PCBleed: Fuzzing for CPU Bugs Through Use of Performance Counters

by

Natalie Muradyan

S.B., Computer Science and Engineering, Massachusetts Institute of Engineering, 2023

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2024

Authored by:	Natalie Muradyan Department of Electrical Engineering and Computer Science May 10, 2024
Certified by:	Mengjia Yan Assistant Professor of Electrical Engineering and Computer Science Thesis Supervisor
Accepted by:	Katrina LaCurts Chair, Master of Engineering Thesis Committee

PCBleed: Fuzzing for CPU Bugs Through Use of Performance Counters

by

Natalie Muradyan

Submitted to the Department of Electrical Engineering and Computer Science
on May 10, 2024 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

ABSTRACT

In recent years, the increasing complexity of hardware designs has given rise to a growing array of vulnerabilities and security threats, as exemplified by instances such as Spectre, Microarchitectural Data Sampling, and Zenbleed. The inherent permanence of hardware vulnerabilities poses a significant threat, making early identification crucial for preventing security compromises once a device is manufactured. However, identifying hardware vulnerabilities is challenging due to the large and complex design of current CPUs, resulting in a substantial search space and numerous unknowns.

This thesis proposes leveraging software fuzzing methods for hardware testing, focusing on the automated generation of instruction sequences that reveal hardware vulnerabilities. Unlike software fuzzing, hardware fuzzing faces challenges such as a lack of visibility into the microarchitectural processor states and difficulty in directing the search for test case generation.

To address these challenges, this research draws inspiration from software fuzzers that use insights into the internal workings of the software for effective test case generation. We propose PCBleed, a coverage-guided mutational hardware fuzzer that enhances CPU fuzzing by using hardware performance counters as insight into the CPU's behavior to improve test case generation. Since performance counters measure architectural events relevant to CPU performance, they provide insights that we use to estimate coverage, marking instruction sequences as novel. This approach aims to maximize the functionality exercised during hardware fuzzing, ultimately identifying interesting, bug-triggering behavior.

Our methodology is distinctive, utilizing performance counters for hardware fuzzing enhancement, and aligns with recent research findings that highlight the versatility of performance counters in debugging, dynamic software profiling, CPU power modeling, malware detection, and cache side-channel attack detection.

By incorporating performance counters into the hardware testing paradigm, this research seeks to contribute to the proactive fortification of hardware security through insightful analyses.

Thesis supervisor: Mengjia Yan
Title: Assistant Professor of Electrical Engineering and Computer Science

Acknowledgments

First and foremost, I would like to express my sincerest gratitude and appreciation to my research supervisor, Mengjia Yan, for her invaluable guidance, support, and mentorship throughout this research endeavor. It was thanks to her course on Secure Hardware Design that I got interested in this field, and I am thankful for the opportunity to work with her and her group, the MATCHA group (Microarchitecture ATtacks and CHAllenges). I am truly grateful for her dedication and for fostering an environment that facilitates growth and learning.

Second, I would like to extend my deepest gratitude to my research mentors, Peter Deutsch and Vincent Ulitzsch. Their invaluable contributions to this work cannot be overstated. From engaging in thought-provoking discussions on ideas and concepts to providing patient guidance during debugging sessions, their unwavering support has been crucial throughout this journey. I am grateful for their willingness to share their expertise, offer insightful advice, and help in shaping and refining this thesis. I appreciate their dedication and commitment to my growth as a researcher, and this accomplishment would not have been possible without their mentorship.

Additionally, I would like to express my sincere gratitude to my friends, colleagues, and past mentors from internships, research experiences, and high school clubs. My friends have been a constant source of encouragement, offering invaluable advice and a listening ear, especially during times when I felt stuck or faced challenges. My past mentors have played a pivotal role in my personal and professional growth with their guidance and wisdom. Their mentorship has not only imparted valuable knowledge but has also taught me the importance of staying organized, overcoming obstacles, and believing in my abilities.

I am also deeply grateful to the Armenian community, especially those who generously contributed donations to support my studies and my move to the United States at a young age, making it possible for me to pursue my dreams.

Last but not least, I would like to express my heartfelt gratitude to my family – my parents, my brother, and my extended family – for their unwavering love, support, and encouragement throughout this journey. I am forever thankful for their support when I embarked on the seemingly impossible idea of moving to the United States from Armenia at the young age of 17 to study at MIT. Their encouragement and understanding have been a constant source of strength, enabling me to stay true to myself. Their unconditional love and faith in my abilities have been the driving force behind this accomplishment, and I am eternally grateful for their sacrifices and unwavering support.

Contents

Title page 1

Abstract 3

Acknowledgments 5

List of Figures 9

List of Tables 11

1 Introduction 13

2 Background 17
2.1 Hardware Attacks . . . 17
2.2 Software Fuzzing . . . 17
2.3 Hardware Fuzzing . . . 18
2.4 Performance Counters . . . 20

3 High-Level Overview 21

4 Design and Implementation 25
4.1 Test Case Execution Engine . . . 25
4.2 Test Case Generation and Mutation Engine . . . 26
4.3 Novelty Measure Function . . . 28
4.4 Bug Detection Engine . . . 28

5 Evaluation 31
5.1 Recreating Zenbleed . . . 31
5.2 Novelty Measure Function Analysis . . . 33
5.3 Bug Detection Engine Performance Analysis . . . 35

6 Limitations 39
6.1 Bug Detection Guarantees . . . 39
6.2 No cycles in program control flow . . . 39

7 Related Work **41**
7.1 White-Box Hardware Fuzzing 41
7.2 Black-Box Hardware Fuzzing 41
7.3 Performance Counters 41

8 Conclusion **43**

A Zenbleed's zen2_leak_train_mm0 implementation **45**

B Recreating Zenbleed **47**

C Novelty Measure Evaluation **49**
C.1 Test Program 1 49
C.2 Test Program 2 49
C.3 Test Program 3 49
C.4 Test Program 4 50
C.5 Test Program 5 50

References **51**

List of Figures

2.1 Flowchart demonstrating a coverage-guided mutational software fuzzer's workflow 18

3.1 Flowchart demonstrating PCBleed's hardware fuzzing workflow 21

4.1 Flowchart demonstrating PCBleed's Mutate function's workflow, when mutating a basic block within an instruction sequence, inspired from SurgeFuzz [26] 27

5.1 Diagram showing a few performance event measurements for 6 different programs. The first 5 programs are Programs 1-5 in Listing C and the sixth program is Listing B 34

List of Tables

5.1 Evaluation results of running the Bug Detection Engine across different register value samples collected, presence of a concurrent process, and CPU cores. The reported metrics include execution time, number of bugs triggered, and bug occurrence rate. The bug occurrence rate measures the percentage of collected register value samples differing from the expected register values in a serialized (non-speculative) execution. 37

1. Introduction

Over recent years, hardware designs have undergone a surge in complexity, driven by the pursuit of elevated performance and the accommodation of intricate software demands. However, this heightened complexity is not without consequences; it gives rise to a growing array of vulnerabilities and potential threats. Instances such as Spectre [1], Microarchitectural Data Sampling [2], Zenbleed [3], among others, underscore the heightened security risks associated with these intricate designs. The inherent permanence of hardware vulnerabilities presents a significant threat, as rectification becomes impractical once a device is manufactured, potentially compromising the security of accompanying software. Therefore, it's crucial to identify hardware vulnerabilities early, before attackers do.

Unfortunately, identifying hardware vulnerabilities turns out to be rather difficult, and the current efforts prove inefficient, as suggested by the recently found vulnerabilities like Spectre [1], Zenbleed [3], etc. This is mainly because of the large and complex design of current CPUs, which leaves researchers and designers with a large search space, many complications, and unknowns. Therefore, automating the process of testing is an effort worth investigating, to help speed up hardware verification and find vulnerabilities early. To achieve this, we turn to well-established software testing methods as inspiration for approaching hardware testing.

Software fuzzing, for one, is a common tool used in software testing and has proven extremely useful in revealing software defects and vulnerabilities [4]. Fuzzing is an automated testing method that injects random (but often guided) inputs into a system to reveal some vulnerabilities. Since software fuzzing successfully identifies vulnerabilities in software in a short span, similarly, we envision applying fuzzing to hardware to help generate instruction sequences that reveal hardware vulnerabilities. However, hardware fuzzing is not as commonly used, and even though the seeds have been planted, there's still a lot to explore in this area, as it turns out that delving into the realm of hardware fuzzing proves to be challenging. There are a few reasons for this:

- Testing the hardware design on all possible test cases would require exponential time as the search space is too large. For example, a particular vulnerability may be triggered with a specific register state or a specific order of instructions, with Zenbleed [3] an instance of this, so finding all possible configurations is time-consuming. Similarly, the search space is large for software fuzzing. However, in software fuzzing, there exist tools that help direct the search of test cases, like branch coverage, and catching corner cases becomes easier, while in hardware fuzzing, estimating coverage is difficult, making efficient test case generation a challenging task.

- It's hard to direct the search for test case generation due to a lack of visibility into the microarchitectural processor state and detailed documentation. This is because, unlike software fuzzing, where coverage can be measured through different tools, in hardware fuzzing, we miss crucial information to understand how to improve the test case generation to cover more diverse instruction sequences.

- Race conditions add an extra layer of complexity to spotting vulnerabilities. This is because, in situations where vulnerabilities rely on multithreading, the processor might only reveal the vulnerability under specific interleaving conditions. Without precise control over how the threads interleave, running the application multiple times may be necessary to catch the vulnerability.

Thus, to make our hardware designs more secure, we need to carefully figure out these complexities. Thinking back to software fuzzers, we can get inspiration on how to deal with these issues. For example, many software fuzzers depend on insights into the internal workings of the software: a test case can be considered novel, meaning we want to continue fuzzing it, if it covers new program behavior, like branch coverage of the program [5]. This allows the use of coverage to guide the test case generation process, thus leading to the discovery of code instances that are bug-triggering in a shorter time span. Similarly, when it comes to testing black-box hardware, we need to understand what can be used as a coverage tool, and what can help us mark an instruction sequence as novel.

