardware performance counters as a hardware trace collection method. We also introduce the Flush+Flush data cache side-channel attack into Revizor. Finally, we extend Revizor to collect hardware traces using the instruction cache as side-channel by implementing instruction cache attacks.

3.1 Collecting Hardware Traces Based on Timing Measurements

Introducing timing measurements in Revizor requires first to identify relevant thresholds for cache hits and misses for each side-channel data cache attack that Revizor supports, and then extend Revizor accordingly to generate hardware traces using the timestamp counter instead of the hardware performance counters for the L1 data cache. Revizor currently supports the Flush+Reload, Prime+Probe, and Evict+Reload attacks. In this thesis we focus on the Flush+Reload and the Prime+Probe attacks, and leave the Evict+Reload attack as future work.

3.1.1 Measuring Cache Timing Differences

First we need to measure and find the timing difference between cache hits and misses for each data cache attack. For this step we use a modified version of the calibration program that is provided in the software artifact of the Flush+Flush attack [2, 20] and that supports various data cache side-channel attacks.

3.1.1.1 Flush+Reload

More precisely, for the Flush+Reload attack, the calibration program allocates an array, and accesses an item of the array for a number of times measuring the reload time, i.e., the latency of cache hits, and when the item address is flushed from the cache measuring the latency of cache misses. We modify the original calibration program to use `lfence` between the timestamp measurements, instead of `mfence`, in order to prevent load re-ordering because the use of `mfence` disrupts the filters of Revizor and therefore eliminates its performance optimizations. Figure 3.1 shows the results of measuring the latency of memory accesses as histogram for the Flush+Reload attack (Section 5.1 provides details regarding the evaluation methodology and the system setup).

3.1.1.2 Prime+Probe

The methodology is more complicated for the Prime+Probe attack. The calibration program of the Flush+Flush software artifact supports previous generations of Intel processors than the one we used. As a result, the cycles of hits and misses are indistinguishable and we can not find the proper threshold using the calibration program.

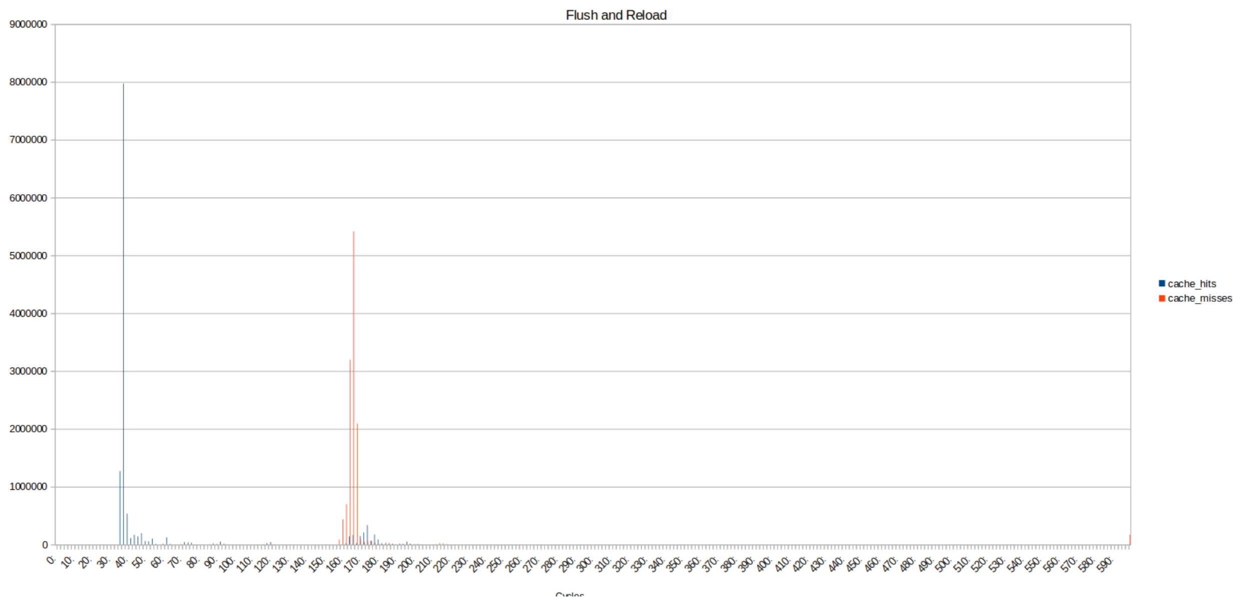

Figure 3.1: Histogram of the latency of memory accesses for the Flush+Reload attack.

Figure 3.2: Histogram of the latency of memory accesses for the Flush+Flush attack.

3.1.1.3 Flush+Flush

Finally, for introducing the Flush+Flush attack we apply the same approach as with the Flush+Reload attack, and use the Flush+Flush calibration program in which every element of the array is flushed after an access and after an access and a flush, measuring the time cache hits and misses, respectively. Figure 3.2 shows the results of measuring the latency of memory accesses as histogram for the Flush+Reload attack.

3.1.2 Generating Hardware Traces based on Timing Measurements

The current version of Revizor collects traces by measuring detailed hardware performance counters for the L1 data cache that is used as side-channel. However, the OS can prohibit the use of those hardware performance counters to user-level programs. This

makes unclear whether speculation randomized attacks that the Revizor generates and reports can be exploited by attackers in real-system setups in which the use of such counters is often restricted. We extend the executor component of Revizor to generate hardware traces using timing measurements, i.e., clock cycles through the available read timestamp counter, instead of data cache misses to allow enhanced testing.

3.1.2.1 Flush+Reload

More specifically, in order to support the Flush+Reload attack in the executor of the Linux kernel, we save the callee saved registers, flush the data cache, read the performance counters (only those used for filtering the test cases for efficiency, i.e., L1 hits, uops issued, uops retirement slots, and misprediction recovery cycles), and we reset the pipeline by adding a number of `lfence` instructions. After that, we execute the test case and then we try to reload the data and measure the latency of the move instruction which we compare with the threshold between the cycles of hits and misses. This threshold is different from system to system and can be found using the calibration method of the previous section. In the case of the Flush+Reload attack, we find the threshold to be 37 cycles in our system (Figure 3.1).

The x86 code from the original Revizor framework that uses hardware performance counters for the reload part of the Flush+Reload attack follows next.

```
xor DEST, DEST
xor OFFSET, OFFSET
1:
xor TMP, TMP
READ_PERFORMANCE_COUNTER
sub TMP, rdx
mov rax, qword ptr BASE + OFFSET
READ_PERFORMANCE_COUNTER
add TMP, rdx
cmp TMP, 0; jne 2f
shl DEST, 1
jmp 3f
2:
shl DEST, 1
or DEST, 1
3:
add OFFSET, 64
cmp OFFSET, 4096; jl 1b
```

The corresponding code that we developed for supporting timing measurements for the reload part of the Flush+Reload attack is the following:

```
xor DEST, DEST
xor OFFSET, OFFSET
1:
xor TMP, TMP
rdtscp
shl rdx, 32; or rdx, rax
sub TMP, rdx
mov rax, qword ptr BASE + OFFSET
```

```
rdtscp
shl rdx, 32; or rdx, rax
add TMP, rdx
cmp rdx, 37; jne 2f
shl DEST, 1
jmp 3f
2:
shl DEST, 1
or DEST, 1
3:
add OFFSET, 64
cmp OFFSET, 4096; jl 1b
```

3.1.2.2 Prime+Probe

Our solution for not knowing the threshold for the Prime+Probe attack in the executor code (due to the difficulty of obtaining meaningful results with the calibration program that was mentioned earlier) is based on the observation that if the victim has accessed the cache line after the prime part, then the probe part will take more time for the attacker. Using this observation, we continuously measure the latency of the probe part and we update dynamically at runtime (while the executor runs) the threshold every time we measure an access time that takes longer than the current threshold. Every measurement that exceeds the current threshold and that will update it, is considered to be a cache access from the victim. The current threshold is reset after each execution of a test case. With this approach, it is possible to stop detecting cache misses after a certain amount of iterations for a specific test case and input set, as the maximum threshold may become too high. Still this approach can detect instances of known speculative vulnerabilities, as shown in Section 5.

The code for our solution in the probe part of the Prime+Probe attack is the following:

```
xor DEST, DEST
xor OFFSET, OFFSET
xor TMP, TMP
xor rdx, rdx
lfence; rdtsc; lfence
shl rdx, 32; or rdx, rax
mov rsi,rdx
PROBE_ONE_SET(BASE, OFFSET)                             \
lfence; rdtsc; lfence
shl rdx, 32; or rdx, rax
sub rdx, rsi
mov TMP,rdx
shl DEST, 1
add OFFSET, 64
1:
lfence
lfence; rdtsc; lfence
shl rdx, 32; or rdx, rax
mov rsi,rdx
```

```
PROBE_ONE_SET(BASE, OFFSET)                    \
lfence; rdtsc; lfence
shl rdx, 32; or rdx, rax
sub rdx, rsi
cmp TMP, rdx
jle 2f
shl DEST, 1
jmp 3f
2:
shl DEST, 1
or DEST, 1
mov TMP, rdx
3:
add OFFSET, 64
cmp OFFSET, 4096; jl 1b
```

3.1.2.3 Evict+Reload

Extending Revizor to use the Evict+Reload attack with timing measurements should be straightforward as the attack consists of steps that we have implemented in the context of the aforementioned cache attacks.

