Maintaining Security in the Era of Microarchitectural Attacks

by

Oleksii Oleksenko

Dissertation
submitted for the degree of
Doctor of Philosophy (PhD)

Technische Universität Dresden

Faculty of Computer Science
Institute of Systems Architecture
Chair of Systems Engineering

Supervisor:	Prof. Dr. (PhD) Christof Fetzer
Subject Expert:	Prof. Dr. (PhD) Mark Silberstein

Submitted: June 24, 2021

Acknowledgments

This thesis would not have been possible without the help and support of many people.

First, I want to express my deep gratitude to Prof. Christof Fetzer. You challenged me with new ideas and helped me develop my own, especially at the beginning of my PhD studies. I might have occasionally been stubborn and skeptical, but you had the patience for our endless discussions, which ended up with several great projects. You also gave me complete scientific freedom and allowed me to pursue my ideas, which I truly appreciate.

I am equally grateful to my Fachreferent, Prof. Mark Silberstein, who helped me enter the brave new world of microarchitectural security, and with whom we together developed the projects I'm especially proud of. Without your attention to detail, the challenge you always provided, and all the time you put into our papers, none of them would be possible.

I want to thank Dmitrii Kuvaiskii for teaching me the craft of research. I thank Prof. Pramod Bhatotia for jump-starting my PhD with an ambitious project. I'm grateful to Boris Köpf for showing me the big picture of microarchitectural security and for the numerous discussions we had. I thank Prof. Pascal Felber for helping at the early stages of my PhD.

I thank my colleagues and close friends—you helped me a lot, and I've learned a ton from you. I'm especially grateful to my office mates, Bohdan Trach and Robert Krahn, with whom I shared countless endeavors on my quest to PhD, and who helped me grow as both a scientist and a person. I thank Franz Gregor, Dmitrii Pukhkaiev, Do Le Quoc, Irina Karadschow, Andre Martin, Sergei Arnautov, Wojciech Ozga, Rasha Faqeh, Saidgani Musaev, Anna Galanou, Pamenas Kariuki, and Tobias Reiher for their help through the years. I also owe thanks to all other people I worked with—you were great colleagues.

Most importantly, I want to thank my amazing parents and my brother. Thank you, dad, for inspiring me and showing me how to do the hard but necessary work. Thank you, mom, for your infinite understanding and support. Thank you, Illia, for being awesome.

Finally, my work was built on the solid foundation of Bash one-liners, tiny Python and AWK scripts, and countless cups of coffee; I'm grateful to the people who invented these things. And, of course, I want to thank the proverbial Reviewer #2, for no true achievement is possible without a nemesis.

Abstract

Shared microarchitectural state is a target for side-channel attacks that leverage timing measurements to leak information across security domains. These attacks are further enhanced by speculative execution, which transiently distorts the control and data flow of applications, and by untrusted environments, where the attacker may have complete control over the victim program. Under these conditions, microarchitectural attacks can bypass software isolation mechanisms, and hence they threaten the security of virtually any application running in a shared environment.

Numerous approaches have been proposed to defend against microarchitectural attacks, but we lack the means to test them and ensure their effectiveness. The users cannot test them manually because the effects of the defences are not visible to software. Testing the defences by attempting attacks is also suboptimal because the attacks are inherently unstable, and a failed attack is not always an indicator of a successful defence. Moreover, some classes of defences can be disabled at runtime. Hence, we need automated tools that would check the effectiveness of defences, both at design time and at runtime. Yet, as it is common in security, the existing solutions lag behind the developments in attacks.

In this thesis, we propose three techniques that check the effectiveness of defences against modern microarchitectural attacks. *Revizor* is an approach to automatically detect microarchitectural information leakage in commercial black-box CPUs. *SpecFuzz* is a technique for dynamic testing of applications to find instances of speculative vulnerabilities. *Varys* is an approach to runtime monitoring of system defences against microarchitectural attacks.

We show that with these techniques, we can successfully detect microarchitectural vulnerabilities in hardware and flaws in defences against them; find unpatched instances of speculative vulnerabilities in software; and detect attempts to invalidate system defences.

Publications

The content of this thesis is based on the following publications:

1. **Revizor: Fuzzing for Leaks in Black-box CPUs.** Oleksenko O., Fetzer C., Köpf B., and Silberstein M. *Under Submission*

2. **SpecFuzz: Bringing Spectre-type Vulnerabilities to the Surface.** Oleksenko O., Trach B., Silberstein M., and Fetzer C. *In Proceedings of the USENIX Security Symposium (USENIX Security), 2020.*

3. **Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks.** Oleksenko O., Trach B., Krahn R., Martin A., Silberstein M., and Fetzer C. *In Proceedings of the USENIX Annual Technical Conference (USENIX ATC), 2018.*

4. **You Shall Not Bypass: Employing Data Dependencies to Prevent Bounds Check Bypass.** Oleksenko O., Trach B., Reiher T., Silberstein M., and Fetzer C. *ArXiv preprint, arXiv:1805.08506, 2018.*

5. **Fex: A Software Systems Evaluator.** Oleksenko O., Kuvaiskii D., Bhatotia P., and Fetzer C. *In Proceedings of the International Conference on Dependable Systems and Networks (DSN), 2017.*

The following publications were produced while working on the thesis, but are not included in the thesis itself:

6. **Intel MPX Explained: A Cross-layer Analysis of the Intel MPX System Stack.** Oleksenko O., Kuvaiskii D., Bhatotia P., Felber P., and Fetzer C. *In Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS), 2018.*

7. **SGXBounds: Memory Safety for Shielded Execution.** Kuvaiskii, D., Oleksenko O., Arnautov S., Trach B., Bhatotia P., Felber P., and Fetzer C. *In Proceedings of the ACM European Conference on Computer Systems (EuroSys), 2017.*

8. **Clemmys: Towards Secure Remote Execution in FaaS.** Trach, B., Oleksenko O., Gregor, F, Bhatotia P., and Fetzer C. *In Proceedings of the ACM International Conference on Systems and Storage (SYSTOR), 2019.*

9. **T-Lease: A Trusted Lease Primitive for Distributed Systems.** Trach B., Faqeh R., Oleksenko O., Ozga W., Bhatotia P., and Fetzer C. *In Proceedings of the ACM Symposium on Cloud Computing, 2020.*

10. **Elzar: Triple modular redundancy using Intel AVX (Practical Experience Report).** Kuvaiskii D., Oleksenko O., Bhatotia P., Felber P., and Fetzer C. *In Proceedings of the International Conference on Dependable Systems and Networks (DSN), 2016.*

Contents

List of Figures

List of Tables

Chapter 1

Introduction

When a person steps on the ground, he leaves a footprint. This side effect may seem harmless in most life situations, yet in the rare case when somebody is chasing the person, the footprint may cost him dearly.

The same happens when a program runs on a computer. The program's actions leave traces, such as when a load affects CPU caches or increases the memory bus traffic. If another, malicious program runs on the same computer, it could read the traces and spy on the victim program. This spying method is called a microarchitectural side-channel attack (hereafter, microarchitectural attack), and the overarching goal of this thesis is to prevent such attacks.

The research on microarchitectural attacks started with a discovery that cache traces of some cryptographic functions change depending on the encryption keys [189, 241, 191, 188]; accordingly, an attacker who can observe the traces, can derive the keys from them. The range of targets later grew out of cryptography into many more applications, such as browsers [185], image rendering [117], and even operating systems [109, 158]. The attack toolbox was growing too: New attack variants were expanding to more types of traces [17, 18, 273, 44] and improving on their precision [276, 157, 104]. Today, microarchitectural attacks turned into a universal primitive for bypassing security boundaries, and they threaten the security of almost any shared system. Modern variants circumvent such security mechanisms as virtual machine isolation [131, 262], trusted execution environments [117, 56, 244, 57, 212], browser sandboxes [185, 142], memory safety [142, 76], secure languages [165], control-flow integrity [165], kernel memory isolation [121, 158, 57], and some could even read physical memory [121, 262].

Despite this threat, hardware and software developers largely dismissed the issue until recently [180, 114, 107]. Very few defences got deployed, except for a handful of high-security systems [93] and cryptographic libraries [1, 3]. The basis for it was a commonlyheld belief that microarchitectural attacks present only a theoretical threat, as they are rarely used in practical exploits [230, 90]. This belief is dangerous: First, practical exploits *do* begin to appear [203, 125], and most security-critical systems are unprepared for them. Second, the existing academic proposals of defences against microarchitectural attacks [222, 248, 161, 183] virtually never get tested in practice.

The tide began to change only in 2018, with the publication of Spectre [142] and Meltdown [158]. These vulnerabilities showed the true harmful potential of microarchitectural attacks (e.g., Meltdown could leak physical memory contents) and forced the hardware and software communities to react with numerous security patches [19, 20, 109, 250].

This first large-scale deployment highlighted a major issue: *We lack the tools to test and validate defences against modern microarchitectural attacks*. The only available testing method was manual, which is inherently incomplete, and it repeatedly led to flaws in the defences [141, 39].

In this thesis, we propose a set of methods and tools to automatically check the defence effectiveness. Yet before we dive into the details, let us first consider how real is the threat of microarchitectural attacks, and whether the protection is necessary in the first place.

1.1 The Rise of Offensive Cybersecurity

The first question we have to consider is: Will attackers eventually start employing microarchitectural attacks in the real world? We have to consider it because if the answer is “no”, there is no point in developing defences against them.

In the past, the commonly accepted answer was: not likely. Software was—and still is—full of conventional vulnerabilities such as buffer overflows or improper input validation [179]. There are plenty of tried-and-true techniques for exploiting them [184, 220] and tools that help with the exploitation [10, 9]. At first glance, it seems unreasonable to search for new attack vectors when the old ones are still effective.

However, this argument may not hold for much longer. As the modern world is becoming more and more fused with computing, the stakes involved in cybersecurity are getting alarmingly high, which fuels exploration of new attack methods.

When we look at virtually any modern device, vehicle, or piece of equipment that could kill or harm people, we will find a computer in it. Power grids and manufacturing facilities are managed by control systems (e.g., SCADA [50]); airplanes and cars are controlled by distributed systems of embedded controllers (e.g., fly-by-wire [87] and drive-by-wire [238]) and are moving towards full autonomy [242]; implantable devices are semi- or fully automated [118]; voting happens electronically and sometimes remotely [144]; and—

the most frighteningly—weapons are becoming autonomous too (e.g., ATR systems [201]). On top of it, many such systems permit remote access [85, 11], and some are built with commodity components [47, 40].

This environment makes cyberattacks extremely appealing to various malicious actors and brings significant resources into the development of new exploits. The price of an exploit on today's black market is thousands of dollars [24] or—when sold to a government organization for exclusive use—it can allegedly reach a million dollars [174, 122]. Most major militaries today include dedicated cybersecurity units (e.g., NSA [52], GRU [68], North Korean Unit [243]) that harness and develop their offensive capabilities [265, 209]. Their work is further supported by defence contractors and private companies that trade in exploits (e.g., Zerodium [12]). Worst of all, their tools occasionally get leaked [210], and become available not only to state-level actors but even to common criminals.

We can already see the first devastating effects of these developments. Among the examples are a takedown of a regional power grid in Ukraine [280], a lockdown of medical equipment and hospitals in the UK [37], a disruption of a worldwide shipping network [106], and a sabotage of oil production facilities in Saudi Arabia [148]. There are also examples of cyberattacks used for military purposes, including subversion of uranium enrichment plants [279] and espionage [135].

At the same time, the defence side does not lay idle. We see an increasing deployment of defence techniques (e.g., memory safety mechanisms [232], input sanitization [34], fuzzing [219]) which prevents many traditional vulnerabilities. It especially applies to safety-critical systems, which have to comply with regulatory standards (e.g., ISO/IEC 27000 [75], ISO/SAE 21434 [208]) and are overall more concerned with security. Accordingly, the old exploitation tools are becoming less effective.

All these factors together create a modern "arms race" in the field of computer security: As the traditional attack techniques are gradually becoming outdated, the attackers are well motivated (and well equipped) to add new vulnerabilities to their toolbox. And given how effective microarchitectural attacks could be at bypassing existing defences, they may soon catch the attacker's attention. Thus, we have to be prepared.

1.2 The Inevitable Front of Shared Hardware

Another common source of skepticism [230] towards microarchitectural attacks is they assume a local attacker; that is, an attacker who can execute code on the same physical machine as the victim. This threat model is sometimes viewed as less realistic, as it may require a breach of several defence layers (e.g., firewall penetration or malware infection).

There are many situations, however, when this perspective is not valid: Modern browsers frequently execute untrusted code on the users' machines (e.g., website-provided JavaScript or WebAssembly). Application markets for phones and desktops are full of applications from untrusted vendors, some of which contain malware [149]. Virtual servers rented from a cloud provider inherently share the hardware with other tenants. Even private high-security networks are not immune to local attacks as they can be breached through phishing [73], physical penetration [25], and infection from the devices that employees bring in [152].

One may argue that in most of these scenarios the malicious code runs in isolation: the untrusted code in browsers is confined by sandboxes [36]; the third-party applications on smartphones are isolated by the operating system [81]; and virtual servers are defended by Virtual Machine Managers (VMM), with a possible further reinforcement from a Trusted Execution Environment (e.g., Intel SGX [168]).

However, microarchitectural attacks are notorious for bypassing isolation mechanisms, and all of these techniques have been shown vulnerable [185, 131, 127, 117]. Thus, we have to enhance them with specialized defences against microarchitectural attacks.

1.3 The Undefended Flanks of Microarchitecture

Fortunately, we already have the tools for defending against most known microarchitectural attacks, some of which are even deployed in practice (§2.5).

Unfortunately, virtually all these tools target either a single attack variant, or a small set thereof. This strategy is problematic because side channels are not a single vulnerability but a whole class encompassing dozens of vulnerabilities, and with new ones discovered every couple of years. Thus, when we patch one attack vector, the adversary can simply pivot to another one.

To create a holistic protection, we need to approach microarchitectural attacks as a multifaceted problem, which is solved only by a fleet of defences, each targeting its own strictly defined problem. For example, a complete defence against known speculative attacks would include an OS patch against Meltdown [109], a firmware patch against MDS [247, 57, 212], and an application-level mitigation of Spectre [142] and LVI [244].

There is one major obstacle, however, to creating such "combined arms" of defences: We have to ensure that every component correctly plays its part. Specifically, we need to test that every component indeed provides the guarantees that we expect it to provide, and we have to continuously validate that all components function correctly. Otherwise, we might get a mere patchwork of defences, with some attack vectors covered redundantly, while the others are left wide open.

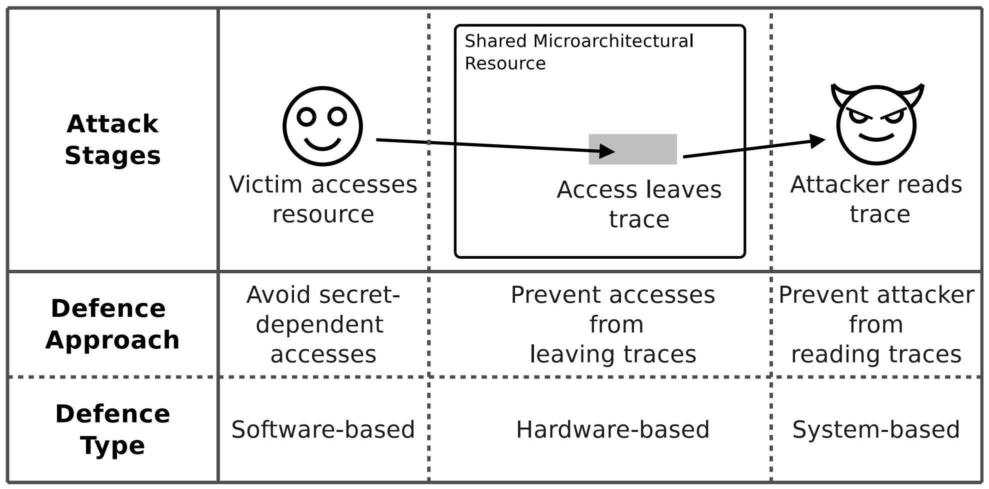


Figure 1.1: Types of microarchitectural defences.

1.4 Contributions

The main goal of this thesis is to make it possible to *automatically check the effectiveness of defences against modern microarchitectural attacks*. The secondary goal is to implement practical tools for it and make them usable with real-world applications.

These goals stem from the fact that the effectiveness of defences is not apparent, and specialized methods are required to test if a defence indeed provides the protection we expect from it. In contrast to more traditional security techniques, manual testing of microarchitectural defences is hardly possible because they operate at the abstraction level below the software and are invisible to traditional debuggers. Moreover, microarchitectural attacks are often unstable and non-deterministic, and an attack failure may be a mere coincidence, not a result of the defence. This obscurity of the microarchitectural security guarantees leads to flaws in defences [141, 39] and their ineffectiveness under certain threat models.

At the same time, the existing testing methods lag behind the developments in microarchitectural attacks. Even though there are numerous approaches for microarchitectural modeling [239], for formal verification of system defences [93], for testing application-level defences [181], and for fuzzing [183], they are unfit for novel adversarial capabilities or for modern attack variants (Chapter 2 discusses it in detail).

In this thesis, **we target three problems**, one for each of the three main types of microarchitectural defences: hardware-, software-, and system-based defences. The division into three types comes naturally from the structure of a microarchitectural attack (Figure 1.1).

It consists of three stages: ① The victim accesses a shared resource in a secret-dependent fashion. ② The access changes the resource's state (i.e., leaves a trace). ③ The attacker observes the state change (i.e., reads the trace). Accordingly, *software-level* defences ensure that the victim does not perform distinct secret-dependent accesses; *hardware-level* defences prevent the CPU operations from leaving observable traces; and *system-level* defences preclude the attacker from reading traces, or make them secret-independent.

Problem 1 (Hardware Level): *How to automatically check if CPU operations can leave unexpected and observable microarchitectural traces?*

To test hardware defences (and the hardware behavior in general), we need to check if the traces left by the applications running on the CPU match our expectations about them. Specifically in this thesis, we target automated testing of black-box CPUs featuring speculative execution, because the existing techniques in this domain either target white-box CPUs or do not cover speculation. Thus, there is no method to independently test the microarchitectural security of commercial CPUs.

In Chapter 3, we propose a method that enables such independent testing. We rely on speculation contracts—formal microarchitectural specifications that allow us to express our expectations about the hardware's behaviour in terms of the traces that we can observe via side channels. Our approach is to compare our expectations of the CPU behavior (encoded in a contract) to how the CPU actually behaves. The comparison happens by executing random instruction sequences on both the silicon CPU and its model, and comparing the information leakage observed via side channels. We implement this method in a fuzzing framework Revizor and demonstrate its effectiveness by testing Intel x86 CPUs.

Problem 2 (Software Level): *How to automatically check if an application has speculative accesses that can expose its secrets?*

At the software level, the goal of testing is to find the parts of the target application that can leave secret-dependent microarchitectural traces. Specifically, we focus on *speculative* vulnerabilities, as there are no effective tools to test against them, in contrast to traditional side channels. The main challenge is that speculation is inherently invisible to software, and regular software testing tools cannot directly detect speculative security violations.

In Chapter 4, we propose a method of speculation exposure, which transforms speculative execution into normal, software-observable execution. The idea is to simulate microarchitectural features within the application, thereby making speculative security violations visible to integrity checkers. We showcase the idea in SpecFuzz, a combination of a compiler extension and a fuzzer that allows us to detect Spectre-type vulnerabilities in software.

Problem 3 (System Level): *How to ensure an execution environment that precludes the attacker from reading traces even if the system software is compromised?*

At the system level, our focus is on the runtime validation of defences. It is necessary because system-level defences could be disabled and reconfigured at any moment, and the applications relying on them will be oblivious of the change. This scenario is a real threat in the computing environments where the adversary may have control over the privileged software (e.g., OS, VMM), such as under a compromised cloud provider.

In Chapter 5, we propose a set of techniques to validate within the application that the expected system-level defences are functioning correctly. The key idea is to modify the application such that it constantly monitors itself and thus detects the changes in the execution environment. We implement the technique in Varys, a tool for protecting Intel SGX enclaves [168] against L1/L2 cache side-channel attacks and page table attacks.

Chapter 2

Background

In this chapter, we will review the basic mechanisms underlying microarchitectural attacks and defences against them (§2.1, 2.3, and 2.4). We will also examine why no single defence can prevent microarchitectural attacks completely, and why it takes a fleet of defences to solve the problem (§2.5). Finally, we will consider the existing options for testing and verifying microarchitectural defences (§2.6).

2.1 Relevant Microarchitecture

Microarchitectural attacks take their root at the immense and obscure complexity of modern CPUs. Programs operate a high-level interface to the CPU—*Instruction Set Architecture (ISA)*—which describes instruction semantics, memory addressing formats, the set of available registers, and so on. However, programs do not see the implementation details of the ISA, such as caching, pipelining, and speculative execution. These details are collectively called *microarchitecture*.

A problem arises when a microarchitectural feature introduces software-visible side effects (e.g., timing differences). Since software security properties are normally expressed in terms of ISA, the tools that check security properties do not cover the microarchitectural side effects. Accordingly, a vulnerability caused by the side effects may stay undetected for a long time. Below are several prominent examples of such microarchitecture features.

2.1.1 CPU Caches

Nowadays, the main memory runs at a much lower clock rate than the CPU, meaning that a program with many memory accesses could spend most of its execution time waiting for its loads and stores. To avoid it, CPUs have *caches*—small but fast buffers that temporarily

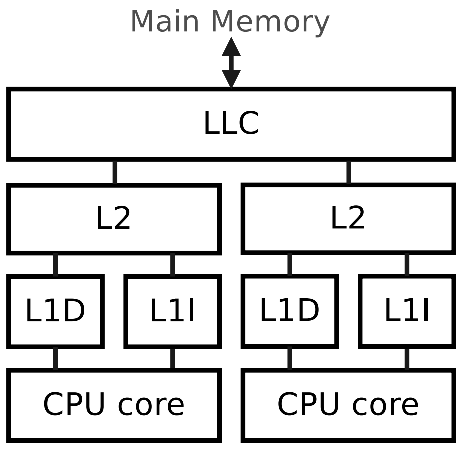


Figure 2.1: Example of a cache hierarchy

keep the values recently read from or written to the memory. When a program runs on a CPU with a cache and repeatedly tries to accesses the same address, the CPU requests the memory only the first time, and the subsequent accesses are served from the fast caches.

Cache hierarchy. CPUs often have several levels of caches, each next larger but slower. A memory access first goes to the lowest level of cache (L1). If the data is there (cache hit), the cache returns it; if not (cache miss), it requests the next level (L2). The process continues until all the levels are checked. Only when the data is missing in all caches, the CPU requests the main memory.

Example: Figure 2.1 shows a cache hierarchy of an Intel Skylake CPU. It has three levels of caches, with the last level cache (LLC) shared among cores, and the lower levels (L1 and L2) in exclusive access to each core. L1 cache is split into an instruction (L1I) and data (L1D) caches to avoid conflicts between instruction fetches and data loads/stores. The first level (L1) has an access latency of 4 cycles, L2 slows down to 12 cycles, and accessing the last level cache (LLC) takes 42 cycles. Each next level is also larger than the previous: L1 keeps 32KB of data, L2—256KB, and LLC—8MB.

Cache addressing. Caches store data in fixed-size blocks called *cache lines* (commonly 64-bytes wide). Depending on a specific cache design, any given memory address can be stored either in any cache line or only in a dedicated subset of cache lines.

In a *fully associative cache*, any cache line can store data from any memory address. Even though this design is simple, it has a large search complexity. To find an address in such a cache, all cache lines may need to be scanned; that is, the search time and/or the hardware complexity is proportional to the cache size. Therefore, fully associative caches are normally used only when the cache is small, with up to several dozen entries.

A more common design is a *set-associative cache*, which groups cache lines into *cache sets*, and has a fixed mapping between memory addresses and the cache sets that serve them. A cache set contains several cache lines, and the number of lines in a set is called *associativity*. Memory addresses form a many-to-one relation with cache sets: A single set serves many addresses, but any given address can be stored only in one set, within any of its cache lines. To find a specific address within a cache, the CPU derives the cache set index from the address, and then exhaustively searches through all cache lines in it. The search complexity in a set-associative cache is proportional to the associativity but not to the overall cache size. Therefore, this design is commonly used to implement large caches.

In a set-associative cache, many addresses are competing for the limited number of cache lines in a set. The competing addresses are called *congruent*. When a cache set is full and the CPU wants to store more data in it, one of the cache lines gets removed from the cache (*evicted*). As such, an access to a non-cached memory address causes eviction of its cached congruent addresses.

Example: The L1D cache in Intel Skylake CPUs is set-associative, with the associativity of eight (i.e., 8 cache lines in every cache set). It maps addresses to sets by taking the address bits 6–12 and using them as a cache set index. Below are several examples of mappings:

Address `0b0000'0000'0100'0000` maps into the cache set #1;
Address `0b0000'0000'0100'1111` maps into the cache set #1;
Address `0b1100'0000'0100'0000` maps into the cache set #1;
Address `0b1100'0000'1000'0000` maps into the cache set #2.

The first three addresses are congruent and compete for cache lines in the first cache set.

TAKEAWAY: On a CPU with caches, one memory access may influence the timing of another access by causing either a cache hit (if they use the same address) or a cache miss (if they use congruent addresses). This may create an unintentional information flow between operations in a program or even between programs.

2.1.2 Virtual Address Translation

One of the core abstractions in modern operating systems is *virtual memory*: The OS presents each process with a view that it has its own contiguous and large memory to which it has exclusive access.

To implement the virtual memory abstraction, the OS maintains a table of mappings between the addresses in the virtual memory (*virtual addresses*) and the addresses in the physical memory that back them (*physical addresses*). This table is called a *page table* and is itself stored in the main memory. Every time an application wants to access a virtual

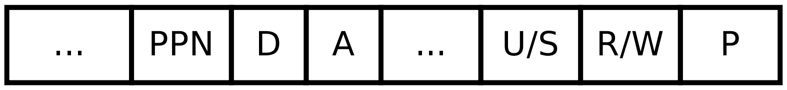


Figure 2.2: Page table entry with bits for permission checks and for swapping.

address, the CPU transparently looks up its physical address in the page table and performs the corresponding access. In modern CPUs, a dedicated Memory Management Unit (MMU) is responsible for this task.

The mappings in page tables are coarse-grained, at the granularity of *pages*, contiguous memory regions of a fixed size. The standard page size is 4KB, but larger sizes—2MB and 1GB—are getting increasingly common.

Translation Lookaside Buffer. Searching for an entry in a large page table is a slow process, and performing it at every memory access would considerably degrade performance. To make the translation faster, its results are cached in a Translation Lookaside Buffer (TLB), which stores the most recently used page table entries. Like regular caches, TLBs could be organized into a multi-level hierarchy and split into instruction TLB (iTLB) and data TLB (dTLB).

TLB introduces a large timing difference between memory accesses: The first access to a page (TLB miss) could take thousands of cycles, while the next ones (TLB hits) would take only a few cycles.

Additional MMU Functions. Besides address translation, MMU also checks permissions before executing a memory access, and serves several support functions for other subsystems. To this end, each page table entry contains fields with metadata about the page.

Example: Figure 2.2 shows a page table entry in an Intel x86-64 architecture. PPN is the physical page number, which implements address translation. Bits U/S (user/supervisor) and R/W (read/write) are used to implement page-level permissions: R/W indicates if writing to the page is permitted and U/S—if userspace code can access it. Bits P (present), A (accessed), and D (dirty) are used to implement swapping: P indicates the presence of the page, A—if it has been accessed after the last swap, and D—that it has been overwritten.

TAKEAWAY: ① Similar to CPU caches, TLBs cause a timing difference between cached and non-cached addresses. ② MMU permission checks enforce isolation between the kernel and the user memory; thus, the isolation relies on the correctness of the checks.

```
LOAD R1, (0x123)
DIV R2, R1
SUB R3, 42
XOR R4, 42
```

(a) Original instruction schedule

```
LOAD R1, (0x123)
SUB R3, 42
XOR R4, 42
DIV R2, R1
```

(b) Reordered instruction schedule

Figure 2.3: Example of instruction reordering.

2.1.3 Pipelining and Pipeline Hazards

A large part of the performance gains in modern CPUs is due to increasing parallelism. *Pipelining* is a common way to improve parallelism at the instruction level. The idea is to split the execution of a single instruction into several stages and execute them in parallel. For example, these stages could be instruction fetch (IF), execution (EX), and write back (WB). Then, while one instruction is storing its result (WB), the next instruction could be computing it (EX), and the next one is fetched from memory (IF).

In certain situations—called hazards—the CPU pipeline stalls because it cannot execute the next instruction immediately. A hazard may happen in three cases: a structural hazard appears when a microarchitectural resource necessary for the current instruction is occupied by another instruction; a data hazard—when there is a data dependency between instructions, and a later instruction has to wait for the result of an earlier one; and a control hazard—when an instruction modifies the control flow (e.g., at a conditional branch) and the CPU does not know what instruction to run next. As the hazards are stalling the CPU, they can significantly reduce its performance.

2.1.4 Techniques to Avoid Hazards

Superscalar Execution. A common way to avoid structural hazards is to replicate some of the stages. For example, the CPU may have several execution units where multiple instructions on the EX stage could perform their computations. This is called *superscalar execution*.

Out-of-order Execution. Even with superscalar execution, a few slow instructions can stall the whole pipeline by occupying all execution units. To prevent it, CPUs with *out-of-order execution* can change the order of instructions in the program as long as the result stays the same.

Example: Consider the following scenario: A CPU has two execution units that can execute any two instructions in parallel. The CPU has registers R1, R2, and R3. Memory accesses take 3 cycles, and arithmetic operations—1 cycle. (We ignore pipeline timings in this example.)

When the CPU executes the program in Figure 2.3a in-order, it takes 5 cycles: In the first cycle, `LOAD` and `DIV` occupy both execution ports, and `DIV` cannot proceed until `LOAD` is completed. After 3 cycles, `LOAD` finishes and releases its execution unit. `DIV` and `SUB` begin execution, taking one cycle. Lastly, `XOR` gets a unit and executes for another cycle.

The CPU can change the schedule and postpone the execution of `DIV` until the result of `LOAD` is ready, as shown in Figure 2.3b. `SUB` and `XOR` are not dependent on the load, and can run in parallel. Accordingly, the execution of the whole program takes 4 cycles.

A problem arises, however, when an instruction in the reordered schedule triggers an exception, and the results must be discarded for all instructions that follow the exception. For example, in Figure 2.3b, if the divider in `DIV` is zero and it triggers an exception, `SUB` and `XOR` are still executed, even though the control flow should have never reached them.

To prevent this issue, the results of all reordered instructions are temporarily kept in an internal buffer (Reorder Buffer, ROB) while the preceding instructions are executed. When no fault happens, they are committed into the architectural state (i.e., stored into registers or memory) in the original program order. Otherwise, the CPU discards the results. Yet the microarchitectural side effects of the discarded instruction (e.g., cache evictions) usually remain untouched.

Speculative Execution of Branches. To deal with control hazards, modern CPUs include a Branch Predictor that tries to predict the outcome of conditional and indirect branches. Based on the prediction, the CPU starts to *speculatively execute* the instructions assumed next.

For example, when the CPU encounters an indirect jump, Branch Predictor predicts the jump target based on the history of recently used targets, and the CPU redirects the control flow to it. While the CPU does not know if the prediction was correct, it keeps track of the speculative instructions in ROB. Eventually, the CPU resolves the hazard and, depending on the outcome, either commits the results or discards them.

Speculative Execution of Loads. Some CPUs also include a predictor for a limited set of data hazards that appear when a load follows a store, and it takes a long time to calculate the store address. In this situation, the CPU has two options: If the addresses match, the CPU should forward the stored value to the load. Otherwise, the load can be executed independently of the store. However, while the store address is unknown, the load has to wait.

To resolve this hazard, some CPUs include a Memory Disambiguation Predictor, which tracks the history of such conflicts, and makes a prediction based on it. Similarly to branch prediction, it triggers speculative execution until the conflict is resolved.

TAKEAWAY: ① With out-of-order execution, an instruction may update the microarchitectural state even if a previous instruction caused a fault. ② With branch prediction, unintended code paths may be executed speculatively. ③ With prediction of memory conflicts, loads may speculatively return outdated values.

2.1.5 Simultaneous Multithreading

Modern high-performance CPUs have many execution ports, more than most real-world programs can utilize. For example, Intel Skylake can execute four integer arithmetic instructions in parallel with two loads and one store. Even with speculative and out-of-order execution, programs rarely have enough independent instructions to make use of these capabilities.

Simultaneous Multithreading (SMT) is a technique to improve hardware utilization by letting multiple independent threads (or even programs) share a core. Each thread has a dedicated set of registers to prevent mixing their data, but most of the other core resources—including execution ports—are used concurrently.

TAKEAWAY: When SMT is enabled, multiple programs (potentially from different security domains) may concurrently use the same microarchitectural resources, such as execution ports and caches.

2.2 Threat Model

To understand how microarchitectural attacks work, we need to first consider in which circumstances such attacks are assumed to happen, and what capabilities the attacker is expected to have.

Main Threat Model. All attacks in this thesis assume the following model: An adversary has control over a program executing on the same CPU as a victim program. Here, we define the term "program" broadly as any code executed on the CPU. Depending on a specific attack, it ranges from a JavaScript snippet executed by a browser to complete control over the operating system.

We expect both programs to be logically isolated, but share some of the microarchitectural state. For example, they may run as two processes with separate virtual address spaces but with a shared CPU cache. Accordingly, the attacker can observe the traces that the victim leaves in the shared microarchitectural elements (e.g., cache evictions).

Finally, we do not set any restrictions on the adversary's knowledge about the source or compiled binary code. Both the source code and the compiled binary could be known.

Privileged Threat Model. When discussing attacks on Trusted Execution Environments (§2.3.3 and Chapter 5), we assume an extended version of the previous threat model. It models a compromised cloud environment where a cloud operator or some of its staff attempts to spy on its customers. Here, the adversary is in complete control over privileged software, in particular the hypervisor and the operating system. She can spawn and stop processes at any point, set their affinity and modify it at runtime, arbitrarily manipulate the interrupt frequency, and cause page faults. The adversary can also read and write to any memory region except the memory that belongs to a Trusted Execution Environment (discussed in Chapter 5), map any virtual page to any physical one, and dynamically re-map pages. Together, it creates lab-like conditions for a microarchitectural attack: it runs in a virtually noise-free environment.

2.3 Microarchitectural Side-Channel Attacks

A microarchitectural side-channel attack is a method of spying on a program. It is based on the idea that most operations on a CPU have side effects. As discussed in §2.1, modern microarchitectures have numerous internal buffers for speeding up slows operations (e.g., caches, TLB) and many modules are shared between processes (e.g., execution ports, lower-level caches). When a CPU executes a program, its instructions change the state of these modules. The changes impact the execution time of other instructions using the same modules. For example, when a memory access brings data into the cache, the later accesses to the same address will be faster. Hence, a malicious program can observe the side effects of the victims' instructions by measuring the execution time of its own instructions.

Microarchitectural attacks can be a powerful tool in an attacker's toolbox since they provide means to bypass software isolation. Previous research has shown that microarchitectural attacks are possible across all major isolation levels, including virtual machines [131, 262], OS kernel [121, 262], secure enclaves [117, 56, 244, 57, 212], and sandboxes [185, 142].

In this and the next section, we review how microarchitectural attacks work and consider a few of the most common variants. We also briefly review the spectrum of existing attacks to illustrate the variety of options available to an adversary and the difficulty of preventing them.

Note on Terminology: In the remainder of the thesis, we refer to traditional (i.e., non-speculative) microarchitectural side-channel attacks as *side-channel attacks*, and to speculative microarchitectural side-channel attacks as *speculative attacks*. The term *microarchitectural attack* encompasses both traditional and speculative attacks.

2.3.1 Attack Overview

We can model a side-channel attack as two programs—*VictimProg* and *AttackerProg*—interacting with a microarchitectural module. The module consists of one or more elements. A program can interact with one element at a time, and the interaction changes the element's state. The duration of such an interaction depends on the previous state of the element.

Example: CPU cache (§2.1.1) is an example of such a microarchitectural module. The elements of a cache are cache sets. When a program accesses memory, it uses one of the cache sets. The access may evict one cache line from the set and bring another one into it. The access latency depends on whether the address is already cached (cache hit), or it needs to be fetched from memory (cache miss).

A side-channel attack consists of five stages [74]:

1. Pre-Attack: At the preparation stage, the attacker analyses *VictimProg* to find vulnerable instruction sequences; that is, instructions that interact with the target module such that the interaction depends on the victim's data.
2. Setup: When the attack begins, *AttackerProg* sets the module to a known state (e.g., flushes the cache). This happens either before *VictimProg* starts, or during its execution.
3. Tracing [optional]: *AttackerProg* stops and waits for *VictimProg* to interact with the module.
4. Measurement: *AttackerProg* interacts with the module's elements and measures the duration of each interaction. The differences in timing reveal to the attacker the elements that were used by *VictimProg*.
5. Analysis: The attacker analyses the measurements to derive the victim's data.

Pre-Attack and Analysis happen off-line, and are necessary only once. Setup, Tracing, and Measurement constitute the on-line phase, and are usually repeated many times during the attack. Setup and Measurement happen either concurrently with the victim's execution, or in a time-sliced fashion.

The implementation of each step varies between specific attacks. Several prominent examples are summarized in Table 2.1, and we discuss them next.

Target	**Cache**					**Page Table**	
Attack Variant	**Prime+ Probe**	**Flush+ Flush**	**Flush+ Reload**	**Evict+ Reload**	**Evict+ Time**	**Page Fault**	**Page Bit**
Pre-attack	Find a secret-dependent memory access in the victim program						
Setup	Fill the target cache set by accessing congruent addresses	Flush the target address		Evict the target address by accessing congruent addresses		Mark the target page as invalid (e.g., reset P-bit)	Reset A/D-bits for the target page
Measurement	Access the congruent addresses again and measure timing	Flush the target address and measure timing	Access the target address and measure timing		Measure the execution time of the victim's code	Catch the page fault within the OS	Read the A/D-bits of the target page
Analysis	Slow timing indicates victim's access	Fast timing indicates victim's access			Slow execution indicates victim's access	The page fault metadata contains the accessed address	If A or D bit is set, the victim accessed the page

Table 2.1: Overview of cache and page table side-channel attacks.

```
1 int8_t array[128];
2 if (secret == 0) {
3     int8_t x = array[0];
4 } else {
5     int8_t x = array[64];
6 }
```

Figure 2.4: Code vulnerable to cache timing attacks

2.3.2 Cache Side-Channel Attacks

One of the most well-researched targets for side-channel attacks is the CPU cache. In a cache timing attack [191, 160, 131, 127, 17, 276], the attacker tries to find out which cache sets the victim program used, and thus infer which memory addresses it accessed. From the addresses, the attacker derives the memory contents and/or the control flow of the program.

Example: Consider the code snippet in Figure 2.4. At the Pre-Attack stage, the attacker reviews it and sees that it includes two memory accesses at lines 3 and 5. The accesses are 64B apart, so they use different cache sets. Moreover, which of the two accesses is executed depends on the `secret` value (line 3 if `secret` is zero, line 5 otherwise). Hence, if the attacker could learn the cache set used by the victim, she will also learn the `secret` value. A cache attack helps with this.