To answer this question, we turn to performance counters. Performance counters measure interesting architectural events relevant to the performance of the CPU, like the number of branch mispredictions, the number of L2 cache misses, the number of prefetcher hits, etc. Essentially, performance counters provide insights into the behavior of the CPU, which, we believe, can be used to estimate coverage and help us mark instruction sequences as novel.

The idea of using performance counters as insight into the CPU's behavior has already appeared in recent research findings. For instance, according to [6], performance counters have been used for debugging purposes, dynamic software profiling, CPU power modeling, malware detection, and malware defenses. Additionally, according to Kosasih et al. [7], performance counters have recently been used for cache side-channel attack detection.

Our approach differs in that we suggest using performance counters as insight into the CPU's behavior to enhance hardware fuzzing by using them to generate better test cases. Similar to maximizing code coverage in software fuzzing, for example with maximizing branch coverage [5], we want to exercise as much of the CPU's functionality as possible when hardware fuzzing. However, for a given black-box CPU, we do not have a metric similar to code coverage indicating how much of the CPU's functionality we have exercised. Performance counter profiles (PCPs) can help us to this end by giving some insights into the CPU's inner workings. We can then try to generate test cases (through mutation-based strategies) that exercise as much of the chip's functionality as possible.

The intuition behind this idea can be further explained with an example. In the search for bug-triggering instruction sequences, assume we generated a test that heavily utilizes the Arithmetic Logic Unit (ALU), a well-tested component. Spending too much time running test cases that mainly use the ALU may not lead to interesting behavior or yield results. Therefore, we'd like to test other components of the chip, hoping to find vulnerabilities.

However, it's hard to tell what the test case is exercising, and which part of the chip is being used. We propose to identify a test case that exercises new behavior by observing the performance counter values. If the values differ from the previously seen performance counter values, we can use this as a signal that the test case is using a new part of the chip and can give us another opportunity to find a bug.

2. Background

2.1 Hardware Attacks

Attacks like Spectre [1], Microarchitectural Data Sampling (MDS) [2], Zenbleed [3], among others, exploit information leaks, each targeting different behavior of the CPU. Zenbleed and similar vulnerabilities highlight the constant challenge of securing computing systems against a myriad of potential threats, necessitating innovative approaches for detection and mitigation. In response to this ever-evolving landscape of security concerns, fuzzing has emerged as a powerful technique to detect vulnerabilities.

Fuzzing can help detect various types of bugs. For instance, functional bugs like Zenbleed, Reptar [8], etc. were detected through fuzzing. Functional bugs violate the specification of the core, which can lead to incorrect results and open routes for using the incorrect results to leak sensitive data. Zenbleed, for one, impacts AMD's line of Zen 2 processors and can leak sensitive data by executing a short sequence of instructions, without any special system calls or privileges, due to incorrectly recovering from a mispredicted `vzeroupper` instruction.

Additionally, fuzzing can detect other security concerns, such as speculative side-channel leakage. For instance, Revizor [9] uses fuzzing techniques to detect vulnerabilities such as Spectre and MDS. Other fuzzers like SpecDoctor [10], IntroSpectre [11], and Transynther [12] use different fuzzing techniques to detect vulnerabilities.

2.2 Software Fuzzing

Fuzzing helps detect vulnerabilities by exhaustively running a wide range of test cases. Software fuzzing, in particular, has gained popularity among security and quality assurance experts and has become one of the most powerful test automation tools that discovers security vulnerabilities in software [4], with American Fuzzy Lop (AFL) [13] and LibFuzzer [14] as some of the most widely-used coverage-based fuzzers.

Fig. 2.1 demonstrates the general workflow of a coverage-guided mutation-based fuzzer. First, we feed some test cases to the fuzzer's test case queue that are either hand-picked or generated based on some templates or using other techniques. A test case in this case refers to a detailed description of inputs, execution conditions, testing methodology, and anticipated outcomes. Then a test case is picked from the queue and mutated by introducing small changes that may still keep the input valid, yet exercise new behavior [15]. Afterward, the fuzzer runs the test case and verifies whether the results are as anticipated, and if not, the test case is considered to have a bug and can be added to a set of buggy test cases. As the

fuzzer runs the test case, it also observes coverage. In software fuzzing, coverage is a measure of how much of the software's source code is exercised when running a test case. Coverage-guided fuzzing aims to find test cases that exercise new parts of the software's source code. When the fuzzer observes that the test case has new coverage (i.e. has novel behavior), it adds the test case to the test case queue for further fuzzing. Usually, the process is repeated until coverage is exhausted.

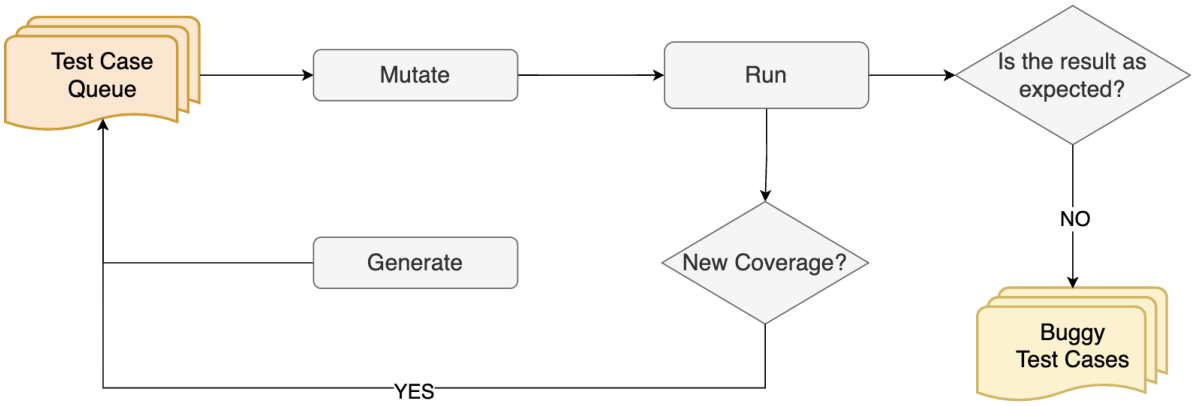

Figure 2.1: Flowchart demonstrating a coverage-guided mutational software fuzzer's workflow

American Fuzzy Lop (AFL) [13] is one of the most successful coverage-guided, mutational fuzzers. There have been a lot of publications on fuzzers using AFL as the baseline [16] [17], and AFL is attributed for finding many bugs in software like OpenSSH [18], Mozilla Firefox [19], etc., proving its success. AFL's coverage metric tries to capture branch (edge) coverage in the form of a tuple, `(branch_src, branch_dst)`, which denotes the branches the edge started and ended at. AFL keeps a count of how many times within one run of the program the edge was taken and then buckets the count to a power of two. This execution trace is then compared to all the previous execution traces. If novel behavior is detected, such as an edge falling into a novel bucket or hitting a previously unexplored edge, AFL considers the test case as exhibiting new behavior and adds it to the test case queue. This approach ensures ignoring changes within a single bucket (for example, a loop going from 47 cycles to 48) while marking transitions from one bucket to another (for example, if a code block is executed twice instead of the regular one hit) an interesting change in program control flow.

2.3 Hardware Fuzzing

Fuzzing, although more successfully implemented in software, has been used as a technique in hardware too. The efforts to bring fuzzing to hardware draw a lot of inspiration from software fuzzing, with some hardware fuzzers using popular software fuzzers in their implementations [12] [20]. However, since hardware fuzzing has not been researched as thoroughly as software fuzzing, exploring various approaches and techniques in the domain of hardware fuzzing is still evolving. Examples of hardware fuzzers illustrate the diversity in strategies, with

some adopting white-box approaches, like leveraging SystemVerilog assertion (SVA) [20], and others employing black-box methodologies, like template-based detection [12]. Black-box fuzzers monitor program inputs, outputs, and other measures they can gather without having access to a source code, while white-box fuzzers do fine-grained program analysis given a source code.

One interesting approach to black-box hardware fuzzing is Revizor [9]. Revizor uses Model-based Relational Testing (MRT) to detect microarchitectural information leakage in black-box CPUs. Oleksenko et al. use speculation contracts to identify what side effects of the program execution are allowed. Speculation contracts identify ISA operations as those that are observable by an attacker via a side channel or operations that can alter control or data flow speculatively. Thus, contracts provide a specification of the permitted microarchitectural side effects. Oleksenko et al.'s approach, MRT, is to search for violations by creating random instruction sequences and random inputs and check if the hardware traces violate the speculation contract traces. This approach allows for creating test cases that can lead to discovering new kinds of vulnerabilities. Oleksenko et al.'s approach to checking for violations is to compare the information exposed by the hardware traces with the information exposed by the contract. They gain insight into the microarchitectural changes of the hardware via some side-channel, producing the hardware trace. For example, they get insight into whether a specific cache set was accessed by the test case or not by using a side channel of priming the cache lines.

However, at the end of the fuzzing round, if Revizor doesn't identify any violations, it triggers a test case diversity analysis, which uses pattern combination coverage. Pattern coverage is a measure that quantifies the number of data and control dependencies likely to lead to pipeline hazards. Pattern coverage is then used as feedback to the test generator and if the coverage is not improved, Revizor reconfigures the test generator by increasing the number of instructions and basic blocks to enable the generation of more diverse tests.

Oleksenko et al. soon realized a performance limitation in the existing tools: ineffective test case generation due to the majority of generated test cases lacking the ability to reveal a speculative leak. Understanding the limitations of this approach of test generation, Oleksenko et al. came up with a follow-up paper [21] that built on top of Revizor to find more effective test cases, thus speeding up the testing campaign. They suggest a few tools for improving test case generation, such as speculation filtering, observation filtering, etc.