3.2 Introducing the Flush+Flush data cache attack

Revizor currently collects traces based on the Prime+Probe, Flush+Reload, and Evict+Reload attacks, as mentioned earlier. However, the Flush+Flush attack [20] is a data cache attack that has been shown to be fast and stealthy. We extend Revizor with that attack, allowing enhanced testing with stealthier attacks that are prone to noise but avoid causing changes in the hardware performance counters.

For the implementation of the Flush+Flush attack we use exactly the same steps as with the Flush+Reload attack; the only difference is in the reload step in which we flush the cache and we measure the time of the flush instruction. The threshold of this attack is trickier in this case because it changes from time to time and, as it can be seen in Figure 3.2, the latency of hits and misses are very close. In our experimental setup we find and set the threshold to be 118 cycles.

The code of our approach for extending Revizor with the Flush+Flush attack is the following, which is practically inserted in the reload part of Flush+Reload side channel:

```
xor DEST, DEST
xor OFFSET, OFFSET
1:
xor TMP, TMP
mfence
rdtsc
mfence
shl rdx, 32; or rdx, rax
mov TMP, rdx
clflush qword ptr [BASE + OFFSET]
```

```
mfence
rdtsc
mfence
shl rdx, 32; or rdx, rax
sub rdx, TMP
cmp rdx, 116; jg 2f
shl DEST, 1
or DEST, 1
jmp 3f
2:
shl DEST, 1
3:
add OFFSET, 64
cmp OFFSET, 4096; jl 1b
```

3.3 Introducing instruction cache attacks

The current version of Revizor uses only the L1 data cache for collecting traces. While this has been the covert channel for most microarchitectural attacks (and the main focus of corresponding mitigation techniques and mechanisms), instruction cache attacks have been also shown to be effective. To allow for more complete testing of black box CPUs against transient execution attacks, we extend Revizor to also use the L1 instruction cache for collecting hardware traces.

In our implementation we flush the instruction cache with the `wbinv` instruction before executing the test case in order to perform the corresponding flush part of data cache attacks and then we let the victim execute. During the victim execution we measure the instruction cache misses cycles using the ICACHE_16B.IFDATA_STALL event and generate the hardware trace. During the execution of Revizor, if for different inputs different hardware traces are generated, then Revizor reports a speculation violation.

The difference in the implementation of instruction cache attacks and data cache attacks is that we have to jump into the executed code using labels and unconditional jumps in order to determine the execution or not of certain instructions. To enable this functionality, we add a pass in the generator in order to return to trace collection and continue with analyzing the traces. Furthermore, we need to execute different assembly code for the executor than the emulator because the produced assembly code is now collecting hardware traces where the emulator is only supposed to collect contract traces, i.e., the difference between the two assemblies is only in the epilogue of the test case where we collect the hardware trace. This is performed in the code generation in order to make it visible to the compiler where to jump in the newly inserted labels that we use to determine whether the code hase been executed not (otherwise compilation errors would occur in the executor component).

The generator ending part in the original Revizor is the following:

```
.test_case_exit:
MFENCE
```

In the model part, we have to execute the following in the ending part of the test case execution, to avoid collecting hardware traces in the model part:

```
.test_case_exit:
```

```
lfence
mov r8,1
xor r13, r13
mov rcx,0
lfence
shl rdx, 32; or rdx, rax
sub r13, rdx
jmp .bb_main.0
.executor_traces.0:
```

In the executor part we generate the following assembly with hardware traces collection:

```
.test_case_exit:
lfence
mov r8,1
xor r13, r13
xor r11, r11
mov rcx,0
lfence
rdpmc
lfence
shl rdx, 32; or rdx, rax
sub r13, rdx
jmp .bb_main.0
.executor_traces.0:
mov rcx,0
lfence; rdpmc; lfence
shl rdx, 32; or rdx, rax
add r13, rdx
cmp r13, 0; jne _x86_executor_cache_failed0
_x86_executor_cache_success0:
shl r11, 1
or r11, 1
jmp _x86_executor_cache_loop_check0
_x86_executor_cache_failed0:
shl r11, 1
_x86_executor_cache_loop_check0:
```

We also tried to implement an instruction cache attack as shown in [13] in order to test if an attacker with user mode access can perform such a side-channel attack. More specifically, we tried flushing the instruction cache by executing a lot of NOP instructions, avoiding the use of the cache flushing instruction that is provided in kernel mode of x86 or user space mode of RISC-V, and use timing measurements for generating hardware traces. Figure 3.3 depicts the histogram of the latency of accesses in the instruction cache using calibration logic that is similar to that used in [2]). The figure shows the best case histogram that we could produce on a freshly booted machine in which no other programs were running in the background). The results still contain a lot of noise, so we conclude that the use of timing measurements for the instruction cache side-channel attacks make it difficult for the attacker to leak information about the victim execution.

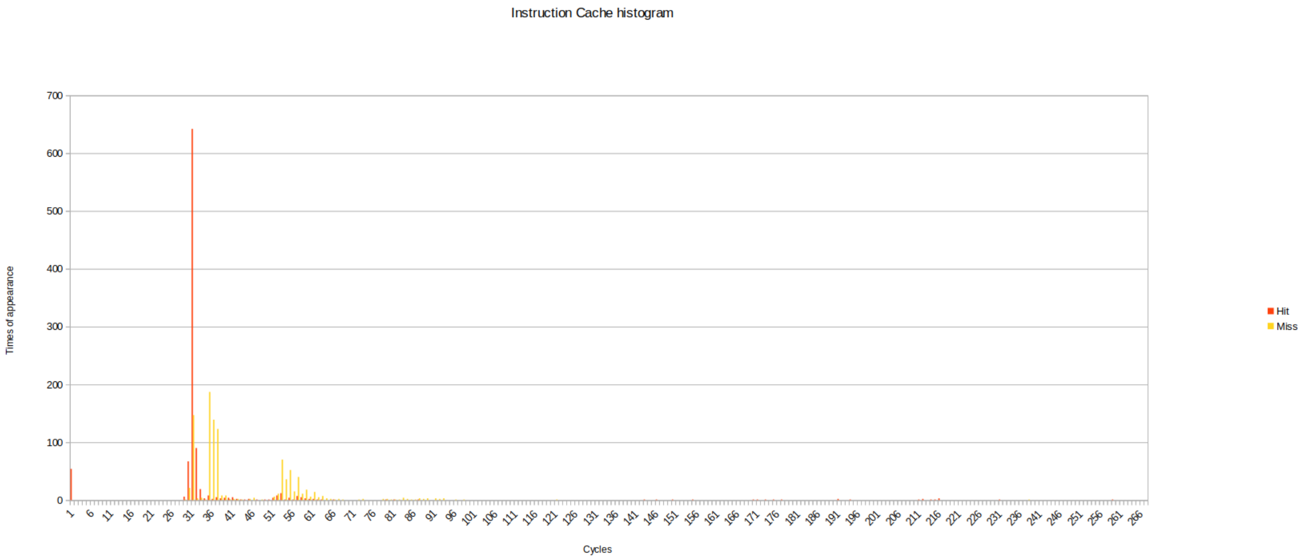

Figure 3.3: Histogram of the latency of accesses in the instruction cache.

4. ENHANCING REVIZOR FOR TESTING RISC-V CPUS

In this part of the thesis, we extend Revizor to be able to test real implementations of RISC-V processors by porting the executor component to RISC-V and implementing data cache side channel attacks and instruction cache side channel attacks. We also implement a detector for the architectural CycleDrift RISC-V vulnerability, and finally we port an architectural fuzzer to RISC-V in order to detect bugs in the implementation of the emulator.

4.1 Front-end

In the current Revizor infrastructure, only the front-end component has been ported to the RISC-V ISA. That component was ported by Eric García Arribas in his thesis [12] and targeted the testing of RISC-V CPUs using the gem5 simulator. In this thesis we reuse that front-end component.