At the Setup stage, the attacker program sets both cache sets to the same state (e.g., fills them with its data). At the Tracing stage, it lets the victim execute the whole snippet. Suppose the `secret` value is zero: The victim program executes line 3 and changes the state of the first cache set. At the Measurement stage, the attacker program accesses both cache sets and measures the access timing. The timing difference reveals that the first set was accessed.

Finally, at the Analysis stage, the attacker matches the results with the pre-attack analysis: State change in the first cache set implies that the victim executed memory access at line 3, thus the `secret` value was zero.

There are several attack variants, mainly diverging in their Setup and Measurement.

Prime+Probe [191, 188] attacks use cache eviction as a signal of a state change. The stages of a Prime+Probe attack are:

- Setup (Prime): The attacker fills target cache sets with its own data by accessing several congruent addresses.
- Tracing: A victim's access to a target cache set evicts one of the attacker's cache lines.
- Measurement (Probe): The attacker accesses the same congruent addresses, and measures the timing of each access. Accesses to those sets used by the victim experience a cache miss, thus they take longer time.

Prime+Probe attacks can target any level of cache: Although the original variants [191, 188, 258] targeted only L1 data cache, later research has shown that instructions cache is vulnerable too [18], as well as higher levels of the caches hierarchy [160, 131, 157, 127, 105].

Compared to other attack variants, Prime+Probe is less demanding to the attack conditions, but is more affected by the measurement noise: On one hand, Prime+Probe does not require shared memory (in contrast to Flush+Reload, see next) and could be launched against any program that shares a cache with the adversary. On the other hand, it targets a cache set and not a specific memory address. Hence, if there are several programs besides the victim that use the cache, the adversary will not be able to distinguish their accesses from the victim's ones.

Flush+Reload [276, 111, 285] uses cache fill as a signal (i.e., an opposite signal to Prime+Probe). Its stages are:

- Setup (Flush): The attacker flushes the target addresses (e.g., `array[0]` and `array[64]` in Figure 2.4).
- Tracing: The victim accesses one of these addresses and brings it into the cache.
- Measurement (Reload): The attacker accesses the addresses again and measures the timing. If a cache line was filled, it will take less time to access it.

Flush+Reload requires shared memory. To check if a cache line was filled, the attacker must be able to access the same address as the victim did. It is possible only if the victim is sharing memory with the attacker. This could happen, for example, when using a shared library or when memory de-duplication is enabled.

Evict+Reload [111, 157] is a variant of Flush+Reload for the architectures that do not provide a flush instructions or restrict it to privileged code (e.g., ARM). At the Setup stage, the attacker forces eviction of the target address by accessing several other, congruent addresses. The Measurement stage is identical to Flush+Reload.

Flush+Flush [110] is a more stealthy version of Flush+Reload. As the original attack produces a lot of memory traffic, the victim could observe this traffic and thus detect an attack attempt. Flush+Flush avoids it by relying on the timing of the flush instruction itself, which does not cause a memory access.

Other variants. Evict+Time [241, 42] was the earliest variant of a cache side-channel attack. It uses execution time of the victim's program as a signal. At the Setup stage (Evict), such an attack evicts several target addresses (or simply as many addresses as possible). At the Measurement stage, it sends some request to the victim program and measures the response time. The time will depend on the number of memory accesses the victim program executes, as most of the accesses will experience a cache miss after the eviction. Accordingly, if the number of accesses depends on a victim's secret, the execution time will expose the secret to the attacker. The problem is, however, that the execution time also depends on many other parameters, such as scheduling, number of system processes, and so on. It makes the measurements noisy and the attack—hard to launch in practice.

Reload+Refresh [54, 272] is a recent attack that exploits the cache replacement policy. Most caches use a variation of the Least Recently Used (LRU) algorithm to select which of the cache lines within a cache set will be evicted next. As such, when a victim program uses a cache line, the line becomes the most recently used, and all other lines become closer to eviction. Reload+Refresh uses this observation to detect victim's memory accesses without causing a cache eviction. It makes the attack more stealthy towards the defences that use cache evictions as an indicator of an ongoing attack.

2.3.3 Page Table Side-Channel Attacks

Address translation attacks reveal page-level access patterns of a program by exploiting Memory Management Unit (MMU, see §2.1.2). They are usually considered in the context of Trusted Execution Environments (e.g., Intel SGX) as they are possible only if privileged software is compromised, hence they are also called *controlled-channel attacks* [273]. Page table attacks can be classified into page-fault based and page-bit based.

Page-fault based attacks [273, 224] intercept page-level memory accesses by intentionally triggering page faults:

- Setup: The attacker sets up the execution environment such the first access to the target page will cause a page fault. For example, the attacker may swap the page or mark the page as invalid in the page table. In case of Intel SGX, the attacker may also evict the page from Enclave Page Cache (discussed in Chapter 5).
- Tracing: The victim accesses the target page, a page fault happens, and the CPU calls the OS to handle the fault.
- Measurement: The attacker-controlled fault handler reads the page fault descriptor, which reveals the accessed address.

Page-bit based attacks [254, 245] use the *Accessed* and *Dirty* page table bits as a page access indicator.

- Setup: The attacker clears Accessed and/or Dirty bits for the target page.
- Tracing: When the victim accesses the page, the CPU automatically sets the bits.
- Measurement: The attacker reads the bits. If they are set, the victim used the page.

2.3.4 Side-Channel Attack Landscape

Side-channel attacks could target many more microarchitectural modules besides caches and MMU. In fact, virtually any module could become a target if two programs from different security domains can share it. Below is a brief overview of such targets[1]. It illustrates how diverse side-channel attacks are and why it is so challenging to prevent them.

On-Core Targets. *Branch Target Buffer* (BTB) is a buffer used to implement speculative execution of branches (§2.1.4). BTB keeps a history of the recent branch targets. Aciiçmez et al. [17, 14, 15] showed that an attacker can intentionally fill BTB with wrong values, and thus cause frequent branch mispredictions. Each misprediction has a penalty of several clock cycles, which increases the overall execution time of the victim program. This divergence in timing reveals to the attacker the control flow paths executed by the victim.

The actual values stored in BTB (i.e., branch targets) can also be recovered with a side-channel attack [82, 83]. To this end, the attacker has to cause collisions between the branches in its program and in the victim's program. Then, the mispredictions in the attacker's program will reveal which of the branches the victim executed. Sangho et al. [155] further developed this approach, and showed that the attack becomes particularly precise when the attacker can completely mirror the victim's code layout. This, however, assumes a privileged attacker (§2.2).

Execution port contention can create a signal when two threads use the same core concurrently (i.e., when SMT is enabled, §2.1.5). If one thread executes an instruction that occupies an execution port for a period of time, the other thread's instructions of the same type have to wait until the port is released. Hence, the execution time of the attacker's instructions depends on the instructions that the victim executes. This signal was shown sufficient to leak victim's data [258, 16]. Recently, this side channel was also demonstrated in combination with speculative execution [44, 92].

Floating-point operations are another target. Andrysco et al. [27] showed that the execution time of floating-point operations highly differs depending on its operands, and on the previous state. As such, they can cause secret-dependent variance in the programs' execution time, which the attacker could use as a signal. Kohlbrenner et al. [143] further detailed these findings.

Translation Lookaside Buffer (TLB) is effectively a cache, and thus it is vulnerable to cache attacks. Gras et al. [104] demonstrated the possibility of such an attack. The attack assumed a privileged adversary (§2.2) because a direct manipulation of the TLB state is only possible by privileged software (at least, on x86).

[1]For an exhaustive overview, refer to the excellent survey by Ge et al. [94]

Single-instruction Multiple-data (SIMD) operations often have high energy consumption. To save power, the OS turns off the SIMD execution unit when it is not in use. Hence, when a program uses a SIMD operation for the first time, the unit has to be turned on, which takes a significant amount of time. Schwarz et al. [215] demonstrated that it constitutes a side channel.

Off-Core Targets. *Memory Bus* is a bus that transfers data between a CPU and the main memory. All processes running on the CPU use the memory bus concurrently. As the bus has only a limited throughput, high memory traffic from one process increases the memory transfer delays for the other processes. Wu et al. [269] demonstrated these delays constitute a side channel.

Row buffer is a small buffer that stores the most recently accessed row in a memory bank (a bank is the smallest structural unit of DRAM memory). It is effectively a cache with a single entry, and it creates a timing difference between a row buffer hit and miss. Pessl et al. [192] showed that this difference is observable at the software level, and thus it could be used to launch a side-channel attack. Later, Schwarz et al. [213] demonstrated that the signal is strong enough to launch an attack from a JavaScript application running in a browser.

Intel Data Direct I/O (DDIO) is a feature of recent Intel CPUs that improves performance of network packet processing. It allows to bypass main memory and transfer network packets directly into the LLC. Taram et al. [234] presented an attack that finds the location of network packets in the LLC, monitors them via Prime+Probe, and can detect the frequency and the size of the received packets.

Accelerators can be a target for side-channel attacks too. Naghibijouybari et al. [178] demonstrated several attacks on GPUs which used various side-channel signals, including the execution time of a program on GPU, its allocated memory, and the readings of GPU performance counters.

2.4 Speculative Attacks

Speculative microarchitectural side-channel attacks (in short, *speculative attack*) further enhance the traditional side-channel attacks by exploiting speculative and out-of-order execution (§2.1.4) to manipulate the victim's data or control flow. In a speculative attack, the adversary intentionally forces the CPU into making a wrong prediction and speculatively executing the victim program incorrectly (e.g., taking a wrong path or using wrong data). Because the incorrect speculative execution violates the application semantics, it may bypass

```
1 int i = input[0];   // attacker-controlled
2 if (i < size) {     // speculatively mispredicted
3     int secret = array1[i];
4     int x = array2[secret];
5 }
```

Figure 2.5: Code snippet vulnerable to Spectre V1.

security checks within the application. Moreover, should any exceptions appear during the speculative execution, they will be handled only during the last pipeline stage (retirement).

For a long time, this behavior was considered safe because the CPU never commits the results of a wrong speculation. However, as the authors of Spectre [142] and Meltdown [158] discovered, some traces of speculative execution are visible on the microarchitectural level. For example, the data loaded on the mispredicted path will not show up in the CPU registers, but will modify the state of the CPU caches. The attacker can later launch a side-channel attack to retrieve the traces and, based on them, deduce the speculatively computed data.

Speculative attacks are commonly divided (e.g., by Canella et al. [58]) into prediction-based (Spectre-type) and fault-based attacks (Meltdown-type).

2.4.1 Spectre-type

Spectre-type attacks exploit prediction of pipeline hazards (§2.1.4) to force the CPU into incorrect (speculative) execution of a program.

Spectre V1 [142]. In Spectre V1, the attacker exploits conditional branch prediction to bypass bounds checks, which are a traditional mechanism to prevent invalid memory accesses. The idea of bounds checking is simple: Before loading/storing a value into memory, the program checks that the address is valid and points to the intended object in memory. Bounds checks are normally implemented with conditional branches. On a CPU with branch prediction, the attacker can force a misprediction, the CPU will speculatively take a wrong branch, and accesses an invalid memory address.

Example: Consider the code snippet in Figure 2.5 and assume the attacker can control the `input` value. The memory safety of this code hinges on the bounds check at line 2, which prevents the adversary from reading arbitrary memory values with the load at line 3. However, if the CPU implements branch prediction, it can execute the check in parallel with the vulnerable load if it predicts the check will not fail. Later, the CPU will find out the prediction was wrong and discard the speculated load, but its cache traces will stay. The adversary can access the traces by launching a side-channel attack.

```
1 int64_t invalid_address = input[0]; // attacker-controlled
2 int *pointer = (int *) invalid_address;
3 ... // later
4 pointer = (int *) valid_address;
5 int spec_secret = *pointer;  // speculatively loads invalid_address
6 int x = array[spec_secret];
```

Figure 2.6: Code snippet vulnerable to Spectre V4.

The attack works as follows: The adversary sends several in-bounds inputs that train the branch predictor to anticipate that the check at line 2 will pass. Then, she sends an out-of-bounds input, the branch predictor makes a wrong prediction, and the program speculatively executes lines 3–4 even though the program's semantics forbid so. It causes a speculative buffer overread at line 3 and the read value is used as an index at line 4. The second access leaves a secret-dependent cache trace.

The adversary can access the trace by launching a side-channel attack and use it to deduce the `secret` value: The address read at line 4 depends on the `secret` and, correspondingly, finding out which cache line was used for this memory access allows the attacker to also find out the `secret` value loaded on the speculative path.

Spectre V4 [101]. This attack exploits the memory disambiguation predictor (§2.1.4). When it produces a wrong prediction, a load may speculatively fetch a stale value from memory, even though the program semantics dictates that it was supposed to be updated by a recent store. This is effectively a lost memory update, although a speculative one. An attacker could use this misprediction to leak the values of memory addresses that the victim's program never accesses architecturally.

Example: Consider Figure 2.6. The attacker has control over the input (line 1), and thus controls a pointer in a program (line 2). Without the speculation, the program never dereferences the pointer as it overwrites it with another value (line 4). With the speculative store bypass, however, the load at line 5 may miss the update at line 4, and fetch a stale pointer value. This will lead to a memory access with an attacker-controlled address (line 5), which later leaves a data-dependent cache trace (line 6).

Other variants. Other types of prediction have been shown vulnerable too. Indirect branch misprediction can lead to the execution of unintended code (Spectre V4 [142]). Return address misprediction leads to a speculative return to an unintended address (Spectre V5 [147]).

```
1 int secret = *kernel_memory_address;  // triggers a page fault
2 int x = array[secret];
```

Figure 2.7: Code vulnerable to Meltdown.

2.4.2 Meltdown-type

Meltdown-type attacks exploit speculative execution of faulting instructions. In some CPUs, when a memory load causes a fault (e.g., a page fault), the CPU may still speculatively execute it. This load will not fetch a value from memory, but rather return a previously-loaded value stored in one of the CPU's microarchitectural buffers (e.g., in a CPU cache). The value might have been brought into the buffer by another thread or process, hence Meltdown-type attacks can leak data across security domains.

This type of speculation was so far observed only on Intel CPUs, but there are signs that other vendors (e.g., AMD) may have it too [58].

Meltdown [142, 158]. The original Meltdown attack (called Meltdown-US in the classification of Canella et al. [58]) exploited speculation on the page faults that happens when a user-space application attempts to read from an address in kernel memory (i.e., when the U/S page table entry bit is set). If the kernel has recently accessed this address and the value is stored in a CPU cache, the CPU vulnerable to Meltdown will speculatively return the value. Similar to Spectre attacks, the attacker can fetch this value by using it as an offset in another memory access, and thus leaving a side-channel observable trace.

This way, a user-space application can leak values from kernel memory. It constitutes a breach of one of the most fundamental isolation primitives—the isolation between privileged and non-privileged software.

Example: Consider Figure 2.7. This code is a part of the attacker's program. It takes an address from kernel space and tries to read it (line 1). This causes a page fault, but while the fault is handled, the CPU speculatively returns the `secret` value, if the value is cached. Then, the value is used as an index in an array access (line 2). Its accessed address depends on the `secret`, which means it leaves a `secret`-dependent cache trace. After the fault is handled, the attacker can read the trace and thus get the `secret` value.

Following Meltdown, many more fault-based attacks have been discovered in Intel CPUs. Foreshadow [56, 262] exploited page faults from accessing non-present pages (i.e., when the "Present" page table bit is set to zero). It can leak data from Intel SGX enclaves, from the OS kernel, and across virtual machines. Spectre V3a [29]—the attack was initially mis-classified as a Spectre variant—speculatively bypassed the faults that happen when a user-space application tries to access a privileged system register. LazyFP [228] exploited faults that happen when a non-available device is accessed. Even more vulnerabilities may still stay undiscovered; investigation of Meltdown-type attacks is an ongoing research topic.

Microarchitectural Data Sampling. MDS [247, 57, 212] is a sub-family of Meltdown-type attacks which relies on microcode assists. A microcode assist is a type of fault that is handled by firmware instead of software. It is commonly used to optimize swapping algorithms by automatically setting the relevant page table bits. For example, an assist may happen when a program loads from a page that was recently swapped in. This access triggers a fault, and the firmware automatically sets the “Accessed” bit in the corresponding page table entry (see §2.1.2).

Some Intel CPUs speculate on microcode assists. Similarly to the original Meltdown, such a CPU will speculatively execute the load that triggered an assist, but will return a value from a microarchitectural buffer instead of memory. In case of MDS attacks, however, the value may come from one of several buffers, not only CPU caches. Depending on the microarchitectural state and on the type of assist, the value may be fetched from Store Buffer, Line-Fill Buffer, or Load Port. (These are buffers that temporary keep store and load values).

2.5 Microarchitectural Defences

We next review the existing defences against microarchitectural attacks. The section showcases why we need the tools described in this thesis.

Microarchitectural defences have been proposed at all three main abstraction layers: Hardware defences modify microarchitectural modules such that sharing them becomes safe. When hardware modifications are impossible or impractical, system defences try to create a side-channel free execution environment. And when neither hardware nor system defences succeed, application-layer defences change the application to make its microarchitectural traces secret-independent or hide them in noise.

2.5.1 Hardware: Securing Microarchitecture

The most fundamental—albeit the most costly—solution to microarchitectural vulnerabilities is to improve the microarchitecture itself. To prevent traditional side channels, shared microarchitectural modules have to be modified such that the state changes made by one program become invisible to other programs. For speculative attacks, the microarchitectural side-effects of speculative execution need to be confined until the CPU knows the speculation was correct.

Secure Microarchitectural Sharing. In practice, a common defence strategy is to simply disable sharing. For example, Microsoft Azure security guidelines [166] recommend to disable SMT when running security-critical applications. This prevents concurrent attacks on lower-level caches as well as other on-core resources. This recommendation was also a part of the Intel's response [128] to Spectre/Meltdown attacks.

When it is impossible to disable sharing, a microarchitectural module could be partitioned, and each program will have an exclusive access to its partition. Page [190] proposed a cache design that supports such partitioning. Later, Intel implemented a Cache Allocation Technology (CAT). Liu et al. [161] and Dong et al. [77] used CAT to dedicate a portion of LLC to a single process, thus preventing cross-core cache attacks. Kiriansky et al. [139] proposed a design that improves performance of such partitioning by allocating cache ways.

Alternatively, caches could be protected by modifying the eviction algorithm. Wang et al. [259] developed a cache indexing scheme that randomly selects the cache set for eviction, thus removing hard mapping between memory addresses and cache sets. Yan et al. [275] proposed a cache replacement scheme that completely forbids cross-core evictions. Saileshwar et al. [205] developed a cache design that retains those cache flush patterns that are typical for Flush+Reload attacks. Fang et al. [86] suggested a dedicated hardware module that detects suspicious cache activity and preloads the affected cache lines.

Secure Speculation. The discovery of Spectre and Meltdown led to a plethora of proposals for securing speculative execution.

The most conservative solutions proposed to disable prediction either entirely [226] (although not all processors support it) or on a targeted basis, with serializing instructions (e.g., `LFENCE` on Intel CPUs [129] or `DSBSY` on ARM). Experiments show [193, 194] that this solution is impractical as it leads to a dramatic performance drop, up to an order of magnitude.

The next iteration of this approach was to disable speculation of those instructions that are considered unsafe. Although the precise definition of "unsafe" varied from paper to paper, most of them agreed that speculatively-computed (or at least speculatively-loaded) values should not be used for further speculative computations. One of the first proposals in this area was Context-Sensitive Fencing [233], which tracks the information flow of a program at the CPU front-end and inserts serialization barriers upon detecting an unsafe memory access. STT [277], NDA [263], and SpecShield [35] further extended this idea to implicit information flow and to non-cache side channels. ConTExT [214] applied a similar strategy but selectively to the user-defined critical memory regions.

An alternative approach is to permit all speculation but keep the speculative data in a dedicated buffer until the speculation is resolved. InvisiSpec [274] and SafeSpec [137] showed implementations of this idea with a focus on cache-based attacks. CleanupSpec [204] used an inverse strategy: It allowed speculative microarchitectural changes, kept the old values in a temporary buffer, and reverted the changes upon a detected misprediction.

TAKEAWAY: ① The proposals for secure microarchitecture target only one or a few variants of microarchitectural attacks but do not provide a comprehensive defence. ② Many techniques for secure speculation make a tradeoff between security and performance, by allowing those speculative actions that are considered safe. As the definition of "safe" is informal, the exact security guarantees of these proposals are also uncertain. This motivates Chapter 3, which describes a tool for automatic detection of such vulnerabilities.

2.5.2 System: Creating a Protected Environment

System defences act as overseers of microarchitectural resources, thus striving to create a side-channel protected environment for the programs executing on the system. They ensure programs from different security domains never share resources, or when they do, that sharing does not provide any useful information to the attacker.

Partitioning. Many system defences prevent the attacker from accessing a microarchitectural module concurrently with the victim. The most prominent example of this strategy is SeL4 [176], a microkernel OS that implements extensive microarchitectural isolation [93, 216] by scheduling most of the CPU resources to one security domain at a time. A limited version of this approach was also recently introduced in the Linux kernel as Core Scheduling [88], a technique that mitigates on-core side channels by forbidding threads from different processes (or groups of processes) to share a core.

Several system defences focused on partitioning CPU caches. To this end, they applied cache coloring [136], a technique to allocate memory to processes such that one process never has access to congruent addresses of another process. Notable examples of this approach are the work of Shi et al. [222], who implemented dynamic cache coloring in a hypervisor, and of Kim et al. [138], who used static cache coloring to create special side-channel protected pages for storing sensitive information.

Off-core resources could be partitioned too: Wang et al. [257] and Wassel et al. [260] showed techniques to partition shared networking hardware. Wang et al. [256] suggested a scheme for allocating memory bus bandwidth.

Partitioning is also the primary defence against Meltdown: Kernel Page Table Isolation [109] isolates kernel and user memory by splitting them into separate virtual address spaces.

Flushing. Partitioning on its own is not sufficient: Although the attacker program cannot access the partitioned module concurrently with the victim, it can still access the traces left on the module after the victim finishes using it. Therefore, besides partitioning the resource, it must also be flushed after a process finishes using it. Zhang et al. [287] suggested to periodically interrupt applications and flush the caches they use; Godfrey et al. [96] proposed to flush caches at VM context switches; Cock [67] used lattice scheduling [72] as an optimization of flushing; SeL4 implementation of timing protection [93] flushes multiple microarchitectural modules when switching context.

Noise Injection. Instead of restricting microarchitectural sharing, the system could inject noise into the microarchitectural state such that victim's traces become indistinguishable from noise. Zhang et al. [283] introduced this technique in a form of a bystander VM that continuously uses CPU caches. Venkatanathan et al. [248] achieved the same result by ensuring that processes run uninterrupted for a prolonged period of time, thus making too many state changes to be useful for an attack.

Alternatively, the noise could be injected not into the microarchitectural state but in the attacker's measurements. Vattikonda et al. [249] modified a hypervisor to insert noise in the `rdtsc` instruction readings. Most major browsers have an intentionally reduced resolution of the timer readings returned by the 'performance.now()' command [170, 202].

Monitoring. Finally, the system may permit unrestricted sharing but monitor all applications in search for the behavior typical for a side-channel attack. This is the least intrusive method of defence but also the least reliable. Chiappetta et al. [64] implemented a monitoring process that continuously reads cache-related performance counters to detect an unusual rate of cache evictions. Zhang et al. [284] implemented a similar approach in a hypervisor that monitors its VMs. Raj et al. [197] extended this idea to monitoring the memory bus traffic.

TAKEAWAY: System-level defences inherently rely on privileged software, which could be compromised in a could environment or via a privilege escalation vulnerability. In this case, the security guarantees of system defences get compromised too. This motivates Chapter 5, which describes a technique for validating the effectiveness of system defences.

2.5.3 Application: Avoiding Information Leakage

The final line of defence is the application itself. To make it protected against microarchitectural attacks, the application's actions should be secret-independent or should not leave microarchitectural traces.

Constant-time Programming is a classical defence technique where the application is changed to make all the program's memory accesses and control-flow paths independent of the processed data. Traditionally, it is implemented manually, with carefully-crafted libraries. For example, NaCl [41] and BearSSL [3] are libraries that provide constant-time implementations of cryptographic functions. libfixedtimefixpoint [27] and Escort [199] are libraries of constant-time floating-point operations. Such a manual approach is labor-intensive and error-prone, thus it is practical only for narrow and well-defined applications, such as cryptography.

Several techniques have been proposed to automate rewriting. Wu et al. [268] developed a compiler extension to transform programs into their constant-time versions. Van Cleemput et al. [246] implemented a similar idea but as a JIT compiler, thus benefiting from dynamic profiling. Both techniques are effective for small functions but introduce prohibitive overheads when applied to a complete application.

Preloading. When the range of secret values is narrow, the victim application can access all of them, independently of the secret. Then, the attacker will not be able to define which of the values the victim actually used. Osvik et al. [188] and Brickell et al. [53] applied this idea to protect AES lookup tables. Shinde et al. [224] proposed to use preloading of all application pages to protect Intel SGX enclaves against page-table attacks. Preloading is impractical, however, for full-application defence as the range of potentially secret values is prohibitively large.

Randomization. The application could randomize the location of its secrets, thus making it harder for the attacker to map traces into the secret values. A common technique for randomization is Address Space Layout Randomization (ASLR) [89], which randomizes the location of objects in memory. Randomization has been also proposed for the program's control flow [123], and applied to Trusted Execution Environments [217]. These approaches provide static randomization (i.e., do not re-randomize during the program execution), which provides only best-effort guarantees as the attacker can still find out the location secrets with repeated measurement [82, 109].

Oblivious RAM (ORAM) [98, 99, 229, 159] is a continuous randomization technique that provides strong security guarantees, but at a high performance cost. The idea is to makes the program's memory accesses independent of the input by re-shuffling data after every memory access. The early implementations of ORAM [98, 99] introduced overheads of 200$\times$ and higher; more recent versions [198] reduce it to 20$\times$, which is still too high for most applications.

Replace Vulnerable Patterns. In the context of speculative attacks, the instructions that trigger speculation could be replaced with the instructions that do not. For example, Retpolines [156] and their derivatives [26] replace indirect branches (vulnerable to Spectre

V2) with a return-based instruction sequence. The issue, however, is that the replacement sequence could be vulnerable too: The return instructions used in Retpolines are vulnerable Spectre V5 [164] and to LVI [244].

Delayed Speculation. An approach similar to InvisiSpec and SafeSpec could be implemented in software: Delay speculative memory accesses until the hazard is resolved. SLH [59] achieves this effect by adding a data dependency between conditional branches and the later memory accesses. It protects against Spectre V1, but causes a considerable slowdown (~50%).

TAKEAWAY: Reliable application-level defences are either labor-intensive or cause a considerable slowdown. As such, they are practical only when applied in a targeted fashion to the potentially-vulnerable parts of the application. In Chapter 4, we will consider a method to find these vulnerable parts.

2.6 Related Work: Testing Microarchitectural Defences

Finally, we consider the existing options for testing microarchitectural defences. This section illustrates that the available tools are not completely fit for the context of modern microarchitectural attacks, thus motivating the new methods described in the next chapters.

2.6.1 Hardware Layer

Formal microarchitectural models. While several works [30, 71, 97] present formal models for (parts of) the ARMv8-A, RISC-V, MIPS, and x86 ISAs, the work on formal models of *micro*architectural aspects is only now emerging. For instance, Coppelia [282] is a tool to automatically generate software exploits for hardware designs. This tool is not applicable to black-box testing and does not cover speculative execution.

Detection of microarchitectural vulnerabilities. Several recent papers investigated detection of speculative vulnerabilities in CPUs. Medusa [175] is a method for detecting variants of MDS with fuzzing. Nemati et al. [182] showed a method for detection of Spectre V1 in ARM CPUs. SpeechMiner [271] is a tool for detailed investigation of speculative vulnerabilities. All these works targeted specific attacks and did not propose a generic detection method.

Nemati et al. [181] proposed a method to fuzz a CPU against an abstract cache model. Our approach proposed in Chapter 3 is similar to this method, but their model did not include speculation, and they fuzzed a simpler in-order CPU (Cortex-A53).

ABSynthe [103] automatically detects contention-based side channels, but cannot detect if a speculation feature introduces leakage through a side channel.

White-box testing. Some works focused on testing CPUs at design time. Zhang et al. [281] proposed to annotate Verilog HDL specifications with information-flow properties, and thus enable verification of the properties at design time. Fadiheh et al. [84] propose a SAT-based bounded model checking methodology to check whether a given register-transfer level (RTL) processor design exhibits covert channel vulnerabilities. In the terminology of speculative contracts (see §3.2.2) they check whether the processor design satisfies the *ARCH-SEQ* contract. CheckMate [239] automatically searches for predefined attack patterns in a given ISA model. These tools are not applicable to testing of commercial, black-box CPUs.

Architectural Fuzzing. Several tools apply fuzzing for detecting architectural bugs in the CPU implementations; that is, detecting when a CPU's behavior does not comply with its ISA. RFuzz [150] proposed a tools for fuzzing on RTL level as well as a method for measuring its coverage. TestRIG [267] is a tool for mutational random testing of RISC-V implementations. These tools cannot detect microarchitectural issues.

TAKEAWAY: Several tools exist for testing hardware defences against traditional side-channel attacks, but few of them can detect speculative vulnerabilities; those that can, are not applicable to black-box CPUs.

2.6.2 System Layer

Most of the software-layer testing tools are equally applicable to both system- and application-layer defences, and we will review them together in §2.6.3. However, a few works target exclusively system-layer defences, which we consider next.

Formal Verification. SeL4 [93] microkernel OS is in the process of developing a formally verified timing protection [216]. Albeit providing strong information-flow guarantees, this method is laborious and it is remains doubtful whether it can be extended to a large monolithic kernel, such as Linux. It also assumes the OS is trusted.

Monitoring Page Faults. Several works proposed a page fault monitoring mechanisms to detect attempts to launch a page-table attack (§2.3.3). T-SGX [223] used Transactional Synchronization Extensions (TSX) to detect and hide page faults from the OS. Cloak [108] and its concurrent work [63] strive to extend T-SGX guarantees to cache attacks by preloading sensitive data, but requires source code modifications and protects only selected parts of the application; a complete solution would have an unacceptably high overhead. Similarly, Strackx et al. [231] detect page faults via TSX and avoid cache side channels by preloading of pages at AEX. SGX-LAPD [91] reduces the precision of page table attacks on code pages by requesting large pages and validating their size within the application. These approaches became a basis for our method of self-monitoring described in Chapter 5.

Concurrent Work. Concurrent to the work described in Chapter 5, an alternative approach to establishing thread co-location was proposed in HyperRace [62]. It used data races on a shared variable as a way of distinguishing L1 from LLC sharing. Accordingly, it does not require a timer thread.

TAKEAWAY: Few methods exist to detect when a malicious OS neglects a request for a system-layer microarchitectural defence; those that do exist, targeted only a narrow set of page-table attacks (except for HyperRace, developed concurrently).

2.6.3 Application Layer

The goal of application-layer testing tools is to find side-channel *information leakage*. A program leaks information via side-channels when its traces depend on the data it processes, thus exposing the data to the attacker. The testing tools focus on finding the places in the program that trigger such leakage, and on quantifying the amount of information exposed.

Testing if software is constant time. The earliest line of work in this field is testing whether a given application (usually, a cryptographic library) is constant-time. Langley [151] used dynamic taint analysis with Valgrind to find secret-dependent branches and memory accesses. Zankl et al. [278] used Intel PIN tool to detect control-flow leaks in RSA implementations. These approaches only answer *if* there is a leakage, but not *where* it is and *how much* it leaks.

Quantifying side-channel leakage. Concurrently, another line of research attempted to quantify *how much* information a given program leaks. They statically calculate the upper bound on the amount of information a program leaks on a given hardware and on a given threat model. Köpf et al. [145] built a generic formal model for it and automated the quantification [33]. They later [146] developed a model for cache side channels, and Doychev et al. [78, 79] implemented a tool that quantifies the upper bound for programs based on this model. Because of the analysis complexity, the tool is applicable mainly to small software, but it laid a theoretical foundation for later work.

Finding side-channel leakage. More recently, several papers attempted to find the specific location of side-channel leakage in software. Irazoqui et al. [132] applied dynamic taint analysis to find the parts of software that expose information via instruction or data caches. Chattopadhyay et al. [61] targeted the same goal, but with symbolic execution. Wang et al. [253] combined taint analysis and symbolic execution to improve precision. Brotzman et al. [55] implemented a symbolic model of cache evictions rather than cache accesses, thus further improving precision. These methods do not consider speculative execution.

Fuzzing has also been applied to detecting side channels. MicroWalk [264] and DATA [261] are fuzzers that instrument the target binary with PIN and collect the expected side channel traces, thus revealing side-channel leakage. CT-Fuzz [119] and DiffFuzz [183] detect information leaks by empirically measuring cache traces. Wang et al. [255] finds control-flow side channels by collecting control flow traces with Intel PT [133]. These fuzzers also do not take speculative execution into account.

Finding speculative leakage. A number of tools have been proposed for detecting and mitigating speculative information leaks in software, and many of them relied on symbolic execution. Spectector [112], Pitchfork [60], SpecuSym [115], and KLEESpectre [251] used it to detect Spectre-type vulnerabilities. Although they often provide strong security guarantees, an inherent problem of symbolic execution is combinatorial explosion, further exacerbated by nested speculation.

Static analysis is often used to detect the Spectre-type vulnerabilities and avoid the high performance cost of full hardening. Tools like Spectre 1 Scanner [66], MSVC Spectre 1 pass [173], and Respectre [126] analyze the binary and search for Spectre gadgets. Although mature tools like Respectre can detect many vulnerabilities, the reliance on predefined patterns may leave an unexpected variant to stay unnoticed.

oo7 [252] relies on static taint analysis to detect the memory accesses that are dependent on the program input. (This is the same criteria that we use to identify uncontrolled vulnerabilities in Chapter 4.) This approach is more universal than the pattern-matching techniques, but it is affected by the inherent problems of static taint analysis: A limited analysis depth may cause false positives and overtainting causes false negatives.

TAKEAWAY: The existing tools for detecting speculative vulnerabilities exercise only the extreme points in the tradeoff between precision and performance: they offer either poor performance with high security, or poor security with high performance.

We reviewed the landscape of microarchitectural vulnerabilities as well as methods to prevent and detect them. We also identified several issues in the existing techniques. The next chapters describe methods for solving these issues.

Chapter 3

Revizor: Fuzzing for Leaks in Black-box CPUs

In this chapter, our goal is to develop a method to *automatically check if CPU operations can leave unexpected and observable microarchitectural traces.*

Even though multiple tools for it already exist (§2.6.1), they all require introspection into the CPU, making them applicable only to design-time (white-box) testing. It means, only the CPU vendors can test the microarchitectural security, but not their customers.

We argue, it is insufficient: When customers cannot test the microarchitecture, its security turns into a vague and obscure property. This, in turn, skews the vendors' incentives towards more tangible CPU properties. For example, it is routine to benchmark and compare the performance of CPUs by different vendors, but there is no established way to compare their microarchitectural security. Due to this asymmetry, the market rewards vendors that increase performance at the expense of security [38], which ultimately brings on the vulnerabilities like Spectre [142] and Meltdown [158].

This market failure calls for a change: Similar to how government agencies test the safety of commercial vehicles, third parties should be able to test commercial CPUs. Indeed, security researchers have already started the process by analyzing vendors' documentation, patents, and by manually probing the microarchitecture. It prompted discoveries of even more vulnerabilities such as Foreshadow [56], MDS [247, 57, 212], and others.

The testing process, despite these achievements, remains predominantly manual, which restricts its scope to public information and limited human capabilities. The few automated tools that do exist, test for variants of only *known* vulnerabilities (see §2.6.1). Thus, it remains an open challenge to find a systematic approach to make microarchitectural security tangible.

3.1 System Overview

We introduce *Model-based Relational Fuzzing* (MRF), a method to *automatically* test microarchitectural security properties of black-box CPUs. Informally, MRF confronts our expectations about a microarchitecture with the reality of an actual CPU.

To describe the expectations, MRF relies on *speculation contracts* [113]. A contract is a specification that enhances an ISA by describing which CPU operations an attacker can observe through a side channel, and which operations can speculatively change the control/data flow. For example, a contract may state: an attacker can observe addresses of memory stores and loads, and the CPU may mispredict the targets of conditional jumps. If the CPU under test permits the attacker to observe more than that (e.g., addresses of loads after mispredicted indirect jumps), the CPU violates the contract, indicating a potential microarchitectural vulnerability.

The goal of MRF, phrased in terms of speculation contracts, is to detect contract violations automatically. To this end, it searches for a *contract counterexample*, an instruction sequence and a pair of inputs that are supposed to be indistinguishable to the attacker according to the contract, but produce two different sequences of side-channel observations (*traces*) on the CPU under test. Such a counterexample would indicate that the CPU behaves differently from what we expect, and it exposes more information than the contract permits.

Our approach is to search for counterexamples by fuzzing: Generate random instructions sequences (*test cases*) together with random inputs to them, and check if any of them constitutes a counterexample. To check this, we compare the side-channel traces observed while executing the test case on the CPU (*hardware traces*) with the traces we expect to observe based on the contract (*contract traces*). To produce hardware traces, we execute the test case with its inputs on the CPU while mounting a side-channel attack. To produce contract traces, we execute them on a functional model of the CPU (e.g., an emulator) that implements the contract. We modify the model to explore both normal and speculative execution paths (correct and mispredicted), and to record the contract-prescribed observations during program execution.

Tool. We implement MRF as a fuzzing framework Revizor[1]. It overcomes several obstacles stemming from the difficulty of black-box fuzzing of commercial CPUs.

The main obstacle is the sheer size of the search space: An ISA may include hundreds of instructions, dozens of registers, and permit large memory spaces. This creates an intractable number of possibilities for both test cases and for inputs to them. Moreover, there are no

[1]Revizor is a name of a classical play by Nikolai Gogol about a government inspector arriving into a corrupt town for an incognito investigation.

means to measure coverage for black-box CPUs, which precludes a guided search.

Nevertheless, and maybe surprisingly, we discovered that finding counterexamples is possible; under certain restrictions, it is even fast:

- For *test cases* we found that fully random generation (based on a restricted subset of instructions) is sufficient to uncover violations. The reason is that finding a counterexample is *not* akin to finding a needle in a haystack: Microarchitectural vulnerabilities have numerous counterexamples, and it is sufficient to generate only one of them to surface a violation. For example, any program with a conditional branch followed by an input-dependent memory access can surface a violation representing Spectre V1.

- For *inputs*, however, we had to apply several measures to reduce the search space. Revizor uses a subset of registers, confines all memory accesses to a narrow memory range, and bounds the entropy of inputs. These measures could prevent us from detecting "deep" violations, but they considerably improve the speed of detection: Revizor surfaces many known vulnerabilities in under two hours of fuzzing on a consumer desktop PC. As such, it trades the maximum reachable hardware coverage for improved input coverage and faster detection.