To achieve speculation filtering, they use speculation-related performance counters, only three or four performance counters from the dozens available. If during at least one of the test executions the speculation-related performance counter increases, the test case is passed to the next filter, the observation filter.

The observation filter first serializes the program by injecting a serialization fence, `lfence`, after every instruction. Then for all inputs in the test case, the filter executes both the serialized and the non-serialized programs and observes their hardware traces. If the traces differ for any of the inputs, the test case continues to the next steps, otherwise, it's discarded.

Although the proposed algorithm for better test case generation in the Revizor follow-up paper uses two filtering algorithms and contract-driven input generation (which ensures the effectiveness of the inputs), the generated test cases are always random and not mutations of previous test cases. More specifically, the test case generation process involves generating random basic blocks and adding terminators, such as jump instructions, that ensure the

control flow forms a Directed Acyclic Graph (DAG), meaning the generated programs have no loops. The basic blocks are filled with random instructions from a provided ISA subset.

2.4 Performance Counters

Performance counters monitor and measure interesting events that occur at the CPU level. Modern processors measure various performance events, such as cycles, cache hits, retired instructions, etc. Performance counters are generally used to understand the behavior or performance of a program on a given processor and depending on the machine, the available performance counters can vary. For instance, Intel provides a list of available performance monitor events that can be observed that vary from processor to processor [22].

Looking back at software fuzzers like AFL [13], we can draw parallels between AFL's coverage metric of hit counts that count how many times each edge was taken and performance counters that count how many times an event occurred. This parallel led us to look for research that used performance counters as a coverage metric in fuzzing. We soon found Zenbleed's [3] discovery quite interesting. Zenbleed uses a functional bug, an incorrect recovery from a mispredicted `vzeroupper`, to leak sensitive data. We found that T. Ormandy, the researcher who discovered Zenbleed, used performance counters as a tool for CPU coverage and fed the data to a fuzzer, which slowly discovered interesting test cases that would otherwise not be discovered. Although the discovery process was not documented thoroughly, the code for triggering Zenbleed is open-source [23] and later proved very helpful, as we used it to evaluate our fuzzer.

3. High-Level Overview

Due to evolving software demands, hardware designs are becoming increasingly sophisticated, introducing a multitude of vulnerabilities that, if left undetected, could lead to malicious attacks [1]. Therefore, it is crucial to identify hardware vulnerabilities early, before attackers do. In light of this, our proposed approach focuses on expediting the hardware verification process by harnessing the power of fuzzing—an automated testing method renowned for its effectiveness in uncovering vulnerabilities [4]. In particular, we propose fuzzing black-box CPUs using performance counters as a means to gain insight into the hardware's behavior.

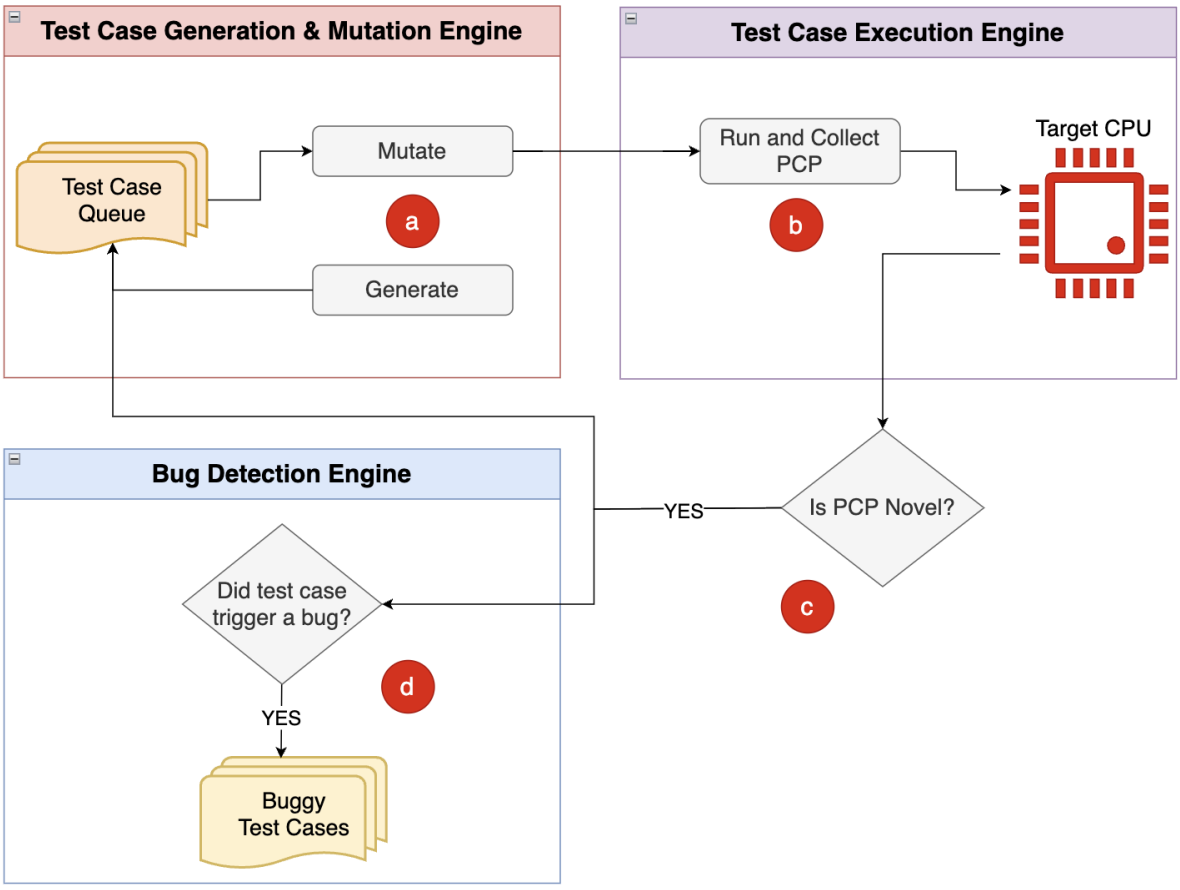

Figure 3.1: Flowchart demonstrating PCBleed's hardware fuzzing workflow

As such, our main objective is to enhance CPU fuzzing through performance counter

insights by using them to improve upon test case generation, drawing inspiration from coverage-guided software fuzzing techniques.

To achieve this objective, we introduce a comprehensive CPU fuzzing workflow, as depicted in Fig. 3.1, which serves as a roadmap for our methodology. This workflow encapsulates a series of steps, each carefully designed to address the challenges inherent in the fuzzing process.

The high-level idea of the design flow is to either start with or generate a queue of test programs at the Test Case Generation and Mutation Engine (GenMut Engine), pop one test from the queue, and mutate it (step a). Next, we pass the mutated program to the Test Case Execution Engine (Execution Engine), where we run it on the target CPU while observing its behavior by collecting its Performance Counter Profile (step b). Then, we use a novelty measure function to check if the test case exercises so-far unseen behavior, meaning the test case exercises a new CPU functionality (i.e. is novel) and if it's novel, add it to the test queue and pass it to the Bug Detection Engine for further analysis (step c). At the Bug Detection Engine, we check if the test case triggers a bug, and if it does, we add it to a queue of buggy test cases, which we later view manually to check for false positives or to understand how the bug was triggered (step d).

However, progressing through these stages presents its set of challenges, as each introduces complexities that require careful consideration. To provide a comprehensive understanding, let's explore a detailed explanation of each step in the proposed workflow.

Step a: At the core of our design, we initiate the process with a queue of test programs we'd like to fuzz-test. To generate the queue, we either handpick a few test cases we believe are good starting points for the mutator, for example, a known bug-triggering test case, or use a generation function to randomly generate programs given an ISA subset. These test programs should be semantically valid instruction sequences we believe are novel to look at, where a novel program is one that we suspect can be bug-triggering due to its novel Performance Counter Profile (PCP). The intuition is that a test case with a different PCP is a test case that exhibits different behavior since performance counters measure the CPU's internal events while executing a program. Therefore, a change in the performance counter's measurement may indicate a change in the CPU's behavior during execution, and make it a novel test case that could be bug-triggering or lead to a bug-triggering test case after a few mutations.

We then pop a test case from the queue and mutate it using a mutation function. The mutation function should be fine-tuned so as to mutate the instruction sequence in a way that is more likely to have a different PCP while keeping the program semantically valid.

Step b: After mutation, we pass the mutated program to the Execution Engine, where we run the test case on the target CPU. The execution of the program should be completely isolated, crashes should be handled properly, without affecting the workflow, and the engine should allow running a test case multiple times, every repetition starting with the given microarchitectural state, indicating register and memory values. Additionally, for every run of the instruction sequence, the Execution Engine should observe the behavior of running the program by collecting its PCP. The PCP is a vector of performance counter measurements for every performance event. Specifically, for every event, we get the difference between the performance counter value before and after executing the test case, thus observing the increase in the performance counter value during program execution (more details about

PCP collection are found in Section 4.1).

Step c: Following this, we use a novelty measure function to assess whether we want to continue fuzzing the test case by adding it to the test queue and also pass the program to the Bug Detection Engine for further analysis, or discard it. Our approach suggests assessing the novelty of the test case based on its PCP and marking it as novel if it has a PCP with different coverage than the other programs in the queue (more detailed discussion in Section 4.3). The idea is to find programs that are potentially bug-triggering or not bug-triggering but with a novel profile, which could later lead to bug-triggering programs after a few more fuzzing rounds.

Step d: Following execution, if the test case is marked novel, we pass it to the Bug Detection Engine, where we conduct an analysis, categorizing the test case as either bug-triggering or not. This helps us determine whether or not we want to mark the instruction sequence for further manual analysis, where we'll take a closer look to verify if it was a true positive and understand what triggered the bug.