More specifically the front-end is used for generating test cases and inputs for them. The instructions of the test cases are parsed from the ISA specification that is found in the open hardware repository Force-riscv [3]. The repository contains xml files which the parser transforms into json files. These json files are parsed by the test generator in order to create the executable binary and mapping of the relative addresses of the instructions. Furthermore, the test case must comply with additional rules, e.g., it must access memory in a restricted space in order not to cause corruption in memory and some instruction operands must not be zero. Those restrictions are complied with passes that the generator does in order to fix the generated test case. Another part of the front-end is the Model which executes the test cases in the emulator and collects the contract traces. The observation clauses that are currently supported are: MEM (exposure of addresses of memory loads or stores), CT (MEM with additional exposure of the Program Counter), ARCH (CT with additional exposure of loaded values from memory). The execution clauses that are supported by this port are: SEQ (sequential non speculative execution of the test cases), COND (collection of traces is done with conditional jump missprediction from correct and false path of the condition), BPAS (the execution is sequential and the traces are extracted after store bypass), and COND-BPAS (combination of COND and BPAS).

4.2 Executor

The modification we insert is the addition of the executor component, i.e., the back-end component of Revizor that is implemented as a kernel module. The executor aims to execute the test cases and collect the hardware traces that speculation may leave on microarchitectural components, such as the data cache and the instruction cache.

In the implementation of the back-end, we basically follow the same flow of Revizor as done for x86 processors (Figure 4.1). We first read through the compilation of the module the size of L1 Data cache and the ways of associativity of the cache. During the initialization of the module we provide a function that stores the binary of the test case, the inputs, the type of side channel attack that will be performed to generate the hardware traces, and the number of warm-up rounds that will be performed in order to set the microarchtectural state. During the initialization phase, we allocate memory for the test case and set its

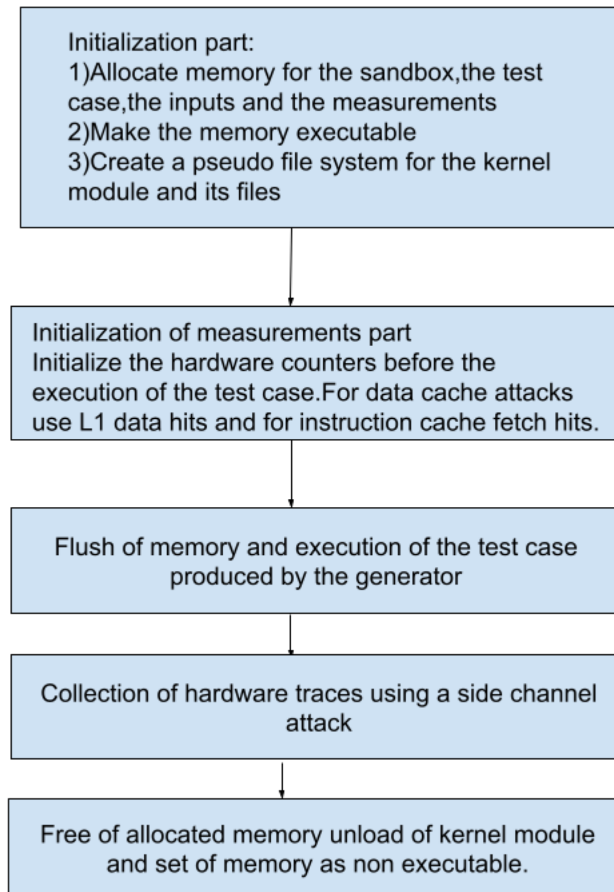

Figure 4.1: Workflow of the Executor component.

memory as executable. We also allocate memory for the inputs and the sandbox, with the sandbox being page aligned. After the execution of the test cases, the memory is deallocated.

Next we describe the implementation of the data and instruction cache side channel attacks in the executor for the RISC-V ISA. Note that the RISC-V ISA does not specify hardware performance counters related to microarchitectural events, e.g., the L1 data and instruction caches, as those are specific to the processor implementation. The ISA specifies the appropriate instruction extension that includes three generic performance counters (related to cycles, instructions, and time) and generic support for accessing any hardware performance counter that is implemented in the processor that is used. Hence, we implement the cache side-channel attacks using timing measurements; modifying the attacks to use specific hardware performance counters should be straightforward.

4.2.1 Data cache side channel attacks

The RISC-V ISA does not contain instructions for flushing the data cache. Therefore, the Flush+Reload and Flush+Flush side channel attacks can not be performed. Our solution to this challenge is using attacks that do not require cache maintenance instructions. This type of side channel attacks are the Prime+Probe and Evict+Reload attacks.

4.2.1.1 Prime+Probe

In our implementation we must first save the callee-saved registers of the previous executing function, because we do not use a compiler to perform clobbering, set the offsets of sandbox, and zero the registers. After that we prime the data cache by filling all the ways of the cache set; in our implementation this is simply performed in a loop where each location inside the eviction region of the sandbox is loaded inside a temporary register for all lines of the cache set. After the prime part, we initialize the registers that are going to be used in the test case and execute the test case. After the execution of the test case, we probe the cache set by refilling all the ways, and checking the hardware performance counters or measuring the timing differences; this step is also implemented in a loop in which every location of the eviction set of every line of the cache sets is loaded into a temporary register where we read the performance counters or the timestamp difference. Collecting the differences essentially gives as the hardware trace which is stored in the results in the epilogue of the side channel attack. Finally, the registers are restored.

The prime part is the following for a 2-way set associative cache:

```
fence rw,rw
add COUNTER ,zero,REPS
_riscv_executor_prime_outer:
add OFFSET,zero, zero
_riscv_executor_prime_inner:
li TMP, xstr(EVICT_REGION_OFFSET)
sub TMP, BASE, TMP
fence rw,rw
add TMP, TMP, OFFSET
ld ACC, 0(TMP)
fence rw,rw
li ACC,xstr(L1D_CONFLICT_DISTANCE)
add TMP,TMP,ACC
ld ACC, 0(TMP)
fence rw,rw
addi OFFSET, OFFSET, 64
li ACC, xstr(L1D_CONFLICT_DISTANCE)
blt OFFSET, ACC,_riscv_executor_prime_inner
li TMP, 1
sub COUNTER , COUNTER ,TMP
bne COUNTER , zero,_riscv_executor_prime_outer
fence rw,rw
```

The probe part follows next, in which the THRESHOLD value can either be based on timing measurements or hardware performance counters:

```
xor DEST, DEST, DEST
xor OFFSET, OFFSET, OFFSET
_riscv_executor_probe_loop:
  fence rw,rw
  xor TMP, TMP, TMP
  rdcycle TMP
  add ACC, zero,TMP
  li s8, xstr(EVICT_REGION_OFFSET)
```

```
sub TMP, BASE, s8
add TMP, TMP, OFFSET
ld TMP2, 0(TMP)
fence rw,rw
li s8, xstr(L1D_CONFLICT_DISTANCE)
add TMP, TMP, s8
ld TMP2, 0(TMP)
fence rw,rw
rdcycle TMP
sub ACC, TMP, ACC
li t0,THRESHOLD
bge ACC, t0,_riscv_executor_probe_failed
_riscv_executor_probe_success:
SLLI s8,DEST,1
add DEST, zero, s8
ori DEST, DEST, 1
  c.j _riscv_executor_probe_loop_check
_riscv_executor_probe_failed:
  SLLI s8,DEST,1
  add DEST, zero, s8
_riscv_executor_probe_loop_check:
addi OFFSET, OFFSET, 64
li s8, xstr(L1D_CONFLICT_DISTANCE)
blt OFFSET,s8,_riscv_executor_probe_loop
```

4.2.1.2 Evict+Reload

The same initial step must be performed to save the registers. After that we must evict the data cache using the same algorithm as in the probe part, and let the test case execute after we set its registers. After the execution, we reload the data. This is implemented with a loop that iterates through the eviction space of our sandbox, loading every memory location inside it and using the differences in cache misses through the hardware performance counters or in timing measurements through the timestamp counter. After that we store the traces and save the registers.