The second obstacle is measuring the effectiveness of a fuzzing campaign. When the fuzzer times out without detecting a violation, the user would want to know whether the campaign was deep or if it tested only a narrow set of features. Typically, the answer to it is coverage, but it is not available for black-box CPUs. We propose to use *pattern coverage* instead, a metric that shows the diversity of test cases and inputs in a campaign. Pattern coverage measures the number of tested instruction patterns that are likely to cause a contract violation. We also incorporate this metric for simple feedback-driven test case generation.

The third obstacle is the complexity and unpredictability of the microarchitecture in modern high-performance CPUs, which makes it challenging to obtain reliable hardware traces. To overcome this obstacle, Revizor creates a low-noise measurement environment by executing test cases in complete isolation, and it relies on performance counters instead of timing.

In addition, Revizor provides several practical features such as test case minimization, multiple measurement modes, and provides a configuration interface for fuzzing. It currently supports only Intel x86 CPUs. We selected them as the first target because they represent the worst case for our method: a superscalar CPU with several unpatched microarchitectural vulnerabilities, no detailed descriptions of speculation mechanisms, and no direct control over the microarchitectural state.

3.2 Background: Speculation Contracts

In this section, we recall speculation contracts, a generic framework for characterizing side-channels in speculative execution. However, the original formalization of contracts in terms of an operational semantics [113] is not effective for our purposes. This is why we first give an abstract account of the core idea behind contracts before we present a novel notation for expressing contracts for real-world CPUs with complex ISAs.

3.2.1 Side-channel Leakage

We consider a victim program *Prog* that executes on the same CPU as an adversary program. We assume both programs are logically isolated, but the adversary can use side-channels [240, 276, 188] to observe the shared microarchitectural state, and thus learn about *Prog*'s computation. We call a sequence of all side-channel observations made by the attacker during a program execution a *hardware trace*.

We represent the hardware trace corresponding to a specific adversary as a deterministic function

$$HTrace = Attack(Prog, Data, Ctx)$$

that takes as input three parameters: (1) the victim's program *Prog*; (2) the input $Data$ processed by the victim's program (i.e., the architectural state including registers and main memory); (3) the microarchitectural context *Ctx* in which it executes (i.e., the contents of internal CPU buffers and caches).

Which observations are exposed in a hardware trace depends on the threat model. That is, different CPUs and different adversary capabilities correspond to different functions $Attack$.

Example: If the threat model includes a shared L1 data cache, $HTrace$ is composed of the cache set indexes used by *Prog*'s loads and stores. If the threat model includes a shared LLC cache, $HTrace$ will additionally contain *Prog*'s control flow transitions such as conditional branch targets.

A program *leaks* information via side-channels when its hardware traces depend on the inputs: We assume the attacker knows *Prog* and can manipulate *Ctx*, hence any change in the hardware trace implies the input $Data$ has changed, which effectively exposes information to the attacker. For capturing such dependencies, we compare hardware traces of program executions with *different* inputs.

Definition 1: A program *Prog* leaks information on a given CPU when there is an input pair $(Data, Data')$ and a microarchitectural state *Ctx* such that:

$$Attack(Prog, Data, Ctx) \neq Attack(Prog, Data', Ctx)$$

Comparing traces as in Definition 1 is standard for reasoning about side-channel leakage (more generally: information flow properties [65]). Intuitively, this is because the information carried by a trace depends on the number of *alternative* traces one could observe. For example, if there is only one possible trace, its observation does not carry an information; if there are $n \geq 2$ possible traces, its observation can carry up to $\log_2 n$ bits of information.

3.2.2 Characterizing Leakage via Contracts

A *speculation contract* [113] (contract, for short) is a specification of the information leakage of a CPU under a given threat model. A contract specifies the information exposed by CPU instructions via side channels, both during speculative and non-speculative execution. A key feature of contracts is that they can be expressed in terms of ISA instructions. That is, they do not rely on a specific microarchitecture and can hence be applied to many CPUs, even from different vendors.

We model a contract as a deterministic function *Contract* that maps the program *Prog* and its input *Data* to the trace *CTrace* we expect to observe, and which we call a *contract trace*:

$$CTrace = Contract(Prog, Data)$$

An alternative interpretation of contracts is that all information that is *not* exposed via *CTrace* is meant to be kept secret during program execution.

Example: Consider a contract that specifies that all instructions are executed in order, and instructions accessing memory disclose their accessed addresses to the attacker. We call this contract *MEM-SEQ*, see §3.2.3 for a detailed description. When executing the program in Figure 3.1 with inputs `data={x=10,y=20}`, the *MEM-SEQ* contract trace is `ctrace=[0x110]`.

A program can also leak information during speculative execution. For example, a branch misprediction may cause a data-dependent memory access that would otherwise be impossible. Speculative leakage was exploited in Spectre attacks [142, 57, 158, 247] and can be captured using contracts.

```
1 z = array1[x]; // base of array1 is 0x100
2 if (y < 10)
3   z = array2[y]; // base of array2 is 0x200
```

Figure 3.1: Example of Spectre V1

Example: Consider a contract *MEM-COND* that is similar to *MEM-SEQ* but that additionally discloses addresses of the memory accesses that happen after a possible branch misprediction. When executing the program in Figure 3.1 with inputs `data={x=10,y=20}`, the *MEM-COND* contract trace is `ctrace=[0x110,0x120]`, accounting for the fact that the load at line 3 can be speculatively executed.

A CPU *complies* [113] with a contract when its hardware traces (collected on the CPU) leak at most as much information to the attacker as the contract traces. Formally, we require that whenever two executions of a program have the same contract trace, they must produce the same hardware trace.

Definition 2: A CPU complies with a $Contract$ if, for all programs $Prog$, all input pairs $(Data, Data')$, and all initial microarchitectural states Ctx:

$$\begin{gathered} Contract(Prog, Data) = Contract(Prog, Data') \\ \implies Attack(Prog, Data, Ctx) = Attack(Prog, Data', Ctx) \end{gathered}$$

Note that Definition 2 is equivalent to the requirement that any leak according to Definition 1 is visible at the ISA level in terms of mismatching contract traces.

Conversely, a CPU *violates* a contract if there is a program $Prog$ and two inputs $Data, Data'$ that agree on their contract traces but *dis*agree on the hardware traces. We call the triple $(Prog, Data, Data')$ a contract *counterexample*: it witnesses that an adversary can learn more information from hardware traces than what the contract specifies. A counterexample is a potential microarchitectural vulnerability that was not accounted for by the contract.

Example: A CPU with branch prediction will violate *MEM-SEQ*. Its counterexample is the program on Figure 3.1 together with inputs `data1={x=10,y=20}`, `data2={x=10,y=30}`: The contract trace for both inputs is `ctrace=[0x110]`, but the hardware traces will diverge (`htrace1=[0x110,0x220]` and `htrace2=[0x110,0x230]`) if the CPU mispredicts the branch (line 2) and speculatively accesses memory (line 3). At the same time, this is not a counterexample to *MEM-COND*, because the contract traces for both inputs already expose the memory accesses on mispredicted paths.

```
1 observation_clause: MEM
2   - instructions: MemReads
3     format: READ (ADDR), DEST
4     observation: expose(ADDR)
5   - instructions: MemWrites
6     format: WRITE SRC, (ADDR)
7     observation: expose(ADDR)
8 execution_clause: COND
9   - instructions: CondBranches
10    format: COND_BR CONDITION (DEST)
11    execution: {
12      if NOT CONDITION then
13          IP := IP + DEST
14      fi }
```

Figure 3.2: *MEM-COND* described in pseudocode.

3.2.3 Describing Contracts

In [113] contracts are specified in terms of an operational semantics for a toy assembly language. For scaling contracts to complex ISAs such as x86, we take the following steps:

First, we specify contracts not for a complete ISA (as in the original paper), but for a *subsets* of it. That is, we view contracts as partial specifications of leakage that make no assertion about the instructions not covered. This permits us to test complex CPUs through incremental expansion of the covered ISA subset.

Second, we specify contracts in terms of the *instruction classes*, rather than in terms of individual instructions.

Finally, we describe contracts in a modular fashion using two building blocks: *observation clause*, which describes what is exposed to an adversary, and *execution clause*, which expresses how to handle instructions that trigger speculation.

Example: Figure 3.2 shows *MEM-COND*, written in our custom (informal) notation. The observation clause (lines 1–7), specifies observations for two instruction classes: memory reads (lines 2–4) and memory writes (lines 5–7). The `format` field describes the format common across all instructions in the class: e.g., line 3 specifies that the first operand of all instructions in `MemReads` is the memory address of the read, and the second operand is the read destination. The `observation` field specifies the information exposed by the instructions: Line 4 specifies that the addresses of all reads are exposed to the attacker (i.e., added to the contract trace). The execution clause (lines 8–14) describes the speculation algorithm. It states that conditional branches (line 9) mispredict their targets as described by the `execution` field (11–14): If the condition does *not* hold, write the jump target (`DEST`) to the instruction pointer (IP).

3.2.4 Concrete Contracts

We will consider a wide range of contracts that are constructed from several common observation and execution clauses. For example, the *MEM-SEQ* contract is a combination of the *MEM* observation clause and *SEQ* execution clause. We informally describe the individual clauses below; see Appendix B for detailed specifications.

Observation clauses:

- *MEM (Memory Address)*: exposes the addresses of loads and stores. Represents a data cache side-channel attack.
- *CT (Constant-Time)*: MEM, but also exposes the program counter. Represents data and instruction cache. Based on a typical constant-time programming threat model.
- *CTR*: CT, but additionally exposes register values. Represents a cache attack in an execution environment that does not clear registers when switching security domains.
- *ARCH (Architectural Observer)*: CTR, but additionally exposes the values loaded from memory. Represents a same-address-space attack, such as the threat model used in STT [277]. In this contract, only transiently loaded data is considered secret.

Execution clauses:

- *SEQ*: Sequential (i.e., in-order) execution.
- *COND*: Conditional jump misprediction. All jumps speculatively take a wrong target.
- *BPAS*: Speculative store bypass. All stores are speculatively skipped.
- *COND-BPAS*: Combination of *COND* and *BPAS*.

3.3 Background: Fuzzing

Fuzzing is a technique for discovering bugs and vulnerabilities by exposing the fuzzing target to diverse conditions and inputs. A fuzzing tool (*fuzzer*) automatically generates randomized inputs either from scratch, based on input grammars, or by mutating an existing input corpus. The fuzzer then feeds these inputs to the application and monitors its behavior: If an abnormal behavior (e.g., a crash) is observed, the fuzzer reports a bug. Fuzzing is typically applied to testing software, but in this chapter, we transfer it to hardware.

One important parameter of fuzzing is its *coverage*, which indicates how extensively the target was tested. Coverage mainly depends on how effectively a fuzzer can generate inputs that trigger new behaviour in the fuzzing target.

```
 1 def fuzzing_round():
 2   test_case = generate_test_case()
 3   inputs = generate_inputs(test_case)
 4   ctraces = collect_traces_on_model(test_case, inputs)
 5   htraces = collect_traces_on_CPU(test_case, inputs)
 6   if has_violations(ctraces, htraces):
 7     report_violation()
 8
 9 def has_violations(ctraces, htraces):
10   classes = group_by_ctrace(ctraces, htraces)
11   for class in classes:
12     if not all_htraces_equal(class.htraces):
13       return True
14   return False
```

Figure 3.3: Model-based Relational Fuzzing Algorithm.

3.4 Method: Model-based Relational Fuzzing

This section presents Model-based Relational Fuzzing (MRF), a method for identifying unexpected information leaks in black-box CPUs. MRF searches for a program and a pair of inputs that produce identical contract traces but different hardware traces. Such counterexamples witness that the observed leakage on the CPU diverges from our expectations.

3.4.1 Core Concepts

As the acronym suggests, MRF builds on three main ideas, which we describe next. See Figure 3.3 for accompanying pseudocode.

Fuzzing. We use random generation to create candidate counterexamples: We generate small random programs (test cases) and random inputs to them (lines 2–3).

Comparing against Model. We collect contract traces corresponding to the test cases and the inputs (line 4) on a CPU emulator. For this, we modify the emulator to implement speculative execution as described in the contract's execution clause, and to record the traces corresponding to its observation clause (see §3.2.4). This contract implementation serves as the *model* of the contract-prescribed leaks.

We also collect hardware traces corresponding to the test cases and the inputs under a given threat model (line 5). For this, we mount a side-channel attack on the CPU under test during the execution of the test cases. For example, we may run a Prime+Probe attack and record which cache sets are touched by the test case.

Relational Analysis. We process the contract and hardware traces to identify potential violations of Definition 2 (lines 9–14). This requires *relational* reasoning, namely the comparison of contract (and hardware) traces for different inputs: (1) We partition inputs into groups, which we call *input classes*[2], such that all inputs within a class have the same contract trace (line 10). (2) For each of the classes, we check if all inputs within a class have the same hardware trace (lines 11–14). If the check fails, we found a contract violation according to Definition 2.

3.4.2 Practical Challenges

There are several challenges in applying MRF to commercial CPUs:

CH1: Search Complexity. The main challenge for MRF is the overwhelming number of possible test cases and inputs to them, making exhaustive search impossible.

CH2: Input Effectiveness. The relational analysis algorithm (Figure 3.3) groups inputs into classes based on the contract traces. For a class to be useful in the analysis, it needs to contain at least two inputs (we call them *effective classes*). The *in*effective classes constitute a wasted effort as they cannot, by definition, reveal a contract violation. This becomes a challenge when the inputs are generated randomly, as the probability that multiple random inputs will produce the same contract trace (i.e., form an effective class) is low.

CH3: Measurement Noise. The measurement of hardware traces is influenced by other processes running on the system, by hardware effects (e.g., prefetching), and by the inherent imprecision of the measurements (e.g., timing measurements). This may lead to a divergence between the hardware traces that would otherwise be equal and, accordingly, to a false contract violation (false positive).

CH4: Non-deterministic Microarchitectural State. There is no direct way of setting a black-box CPU to a desired microarchitectural state. Thus, the effects of program execution (e.g., whether speculation is triggered) are partially non-deterministic. It may lead to the divergence among hardware traces when some of the measurements experienced speculation while others did not (false positive).

False compliance (false negative) is also possible if the speculation is never triggered during the measurement, although we expect it unlikely when fuzzing for an extended time.

CH5: Cross-training. The previous issue is especially prominent when testing inputs in a sequence: The earlier inputs may prime the microarchitectural state (e.g., train branch predictors), and the latter inputs will not experience speculation, leading to a false positive.

[2]Input classes correspond to the equivalence classes of equality on contract traces, hence the name.

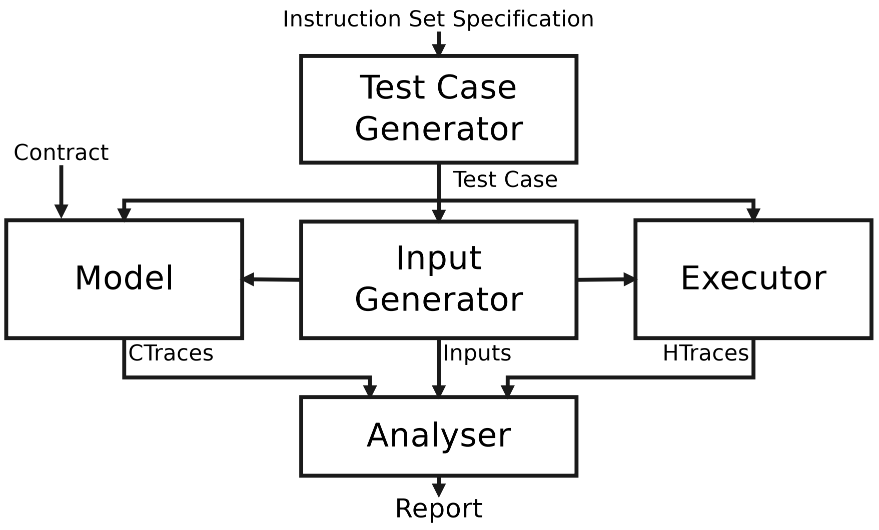


Figure 3.4: Components of Revizor.

3.5 Design and Implementation

We built Revizor, a fuzzing framework that implements MRF for testing a CPU against a contract, as described in §3.4.

Figure 3.4 shows Revizor's architecture and main components: *Test Case Generator* and *Input Generator* are responsible for creating test cases and generating inputs to them, respectively. *Model* executes them on an emulator according to the contract and collects contract traces. *Executor* executes the test cases on the target CPU, records observations via a side-channel attack, and produces hardware traces. *Analyser* compares the traces and searches for violations. If found, the contract counterexample is reported to the user.

We describe the individual components of Revizor in the remainder of this section. For illustration purposes, we will build on the following running example.

Example: Consider a user who seeks to find whether a CPU leaks information during speculation via L1D cache. The appropriate contract for compliance testing is *MEM-SEQ*, since it prohibits any information to be exposed during the speculation, so its violation indicates CPU leaks under this threat model. The user configures Revizor to apply a Prime+Probe attack mechanism on L1D cache while collecting hardware traces. To expedite testing the user starts with a subset of x86: arithmetic operations and conditional branches. For the initial quick test, they select a configuration with small test cases: 8 instructions long, with at most 4 memory accesses, and 3 basic blocks. The user encodes all of this information into a configuration file passed to Revizor and starts the fuzzer. The following examples illustrate different aspects of the fuzzing process.

3.5.1 Test Case Generator

In Revizor, test case generation is random but guided by a user configuration. The configuration specifies, for example, the total number of instructions, functions, and basic blocks per function, as well as the subset of the ISA from which test cases are built. The description of instructions comes from the structured ISA specification shipped with nanoBench [13].

The generation algorithm is a series of passes, similar to a typical compiler:

1. Generate a random Directed Acyclic Graph (DAG), following the user configuration;
2. Add jump instructions (terminators) at the end of each node (basic block) to ensure control flow matches the DAG;
3. Populate the basic blocks with random instructions from the selected ISA subset;
4. Instrument instructions to avoid faults[3]:
   (a) mask memory addresses to confine them within a dedicated memory region (sandbox);
   (b) modify division operands to avoid division by zero;
   (c) truncate bit test offsets;
5. Compile the test case into a binary.

We opted for DAG-based generation in contrast to entirely random instruction sequences because it allows us to contain the control flow of test cases.

Example: Figure 3.5 shows an example of a test case that is produced in multiple steps: ① The generator created three nodes (recall, the user requested 3 basic blocks). ② Connected the nodes to form a DAG. As `BASE+MEM+CB` includes conditional jumps, the generator randomly connected each node with either one or two other nodes by placing the corresponding jump at the node's end (lines 6–7, 14). ③ Added random instructions with random arguments to each basic block until the requested size—eight instructions—was reached (lines 2, 5, 11–13, 16, 19, 20). The instructions were selected from the user-requested ISA subset. ④ Masked memory accesses to ensure they stay within a sandbox, and aligned them to the sandbox base in R14 (lines 3–4, 9–10, 17–18). The generator also added `MFENCE` at the beginning and the end of the test case to contain speculative execution. ⑤ Compiled the resulting test case.

Practical challenges. To improve effectiveness (CH2), all generated instruction sequences employ only four CPU registers, thus increasing the likelihood of grouping random inputs. For the same reason, all memory accesses are aligned to a cache line size (64B), thus

[3]More specifically, we avoid those faults for which the executor does not have a handler. Otherwise, the control could be transferred to the OS, which would corrupt the measurements and might crash the machine.

```
MFENCE
OR RAX, 468722461
AND RAX, 0b111111000000
ADD RAX, R14
LOCK SUB byte ptr [RAX], 35
JNS .bb1
JMP .bb2
.bb1:
AND RCX, 0b111111000000
ADD RCX, R14
REX SUB byte ptr [RCX], AL
CMOVNBE EBX, EBX
OR DX, 30415
JMP .bb2
.bb2:
AND RBX, 1276527841
AND RDX, 0b111111000000
ADD RDX, R14
CMOVBE RCX, qword ptr [RDX]
CMP BX, AX
MFENCE
```

Figure 3.5: Randomly generated test case

reducing the range of accessible memory addresses. To still test different cache offsets, the addresses are offset by a random value between 0 and 64 (cache line boundary), such that the offset is the same for all accesses within a test case but different across test cases.

3.5.2 Input Generator

An input is a set of values to initialize the architectural state, which includes registers used by the test case (including `FLAGS`) and memory (one or two pages). The input generator produces the values randomly, with a 32-bit PRNG.

Practical challenges. To increase the input effectiveness (CH2), the generator can optionally reduce the entropy of inputs by masking some of the output bits.

3.5.3 Executor

The executor's task is to collect hardware traces for test cases and inputs executed on a real CPU. It involves the following steps:

1. Load the test case into a dedicated region of memory,
2. Set memory and registers according to the inputs,
3. Prepare a side-channel attack (e.g., prime cache lines),

4. (Optional) Prime the microarchitectural state in an attempt to reset it to a known state,
5. Invoke the test case,
6. Measure the microarchitectural changes with a side-channel attack, thus producing a hardware trace.

The executor repeats the algorithm for each test case, and the measurement (steps 2–6) for all inputs, thus collecting a hardware trace for each test case-input pair.

Our executor implementation supports several attack modes, each corresponding to a different threat model:

- In *Prime+Probe* [188], *Flush+Reload* [110], and *Evict+Reload* [111] modes, the executor uses the corresponding attack on L1D cache and avoids faults during measurements. These modes correspond to an attacker that passively observes the victim.

- In **+Assist* mode, the executor extends each of the modes above to include microcode assists. To this end, the executor clears the "Accessed" page table bit for one of the pages accessible by the test cases such that the first store or load to this page triggers an assist. This mode corresponds to an attacker that actively manipulates the execution environment.

Example: When the executor receives the test case (Figure 3.5), it loads the binary, and initializes memory and registers. Next, it mounts a Prime+Probe attack (as specified in the configuration): it evicts the cache lines that map to the test case memory, executes the test case, and then reads the eviction set while monitoring cache misses. Each miss exposes a memory access.

The attack produces a hardware trace consisting of a sequence of bits, each representing whether a cache set has been touched by a memory access of the test case. The following is an example trace (representing the first 32 cache sets) that shows memory accesses to cache sets 5, 14, and 32: `00001000000001000000000000000001`

Practical challenges. The executor eliminates four sources of inconsistency as follows:

1. *Eliminating side-channel noise* (CH3). Instead of relying on timing, the executor detects evictions with performance counters by comparing the L1D miss counter before and after probing a cache line. Performance counters give more stable results than timing.

2. *Eliminating external software noise* (CH3). We run the executor as a kernel module (based on nanoBench [13]). A test is executed on a single core, with hyperthreading, prefetching, and interrupts disabled. The executor also monitors System Management Interrupts (SMI) [130] to discard those measurements polluted by an SMI.

3. *Reducing nondeterminism* (CH4). As we cannot fully reset the microarchitectural state before each execution, we perform each measurement multiple times (50 in our

experiments) after several rounds of warm-up. The traces that appeared only once (i.e., include unique memory accesses) are discarded as they are likely caused by the noise. The remaining traces are merged—the final hardware trace is a union of all traces collected by repeatedly executing the same test case with the same input.

4. *Eliminating cross-training effects* (CH5). Since the executor collects traces for different inputs in a sequence, one input might affect the trace of the others. In this case, the difference in traces is an artifact of the difference in the microarchitectural state.

We filter such cases as follows. Consider two inputs $Data_1$ and $Data_2$ that produced mismatching traces $HTrace_1$ and $HTrace_2$. These inputs were executed in the microarchitectural contexts Ctx_1 and Ctx_2, respectively:

$$Attack(Prog, Data_1, Ctx_1) \neq Attack(Prog, Data_2, Ctx_2)$$

To verify that the divergence was caused by the information leakage and not by the difference in the microarchitectural context, we test $Data_1$ with the context Ctx_2 and vice versa. That is, we test if the following holds:

$$\begin{aligned} &Attack(Prog, Data_1, Ctx_2) = Attack(Prog, Data_2, Ctx_2) \\ \wedge\ &Attack(Prog, Data_2, Ctx_1) = Attack(Prog, Data_2, Ctx_1) \end{aligned}$$

If both hold, this is a false positive; otherwise, it is a violation.

As we cannot directly control black-box microarchitecture, we set Ctx_1 by executing the sequence of inputs that preceded the input $Data_1$. We call this process *priming*.

Example: Consider two inputs from the same input class that produced different traces. Within the measurements sequence, the first input was at position 100 (i_{100}) and the second–200 (i_{200}). For priming, the executor creates input sequences $(i_1 \ldots i_{99}, i_{200}, i_{101} \ldots i_{199}, i_{200})$ and $(i_1 \ldots i_{99}, i_{100}, i_{101} \ldots i_{199}, i_{100})$ and re-measures the traces. The executor will consider it a false positive if i_{100} at position 200 produces the same trace as i_{200} at position 200, and vise versa.

Ideally, priming should be done for every pair of inputs, but it would slow down the fuzzing dramatically. Therefore, we apply priming only to the inputs that are suspected to show information leakage, as we explain in §3.5.5. If the number of suspects is large, we prime them in batches.

3.5.4 Model

Model collects contract traces for test cases and inputs by executing them on an ISA-level emulator modified according to the target contract. The emulator speculatively executes instructions based on the contract's execution clause and collects observations based on the observation clause. The complete contract trace is a list of all such observations collected while executing a test case with a single input. In our implementation, we use Unicorn [196], a customizable x86 emulator, which we manually modified to implement the clauses listed in §3.2.3.

Observation Clauses. Every time the emulator executes an instruction listed in the observation clause, it records its exposed information into the trace. This happens during both normal and speculative execution, unless the contract explicitly states otherwise.

Example: The contract (*MEM-SEQ*) exposes addresses of reads and writes. Thus, when executing lines 5, 11, 19 (Figure 3.5), the model records the accessed addresses. Suppose, the branch at line 6 was not taken; the store at line 5 accessed the address `0xaaaa100`; and the load at line 19 accessed `0xaaaa340` (here, `0xaaaa000` is the sandbox base, `0x100` and `0x340` are offsets within the sandbox). Then, the resulting contract trace is `ctrace=[0xaaaa100, 0xaaaa340]`.

Execution Clauses. Executions clauses are implemented as follows: The emulator takes a checkpoint of the architectural state when it encounters an instruction listed in the clause (e.g., a conditional branch in *MEM-COND*). Then, it executes a misprediction as described by the instruction's `execution` filed (e.g., lines 14–16 in Figure 3.2). It continues speculative execution until the end of the test case, until the maximum possible speculation window is reached, or until the first serializing instruction. After that, the model rolls back to the checkpoint and continues normal execution.

Since several mispredictions might happen together, the model supports nested speculation. To this end, it maintains a stack of checkpoints: at a speculatable instruction a checkpoint is pushed on the stack and, when the simulation is over, the model rolls back to the topmost checkpoint on the stack.

This method explores all possible speculative behaviors permitted by the contract. Such a worst-case simulation ensures that we reliably collect all contract traces representing legitimate executions of the given test case. Practically, however, nesting greatly reduces the fuzzing speed. This is why we disable nesting by default but, upon a violation, Revizor re-enables it and re-runs the experiment.

Practical challenges. Performing exhaustive exploration of all speculative behaviors permitted by the contract may require emulating a very large number of paths, which may lead to a combinatorial explosion. To avoid it, we restrict the test case generator to create code sequences according to certain rules: restrict the number of memory locations, branch targets, and the total number of instructions in the generated test cases.

3.5.5 Analyser

The analyser detects unexpected leakage by comparing hardware traces that belong to the same input class. It implements an algorithm similar to `has_violations` in Figure 3.3. In practice, however, the cross-training effect (CH5) may cause a mismatch. Even though microarchitectural priming (§3.5.3) can prevent false positives in these cases, it is a slow operation, and the effect is common enough to turn it into a major performance bottleneck.

Therefore, we relax the requirement for contract compliance by requiring only a subset relation between hardware traces, not a full match: One hardware trace is a subset of another trace if every observation included in the first trace is also included in the second trace.

The intuition behind this heuristic is as follows. If the difference in traces is caused by speculation not being triggered in a measurement, it will lead to fewer instructions being executed and, accordingly, to fewer observations in the trace, yet the observations in the sub-trace would still match those in the full trace. On the other hand, if the difference is caused by a genuine information leakage (e.g., secret-dependent memory access), the traces would contain about the same number of observations, but their values would differ.

Example: Revizor executed the test case with two inputs, which produced the same contract trace `ctrace=[0xaaaa100,0xaaaa340]`. It contains two addresses because the branch (line 6) was not taken. The same inputs produced two different hardware traces because the branch was mispredicted:

```
htrace1=00001010000001000000000000000000
htrace2=00001001000001000000000000000000
```

The analyser detects this mismatch: Since inputs 2 and 3 agreed on their contract traces but produced different hardware traces, Revizor reports them as a violation.

3.5.6 Postprocessor

Finally, when a true violation is detected, the test case is (optionally) passed to the postprocessor. It minimizes the test case in three stages.

First, it searches for a minimal input sequence. Since we cannot directly control the microarchitectural state, we need not only the pair of inputs that produce mismatching hardware traces, but also several preceding inputs to prime the microarchitectural state. Thus, the postprocessor gradually removes inputs until it finds a minimum sequence that still reproduces the violation.

Second, the postprocessor creates a minimized test case: It removes one instruction at a time and checks if the violation persists.

```
1  AND RAX, 0b111111000000
2  LFENCE
3  ADD RAX, R14
4  LFENCE
5  LOCK SUB byte ptr [RAX], 35
6  JNS .bb1
7  LFENCE
8  JMP .bb2
9  .bb1:
10 AND RCX, 0b111111000000
11 ADD RCX, R14
12 REX SUB byte ptr [RCX], AL
13 LFENCE
```

Figure 3.6: Minimized test case.

Finally, it minimizes the speculation that led to the violation: It gradually adds `LFENCE`s, starting from the last instruction, while checking for violations. As a result, the region of the test case without `LFENCE`s is the region where the speculative leakage happens.

Example: Figure 3.6 shows a minimized version of Figure 3.5. The highlighted region without `LFENCE`s is the speculative leakage: The load at line 5 delays the next conditional jump, thus creating a sufficiently large speculation window. The jump at line 6 speculatively jumps into line 9. This causes a speculative execution of `SUB` (line 12), which has a memory operand and thus leaks the value of `RCX`.

3.6 Coverage

We fuzz CPUs in a black-box fashion, which is why we do not attempt to measure coverage of implementation-specific features. Instead, we focus on measuring a contract's input coverage. Full input coverage means to consider all possible programs *Prog*, all possible inputs *Data*, and all possible contexts *Ctx*, see Definition 2.

As this space is dauntingly large, we introduce the notion of *pattern coverage*, which captures program patterns and inputs that are likely to surface speculative leaks. This metric measures the diversity of test cases, but it does not correspond to the coverage of microarchitectural features.

3.6.1 Pattern Coverage

Pattern coverage captures the number of user-defined instruction sequences of interest (i.e., instruction patterns) encountered within a fuzzing campaign. Specifically, we focus on

```
1 AND rbx, rax
2 CMP rbx, 0
3 JNE .l1:
4   MOV rax, [rcx]
5 .l1: MOV [rdx], rax
```

Figure 3.7: Illustration of pattern coverage.

the patterns of data and control dependencies as they cause pipeline hazards (§2.1.3), and hazards can cause speculation. That is, we expect good pattern coverage to increase the chances to surface speculative leaks.

As we will see next, covering all patterns is trivial; therefore, we set the goal of fuzzing as to cover as many *pattern combinations* as possible. This ensures we have many possibilities for speculation in a campaign, and also test the interaction between different types of speculation happening together.

Dependency Patterns. We define dependency patterns in terms of *pairs* of instructions. We distinguish between three classes:

- Memory dependency patterns occur when two memory accesses use the same address. We consider 4 patterns: store-after-store, store-after-load, load-after-store, load-after-load.
- Register dependency patterns occur when one instruction uses a result of another instruction. We define 2 patterns: dependency over a general-purpose register, and over `FLAGS`.
- Control dependency patterns occur when the first instruction modifies the control flow. For the instruction sets we consider in this chapter, there are two types of control dependencies: conditional and unconditional jumps. A larger instruction set may also include indirect jumps, calls, returns, etc.

Example: Consider the test case on Figure 3.7. Lines 1–2 are a dependency pattern over a general-purpose register. Lines 2–3 are a dependency over the `FLAGS` register. Lines 3–4 (and lines 3 and 5) are conditional control dependencies. Lines 4–5 are a potential store-after-load pattern.

To facilitate counting occurrences of patterns in a fuzzing campaign, we add the requirement that dependency patterns appear as consecutive instructions, which corresponds to the worst case in terms of creating hazards. We say that a test case with an input *matches* a pattern if the pattern occurs as two consecutive instructions in the corresponding instruction stream.

Covering Patterns. Simply matching a dependency pattern is not sufficient for surfacing a contract violation. To find a counterexample to Definition 2, a test case must be executed with multiple inputs from the same input class, which we call colliding inputs.

Definition 3: A pattern is *covered* by a test case *Prog* and a set of inputs *S*, if the pattern matches *Prog* for at least two inputs in *S* that are in the same input class.

Pattern Combinations. A combination is a multiset of patterns; that is, a set where each element can occur multiple times. A test case and a pair of inputs covers a pattern combination if the inputs are colliding and they covers all patterns in the combination (with the respective number of occurrences).

Implementation. We implement tracking of patterns within the model (§3.5.4). While executing a test case to collect its contract traces, the model also records the executed instructions and the addresses of memory accesses. These data are later analysed to find the patterns in the instruction streams.

3.6.2 Coverage Feedback

We use coverage of pattern combinations as feedback for the test case generator. We apply a simple fuzzing strategy where we begin with small test cases and increase their size based on the coverage.

Test Case Size. Our goal is to cover the largest possible number of pattern combinations while minimizing the number of ineffective inputs. Intuitively, the longer a test case is, the harder it is to find effective inputs.

To illustrate this under suitable independence assumptions, consider contract traces of length ℓ consisting of observations from a set of size s, i.e. $d = s^\ell$ possible contract traces. Consider a set S consisting of $n-1$ inputs. The probability $p(n,d)$ that a new (nth) input collides with an input in S (i.e., that its contract trace is identical to one of those generated by inputs in S) is given by

$$p(n;d) = 1 - \left(1 - \frac{1}{d}\right)^n$$

The probability $p(n;s^\ell)$ quickly approaches 0 as ℓ grows, which indicates that small test cases require fewer inputs (assuming random input generator), and are thus faster to test.

Fuzzing Strategy. We start fuzzing with test cases of size n and control flow graphs of at most m nodes, each tested with k inputs (e.g., 10 instructions, 2 basic blocks, 50 inputs per test case). We fuzz until we have covered all pattern combinations of size 1 (i.e., all patterns on their own). Then, we increase the sizes by constant factors (e.g., to 15 instructions, 3 basic blocks, 75 inputs), and we continue fuzzing until we have covered all pattern combinations of size 2 (i.e., all possible pairs of patterns), and so on.

3.7 Evaluation

In this section, we demonstrate Revizor's ability to efficiently check and expose CPU violations of speculation contracts by fuzzing two generations of commodity Intel CPUs.

Testbed. We used two machines: The first was Intel Core i7-6700 CPU (Skylake) with 32GB RAM, the second was Intel Core i7-9700 CPU (Coffee Lake) with 16GB RAM. Both run Linux v5.6.13.

Both Skylake and Coffee Lake have multiple known vulnerabilities. In the context of our experiments, the relevant vulnerabilities are Spectre V1 and, if the corresponding patch is disabled, V4. Skylake is also vulnerable to MDS, and Coffee Lake to LVI-Null.

Instruction Sets. All experiments were performed on one of the following subsets of x86:

- `BASE`: in-register arithmetic, logic and bitwise operations;
- `BASE+MEM`: `BASE`, but with memory operands and instructions involving memory accesses (e.g., `PUSH/POP`);
- `BASE+CB`: `BASE` with conditional jumps[4];
- `BASE+MEM+CB`: `BASE` with memory accesses and branches.

We excluded all multiplication, bit count, bit test, and shift instructions because Unicorn sometimes emulates them incorrectly. This totalled in the following number of instructions and variants (a variant is when the same instructions may have different types of operands): `BASE`—336; `BASE+MEM`—687; `BASE+CB`—368; `BASE+MEM+CB`—719.

Configuration. Unless mentioned otherwise, the test case generator was in a feedback-driven mode (§3.6.2): started from 8 instructions, 2 memory accesses, and at most 2 basic blocks per test case, and increased based on coverage. The entropy started from 2 bits and the number of inputs from 50, and also increased with coverage. Executor was in Prime+Probe mode.

3.7.1 Fuzzing CPUs

In our main experiment, we fuzzed Skylake and Coffee Lake to test the behaviour of different features in these CPUs. To this end, we fuzzed them against an increasingly wide instruction

[4]We did not include indirect jumps as they create more complicated control flow, which is harder to confine. This is not a principled issue, but it will require additional features added to the test case generator.

CPU	Skylake					Coffee Lake
Assist Mode	off				on	on
V4 patch	off		on		on	on
Instruction Set	`BASE`	`BASE+MEM`	`BASE+MEM`	`BASE+MEM+CB`	`BASE+MEM`	`BASE+MEM`
CT-SEQ	✓	× (V4)	✓	× (V1)	× (MDS)	× (LVI-Null)
CT-BPAS	✓’	⊗ (V4-var)	✓’	× (V1)	× (MDS)	× (LVI-Null)
CT-COND	✓’	× (V4)	✓’	⊗ (V1-var)	× (MDS)	× (LVI-Null)
CT-COND-BPAS	✓’	⊗ (V4-var)	✓’	⊗ (V1-var)	× (MDS)	× (LVI-Null)

Table 3.1: Summary of the fuzzing results. ✓ means that no violation was detected, and ✓’ that we did not repeat the experiment as a stronger contract was already satisfied. × means Revizor detected a contract violation, and ⊗ that the detected violation was unexpected. In parenthesis are the names of vulnerabilities which are similar to the detected violations.

set and gradually enabled more and more CPU features. We also tested each configuration with several contracts to “permit” some of the known vulnerabilities (e.g., *CT-COND* permits branch prediction). For each combination, we fuzzed for 24 hours or until the first violation. The results are in Table 3.1.

Baseline (column 2). To establish the baseline, we began with fuzzing the most vulnerable target—Skylake with V4 patch disabled—against the most narrow instruction set (`BASE`). As this instruction set does not cause speculation, Revizor did not detect any violations, even for the most restrictive contract we have (*CT-SEQ*); that is, Revizor did not produce false positives.

Testing Memory Accesses (column 3). To test the handling of the memory accesses in Skylake, we extended the instruction set with them (`BASE+MEM`). For *CT-SEQ* and *CT-COND*, Revizor found violations (Appendix C.2), and under inspection they appeared to be representative of Spectre V4 [101].

For the contracts that permit store bypass (*CT-BPAS* and *CT-COND-BPAS*) we did not expect to find a violation, but Revizor detected one. The violation was an instance of a novel variant of Spectre V4, described in §3.7.3. (Notably, this violation was rare and Revizor did not always detect it when we repeated this experiment.)

Testing V4 patch (column 4). To test the effectiveness of the V4 patch, we enabled it and repeated the same experiment. The patch appeared to be effective as Revizor timed out w/o violations.