Our outlined CPU fuzzing workflow introduces a systematic approach to identifying instruction sequences that are either bug-triggering or can lead to bug-triggering test cases. Our framework leverages a curated test program queue and a finely tuned mutation function inspired by established fuzzers. The execution process ensures a smooth workflow even in the presence of crashes, ensuring uninterrupted fuzzing while observing the hardware's behavior through collecting PCPs. The classification of buggy test cases and assessment of novelty contribute to a focused exploration of potential vulnerabilities. This cohesive methodology aims to fortify hardware security through proactive and insightful performance counter analyses.

4. Design and Implementation

This section presents the PCBleed framework that implements a black-box hardware fuzzer that uses performance counter data to improve test case generation. We present the design of all the components as demonstrated by Fig. 3.1 - the Test Case Execution Engine, the Test Case Generation and Mutation Engine (GenMut Engine), the Novelty Measure Function, and the Bug Detection Engine.

4.1 Test Case Execution Engine

We designed the Test Case Execution Engine to run on our black-box CPU, and for a given instruction sequence, execute it and observe the behavior by collecting performance counter data for all events - performance counter profile (PCP). The PCP we collect per instruction sequence measures for every performance event, by how much the performance counter value has increased during the program's execution. We built upon the open-source architecture of Revizor [9], used it as the starting block for our framework, made some adjustments, and added the PCP collection feature on top of it to build our Execution Engine.

The choice of using Revizor as the foundation of PCBleed was informed by various factors, including the accessibility of its source code [24], the inclusion of a code generation algorithm, support for multiple repetitions and crash recovery, and the provision of fundamental functionality for PCP collection, such as a function that adds necessary instructions to the test case before and after execution to collect a performance counter value.

Using Revizor as the baseline for PCBleed, we added PCP collection per repetition to complete the Execution Engine. Revizor only collects a few performance counter events per execution [24], while we wanted a more thorough profile. To bridge this gap, we first focused on understanding the intricacies of collecting performance counters.

To collect the performance counter for an event, we write and read from model-specific registers, MSRs, which are control registers commonly used for performance monitoring and other purposes. Specifically, we write to the MSRs that are designated for performance events. To get the performance counter value for a specific event, we write the event performance monitor select value and the unit mask, which details what event exactly to look for. For example, on our AMD Zen 2 CPU, to count the number of instruction fetches that hit in the L1 Instruction TLB (Translation Lookaside Buffer), we can write the performance monitor select value of `0x094` and unit mask value of `0x04` to specify that we want to count L1 Instruction TLB hits for 1-Gigabyte page sizes. On the other hand, the unit mask value of `0x02` would count the L1 Instruction TLB hits for 2-Megabyte page sizes.

The available performance events and the specific unit mask values may vary depending

on the CPU architecture and the available performance monitoring features. On our AMD Ryzen 7 4700G with Radeon Graphic machine, we collect a list of performance monitor counters with their available unit masks and descriptions using AMD uProf's AMDuProfPcm tool [25].

Revizor provides some basic functions that allow reading a performance counter value given its performance monitor counter (PMC) and unit mask. Revizor achieves this by writing the PMC and unit mask to the correct MSR that is designated for performance events and then reading from it. Although the function is there, Revizor only uses this when filtering for speculation leakage [21] (more in Section 2.3) and so the PMC and unit mask values are hardcoded at the beginning of the execution based on the CPU and Revizor keeps reading the same performance counter's value per test case to decide if it should be filtered or not.

To collect the performance counter values for all events, we modified Revizor's code to accommodate this. To instruct the framework on which performance event counter to get, we updated Revizor's Executor, the kernel module. We created a new sysfs file in the kernel that stored the PMC and unit mask information. This allows us to perform low-level instructions from the user level by reading or writing from the sysfs file. With this addition, we updated Revizor's code [24] to iterate over all performance events and unit masks that we collected using AMD uProf's AMDuProfPcm tool [25], execute the program, get the counter increase measure, and do this for every repetition. Since there are a few MSRs we can write to at a time, it's impossible to collect the whole performance counter profile in one execution of the program and thus, we execute the program multiple times.

4.2 Test Case Generation and Mutation Engine

The purpose of the Test Case Generation and Mutation Engine is to support a generation function that generates semantically valid programs given an Instruction Set Architecture (ISA) subset and a mutation function that mutates a given program by introducing small changes to the instruction sequence while keeping it a valid program. Our GenMut Engine's generation function uses Revizor's Test Case Generator [9] that ensures the control flow forms a Directed Acyclic Graph (DAG), which ensures that there are no loops in the program and that the program execution halts naturally without timing out. The GenMut Engine's mutation function is inspired by Surgefuzz's mutator [26] and mutates an instruction sequence by using random number generation and conditional statements to decide what type of mutation to perform and ensures that the DAG format is maintained.

Since we built our framework on top of Revizor, we referred to its open-source code to build PCBleed's generation function. Revizor, rooted in generating instruction sequences within a specified Instruction Set Architecture (ISA) subset, allowed us to tailor our generation function to consider various types of instructions, ranging from arithmetic to advanced vector operations. Hence, with little modification, we incorporated Revizor's generation routine in PCBleed.

PCBleed's mutation function, on the other hand, is inspired by SurgeFuzz's open-source code, specifically, the `mutate` function [27]. The mutator ensures that the control flow stays a DAG in order to avoid getting stuck in an infinite loop by iteratively mutating every basic

block in the instruction sequence, without introducing any new terminators and without mutating the existing terminators.

Fig. 4.1 gives a summary of the main idea behind how the mutation function mutates a basic block in an instruction sequence. There are five types of mutations that our mutator supports: swapping instructions, mutating an instruction, mutating operands of an instruction, inserting an instruction, and removing an instruction. When swapping instructions, we pick two random instructions in the basic block and swap them, ensuring that there are no dangling pointers. The second type of mutation, instruction mutation, randomly picks an instruction and replaces it with an instruction from the original ISA subset that is not a terminator. The third type, operand mutation, randomly picks an instruction and mutates its operands, but keeps the opcode the same. Lastly, inserting and removing instructions mutate the basic block by picking a random instruction to insert or remove, respectively.

Given the definitions provided above, we adopted the following approach for mutating a basic block (inspired by SurgeFuzz): we randomly decide whether to choose a mutation type and apply it to the basic block in a loop, or, in a loop, choose a mutation type and apply it to the basic block (Fig. 4.1).

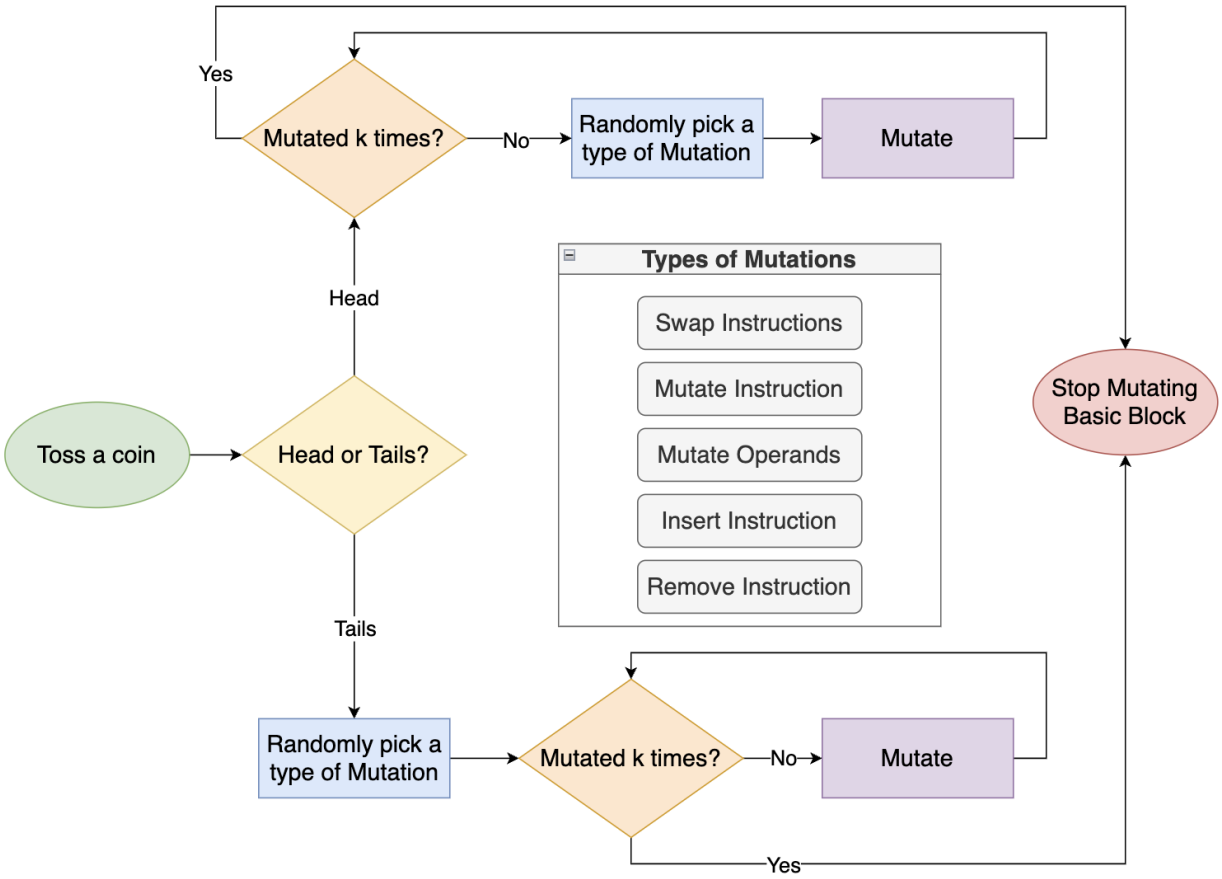

Figure 4.1: Flowchart demonstrating PCBleed's Mutate function's workflow, when mutating a basic block within an instruction sequence, inspired from SurgeFuzz [26]

4.3 Novelty Measure Function

With no metric of coverage when fuzzing CPUs in a black-box setting, we turn to performance counters to gain insight into the internal workings of the CPU while executing a program. Thus, our novelty measure function marks instruction sequences as novel based on their Performance Counter Profiles (PCPs), which we collect using our Execution Engine. In Section 4.1, we described that the PCP is a collection of measurements for every performance event, denoting the increase in performance counter value while executing the program. Given a PCP for a program that PCBleed is considering at the moment, the novelty measure function, inspired by AFL [13], buckets all the performance counter measurements to a power of two, as demonstrated in Eq. 4.1. Finally, the novelty measure function marks the program as novel if the bucketed PCP (BPCP) is different from all previously seen programs' BPCPs.

$$\text{BPCP} = \left[2^{\lfloor \log_2(\text{performance\_counter}) \rfloor} \text{ for performance\_counter in PCP}\right] \tag{4.1}$$

PCBleed's novelty measure function was inspired by AFL's metric for detecting interesting behavior [13], which is described in more detail in Section 2.2. The main idea is that changes within a bucket should be ignored, while transitioning from one bucket to another should be marked as new behavior, as it's trying to ignore empirically less notable changes.