The code for the evict part in the executor follows next:

```
fence rw,rw
add "COUNTER ",zero,"REPS
_riscv_executor_evict_outer:
add "OFFSET",zero, zero
_riscv_executor_evict_inner:
li "TMP", "xstr(EVICT_REGION_OFFSET
sub "TMP", "BASE", "TMP"
fence rw,rw
add "TMP", "TMP", "OFFSET
ld "ACC", 0("TMP")
fence rw,rw
li "ACC","xstr(L1D_CONFLICT_DISTANCE)
add "TMP","TMP","ACC"
```

```
ld "ACC", 0("TMP")
fence rw,rw
addi "OFFSET", "OFFSET", 64
li "ACC", "xstr(L1D_CONFLICT_DISTANCE)
blt "OFFSET", "ACC",_riscv_executor_evict_inner
li "TMP", 1
sub "COUNTER ", "COUNTER ","TMP
bne "COUNTER ", zero,_riscv_executor_evict_outer
fence rw,rw
```

For the reload:

```
xor OFFSET, OFFSET,OFFSET
xor TMP, TMP,TMP
xor TMP2, TMP2,TMP2
xor ACC, ACC,ACC
xor DEST, DEST,DEST
_riscv_executor_reload_loop:
fence rw,rw
xor TMP, TMP,TMP
rdcycle TMP
add ACC, zero,TMP
fence rw,rw
add TMP, BASE, OFFSET
ld TMP2, 0(TMP)
fence rw,rw
rdcycle TMP
sub ACC, TMP, ACC
li TMP2,120
blt ACC,TMP2,_riscv_executor_reload_failed
_riscv_executor_reload_success:
SLLI s8,DEST,1
add DEST, zero, s8
ori DEST, DEST, 1
c.j _riscv_executor_reload_loop_check
_riscv_executor_reload_failed:
SLLI s8,DEST,1
    add DEST, zero, s8
  _riscv_executor_reload_loop_check:
  addi OFFSET, OFFSET, 64
  li TMP, xstr(MAIN_REGION_SIZE)
blt OFFSET,TMP,_riscv_executor_reload_loop
```

4.2.2 Instruction cache side channel attacks

The implementation of the instruction cache attacks follows the same methodology as with the x86 version. The only difference is that we use the `fence.i` instruction, instead of the `wbinv` instruction.

The code that must be produced from the generator in the end is the following, where the threshold can be a timing difference between hits and misses:

```
.test_case_exit:
FENCE rw, rw # instrumentation
li x6,1
rdcycle s4
c.j .bb_main.0
.executor_traces.0:
rdcycle t2
sub s4,t2,s4
li t2,THRESHOLD
blt s4,t2,_riscv_executor_cache_failed0
_riscv_executor_cache_success0:
  SLLI s8,s3,1
  add s3, zero, s8
  ori s3, s3, 1
  c.j _riscv_executor_cache_loop_check0
  _riscv_executor_cache_failed0:
SLLI s8,s3,1
add s3, zero, s8
_riscv_executor_cache_loop_check0:
```

4.3 CycleDrift vulnerability detection

In this part of the thesis we reuse components from Revizor to detect the CycleDrift architectural vulnerability. In this methodology we generate the test case checking the necessary hardware performance counters that the attack relies on and that the RISC-V ISA exposes to the user through the `rdinstret` and `rdcycle` instructions, before and after the execution. Then, we execute it only on the real system. In the final step, we store the difference of those measurements. If for two different inputs we observe different hardware traces, we report the architectural violation.

This methodology can also be used directly on assembly code to test whether it complies with the constant time and constant cycles property through fuzzing, i.e., for random inputs. The methodology can become really helpfully when writing cryptographic applications on RISC-V platform where a fast verification of a few seconds is needed to increase confidence whether the application is complying with the constant time policy. Note that this methodology does not use the emulator, nor relies on the concept of the speculation contracts.

The following code should be produced by executor,noting that the generated tests are placed in the `TEMPLATE_INSERT_TC` the retired instruction and cycles are read in `READ_PFC_START` and `READ_PFC_END`.

```
asm volatile(".word "xstr(TEMPLATE_ENTER));

    prologue();
    //read PFC
    READ_PFC_START();
    // Initialize registers
    SET_REGISTER_FROM_INPUT();
```

```
// Execute the test case
asm("\nfence rw,rw\n"
    ".word "xstr(TEMPLATE_INSERT_TC)" \n"
    "fence rw,rw\n");

// Read pfcs and return hardware trace
READ_PFC_END();

asm volatile("" \
    // s11 <- &latest_measurement
    "li s11, "xstr(MEASUREMENT_OFFSET)"\n"
    "add s11,t5,s11\n"
    "sd s4, 0(s11) \n"
    "sd s5, 8(s11) \n"
    "sd s4, 16(s11) \n"

    // rsp <- stored_rsp
    "li s2, "xstr(RSP_OFFSET)"\n"
    "add s2,t5,s2\n"
    "ld sp, 0(s2)\n"

    // restore registers
    "ld s11, 0(sp) \n"
    "ld s10, 8(sp) \n"
    "ld s9, 16(sp) \n"
    "ld s8, 24(sp) \n"
    "ld s7, 32(sp) \n"
    "ld s6, 40(sp) \n"
    "ld s5, 48(sp) \n"
    "ld s4, 56(sp) \n"
    "ld s3, 64(sp) \n"
    "ld s2, 72(sp) \n"
    "ld s1, 80(sp) \n"
    "ld s0, 88(sp) \n"
    "addi sp,sp,96 \n"
    "ret \n"
);
asm volatile(".word "xstr(TEMPLATE_RETURN));
```

4.4 Architectural Fuzzer

The architectural fuzzer is a mode in the executor component that aims at detecting bugs in the implementation of the emulator, i.e., cases in which the execution in the emulator and the real system do not produce the same results, instead of commencing a side channel attack. The architectural fuzzer is available only in the x86 version. In this thesis we port it to the RISC-V ISA.

The same appraoch as that in the previous section is followed by our architectural fuzzer. Initially, the fuzzer generates and executes the test case in both the emulator and in the

real system. Then, the fuzzer reads the used registers. With that, the fuzzer can verify whether the emulator and the real system agree on the execution of the program or if there exists a bug in the emulator.

The following code describes what is returned from the executor:

```
asm volatile(".word "xstr(TEMPLATE_ENTER));

prologue();
// Initialize registers
SET_REGISTER_FROM_INPUT();

// Execute the test case
asm volatile("\nfence.i\n"
    "xor t1,t1,t1\n"
    ".long "xstr(TEMPLATE_INSERT_TC)" \n");
asm volatile("" \
    // s11 <- &latest_measurement
    "li s11, "xstr(MEASUREMENT_OFFSET)"\n"
    "add s11,t5,s11\n"
    "sd s3, 0(s11) \n"
    "sd s4, 8(s11) \n"
    "sd s5, 16(s11) \n"

    // rsp <- stored_rsp
    "li s2, "xstr(RSP_OFFSET)"\n"
    "add s2,t5,s2\n"
    "ld sp, 0(s2)\n"

    // restore registers
    "ld s11, 0(sp) \n"
    "ld s10, 8(sp) \n"
    "ld s9, 16(sp) \n"
    "ld s8, 24(sp) \n"
    "ld s7, 32(sp) \n"
    "ld s6, 40(sp) \n"
    "ld s5, 48(sp) \n"
    "ld s4, 56(sp) \n"
    "ld s3, 64(sp) \n"
    "ld s2, 72(sp) \n"
    "ld s1, 80(sp) \n"
    "ld s0, 88(sp) \n"
    "addi sp,sp,96 \n"
    "ret \n"
);
asm volatile(".word "xstr(TEMPLATE_RETURN));
```

5. EVALUATION

5.1 Methodology

The experimental platform for the x86 part consists of an Intel Core i7-8750H CPU (Coffee Lake) operating at 2.20GHz. We use Ubuntu 22.04 ,Revizor v1.2.4 and kernel version 6.8.0-40-generic. Furthermore, python is required more specifically version 3.9 and later. Kernel headers are required for the executor kernel module. In our experiments we run every side channel attack in Revizor for detecting every speculative vulnerability 10 times using different seeds (the seeds are the same for all side channels) and report the average time required to detect a violation.

The experimental platform for the RISC-V part consists of an emulated virtual machine based on QEMU running Ubuntu 20.04. Again, we use Revizor v1.2.4. Python and kernel headers requirements must be satisfied as mentioned in the previous paragraph. In our experiments we verified that the RISC-V port of the executor compiles and runs. We also verified that we are able to detect if a program is vulnerable to the CycleDrift attack.

5.2 x86 Results

Table 5.1 summarizes the ability of the various hardware tracing methods to detect or not various speculation attacks, while Table 5.2 summarizes the corresponding execution times. Figures 5.1, 5.2, 5.3, and 5.4 show the execution time results for the Spectre V1, Spectre V1-VAR, MDS, and Spectre V4 vulnerabilities, respectively.

5.2.1 Using Timing Measurements for Generating Hardware Traces

We first examine the impact of using timing measurements and compare the performance of using various attacks.

Variant	Hardware tracing method					
	F+R	P+P	F+R(timing)	F+F	P+P(timing)	I-Cache
V1	Yes	Yes	Yes	Yes	Yes	Yes
V1-Var	Yes	Yes	Yes	Yes	Yes	Yes
V4	Yes	Yes	Yes	Yes	Yes	No
MDS	Yes	Yes	Yes	Yes	Yes	No

Table 5.1: Ability to detect or not various known vulnerabilities through different side-channel attacks.

Variant	Average time per tracing method					
	F+R	P+P	F+R(timing)	F+F	P+P(timing)	I-Cache
V1	28.38	54.08	24.94	84.82	426.22	3941.46
V1-Var	283.325	62.025	201.675	349.325	X	164.05
V4	9927.35	5812.25	6118.2	174.4	X	X
MDS	2019.62	436.4	902.94	1959.95	X	X

Table 5.2: Detection times for each vulnerability through side channels.