Testing Conditional Branches (column 5). To test the handling of conditional branches in Skylake, we extended the instruction set with them (`BASE+MEM+CB`). For *CT-SEQ* and *CT-COND*, Revizor found violations (Appendix C.1) representative of Spectre V1 [142]. For the contracts that permit branch prediction (*CT-COND*, *CT-COND-BPAS*), Revizor found a

	V4	Detection time			
Contract	patch	V1-type	V4-type	MDS-type	LVI-type
CT-SEQ	on	4’51” (0.9)	73’25” (0.7)	5’35” (0.7)	7’40” (1.1)
CT-BPAS	off	3’48” (0.7)	N/A	6’37” (0.8)	3’06” (1.0)
CT-COND	on	N/A	140’42” (0.6)	7’03” (0.8)	3’22’ (0.3)

Table 3.2: Vulnerability detection time. Each cell is the time from the fuzzing start to the first detected violation. The numbers are mean values over 10 measurements, and the coefficients of variation are in parentheses.

novel variant of V1, described in §3.7.3.

Testing Microcode Assists (column 6). To test the handling of microcode assists, we enabled them during fuzzing. Revizor detected violations (Appendix C.3) representative of MDS [212, 57].

Testing MDS patch (column 7). Finally, to test the hardware patch against MDS, we repeated the experiment on Coffee Lake that has this patch. Revizor detected violations on it as well (Appendix C.4), representative of LVI-Null [244].

Overall, the experiments show that Revizor is stable, can successfully detect known and novel vulnerabilities of diverse types, and is useful in testing microarchitectural defenses.

3.7.2 Detection Time

Next, we evaluated how much time it takes for Revizor to detect a violation. We fuzzed each of the violation types detected in §3.7.1 (except for the new variants of V1 and V4 as they are too rare for practical measurements). Detection of each of these vulnerabilities was first tested in isolation, and then in presence of another vulnerability that was permitted by the contract.

To surface these vulnerabilities, we had the following setups:

- V1-type: `BASE+MEM+CB`; Skylake; assists disabled.
- V4-type: `BASE+MEM`; Skylake; assists disabled.
- MDS-type: `BASE+MEM`; Skylake; assists enabled.
- LVI-type: `BASE+MEM`; CoffeeLake; assists enabled.

To test detection of a vulnerability in isolation, we fuzzed against *CT-SEQ* (disallows all speculation) with Spectre V4 microcode patch disabled. To include another vulnerability, we fuzzed (1) against *CT-COND* (permits Spectre V1) with the Spectre V4 patch enabled and (2) against *CT-BPAS* (permits Spectre V4) with the patch disabled. We fuzzed in each configuration until a violation was detected, and recorded the elapsed time. Each experiment was repeated 10 times, and the mean values are shown in Table 3.2.

Vulnerability	Inputs to Vulnerability
Spectre V1 [142]	6
Spectre V1.1 [140]	6
Spectre V2 [142]	4
Spectre V4 [101]	62
Spectre V5-ret [164, 147]	2
MDS-AD-LFB [247, 212]	2
MDS-AD-SB [57]	12

Table 3.3: Detection of known vulnerabilities on manually-written test cases. *Inputs to Vulnerability* is the average minimal number of random inputs necessary to surface a violation.

Most of the time, Revizor detected the vulnerability in under 10 minutes, and the presence of a permitted vulnerability did not hinder detection. It took longer to detect V4 vulnerabilities because they required a longer speculation window, and it was harder to find a test case with misprediction as the address predictor tended to be conservative. It was also more impacted by the contract complexity, and the presence of branch misprediction doubled the detection time as it made harder to find effective inputs.

Manually Written Test Cases. A more sophisticated test case generator could surface more vulnerabilities than shown in Table 3.2, and faster. To illustrate it, we tested several manually-written test cases representing Spectre-type vulnerabilities and versions of MDS. Each test case was tested with an increasing number of inputs until a violation was detected. The results are in Table 3.3 (the reported number is an average of 100 experiments, each with a different input generation seed). As Revizor successfully detected all violations with very few inputs (i.e., in less than a second), which shows the importance of further research on targeted test case generation.

3.7.3 Unexpected Contract Violations

In our experiments, Revizor discovered two new (to the extent of our knowledge) variants of speculative vulnerabilities. As they represent variations on the known vulnerabilities (Spectre V1 and V4) and the corresponding defences prevent them, we did not report them to Intel. Yet, these nuances may be important to consider when developing future defences, hence we describe them here.

Speculative Latency-Dependent Leakage (V1-Var and V4-Var). The execution time of some operations on Intel CPUs depend on their operand values (e.g., division). When a speculative memory access is dependent on such a variable-latency operation, it creates a

```
1 b = variable_latency(a)
2 if (...)  # misprediction
3     c = array[b] # evicts only when ‘a‘ causes small latency
```

Figure 3.8: New Spectre V1 variant (V1-Var), found by Revizor.

race condition between this operation and the instruction that triggered speculation.

Revizor found a case where such race condition causes a side-channel leakage, shown in Figure 3.8 (the complete test case is in Appendix C.5). Here, if the variable-latency operation (line 1) is faster than the speculation window of the branch (line 2), the dependent memory access (line 3) will leave a cache trace. Otherwise, the speculation will be squashed before the operation completes, and the memory access will not happen. As such, the hardware traces expose not only the memory access address, but also the latency of the operation at line 1. It means this variant provides more information to the attacker than the original Spectre V1.

Revizor detected a similar variant of Spectre V4, and we expect it to be applicable to other speculative vulnerabilities as well.

Speculative Store Eviction. In addition to our main fuzzing campaign, we tested the behavior of speculative stores. Several defence proposals (STT [277], KLEESpectre [251]) assumed that stores do not modify the cache state until they retire. To test it, we modified *CT-COND* to capture this assumption (see Appendix B.5), and fuzzed our CPUs against it. Revizor did not discover violations in Skylake, but quickly generated a counterexample on CoffeeLake (Appendix C.6). Under investigation, it appeared to be almost identical to Spectre V1, except that the leakage was caused by a store instruction. This is an evidence that the assumption was wrong, although we do not know the exact microarchitectural technique behind it.

3.7.4 Runtime Parameters

We next detail the runtime parameters of the fuzzing campaign in §3.7.1 for those configurations that did not detect violations (Table 3.4). Revizor was able to reach a significant speed of fuzzing, testing several thousand test cases within 24 hours, each with several hundreds of inputs. The average number of inputs per test case varies because, in the feedback-driven mode, Revizor increases this number together with the test case size based on coverage, and the dynamics of coverage varies slightly from campaign to campaign. Column *Pattern Coverage* shows that the test cases were diverse and Revizor tested millions of pattern combinations within the campaigns.

Contract	Test Cases	Inputs / Test Case	Effective Classes / Test Case	Pattern Coverage
CT-SEQ	5125	493	45 (13.4%)	113’707k
CT-BPAS	5095	480	43 (12.9%)	107’671k
CT-COND-BPAS	5249	477	42 (12.8%)	89’165k

Table 3.4: Runtime parameters of a 24-hour fuzzing campaign. *Test Cases* is the number of tested test cases. *Inputs/Test Case* is the average number of inputs per test case. *Effective Classes/Test Case* is the average number of effective input classes per test case, and in parentheses is the portion of effective classes among all classes. *Pattern Coverage* is the number of covered pattern combinations.

```
if (...)
  b = array[a];
```

(a) *CT-SEQ* violation

```
b = array1[a];
if (...)
  c = array2[b];
```

(b) *CTR-SEQ* violation

```
if (...)
  b = array1[a];
  c = array2[b];
```

(c) *ARCH-SEQ* violation

Figure 3.9: Difference in violations among attacker models.

Column *Effective Classes* illustrates the limitations of completely random input generation. The majority of inputs were ineffective (CH2), and were essentially a wasted effort. This effect worsens with the contract complexity as speculation further reduces the likelihood of forming effective classes. It indicates that future research on more sophisticated input generation may significantly improve the fuzzing speed.

3.7.5 Case Study: Comparing Contract Violations

In this case study, we compare the counterexamples that Revizor surfaces when fuzzing against different contracts. Specifically, we consider *CT-SEQ*, *CTR-SEQ* and *ARCH-SEQ*, which represent different observation models with increasingly powerful attacker, under the same (sequential) execution model.

In the experiment, we first fuzz the CPU against *CT-SEQ* and collect its violations. We then evaluate these counterexamples against *CTR-SEQ* which additionally exposes register values, and identify those that violate *CT-SEQ* but not *CTR-SEQ*. Last, we evaluate the counterexamples that violate *CTR-SEQ* but not *ARCH-SEQ*.

The violations of only *CT-SEQ* (Figure 3.9a and Appendix C.1) speculatively leak values stored in registers as *CT-SEQ* (implicitly) prescribes that registers might contain secrets. The violations of only *CTR-SEQ* (Figure 3.9b and Appendix C.7) load the secret with a non-speculative memory access, and leak it with a speculative one; Such examples are particularly interesting in the context of defences against V1, as some existing proposals do not protect against this variant (e.g., STT [277]). Finally, the violations of *ARCH-SEQ* (Figure 3.9c and Appendix C.8) expose speculatively-loaded values, and they correspond to the classic Spectre V1 example.

3.8 Discussion

3.8.1 Scope and Limitations

False contract violations (false positives). If the model incorrectly emulates ISA, it leads to false positives (although these cases are also valid bugs, but in the emulator instead the CPU). Due to this reason, we excluded from the tests some instructions that are not implemented correctly in Unicorn. Non-determinism in the executor may also cause false positives. However, we inspected a few counterexamples in each of the experiments described in §3.7 and found no false positives.

False contract conformance (false negatives). In several tests, Revizor did not detect violations (Table 3.1). This indicates, but *does not prove*, the absence of leaks: it merely shows that the explored space contained no counterexamples. Deeper testing is an open question left to future work.

Revizor may not detect a violation also because the information leakage does not surface in the cache side channel. This will not prevent detection of the currently-known speculative vulnerabilities because they all can be observed through a cache side channel. However, it may not hold in the future, for the currently-unknown vulnerabilities.

Coverage. In a few setups, Revizor timed out without detecting a violation (Table 3.1). As our notion of coverage is only approximate, this indicates, but *does not prove*, the absence of leaks: it merely shows that the explored search space did not contain counterexamples. We leave a more precise notion of coverage to future work.

Feedback. Currently, the fuzzing process is completely random and unguided. This is, however, not a principled limitation: Future work may improve the search for counterexamples by incorporating the feedback from performance counters, by relying on a grey-box model of the CPU, or by employing debugging interfaces.

Dependency patterns. We used control and data hazards as a proxy for speculative execution. However, a hazard is a necessary, but not sufficient condition to trigger speculation, and a certain microarchitectural state is normally required as well. For example, to trigger a branch misprediction, Branch Target Buffer must contain a wrong target. In addition, the speculation must be long enough to leave a microarchitectural trace. These preconditions are hard to control on commercial CPUs and, thus, we have to rely on execution of random test cases to indirectly randomize the microarchitectural state.

Input generation. An unavoidable consequence of random input generation is low input effectiveness (CH2). To tackle it, Revizor applies several restrictions, such as limiting the number of registers, aligning accesses, and reducing the input entropy. These restrictions could prevent Revizor from detecting unknown vulnerabilities. Future work may develop a more targeted generation method that would ensure all inputs are effective (e.g., via program analysis, similar to Spectector [112]), which would eliminate the need for restrictions.

Other attacks. Revizor currently supports only attacks on L1D caches. For other side-channels, we have to implement them within the executor (e.g., execution port attacks require reading of the port load). For certain speculative attacks, the executor would have to be modified (e.g., Meltdown requires handling of page faults). It may also require re-implementation of the executor in user space or in SGX enclaves.

3.8.2 Applications beyond Black-Box Fuzzing

Revizor can be extended, with very few modifications, to white-box fuzzing of open-source CPUs such as RISC-V. For this, one needs to implement a new executor that runs on a CPU simulator instead of a silicon CPU. White-box fuzzing will also enable us to collect more precise coverage and better guide the fuzzing.

Likewise, our approach can be applied to check the security claims of secure speculation mechanisms, e.g [263, 277, 206]. This would again require a simulator-based executor (e.g., based on gem5 [46]) with the respective mechanism, modified to collect hardware traces.

Beyond that, Revizor can be applied for reverse-engineering CPU internals as it can surface unknown CPU behavior (e.g., Revizor detected speculative store eviction, §3.7.3).

We presented Model-based Relational Fuzzing (MRF), a technique to test silicon black-box CPUs for speculative vulnerabilities. We implemented MRF in a fuzzing framework called Revizor, and used it to test recent Intel x86 CPUs for compliance with a wide range of speculation contracts.

Our experiments show that Revizor can automatically find contract violations, together with code snippets to trigger and reproduce them—without reporting false positives. Revizor also successfully detected a flaw in a hardware defence against MDS (namely, LVI-Null), and it found novel variants of V1 and V4. This demonstrates that MRF is a promising approach for third-party assessment of microarchitectural security in black-box CPUs.

Chapter 4

SpecFuzz: Fuzzing Software for Speculative Vulnerabilities

In this chapter, our goal is to develop a method to *find instances of speculative vulnerabilities in software.*

We focus only on speculative vulnerabilities because similar tools for detecting instances of traditional side channels already exist (see §2.6.3). Several works attempted to solve the problem of detecting speculative vulnerabilities too [173, 66, 126], but they relied on static analysis, which restricted their precision. They analyse the application code in search of *gadgets*—the code patterns typical for the attacks. However, the analysis is imprecise and may overlook vulnerabilities, either because the vulnerable code does not match the expected patterns [141], or due to the limitations of the analysis itself (e.g., considers each function only in isolation).

We seek to find a method of detecting instances of speculative vulnerabilities without restricting ourselves to specific gadgets. Instead, we harness a *dynamic* testing technique, namely fuzzing.

The main challenge is that speculative vulnerabilities are inherently invisible to conventional dynamic testing tools. For example, Spectre-type attacks are closely related to memory safety violations, such as buffer overflows, and differ only in their speculative components. Dynamic testing tools catch memory safety violations by verifying the validity of every memory access. Unfortunately, this is not possible for speculative violations, because the accesses are invoked *speculatively, on mispredicted paths*, therefore are discarded by hardware without being exposed to software. As a result, they remain invisible to runtime integrity checkers. It remains an open challenge to make speculative vulnerabilities visible to dynamic testing tools.

4.1 System Overview

We introduce *speculation exposure*, a method to enable dynamic testing for speculative vulnerabilities. Speculation exposure leverages *software* simulation of speculative execution to turn speculative vulnerabilities into conventional ones and, thus, make them detectable by memory safety checkers.

Speculation exposure consists of four phases executed for every speculatable instruction: ① take a checkpoint of the process state, ② simulate a misprediction, ③ execute the speculative path, and ④ rollback the process to the checkpoint and continue normal execution. This way, we temporarily redirect the normal application flow into the speculative path so that all invalid memory accesses on it become visible to software. This method simulates the worst-case scenario by examining each possible mispredicted path, without making assumptions about the way the underlying hardware decides whether to speculate or not.

We further extend speculation exposure to *nested speculation*, which occurs when a CPU begins a new speculation before resolving the previous one. To simulate it, for each speculatable instruction, we dynamically generate a *tree* of all possible speculative paths starting from this instruction and branching on every next speculatable instruction. The complete nested simulation, however, has proven to be too slow. To make fuzzing practical we develop a heuristic that prioritizes traversal of the speculation sub-trees with a high likelihood of detecting new vulnerabilities.

To showcase the method, we implement SpecFuzz, a tool for detecting Bounds Check Bypass (BCB) vulnerabilities. We picked BCB as the main target for two reasons: First, Spectre-type vulnerabilities are not expected to be fixed in hardware [70], and the burden of defence lies entirely on software methods [167]. Second, the conservative protection techniques against BCB [59, 128, 156, 225] are impractical because they pessimistically harden every conditional branch, and thus significantly hurt program performance (see §4.5.3).

SpecFuzz simulates conditional jump mispredictions by placing an additional jump with an inverted condition before every conditional jump. During the simulation it executes the inverted jump and then rolls back to return to the original control flow. To detect invalid accesses on the simulated speculative path, SpecFuzz relies on AddressSanitizer [218].

SpecFuzz may serve as a tool for both offensive and defensive security. For the former (e.g., penetration testing), it finds vulnerabilities in software, records their parameters, and generates test cases. For the latter, the fuzzing results are passed to automated hardening tools (e.g., Speculative Load Hardening [59]) to elide unnecessary instrumentation of the instructions deemed safe. Note that the code not covered by fuzzing remains instrumented conservatively, hence lower fuzzing coverage might affect performance but not security.

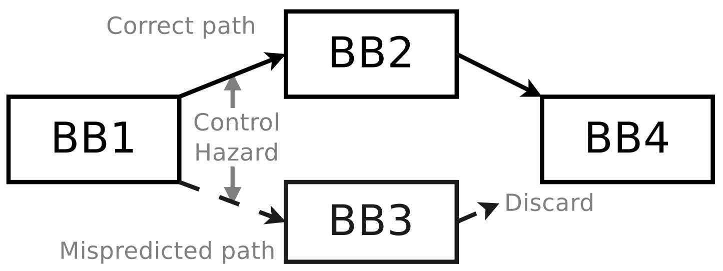


Figure 4.1: Speculative execution. Due to a misprediction, the program executes basic blocks BB3 and BB4, then detects the mistake, discards the results, and continues execution starting from BB2.

4.2 Method: Speculation Exposure

Speculative vulnerabilities are notoriously hard to find because hardware strives to hide the effects of speculative execution from software, making it impossible to detect such vulnerabilities with conventional testing methods. In this chapter, we approach the problem by simulating the unsafe hardware optimization in software. We call this approach *speculation exposure*.

To understand how we construct the simulation, first consider how speculative execution (§2.4.1) is implemented in hardware. When a hazard appears (e.g., at a conditional or an indirect jump), the CPU ① makes a prediction of its outcome, ② executes the speculative path while temporarily keeping the results in internal buffers, ③ eventually eliminates the hazard and either commits the results (correct prediction) or discards them (wrong prediction), and ④ proceeds with the correct path.

Example: In Figure 4.1, the CPU might make a wrong prediction that BB1 (Basic Block 1) will proceed into BB3. It will start executing BB3, BB4, and maybe even further, depending on how long it takes to resolve the hazard. When the hazard is resolved, the CPU determines that the prediction was wrong and discards all changes made on the speculative path. Afterward, it redirects the control flow to the correct path and proceeds with the execution starting from BB2.

The core idea behind speculation exposure is to simulate this behavior in software with a *checkpoint-mispredict-rollback* scheme: At a potential hazard, we ① take a checkpoint of the current process state. Then, we ② diverge the control flow into a wrong (mispredicted) path and start executing it. When a termination condition is reached (e.g., a serializing instruction is executed), we ③ rollback to the checkpoint and ④ proceed with normal execution. The pattern can be applied to data hazards too: Instead of diverging the control flow, we would replace a memory/register value with a mispredicted one.

This basic mechanism simulates the worst-case scenario when a CPU always mispredicts and always speculates to the greatest possible depth. Such a pessimistic approach makes the testing results universally applicable to different CPU models and any execution conditions. Moreover, it also covers all possible combinations of correct and incorrect predictions that could happen at runtime (see §4.2.2).

4.2.1 Components of Speculation Exposure

There are four core components: a checkpointing mechanism, a simulation of mispredictions, a detection of faults on the simulated path, and a mechanism for detecting termination conditions.

Checkpointing. For storing the process state, we could use any of the existing checkpointing mechanisms, ranging from full-process checkpoint (e.g., CRIU [80]) to transactional memory techniques (e.g., Intel TSX [130]). However, checkpointing is on the critical path in our case, thus heavy-weight mechanism would either increase the testing time, or reduce the number of inputs used in fuzzing under a fixed time budget. We describe the checkpointing mechanism used in our implementation in §4.3.1.

Simulating Misprediction. To simulate misprediction, we instrument basic blocks in a way that forces control flow to enter the paths that the CPU would otherwise take speculatively. The nature of the instrumentation depends on the exact type of the speculative execution attack being simulated (see §4.3 and §4.6.1 for a detailed discussion about applying this technique to different Spectre attacks).

Detection of Vulnerabilities. In Spectre-type attacks, the data is leaked when a program speculatively reads from or writes to a wrong object. Therefore, when we have a mechanism for simulating speculative execution, the detection of actual vulnerabilities boils down to the conventional memory safety problem; detecting bounds violations. This is a well-developed field with many existing solutions [177, 130, 218]. In this work, we rely on AddressSanitizer [218].

Terminating Simulation. The simulation mimics the termination of the speculative execution by hardware. Speculative execution terminates: *(i)* upon certain *serializing* instructions (e.g., `LFENCE`, `CPUID`, `SYSCALL`, as listed in the CPU documentation [130]), and *(ii)* after the speculation exhausts certain hardware resources.

Note that terminating the simulation earlier results in faster fuzzing and could be used as an optimization, but it could miss vulnerabilities. Below we discuss the hardware resources used in speculation to determine the simulation termination conditions.

Termination conditions. All program state changes made during the speculative execution must be temporarily stored in internal hardware buffers, so that they can be reverted if the prediction is incorrect. Accordingly, once at least one of these buffers becomes full the speculation stops.

On modern Intel CPUs, there are several buffers that can be exhausted [130]: Reorder Buffer (ROB), Branch Order Buffer (BOB), Load Buffer (LB), Store Buffer (SB), Reservation Station (RS), Load Matrix (LM), and Physical Register Reclaim Table (PRRT). We seek to find the one that overflows first.

LM and PRRT are not documented by Intel. LB, SB, and RS are also not useful for practical simulations as their entries could be reclaimed dynamically (policy is undocumented) during speculative execution. Therefore, we do not simulate these buffers and assume that they do not restrict the depth of the speculation. Correctly simulating them would require an often-undocumented information about the reclamation algorithms in Intel CPUs.

We are left with ROB, which keeps track of all speculative microoperations (μops), and BOB, which tracks unresolved branch predictions. We choose ROB because BOB is not portable as it is a specific optimization of Intel CPUs [49].

In Intel x86, any speculative path can contain at most as many μops as there are entries in ROB[1]. In modern CPUs, its size is under 250 μops (the largest we know is 224 entries, on Intel Skylake architecture [69]).

The simulation terminates after reaching 250 instructions, which is a conservative estimate because one instruction is typically mapped into one or more μops. The only exception is μops fusion, when CPU merges several instructions into one. However, on Intel CPUs, it is limited to a small set of instruction combinations [69]. To account for this effect, we count these combinations as a single instruction.

Note that a tighter bound on the number of speculated instructions (e.g., through simulation of a smaller buffer) could have improved the fuzzing time without affecting correctness.

4.2.2 Nested Speculation Exposure

The CPU may perform *nested* speculation; that is, it can make a prediction while already executing a speculative path. Since we do not make any assumptions about the predictions, every speculatable instruction triggers not a single simulation, but a series of *nested simulations*. We refer to a tree of all possible speculative paths as a *simulation tree*. A simulation tree for each speculatable instruction is regenerated for each program input.

[1]Some CPU architectures (e.g., CPR [21]) could speculate beyond the ROB size. However, to the best of our knowledge, that is not the case for the existing x86 CPUs

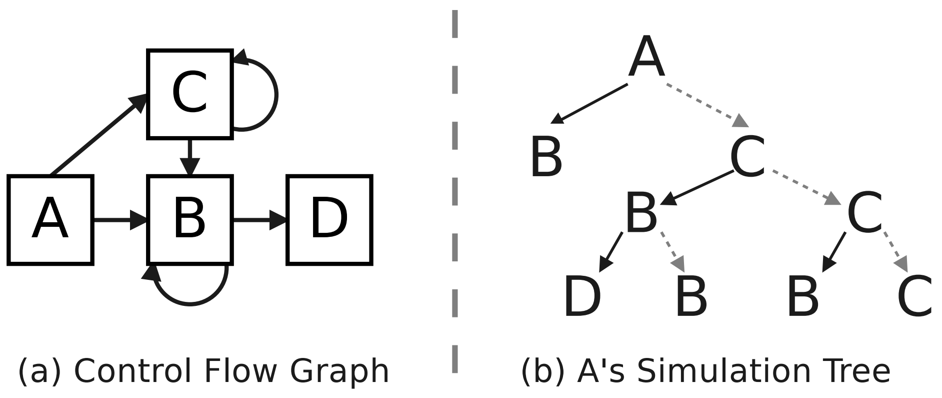


Figure 4.2: Nested speculation exposure for the flow A→B→D. Dashed lines are mispredicted speculative paths.

Instead of traversing the complete simulation tree (*complete simulation*), we could simulate only a subset of all mispredictions. Then, an *order of a simulation* is the maximum number of nested mispredictions it simulates. In other words, an order is the maximum depth of the simulation tree. Accordingly, an *order of a vulnerability* is defined as the minimum order of a simulation that triggers this vulnerability. An *order of a speculative path* is the number of mispredictions required to enter it.

Example: Consider Figure 4.2. The left side (Figure 4.2a) is a control-flow graph. Suppose that the correct flow is `ABD`.

If we simulate branch mispredictions, then the simulation tree of branch `A` would be as shown in Figure 4.2b. The simulation of order 1 for that branch traverses only the path `(ACBD)`, simulating only the first misprediction, and then following the original flow graph. The simulation of order 3 would traverse three additional paths: `ACBB`, `ACCB` and `ACCC`, according to misspeculation of `A` and `B`; `A` and `C`; and `A`, `C`, `C` respectively. The four paths constitute a complete simulation tree of the branch `A`. Every branch (or, more generally, every speculatable instruction) has its own simulation tree and the tree has to be traversed every time the branch is executed.

Nested simulation dramatically increases the fuzzing time. However, in SpecFuzz we use a heuristic which, while traversing only a small portion of the speculation tree on each input, shows high detection rates. We discuss it in detail in §4.3.2.

4.3 Design

To showcase speculative exposure on a specific class of vulnerabilities, we develop SpecFuzz, a tool for simulating and detecting Bounds Check Bypass (BCB) [142]. We discuss other Spectre-type attacks in §4.6.1.

```
1
2
3
4 if (x < array_size)
5
6     int result = array[x];
7     ...
8
9
```

(a) Native version

```
1 checkpoint();
2 if (x >= array_size)
3     goto skip_branch;
4 if (x < array_size)
5 skip_branch:
6     int result = array[x];
7     ...
8 if (terminate_simulation())
9     rollback(); // to line 4
```

(b) Simulation of conditional branch misprediction

Figure 4.3: SpecFuzz instrumentation.

As described in §2.4.1, BCB in its core contains a speculative out-of-bounds access caused by a conditional jump misprediction. To expose such accesses, we create a modified (instrumented) version of the application which executes not only the normal control flow but also enters all possible speculative paths.

SpecFuzz works as follows (Figure 4.3): Before every conditional branch (line 4), it inserts a call to a checkpointing function (line 1) that stores the process state and initializes simulation. Then, it adds an instruction sequence that simulates a misprediction (lines 2–3) and forces the control flow into the mispredicted path. Specifically, SpecFuzz inserts a jump with an inverted condition (line 2), followed by a jump into the basic block, thus skipping the original branch (line 3). During the simulation, SpecFuzz periodically checks if a termination condition has appeared (line 8). If the check passes, SpecFuzz restores the process state from the previous checkpoint (line 9) and continues the program execution.

We implement this design as a combination of an LLVM [153] compiler backend pass for the x86 architecture and a runtime library.

4.3.1 Basic Simulation

Simulating Branch Misprediction. SpecFuzz simulates mispredictions by forcing the application into taking a wrong branch at every conditional jump. We implement this behavior by replacing all conditional terminators in the program with the ones that have an inverted condition (see Figure 4.4). Now, when the original basic block (BB) would proceed into the successor $S1$, the modified terminator diverges the control flow into $S2$. The original terminator is moved into a separate BB, and the control flow returns to normal execution by rolling back into this BB after the simulation.

As a result, every time the program reaches this BB, it first executes the simulated path, then rolls back to the BB and continues with normal execution.

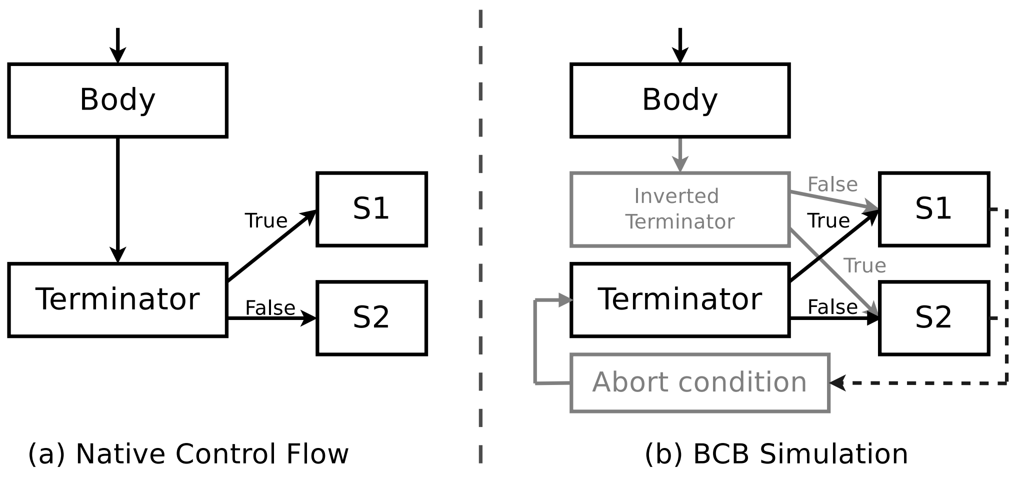


Figure 4.4: Simulation of conditional branch mispredictions: On simulated speculative paths, all conditional terminators are replaced by terminators with inverse conditions.

Saving and Restoring Process State. The main requirement to the rollback mechanism used in SpecFuzz was to have low performance impact so that the fuzzing time is kept short. To this end, we implement a light-weight in-process mechanism that snapshots the CPU state before starting a simulation and records the memory changes during the simulation.

To store the CPU state, we add a call to a checkpointing function (a part of the runtime library) before every conditional jump. The function takes a snapshot of the register values (including GPRs, flags, SIMD, floating-point registers, etc.) and stores it into memory. During the rollback, we restore the register values based on the snapshot. The function also stores the address of the original conditional jump (i.e., original terminator) that we later use as a rollback address.

This approach, however, is not efficient when applied to saving the memory state because it would require dumping the memory contents into the disk at every conditional jump. To avoid the performance overhead linked with this expensive operation, we instead rely on logging the memory changes that happen during the simulation. Before every instruction that modifies memory (e.g., `mov`, `push`, `call`), we store the address it modifies and its previous value onto a stack-like data structure. Then, to do a rollback, we go through this data structure in the reverse order and restore the previous memory values.

Currently, SpecFuzz supports only fixed-width writes; If the pass encounters `REP MOV`, compilation fails with an error. Yet, we did not encounter any issues with that during our experiments because Clang in its default configuration does not use these instructions.

Detecting and Handling Errors. With the simulation mechanism at hand, we now need a mechanism to detect any speculative out-of-bounds accesses that happen on the speculative paths. To this end, we utilize AddressSanitizer [218][2] (ASan), and we use a custom signal handler to handle the errors that inevitably appear during the simulations.

We had to modify the behavior of ASan to our needs. In contrast to normal, non-speculative execution, the process does not crash if an error happens during the speculation. Instead, the CPU silences the error by discarding its effects when the misprediction is detected. To simulate this behavior in SpecFuzz, we adjusted the error response mechanism in ASan to record the violation in a log and continue the simulation. Accordingly, one test run might detect several (sometimes, hundreds of) violations.

Similarly, we have to recover from runtime faults. We register a custom signal handler that logs and rolls back after the signals that could be caused by an out-of-bounds access, such as `SIGSEGV` and `SIGBUS`. We also rollback after other faults (e.g., division by zero), but we do not record them in the log as they are irrelevant to the BCB vulnerability. We perform an immediate rollback because hardware exceptions are supposed to terminate speculative execution. Even though on some CPU models exceptions may not terminate speculation (see Meltdown-type attacks [158, 58]), we ignore such cases assuming they will be fixed at the hardware level similarly to Meltdown.

Terminating Simulation. As discussed in §4.2, we terminate the simulation either when we encounter a serializing instruction or when the maximum depth of speculation is reached.

To implement the first case, we simply insert a call to the rollback function before every serializing instruction. As serializing, we consider the instructions listed as such in the Intel documentation [130] (e.g., `LFENCE`, `CPUID`, `SYSCALL`).

To count instructions at runtime, we keep a global instruction counter and set it to zero when a simulation begins. At the beginning of every basic block, we add its length to the counter. (We know the length at compile time because SpecFuzz is a backend pass). When the counter reaches 250 (maximum ROB size, see §4.2), we invoke the rollback function.

4.3.2 Nested Simulation

To implement nested simulation, we maintain a stack of checkpoints: Every time we encounter a conditional branch, we push the checkpoint on the stack, as well as the current value of the instruction counter and a pointer to the previous stack frame. All later writes will be logged into the new stack frame. At rollback, we restore the topmost checkpoint and revoke the corresponding memory changes. This way, SpecFuzz traverses all possible combinations of correct and incorrect predictions in the depth-first fashion.

[2]The inherent limitation of AddressSanitizer is that it cannot detect intra-object out-of-bounds accesses; hence we consider them out of scope.

Order	JSMN	Brotli	HTTP	libHTP	libYAML	OpenSSL
1	6	74	6	221	77	1254
2	5	9	4	64	92	366
3	7	12	2	33	14	253
4	1	6	3	5	16	91
5	1	2	1	2	6	-
6	0	0	0	2	2	-
Total	20	103	16	327	207	1964
Iterations	933	3252	1582	540	1040	227

Table 4.1: Distribution (by order) of the vulnerabilities detected by 24 hours of fuzzing non-prioritized 6th-order simulation. This experiment motivates prioritized simulation: Even though all fuzzing rounds simulated all 6 orders of misprediction, most of the detected vulnerabilities required only a few mispredictions. Since execution of OpenSSL was too slow, we simulated it only to the 4th order.

Coverage Trade-off. The number of paths to traverse increases exponentially with the order of the simulation. In most programs, the density of conditional branches is approximately one in ten instructions. If we assume the maximum depth of speculative execution to be 250 instructions, then it creates over 30 million speculative paths on average per conditional branch. Often the actual number of paths is smaller because the tree is not balanced, or because the tree is shallow due to serializing instructions (e.g., system calls), however the costs are still high, slowing down the fuzzing driver by orders of magnitude. It could be acceptable for very small fuzzing drivers (e.g., when fuzzing a single function), but not for larger libraries.

The trade-off between the fuzzing speed and the completeness of nested simulation is a non-trivial one. In particular, it is not clear to what extent added depth of the simulation improves the detection of speculative vulnerabilities compared to the loss in input coverage.

To estimate the effectiveness of deeper simulation we compiled our test libraries (see §4.5) with SpecFuzz configured for a 6th-order simulation and fuzzed them for 24 hours. Table 4.1 contains a breakdown of the vulnerabilities we detected by their order. Clearly, the bulk of the vulnerabilities is detected with only a few levels of nesting, and the higher the order the fewer vulnerabilities we find[3].

A plausible explanation of this result is as follows. Most memory accesses are guarded by only one safety check (e.g., a bounds check) which we would need to bypass speculatively (first order vulnerabilities). More rarely, the bounds checks would be duplicated across

[3]The real distribution is even more contrasting. Here, the 6th-order simulation caused a high overhead and few iterations were executed (Table 4.1). Therefore, the fuzzer could not generate the inputs to trigger the vulnerabilities with fewer mispredictions. In fact, in §4.5.2, many of these vulnerabilities were discovered by lower-order simulations with more iterations.

functions or, for example, accompanied by an object type check; In this case, detecting such a vulnerability would require two mispredictions (second order). Higher order vulnerabilities usually require the speculative path to cross several function boundaries.

We can conclude that the speed of fuzzing is a higher priority than the order of simulation. Most of the vulnerabilities have low orders and we are likely to find more vulnerabilities if we have many iterations of low-order simulation compared to running few iterations of high-order simulation. In fact, in our later experiments (§4.5.2), SpecFuzz detected more vulnerabilities within an hour of low-order fuzzing compared to 24 hours with a 6th order simulation.

Prioritized Simulation. Based on this observation, we propose the following fuzzing heuristic. Our *prioritized simulation* tests the low-order paths more rigorously, allocating less time to higher-order paths.

A simple approach would be to always run the simulation at a baseline order and once every N iterations run a higher-order simulation. For example, all runs simulate order 1, every 4th run simulates up to order 2, every 16th up to order 3, and so on.

However, since not all runs invoke all the branches, the distribution would be uneven. Instead, we should calculate the shares per branch.

Suppose we have only two branches—X and Y—in the program under test, and we test the program with six inputs. X is executed in every run, but Y is invoked only in the runs 1, 2, 3, and 5. With the prioritized simulation, we simulate only the first-order paths of the branch X in the runs (1, 2, 3, 5, 6) and both the first and the second order paths in the run 4. As of the branch Y, we simulate the first order in runs (1, 2, 3) and up to the second order in the run 5.

We implemented this strategy in SpecFuzz and used it in our evaluation.

Simulation Coverage. Because prioritized simulation begins by traversing only one speculative path in every simulation tree and only gradually enters more and more paths, it would be important to know which share of all possible speculative paths it managed to cover within a given fuzzing round. We call this metric a *simulation coverage*. This metric provides an estimate of the portion of the covered speculated paths out of all possible paths for all the branches.

The trade-off different simulation heuristics might explore is a trade-off between fuzzing coverage and simulation coverage. For example, prioritized simulation gives preference to the fuzzing coverage. Unfortunately, estimating the precise number of speculative paths for each branch is a complex problem because the trees are not balanced. Solving it would require detailed program analysis, which we leave to future work.

4.3.3 Other Implementation Details

External calls and indirect calls. By the virtue of being implemented as a compiler pass, SpecFuzz cannot correctly run the simulation beyond the instrumented code. Therefore, we have to consider all calls to external (non-instrumented) functions as serialization points, even though it is not necessarily a correct behavior (see §4.6).

As the complete list of program functions is not known at compile time, SpecFuzz first collects the functions with a dummy compilation, and only then does the full instrumentation. The collected function list can be reused for later compilations if the source does not change.

This approach, however, does not work for indirect calls as we do not know the call target at compile time. Instead, we have to detect the callee type at run time. To this end, SpecFuzz inserts a `NOP` instruction with a predefined argument into every function entry. Before indirect calls, it adds a sequence that fetches the first instructions and compares it with the opcode of this `NOP`. If they match, we know that the function is instrumented and it is safe to continue the simulation.

Callbacks. There could be a situation where a non-instrumented function calls an instrumented one (e.g., when a function pointer is passed as an argument). In this case, the instrumented function might return while executing a simulation and the simulation will enter the non-instrumented code, thus corrupting the process state. To avoid it, SpecFuzz globally disables simulation before calling external functions and re-enables it afterward. Accordingly, our current implementation does not support simulation in callbacks (see a potential solution to this problem in §4.6).

Long Basic Blocks. In the end of every basic block (BB), SpecFuzz checks if the speculation window has expired (i.e., if the instruction counter has reached 250). This could unnecessarily prolong the simulation when we encounter a long BB, which could be created, for example, by loop unrolling. To avoid this situation, SpecFuzz inserts additional checks every 50 instructions in the long BBs.