PCBleed marks a test case as novel based on its PCP. When considering a PCP, for every performance counter measurement in the vector, we bucket the value to a power of two to get the bucketed performance counter profile, BPCP - a vector of bucketed measurement values. We then mark a test case as novel if its BPCP is different from all previously seen test case BPCPs. If the test case is marked novel, we add it to the Test Case Queue to be further mutated in the future fuzzing rounds and we pass the test case to the Bug Detection Engine to test it for any bugs.

4.4 Bug Detection Engine

The last step in PCBleed before fuzzing the next program is passing the test case that was classified as novel to the Bug Detection Engine. This engine aims to detect various types of bugs, including functional bugs and other security issues like speculative side-channel leaks, marking the test case as bug-triggering if any such bugs are found. PCBleed's Bug Detection Engine is inspired by T. Ormandy's Oracle Serialization that's mentioned in Zenbleed's discovery [3]. Similar to Ormandy's Oracle Serialization, our Bug Detection Engine compares the register values at the end of program execution to the register values collected at the end of executing the serialized version of the program, where a serialized program is the original instruction sequence injected with a serialization fence, `lfence`, after every instruction. Our Bug Detection Engine collects register values for the non-serialized program multiple times and compares all the results to the serialized version's register values. If a difference is found, we add the test case to the Buggy Test Cases, which we later assess manually for false positives or to understand where the bug stems from.

To collect register values, we initiate a new PCBleed executor instance without PCP collection but instead enter Revizor's General Purpose Register (GPR) mode [24], and collect

the register values for the `RAX`, `RBX`, `RCX`, `RDX`, `RSI`, and `RDI` registers. We collect the register values for the regular program multiple times, where the number of samplings can be determined by the user, while we collect the serialized program's register values only once. This is because the serialized program runs without speculation, while the regular program may trigger a functional bug or open a side channel depending on the branch predictor or other optimizations, so we run it multiple times to catch such behavior.

After every round of collecting register values for the regular test case, we compare the results with the serialized program's register values. If any difference is found, we mark the test case as bug-triggering and add it to the queue of Buggy Test Cases that we later analyze manually.

5. Evaluation

This section evaluates the PCBleed framework by recreating Zenbleed [3], a vulnerability affecting AMD's line of Zen 2 processors that is borne from a functional bug. We analyze PCBleed's Novelty Measure Function and Bug Detection Engine for performance and discuss the tradeoffs between execution time and effectiveness. We conducted our evaluation on an AMD Ryzen 7 4700G with Radeon Graphics (CPU family: 23, Model: 96), which is known to be affected by Zenbleed.

5.1 Recreating Zenbleed

Zenbleed [3], as discovered by T. Ormandy, is a vulnerability that affects the AMD Zen 2 processors and can leak sensitive data. Zenbleed takes advantage of a functional bug, the `vzeroupper` instruction's incorrect recovery from a mispredicted branch execution. The `vzeroupper` instruction zeroes the upper bits (position 128 and higher) of vector registers by setting the z-bit flag instead of writing bits. As Ormandy explains in the Zenbleed blog post [3], when `vzeroupper` is executed speculatively but needs to be reverted, the setting and unsetting of the z-bit flag can lead to something similar to a use-after-free vulnerability. One way to execute `vzeroupper` speculatively is through branch predictions, and if the branch mispredicts, the `vzeroupper` instruction needs to be reverted. This is the approach that Ormandy took when designing Zenbleed.

Zenbleed's open-source code [23] provides four implementations that trigger the vulnerability, with all four relying on the same base idea: a process may recover from a mispredicted `vzeroupper` incorrectly and leak some data. To trigger an incorrect recovery from a mispredicted `vzeroupper`, a series of events have to happen within a precise window. According to Ormandy, first, an XMM Register Merge Optimization has to happen, followed by a register rename, and, finally, a branch misprediction that executes the `vzeroupper` instruction.

To understand how Zenbleed is triggered, we can consider one of its implementations. For example, Listing A shows a somewhat simplified version of the `zen2_leak_train_mm0` implementation of Zenbleed. In this implementation, the instruction `cvtpi2pd` triggers the merge optimization, while `vmovdqa` triggers a register rename, and finally, `vzeroupper` is mispredicted. It's interesting to understand how the misprediction is triggered. The instruction `js .overzero` (line 16) checks for the sign flag (SF), and if it's set, meaning that the result of the previous arithmetic or logical operation was negative, then it jumps to the label `.overzero`. Since the previous arithmetic operation is `dec rcx` (line 9), the `js` instruction is essentially checking for the `rcx` register value to become negative. Therefore, for around 90 cycles (since `rcx` is initialized to be 90 on line 7) the branch predictor learns not to jump

to label `.overzero` and instead executes the instruction `vzeroupper` on line 17. However, once the `rcx` register value becomes negative and the sign flag is set, the instruction `js .overzero` succeeds. However, at that point, the branch predictor is trained not to jump to the label, so it mispredicts and runs the `vzeroupper` instruction by habit.

Afterward, the code jumps to the `.restart` label (line 6) as long as the instruction `jz .restart` (line 21) determines that the zero flag is set. This means that as long as the zero-flag is set (indicating that the `vptest` instruction didn't find any non-zero bits in `ymm0`), the loop will continue indefinitely and no information will leak. The line above, `vptest ymm0, ymm0` (line 20), sets the zero flag based on the contents of `ymm0`. If `ymm0` contains all zero bits, the zero flag will be set; otherwise, it will be cleared. The Zenbleed vulnerability is triggered when the zero flag is not set, which is incorrect because all the operations above make sure `ymm0` contains all zero bits. But if `vzeroupper` is mispredicted, then the z-bit flag will be set for `ymm0`, and any resources assigned to the register will be released. Then, another process can set `ymm0` to a non-zero value and so the loop (`jz .restart`) won't continue, and, instead, the next lines will be processed and the leaked data will be printed.

To trigger a Zenbleed-type bug in PCBleed, we adapted the open-source code for the implementation discussed above, `zen2_leak_train_mm0`, by keeping its main components. Listing B shows PCBleed's implementation of Zenbleed. The first and last few lines (lines 1-3 and 36-37) are necessary parts of every program in Revizor [9] and therefore in PCBleed. In the entry basic block (lines 5-9), we reset the AVX vector registers to zero, set the `RAX` register to zero, and move the value of `RAX` to the `MM0` register. The next basic block, lines 10-11, initialize the value of the register `RCX` to be 90, which is the number of times the code loops basic blocks 1 and 2 (lines 12-23) to train the branch predictor to think it needs to execute line 22, the `VZEROUPPER` instruction.

To see how the branch predictor gets trained, let's consider the contents of basic blocks 1 and 2. Line 13 decreases the `RCX` value by 1, lines 14-18 convert the packed integer value in `MM0` to double-precision floating-point values and stores them in `XMM4`, `XMM3`, `XMM2`, `XMM1`, and `XMM0`, with all these operations using the merge optimization. Then line 19 does a register rename (moves the data from `YMM0` to `YMM0`), and line 20 conditionally jumps to the third basic clock. This jump is only successful when the `RCX` register value becomes negative; otherwise, the code runs line 23, the `VZEROUPPER` instruction, and then line 23 jumps back to basic block 1 to continue looping. This loop ensures that for as long as the `RCX` register value is non-negative, the `VZEROUPPER` instruction on line 22 is executed after a merge optimization and a register rename. Once `RCX` is negative, line 20 should successfully jump to basic block 3. However, since for 90 times, the branch predictor has observed that the branch is not taken, it speculatively executes line 22, the `VZEROUPPER` instruction. Then, as described above, it may recover incorrectly, and the next basic blocks try to capture this case.

Once the code is at basic block 3 after recovering from a mispredicted branch and following the true label, we check if all data elements in `YMM0` are zero and set the zero flag (ZF) accordingly. Since in the beginning, in line 6, we zeroed all AVX vector registers, including `YMM0`, and we didn't touch the contents of `YMM0` again, then we'd expect that all data elements in `YMM0` are zero, and that the conditional branch in line 27 jumps to basic block 4, where we set the `RDI` register value to 1 (the choice of 1 is arbitrary). However, if something went wrong, and the `VZEROUPPER` instruction didn't recover correctly, giving another process access to register `YMM0`, the contents of the register may not all be zero. In this case, the

code jumps to basic block 5, where we set the RDI register value to 2 (the choice of 2 is arbitrary). At the end of executing the code, we check if the bug was triggered by observing the register values. We expect the RDI register to have a value of 1. But if the register value is instead 2, that is only possible if VZEROUPPER didn't recover correctly and another process had time to get access to YMM0, thus modifying some of its data elements to be non-zero.