5.2.1.1 Flush+Reload

The use of timing measurements in the Flush+Reload attack proved to be almost as effective as the performance counters equivalent. This proves the point that the Flush+Reload with timing differences side channel attack is the most precise of all (as it can even detect vulnerabilities with all filters and disabled flaky violations). As seen on all figures, the use of the Flush+Reload attack with timing measurements is a fast and reliable method for Revizor in order to collect hardware traces and only second in speed after Prime+Probe with hardware traces.

5.2.1.2 Prime+Probe

Prime+Probe relies on the replacement policy of the cache of the processor. Therefore, our generic approach of the Prime+Probe attack with timing differences was slower than Flush+Reload and prone to noise. We did not test for more complex vulnerabilities than Spectre V1 with Prime+Probe because of the longer detection times that would be required. As seen in Figure 5.1, the generic approach is more than five times slower than the other approaches. This shows that the dynamic (and pessimistic) approach of calculating the timing threshold between hits and misses at the runtime (as explained in Section 3.1) can be used only on simple speculative vulnerabilities.

5.2.1.3 Flush+Flush

Due to the fact that we had to disable the observation filter (that executes the program with and without all its instructions surrounded in `lfence`), we encountered longer detection times than for all other side-channel attacks. We also had to enable flaky violations (i.e., a test case shows contract violation but the same hardware trace cannot be reproduced in two consecutive executions of the same test case with the same inputs) because of the noise that disturbs the execution of the test cases and prevents us from reading exactly the same hardware trace for the same execution (i.e., test case and input set). Nevertheless, when comparing all hardware extraction methods, we observe that the average detection times are on the same order of magnitude. This more noisy method of collection can

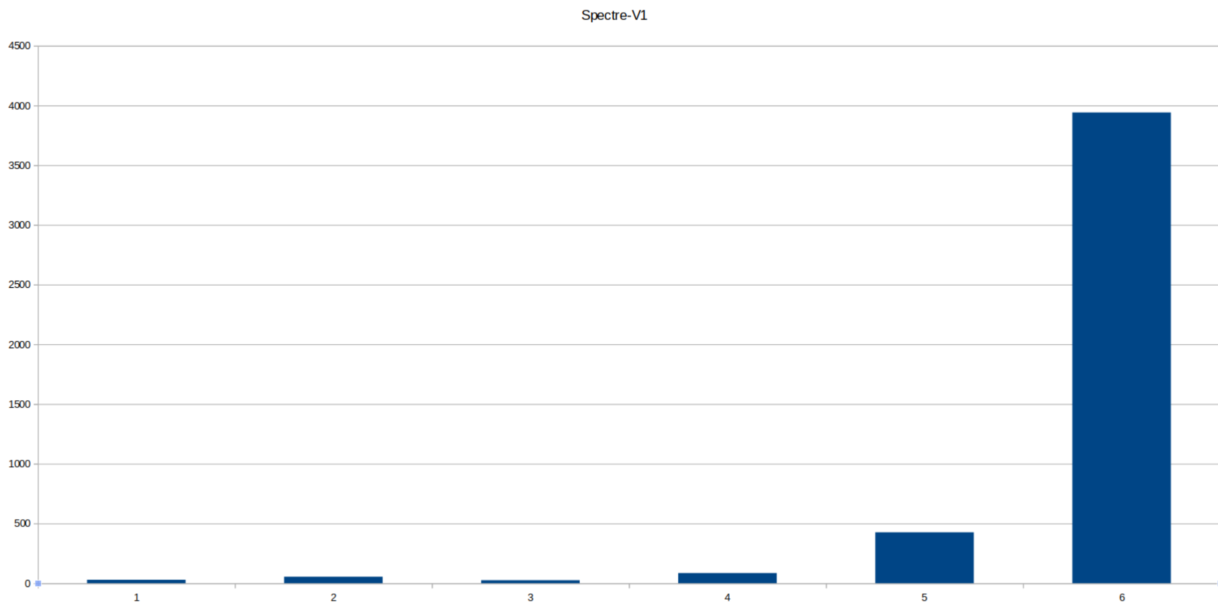

Figure 5.1: Spectre V1 detection time for (1) Flush+Reload, (2) Flush+Reload with timing differences, (3) Prime and Probe, (4) Flush and Flush, (5) Prime and Probe with timing differences, and (6) Instruction cache side channel attack.

be seen to be slower than all the other methods, except for Spectre v4 in which case Flush+Flush is the fastest among all and for MDS in which case Flush+Flush is slightly faster thatn F+R.

5.2.2 Using Instruction Cache for Generating Hardware Traces

Our experiments consisted of testing the processor against the original Spectre vulnerability. We tested with enabled SMT and without it. With SMT enabled we managed to reproduce Spectre using the instruction cache. We also tested with SMT disabled because in some confidential environments it is preferred to disable SMT in order to prevail security. This made clear to us that the extra noise of enable SMT would not disturb our measurements.

The results show that instruction cache can be used as a side channel to leak information from speculative execution, but the instruction cache attacks contain more noise than data cache attacks. As shown in Figure 3.3, the attacker can not reliably distinguish between a fast or slow execution through timing differences (cached or not instruction) and the instruction cache flushing instruction is only available in kernel level and requires root access. Finally, the instruction cache can only be used to detect the original Spectre V1 vulnerability which is patched in all processors at the time of writing. Nevertheless this enhancement can be used for secure processors design and detection of instruction cache leakage.

5.3 RISC-V Results

We test our RISC-V implementation using QEMU which allows us to verify that the code builds and runs but does not simulate any microarchitectural components, such us the processor pipeline and the caches. Therefore the RISC-V port is not tested on real RISC-

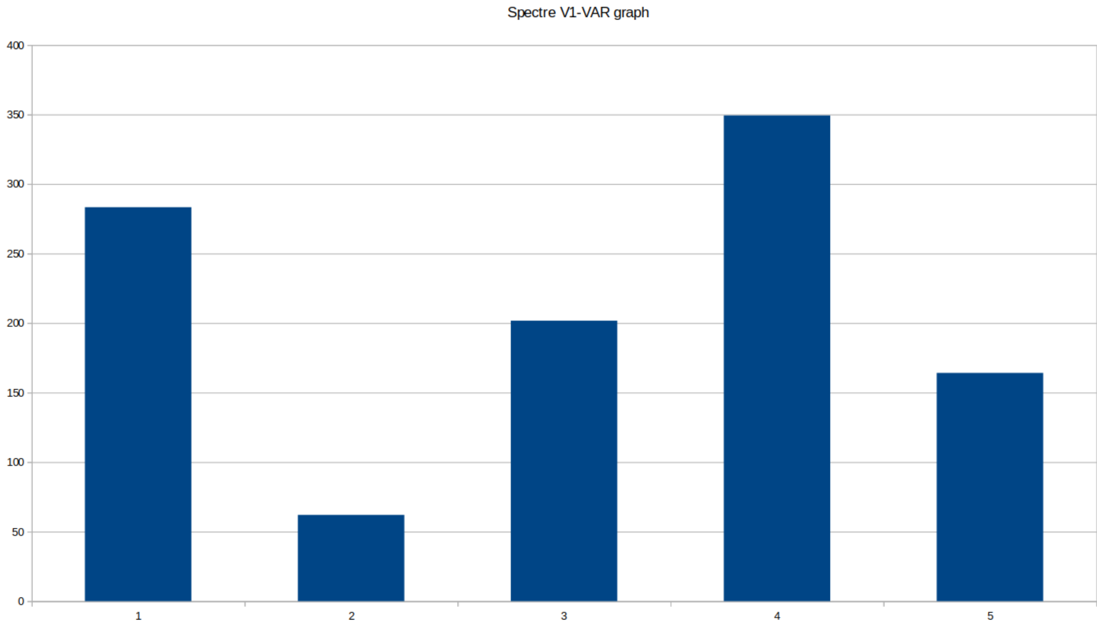

Figure 5.2: Spectre V1-VAR detection time diagram for (1) Flush+Reload, (2) Flush+Reload with timing differences, (3) Prime and Probe, (4) Flush+Flush, and (5) Instruction cache side channel.