Preserving the Process State. When a function returns while executing a simulation, the value of the stack pointer becomes above its checkpointed value. Therefore, if we call a function from the SpecFuzz runtime library or from ASan, it would corrupt the checkpointed stack frame. This could be avoided by logging all changes that these functions do to the memory, but it would have a high performance cost. Instead, we use a disjoint stack frame for these functions and replace the stack pointer before calling them.

The same applies to the code that SpecFuzz compiler pass inserts: We had to ensure that the code that could be executed on a speculative pass never makes any changes to memory besides modifying dedicated variables of the SpecFuzz runtime.

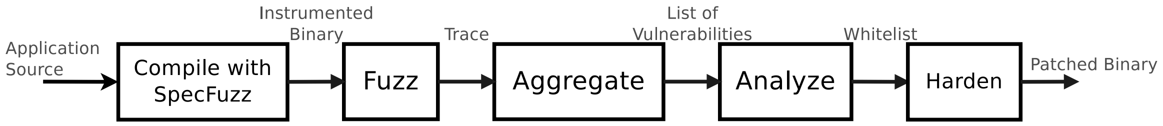


Figure 4.5: Workflow of testing an application with SpecFuzz.

Code pointer checks. Besides causing out-of-bounds accesses, misprediction of conditional branches may also change the program's control flow. This happens when a corrupted code pointer is dereferenced. For example, if speculative execution overwrites a return address or the stack pointer, the program can speculatively return into a wrong function or even attempt to execute a data object. This vulnerability type is especially dangerous as it may allow to launch a ROP-like attack [220]. To detect such corruptions, SpecFuzz checks return and indirect jump addresses to verify that they point to the code section of the binary.

4.4 Fuzzing with SpecFuzz

The workflow is depicted in Figure 4.5.

1. Compile the software under test with Clang and apply the SpecFuzz pass (§4.3), thus producing an instrumented binary that simulates branch mispredictions.
2. Fuzz the binary. We used HonggFuzz [100], an evolutionary coverage-driven fuzzer, and we relied on a combination of custom coverage tracking and Intel Processor Trace [133] for measuring coverage.
3. Aggregate the traces and analyze the detected vulnerabilities to produce a *whitelist* of conditional jumps that were deemed safe by our analysis.
4. Patch the application with a pass that hardens all but the whitelisted jumps.

We now describe these stages in detail.

4.4.1 Coverage and Fuzzing Feedback

Using existing coverage estimation techniques (e.g., SanitizerCoverage [162], Intel PT [133]) with SpecFuzz is incorrect: the values become artificially inflated because SpecFuzz adds the speculative paths that do not belong to normal program execution.

Instead, we implement a custom coverage mechanism that tracks the conditional branches executed during fuzzing, but excludes the branches executed while simulating mispredictions and when the simulation is globally disabled (i.e., in callbacks). We implement the mechanism through a hashmap that tracks the executed branches as well as the number of unique inputs that triggered every branch. In addition to coverage, this map is also used for prioritized simulation (§4.3.2).

We also maintain a hashmap of vulnerabilities as an additional feedback source for evolutionary fuzzing. This way, every time we detect a new vulnerability, HonggFuzz stores the input that triggered it and adds it to the corpus. It provides a better feedback to the fuzzer and also allows us to preserve the test cases that trigger specific vulnerabilities.

4.4.2 Aggregation of Results

As a result of fuzzing, we get a trace of detected speculative out-of-bounds accesses. Each entry in the trace has a form:

```
    (accessed_address; offset; offending_instruction; mispredicted_branches)
```

Here, `offending_instruction` is an address of the instruction that tried to access a memory outside the intended object's bounds (`accessed_address`), and `mispredicted_branches` are the addresses of the mispredicted branches which triggered the access. `offset` is the distance to the nearest valid object, if we found one.

To make the trace usable, we aggregate the results per run and per instruction. That is, for every test run, we collect all the addresses that every unique offending instruction accessed as well as the addresses of the mispredicted branches.

4.4.3 Vulnerability Analysis

After the aggregation, we have a list of out-of-bounds accesses with an approximate range of accessed addresses for each of them. As we will see in §4.5.2, the list may be rather verbose and contain up to thousands of entries. Yet we argue that most of them are not realistically exploitable.

In many cases, the violation occurs as a result of accessing an address that remains constant regardless of the program input. Therefore, the attacker cannot control the accessed address, and cannot leak secrets located in other parts of the application memory. This could happen, for example, when the application tries to speculatively dereference a field of an uninitialized structure. In this case, the attacker would be able to leak values from only one address, which is normally not useful unless the desired secret information happens to be located at this address[4]. We call such vulnerabilities *uncontrolled*.

We identify the uncontrolled vulnerabilities by analyzing the aggregated traces. We

[4]In this work, we do not consider this corner case and leave it to future work. Its identification would require more complex program analysis (e.g., taint analysis).

estimate the presence of the attacker's control by comparing the accessed addresses in every run (i.e., every new fuzzing input). If a given offending instruction always accesses the same set of addresses, we assume that the attacker does not have control over it. Note, however, that the heuristic is valid only after a large enough number of test runs (and even then, it may be prone to false positives, see §4.6.2).

After the analysis, we collect a list of safe conditional branches (*whitelist*). The safety criteria is defined by the user of SpecFuzz. In our experiments, the criteria were: *(i)* the branch was executed at least 100 times; *(ii)* it never triggered a non-benign vulnerability. The criteria for defining whether a vulnerability is benign could be controlled too. In our experiments, they were: *(i)* the vulnerability was triggered at least 100 times; *(ii)* the vulnerability is uncontrolled. In the future, additional criteria could be added to reduce the rate of false positives.

The resulting *whitelist* is a plaint-text file with a list of corresponding code locations, which we get based on accompanying DWARF debugging symbols.

4.4.4 Patching

Finally, we pass the whitelist created at the analysis stage to a tool that would harden those parts of the application that are not in the list. We opted for this approach (in contrast to directly patching the detected vulnerabilities) because it ensures that we do not leave the non-tested parts of the application vulnerable.

In our experiments, we used two hardening techniques: adding serializing instructions (`LFENCE`s) and adding data dependencies (SLH [59]).

LFENCE Pass. The simplest method of patching a BCB vulnerability is to add an `LFENCE`—a serializing instruction in Intel x86 architecture that prevents [130] speculation beyond it. Adding an `LFENCE` after a conditional branch ensures that the speculative out-of-bounds access will not happen. We used an LLVM pass (shipped as a part of SLH) that instruments all conditional branches with this technique and modified it to accept the whitelist.

Speculative Load Hardening (SLH). An alternative mechanism is to introduce a data dependency between a conditional branch and the memory accesses that follow it. This mechanism is implemented in another LLVM pass called SLH. We similarly modified the pass to accept the whitelist.

4.4.5 Investigating Vulnerabilities

Often, it is necessary to go beyond automated analysis and investigate the vulnerabilities manually. For example, this may be required for penetration testing, for weeding out false positives, or for creating minimal patches where the performance cost of automated instrumentation is not acceptable.

To facilitate the analysis, SpecFuzz reports all the information gathered during fuzzing. For vulnerabilities, this information includes: all accessed invalid addresses and their distance to nearby valid objects (when available); all sequences of mispredicted branches that triggered the vulnerability; the order (i.e., the minimal number of mispredictions that can trigger it); the code location of the fault (based on debug symbols); whether different inputs triggered accesses to different addresses (controllability); the execution count. For branches, the SpecFuzz reports: which vulnerabilities this branch can trigger; the code location of the branch; its execution count (how many unique inputs covered this branch).

SpecFuzz also stores the inputs that triggered the vulnerabilities, which could later be used as test cases.

Finally, when the gathered information is not sufficient, SpecFuzz can instrument a subset of branches instead of the whole application. This way, we can quickly re-fuzz the locations of interest because such targeted simulation normally runs at close-to-native speed.

4.5 Evaluation

In this section, we focus on the following questions:

- How effective is SpecFuzz at detecting BCB?
- How many vulnerabilities does it find compared to the existing static analysis tools?
- How much performance does SpecFuzz recover over conservative instrumentation of all the branches?

Applications. We use SpecFuzz to examine six popular libraries: a cryptographic library (OpenSSL [1] v3.0.0, `server` driver), a compression algorithm (Brotli [4] v1.0.7), and four parsing libraries, JSON (JSMN [5] v1.1.0), HTTP [8] (v2.9.2), libHTP [6] (v0.5.30), and libYAML [7] (v0.2.2). We chose them because they directly process unsanitized input from the network, potentially giving an attacker the opportunity to control memory accesses within the libraries, which together with BCB enables random read access to victim's memory by the attacker.

MSVC	RH Scanner	Spectector	SpecFuzz	**Total**
7	12	15	15	**15**

Table 4.2: BCB variants detected by different tools.

	JSMN	Brotli	HTTP	libHTP	libYAML	OpenSSL
Native	370	392	463	251	457	84
SpecFuzz	2.8	6.6	20.4	2.4	5	0.15

Table 4.3: Average number of fuzzing iterations executed by native version and by SpecFuzz simulation per hour, in thousands.

Other tools. To put the results into a context, we compare SpecFuzz against two existing mitigation and detection tools:

- RedHat Scanner [66]: Spectre V1 Scanner, a static analysis tool from RedHat.
- Respectre [126]: a static analysis tool from GRSecurity. Tested only on libHTP as we did not have a direct access to the tool.

As a baseline we use LFENCE instrumentation and Speculative Load Hardening (SLH) [59] (shipped with Clang 7.0.1) described in §4.4.4.

In §4.5.1, we additionally tested the `/Qspectre` pass of MSVC [173] (v19.23.28106.4) and a symbolic execution tool Spectector [112] (commit `839bec7`). Due to low effectiveness, we did not perform further experiments with MSVC. As of Spectector, we report results only for microbenchmarks because larger libraries (Brotli, HTTP, JSMN) exhibited large number of unsupported instructions.

Testbed. We use a 4-core (8 hyperthreads) Intel Core i7 3.4 GHz Skylake CPU, 32 KB L1 and 256 KB L2 private caches, 8 MB LLC, and 32 GB of RAM, running Linux kernel 4.16.

4.5.1 Detection of BCB Gadgets

We tested 15 BCB gadgets by Paul Kocher [141]. They were originally designed to illustrate the shortcomings of the BCB mitigation mechanism in MSVC [173]. While the suite is not exhaustive, this is a plausible microbenchmark for the basic detection capabilities.

Table 4.2 shows the results. SpecFuzz and Spectector expose all speculative vulnerabilities. MSVC and RedHat Scanner rely on pattern matching and overlook a few cases.

4.5.2 Fuzzing Results

To see how effective SpecFuzz is at detecting vulnerabilities in the wild, we instrumented the libraries with SpecFuzz configured for prioritized simulation (§4.3.2) and fuzzed them

	JSMN	Brotli	HTTP	libHTP	libYAML	OpenSSL
Native	96.6	84.1	64.1	60.6	63.9	24.0
SpecFuzz	96.6	84.1	63.5	60.6	63.3	24.0

Table 4.4: The highest reached coverage of the libraries. In percent, out of all branches.

for varying duration of time: 1, 2, 4, 8, 16, and 32 hours (63 hours in total). We used one machine and fuzzed on a single thread. Every next round used the input corpus generated by the previous ones. The initial input corpus was created by fuzzing the native versions of the libraries for an hour. Where available, we also added the test inputs shipped with the libraries.

Fuzzing iterations. Over the experiment, the average rate of fuzzing was as presented in Table 4.3. Compared to native, non-instrumented version, SpecFuzz is definitely much slower. Yet, the rate is still acceptable: For example, we managed to test over 400’000 inputs within 63 hours of fuzzing Brotli.

Coverage. The final coverage of the libraries is shown in Table 4.4. The presented numbers are branch coverages; that is, which portion of all branches in the libraries was tested during the fuzzing. We show only the final number (i.e., after 63 hours of fuzzing) because we started with an already extensive input corpus and the coverage was almost not changing across the experiments. The largest difference was in OpenSSL compiled with SpecFuzz, where after one hour the coverage was 22.9% and, in the end, it reached 24%.

The difference between the native and the SpecFuzz versions is caused by our handling of callbacks. As discussed in §4.3.3, we globally disable the simulation before calling non-instrumented functions. Hence, some parts of the application are left untested. However, it affects only performance, not security – the untested branches remain protected by exhaustive instrumentation.

Detected Vulnerabilities. The total numbers of detected vulnerabilities in each experiment is in Table 4.5. There is a vast difference between the results, ranging from thousands of violations detected in OpenSSL to only 16 found in the HTTP parser. The main factor is the code size: OpenSSL has ~330000 LoC while HTTP has fewer than 2000 LoC.

Vulnerability types. For most of the vulnerabilities, however, we did not observe any correlation between the input and the accessed address, which puts them into the category of uncontrolled vulnerabilities (see §4.4.3). The results of the analysis are in Table 4.6. Note that we marked the violations as uncontrolled only if they were triggered by at least 100 different inputs. Those under the threshold are in the row *unknown*. SpecFuzz also detected several cases where the vulnerability corrupted a code pointer (*code*).

Duration	JSMN	Brotli	HTTP	libHTP	libYAML	OpenSSL
1 hr	20	96	16	322	175	1940
2 hr	20	101	16	330	202	1997
4 hr	20	104	16	332	211	2060
8 hr	20	106	16	334	230	2104
16 hr	20	108	16	337	244	2139
32 hr	20	108	16	344	251	2155

Table 4.5: Total number of detected vulnerabilities in each experiment.

Type	JSMN	Brotli	HTTP	libHTP	libYAML	OpenSSL
code	0	2	1	2	3	16
controlled	16	68	9	91	140	589
uncontrolled	34	36	6	222	49	1127
unknown	0	4	0	29	59	423

Table 4.6: Breakdown of the detected vulnerabilities by type. Here, *code* are speculative corruptions of code pointers (e.g., of a return address) and the rest are corruptions of data pointers. *Cont.* are controlled vulnerabilities and *uncont.* are uncontrolled. *Unknown* are likely uncontrolled vulnerabilities, but they were triggered too few times (less than 100 times).

The row *checked* is the number of vulnerabilities that are left after we checked the results for common false positives (i.e., false labeling as controlled). The difference between the last two rows is caused by the fact that ASan does not provide precise information about the overflows and SpecFuzz sometimes falsely marks them as *controlled* (see §4.6). We did the checking by using a set of heuristics (implemented as scripts) that identified these cases, which we later verified manually. Overall, the manual analysis took 5 hours of work.

It is still possible that some of the less-typical false positives where not detected by SpecFuzz; unfortunately, we do not have a method to filter them out. However, with our automated patching approach (§4.4.4), the false labeling is acceptable as its only cost is higher overhead.

Vulnerability orders. Finally, Table 4.7 shows a distribution of the detected vulnerabilities by order. As we can see, prioritized simulation successfully managed to surface the vulnerabilities up to the 6th order.

4.5.3 Performance Impact

We used the whitelists produced in the previous experiment to patch the libraries with `LFENCE`s and with a modified version of Speculative Load Hardening (see §4.4.4). Specifi-

Order	JSMN	Brotli	HTTP	libHTP	libYAML	OpenSSL
1	6	79	6	232	97	1344
2	7	9	4	66	81	428
3	5	14	3	33	33	216
4	2	4	3	5	28	91
5	0	2	0	6	6	55
6	0	0	0	2	6	21
7	0	0	0	0	0	0

Table 4.7: Breakdown (by order) of the detected vulnerabilities.

Hardening Type	JSMN	Brotli	HTTP	libHTP	libYAML	OpenSSL
SLH (all)	65%	48%	44%	41%	26%	15%
SLH (c,100)	69%	49%	44%	50%	27%	16%
SLH (c,10)	69%	49%	44%	51%	37%	18%
LFENCE (all)	73%	50%	56%	43%	27%	16%
LFENCE (c,100)	77%	51%	56%	52%	28%	18%
LFENCE (c,10)	77%	51%	56%	53%	39%	20%

Table 4.8: Shares of branches that avoided instrumentation based on the results of fuzzing. *All* means that we patched all detected out-of-bounds accesses, regardless of the type; *c,100* means that we did not patch uncontrolled vulnerabilities that were triggered at least 100 times, and *c,10*—uncontrolled that were triggered at least 10 times.

cally, we used two whitelists for every library: a list based on all out-of-bounds accesses detected by SpecFuzz and a list that excludes uncontrolled vulnerabilities.

Table 4.8 shows the shares of the branches that were not instrumented because of whitelisting (out of the total number of branches in the application). Naturally, the shares directly correlate with the fuzzing coverage and with the number of detected vulnerabilities. If the coverage is large, the whitelisting proves to be very effective: In JSMN, SpecFuzz reduced the necessary instrumentation by ~77%.

Based on these builds, we evaluated the performance impact of the patches. For the measurements, we used benchmarks included in the libraries, where available; Otherwise, we used example applications. As such, we executed: the `speed` benchmark in OpenSSL (specifically, RSA, DSA, and ECDSA ciphers); `unbrotli` in Brotli; `bench` in HTTP; `test_bench` in libHTP; `run-loader` in libYAML; and a sample parser in JSMN.

The results are presented in Figure 4.6. For clarity, Table 4.9 shows the same results but interpreted as a speedup of a whitelisted patch compared to full hardening. As we can see, the overhead is considerably reduced. The performance cost was, on average, reduced by 23% for SLH and by 29% for `LFENCE`.

An overall tendency is the higher the coverage of fuzzing, the lower the overhead

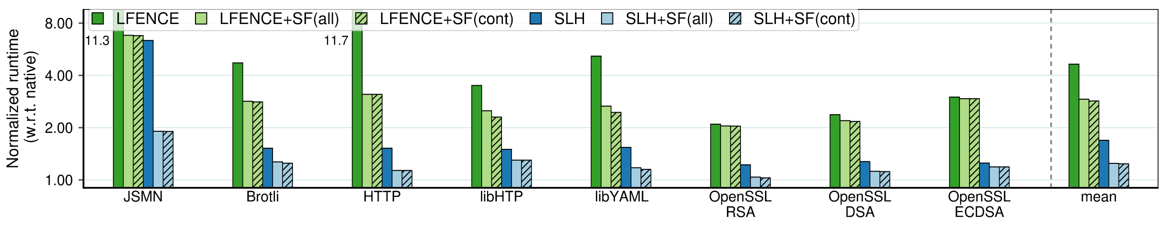


Figure 4.6: Performance overheads of hardening (Lower is better). *+SF(all)* means that we patched all detected out-of-bounds accesses, regardless of the type; *+SF(cont)* means that we did not patch uncontrolled vulnerabilities that were triggered at least 100 times.

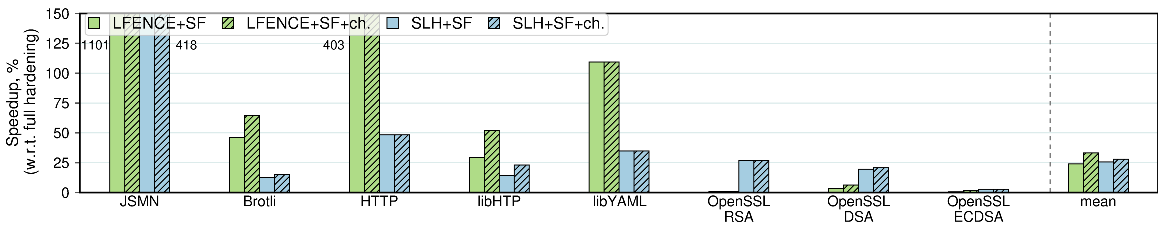


Figure 4.7: Performance improvement of SpecFuzz-based patched compared to full hardening (Higher is better). *+SF* are builds with fully-automatic whitelists and *+SF+ch.* are with whitelist based on manually checked vulnerabilities.

becomes. It stems from our benchmarks executing some of the code paths that could not be reached by the fuzzing drivers.

Another parameter is the number and the location of detected vulnerabilities. In ECDSA, SpecFuzz detected vulnerabilities on the hot path and, hence, we were not able to remove instrumentation from the places where it caused the highest performance overhead. SpecFuzz was also not effective at improving the `LFENCE` instrumentation of OpenSSL because it detected speculative bounds violations in the `bignum` functions that are located on the hot path.

A major reason for relatively high overheads is an issue with debug symbols that we encountered in LLVM. Sometimes, the debug symbols of the same code location would mismatch between compilations with different flags or would be completely absent for some instructions. Accordingly, some of the whitelisted locations would still be hardened. Note that this bug only impacts the performance, not the security guarantees. Nevertheless, when the issue is resolved, the overheads are likely to get lower.

One interesting example is JSMN, which experienced 5x slowdown with SLH and 11x with the `LFENCE` instrumentation. It is caused by an extremely high density of branches in the application (approximately one branch executed every cycle) and, thus, high reliance on branch prediction to efficiently utilize instruction parallelism. Complete hardening effectively disables this optimization and makes the execution much more sequential. At the

	SLH		LFENCE	
	+SF(all)	+SF(cont)	+SF(all)	+SF(cont)
JSMN	233%	234%	131%	132%
Brotli	20%	22%	66%	67%
HTTP	34%	34%	243%	242%
libHTP	15%	15%	40%	52%
libYAML	30%	33%	93%	110%
RSA	17%	19%	2%	2%
DSA	13%	14%	8%	9%
ECDSA	5%	5%	2%	2%

Table 4.9: Performance improvement of SpecFuzz-based patches compared to full hardening. +*SF(all)* means that we patched all detected out-of-bounds accesses, regardless of the type; +*SF(cont)* means that we did not patch uncontrolled vulnerabilities that were triggered at least 100 times.

Order	JSMN	Brotli	HTTP	libHTP	libYAML	OpenSSL
Both	1	6	1	78	3	992
RHS	0	4	3	36	3	601
RHS/covered	0	(1)	0	0	0	(1)

Table 4.10: Vulnerabilities detected by SpecFuzz and RH Scanner. The first row are the vulnerabilities detected by both tools; the second—only by RH Scanner; the third row are the vulnerabilities detected only by RH Scanner and located on the paths covered during our fuzzing experiments.

same time, SpecFuzz found very few vulnerabilities in JSMN and had high coverage (96%). Hence, the patches improved the performance by 230% (`LFENCE`) and 130% (SLH)

4.5.4 Comparison with Other Tools

Spectre Scanner. For comparison, we also tested the libraries with RedHat Scanner (Table 4.10). Although it detected fewer vulnerabilities than SpecFuzz, it found many vulnerabilities that SpecFuzz did not (second row). The reason behind it is almost all of them were located in the parts of code not covered during fuzzing. There were only two exceptions (row three), but both turned out to be false positives. (Because of the overwhelming amount of data, we did not investigate which share of the second row were false positives).

Note that when the whitelist-based patching is applied (§4.4.4), these undetected vulnerabilities do not compromise the security guarantees of SpecFuzz.

Respectre. Thanks to a cooperation with GRSecurity, we were able to also compare our

```
int base64_decode_single(signed char value_in) {
    static signed char decoding[] = {62, -1, ...}; // 80 elements
    value_in -= 43;
    if ((value_in < 0) || (value_in > decoding_size - 1))
        return -1;
    return decoding[(int) value_in];
}
...
int htp_base64_decode(const void *code_in, ...) {
    signed char fragment;
 ...
    do {
        ...
        fragment = base64_decode_single(*code_in++);
    } while (fragment < 0);
...
}
```

Figure 4.8: A BCB vulnerability in a Base64 decoding function.

results to a commercial static analysis tool Respectre [126]. As a test case we selected libHTP. In total, Respectre detected 167 vulnerabilities, out of which SpecFuzz found 79. Similarly to the previous experiment, the other 88 are located in the parts of libHTP not covered by fuzzing.

SpecFuzz was able to detect more vulnerabilities due to its more generic nature: For example, it can detect vulnerabilities that span multiple functions. On the other hand, Respectre is not confined by coverage and it can detect vulnerabilities in the parts of the application that cannot be reached by fuzzing.

Note, however, that these numbers are based on the raw outputs of both tools. It is possible that they include false positives. We attempted to manually analyze them, but the task appeared to be too complex. Speculative vulnerabilities in real software constitute of complex interactions of correct and wrong predictions, and analyzing them by just looking at the code has proven to be too error prone. Therefore, we report only raw numbers and leave the analysis of false positives to future work (e.g., to program analysis tools, as discussed in §4.6).

4.5.5 Case Studies

In this section, we present a detailed overview of three potential vulnerabilities found by SpecFuzz. Note that we did not test them in practice.

```
const unsigned long tag2bit[32] = {...};
unsigned long ASN1_tag2bit(int tag) {
    // misspeculation required
    if ((tag < 0) || (tag > 30)) return 0;
    return tag2bit[tag];
}
int asn1_item_embed_d2i(ASN1_VALUE **pval, ...) {
    int otag;
    ...
    switch (it->itype) {
    case ASN1_ITYPE_MSTRING:
        ret = asn1_check_tlen(..., &otag, ...);
         ...
        if (!(ASN1_tag2bit(otag) & it->utype)) {...}
    }
}
int asn1_check_tlen(..., int *otag, int expclass) {
  ...
  // decodes the ptag from message
  i = ASN1_get_object(..., &ptag);
  ...
  if (exptag >= 0) {
    if ((exptag != ptag) || (expclass != pclass)) {
      // misspeculation required
  ...
}
```

Figure 4.9: A BCB vulnerability in a ASN1 decoding function.

Speculative Overflow in libHTP base64 decoder. One of the utility functions that libHTP provides is base64 decoder, which is used to receive user data or parameters that may be sent in text format. This functionality is implemented in function `htp_base64_decode`, which calls function `base64_decode_single` in a loop. `base64_decode_single` decodes a Base64 encoded symbol by looking it up in a table of precomputed values (array `decoding`, lines 2–3). Before fetching the decoded symbol, the function checks the value for over- and underflows. The attacker can bypass the check by training the branch predictor and, thus, trigger a speculative overread at line 7.

Two properties make this vulnerability realistically exploitable. First, the attacker has control over the accessed address because the array index (`value_in`) is a part of the HTTP request. Second, the fetched value is further used for defining the control flow of the program (see the comparison at line 16), which allows the attacker to infer a part of the value (specifically, its sign) by observing the cache state.

The attacker could execute the attack as follows. She begins by sending a probing message to find out which cache line the first element of the array `decoding` uses. Then, she sends a valid message to train the branch predictor on predicting the bounds check (line 5) as true. Finally, she resets the cache state (e.g., flushes the cache) and sends a message that contains a symbol that triggers an overread, followed by a symbol that triggers a read from the first array element. If the read value is negative, the loop will do one more iteration, execute the second read, and the attacker will see a change in the state of the corresponding cache line. Otherwise, the loop will be terminated and the state will not change.

Speculative Overflow in OpenSSL ASN1 decoding. Another vulnerability is in OpenSSL ASN1 decoder. It is used to decode, for example, certificates that clients send to the server.

The attacker sends malicious ASN1 data to the victim. The victim uses `asn_*_d2i` family of functions to parse the message. One of the functions is `asn1_item_embed_d2i`, which, among others, decodes components of type `MSTRING`, verifying its tag in the process. The tag of the message is extracted through a call to `asn1_check_tlen` function, which delegates this calculation to `ASN1_get_object`. `asn1_check_tlen` verifies if the received tag matches the expected one (lines 22 and 23), however a misspeculation on any of these lines can nullify this check. Later, `asn1_item_embed_d2i` calls `ASN1_tag2bit` on the decoded tag value. If misspeculation happens in this function as well (line 4), the array `tag2bit` will be indexed with a potentially unbounded 4-byte integer. Later, this value is used to derive the control flow of the application (line 14), which may be used to leak user information.

4.6 Discussion

4.6.1 Other Spectre Attacks

Bounds Check Bypass is not the only type of speculative vulnerabilities that could be detected by speculative exposure. Below we give an overview of instrumentation that can be used for other Spectre-type attacks.

Branch Target Injection [142] is a Spectre variant targeting speculation at indirect jumps. When an indirect jump instruction is executed, the CPU speculates the jump target using the branch predictor without waiting for the actual target address computation to finish. The attacker can exploit this behavior by training the branch predictor to execute a jump to a code snippet that would leak program data via a side channel.

SpecFuzz could be modified to simulate BTI by maintaining a software history buffer for every indirect branch in the application. Then, at an indirect branch, SpecFuzz would *(i)* record the current branch target into the history buffer and *(ii)* run a simulation for every previously recorded target. This approach works, however, only under the assumption that attacker can train the branch predictor only by providing data to the application and cannot inject arbitrary targets into the branch predictor's history buffer from another application on the same core.

Return Address Misprediction [164, 147] attack is a variant of Branch Target Injection. The CPU maintains a small number of most recently used return addresses in a dedicated cache, pushing the return address into this cache on each call instruction and popping it from the cache on each return instruction. When this cache becomes empty, the CPU will speculate the return address using the indirect Branch Target Buffer. To simulate this vulnerability, SpecFuzz could instrument call and return instructions to, correspondingly, increment and decrement a counter, jumping to an address from history buffer on return addresses with negative or zero counter value. This simulation should be combined with the previous one as the return address prediction could fall back to indirect branch target prediction.

Speculative Store Bypass [101] is a microarchitectural vulnerability caused by CPU ignoring the potential dependencies between load and store instructions during speculation. When a store operation is delayed, a subsequent load from the same address may speculatively reuse the old value from the cache. To simulate this attack, SpecFuzz could be extended to start a simulation before every write to memory. Then, SpecFuzz would skip the store during the simulation, but execute it after the rollback.

4.6.2 Limitations

In this section, we discuss the conceptual problems we have discovered while developing SpecFuzz as well as potential solutions to them.

Reducing the Complexity of Nested Simulation. As we discussed in §4.3.2, complete nested simulation is too expensive and limiting the order of simulation may lead to false negatives. One way we could resolve this problem is by statically analyzing the program before fuzzing it, such that the typical vulnerable patterns as well as typical false positives would be purged from the simulation, thus reducing its cost.

False Negatives. SpecFuzz will not find a vulnerability if the fuzzer does not generate an input that would trigger it. Unfortunately, it is an inherent problem of fuzzing.

SpecFuzz will also not detect the vulnerabilities of different types other than out-of-bounds accesses (e.g., use-after free, missing permission checks). Detecting them would require applying the corresponding integrity checking techniques instead of AddressSanitizer.

Fuzzing Driver. Another inherent issue of all fuzzing techniques is their coverage. As we saw in §4.5, it highly depends on the fuzzing driver and a bad driver may severely limit the reach of testing. Since we use whitelist-based patching, low coverage may cause high performance overhead in patched applications. It could be improved by applying tools that generate drivers automatically, such as FUDGE [32].

Mislabeling. During the evaluation, we discovered that our vulnerability analysis technique (see §4.4) sometimes gives a false result and mistakenly labels an uncontrolled vulnerability as a controlled one. It happens because AddressSanitizer reports only the accessed address and not the distance between the address and the referent object (i.e., offset). Therefore, if the object size differs among the test runs, the accessed address will also be different, even if the offset is the same.

For example, one common case of mislabeling is off-by-one accesses. If an array is read in a loop, our simulation will force the loop to take a few additional iterations and read a few elements beyond the array's bounds. If the array size differs from one test run to another, the analysis would mark this vulnerability as controllable.

To avoid this issue, we could use a more complete memory safety technique (e.g., Intel MPX [130]) that maintains metadata about referent objects. Unfortunately, none of such techniques is supported by LLVM out-of-the-box. To resolve this issue, we would have to implement MPX support or migrate SpecFuzz to another compiler.

An even better solution would be to use a program analysis technique (e.g., taint analysis or symbolic execution) to verify the attacker's control. We leave it to future work.

Legacy Code and Callbacks. Because we implemented SpecFuzz as a compiler pass, it cannot run the simulation in non-instrumented parts of the application (e.g., in system libraries) as well as in the calls from these parts (callbacks). To overcome this problem, we could have implemented SpecFuzz as a binary instrumentation tool (e.g., with PIN [163]). Yet, techniques of this type are normally heavy-weight and it would considerably increase the required fuzzing time.

We presented a technique to make speculative execution vulnerabilities visible by simulating them in software. We demonstrated the technique by implementing a Bounds Check Bypass detection tool called SpecFuzz. During the evaluation, the tool has proven to be more effective at finding vulnerabilities than the available static analysis tools and the patches produced based on the fuzzing results had better performance than conservative hardening techniques.

Chapter 5

Varys: Runtime Validation of System Defences

In this chapter, our goal is to develop a method to *continuously check if the system creates such an execution environment that precludes the attacker from reading traces.*

A distinct property of system defences is that they can be disabled or reconfigured at runtime, and the protected applications will not know about it. Thus, to ensure a secure execution environment, we have to continuously check if the system correctly serves the application requests.

The task becomes particularly important when privileged software (e.g., the OS kernel or the hypervisor) is under the attacker's control. It may happen, for example, when the program executes in an untrusted cloud environment, which corresponds to the privileged threat model in §2.2. Under a privileged attacker, the victim has no guarantees that the system will provide the requested defences. Instead, it may accept the request, pretend that it enabled the defence, but in fact leave it disabled. Thus, the application needs a way of validating the defence without relying on the OS or the hypervisor.

The first step to protecting an application from a privileged adversary is to rely on a Trusted Execution Environment, such as Intel SGX [130]. SGX enclaves provide a shielded environment to securely execute sensitive programs on commodity CPUs. They enforce integrity and confidentiality of the program data on the hardware level, thus preventing any code outside an enclave from accessing it. As such, Intel SGX prevents a privileged adversary from directly attacking the victim program. Unfortunately, its security guarantees do not extend to microarchitectural attacks [51, 102, 273, 245, 254, 117], and enclaves still have to rely on system software for microarchitectural protection. Thus, the problem of validating system defences from within Intel SGX enclaves remains open.

5.1 System Overview

We introduce a technique for validating system defences from within an Intel SGX enclave. The key idea is *self-monitoring*: we modify the application to constantly monitor its execution environment in search of side-effects of a side-channel attack. We also request the system to make these side-effects impossible under normal circumstances, so that their presence becomes a clear indicator of an ongoing attack.

We showcase the technique in Varys, a tool that defends SGX enclaves against cache timing and page table attacks. Varys requests a potentially malicious OS to provide a side-channel protected execution environment. To validate the request, a trusted runtime inside the enclave continuously monitors its execution to detect violations of the requested environment.

Protected Environment. With respect to the requested environment, our main observation is that all published page table and L1/L2 cache timing attacks on SGX require either a high rate of enclave exits, or control of a sibling hyperthread on the same CPU core with the victim. Consequently, if an enclave is guarded against frequent exits and executes on a dedicated CPU core without sharing it with untrusted threads, it would be protected.

Varys relies on *trusted core reservation*[1]. It requests the OS to create such an environment that the attacker cannot access the core's resources shared with the enclave threads while they are running, nor can it recover any secrets from the core's L1 and L2 caches afterward. A simple way to achieve it would have been to disable hyperthreading, but it causes performance degradation [236]. Instead, Varys requests the OS to schedule enclave threads in pairs such that they always run *together* on the same physical core. Assuming a standard processor with SMT §2.1.5, the threads occupy both hardware threads of the core, and the attacker cannot access the core's caches. It also prevents exit-less side-channel attacks on page table attributes [245] because they require the attacker's access to the core's TLB—available only if the attacker thread is running on that core. Additionally, to ensure that the victim leaves no traces in the caches when de-scheduled from the core, Varys explicitly evicts the caches when the enclave threads are preempted.

To reduce the frequency of legitimate exits and to lower the false positives, Varys uses exitless system calls [31] and in-enclave thread scheduling such that multiple application threads can share the same OS thread. Moreover, Varys configures the OS to reduce the frequency of interrupts routed to the core in order to avoid interference with attack-free program execution.

[1]After the publication of our original paper, this feature was introduced in Linux under the name Core Scheduling [88]. To the extent of our knowledge, it was developed independently of our work.

Self-monitoring. The primary challenge that Varys addresses is maintaining the protected environment in face of a potentially malicious OS. An untrusted OS may ignore the request to pin two enclave threads to the same physical core, and may preempt each of the threads separately in an attempt to break Varys's defense. Varys prevents these threats via two mechanisms:

- *Thread Handshake*: To validate trusted reservation, the paired application threads periodically communicate with each other via an on-core side channel (we call it a handshake). If they are scheduled correctly (i.e., on the same core), the signal will pass through the channel; otherwise, the threads will not observe the signal.
- *Asynchronous Enclave Exit Monitoring*: Varys monitors enclave exits (e.g., for scheduling another process on the core or handling an exception), and restricts their frequency, terminating the enclave once the frequency bound is exceeded. Varys sets the bound to the values that render all known attacks impossible. Notably, the bound is much higher than the frequency of exits in an attack-free execution, thereby minimizing the chances of false positives (see §5.3.1).

If either of these checks fails, Varys terminates the enclave.

Hardware Extensions. Due to the lack of appropriate hardware support in today's SGX hardware, Varys remains vulnerable to timing attacks on Last Level Cache (LLC) as we explain in §5.7. We suggest a few minor hardware modifications that hold the potential to solve this limitation and additionally, eliminate most of the runtime overhead. These extensions allow the operating system to stay in control of resource allocations but permit an enclave to determine if its allocation has changed.

5.2 Background: Intel Software Guard Extensions

A Trusted Execution Environment (TEE) is a special execution mode where the hardware guarantees integrity and/or confidentiality of the program's code and data.

Intel Software Guard Extensions is an ISA extension that adds hardware support for TEE in Intel CPUs. SGX offers creation of *enclaves*—isolated and encrypted memory regions, with code running in a novel enclave execution mode. As such, SGX enclaves prevent privileged software from reading and modifying the application data, and from directly observing its execution.

The memory region protected by SGX is called Enclave Page Cache (EPC). Accesses to the EPC go through the Memory Encryption Engine (MEE), a hardware module that transparently encrypts all the data that the program writes into EPC, and decrypts when the program reads it.

EPC paging. In the early versions of SGX, the EPC size was limited (as small as 128 MB in SGX v1), but enclaves could use more virtual memory via paging. EPC paging happens when a page not backed by physical memory is accessed. The CPU, in coordination with the OS kernel, encrypts contents of the EPC page, stores it in untrusted memory, and allocates the EPC page to a new virtual page. The eviction mechanism preserves confidentiality, integrity, and freshness of EPC pages.

AEX. The enclave data is encrypted only when it is stored in memory (i.e., in EPC), but not when it is in the CPU registers or caches. Thus, upon a hardware exception (e.g., fault or interrupt), the OS could read enclave's data in the CPU registers. To prevent it, SGX has a mechanism of *Asynchronous Enclave Exits (AEX)*, where the CPU automatically stores the register values into the encrypted memory and resets the registers before re-directing the exception to the OS.

Restrictions. Code running inside an enclave is subject to several restrictions: it can neither directly jump to code outside the enclave nor directly invoke system calls and several trapping instructions. Enclaves are running in the address space of the host userspace application and can access all data that is mapped into this address space.

5.3 Protected Environment

Varys relies on the OS to provide an environment protected from side-channel attacks on L1/L2 caches and on page tables. This section analyzes the runtime conditions required for these attacks to be successful, and describes the environment that prevents the conditions.