PCBleed detects this bug when we pass the abovementioned test case (Listing B) to our Test Case Queue. When the test case reaches the Bug Detection Engine, we collect register values from the serialized version of the code. The serialized program consists of the original instruction sequence injected with an lfence after every instruction. Since the serialized code executes instructions sequentially, line 20 (Listing B) does not mispredict and skips executing VZEROUPPER speculatively, instead jumping to the correct label, basic block 3. As a result, the register value for RDI is deterministically set to 1.

However, the Bug Detection Engine also collects register values after running the non-serialized original program multiple times. During at least one of these runs, if VZEROUPPER recovers incorrectly and another process gains access to YMM0, the RDI register value becomes 2. In such cases, the Bug Detection Engine marks the test case as bug-triggering.

5.2 Novelty Measure Function Analysis

PCBleed's novelty measure function marks test cases as novel based on their Performance Counter Profiles (PCPs) since the performance counters give insight into the internal workings of the CPU as the test case is executed. The PCP of a program is a vector of measurements indicating the increase in performance counter value for every performance event. The novelty measure function buckets all the performance counter measurements of the PCP to a power of two and creates the Bucketed PCP (BPCP) according to Eq. 4.1. It then marks a test case novel if its BPCP is different from all previously seen BPCPs.

To evaluate PCBleed's novelty measure function, we look at the PCPs of six programs. The first program is Program 1 of Listing C and it does a few basic data transfer instructions using the instruction MOV. The second program, Program 2 of Listing C, does basic arithmetics with the ADD instruction. The third program, Program 3 of Listing C, consists of a few VZEROUPPER instructions, where VZEROUPPER is related to the AVX (Advanced Vector Extensions) instruction set. The fourth program, Program 4 of Listing C, has a few CVTPI2PD instructions and a VMOVDQA instruction, where CVTPI2PD prompts the XMM register merge optimization while VMOVDQA does a register rename. Program 5 (Listing C) is the same as Program 4 but without the VMOVDQA, register rename instruction. Finally, the sixth program we consider is Listing B, the instruction sequence that triggers Zenbleed, which includes a merge optimization, a register rename, and a mispredicted branch.

Fig. 5.1 plots a small part of the PCPs we collected for the abovementioned six programs. The figure only shows the performance counter values for select performance events that best demonstrate the differences between the programs. From Fig. 5.1, we see that Program 1 and Program 2 have very similar PCPs. Program 3's PCP (red bars) stands out for the performance event OCB.04 that counts the number of retired SSE (Streaming SIMD Extensions) instructions. Since Program 3 includes VZEROUPPER a few times, while the other programs either don't have the instruction or have it appear in a different frequency, it's

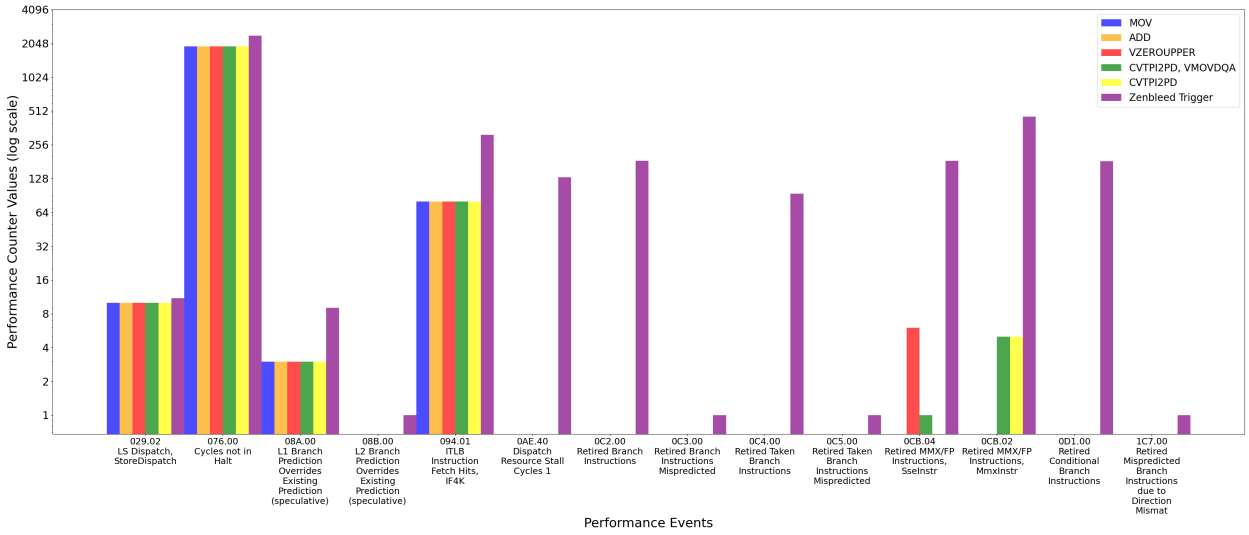

Figure 5.1: Diagram showing a few performance event measurements for 6 different programs. The first 5 programs are Programs 1-5 in Listing C and the sixth program is Listing B

considered novel among this distribution. The same performance event `0CB.04` has a high counter value for Program 6, since it runs various SSE instructions, such as `VZEROUPPER` and `VMOVDQA`. Additionally, there's a small counter value for Program 4, which includes the `VMOVDQA` instruction once.

Program 4 includes instructions that trigger XMM Register Merge Optimizations - `CVTPI2PD`, followed by a register rename operation - `VMOVDQA`. This makes the program's PCP a bit more diverse, as not only performance event `0CB.04` counts the `VMOVDQA` instruction but we also see performance event `0CB.02` with a green bar, which counts the number of retired MMX (MultiMedia Extensions) instructions. Because performance counter values can get more specific with the help of unit masks, the events `0CB.04` (unit mask of `0x04`) and `0CB.02`(unit mask of `0x02`) can keep track of different architectural events.

Program 5, although similar to Program 4, doesn't have the register rename instruction. PCBleed marks this program novel because the yellow bar in Fig. 5.1 only appears for the performance event `0CB.02` unlike Program 4, which has non-zero counter increases showing for both `0CB.02` and `0CB.04`.

Program 6, the instruction sequence that triggers Zenbleed, has the most diverse and interesting PCP. First, note that for performance event `076.00`, which counts the number of cycles not in halt, Program 6 has a slightly different counter than the other 5 programs. However, since the counters are both in the same power-of-two bucket when we look at the BPCPs, this difference is ignored, since the difference between the 10 and 11 cycles in halt is not as significant. Second, Program 6's PCP has many non-zero performance counter increase measures that indicate that there was a branch misprediction. For example, the performance events `08A.00` and `08B.00` count the number of times the speculative branch prediction is overridden by L1 Branch Prediction and L2 Branch Prediction, respectively. Both of these events have non-zero values in Fig. 5.1, which is not the case for the other 5 programs. Additionally, the performance event `0C3.00` counts the number of retired branch instructions of any type that were not correctly predicted, and this performance event has

a non-zero value for Program 6, demonstrating that the `JS` branch instruction on line 20 of Listing B is mispredicted.

These results demonstrate that PCBleed can notice differences in behavior. For instance, to trigger Zenbleed, we need to trigger an XMM Register Merge Optimization, a register rename, and a branch misprediction in order to execute `VZEROUPPER` speculatively and potentially recover incorrectly. Program 4 has the first two components, Program 5 has only the first component, Program 6 has all three components, and PCBleed marks all three programs as having novel behavior. Therefore, if during the fuzzing round, PCBleed took Program 5, mutated, and got Program 4, it would add this program to its Test Case Queue instead of discarding it. Then in future rounds, if PCBleed picked Program 4 and mutated it many times and got a program that met all three criteria for triggering Zenbleed, it would mark the test case as novel and send it to the Bug Detection Engine to be evaluated, where the bug could be detected.

5.3 Bug Detection Engine Performance Analysis

PCBleed's Bug Detection Engine compares the register values collected at the end of executing the original program to the register values collected at the end of executing the serialized version of the program (more details in Section 4.4). We evaluate the performance of the tool using the Listing B program that triggers Zenbleed. We base our evaluation on various factors, such as the number of register value samples collected, the impact of running a concurrent process, and the core on which we execute the program.

The first factor in our evaluation is to consider the number of register value samples. Our Bug Detection Engine collects register values for the non-serialized program multiple times and compares each register value sample to the serialized version's collected register values (more details in Section 4.4). Therefore, we consider how the number of collected register value samples improves the effectiveness of our tool in identifying a buggy program and how it affects the runtime of detecting a bug.

The second factor we consider in our evaluation is the impact of running a concurrent process. Zenbleed's open-source code [23] recommends running the program on a server that's not quiet. The idea is to run a concurrent process that potentially overwrites the `YMM0` register before the `VZEROUPPER` can recover (more details in Section 5.1). Lastly, we consider the core we execute the program on to evaluate whether the bug is triggered on a specific core or on all cores.

We evaluate the effectiveness of our Bug Detection Engine, depending on the factors, by looking at the number of samples that have a difference in the register values compared to the serialized program. This difference denotes the number of times a bug is triggered.

Table 5.1 shows some of the results we gathered by running the Listing B program with different factors. The table includes information such as the number of register value samples collected, whether we run a concurrent process, and the core we execute the program on. The results we collect and show in the table are the time to execute (in seconds), the number of bugs triggered (the number of samples that differ from the serialized program's sample), and the bug occurrence rate, which demonstrates what percentage of the collected register values differ from the serialized program's register values. For instance, Test 2 executes the

program on core 0, without running a concurrent process, and by getting 14000 register value samples. The results of this test show that the Bug Detection Engine ran for 75.4 seconds and triggered a bug 9 times out of the 14000, with a 0.064% bug occurrence rate.

The results show the tradeoff between guaranteeing to mark a program as bug-triggering and execution time. For instance, Test 1 runs the Listing B program on core 0, without running a concurrent process, and by collecting 7000 register value samples and does not mark the test case as bug-triggering. However, when in Test 2 we increase the number of register value samples to 14000, 9 of the samples turn out to be different from the serialized program's register value sample, marking the test case bug-triggering. Additionally, the execution time for Test 1, Test 2, and Test 3 demonstrate that increasing the number of collected register value samples affects the program execution time linearly. This is further demonstrated by Test 7 and Test 8.