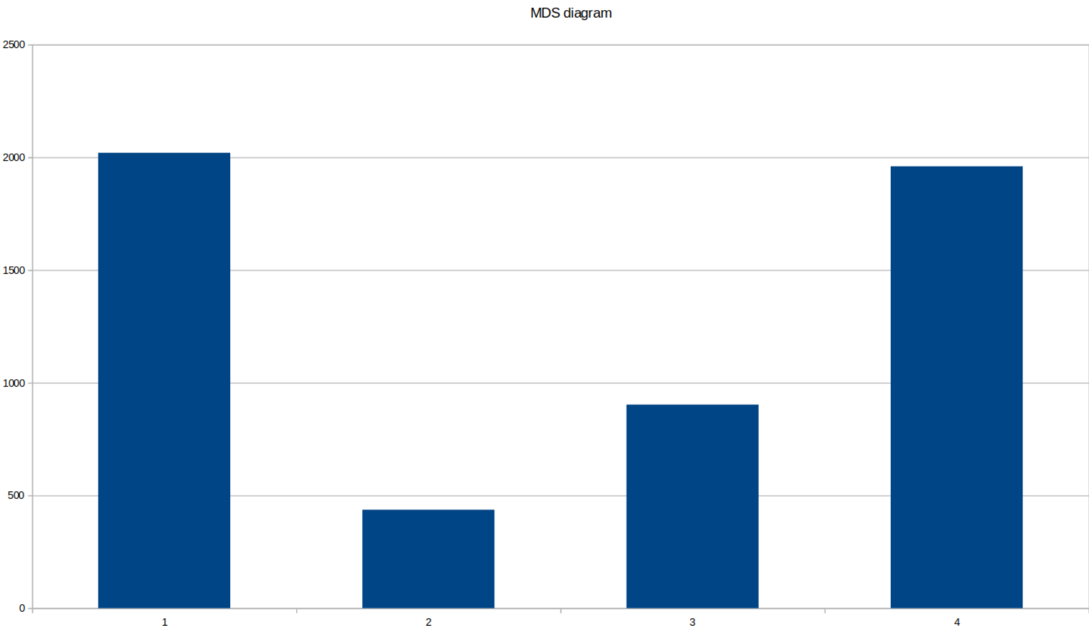

Figure 5.3: MDS detection time diagram for (1) Flush+Reload, (2) Flush+Reload with timing differences, (3) Prime and Probe, and (4) Flush+Flush.

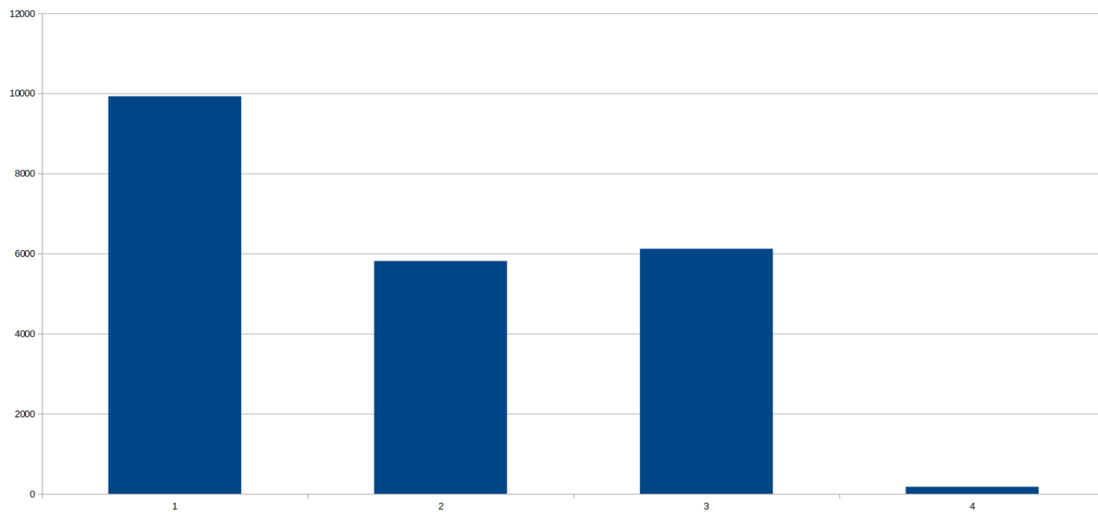

Figure 5.4: Spectre V4 detection time diagram for (1) Flush+Reload, (2) Flush+Reload with timing differences, (3) Prime and Probe, and (4) Flush+Flush.

V systems or simulators; we leave that as future work.

5.3.1 CycleDrift

The first part of our experimental evaluation consists of Revizor producing a test case and checking whether this program is vulnerable to CycleDrift (see Appendix B produced code for proof of concept). After that we create our own test case and test it using Revizor; it successfully managed to find inputs as examples of the CycleDrift vulnerability. Hence, we verify the functionality of the detector for the CycleDrift architectural vulnerability.

5.3.2 Instruction and Data cache attacks

The instruction and data cache attacks performed by Revizor are a port of the x86 version of Revizor and therefore they are expected to detect the same speculative vulnerabilities.

6. RELATED WORK

In this section we review prior works that have focused on detecting automatically (both black-box and white-box approaches) and semi-automatically microarchitectural and architectural leaks due to side-channel attacks.

Black-box approaches for identifying microarchitectural leaks. Black box approaches do not leverage any information regarding the processor design and treat the processor as a black box. Transynther [37] was developed in order to detect and analyze the MDS vulnerability, and Speechminer [52] for studying transient execution attacks. ABSynthe [16] and Osiris [51] were developed towards finding new side channel attacks. Another interesting approach is AutoCAT [32] which uses machine learning in order to detect cache side channels. Finally, Scam-V [38] can be used to detect security violations in in-order ARM processors with model relational testing.

White-box approaches for identifying microarchitectural leaks. White box-approaches are based either on the RTL design, on known microarchitectural details regarding the processor design, or on discovering first such details through reverse-engineering techniques. Fadiheh et al. [10] use RTL analysis of the processor design to identify hardware security vulnerabilities. CheckMate [44] uses formal verification in order to detect known CPU designs that are vulnerable to speculative attacks. WhisperFuzz [7] uses fuzzing to examine the RTL design in order to find microarchitectural side channels using a static analysis methodology. INTROSPECTRE [14] is a pre-silicon framework for early discovery of transient execution vulnerabilities at the RTL level. SpecCheck [34] is a tool for early identification of speculative vulnerabilities in the gem5 simulator.

Approaches for identifying architectural leaks. Several prior works have also used fuzzing to find flaws at the architectural level. This approach is followed in TestRIG [4] for RISC-V processors, RFuzz [28] for RTL designs, and Coppelia [55] for producing software attacks and identifying vulnerable CPUs through symbolic execution. Finally, Cascade [11] uses fuzzing to generate programs in order to find bugs in available RISC-V RTL designs.

Semi-automated approaches. Gruss et al. [21] proposed the concept of cache template attacks to automate the generation of cache side-channel attacks, focusing on cryptographic implementations and tracking events such as keyboard usage. Gerlach et al. [13] followed a similar approach and systematically analyze every side-channel that is available in current RISC-V processors.

7. CONCLUSIONS AND FUTURE WORK

In this thesis we extended the Revizor framework for testing x86 and RISC-V CPUs against speculation contracts. On the x86 front, we enhanced Revizor by: (i) using timing measurements instead of performance counters to collect hardware traces, (ii) implementing the Flush and Flush data cache attack, and (iii) introducing instruction cache attacks. Furthermore, on the RISC-V front, we (iv) ported the front-end component of Revizor to detect speculative vulnerabilities in RISC-V CPU, and (iv) implemented the CycleDrift architectural vulnerability.

Our evaluation results show that the use of timing measurements can introduce noise into the experimental campaign but it can also reveal how easily an attacker can exploit the leakage of the speculative execution in a more realistic execution setup. The use of the Flush+Flush data cache attack proved to be noisy, because of the changes in cache hit/misses threshold, but with the activation of flaky violations we managed to detect speculative vulnerabilities. In addition, our results show that an attacker can use the instruction cache as a side-channel for implementing transient execution attacks as proposed in the Spectre paper [27] and CVE-2017-5753 and CVE-2017-5715, despite the fact that the instruction cache is more sensitive to noise than the data cache. On the RISC-V part, we verified that the ported Revizor front-end can be built and run in RISC-V systems using QEMU. Finally, we tested the ported Revizor against the architectural CycleDrift vulnerability and identified vulnerable code patterns.

Future work can focus on extending Revizor with using more side-channels of the memory system through fuzzing, such as prefetch attacks on KASLR [19] or on cryptographic implementations, and TLB attacks [17]. Another interesting direction is to enhance Revizor to support SIMD execution, such as the AVX extension of the x86 ISA and the vector extensions of the RISC-V ISA. Finally, more research is needed in order to quickly and automatically identify security vulnerabilities in RISC-V processors, as new boards from various vendors are expected to become widely available soon that will include more advanced, high-performance, out-of-order cores with performance optimizations that may leak information about the execution through their microarchitectural design. Automated tools like Osiris [51] can help on this attempt lifting the burden of long manual experiments in order to find new side channels.