5.3.1 Attack Requirements

Cache attacks (§2.3.2) can be classified into either *concurrent*, i.e., running in parallel with the victim, or *time-sliced*, i.e., time-sharing the core (or a hyperthread) with the victim.

Time-sliced cache attacks $\Longrightarrow$ high AEX rate. For a time-sliced attack to be successful, the runtime of the victim in each time slice must be short; otherwise, the cache noise will become too high to derive any meaningful information. For example, the attack described by Zhang et al. [286] can be prevented by enforcing minimal runtime of 100us [248], which translates into the interrupt rate of 10kHz. This rate is dramatically higher than the preemption rate under normal conditions—below 100Hz (see §5.3.3). If the victim is an enclave thread, its preemption implies an AEX.

Concurrent cache attacks $\Longrightarrow$ shared core. The adversary running in parallel with the victim must be able to access the cache level shared with it. Thus, L1/L2 cache attacks are not possible unless the adversary controls a sibling hyperthread.

We note that with the availability of the Cache Allocation Technology (CAT) [130], the share of the Last Level Cache (LLC) can also be allocated to a hardware thread, preventing any kind of concurrent LLC attacks [161]. However, this defense is ineffective for SGX because the allocation is controlled by an untrusted OS. We suggest one possible solution to this problem in §5.7.

Page-fault page table attacks $\Longrightarrow$ high AEX rate. These attacks inherently increase the page fault rate, and consequently AEX rate, as they induce page faults to infer the accessed addresses. For example, as reported by Wang et al. [254], a page table attack on EdDSA requires approximately 11000 exits per second. In fact, high exit rates have already been used as an attack indicator [223, 108].

Interrupt-driven page-bit attacks $\Longrightarrow$ high AEX rate. If the attacker does not share a physical core with the victim, these attacks incur a high exit rate because the attacker must flush the TLB on a remote core via Inter-Processor Interrupts (IPIs). The rate is cited to be around 5500Hz [245, 254]. While lower than other attacks, it is still above 100Hz experienced in attack-free execution (§5.3.3).

Exit-less page-bit attacks $\Longrightarrow$ shared core. The only way to force TLB flushes without IPIs is by running an adversary thread on the same physical core as the victim to force TLB evictions [254]. These attacks involve no enclave exits, hence are called *silent*.

In summary, all the known page table and L1/L2 cache timing side-channel attacks on SGX rely on ① an adversary-controlled sibling hyperthread on the same physical core, or/and ② an abnormally high rate of asynchronous enclave exits. The only exception is the case when the victim has a slowly-changing working set, which we discuss in §5.6.3.

5.3.2 Trusted Core Reservation

The simplest way to ensure that an adversarial hyperthread cannot perform concurrent attacks on the same physical core would be to disable hyperthreading [166]. However, doing so significantly hampers application performance.

An alternative approach is to allow sharing of a core only by benign threads. Considering that in our threat model only the enclave is trusted, we allow core sharing only among the threads from the same enclave. We can achieve this goal by dividing the application threads into pairs (if the number of threads is odd, we spawn a dummy thread) and requesting the OS to schedule the pairs on the same cores.

5.3.3 Restricting Enclave Exit Frequency

To prevent attacks with a high AEX rate, we request the system to reduce the baseline rate of exits. Ensuring it is imperative for our approach because a high exit rate under normal conditions makes it harder to distinguish an attack from the attack-free execution. In the worst case, if the application's normal exit rate is sufficiently high (i.e., more than 5500 exits/second, see below), the adversary does not have to introduce any additional exits and can abuse the existing ones to retrieve information. Therefore, we have to analyze the sources of exits and the ways of eliminating or reducing them.

Under SGX, an application may exit the enclave for one of the following reasons:

1. When the application needs to invoke a system call;
2. To handle system timer interrupts, with up to 1000 AEX/s, depending on the kernel configuration;
3. To handle other interrupts, which could happen especially frequently if Varys runs with a noisy neighbor (e.g., a web server);
4. To perform EPC paging when the memory footprint exceeds the EPC size;
5. To handle minor page faults, which could happen frequently if the application works with large files.

Additionally, SGX enclaves may exit when the application tries to execute a forbidden instruction in the enclave mode. Unfortunately, there is no workaround for such instructions except re-writing the application code.

We strive to reduce the number of enclave exits as follows. We use asynchronous exit-less system calls implemented, for example, in Eleos [187] and SCONE [31] (which we use in our implementation). Further, we combine asynchronous system calls with user-level thread scheduling inside the enclave to avoid reliance on the OS scheduling. When a user thread issues a system call, the arguments of this system call are copied to the outside of the enclave in the untrusted shared memory region. The enclave thread scheduler switches to another user thread, and restarts the preempted one only when the system call returns. We avoid timer interrupts by setting the timer frequency to the lowest available (100 Hz), and enabling the DynTicks feature of the Linux kernel, which further reduces the rate of timer interrupts. Regular interrupts are re-routed to non-enclave cores. Last, we prevent minor page faults when accessing untrusted memory via `MAP_POPULATE` flag to `mmap` calls.

To evaluate the overall impact of these changes, we measure the exit frequencies of the applications in PARSEC and Phoenix benchmark suites (see §5.6 for the benchmarks' description). The results are shown in Figure 5.1.

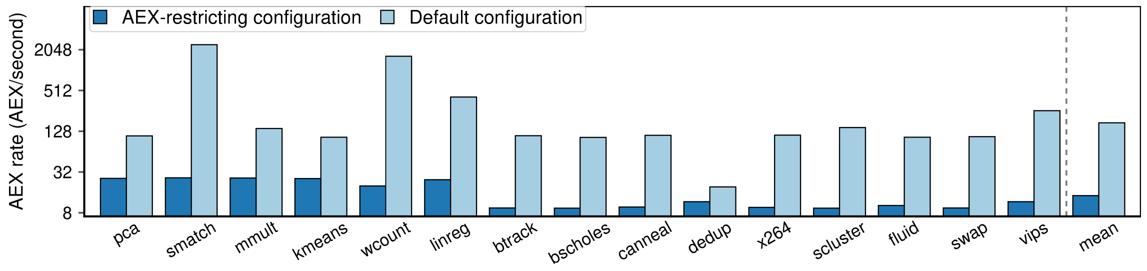


Figure 5.1: AEX rates under normal system configuration and with re-configured system.

As we see, the rate is relatively stable across the benchmarks and much lower than the potential attack rate of more than 1000 exits per second. Specifically, the attack presented by Van Bulck et al. [245] has one of the lowest interrupt rates among the published time-sliced attacks. We ran the open-sourced version of the attack and observed the rate of at least 5500 exits per second, which is in line with the rate presented in the paper. Correspondingly, if we detect that the AEX rate is getting above 100 Hz, we can consider it a potential attack and take appropriate measures. To avoid false positives, we could set the threshold even higher—around 2kHz—without compromising security (this is illustrated in §5.6.3).

5.4 Method: Enclave Self-Monitoring

To validate the protected environment, Varys monitors the enclave execution. It checks if any of the enclave threads are sharing a core with another process, and if the AEX rate is abnormally high.

5.4.1 Thread Handshake

Since we do not trust the OS, we ensure collocation by establishing a covert channel in the L1 cache. The idea is to determine whether the threads share the L1 cache or only LLC. The latter would imply the threads are on different physical cores.

We refer to the procedure that determines the threads co-location on the core as a *handshake*. To perform the handshake, we establish a simple covert channel between the threads via L1: One of the two threads writes a dummy value to a shared memory location, thus forcing it to L1. Then, the sibling thread reads the same memory location and measures the timing. If the reading is fast (up to 10 cycles per read), both threads use the same L1 cache, otherwise (more than 40 cycles) they share only LLC, implying they are on different cores. If the threads indeed run on different cores, the OS did not satisfy the scheduling request made by Varys. We conclude that the enclave is under attack and terminate it. As SGX does not provide a trusted fine-grain time source, we implement our own (§5.5.2).

The operating system could compromise the handshake by rescheduling these threads on different cores immediately *after* the handshake. However, this rescheduling would cause an asynchronous enclave exit (AEX), which we detect via AEX monitoring as we discuss next.

5.4.2 AEX Monitoring

To detect an asynchronous enclave exit, we monitor the SGX State Save Area (SSA). The SSA is used to store the execution state of an enclave upon an AEX. Since some parts of the enclave execution state are deterministic, we can detect an AEX by overwriting one of the SSA fields with an invalid value and monitoring it for changes.

For example, the Program Counter register (PC) never contains a zero value during a normal program execution, because it implies execution of the NULL page. Thus, if we write zero to the SSA field that stores PC, SGX will overwrite it with a non-zero value at every AEX. To detect an AEX, we periodically read and compare this field with zero. Note, it is not the only SSA field that we could use for this purpose; many other fields, such as `EXIT_TYPE`, could be used too.

Now that we have a detection mechanism, it is sufficient to count the AEX events and abort the application if they are too frequent. Yet to calculate the frequency, we need a trusted time source that is not available for enclaves. Fortunately, precise timing is not necessary for this particular purpose as we would only use the time to estimate the number of instructions executed between AEXs. It is possible to estimate it through the AEX monitoring routine that our compiler pass (§5.5.1) adds to the application. Since it adds the routine every few hundred LLVM IR instructions, counting the number of times it is called serves as a natural counter of LLVM IR instructions. In Varys, we define the AEX rate as a number of AEXs per number of executed IR instructions.

Even though IR instructions do not correspond to machine instructions, one IR instruction maps on average to less than one x86-64 machine instruction[1]. Thus, we overestimate the AEX rate, which is safe from the security perspective.

Originally, we considered using TSX (Transactional Synchronization Extensions) to detect AEXs—similar to the approach proposed by Gruss et al. [108]. The main limitation of TSX is, however, that it does not permit non-transactional memory accesses within transactions. Hence, handshaking is not possible within a TSX transaction—this would lead to a transaction abort. Moreover, the maximum transaction length is limited, and we would need to split executions into multiple transactions.

[1]In our experience with Phoenix and PARSEC benchmark suites, calling the monitoring routine every 100 IR instructions resulted in the polling period of 70–150 cycles.

```
while(true) {
    wait_for_request();
    if (secret == 0) {
        int response = *a;
    } else {
        int response = *b;
    }
}
```

Figure 5.2: An example of code leaking information in cache side-channel even with a low frequency of enclave exits. If a and b are on different cache lines and the requests are coming infrequently, it is sufficient to probe the cache at the default frequency of OS timer interrupts.

5.4.3 Removing Residual Cache Leakage

Even with a low frequency of enclave exits, some leakage will persist if the victim has a slowly changing working set. Consider the example in Figure 5.2: The replies to user requests depend on the value of a secret. If requests arrive infrequently (e.g., one per second), restricting the exit frequency would not be sufficient; even if we set the bar as low as 10 exits per second (the rate we achieved in §5.3.3), the victim will touch only one cache line and thus, will reveal the secret.

To completely remove the leakage at enclave exits, we need to flush the cache before exiting the enclave. This would remove any residual cache traces that an adversary could use to learn whether the enclave has accessed certain cache lines. Unfortunately, this operation is unavailable at userspace on Intel CPUs [130], and we have no possibility to request a cache flush at each AEX. Moreover, Ge et al. [95] have proven that the kernel-space flush commands do not flush the caches completely. The `CLFUSH` instruction does not help either as it flushes a memory address, not the whole cache set. Thus, it cannot flush the adversary's eviction set residing in a different virtual address space, as it is the case in Prime+Probe attacks.

Instead, on each enclave entry, we write a dummy value to all cache lines in a continuous cache-sized memory region (e.g., 32 KB for L1), hereafter called *eviction region*. In case of L1, for which instruction and data are disjoint, we also execute a 32 KB dummy piece of code to evict the instruction cache. This way, regardless of what the victim does in between the exits, an external observer will see that all the cache sets and all the cache ways were accessed and no information will be leaked.

5.5 Implementation

We implement Varys as an LLVM compiler pass that inserts periodic calls to a runtime library. We use SCONE to provide us with asynchronous system calls as well as in-enclave threading so that we minimize the need for an application to exit the enclave.

5.5.1 LLVM Compiler Pass

The cornerstone of Varys is the enclave exit detection. As discussed in §5.4.2, it requires all application threads to periodically poll the SSA region. Although we implement the checks as a part of a runtime library (§5.5.2), calls to the library have to be inserted directly into the application. To do this, we instrument the application using LLVM [153].

The goal of the instrumentation pass is to call the library and do the SSA polling with a predictable and configurable frequency. We achieve it by inserting the following sequence before every basic block: We increment a counter by the length of the basic block (in LLVM IR instructions), call the library if the counter reached a threshold, and skip the call otherwise. If the basic block is longer than the threshold, we add several calls. This way, the checks will be performed each time the application executes a given (configurable) number of IR instructions. We also reserve one of the CPU registers for the counter, as it is manipulated every few instructions, and having the counter in memory would cause much higher overheads.

A drawback of SSA polling is that it has a blind zone. If a malicious OS preempts a thread multiple times in a very short period of time, the interrupts may happen before the counter reaches the threshold, and the thread checks the SSA value. Hence, they will be all counted as a single enclave exit. This allows an adversary to launch stealthy cache attacks on small pieces of code by issuing occasional series of frequent preemptions. Yet this vulnerability would be hard to exploit because the blind zone is narrow—on the order of dozens of cycles, depending on the configuration—and the adversary must run in tight synchronization with the victim to retrieve any meaningful information.

Optimization. Adding even a small piece of code to every basic block could be expensive as the blocks themselves are often only 4–5 instructions long. We avoid this by applying the following optimization.

Consider a basic block B0 with two successors, B1 and B2. In a naive version, at the beginning of each basic block we increment the IR instruction counter by the length of the corresponding basic block. However, if B0 cannot jump into itself, it will always proceed to a successor. Therefore, it is sufficient to increment the counters only at the beginnings of B1

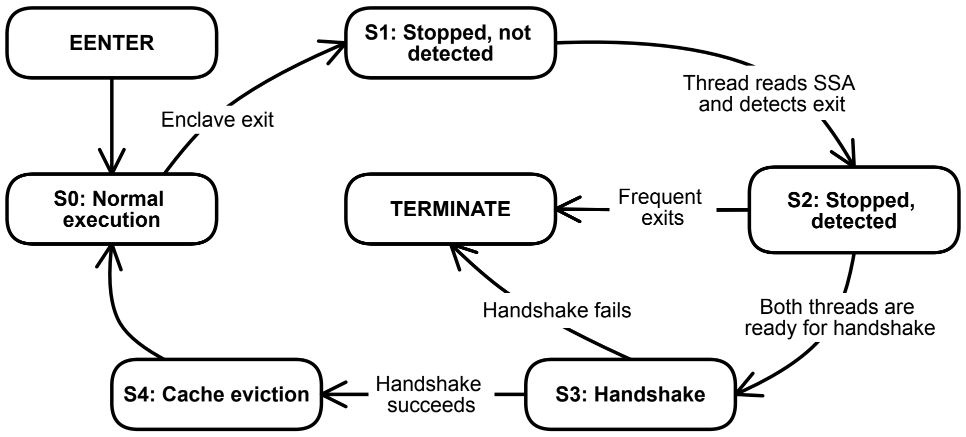


Figure 5.3: State diagram of a Varys-protected application.

and B2 by, accordingly, `length(B0)+length(B1)` and `length(B0)+length(B2)`. If B1 or B2 have more than one predecessor, it could lead to overestimation and more frequent SSA polling, which only reduces the blind zone.

5.5.2 Runtime Library

Most of Varys' functionality is contained in a runtime library implementing the state machine in Figure 5.3. When a program starts, it begins *normal execution* (S0). As long as the program is in this state, it counts executed instructions thus simulating a timer.

When one of the threads is interrupted, the CPU executes an AEX and overwrites the corresponding SSA (S1). As its sibling thread periodically polls the SSA, it eventually detects the exit. Then, if the program has managed to make sufficient progress since the last AEX (i.e., if the IR instruction counter has a large enough value), it transfers to the *detected* state (S2). Otherwise, the program terminates. To avoid false positives, we could terminate the program only if it happens several times in a row.

In S2, the sibling declares that the handshake is pending and starts busy-waiting. When the first thread resumes, it detects the pending handshake, and the pair enters state S3. If the handshake fails, the program is terminated[2]. Otherwise, one of the threads evicts L1 and L2 caches, and the pair continues normal execution.

Software Timer. To perform cache measurements during the handshake phase, we need a trusted fine-grained source of time. Since trusted hardware time counters are not available in the current version of SGX, we implement it in software (similar to Schwarz et al. [211]). We spawn an enclave thread incrementing a global variable in a tight loop, giving us

[2]In practice, timing measurements are noisy and the handshake may fail for benign reasons. Therefore, we retry it several times and consider it failed only if the timing is persistently high.

```
.align 64
label1: JMP label2  // jump to the next cache line
.align 64
label2: JMP label3
```

Figure 5.4: A code snippet evicting cache lines in the L1 instruction cache. For evicting a 32 KB cache, the pattern is repeated 512 times.

approximately one tick per cycle.

However, the frequency of the software timer is not reliable. An adversary can abuse the power management features of modern Intel CPUs and reduce the timer tick frequency by reducing the operational frequency of the underlying core. If the timer becomes slow enough, the handshake will always succeed. To protect against it, we measure the timing of a constant-time operation. Then, we execute the handshake only if the measurement matches the expected value.

Instruction Cache Eviction. Writing to a large memory region is not sufficient for evicting L1 or L2 caches. L1 has separate caches for data (L1D) and instructions (L1I), and L2 is non-inclusive, which means that evicting L2 does not imply evicting L1I. Hence, the attacks targeting execution path are still possible.

To evict L1I, we have to execute a large chunk of code. The fastest way of doing so is shown in Figure 5.4. The code goes over a 32 KB region and executes a jump for each cache line thus forcing it into L1I.

L2 Cache Eviction. Evicting L2 cache is not as straightforward as L1 as it is physically-indexed physically-tagged (PIPT) [270]. For the L2 cache, allocating and iterating over a continuous virtual memory region does not imply access to continuous physical memory, and therefore does not guarantee cache eviction. A malicious OS could apply cache colouring [43, 136] to allocate physical pages in a way that the vulnerable memory locations map to one part of the cache and the rest of the address space to another. This way, the vulnerable cache sets would not be evicted, and the leakage would persist.

With L2 cache, we do two passes over the eviction region. The first time, we read the region to evict the L2 cache. The second time, we read and measure the timing of this read. If the region is continuous, the first read completely fills the cache and the second read should be relatively fast as all the data is in the cache. However, if it is not the case, some pages of the eviction region would be competing for cache lines and evicting each other, thus making the second read slower. We use this as an indicator that L2 eviction is not reliable and we should try to allocate another region. If the OS keeps serving non-continuous memory, we terminate the application as the execution cannot be considered reliable anymore.

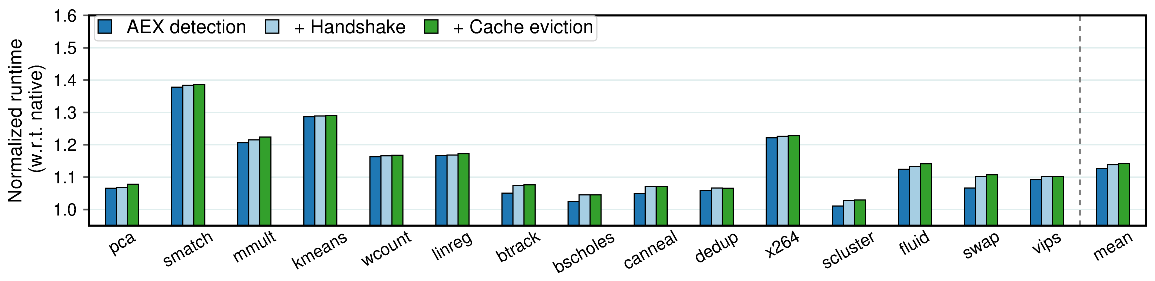


Figure 5.5: Performance impact of Varys security features with respect to native SGX version. Each next bar includes all the previous features. (Lower is better.)

5.5.3 SCONE

We base our implementation on SCONE [31], a shielding framework for running unmodified applications inside SGX enclaves. Among other benefits, SCONE provides two features that make our implementation more efficient and compact. First, it implements user-level threading, which significantly simplifies thread pairing. As the number of enclave threads is independent of the number of application threads and fixed, it suffices to allocate and initialize thread pairs at program startup. Second, it provides asynchronous system calls. They not only significantly reduce the rate of enclave exits but also make this rate more predictable and application agnostic.

We should note that Varys is not conceptually linked to SCONE. We could have avoided user-level threading by modifying the standard library to dynamically assign thread pairs. The synchronous system calls are also not an obstacle, but they require a mechanism to distinguish different kinds of enclave exits.

5.6 Evaluation

In this section, we answer the following questions:

- What are the runtime overheads of Varys?
- How does it influence scalability of multithreaded applications?
- How efficiently does it detect violations of trusted reservation and high AEX rates?
- What is the rate of false positives?

5.6.1 Experimental Setup

Applications. We base our evaluation on the Fex evaluation framework (Appendix A), with PARSEC [45] and Phoenix [200] benchmark suites as workloads. The following

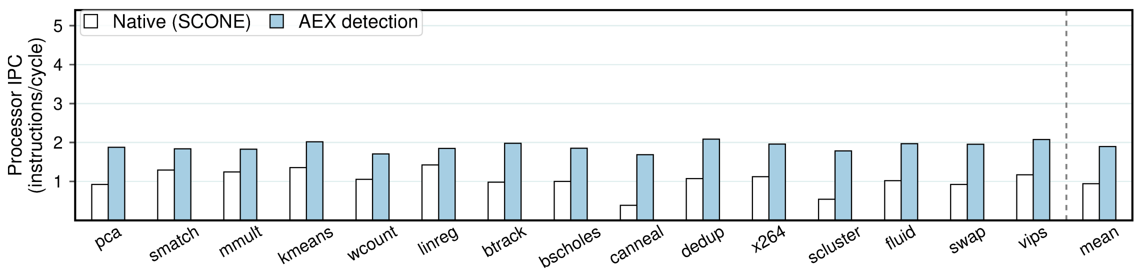


Figure 5.6: IPC (instructions/cycle) numbers for native and protected versions.

benchmarks were excluded: *raytrace* depends on the dynamic X Window System libraries not shipped together with the benchmark; *freqmine* is based on OpenMP; *facesim* and *ferret* fail to compile under SCONE due to position-independent code issues. Together with the benchmarks, we recompile and instrument all the libraries they depend upon. We also manually instrument the most frequently used *libc* functions so that at least 90% of the execution time is spend in a protected code. We used the largest inputs that do not cause intensive EPC paging as otherwise, they could lead to frequent false positives.

Methodology. All overheads were calculated over the native SGX versions build with SCONE. The reported results are averaged over 10 runs and the "mean" value is a geomean across all the benchmarks.

Testbed. We ran all the experiments on a 4-core (8 hyperthreads) Intel Xeon CPU operating at 3.6 GHz (Skylake microarchitecture) with 32 KB L1 and 256 KB L2 private caches, an 8 MB L3 shared cache, 64 GB of RAM, and a 1TB SATA-based SSD. The machine was running Linux kernel 4.14. To reduce the rate of enclave exits, we configured the system as discussed in §5.3.3.

5.6.2 Performance Evaluation

Runtime. Figure 5.5 presents the runtime overheads of different Varys security features. On average, the overhead is ~15%, but it varies significantly among benchmarks.

A major part of the overhead comes from the AEX detection, which we implement as a compiler pass. Since the instrumentation adds instructions that are not data dependent on the application's data flow, they can run in parallel. Therefore, they highly benefit from instruction level parallelism (ILP), which we illustrate with Figure 5.6. The applications that have lower ILP utilization in the native version (e.g., *canneal* and *stream cluster*) can run a larger part of the instrumentation in parallel, thus amortizing the overhead.

Since we apply instrumentation per basic block, another factor that influences the overhead is the average size of basic blocks. The applications dominated by long sequences

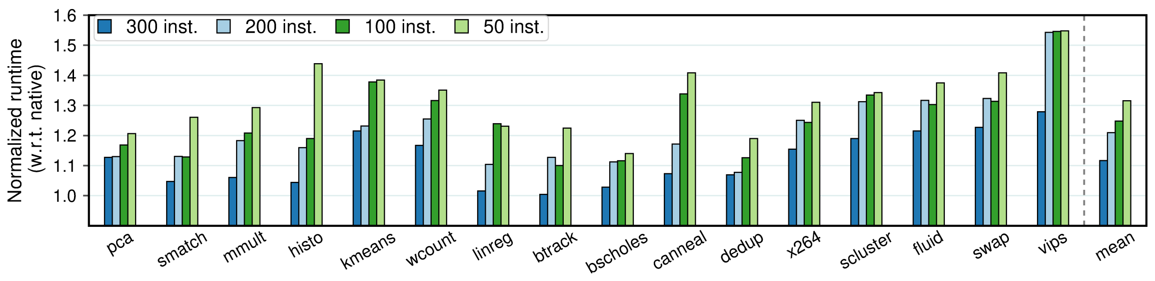


Figure 5.7: Impact of the SSA monitoring frequency on performance. (Lower is better.)

of arithmetic operations (e.g., *linear regression*) tend to have longer basic blocks and lower number of additional instructions (53% in this case), hence the lower overhead. At the same time, the applications with tight loops on the hot path cause higher overhead. Therefore, *string match* has higher overhead than *kmeans*, even though they have approximately the same ILP utilization.

The second source of overhead is trusted reservation. It does not cause a significant slowdown because the handshake protocol is relatively small, including ten memory accesses for the covert channel and the surrounding code for the measurement. The overhead could be higher as the handshake is synchronized, i.e., two threads in a pair can make progress only if both are running. Otherwise, if one thread is de-scheduled, the second one has to stop and wait. Yet, as we see in Figure 5.5, the overhead of this procedure is negligible.

Finally, cache eviction involves writing to a 256 KB data region and executing a 32 KB code block. Due to the pseudo-LRU eviction policy of Intel caches, we have to repeat the writing several times (three, in our case). Together, it takes dozens of microseconds to execute, depending on the number of cache misses. Fortunately, we evict only after enclave exits, which are infrequent under normal conditions (§5.3.3) and the overhead is low.

Impact of SSA Monitoring. Varys does not detect AEXs immediately, but with a configurable latency that depends on how frequently we check the SSA value. It introduces a trade-off between security and performance: longer latencies create an opportunity for the adversary to launch a brief attack and stay undetected, but shorter latencies require more work from the application's side and hamper performance. To evaluate the impact of this parameter, we measured runtime overheads of different frequencies. The results are shown in Figure 5.7.

Multithreading. As Varys is primarily targeted at multithreaded applications, it is crucial to understand its impact on multithreaded performance. To evaluate this parameter, we measured the execution time of all benchmarks with 2, 4, and 8 threads with respect to native versions with the same number of threads. Mind that these are user-level threads; the number of underlying enclave threads is always 4. The results are presented in Figure 5.8.

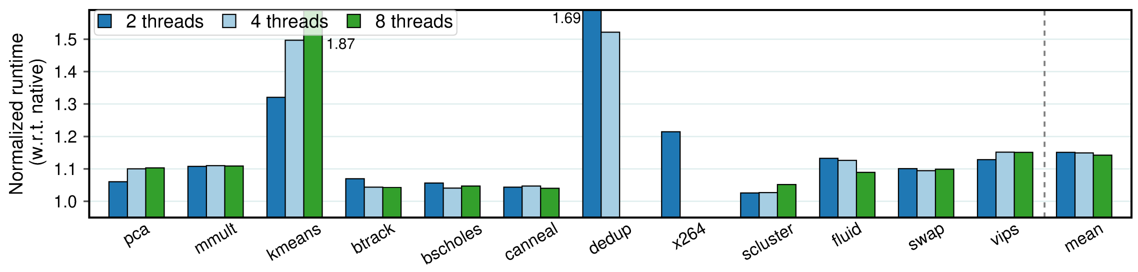


Figure 5.8: Runtime overhead with different number of threads. (Lower is better.)

Generally, Varys does not have a significant impact on multithreaded scaling. However, there are a few exceptions. First, larger memory consumption required for multithreading causes EPC paging[3], thus increasing the AEX rate and sometimes even causing false positives. We can see this effect in *dedup* and *x264*: the higher AEX rate makes the flushing more expensive and eventually leads to false positives with higher numbers of threads. For the same reason, we excluded *linear regression*, *string match*, and *word count* from the experiment.

Another interesting effect happens in multithreaded *kmeans*. The implementation of *kmeans* that we use frequently creates and joins threads. Internally, `pthread_join` invokes memory unmapping, which in turn causes a TLB flush and an enclave exit. Correspondingly, the more threads *kmeans* uses, the more AEXs appear and the higher is the overhead.

Case Study: Nginx. To illustrate the impact of Varys on a real-world application, we measured throughput and latency of Nginx v1.13.9 [23] using `ab` [28]. Nginx was running on the same machine as previous experiments, and the load generator was connected via a 10 GB network. The results are presented in Figure 5.9.

In line with the previous measurements, Varys reduces the maximum throughput by 19% if the system is configured for a low AEX rate. Otherwise, the AEX rate becomes higher, cache flushing has to happen more frequently and the overhead increases. The higher rate comes from two sources: disabling DynTicks increases the frequency of timer interrupts and disabling interrupt redirection adds exits caused by network interrupts. Finally, the "Over-assignment" line is the throughput of Nginx in the scenario, when we do not dedicate a core exclusively to Nginx and assign another application that competes for the core (in our case, we use *word_count* from Phoenix). Since the Nginx threads are periodically suspended, the cost of the handshake becomes much higher as both threads in a pair have to wait while one of them is suspended.

[3]This experiment was performed on a CPU with SGXv1, which has a small EPC size (128 MB). Newer versions of SGX have larger EPC sizes and, hence, this issue is less pronounced on them.

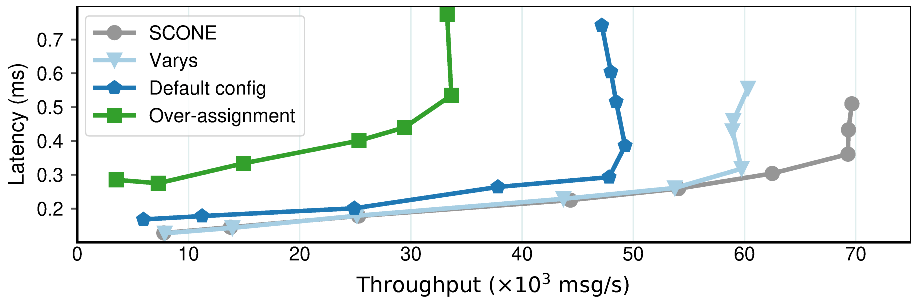


Figure 5.9: Throughput-latency plots of Nginx. Varys: low-exit system configuration, Default config: default configuration of Linux, Over-assignment: another process is competing for a core with Nginx.

Time threshold, ticks	**False positives, %**	**False negatives, %**
130	90.1	0.0
140	4.0	0.0
160	0.0	0.0
250	0.0	0.1

Table 5.1: Rate of false positives and false negatives depending on the value of handshake threshold. The threshold is presented for 10 memory accesses.

5.6.3 Security Evaluation

Violation of Trusted Reservation. To evaluate how effective Varys is at ensuring trusted reservation (i.e., if a pair of threads is running on the same physical core), we performed an experiment that emulates a time-sliced attack. We launched a dummy Varys-protected application in normal configuration (all threads are correctly paired) and then, at runtime, change affinity of one of the threads. Additionally, to evaluate the rate of false positives, we ran the application without the attack. As trusted reservation is implemented via a periodic handshake, the main configuration parameter is the time limit distinguishing cache hits from cache misses.

The results are presented in Table 5.1. False negatives represent the undetected attacks and false positives—the cases when there was no attack, but a handshake still failed. The results are aggregated over 1000 runs.

As we see, trusted reservation can be considered reliable if the limit is set to 160 ticks of the software timer (§5.5.2). The fact that we neither have false positives nor false negatives is caused by the difference in timing of L1 and LLC cache hits. If the threads are on the same core, the handshake will have timing of 10 L1 cache hits. Yet, if they are on different cores, the only shared cache is LLC and all 10 accesses would miss both L1 and L2.

Increased Rate of AEX. To evaluate Varys's effectiveness at detecting attacks with high

MRT, IR instructions	**Normal execution**	**Low-AEX attack**	**Common attack**
60M	0.2%	100%	100%
62M	1.2%	100%	100%
64M	10%	100%	100%

Table 5.2: Varys abort rate depending on the system interrupt rate and on the value of minimum runtime (MRT).

AEX frequencies, we ran a protected application under different system interrupt rates, and counted the number of aborts (i.e., detected attacks). For the purity of the experiment, the victim was a dummy program that does not introduce additional AEXs on top of the system interrupts. In each measurement, we tested several limits on the minimal runtime (MRT), which is inverse of the AEX rate. As in the previous experiment, we had 1000 runs.

The results are presented in Table 5.2. We tested three setups:

- "Normal execution" represents an attack-free environment, with an interrupt rate of 100 Hz (§5.3.3);
- "Low-AEX attack" represents stealthy attacks from Wang et al. [254] and Van Bulck et al. [245], which cause an interrupt rate of 5.5 kHz;
- "Common attack" represents a typical cache attack, with an interrupt rate of 10 kHz.

We can see that if we set the threshold on the number of IR instructions between enclave exits to 60 millions, it achieves both low level of false positives (0.2%) and detects all simulated attack attempts.

Residual Cache Leakage. For small applications (i.e., applications with small or slowly changing working set), cache leakage may persist even after we limit the frequency of enclave exits.

As a worst case, we consider the application on Figure 5.10: It iterates over cache sets, accessing all cache lines in a set for a long time. With such applications, limiting the interrupt frequency will not help, because even a few samples are enough to derive the application state. We use this application to evaluate effectiveness of the cache eviction mechanism proposed in §5.4.2.

We used a kernel module to launch a time-slicing cache attack on the core running the victim application. The attack delivers an interrupt every 10 ms, and does an L1D cache measurement on all cache sets. We normalize the results into the range of $[0,1]$. Additionally, we disable CPU prefetching both for the victim and attack code to reduce noise. Essentially, it is a powerful kernel-based attack that strives to stay undetected by Varys.

The results of the measurements are on Figure 5.11. Without eviction (Figures 5.11a

```
for (Set in L1_Cache_Sets):
    for (Very Long):
        for (CLine in CacheLine1..CacheLine8):
            Read(Set, CLine)
```

Figure 5.10: Worst-case victim for a defense mechanism based solely on interrupt frequency.

and 5.11b), the attack succeeds and the state of application can be deducted even with a few samples. Then, we apply Varys with L1I and L1D cache eviction to the application (Figure 5.11c). Even though the amount of information leaked decreases greatly, we can still distinguish some patterns in the heatmap due to residual L2 cache leakage. When we enable L2 eviction in Varys, the results contain no visible information about the victim application (Figure 5.11c).

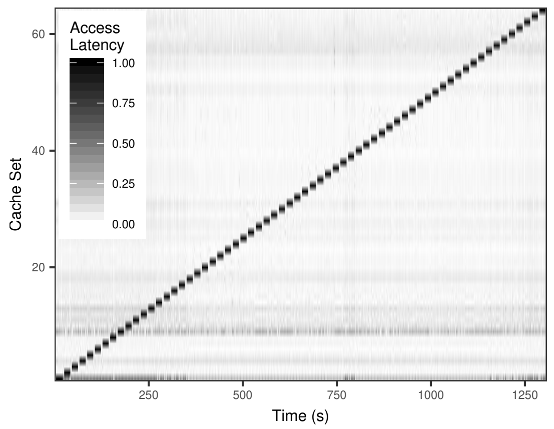
5.7 Hardware Extensions

Many parts of Varys's functionality could be implemented in hardware to improve its efficiency and strengthen the security guarantees. In this section, we propose a few such extensions. We believe that introducing such a functionality would be rather non-intrusive and should not require significant architectural changes.

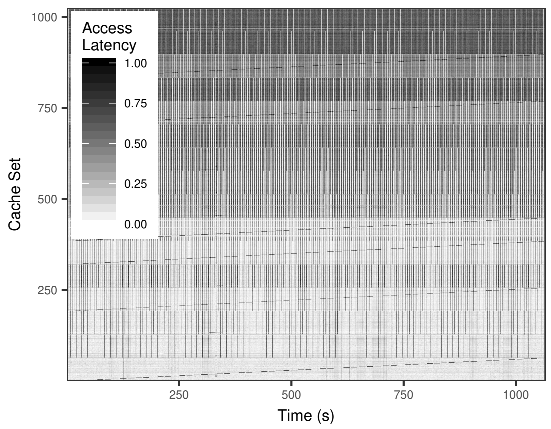
5.7.1 Userspace AEX Handler

Varys relies on the SGX state saving feature for detection of enclave exits. However, this approach has certain drawbacks: it requires the application to monitor the SSA value, thus increasing the overhead, and it introduces a window of vulnerability (§5.5.1). An extension to the AEX protocol could solve both of the issues.

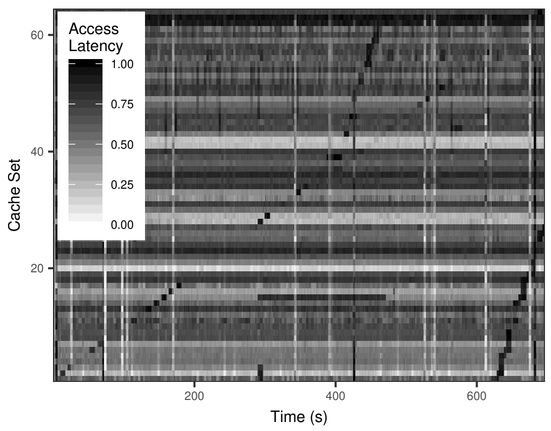
Normally during an AEX, the control is passed to the OS exception handler, which further transfers control to the userspace AEX handler, provided by the user. The user AEX handler then executes `ERESUME` instruction, which re-enters the enclave. However, there is no possibility for an in-enclave handler. Our proposed extension adds a hardware triggered callback to the `ERESUME` instruction, specified in the `TCS: TCS.eres_handler`. After each `ERESUME` executed by unprotected code, the enclave is re-entered, and the control is passed to code located at the address `TCS.eres_handler`. To continue executing interrupted in-enclave code, the `ERESUME` handler will execute the `ERESUME` instruction once again, this time, inside the enclave. Note that calling `ERESUME` inside of an enclave is right now not permitted. One difficulty of this extension would be an AEX during the processing of a handler. We would allow recursive calls since handlers could be designed to deal with such recursions.



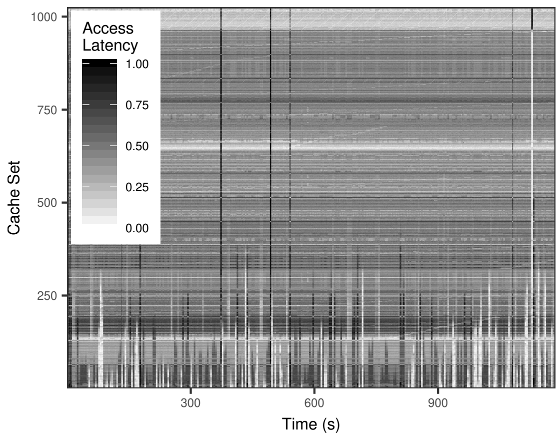
(a) L1 attack: no eviction.