Table 5.1 also demonstrates that running a concurrent process while executing the program can help improve the Bug Detection Engine mark programs more accurately. For example, Test 4 runs the Listing B program on core 0 while running a concurrent process, and by collecting 7000 register value samples and marks the test case as bug-triggering. If we compare this to Test 1, which differs from Test 4 only in that it does not run a concurrent process, Test 4 marks the test case bug-triggering correctly while Test 1 does not.

Additionally, we learned that depending on what core we execute the program on, we may need more register value samples to detect a bug, as demonstrated by Test 1 and Test 2, where we see that on core 0, we cannot catch a bug with 7000 register value samples but can detect 9 bugs with 14000 register value samples. It is also possible that some cores are not good for detecting the program bug, as is the case in Test 7 and Test 8, where we demonstrate that even increasing the number of register value samples from 7000 to 21000 does not help in detecting a bug on core 2.

Table 5.1: Evaluation results of running the Bug Detection Engine across different register value samples collected, presence of a concurrent process, and CPU cores. The reported metrics include execution time, number of bugs triggered, and bug occurrence rate. The bug occurrence rate measures the percentage of collected register value samples differing from the expected register values in a serialized (non-speculative) execution.

#	# of Register Value Samples	Concurrent Process	Core	Time (sec)	# of Bugs Triggered	Bug Occurrence Rate
1	7000	Not Running	0	37.7	0	0%
2	14000	Not Running	0	75.4	9	0.064%
3	21000	Not Running	0	110.8	16	0.076%
4	7000	Running	0	35.4	8	0.114%
5	7000	Not Running	1	38	3	0.042%
6	7000	Running	1	35.7	5	0.071%
7	7000	Running	2	33.5	0	0%
8	21000	Running	2	98.8	0	0%
9	7000	Running	3	34.9	7	0.1%
10	7000	Running	4	31.2	7	0.1%
11	7000	Running	5	31.7	19	0.271%
12	7000	Running	6	31.6	15	0.214%
13	7000	Running	7	31.9	7	0.1%
14	7000	Running	8	35.5	11	0.157%
15	7000	Running	9	35.8	3	0.042%
16	7000	Running	10	33.5	0	0%
17	7000	Running	11	35	4	0.057%
18	7000	Running	12	31.5	16	0.228%
19	7000	Running	13	31.7	11	0.157%
20	7000	Running	14	31.5	12	0.171%
21	7000	Running	15	31.4	13	0.185%

6. Limitations

6.1 Bug Detection Guarantees

As discussed in Section 5.3, the Bug Detection Engine's performance is affected by various factors, such as the number of register value samples collected, running concurrent processes, and the core on which the program is executed. Our results demonstrate that increasing the number of collected register value samples can help mark a program correctly (for example, Test 2 of Table 5.1) or may not help at all (for instance, Test 8 of Table 5.1). We also learned that increasing the number of collected register value samples affects program execution time linearly. Therefore, there's a tradeoff between accurately identifying a program as bug-triggering by increasing the number of register value samples and the program execution time. Moreover, increasing the number of samples does not guarantee to mark a program as bug-triggering.

Table 5.1 demonstrates that the number of register value samples we need to collect to correctly identify a bug-triggering program varies from core to core. Therefore, to mark a test case as bug-triggering with more accuracy, one approach is to run the Bug Detection Engine on all the cores. However, this is very time-consuming, since testing the program on 16 cores costs 16 times the cost of a regular check, which costs approximately the time to check the program after increasing the number of collected register value samples 16 times.

Lastly, our analysis in Section 5.3 also mentions that running a process in the background while executing the Zenbleed program (Listing B) improves the bug-detection mechanism since we see an increase in the number of samples that are different from the register values of the serialized program. For the Zenbleed program, our aim with the background process was to choose a process that could potentially overwrite the `YMM0` register value before the `VZEROUPPER` instruction could be rolled back from speculative execution. However, when testing a program, if we don't know what type of bug we're looking for (which is usually the case), it's difficult to pick a background process that is most helpful for the Bug Detection Engine. This is another limitation that does not guarantee the Bug Detection Engine's accuracy in correctly marking test cases as bug-free or bug-triggering.

6.2 No cycles in program control flow

As described in Section 4.2, PCBleed's Test Case Generation and Mutation engine ensures that the instruction sequence's control flow forms a Directed Acyclic Graph (DAG), meaning the instruction sequence has no loops. Revizor inspires the reason for taking such an approach

[9] and the idea is to avoid infinite loops, since such loops lead to timeouts and disrupt the fuzzing procedure. Additionally, even if an instruction sequence doesn't contain an infinite loop, but contains a loop of a certain number of rounds, if we don't add constraints to GenMut Engine's mutation function, the future mutations of the program may contain infinite loops.

In Section 5.1, we use Listing B as a program that triggers Zenbleed, and we provide a thorough explanation of how the bug is triggered and what components are at play. We discuss that Zenbleed can be triggered with an XMM Register Merge Optimization, followed by a register rename, and a branch misprediction that executes `VZEROUPPER` speculatively, which, in order, may recover incorrectly and leak information. The last component for triggering Zenbleed is to execute `VZEROUPPER` speculatively and roll it back. It turns out quite difficult to control the execution process and force this. T. Ormandy achieved this by forcing a branch misprediction [3], which executes `VZEROUPPER` speculatively and then rolls it back since the branch was mispredicted. Forcing a branch misprediction is not so simple, and the way Ormandy achieves this is by training the branch predictor. Ormandy suggests running a conditional branch instruction in a loop multiple times, training the branch predictor not to take the branch. On the last round of the loop, when the branch should be taken, the branch predictor mispredicts.

Listing B adapts Zenbleed's open-source implementation, `zen2_leak_train_mm0`, by keeping its main components. To execute `VZEROUPPER` speculatively, basic block `.bb_main.1` runs in a loop and trains the conditional branch on line 20 that the branch should not be taken since the register value `RCX` is non-negative. However, once the `RCX` register becomes negative and the loop ends, the branch predictor predicts that the branch `.bb_main.3` is not taken, since it predicted so for the many past loops, and line 22, the `VZEROUPPER` instruction, executes speculatively.

Since many bugs surface in the presence of speculation [3][1], forcing some instructions to execute speculatively may help detect bugs. The above case study demonstrates that training the branch predictor is one approach for forcing speculative execution. Specifically, training the branch predictor by running the conditional branch that we intend to mispredict in a loop can force the speculative execution of an instruction. However, including a loop in the instruction sequence means disrupting the control flow being a DAG, which is a requirement we maintain throughout the fuzzing rounds in our Test Case Generation and Mutation Engine. This requirement limits the set of test cases PCBleed considers in the Bug Detection Engine, thus making it difficult to arrive at a test case that the Bug Detection Engine can mark as bug-triggering.

One approach to mitigate this limitation is to add an additional step in the Bug Detection Engine that modifies the instruction sequence that's being considered by adding loops that can help train the branch predictor. Since the Bug Detection Engine's job is only to mark test cases as bug-triggering, we can disrupt the acyclic control flow without worrying about future mutations of the GenMut Engine getting affected (the test case is never passed to the GenMut Engine from the Bug Detection Engine), as long as the introduced loops are not infinite loops.

7. Related Work

7.1 White-Box Hardware Fuzzing

Trippel et al. [20] approach white-box fuzzing by translating RTL hardware to a software model and fuzzing that model directly. The researchers employ SystemVerilog assertion (SVA) to detect bugs in the hardware design. Nevertheless, SVA requires manual pre-instrumentation by the developer, and Trippel et al. were able to identify known bugs rather than exploring novel test cases that can lead to unknown bugs.

Another white-box fuzzer, Introspectre [11] tackles the issue of limited visibility into the microarchitectural processor state by integrating Introspectre into the register transfer level (RTL) design flow. However, Introspectre focuses on finding Meltdown-type leaks and therefore not exhausting the vast space of potential test cases.

7.2 Black-Box Hardware Fuzzing

Turning to black-box fuzzers, there are examples like Transynther [12] which takes a fuzzing-based approach to perform an in-depth analysis of Meltdown-style attacks and generate new subvariants. One limitation of Transynther is that it does random mutations without having fine-grained feedback such as code coverage. Therefore, the starting test case has to be chosen carefully, otherwise, the fuzzer won't converge to test cases that trigger vulnerabilities. Additionally, this is a template-based approach that looks for vulnerabilities of the same nature as some known code triggering speculation leading to leaks and therefore does not allow for identifying new kinds of vulnerabilities.

7.3 Performance Counters

Some recent works build on the idea of using performance counters as a means to get insight into the hardware's behavior. These works have found performance counters helpful in identifying cache side-channel attacks [7], [28], and in developing malware detection and defense mechanisms [6]. For instance, a popular approach involves using machine learning models that leverage the performance counter profile to identify vulnerabilities [29][28].

8. Conclusion

In this thesis, we propose PCBleed, a coverage-guided mutational hardware fuzzer that enhances CPU fuzzing by using hardware performance counters as insight into the CPU's behavior to improve test case generation. Our approach leverages the power of fuzzing, an automated testing method proven effective in uncovering vulnerabilities, by incorporating performance counter insights to guide the fuzzing process.

PCBleed introduces a comprehensive CPU fuzzing workflow that encompasses test case generation, mutation, execution, novelty assessment, and bug detection. The workflow begins with a queue of initial test programs, which are iteratively mutated and executed on the target CPU while collecting performance counter profiles (PCPs). These PCPs provide valuable insights into the CPU's behavior during execution, enabling the assessment of test case novelty based on their coverage of unique CPU functionalities.