ABBREVIATIONS - ACRONYMS

F+R	Flush and Reload
F+F	Flush and Flush
P+P	Prime and Probe
SMT	Simultaneous multithreading
AVX	Advanced Vector Extensions
RISC	Reduced Instruction Set Computer

APPENDIX A. ARTIFACT APPENDIX FOR TIMING MEASUREMENTS ENHANCEMENT

A.1 Calibration

First of all, clone the repository of the Flush+Flush artifact found here. After that navigate in folders histogram and in the folders ff (flush and flush), pp (prime and probe), fr(flush and reload) compile them with `make all` and execute them. The produced histograms should look like figures in Section 3. By using these diagrams, you can define the threshold or use the max of cycles in prime and probe. The threshold is defined in the templates c file in the src/x86/executor folder.

A.2 Execution

After the calibration it is necessary to compile and install the Revizor linux kernel module. We achieve this with the execution of the following commands inside the src/x86/executor: `make unistall` , `make clean`, `make`, and finally `make install`. It is also required to change the configuration for each fuzzing execution. For each vulnerability that we want to detect, we choose the corresponding config file in the /demo folder and also add a line containing the hardware trace collection method. This line starts with `executor_mode :` and continues with the following arguments for each hardware tracing method.

F+R	Flush and Reload
F+T	Flush and Reload with timing differences
P+P	Prime and Probe
P+T	Prime and Probe with timing differences
F+F	Flush and Flush

After choosing the hardware trace method we can execute Revizor with the configuration file and the target ISA. It must also be noted that we must execute the python file of Revizor installed from sources and not the pip module.

A.3 Analysis of results

After the execution of Revizor we get the statistics of the execution in the terminal including the number of test cases executed, the inputs per test case, the number of flaky violations detected (i.e., violations in which the hardware trace could not be reproduced more than one time), the effectiveness, the number of test cases that passed the speculation and observation filters, and finally the duration of execution. There is also available the test case that caused the violation in assembly language in the generated.asm file.

A.4 Noise reduction

We noticed that the Flush+Flush side channel attack is prone to noise so it is optimal to execute Revizor in a freshly booted system, that does not execute any other applications and with SMT disabled.

APPENDIX B. ARTIFACT APPENDIX FOR THE RISC-V CYCLEDRIFT VULNERABILITY

B.1 Installation and configuration

Similarly to the x86 version of Revizor, it is required to install the Revizor linux kernel module. This is installed with `make unistall` , `make clean`, `make`, and `make install` inside the src/RISCV/executor folder. After the installation of the linux kernel module, we need to choose the configuration file that will be used for the execution of the testing campaign. For the cycle drift vulnerability we need to choose the conf-cycle.yaml in the /src/demo folder. Finally, we can test our CPU against cycle drift vulnerability.

B.2 Analyzing the results

After the execution, we get which inputs caused the violation and the number of retired instructions and cycles for each execution. Furthermore, we can see the code that is vulnerable to the CycleDrift architectural vulnerability. An example can be seen below:

```
FENCE rw, rw # instrumentation
.test_case_enter:
.function_main:
.bb_main.entry:
C.J .bb_main.0
.bb_main.0:
LUI x31, 2 # instrumentation
ADDI x31, x31, -1 # instrumentation
SLLI x10, x10, 11 # instrumentation
AND x10, x10, x31 # instrumentation
ADD x10, x10, x30 # instrumentation
LUI x31, 2 # instrumentation
ADDI x31, x31, -1 # instrumentation
LUI x31, 2 # instrumentation
ADDI x31, x31, -1 # instrumentation
LB x11, 1869(x10)
C.NOP
SLLI x11, x11, 2 # instrumentation
AND x11, x11, x31 # instrumentation
ADD x11, x11, x30 # instrumentation
AMOMINU.W x12, x14, (x11)
AND x9, x0, x0 # instrumentation
ADDI x9, x9, 1 # instrumentation
C.LI x9, 12
SLLI x13, x13, 2 # instrumentation
AND x13, x13, x31 # instrumentation
ADD x13, x13, x30 # instrumentation
AMOMIN.W x9, x14, (x13)
SLLI x13, x13, 2 # instrumentation
AND x13, x13, x31 # instrumentation
```

```
ADD x13, x13, x30 # instrumentation
AMOMINU.W x9, x12, (x13)
C.BEQZ x9, .bb_main.1
C.J .bb_main.exit
.bb_main.1:
SLLI x10, x10, 2 # instrumentation
AND x10, x10, x31 # instrumentation
ADD x10, x10, x30 # instrumentation
AMOOR.W x13, x13, (x10)
C.ANDI x9, 15
AND x14, x0, x0 # instrumentation
ADDI x14, x14, 1 # instrumentation
AND x12, x0, x0 # instrumentation
ADDI x12, x12, 1 # instrumentation
C.ADD x14, x12
C.ANDI x9, 5
SLLI x11, x11, 2 # instrumentation
AND x11, x11, x31 # instrumentation
ADD x11, x11, x30 # instrumentation
AMOOR.W x13, x10, (x11)
C.SUB x12, x9
OR x10, x13, x14
SLLI x10, x10, 11 # instrumentation
AND x10, x10, x31 # instrumentation
ADD x10, x10, x30 # instrumentation
FLD f15, 1951(x10)
SLLI x10, x10, 11 # instrumentation
AND x10, x10, x31 # instrumentation
ADD x10, x10, x30 # instrumentation
FLW f8, 1868(x10)
C.NOP
SLLI x11, x11, 2 # instrumentation
AND x11, x11, x31 # instrumentation
ADD x11, x11, x30 # instrumentation
LR.W x9, (x11)
C.SRAI x10, 28
REM x11, x13, x11
SLLI x11, x11, 10 # instrumentation
AND x11, x11, x31 # instrumentation
ADD x11, x11, x30 # instrumentation
LBU x10, 928(x11)
.bb_main.exit:
.test_case_exit:
FENCE rw, rw # instrumentation
```

We can even use existing code to test against cycle drift vulnerability.An example can be seen here:

```
FENCE rw, rw # instrumentation
.test_case_enter:
.function_main:
```

```
.bb_main.entry:
C.J .bb_main.0
.bb_main.0:
LUI x31, 2 # instrumentation
ADDI x31, x31, −1 # instrumentation
SLLI x10, x10, 11 # instrumentation
AND x10, x10, x31 # instrumentation
ADD x10, x10, x30 # instrumentation
LUI x31, 2 # instrumentation
ADDI x31, x31, −1 # instrumentation
LUI x31, 2 # instrumentation
ADDI x31, x31, −1 # instrumentation
LB x11, 1869(x10)
C.NOP
SLLI x11, x11, 2 # instrumentation
AND x11, x11, x31 # instrumentation
ADD x11, x11, x30 # instrumentation
AMOMINU.W x12, x14, (x11)
AND x9, x0, x0 # instrumentation
ADDI x9, x9, 1 # instrumentation
C.LI x9, 12
SLLI x13, x13, 2 # instrumentation
AND x13, x13, x31 # instrumentation
ADD x13, x13, x30 # instrumentation
AMOMIN.W x9, x14, (x13)
SLLI x13, x13, 2 # instrumentation
AND x13, x13, x31 # instrumentation
ADD x13, x13, x30 # instrumentation
AMOMINU.W x9, x12, (x13)
C.BEQZ x9, .bb_main.1
C.J .bb_main.exit
.bb_main.1:
SLLI x11, x11, 10 # instrumentation
AND x11, x11, x31 # instrumentation
ADD x11, x11, x30 # instrumentation
LBU x10, 928(x11)
.bb_main.exit:
.test_case_exit:
FENCE rw, rw # instrumentation
```

APPENDIX C. ARTIFACT APPENDIX FOR INSTRUCTION CACHE SIDE CHANNEL LEAKAGE

For the instruction cache side-channel leakage through speculation vulnerabilities, it is required to use the branch of each repository because of the different way the test cases are produced. The installation is similar to Appendix A and Appendix B for each version of Revizor. Regarding the configuration, the executor mode argument in the configuration file should be set to I for each vulnerability needed to be tested. Finally, after execution we will see the same statistics as for the data cache side-channels with the main difference being the time taken because the instruction cache is nosier channel than the data cache.

BIBLIOGRAPHY

[1] Intel® 64 and ia-32 architectures software developer manuals. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[2] flush_flush. https://github.com/IAIK/flush_flush, 2016.

[3] force-riscv. https://github.com/openhwgroup/force-riscv, 2020.

[4] testrig. https://github.com/CTSRD-CHERI/TestRIG, 2021.

[5] Onur Aciiçmez. Yet another microarchitectural attack: exploiting i-cache. In *Proceedings of the 2007 ACM Workshop on Computer Security Architecture*, CSAW '07, page 11–18, New York, NY, USA, 2007. Association for Computing Machinery.

[6] Onur Aciiçmez, Billy Bob Brumley, and Philipp Grabher. New results on instruction cache attacks. In *Proceedings of the 12th International Conference on Cryptographic Hardware and Embedded Systems*, CHES'10, page 110–124, Berlin, Heidelberg, 2010. Springer-Verlag.