(b) L2 attack: no eviction.



(c) L1 attack: L1 eviction.



(d) L2 attack: L2 eviction.

Figure 5.11: An experiment proving the effectiveness of cache eviction. Without eviction, we can easily see the program behavior. With L1 eviction, the L2 residual leak exposes some information. With L2 eviction, no visible information is exposed. Graphs have different time scales due to different overhead from L1/L2 measurement and presence of eviction mechanism. Color reflects normalized values, with different absolute minimum and maximum values for every graph.

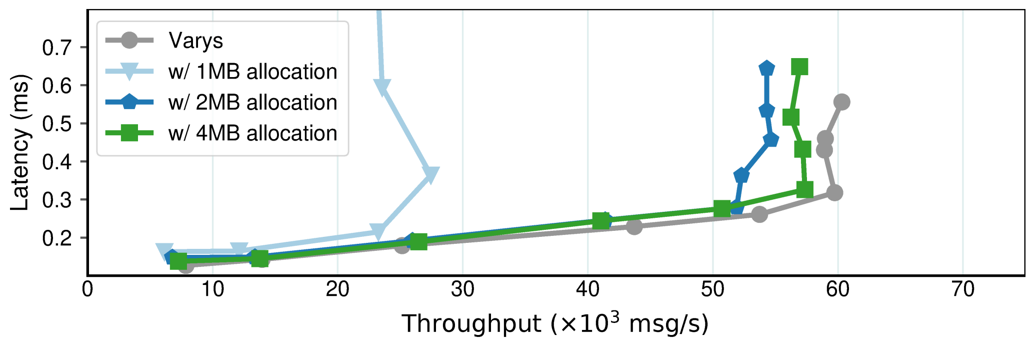


Figure 5.12: Impact of different cache allocation sizes on throughput and latency of Nginx protected by Varys.

Recently (after the publication of our original paper), Intel introduced User Interrupts, a feature that allows userspace applications to directly handle interrupts. This feature relies on OS assistance, thus it is still not applicable to the SGX threat model.

5.7.2 Intel Cache Allocation Technology

Although Intel CAT could be used to prevent concurrent LLC attacks, the OS has complete control over the CAT configuration, which renders the defense ineffective. It can be solved by associating the CAT configuration registers with *version numbers* that are automatically incremented each time the configuration changes. The application could check the version number in the AEX handler after each AEX and thus easily detect the change. In case, no support for AEX handlers is added, the application could perform periodic checks within the enclave instead.

To estimate the potential impact of the extension, we ran an experiment where Nginx was protected by Varys and had a slice of LLC exclusively allocated to it (Figure 5.12). As we see, allocating 4 and 2 MB of cache did not cause a significant slowdown for the given workload. The difference in throughput comes mainly from the larger eviction region: Varys had to flush 4 MB instead of 256 KB. However, allocating this large part of the cache can significantly reduce the overall system performance. At the same time, if we try a more modest allocation, we risk causing a much higher rate of cache misses, which is what happened with the 1 MB allocation in our experiment.

5.7.3 Trusted HW Timer

Since the hardware timer (`RDTSC/P` instruction) is not available in SGX1, we used a software timer, which wastes a hyperthread. More recent SGX2 supports the timer, but we cannot rely on it either as privileged software can overwrite and reset its value.

We see two ways of approaching this problem: We may introduce a monotonically increasing read-only timer which could be used as-is. Alternatively, we could introduce a version number that is set to a random value each time the timer is overwritten. To ensure the timer correctness, the application would have to compare the version of this register before and after the measurement.

5.8 Discussion

Security Guarantees. Varys provides complete protection against all L1 and L2 cache attacks thanks to trusted reservation. As a by-effect, trusted reservation also prevents concurrent attacks on on-core resources such as branch predictor, floating point unit, etc. Concurrent LLC attacks could be prevented with an extension to Intel CAT (§5.7.2). Higher levels of memory hierarchy are not guarded by Varys. This includes, among others, attacks on DRAM buffers and memory bus snooping (§2.3.4).

Availability. A potential drawback of our approach is its impact on availability. Our default policy upon detection of a potential attack—termination—might be an issue in a noisy environment that creates false positives. If we run programs with the help of Docker Swarm (as SCONE supports), Docker Swarm will restart the program. After repeated failures, it will restart on a different machine.

We presented Varys, an approach to runtime validation of side-channel protected environments within SGX enclaves. Varys protects from multiple side channels and reliably detects violations of the protected environment, all while causing low overheads. With additional hardware support, we would not only expect a more straightforward implementation of Varys but also lower overhead and protection against a wider range of side channel attacks.

Chapter 6

Conclusion

This thesis introduced several methods that enable testing and runtime validation of defences against modern microarchitectural attacks. We conclude with a brief summary of each technique, with a description of their impact so far, and with the potential directions for future work.

6.1 Summary of Techniques

Model-based Relational Fuzzing allows us to check if the microarchitecture of a CPU under test exposes as much information as we expect it to expose. The idea is to run random instruction sequences on the tested CPU and on its reference model, collect the side-channel traces on both, and compare the amount of information exposed by the traces. The tool implementing this approach, Revizor, managed to automatically surface several known speculative vulnerabilities and detect two new variants.

Speculation Exposure allows us to find instances of speculative vulnerabilities with dynamic software testing. The idea is to simulate speculative execution in software and, thus, transform speculative security violations into normal ones, detectable with traditional integrity checking mechanisms. We implemented the idea in SpecFuzz, a fuzzing framework that automatically detects instances of Spectre V1. It successfully detected hundreds of V1 instances in several real-world libraries, and its results allowed us to significantly reduce the cost of Spectre V1 mitigations.

Enclave Self-Monitoring enables runtime validation of the side-channel protected environments created by untrusted system software. The idea is to modify the target application such that it continuously monitors its execution environment in search of signs of a misbehaving OS or an ongoing attack. We implement it in Varys, a tool that works with Intel SGX enclaves and monitors system defences against L1/L2 cache side-channel attacks and page attacks. We show that Varys successfully detects when the defences are compromised, and it can prevent the attacks.

6.2 Impact

SpecFuzz:

- Two papers built on top of SpecFuzz and extended it with more precise analysis of vulnerabilities: SpecTaint [195] used dynamic taint analysis to filter out uncontrolled vulnerabilities; FastSpec [235], in one of its case studies, post-processed the vulnerabilities detected by SpecFuzz with a Generative Adversarial Network.
- A few papers used SpecFuzz as a point of comparison, including KLEESpectre [251] and the work of Guo et al. [116].
- SpecFuzz took the first place in CSAW Applied Research Competition (Europe) 2020.

Varys:

- Three papers built on top of the ideas of Varys: S-FaaS [22] used trusted core reservation; Autarky [186] extend the idea of AEX detection; T-Lease [237] used the CPU frequency validation mechanism.
- Several papers assumed a Varys-like technique in their threat model, e.g., Replica-TEE [227], Occlum [221], and EnclaveDom [171].
- Keystone [154] used Varys as a point of comparison.

Revizor is under submission at the time of writing, thus it has had no impact so far.

6.3 Future Work

Our research showed the feasibility of testing against modern microarchitectural attacks, but many challenges remain unsolved to make it practical and widely applicable.

End-to-End Guarantees. The tools presented in this thesis work independently of each other. This would be insufficient when the security of a system relies on multiple defence techniques implemented at different abstraction layers. Instead, we need end-to-end security guarantees, which requires a unified specification method for the guarantees and end-to-end testing mechanisms. To get such a unified specification, the speculation contracts framework

(§3.2.2) has to be extended to system- and application-layer defences. To perform end-to-end testing, we need to connect the results of Revizor and SpecFuzz and implement a tool for similar system-level testing. In addition, Varys would need to be extended to work with the new contract-like specification.

Systematic Testing of Existing Defences. A natural step after implementing the testing tools is to apply them to a wide range of real-world applications. The numerous existing defence proposals (Chapter 2) often have uncertain or untested security guarantees, and our tools can test them. As such, the security community could benefit from an evaluation study that empirically validates the guarantees of existing defences.

Microarchitectural Fuzzing. The implementation of Revizor was only the first step into practical testing of microarchitectural security properties, and many questions are still open. The most important one is the generation method: Revizor generates both test cases and inputs to them randomly, which hinders detection of "deep" vulnerabilities. More precise techniques, such a feedback-driven mutational fuzzing, could significantly improve the chances of detection. Related to it is the notion of coverage, which is still only approximate in our tool, and it needs to be further developed. Revizor also needs to be ported to other architectures to truly serve its purpose of comparing security of different CPUs. We hope that the open-source release of Revizor will provide a solid foundation for this future work.

Contract-based Software Fuzzing. Currently, SpecFuzz supports only one type of speculative vulnerabilities—Spectre V1. Real-world applications would require extension to other vulnerabilities as well. The best option would be to re-implement SpecFuzz such that it could interpret the speculation contracts, and the user would be able to directly control the CPU model to be tested against.

System Defence Validation. Varys is tightly coupled with Intel SGX, even though the problem of unreliable defences is common across all TEEs. As such, Varys needs to be ported to other TEEs. In case of open-source TEEs (e.g., Keystone [154]), it creates an opportunity [266] to implement the missing hardware extensions, as discussed in §5.7.

6.4 Code Availability

The source code of several presented tools is publicly available:

- Revizor: `https://github.com/hw-sw-contracts/Revizor`
- SpecFuzz: `https://github.com/tudinfse/SpecFuzz`
- Fex: `https://github.com/tudinfse/fex`

Appendix A

Fex: Systems Evaluation Framework

As a by-effect of working on this thesis, we developed an evaluation framework, called Fex, which we used in Chapter 5 and Chapter 4. Fex started as a small set of frequently reused scripts and quickly matured in a full-fledged framework. During years of internal usage, we refactored and expanded Fex to accommodate our growing needs, until it turned into a stand-alone contribution. The end result is presented in this appendix.

The key goals of developing Fex were to make it *extensible* (can be easily extended with custom experiment types), *practical* (supports composition of different benchmark suites and real-world applications), and *reproducible* (it is built on container technology to guarantee the same software stack across platforms).

Out-of-the-box, Fex is integrated with several well-known benchmark suites (SPLASH, Phoenix, PARSEC), standalone programs (Apache, Memcached, Nginx), compilers (GCC, Clang), and measurement tools (perf, time). Internally, Fex is comprised of a number of tools for automating installation, building and running benchmarks, collecting logs, and plotting the final results.

Going forward, we discuss the architecture of the framework, explain its interface, and show common usage scenarios.

A.1 System Interface

Fex was developed with reproducibility as one of the main goals. Therefore, we prepare the environment and run all experiments in a Docker container in such a way that they are as independent from the actual host system as possible [172, 48]. The Docker image contains a bare minimum to run the experiments: sources for all programs in benchmark suites with corresponding makefiles, Bash scripts for environment setup, Python scripts to actually perform experiments and to aggregate and plot their results. Note that the packages put

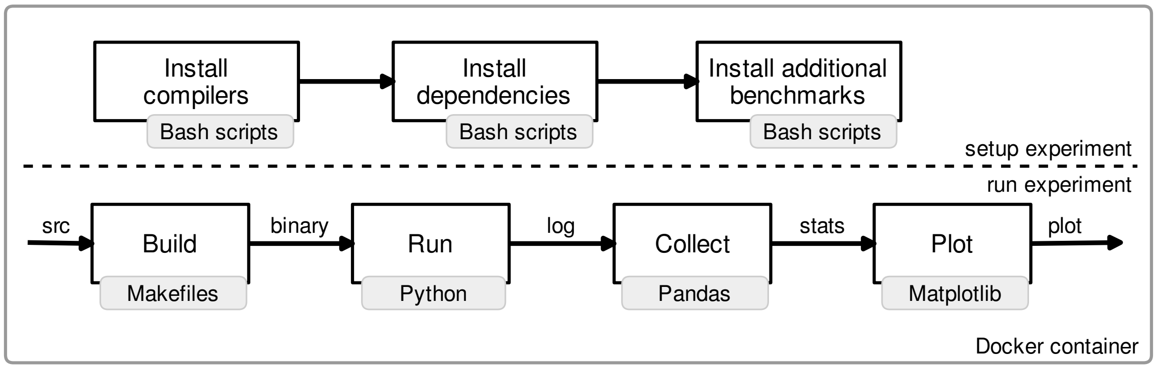


Figure A.1: System interface of Fex.

in the image—git, python3, wget, perf, etc—are used by the framework itself and do *not* influence the experiments (i.e., they do not affect measurements neither through the build system nor through dynamically linked libraries).

Figure A.1 shows the general workflow and the exposed system interface of Fex. Any of the actions in the workflow can be executed via call to the framework's entry point—`fex.py` file:

```
>> fex.py <action> -n <name> [other_arguments]
```

For example, running Phoenix benchmark suite with GCC will look like this:

```
>> fex.py run -n phoenix -t gcc_native
```

The workflow is divided into two stages: setup and run. The first stage prepares an environment for the second stage by installing all the necessary components from the Internet.

Experiment Setup. The Docker image we ship contains only the source codes of benchmarks and a set of scripts to build and run them. The actual dependencies—compilers to build, shared libraries to link against, additional tools and benchmarks—are downloaded from the Internet and installed at the experiment setup stage. The reasons for this flow are twofold. First, the Docker image would swell to approx. 17GB in size[1] if all dependencies would be built-in. Images of such size would be cumbersome to distribute. Second, this allows the end user to install only those dependencies and only those versions needed for her experiments. (For reproducibility, it is important that the exact versions of software crucial for experiments are installed.)

For simplicity, the installation scripts are written in Bash (they can also be written in Python). To run an installation script, Fex provides an "install" command. The following

[1]Our current image is 1.04GB, with 122MB Ubuntu files, 300MB of benchmarks' source files, and the rest helper packages

example installs GCC 6.1.

```
>> fex.py install -n gcc-6.1
```

As shown in Figure A.1 (top), this stage includes three steps:

- *Installing compilers* with specific versions is a prerequisite. Fex cannot rely on Linux default package managers to automatically install required compilers, e.g., APT or RPM, because compiler versions in their repositories change over time and thus hinder reproducibility.
- *Installing dependencies* implies tools required for the build process or for specialized measurements. For example, several PARSEC benchmarks require `gettext` system for Autoconf—this software does not affect performance but is simply needed to resolve all build dependencies of a particular benchmark. These tools are optional and may not be needed for simple experiments.
- *Installing additional benchmarks* may be necessary to perform experiments on large unmodified programs. Fex encourages to put program sources in its repository; this simplifies changing and tweaking original code to the user's needs. However, sometimes it is simpler to fetch the sources from elsewhere. For example, we install Apache and Nginx in this way because we want to experiment with their different versions (those that are vulnerable to a particular bug and those that are not).

Experiment Runs. After installing all prerequisites, users can start running experiments. All experiments are usually performed as a sequence of steps depicted in Figure A.1 (bottom):

- The *build* step is performed once before running each benchmark in the experiment. Fex consults the makefile corresponding to the benchmark-to-run and puts a final binary in the `build` directory. It is important to re-build all benchmarks for each experiment, otherwise a mix of old and new compilation flags and/or libraries could skew the results. For quick preliminary experiments, the build step can be omitted via `--no-build` flag.
- The *run* step is the experiment itself. Fex includes a Run component, which provides several Python hooks to specify a list of benchmarks-to-run with their inputs and to control how exactly these benchmarks are started. For example, we implement an additional "dry run" for Phoenix benchmarks using a `per_benchmark_action` hook. Multithreaded benchmarks are automatically run with a set of number of threads specified in the command line, e.g., `-m 1 2 4`.

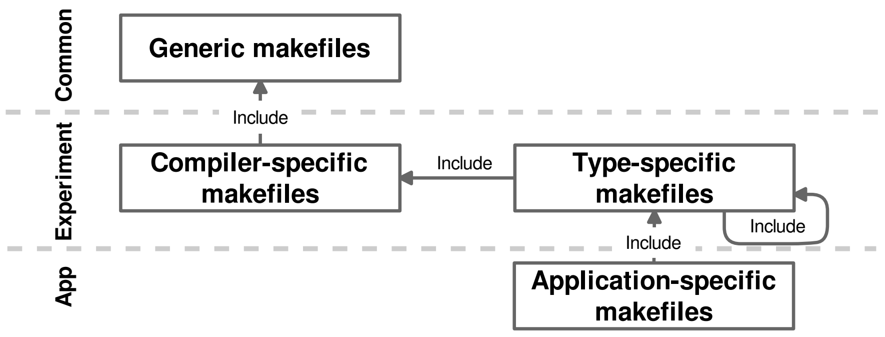


Figure A.2: Build system hierarchy.

- The *collect* step parses the log, extracts the measurement results, processes them in a user-specified way, and stores into a CSV table. We use Pandas Python library [169] for efficient data analysis and aggregation.

- The *plot* step is performed after the whole experiment is finished. It is usually performed on a local user machine. The powerful matplotlib [124] is used to emit different kinds of plots. Again, the user is provided with hooks to change the appearance of emitted plots.

Building and running the benchmarks is sensitive to environment variables. Fex provides a convenience wrapper to specify default variables for these steps (see §A.2).

A.2 System Architecture

Following the workflow presented in §A.1, Fex consists of 4 subsystems: build, run, collect, and plot. The later two have plain structure which does not require any explanations. The former two, however, are slightly more complex and we will discuss them in detail.

Build subsystem. In the build stage, two scenarios are possible: (1) a single application can be built many times with varying build parameters or (2) the same parameters can be reused for many different applications. To aid this variability, our build subsystem was divided into three layers (see Figure A.2): common, experiment, and application layers.

Common layer contains parameters that are applicable to all benchmarks and all build types. This includes, for example, optimization levels, debugging information (if enabled), common compilation flags and generic compilation targets.

Experiment layer is responsible for parameters of the current build type. For example, if a benchmark suite has to be built by GCC and with enabled AddressSanitizer, the makefiles will set `CC` variable to `gcc` and `CFLAGS` to `-fsanitize=address`. Note that there might be

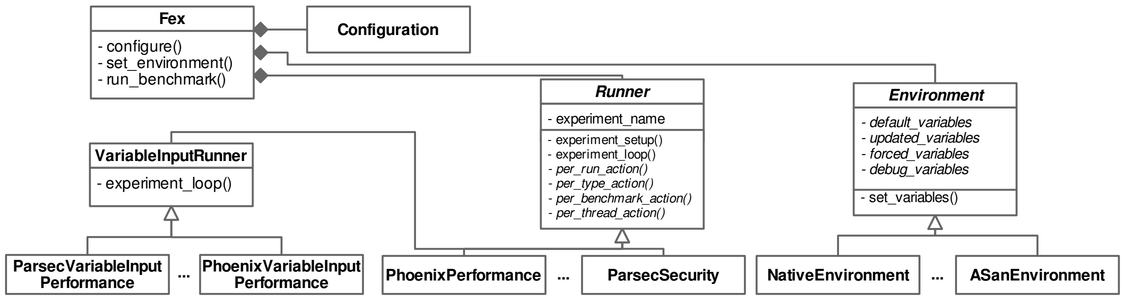


Figure A.3: Class diagram of the experiment infrastructure.

multiple levels of makefiles in this layer: some may set parameters that are applicable to all configurations of a single compiler, and the others will refine them to a concrete configuration.

Finally, the *application* layer defines the structure and the procedure of the build. It specifies the location of source files, lists dependencies, and sets application-specific flags.

The overall build system is structured in such a way that these layers can be replaced independently of each other. Accordingly, any application can be compiled with any of the existing build configurations without additional efforts.

Experiment runners. When an experiment is started via

```
>> fex.py run ...
```

a new instance of the Fex class is created (see Figure A.3). This object controls the overall experiment execution. Firstly, it retrieves a configuration file and sets experiment parameters accordingly. Then, it sets environment variables to the necessary values by instantiating child classes of the Environment abstract class. In the end, it instantiates and calls the child of the Runner class that corresponds to the current experiment. This new Runner object will perform the actual experiment.

Since environmental variables can vary in accordance to parameters of the given experiment (e.g., when debug mode is turned on), we define four types of the variables:

1. Default: the default values of environment variables.
2. Updated: the values of this type are appended if the variable exists, and assigned otherwise.
3. Forced: the variables are overwritten regardless of the previous value.
4. Debug: the values are set only in the debug mode.

Note that the order is important here: each next type has higher priority that the previous one. For example, if variable `BIN_PATH` is assigned to `/usr/bin/` among default variables and to `/home/usr/bin/` among the forced ones, the end value will be `/home/usr/bin/`. On top of it, if a user wants to add another type, she can do it simply by writing a subclass of

```
for each build type:
  self.per_type_action(type)
  for each benchmark:
    self.per_benchmark_action(type, benchmark)
    for each thread count:
      self.per_thread_action(type, benchmark, thread_num)
      for i in range(0, number_of_repetitions):
        self.per_run_action(i)
```

Figure A.4: Experiment loop

Environment and redefining the `set_variables` function.

The key element of the Runner class is `experiment_loop` function. For each of the execution parameters, it iterates over all their values by going through a series of nested loops, as shown in Figure A.4. For example, the outermost loop may go through GCC and Clang compilers, the next one—through Nginx, Apache, and Memcached applications, and so forth. Each of the loops has a hook that can be implemented in a subclass. This way, the overall structure of the experiment stays the same, but the concrete actions can be tailored to the needs of the given experiment. Moreover, if even more parameters would be necessary, the `experiment_loop` can be redefined or extended in a subclass, as VariableInputRunner does in Figure A.4.

A.3 Fex Details and Workflow

A.3.1 Creating new experiments

In this section, we explain how Fex facilitates creation of new experiments and evaluation of new benchmark suites and standalone programs. We also detail the implementation of Fex with the help of its standard directory layout (similar to projects like Jekyll [134], Fex assumes a specific directory tree structure).

One (simplified) example of a directory tree is shown in Figure A.5. Here, the end user sets up the environment to evaluate the performance overhead of Google's AddressSanitizer [218] on the Phoenix benchmark suite [200] and on Apache web server [2]. Moreover, for reproducibility she chooses GCC version 6.1 which comes with AddressSanitizer by default.

First, she needs to write installation scripts to install the GCC 6.1 compiler, download input files for the Phoenix benchmark, and install an additional Apache benchmark. For convenience, Fex provides a set of functions for frequently used operations, found in `install/common.sh`, e.g., `download`.

Next, the user must create compiler-specific and type-specific makefiles for different experiment variants and put them under `makefiles/`. The compiler-specific `gcc_native.mk` file would look like:

```
include common.mk
CC := gcc
CXX := g++
```

The type-specific file `gcc_asan.mk` would include the previous file and additionally enable AddressSanitizer:

```
include gcc_native.mk
CFLAGS += -fsanitize=address
LDFLAGS += -fsanitize=address
```

After that, the user must put application-specific makefiles and sources of the Phoenix benchmark suite and Apache web server under `src/`. To keep directories clean, standalone programs like Apache are put in a separate subdirectory named `applications/`. In case of Apache, the sources are downloaded from the Internet using an installation script, thus the only file required is a Makefile. In case of Phoenix, all sources are copied in the Fex directory tree for convenience; we only show `histogram` for the sake of clarity. Application-specific makefiles are generally simple and follow this pattern on the `histogram` example:

```
NAME := histogram
SRC := histogram-pthread
include Makefile.$(BUILD_TYPE) # includes type-specific makefile
all: $(BUILD)/$(NAME) # build target
```

Finally, the user describes the experiments themselves. The Phoenix performance-overhead experiment is put under `experiments/phoenix`. The only required file here is `run.py` which describes benchmarks with their command-line arguments to be run and measured. Additionally, each Phoenix benchmark needs a preliminary dry run: this functionality is implemented through a `per_benchmark_action` hook. Note that most of the functionality to build and run benchmarks is actually inherited from the abstract class implemented in `experiments/run.py`.

There are no specific collect and plot scripts for Phoenix. Instead, since the user has no ad-hoc requirements for them, the generic `collect.py` and `plot.py` are re-used. Also note that Figure A.5 does not show Apache experiment files for simplicity.

Some variants of the experiment may require setting specific environment variables. For example, AddressSanitizer can be fine-tuned via runtime flags in the `ASAN_OPTIONS` variable.

For this, the user shall add this flag in `environment.py`. In addition, the user can modify parameters for collection and plotting of results in a `config.py` file.

In the end, to add a new experiment with new compiler types and new benchmarks, the user needs to create (some of) the following files: (1) installation scripts, (2) compiler- and type-specific makefiles, (3) sources and makefiles for benchmarks, and (4) experiment descriptions. In fact, all of the scripts are already available in the repository of Fex.

Additionally, Fex facilitates creation of tests—short runs of benchmarks with tiny inputs. These tests can be accessed via `-i test` and are useful to check if user-defined makefiles, source files, and other scripts are written correctly. This functionality is implemented inside `run.py` files.

A.3.2 Running new experiments

Now that the experiment description is finished, the user can re-build the Docker container using `Dockerfile` and deploy it on a test server. The actual experiment proceeds in two stages as shown in Figure A.1.

Inside the container, the user sets up the experiment by invoking the relevant installation scripts:

```
>> fex.py install -n gcc-6.1
>> fex.py install -n phoenix_inputs
>> fex.py install -n apache
```

Next, it is sufficient to call the generic all-in-one “run” command like this:

```
>> fex.py run -n phoenix -t gcc_native gcc_asan
```

This command will build all Phoenix benchmarks using native and AddressSanitizer GCC versions, run them once (with a preliminary dry run), collect statistics from logs, and aggregate and save final data in a CSV table. There are several command-line flags to fine-tune the experiment. The user can specify `-v` for verbose output and `-d` to build debug versions of benchmarks. To increase the number of runs of each benchmark, the user adds `-r 10`. To run benchmarks with different numbers of threads, the `-m 1 2 4` flag must be added. To run only one benchmark from the benchmark suite, the user specifies `-b histogram`.

Note that the final binaries of benchmarks are put under `build/` directory, see Figure A.5. Sometimes it is useful to run the binary directly from there, e.g., to debug spurious errors or to perform additional quick measurements.

After the experiment is finished, the user should fetch the final CSV results from the

```
Fex
├── Dockerfile
├── fex.py
├── environment.py
├── config.py
├── install ............................................ installation scripts to prepare environment
│   ├── compilers
│   │   └── gcc-6.1.sh
│   ├── dependencies
│   │   └── phoenix_inputs.sh
│   ├── benchmarks
│   │   └── apache.sh
│   └── common.sh
├── makefiles .............................................. makefiles for different variants to test
│   ├── common.mk
│   ├── gcc_native.mk
│   └── gcc_asan.mk
├── src .................................................... makefiles and sources for benchmarks
│   ├── applications
│   │   └── apache
│   │       └── Makefile
│   └── phoenix
│       └── histogram
│           ├── Makefile
│           └── [...source files...]
├── experiments ............................................ scripts to run-parse-plot results
│   ├── phoenix
│   │   └── run.py
│   ├── collect.py
│   ├── plot.py
│   └── run.py
└── build .................................................. automatically generated final binaries
    └── phoenix
        └── histogram
            ├── gcc_native
            │   └── [...]
            └── gcc_asan
                └── [...]
```

Figure A.5: Example directory tree of Fex.

server and run the "plot" command locally:

```
>> fex.py plot -n phoenix -t perf
```

This builds the "perf" (performance overhead barplot) graph and saves it in a PDF file.

A.3.3 Currently supported experiments

The following list contains the currently supported benchmarks, compilers and compilation types, and experiments:

- Benchmark suites: Phoenix, SPLASH, PARSEC, SPEC CPU2006, RIPE.
- Real-world applications: Apache, Nginx, Memcached, micro.
- Compilers: GCC, Clang/LLVM.
- Types: AddressSanitizer (as example).
- Experiments: Performance and memory overheads, security evaluation.
- Tools: perf-stat (generic), perf-stat (memory), time.
- Plots: Lineplot, regular barplot, stacked barplot, grouped barplot, stacked-grouped barplot.

Fex supports four benchmark suites: Phoenix [200], SPLASH [207], PARSEC [45], and SPEC CPU2006[2] [120]. The reasons to support these suites are as follows. SPEC CPU2006 is a de facto standard to test new static and runtime instrumentation techniques in the research community. However, it only supports single-threaded programs and is rather old by computer science standards. Phoenix and PARSEC are two widely used multithreaded benchmarks, stressing two different computer subsystems. Phoenix benchmarks are simple and are mostly limited by memory, whereas PARSEC ones are of production quality and mostly limited by CPU.

Additionally, Fex comes with Apache, Memcached, and Nginx programs that showcase real-world usage scenarios. They are installed via scripts and *not* put under `src/`. Fex also provides several statically linked libraries like libevent and OpenSSL, required for at least one of the above benchmarks. Lastly, we wrote a suite of microbenchmarks—e.g., reading from an array—that can be useful for debugging purposes.

[2]SPEC CPU cannot be made publicly available and will not be open-sourced as part of Fex

Fex provides installation scripts and makefiles for GCC version 6.1 and Clang/LLVM 3.8.0 [153]. It is easy to update these scripts to install newer versions of these compilers. As of type-specific makefiles, the current version of the framework includes only AddressSanitizer as an example.

The list of supported experiments includes (1) performance- and memory-overhead experiments as well as variable-inputs experiments of Phoenix, PARSEC, and SPEC, and (2) throughput-latency and security experiments of Apache, Nginx, and Memcached.

For plotting, Fex provides the following generic plots: barplot (e.g., for performance and memory overheads), lineplot (for multithreading overheads), stacked barplot, grouped barplot, and stacked-and-grouped barplot (for complicated statistics such as cache misses at different levels).

A.4 Limitations

Currently, Fex has a number of limitations. The framework provides no statistical analysis functionality (except basic statistics such as standard deviation).

Fex supports only single-machine experiments. We are investigating ways to build distributed experiments, e.g., using the Fabric library.

Finally, since we rely on the Docker infrastructure, we do not guarantee reproducibility on levels below user space, i.e., on different hardware setups or across different kernel versions. However, Fex outputs various environment details, so that the complete experimental setup is stored in the log file.

Appendix B

Complete Contract Specifications

This appendix details all contracts used in Chapter 3. The concrete instructions included in each class depend on the target instructions set.

B.1 *MEM-SEQ*

```
observation_clause: MEM
  - instructions: MemReads
    format: READ (ADDR), DEST
    observation: expose(ADDR)
  - instructions: MemWrites
    format: WRITE SRC, (ADDR)
    observation: expose(ADDR)
execution_clause: SEQ
  None
```

B.2 *MEM-COND*

```
observation_clause: MEM
  - instructions: MemReads
    format: READ (ADDR), DEST
    observation: expose(ADDR)
  - instructions: MemWrites
    format: WRITE SRC, (ADDR)
    observation: expose(ADDR)
execution_clause: COND
  - instructions: CondBranches
    format: COND_BR CONDITION (DEST)
    execution: {  # branch misprediction
      if NOT CONDITION then
          IP := IP + DEST
      fi
    }
```

B.3 *CT-SEQ*

```
observation_clause: CT
  - instructions: MemReads
    format: READ (ADDR), DEST
    observation: expose(ADDR)
  - instructions: MemWrites
    format: WRITE SRC, (ADDR)
    observation: expose(ADDR)
  - instructions: CondBranches
    format: COND_BR CONDITION (DEST)
    observation: {   # expose branch targets
      if CONDITION then
          expose(DEST)
      else
          expose(IP)
      fi
    }
execution_clause: SEQ
  None
```

B.4 *CT-COND*

```
observation_clause: CT
  - instructions: MemReads
    format: READ (ADDR), DEST
    observation: expose(ADDR)
  - instructions: MemWrites
    format: WRITE SRC, (ADDR)
    observation: expose(ADDR)
  - instructions: CondBranches
    format: COND_BR CONDITION (DEST)
    observation: {
      if CONDITION then
          expose(DEST)
      else
          expose(IP)
      fi
    }
execution_clause: COND
  - instructions: CondBranches
    format: COND_BR CONDITION (DEST)
    execution: {
      if NOT CONDITION then
          IP := IP + DEST
      fi
    }
```

B.5 *CT-COND*-NonspeculativeStore

```
observation_clause: CT-NonspeculativeStore
  - instructions: MemReads
    format: READ (ADDR), DEST
    observation: expose(ADDR)
  - instructions: MemWrites
    format: WRITE SRC, (ADDR)
    observation: {  # do not expose speculative stores
      if NOT IN_SPECULATION then
        expose(ADDR)
      fi
    }
  - instructions: CondBranches
    format: COND_BR CONDITION (DEST)
    observation: {
      if CONDITION then
          expose(DEST)
      else
          expose(IP)
      fi
    }
execution_clause: COND
  - instructions: CondBranches
    format: COND_BR CONDITION (DEST)
    execution: {
      if NOT CONDITION then
          IP := IP + DEST
      fi
    }
```

B.6 *CT-BPAS*

```
observation_clause: CT
  - instructions: MemReads
    format: READ (ADDR), DEST
    observation: ADDR
  - instructions: MemWrites
    format: WRITE SRC, (ADDR)
    observation: ADDR
  - instructions: CondBranches
    format: COND_BR CONDITION (DEST)
    observation: {
      if CONDITION then
          expose(DEST)
      else
          expose(IP)
      fi
    }
execution_clause: BPAS
  - instructions: MemWrites
    format: WRITE SRC, (ADDR)
    execution:
      # speculatively skip stores
      IP := IP + instruction_size()
```

B.7 *CT-COND-BPAS*

```
observation_clause: CT
  - instructions: MemReads
    format: READ (ADDR), DEST
    observation: expose(ADDR)
  - instructions: MemWrites
    format: WRITE SRC, (ADDR)
    observation: expose(ADDR)
  - instructions: CondBranches
    format: COND_BR CONDITION (DEST)
    observation: {
      if CONDITION then
          expose(DEST)
      else
          expose(IP)
      fi
    }
execution_clause: COND-BPAS
  - instructions: CondBranches
    format: COND_BR CONDITION (DEST)
    execution: {
      if NOT CONDITION then
          IP := IP + DEST
      fi
    }
  - instructions: MemWrites
    format: WRITE SRC, (ADDR)
    execution:
      # speculatively skip stores
      IP := IP + instruction_size()
```

B.8 *CTR-SEQ*

```
observation_clause:
  - instructions: RegReads  # expose register values
    format: READ SRC, DEST
    observation: expose(SRC)
  - instructions: MemReads
    format: READ (ADDR), DEST
    observation: expose(ADDR)
  - instructions: MemWrites
    format: WRITE SRC, (ADDR)
    observation: expose(ADDR)
  - instructions: CondBranches
    format: COND_BR CONDITION (DEST)
    observation: {
      if CONDITION then
          expose(DEST)
      else
          expose(IP)
      fi
    }
execution_clause:
  None
```

B.9 *ARCH-SEQ*

```
observation_clause:
  - instructions: RegReads
    format: READ SRC, DEST
    observation: expose(SRC)
  - instructions: MemReads
    format: READ (ADDR), DEST
    observation: {
      expose(ADDR)
      expose(read(ADDR))  # expose loaded values
    }
  - instructions: MemWrites
    format: WRITE SRC, (ADDR)
    observation: expose(ADDR)
  - instructions: CondBranches
    format: COND_BR CONDITION (DEST)
    observation: {
      if CONDITION then
          expose(DEST)
      else
          expose(IP)
      fi
    }
execution_clause:
  None
```

Appendix C

Examples of Contract Violations

This appendix shows examples of contract counterexamples generated by Revizor. Appendix C.1 is the original generated test case, and the rest are automatically minimized versions.