By prioritizing test cases with novel PCPs, PCBleed focuses its efforts on exploring potentially bug-triggering instruction sequences or those that could lead to such sequences through further mutations. This targeted approach increases the likelihood of uncovering vulnerabilities while optimizing the fuzzing process's efficiency.

The integration of performance counter analysis into the fuzzing process represents a novel contribution to the field of hardware security, enabling a more informed and directed exploration of the CPU's behavior. Through this thesis, we have demonstrated the feasibility and potential of PCBleed in enhancing CPU fuzzing by leveraging hardware performance counters. The proposed approach holds promise for improving the effectiveness of hardware verification, ultimately contributing to the development of more secure and reliable computing systems.

A. Zenbleed's zen2_leak_train_mm0 implementation

```
zen2_leak_train_mm0:
    vzeroall
    xor         rax, rax
    movd        mm0, rax
    align       64
.restart:
    mov         rcx, 90
.again:
    dec         rcx
    cvtpi2pd    xmm4, mm0
    cvtpi2pd    xmm3, mm0
    cvtpi2pd    xmm2, mm0
    cvtpi2pd    xmm1, mm0
    cvtpi2pd    xmm0, mm0
    vmovdqa     ymm0, ymm0
    js          .overzero
    vzeroupper
.overzero:
    jns         .again
    vptest      ymm0, ymm0
    jz          .restart
    vmovdqu     [rdi], ymm0
    ret
    hlt
```

B. Recreating Zenbleed

```
.intel_syntax noprefix
    MFENCE # instrumentation
.test_case_enter:
    .function_main:
        .bb_main.entry:
            VZEROALL
            XOR RAX, RAX
            MOVD MM0, RAX
            JMP .bb_main.0
        .bb_main.0:
            MOV RCX, 90
        .bb_main.1:
            DEC RCX
            CVTPI2PD XMM4, MM0
            CVTPI2PD XMM3, MM0
            CVTPI2PD XMM2, MM0
            CVTPI2PD XMM1, MM0
            CVTPI2PD XMM0, MM0
            VMOVDQA YMM0, YMM0
            JS .bb_main.3
        .bb_main.2:
            VZEROUPPER
            JNS .bb_main.1
        .bb_main.3:
            VPTEST YMM0, YMM0
            VPTEST YMM0, YMM0
            JZ .bb_main.4
            JMP .bb_main.5
        .bb_main.4:
            MOV RDI, 1
            JMP .bb_main.exit
        .bb_main.5:
            MOV RDI, 2
            JMP .bb_main.exit
        .bb_main.exit:
.test_case_exit:
MFENCE # instrumentation
```

C. Novelty Measure Evaluation

C.1 Test Program 1

```
.intel_syntax noprefix
MFENCE # instrumentation
.test_case_enter:
    MOV RAX, 1
    MOV RBX, 2
    MOV RCX, 3
    MOV RDX, 4
    MOV RSI, 5
    MOV RDI, 6
.test_case_exit:
MFENCE # instrumentation
```

C.2 Test Program 2

```
.intel_syntax noprefix
MFENCE # instrumentation
.test_case_enter:
    ADD RAX, 1
    ADD RBX, 2
    ADD RCX, 3
    ADD RDX, 4
    ADD RSI, 5
    ADD RDI, 6
.test_case_exit:
MFENCE # instrumentation
```

C.3 Test Program 3

```
.intel_syntax noprefix
MFENCE # instrumentation
.test_case_enter:
```

```
    VZEROUPPER
    VZEROUPPER
    VZEROUPPER
    VZEROUPPER
    VZEROUPPER
    VZEROUPPER
.test_case_exit:
MFENCE # instrumentation
```

C.4 Test Program 4

```
.intel_syntax noprefix
MFENCE # instrumentation
.test_case_enter:
    CVTPI2PD XMM4, MM0
    CVTPI2PD XMM3, MM0
    CVTPI2PD XMM2, MM0
    CVTPI2PD XMM1, MM0
    CVTPI2PD XMM0, MM0
    VMOVDQA YMM0, YMM0
.test_case_exit:
MFENCE # instrumentation
```

C.5 Test Program 5

```
.intel_syntax noprefix
MFENCE # instrumentation
.test_case_enter:
    CVTPI2PD XMM4, MM0
    CVTPI2PD XMM3, MM0
    CVTPI2PD XMM2, MM0
    CVTPI2PD XMM1, MM0
    CVTPI2PD XMM0, MM0
.test_case_exit:
MFENCE # instrumentation
```

References

[1] P. Kocher, J. Horn, A. Fogh, *et al.*, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 1–19. DOI: 10.1109/SP.2019.00002.

[2] *Mds - microarchitectural data sampling*. URL: https://docs.kernel.org/admin-guide/hw-vuln/mds.html.

[3] T. Ormandy, *Zenbleed*. URL: https://lock.cmpxchg8b.com/zenbleed.html.

[4] A. Takanen, J. D. Demott, C. Miller, and A. Kettunen, *Fuzzing for software security testing and quality assurance*. Artech House, 2018.

[5] C. Lemieux, *Expanding the reach of coverage-guided fuzzing*. URL: https://www.code-intelligence.com/blog/caroline-lemieux-expanding-fuzzing.

[6] S. Das, J. Werner, M. Antonakakis, M. Polychronakis, and F. Monrose, "Sok: The challenges, pitfalls, and perils of using hardware performance counters for security," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 20–38. DOI: 10.1109/SP.2019.00021.

[7] W. Kosasih, *A tale of unrealized hope: Hardware performance counter against cache attacks*, 2023. arXiv: 2311.10542 [cs.CR].

[8] T. Ormandy, *Reptar*. URL: https://lock.cmpxchg8b.com/reptar.html.

[9] O. Oleksenko, C. Fetzer, B. Köpf, and M. Silberstein, *Revizor: Testing black-box cpus against speculation contracts*, 2022. arXiv: 2105.06872 [cs.CR].

[10] J. Hur, S. Song, S. Kim, and B. Lee, "Specdoctor: Differential fuzz testing to find transient execution vulnerabilities," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '22, Los Angeles, CA, USA: Association for Computing Machinery, 2022, pp. 1473–1487, ISBN: 9781450394505. DOI: 10.1145/3548606.3560578. URL: https://doi.org/10.1145/3548606.3560578.

[11] M. Ghaniyoun, K. Barber, Y. Zhang, and R. 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)*, 2021, pp. 874–887. DOI: 10.1109/ISCA52012.2021.00073.

[12] D. Moghimi, M. Lipp, B. Sunar, and M. Schwarz, "Medusa: Microarchitectural data leakage via automated attack synthesis," in *29th USENIX Security Symposium (USENIX Security 20)*, USENIX Association, Aug. 2020, pp. 1427–1444, ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/moghimi-medusa.

[13] *Afl - american fuzzy lop*. URL: https://lcamtuf.coredump.cx/afl/.

[14] *Libfuzzer – a library for coverage-guided fuzz testing*. URL: https://llvm.org/docs/LibFuzzer.html.

[15] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler, "Mutation-based fuzzing," in *The Fuzzing Book*, Retrieved 2023-11-11 18:18:06+01:00, CISPA Helmholtz Center for Information Security, 2023. URL: https://www.fuzzingbook.org/html/MutationFuzzer.html.

[16] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++ : Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, USENIX Association, Aug. 2020. URL: https://www.usenix.org/conference/woot20/presentation/fioraldi.

[17] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Krügel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," in *Network and Distributed System Security Symposium*, 2016. URL: https://api.semanticscholar.org/CorpusID:2388545.

[18] D. Miller, *Openssh null pointer dereference crash in key loading*. URL: https://lists.mindrot.org/pipermail/openssh-commits/2014-November/004134.html.

[19] M. Zalweski. URL: https://lcamtuf.blogspot.com/2014/09/cve-2014-1564-uninitialized-memory-when.html.

[20] T. Trippel, K. G. Shin, A. Chernyakhovsky, G. Kelly, D. Rizzo, and M. Hicks, "Fuzzing hardware like software," in *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA: USENIX Association, Aug. 2022, pp. 3237–3254, ISBN: 978-1-939133-31-1. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/trippel.

[21] O. Oleksenko, M. Guarnieri, B. Köpf, and M. Silberstein, *Hide and seek with spectres: Efficient discovery of speculative information leaks with random testing*, 2023. arXiv: 2301.07642 [cs.CR].

[22] URL: https://perfmon-events.intel.com/.

[23] T. Ormandy, *The vzeroupper instruction on amd zen 2 can leak register file state*, 2023. URL: https://github.com/google/security-research/tree/master/pocs/cpus/zenbleed.

[24] O. Oleksenko, *Revizor - a fuzzer to search for microarchitectural leaks in cpus*. URL: https://github.com/microsoft/sca-fuzzer/tree/main.

[25] AMD, *Uprof*. URL: https://www.amd.com/en/developer/uprof.html.

[26] Y. Sugiyama, R. Matsuo, and R. Shioya, "Surgefuzz: Surge-aware directed fuzzing for cpu designs,"

[27] Shioya-Lab-Public, *Surgefuzz: Surge-aware directed fuzzing for cpu designs (iccad 2023)*. URL: https://github.com/shioya-lab-public/surgefuzz.

[28] M. Mushtaq, A. Akram, M. K. Bhatti, M. Chaudhry, V. Lapotre, and G. Gogniat, "Nights-watch: A cache-based side-channel intrusion detector using hardware performance counters," in *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*, ser. HASP '18, Los Angeles, California: Association for Computing Machinery, 2018, ISBN: 9781450365000. DOI: 10.1145/3214292.3214293. URL: https://doi.org/10.1145/3214292.3214293.

[29] M. Chiappetta, E. Savas, and C. Yilmaz, *Real time detection of cache-based side-channel attacks using hardware performance counters*, Cryptology ePrint Archive, Paper 2015/1034, https://eprint.iacr.org/2015/1034, 2015. URL: https://eprint.iacr.org/2015/1034.