[7] Pallavi Borkar, Chen Chen, Mohamadreza Rostami, Nikhilesh Singh, Rahul Kande, Ahmad-Reza Sadeghi, Chester Rebeiro, and Jeyavijayan Rajendran. Whisperfuzz: White-box fuzzing for detecting and locating timing vulnerabilities in processors, 2024.

[8] Sunjay Cauligi, Craig Disselkoen, Daniel Moghimi, Gilles Barthe, and Deian Stefan. Sok: Practical foundations for spectre defenses. *CoRR*, abs/2105.05801, 2021.

[9] Tobias Cloosters, David Paaßen, Jianqiang Wang, Oussama Draissi, Patrick Jauernig, Emmanuel Stapf, Lucas Davi, and Ahmad-Reza Sadeghi. Riscyrop: Automated return-oriented programming attacks on risc-v and arm64. In *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses*, RAID '22, page 30–42, New York, NY, USA, 2022. Association for Computing Machinery.

[10] Mohammad Rahmani Fadiheh, Dominik Stoffel, Clark Barrett, Subhasish Mitra, and Wolfgang Kunz. Processor hardware security vulnerabilities and their detection by unique program execution checking. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 994–999, 2019.

[11] Kaveh Razavi Flavien Solt, Katharina Ceesay-Seitz. Cascade: Cpu fuzzing via intricate program generation. https://comsec.ethz.ch/research/hardware-design-security/cascade-cpu-fuzzing-via-intricate-program-generation/, 2023.

[12] Eric García Arribas. Fuzzing risc-v processors for speculative leaks. Unpublished, June 2023.

[13] Lukas Gerlach, Daniel Weber, Ruiyi Zhang, and Michael Schwarz. A security risc: Microarchitectural attacks on hardware risc-v cpus. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2321–2338, 2023.

[14] Moein Ghaniyoun, Kristin Barber, Yinqian Zhang, and Radu Teodorescu. Introspectre: A pre-silicon framework for discovery and analysis of transient execution vulnerabilities. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 874–887, 2021.

[15] Xiaofei Guo Gorka Irazoqui. Cache side channel attack: Exploitability and countermeasures. https://www.blackhat.com/asia-17/briefings/schedule/#cache-side-channel-attack-exploitability-and-countermeasures-5373, 2017.

[16] Ben Gras, Cristiano Giuffrida, Michael Kurth, Herbert Bos, and Kaveh Razavi. ABSynthe: Automatic Blackbox Side-channel Synthesis on Commodity Microarchitectures. In *NDSS*, February 2020.

[17] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *USENIX Security*, August 2018. Pwnie Award Nomination for Most Innovative Research.

[18] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. Kaslr is dead: Long live kaslr. In *ESSoS 2017*, 2017.

[19] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch side-channel attacks: Bypassing smap and kernel aslr. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 368–379, New York, NY, USA, 2016. Association for Computing Machinery.

[20] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+flush: A fast and stealthy cache attack. In *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721*, DIMVA 2016, page 279–299, Berlin, Heidelberg, 2016. Springer-Verlag.

[21] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive Last-Level caches. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 897–912, Washington, D.C., August 2015. USENIX Association.

[22] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. Hardware-software contracts for secure speculation. In *IEEE Symposium on Security and Privacy*. IEEE, May 2021.

[23] Jana Hofmann, Emanuele Vannacci, Cedric Fournet, Boris Kopf, and Oleksii Oleksenko. Speculation at fault: Modeling and testing microarchitectural leakage of CPU exceptions. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 7143–7160, Anaheim, CA, August 2023. USENIX Association.

[24] W.-M. Hu. Reducing timing channels with fuzzy time. In *Proceedings. 1991 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 8–20, 1991.

[25] Khaled N. Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Safespec: Banishing the spectre of a meltdown with leakage-free speculation. In *Proceedings of the 56th Annual Design Automation Conference 2019*, DAC ’19, New York, NY, USA, 2019. Association for Computing Machinery.

[26] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. STEALTHMEM: System-Level protection against Cache-Based side channel attacks in the cloud. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 189–204, Bellevue, WA, August 2012. USENIX Association.

[27] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P’19)*, 2019.

[28] Kevin Laeufer, Jack Koenig, Donggyu Kim, Jonathan Bachrach, and Koushik Sen. Rfuzz: Coverage-directed fuzz testing of rtl on fpgas. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, 2018.

[29] J. Liedtke, N. Islam, and T. Jaeger. Preventing denial-of-service attacks on a /spl mu/-kernel for weboses. In *Proceedings. The Sixth Workshop on Hot Topics in Operating Systems (Cat. No.97TB100133)*, pages 73–79, 1997.

[30] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[31] Tao Lu. A survey on RISC-V security: Hardware and architecture. *CoRR*, abs/2107.04175, 2021.

[32] Mulong Luo, Wenjie Xiong, Geunbae Lee, Yueying Li, Xiaomeng Yang, Amy Zhang, Yuandong Tian, Hsien-Hsin S. Lee, and G. Edward Suh. Autocat: Reinforcement learning for automated exploration of cache-timing attacks. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 317–332, Los Alamitos, CA, USA, mar 2023. IEEE Computer Society.

[33] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse engineering intel last-level cache complex addressing using performance counters. In Herbert Bos, Fabian Monrose, and Gregory Blanc, editors, *Research in Attacks, Intrusions, and Defenses*, pages 48–65, Cham, 2015. Springer International Publishing.

[34] Z. McKevitt, A. Trivedi, and T. Lehman. Speccheck: A tool for systematic identification of vulnerable transient execution in gem5. In *2023 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 265–278, Los Alamitos, CA, USA, oct 2023. IEEE Computer Society.

[35] Kaveh Razavi Michele Marazzi. RISC-H: Rowhammer Attacks on RISC-V. In *Fourth Workshop on DRAM Security (DRAMSec)*, 2024.

[36] Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, dec 1990.

[37] Daniel Moghimi, Moritz Lipp, Berk Sunar, and Michael Schwarz. Medusa: Microarchitectural data leakage via automated attack synthesis. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1427–1444. USENIX Association, August 2020.

[38] Hamed Nemati, Pablo Buiras, Andreas Lindner, Roberto Guanciale, and Swen Jacobs. Validation of abstract side-channel models for computer architectures. In *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part I*, page 225–248, Berlin, Heidelberg, 2020. Springer-Verlag.

[39] O. Oleksenko, M. Guarnieri, B. Kopf, and M. Silberstein. Hide and seek with spectres: Efficient discovery of speculative information leaks with random testing. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1737–1752, Los Alamitos, CA, USA, may 2023. IEEE Computer Society.

[40] Oleksii Oleksenko, Christof Fetzer, Boris Köpf, and Mark Silberstein. Revizor: Testing black-box cpus against speculation contracts. In *27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*. ACM, March 2022.

[41] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. The spy in the sandbox: Practical cache attacks in javascript and their implications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 1406–1418, New York, NY, USA, 2015. Association for Computing Machinery.

[42] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of aes. In David Pointcheval, editor, *Topics in Cryptology – CT-RSA 2006*, pages 1–20, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[43] Gururaj Saileshwar and Moinuddin K. Qureshi. Cleanupspec: An "undo" approach to safe speculation. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 73–86, New York, NY, USA, 2019. Association for Computing Machinery.

[44] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. Checkmate: Automated synthesis of hardware exploits and security litmus tests. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 947–960, 2018.

[45] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Security Symposium*. USENIX Association, August 2018.

[46] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *41th IEEE Symposium on Security and Privacy (S&P'20)*, 2020.

[47] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue in-flight data load. In *S&P*, May 2019.

[48] Bhanu C. Vattikonda, Sambit Das, and Hovav Shacham. Eliminating fine grained timers in xen. In *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop*, CCSW '11, page 41–46, New York, NY, USA, 2011. Association for Computing Machinery.

[49] Zhenghong Wang and Ruby B. Lee. New cache designs for thwarting software cache-based side channel attacks. *SIGARCH Comput. Archit. News*, 35(2):494–505, jun 2007.

[50] Zhenghong Wang and Ruby B. Lee. A novel cache architecture with enhanced performance and security. In *2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 83–93, 2008.

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

[52] Yuan Xiao, Yinqian Zhang, and Radu Teodorescu. Speechminer: A framework for investigating and measuring speculative execution vulnerabilities. In *NDSS*. The Internet Society, 2020.

[53] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher Fletcher, and Josep Torrellas. Invisispec: Making speculative execution invisible in the cache hierarchy. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 428–441, 2018.

[54] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, l3 cache Side-Channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, San Diego, CA, August 2014. USENIX Association.

[55] Rui Zhang, Calvin Deutschbein, Peng Huang, and Cynthia Sturton. End-to-end automated exploit generation for validating the security of processor designs. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 815–827, 2018.