C.1 Instance of Spectre V1

```
LEA R14, [R14 + 12] # instrumentation
MFENCE # instrumentation
JMP .bb0
.bb0:
CMOVNL ECX, ECX
AND RBX, 0b0111111000000 # instrumentation
ADD RBX, R14 # instrumentation
ADC dword ptr [RBX], -67100032
NOT RAX
JP .bb1 # < --------------------- mispredicted branch
JMP .test_case_main.exit
.bb1:
AND RBX, 1048197274
ADD AX, 5229
AND RCX, 0b0111111000000 # instrumentation
ADD RCX, R14 # instrumentation
ADC EAX, dword ptr [RCX] # < ----- speculative leak
{load} ADD RCX, RCX
.test_case_main.exit:
LEA R14, [R14 - 12] # instrumentation
MFENCE # instrumentation
```

C.2 Minimized Instance of Spectre V4

```
LEA R14, [R14 + 8] # instrumentation
MFENCE # instrumentation
.test_case_main.entry:
JMP .bb0
.bb0:
MOV RDX, 0 # leftover instrumentation
OR BX, 0x1c # leftover instrumentation
AND RAX, 0xff # leftover instrumentation
CMOVNZ BX, BX
AND RBX, 0b0111111000000 # leftover instrumentation
ADD RBX, R14 # leftover instrumentation
XOR AX, AX
AND RBX, 0b0111111000000 # instrumentation
ADD RBX, R14 # instrumentation
SETNS byte ptr [RBX]  # < ---------- delayed store
AND RAX, 0b0111111000000 # instrumentation
ADD RAX, R14 # instrumentation
MOVZX EDX, byte ptr [RAX]  # < ----- store bypass
AND RDX, 0b0111111000000 # instrumentation
ADD RDX, R14 # instrumentation
AND RCX, qword ptr [RDX]   # < ----- speculative leakage
LEA R14, [R14 - 8] # instrumentation
MFENCE # instrumentation
```

C.3 Minimized Instance of MDS

```
LEA R14, [R14 + 48] # instrumentation
MFENCE # instrumentation
.test_case_main.entry:
JMP .bb0
.bb0:
ADC EAX, -2030331421
AND RAX, 0b1111111000000 # instrumentation
ADD RAX, R14 # instrumentation
MOV CX, word ptr [RAX]  # < ---------- microcode assist
AND RCX, 0b1111111000000 # instrumentation
ADD RCX, R14 # instrumentation
SUB EAX, dword ptr [RCX]   # < ------- speculative leak
LEA R14, [R14 - 48] # instrumentation
MFENCE # instrumentation
```

C.4 Minimized Instance of LVI-NULL

```
LEA R14, [R14 + 8] # instrumentation
MFENCE # instrumentation
.test_case_main.entry:
JMP .bb0
.bb0:
SUB AX, 27095
AND RAX, 0b1111111000000 # instrumentation
ADD RAX, R14 # instrumentation
ADD BL, byte ptr [RAX]  # < -- speculative zero injection
AND RBX, 0b1111111000000 # instrumentation
ADD RBX, R14 # instrumentation
OR dword ptr [RBX], -1193072838 # < ---- speculative leak
LEA R14, [R14 - 8] # instrumentation
MFENCE # instrumentation
```

C.5 Minimized Instance of Spectre V1-Var

```
LEA R14, [R14 + 28] # instrumentation
MFENCE
.test_case_main.entry:
JMP .bb0
.bb0:
NOP
NOP
CDQ
SETZ CL
ADD EDX, 117
REX ADD BL, BL
SETNLE AL
SUB RBX, RBX
TEST AL, 29
MOV RDX, 0 # instrumentation
OR RBX, 0x6d # instrumentation
AND RAX, 0xff # instrumentation
DIV RBX  # < ---------------- variable-latency operation
{disp32} JNO .bb1
.bb1:
AND RCX, 0b111111000000 # instrumentation
ADD RCX, R14 # instrumentation
MOVZX EDX, byte ptr [RCX]
AND RAX, RAX
AND RAX, 0b111111000000 # instrumentation
ADD RAX, R14 # instrumentation
SBB qword ptr [RAX], 39412116
TEST ECX, ECX
AND RAX, 0b111111000000 # instrumentation
ADD RAX, R14 # instrumentation
MOV qword ptr [RAX], 81640764
REX NEG AL
CMC
OR RDX, 37323177
```

```
JNP .bb2  # < --------------------- mispredicted branch
JMP .test_case_main.exit
.bb2:
REX SBB AL, AL
SBB EAX, 74935583
AND RDX, 0b111111000000 # instrumentation
ADD RDX, R14 # instrumentation
CMOVS RDX, qword ptr [RDX]     # < ---- speculative leak
AND RAX, 0b111111000000 # instrumentation
ADD RAX, R14 # instrumentation
MOV qword ptr [RAX], 23088010
AND RBX, 0b111111000000 # instrumentation
ADD RBX, R14 # instrumentation
LOCK AND word ptr [RBX], 5518 # < ---- speculative leak
.test_case_main.exit:
LEA R14, [R14 - 28] # instrumentation
MFENCE # instrumentation
```

C.6 Minimized Instance of Speculative Store Eviction

```
LEA R14, [R14 + 32] # instrumentation
MFENCE # instrumentation
JMP .bb0
.bb0:
AND RBX, 0b0111111000000 # instrumentation
ADD RBX, R14 # instrumentation
LOCK AND dword ptr [RBX], EAX
JZ .bb1 # < --------------------- mispredicted branch
JMP .test_case_main.exit
.bb1:
AND RAX, 0b0111111000000 # instrumentation
ADD RAX, R14 # instrumentation
MOV qword ptr [RAX], 3935 # < ---- speculative leak
.test_case_main.exit:
LEA R14, [R14 - 32] # instrumentation
MFENCE # instrumentation
```

C.7 Minimized *CTR-SEQ* Violation

```
LEA R14, [R14 + 4] # instrumentation
MFENCE # instrumentation
.test_case_main.entry:
JMP .bb0
.bb0:
{store} ADC CX, CX
AND RAX, 0b0111111000000 # instrumentation
ADD RAX, R14 # instrumentation
CMOVNS EBX, dword ptr [RAX]
AND RCX, 0b0111111000000 # instrumentation
ADD RCX, R14 # instrumentation
MOVZX ECX, byte ptr [RCX] # < --- non-speculative load
SUB RBX, 20
{store} ADC EDX, EDX
REX SETNLE AL
{store} REX ADC BL, BL
{disp32} JBE .bb1  # < ---------- branch misprediction
JMP .test_case_main.exit
.bb1:
REX INC AL
AND RCX, 0b0111111000000 # instrumentation
ADD RCX, R14 # instrumentation
MOV EBX, dword ptr [RCX]  # < ---- speculative leak
.test_case_main.exit:
LEA R14, [R14 - 4] # instrumentation
MFENCE # instrumentation
```

C.8 Minimized *ARCH-SEQ* Violation

```
LEA R14, [R14 + 60] # instrumentation
MFENCE # instrumentation
.test_case_main.entry:
JMP .bb0
.bb0:
{load} REX MOV DL, DL
MOVSX RBX, BX
{store} AND BX, BX
ADD EAX, 11839320
AND RDX, 0b0111111000000 # instrumentation
ADD RDX, R14 # instrumentation
SETS byte ptr [RDX]
TEST AX, 6450
AND RDX, 0b0111111000000 # instrumentation
ADD RDX, R14 # instrumentation
SUB word ptr [RDX], CX
{disp32} JB .bb1   # < ---------- branch misprediction
JMP .test_case_main.exit
.bb1:
SUB AX, -29883
AND RAX, 0b0111111000000 # instrumentation
ADD RAX, R14 # instrumentation
ADD EAX, dword ptr [RAX] # < ------- speculative load
AND RAX, 0b0111111000000 # instrumentation
ADD RAX, R14 # instrumentation
CMOVLE RBX, qword ptr [RAX] # < ---- speculative leak
.test_case_main.exit:
LEA R14, [R14 - 60] # instrumentation
MFENCE # instrumentation
```

Bibliography

[1] OpenSSL: Cryptography and SSL/TLS Toolkit. `https://www.openssl.org/`. Accessed: May, 2021.

[2] Apache HTTP server project. `http://httpd.apache.org/`, 2016. Accessed: May, 2021.

[3] BearSSL. `https://bearssl.org/`, 2020. Accessed: May, 2021.

[4] Brotli. `https://brotli.org/`, 2020. Accessed: May, 2021.

[5] JSMN. `https://github.com/zserge/jsmn`, 2020. Accessed: May, 2021.

[6] LibHTP. `https://github.com/OISF/libhtp`, 2020. Accessed: May, 2021.

[7] libyaml. `https://pyyaml.org/wiki/LibYAML`, 2020. Accessed: May, 2021.

[8] Node.js HTTP parser. `https://github.com/nodejs/http-parser`, 2020. Accessed: May, 2021.

[9] IDA Disassembler and Debugger. `https://www.hex-rays.com/products/ida/`, 2021. Accessed: May, 2021.

[10] Metasploit. `https://www.metasploit.com/`, 2021. Accessed: May, 2021.

[11] Tesla Software Updates. `https://www.tesla.com/support/software-updates`, 2021. Accessed: May, 2021.

[12] Zerodium. The world's leading exploit acquisition platform for premium zero-days and advanced cybersecurity capabilities. `https://zerodium.com/`, 2021. Accessed: May, 2021.

[13] Andreas Abel and Jan Reineke. nanoBench: A low-overhead tool for running microbenchmarks on x86 systems. In *ISPASS*, 2020.

[14] Onur Aciiçmez, Shay Gueron, and Jean-Pierre Seifert. New branch prediction vulnerabilities in openssl and necessary software countermeasures. In *IMA International Conference on Cryptography and Coding*, 2007.

[15] Onur Aciiçmez, Çetin Kaya Koç, and Jean Pierre Seifert. On the power of Simple Branch Prediction Analysis. In *AsiaCCS*, 2007.

[16] Onur Aciiçmez and Jean Pierre Seifert. Cheap hardware parallelism implies cheap security. In *FDTC*, 2007.

[17] Onur Aciiçmez, Jean-pierre Seifert, and Çetin Koç. Predicting secret keys via branch prediction. *CT-RSA*, 2007.

[18] Onur Aciiçmez. Yet Another MicroArchitectural Attack: Exploiting I-cache. In *CSAW*, 2007.

[19] Intel Security Advisory. INTEL-SA-00161. Q3 2018 Speculative Execution Side Channel Update. `https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00161.html`, 2018. Accessed: May, 2021.

[20] Intel Security Advisory. INTEL-SA-00233. Q3 2018 Speculative Execution Side Channel Update. `https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00233.html`, 2018. Accessed: May, 2021.

[21] Haitham Akkary, Ravi Rajwar, and Srikanth T. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *MICRO*, 2003.

[22] Fritz Alder, N Asokan, Arseny Kurnikov, Andrew Paverd, and Michael Steiner. S-FaaS: Trustworthy and accountable function-as-a-service using Intel SGX. In *CCSW*, 2019.

[23] Andrew Alexeev. Nginx: The Architecture of Open Source Applications. `www.aosabook.org/en/nginx.html`, 2016. Accessed: May, 2021.

[24] Luca Allodi. Economic factors of vulnerability trade and exploitation. In *CCS*, 2017.

[25] Wil Allsopp. *Unauthorised access: physical penetration testing for IT security teams*. John Wiley & Sons, 2010.

[26] Nadav Amit, Fred Jacobs, and Michael Wei. JumpSwitches: Restoring the Performance of Indirect Branches In the Era of Spectre. In *ATC*, 2019.

[27] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham. On subnormal floating point and abnormal timing. In *S&P*, 2015.

[28] Apache HTTP Server Project. ab - Apache HTTP server benchmarking tool. `https://httpd.apache.org/docs/2.4/programs/ab.html`. Accessed: May, 2021.

[29] ARM. Cache Speculation Side-channels Version 2.5. Technical report, 2020.

[30] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. ISA Semantics for ARMv8-a, RISC-V, and CHERI-MIPS. In *POPL*, 2019.

[31] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Daniel O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure Linux Containers with Intel SGX. In *OSDI*, 2016.

[32] Domagoj Babic, Stefan Bucur, Yaohui Chen, Franjo Ivancic, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. Fudge: Fuzz driver generation at scale. In *ESEC/FSE*, 2019.

[33] Michael Backes, Boris Köpf, and Andrey Rybalchenko. Automatic discovery and quantification of information leaks. In *S&P*, 2009.

[34] Davide Balzarotti, Marco Cova, Vika Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *S&P*, 2008.

[35] Kristin Barber, Li Zhou, Anys Bacha, Yinqian Zhang, and Radu Teodorescu. Isolating Speculative Data to Prevent Transient Execution Attacks. *Computer Architecture Letters*, 2019.

[36] Adam Barth, Collin Jackson, Charles Reis, and Google Chrome Team. The Security Architecture of the Chromium Browser. Technical report, 2008.

[37] BBC News. NHS cyber-attack: GPs and hospitals hit by ransomware. `https://www.bbc.com/news/health-39899646`, 2017. Accessed: May, 2021.

[38] Howard Beales, Richard Craswell, and Steven C Salop. The efficient regulation of consumer information. *The Journal of Law and Economics*, 1981.

[39] Mohammad Behnia, Prateek Sahu, Riccardo Paccagnella, Jiyong Yu, Zirui Zhao, Xiang Zou, Thomas Unterluggauer, Josep Torrellas, Carlos Rozas, Adam Morrison, et al. Speculative interference attacks: Breaking invisible speculation schemes. In *ASPLOS*, 2021.

[40] David Bernick, Bill Bruckert, Paul Del Vigna, David Garcia, Robert Jardine, Jim Klecka, and Jim Smullen. NonStop Advanced Architecture. In *DSN*, 2005.

[41] Daniel Bernstein, Tanja Lange, and Peter Schwabe. The security impact of a new cryptographic library. *LATINCRYPT*, 2012.

[42] Daniel J. Bernstein. Cache-timing attacks on AES. 2005.

[43] Brian N. Bershad, Dennis Lee, Theodore H Romer, and J Bradley Chen. Avoiding conflict misses dynamically in large direct-mapped caches. In *SIGPLAN Notices*, 1994.

[44] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. SMoTherSpectre: Exploiting Speculative Execution through Port Contention. In *CCS*, 2019.

[45] Christian Bienia and Kai Li. PARSEC 2.0: A New Benchmark Suite for Chip-Multiprocessors. In *MoBS*, 2009.

[46] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH computer architecture news*, 2011.

[47] Matt Blaze, Harri Hursti, Margaret MacAlpine, Mary Hanley, Jeff Moss, Rachel Wehr, Kendall Spencer, and Christopher Ferris. DEF CON 27 Voting Machine Hacking Village. *DEFCON*, 2019.

[48] Carl Boettiger. An Introduction to Docker for Reproducible Research. *SIGOPS OS Review*, 2015.

[49] Darrell D. Boggs, Shlomit Weiss, and Alan Kyker. Branch Ordering Buffer, 2004.

[50] Stuart A Boyer. *SCADA: Supervisory Control and Data Acquisition*. 1999.

[51] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. In *WOOT*, 2017.

[52] Thomas Brewster. An NSA Cyber Weapon Might Be Behind A Massive Global Ransomware Outbreak. `https://www.forbes.com/sites/thomasbrewster/2017/05/12/nsa-exploit-used-by-wannacry-ransomware-in-global-explosion`, 2017. Accessed: May, 2021.

[53] Ernie Brickell, Gary Graunke, Michael Neve, and Jean-Pierre Seifert. Software mitigations to hedge AES against cache-based software side channel vulnerabilities. Technical report, 2006.

[54] Samira Briongos, Pedro Malagón, José M. Moya, and Thomas Eisenbarth. RELOAD+REFRESH: Abusing Cache Replacement Policies to Perform Stealthy Cache Attacks. Technical report, 2019.

[55] Robert Brotzman, Shen Liu, Danfeng Zhang, Gang Tan, and Mahmut Kandemir. CaSym: Cache Aware Symbolic Execution for Side Channel Detection and Mitigation. In *S&P*, 2019.

[56] 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 *Usenix Security*, 2018.

[57] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking Data on Meltdown-resistant CPUs. In *CCS*, 2019.

[58] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *Usenix Security*, 2019.

[59] Chandler Carruth. Speculative Load Hardening: A Spectre Variant 1 Mitigation Technique. `https://docs.google.com/document/d/1wwcfv3UV9ZnZVcGiGuoITT_61e_Ko3TmoCS3uXLcJR0/edit#heading=h.phdehs44eom6`, 2018. Accessed: May, 2021.

[60] Sunjay Cauligi, Craig Disselkoen, Klaus v. Gleissenthall, Deian Stefan, Tamara Rezk, Gilles Barthe, Dean Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. Constant-Time Foundations for the New Spectre Era. In *PLDI*, 2020.

[61] Sudipta Chattopadhyay and Abhik Roychoudhury. Symbolic Verification of Cache Side-channel Freedom. *CoRR*, 2018.

[62] Guoxing Chen, Wenhao Wang, Tianyu Chen, Sanchuan Chen, Yinqian Zhang, XiaoFeng Wang, Ten-Hwang Lai, and Dongdai Lin. Racing in Hyperspace: Closing Hyper-Threading Side Channels on SGX with Contrived Data Races. In *S&P*, 2018.

[63] Sanchuan Chen, Fangfei Liu, Zeyu Mi, Yinqian Zhang, Ruby B. Lee, Haibo Chen, and XiaoFeng Wang. Leveraging Hardware Transactional Memory for Cache Side-Channel Defenses. In *AsiaCCS*, 2018.

[64] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. Real time detection of cache-based side-channel attacks using Hardware Performance Counters. Technical report, 2015.

[65] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security*, 2010.

[66] Nick Clifton. SPECTRE Variant 1 scanning tool. `https://access.redhat.com/blogs/766093/posts/3510331`, 2018. Accessed: May, 2021.

[67] David Cock. Practical Probability: Applying pGCL to Lattice Scheduling. In *ITP*, 2013.

[68] Congressional Research Service. Russian Cyber Units. `https://crsreports.congress.gov/product/pdf/IF/IF11718`, 2021. Accessed: May, 2021.

[69] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. 2019.

[70] Intel Corporation. Side Channel Mitigation by Product CPU Model. `https://www.intel.com/content/www/us/en/architecture-and-technology/engineering-new-protections-into-hardware.html`, 2020. Accessed: May, 2021.

[71] Ulan Degenbaev. *Formal Specification of the x86 Instruction Set Architecture*. PhD thesis, Universität des Saarlandes, 2012.

[72] Dorothy E Denning. A lattice model of secure information flow. *Communications of the ACM*, 1976.

[73] Rachna Dhamija, J Doug Tygar, and Marti Hearst. Why phishing works. In *CHI*, 2006.

[74] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX. In *Usenix Security*, 2017.

[75] Georg Disterer. ISO/IEC 27000, 27001 and 27002 for information security management. *Journal of Information Security*, 2013.

[76] Xiaowan Dong, Zhuojia Shen, John Criswell, Alan Cox, and Sandhya Dwarkadas. Spectres, virtual ghosts, and hardware support. In *HASP*, 2018.

[77] Xiaowan Dong, Zhuojia Shen, John Criswell, Alan L Cox, and Sandhya Dwarkadas. Shielding Software From Privileged Side-Channel Attacks. In *Usenix Security*, 2018.

[78] Goran Doychev, Dominik Feld, Boris Köpf, Laurent Mauborgne, and Jan Reineke. CacheAudit: A Tool for the Static Analysis of Cache Side Channels. In *Usenix Security*, 2013.

[79] Goran Doychev and Boris Köpf. Rigorous Analysis of Software Countermeasures against Cache Attacks. In *PLDI*, 2016.

[80] Pavel Emelyanov. Checkpoint/Restore In Userspace. `http://criu.org/`. Accessed: May, 2021.

[81] William Enck, Machigar Ongtang, and Patrick McDaniel. Understanding android security. *S&P Journal*, 2009.

[82] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *MICRO*, 2016.

[83] Dmitry Evtyushkin, Ryan Riley, Nael Abu-Ghazaleh, and Dmitry Ponomarev. BranchScope: A new side-channel attack on directional branch predictor. In *ASPLOS*, 2018.

[84] Mohammad Rahmani Fadiheh, Dominik Stoffel, Clark W. Barrett, Subhasish Mitra, and Wolfgang Kunz. Processor hardware security vulnerabilities and their detection by unique program execution checking. In *DATE*, 2019.

[85] R Fan, L Cheded, and O Toker. Internet-based scada: a new approach using java and xml. *Computing and Control Engineering*, 2005.

[86] Hongyu Fang, Sai Santosh Dayapule, Fan Yao, and Guru Venkataramani. Prefetchguard: Leveraging hardware prefetchers to defend against cache timing channels. In *HOST*, 2018.

[87] Christian Favre. Fly-by-wire for commercial aircraft: the Airbus experience. *International Journal of Control*, 1994.

[88] Joel Fernandes. [PATCH -tip 00/32] Core scheduling (v9). `https://lore.kernel.org/lkml/20201117232003.3580179-1-joel@joelfernandes.org/`, 2020. Accessed: May, 2021.

[89] Stephanie Forrest, Anil Somayaji, and David H Ackley. Building diverse computer systems. In *HotOS*, 1997.

[90] FortiGuard SE Team . Meltdown/Spectre Update. `https://www.fortinet.com/blog/threat-research/the-exponential-growth-of-detected-malware-targeted-at-meltdown-and-spectre`, 2018. Accessed: May, 2021.

[91] Yangchun Fu, Erick Bauman, Raul Quinonez, and Zhiqiang Lin. SGX-LAPD: Thwarting Controlled Side Channel Attacks via Enclave Verifiable Page Faults. In *RAID*, 2017.

[92] Jacob Fustos and Heechul Yun. SpectreRewind: A Framework for Leaking Secrets to Past Instructions. Technical report, 2020.

[93] Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. Time Protection: The Missing OS Abstraction. In *EuroSys*, 2019.

[94] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering*, 2018.

[95] Qian Ge, Yuval Yarom, Frank Li, and Gernot Heiser. Contemporary Processors Are Leaky – and There's Nothing You Can Do About It. *CoRR*, 2016.

[96] Michael Godfrey and Mohammad Zulkernine. A server-side solution to cache-based side-channel attacks in the cloud. In *CLOUD*, 2013.

[97] Shilpi Goel, Warren A. Hunt, and Matt Kaufmann. *Engineering a Formal, Executable x86 ISA Simulator for Software Verification*. 2017.

[98] O. Goldreich. Towards a Theory of Software Protection and Simulation by Oblivious RAMS. *STOC*, 1987.

[99] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *JACM*, 1996.

[100] Google. Honggfuzz. `http://honggfuzz.com/`, 2019. Accessed: May, 2021.

[101] Project Zero Google. Speculative Execution, Variant 4: Speculative Store Bypass. `https://bugs.chromium.org/p/project-zero/issues/detail?id=1528`, 2018. Accessed: May, 2021.

[102] Johannes Gotzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Mller. Cache attacks on intel SGX. In *EuroSec*, 2017.

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

[104] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *Usenix Security*, 2018.

[105] Marc Green, Leandro Rodrigues-Lima, Andreas Zankl, Gorka Irazoqui, Johann Heyszl, Fraunhofer Aisec, and Thomas Eisenbarth. AutoLock: Why Cache Attacks on ARM Are Harder Than You Think. In *Usenix Security*, 2017.

[106] Andy Greenberg. *Sandworm: A New Era of Cyberwar and the Hunt for the Kremlin's Most Dangerous Hackers*. 2019.

[107] Andrew Gross. Java and Speculative Execution Vulnerabilities. `https://mail.openjdk.java.net/pipermail/vuln-announce/2019-July/000002.html`, 2019. Accessed: May, 2021.

[108] Daniel Gruss, Julian Lettner, Felix Schuster, Olga Ohrimenko, Istvan Haller, and Manuel Costa. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In *Usenix Security*, 2017.

[109] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. KASLR is Dead: Long Live KASLR. In *ESSoS*, 2017.

[110] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In *DIMVA*, 2016.

[111] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *Usenix Security*, 2015.

[112] Marco Guarnieri, Boris Köpf, Jose F. Morales, Jan Reineke, and Andres Sanchez. SPECTECTOR: Principled Detection of Speculative Information Flows. *CoRR*, 2018.

[113] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. Hardware-software contracts for secure speculation. In *S&P*, 2021.

[114] Shay Gueron. A Memory Encryption Engine Suitable for General Purpose Processors. In *S&P*, 2016.

[115] Shengjian Guo, Yueqi Chen, Peng Li, Yueqiang Cheng, Huibo Wang, Meng Wu, and Zhiqiang Zuo. SpecuSym: Speculative Symbolic Execution for Cache Timing Leak Detection. *CoRR*, 2019.

[116] Shengjian Guo, Yueqi Chen, Jiyong Yu, Meng Wu, Zhiqiang Zuo, Peng Li, Yueqiang Cheng, and Huibo Wang. Exposing cache timing side-channel leaks through out-of-order symbolic execution. *OOPSLA*, 2020.

[117] Marcus Hähnel, Weidong Cui, and Marcus Peinado. High-Resolution Side Channels for Untrusted Operating Systems. In *ATC*, 2017.

[118] D. Halperin, T. S. Heydt-Benjamin, K. Fu, T. Kohno, and W. H. Maisel. Security and privacy for implantable medical devices. *IEEE Pervasive Computing*, 2008.

[119] Shaobo He, Michael Emmi, and Gabriela Ciocarlie. ct-fuzz: Fuzzing for Timing Leaks. Technical report, 2019.

[120] John L Henning. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 2006.

[121] Jahn Horn. Reading privileged memory with a side-channel. `https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html`, 2018.

[122] Mark Hosenball. FBI paid under $1 million to unlock San Bernardino iPhone | Reuters. `https://www.reuters.com/article/us-apple-encryption-idUSKCN0XQ032`, 2016. Accessed: May, 2021.

[123] Shohreh Hosseinzadeh, Hans Liljestrand, Ville Leppänen, and Andrew Paverd. Mitigating Branch-Shadowing Attacks on Intel SGX using Control Flow Randomization. In *SysTEX*, 2018.

[124] J. D. Hunter. Matplotlib: A 2D Graphics Environment. *Computing in Science Engineering*, 2007.

[125] Immunity Inc. Kernel Memory disclosure & CANVAS Part 1 - Spectre: tips & tricks. `https://www.immunityinc.com/downloads/Kernel-Memory-Disclosure-and-Canvas_Part_1.pdf`, 2018. Accessed: May, 2021.

[126] Open Source Security Inc. Respectre: The State of the Art in Spectre Defenses. `https://www.grsecurity.net/respectre_announce.php`, 2018. Accessed: May, 2021.

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

[128] Intel Corporation. Analysis of Speculative Execution Side Channels. *White Paper*, 2018.

[129] Intel Corporation. Speculative execution side channel mitigations. *White Paper*, 2018.

[130] Intel Corporation. *Intel ® 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation, 2021.

[131] G. Irazoqui, T. Eisenbarth, and B. Sunar. S$A: A Shared Cache Attack That Works across Cores and Defies VM Sandboxing – and Its Application to AES. In *S&P*, 2015.

[132] Gorka Irazoqui, Kai Cong, Xiaofei Guo, Hareesh Khattri, Arun Kanuparthi, Thomas Eisenbarth, and Berk Sunar. Did we learn from LLC Side Channel Attacks? A Cache Leakage Detection Tool for Crypto Libraries. *CoRR*, 2017.

[133] Reinders James. Intel Process Trace. `https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing`, 2013. Accessed: March, 2020.

[134] Jekyll. Directory structure – Jekyll. `https://jekyllrb.com/docs/structure/`. Accessed: May, 2021.

[135] A. L. Johnson. Flamer: Highly Sophisticated and Discreet Threat Targets the Middle East. `https://community.broadcom.com/symantecenterprise/communities/community-home/librarydocuments/viewdocument?DocumentKey=c6d8b8d6-db3e`

-4d63-b947-411739f9e7f6&CommunityKey=1ecf5f55-9545-44d6-b0f4-4e4a7f5f5e68&tab=librarydocuments, 2012. Accessed: May, 2021.

[136] Richard E Kessler and Mark D Hill. Page placement algorithms for large real-indexed caches. *ACM Transactions on Computer Systems (TOCS)*, 1992.

[137] 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 *DAC*, 2019.

[138] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. StealthMem: System-level protection against cache-based side channel attacks in the cloud. *Usenix Security*, 2012.

[139] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. DAWG: A defense against cache timing attacks in speculative execution processors. In *MICRO*, 2018.

[140] Vladimir Kiriansky and Carl Waldspurger. Speculative buffer overflows: Attacks and defenses. *CoRR*, 2018.

[141] Paul Kocher. Spectre Mitigations in Microsoft's C/C++ Compiler. https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html, 2018. Accessed: May, 2021.

[142] 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 *S&P*, 2019.

[143] David Kohlbrenner and Hovav Shacham. On the effectiveness of mitigations against floating-point timing channels. In *Usenix Security*, 2017.

[144] Tadayoshi Kohno, Adam Stubblefield, Aviel D Rubin, and Dan S Wallach. Analysis of an electronic voting system. In *S&P*, 2004.

[145] Boris Köpf and David Basin. An Information-Theoretic Model for Adaptive Side-Channel Attacks. In *CCS*, 2007.

[146] Boris Köpf and Laurent Mauborgne. Automatic Quantification of Cache Side-Channels. In *CAV*, 2012.

[147] Esmaeil Mohammadian Koruyeh, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *WOOT*, 2018.

[148] Christina Kubecka. How To Implement IT Security After A Cyber Meltdown. In *BlackHat*, 2015.

[149] Aleksejs Kuprins. Analysis of Joker—A Spy & Premium Subscription Bot on GooglePlay. `https://medium.com/csis-techblog/analysis-of-joker-a-spy-premium-subscription-bot-on-googleplay-9ad24f044451`, 2012. Accessed: May, 2021.

[150] Kevin Laeufer, Jack Koenig, Donggyu Kim, Jonathan Bachrach, and Koushik Sen. RFUZZ: Coverage-directed fuzz testing of RTL on FPGAs. *ICCAD*, 2018.

[151] Adam Langley. Checking that functions are constant time with Valgrind. `https://www.imperialviolet.org/2010/04/01/ctgrind.html`, 2010.

[152] Ralph Langner. Stuxnet: Dissecting a cyberwarfare weapon. *S&P Journal*, 2011.

[153] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *CGO*, 2004.

[154] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Dawn Song, and Krste Asanović. Keystone: An open framework for architecting trusted execution environments. In *EuroSys*, 2020.

[155] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *Usenix Security*, 2017.

[156] Matt Linton and Pat Parseghian. Retpoline: More details about mitigations for the CPU Speculative Execution issue | Google Online Security Blog. `https://security.googleblog.com/2018/01/more-details-about-mitigations-for-cpu_4.html`, 2018. Accessed: May, 2021.

[157] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache Attacks on Mobile Devices. In *Usenix Security*, 2016.

[158] 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 *Usenix Security*, 2018.

[159] Chang Liu, Austin Harris, Martin Maas, Michael Hicks, Mohit Tiwari, and Elaine Shi. GhostRider: A Hardware-Software System for Memory Trace Oblivious Computation. In *ASPLOS*, 2015.

[160] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In *S&P*, 2015.

[161] Fangfei Liu, Qian Ge, Yuval Yarom, Frank McKeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. CATalyst: Defeating Last-Level Cache Side Channel Attacks in Cloud Computing. In *HPCA*, 2016.

[162] LLVM. LLVM SanitizerCoverage. `https://clang.llvm.org/docs/SanitizerCoverage.html`. Accessed: May, 2021.

[163] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. PIN : building customized program analysis tools with dynamic instrumentation. In *ACM Sigplan Notices*, 2005.

[164] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative Execution Using Return Stack Buffers. In *ACM CCS*, 2018.

[165] Andrea Mambretti, Alexandra Sandulescu, Alessandro Sorniotti, William Robertson, Engin Kirda, and Anil Kurmus. Bypassing memory safety mechanisms through speculative control flow hijacks. Technical report, 2020.

[166] Andrew Marshall, Michael Howard, Grant Bugher, and Brian Harden. Security best practices for developing Windows Azure applications. Microsoft Corp, 2010.

[167] Ross McIlroy, Jaroslav Sevcik, Tobias Tebbi, Ben L. Titzer, and Toon Verwaest. Spectre is here to stay: An analysis of side-channels and speculative execution. *CoRR*, 2019.

[168] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP*, 2013.

[169] Wes McKinney. pandas: a foundational python library for data analysis and statistics. *Python for High Performance and Scientific Computing*, 2011.

[170] MDN. MDN Web Docs: performance.now(). `https://developer.mozilla.org/en-US/docs/Web/API/Performance/now`. Accessed: May, 2021.

[171] Marcela S Melara, Michael J Freedman, and Mic Bowman. EnclaveDom: Privilege separation for large-TCB applications in trusted execution environments. *arXiv preprint arXiv:1907.13245*, 2019.

[172] Dirk Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014.

[173] Microsoft. MSVC compiler reference: /Qspectre. `https://docs.microsoft.com/en-us/cpp/build/reference/qspectre?view=vs-2019`, 2018. Accessed: May, 2021.

[174] Charles Miller. The legitimate vulnerability market: the secretive world of 0-day exploit sales. In *Independent Security Evaluators*, 2007.

[175] Daniel Moghimi, Moritz Lipp, Berk Sunar, and Michael Schwarz. Medusa: Microarchitectural Data Leakage via Automated Attack Synthesis Background Superscalar Memory Architecture. In *Usenix Security*, 2020.

[176] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. SeL4: from general purpose to a proof of information flow enforcement. In *S&P*, 2013.

[177] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *PLDI*, 2009.

[178] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael Abu-Ghazaleh. Rendered insecure: GPU side channel attacks are practical. In *CCS*, 2018.

[179] National Vulnerability Database. CWE Over Time. `https://nvd.nist.gov/general/visualizations/vulnerability-visualizations/cwe-over-time`, 2021. Accessed: May, 2021.

[180] James Nechvatal, Elaine Barker, Lawrence Bassham, William Burr, Morris Dworkin, James Foti, and Edward Roback. Report on the development of the Advanced

Encryption Standard (AES). *Journal of Research of the National Institute of Standards and Technology*, 2001.

[181] Hamed Nemati, Pablo Buiras, Andreas Lindner, Roberto Guanciale, and Swen Jacobs. Validation of Abstract Side-Channel Models for Computer Architectures. In *CAV*, 2020.

[182] Hamed Nemati, Roberto Guanciale, Pablo Buiras, and Andreas Lindner. Speculative Leakage in ARM Cortex-A53. *CoRR*, 2020.

[183] Shirin Nilizadeh, Yannic Noller, and Corina S Pasareanu. DIFFUZZ: Differential Fuzzing for Side-Channel Analysis. Technical report, 2018.

[184] Aleph One. Smashing The Stack For Fun And Profit. `http://phrack.org/issues/49/14.html`, 1996. Accessed: May, 2021.

[185] 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*, 2015.

[186] Meni Orenbach, Andrew Baumann, and Mark Silberstein. Autarky: closing controlled channels with self-paging enclaves. In *EuroSys*, 2020.

[187] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. Eleos: ExitLess OS Services for SGX Enclaves. In *EuroSys*, 2017.

[188] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: The Case of AES. In *CT-RSA*, 2006.

[189] D. Page. Theoretical use of cache memory as a cryptanalytic side-channel. Technical report, 2002.

[190] Dan Page. Partitioned Cache Architecture as a Side-Channel Defence Mechanism. Technical report, 2005.

[191] Colin Percival. Cache Missing For Fun And Profit, 2005.

[192] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM addressing for cross-CPU attacks. In *Usenix Security*, 2016.

[193] Phoronix. Spectre/Meltdown/L1TF/MDS Mitigation Costs On An Intel Dual Core + HT Laptop. `https://www.phoronix.com/scan.php?page=news_item&px=Spec-Melt-L1TF-MDS-Laptop-Run`, 2019. Accessed: May, 2021.

[194] Phoronix. The Brutal Performance Impact From Mitigating The LVI Vulnerability. `https://www.phoronix.com/scan.php?page=article&item=lvi-attack-perf`, 2020. Accessed: May, 2021.

[195] Zhenxiao Qi, Qian Feng, Yueqiang Cheng, Mengjia Yan, Peng Li, Heng Yin, and Tao Wei. Spectaint: Speculative taint analysis for discovering spectre gadgets. In *NDSS*, 2021.

[196] Nguyen Anh Quynh and Dang Hoang Vu. Unicorn: Next generation CPU emulator framework. *BlackHat USA*, 2015.

[197] Arun Raj and Janakiram Dharanipragada. Keep the PokerFace on! Thwarting cache side channel attacks by memory bus monitoring and cache obfuscation. *Journal of Cloud Computing*, 2017.

[198] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing Digital Side-Channels through Obfuscated Execution. In *Usenix Security*, 2015.

[199] Ashay Rane, Calvin Lin, and Mohit Tiwari. Secure, precise, and fast floating-point operations on x86 processors. In *Usenix Security*, 2016.

[200] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. In *HPCA*, 2007.

[201] James A Ratches. Review of current aided/automatic target acquisition technology for military target acquisition tasks. *Optical Engineering*, 2011.

[202] Charles Reis, Alexander Moshchuk, and Nasko Oksov. Site Isolation: Process Separation for Web Sites within the Browser. In *Usenix Security*, 2019.

[203] Röttger, Stephen. Popping Calc with Hardware Vulnerabilities. `https://www.offensivecon.org/speakers/2020/stephen-roettger.html`, 2020. Accessed: May, 2021.

[204] Gururaj Saileshwar and Moinuddin K. Qureshi. CleanupSpec: An Undo Approach to Safe Speculation. In *MICRO*, 2019.

[205] Gururaj Saileshwar and Moinuddin K Qureshi. Lookout for Zombies: Mitigating Flush + Reload Attack on Shared Caches by Monitoring Invalidated Lines. Technical report, 2019.

[206] Christos Sakalis, Stefanos Kaxiras, Alberto Ros, Alexandra Jimborean, and Magnus Själander. Efficient Invisible Speculative Execution through Selective Delay and Value Prediction. In *ISCA*, 2019.

[207] Christos Sakalis, Carl Leonardsson, Stefanos Kaxiras, and Alberto Ros. Splash-3: A properly synchronized benchmark suite for contemporary research. In *ISPASS*, 2016.

[208] Christoph Schmittner, Gerhard Griessnig, and Zhendong Ma. Status of the Development of ISO/SAE 21434. In *EuroSPI*, 2018.

[209] Bruce Schneier. The NSA Is Hoarding Vulnerabilities. `https://www.schneier.com/blog/archives/2016/08/the_nsa_is_hoar.html`, 2016. Accessed: May, 2021.

[210] Bruce Schneier. Who are the shadow brokers? *The Atlantic*, 2017.

[211] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *DIMVA*, 2017.

[212] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *CCS*, 2019.

[213] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript. In *PC*, 2017.

[214] Michael Schwarz, Robert Schilling, Florian Kargl, Moritz Lipp, Claudio Canella, and Daniel Gruss. ConTExT: Leakage-Free Transient Execution. *CoRR*, 2019.

[215] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. NetSpectre: Read Arbitrary Memory over Network. In *ESORICS*, 2019.

[216] SeL4. Timing channels in seL4. `https://ts.data61.csiro.au/projects/TS/timeprotection.html`, 2018. Accessed: May, 2021.

[217] Jaebaek Seo, Byoungyoung Lee, Seong Min Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs. In *NDSS*, 2017.

[218] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *ATC*, 2012.

[219] Kostya Serebryany. OSS-Fuzz - Google's continuous fuzzing service for open source software. In *Usenix Security*, 2017.

[220] Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without function calls (on the x86). In *CCS*, 2007.

[221] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. Occlum: Secure and efficient multitasking inside a single enclave of Intel SGX. In *ASPLOS*, 2020.

[222] Jicheng Shi, Xiang Song, Haibo Chen, and Binyu Zang. Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring. In *DSN-W*, 2011.

[223] MW Shih, S Lee, and T Kim. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *NDSS*, 2017.

[224] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. Preventing Page Faults from Telling Your Secrets. In *Asia CCS*, 2016.

[225] Mark Silberstein, Oleksii Oleksenko, and Christof Fetzer. Speculating about speculation: on the (lack of) security guarantees of Spectre-V1 mitigations. `https://www.sigarch.org/speculating-about-speculation-on-the-lack-of-security-guarantees-of-spectre-v1-mitigations/`, 2018. Accessed: May, 2021.

[226] SUSE Software Solutions. SUSE Security update for kernel-firmware. `https://www.suse.com/de-de/support/update/announcement/2018/suse-su-20180008-1/`, 2018. Accessed: May, 2021.

[227] Claudio Soriente, Ghassan Karame, Wenting Li, and Sergey Fedorov. ReplicaTEE: Enabling seamless replication of SGX enclaves in the cloud. In *S&P*, 2019.

[228] Julian Stecklina and Thomas Prescher. LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels. Technical report, 2018.

[229] Emil Stefanov, Elaine Shi, and Dawn Song. Towards Practical Oblivious RAM. In *NDSS*, 2011.

[230] David Stewart. Improving protections against speculative execution side channel. `https://youtu.be/mSo7J5Ul4E8?t=463`, 2020. Presented at FOSDEM. Accessed: May, 2021.

[231] Raoul Strackx and Frank Piessens. The Heisenberg Defense: Proactively Defending SGX Enclaves against Page-Table-Based Side-Channel Attacks. Technical report, 2017.

[232] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal war in memory. In *S&P*, 2013.

[233] Mohammadkazem Taram and Dean Tullsen. Context-Sensitive Fencing: Securing Speculative Execution via Microcode Customization. In *ASPLOS*, 2019.

[234] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. Packet Chasing: Spying on Network Packets over a Cache Side-Channel. In *ISCA*, 2019.

[235] M Caner Tol, Berk Gulmezoglu, Koray Yurtseven, and Berk Sunar. FastSpec: Scalable generation and detection of Spectre gadgets using neural embeddings. Technical report, 2020.

[236] tom's Hardware. Core i7-980X: Do You Want Six Cores Or 12 Threads? `https://www.tomshardware.com/reviews/hyper-threading-core-i7-980x,2584-5.html`, 2010. Accessed: May, 2021.

[237] Bohdan Trach, Rasha Faqeh, Oleksii Oleksenko, Wojciech Ozga, Pramod Bhatotia, and Christof Fetzer. T-Lease: A trusted lease primitive for distributed systems. In *SoCC*, 2020.

[238] Matthias Traub, Alexander Maier, and Kai L. Barbehon. Future Automotive Architecture and the Impact of IT Trends. *IEEE Software*, 2017.

[239] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. CheckMate: Automated Exploit Program Generation for Hardware Security Verification. In *MICRO*, 2018.

[240] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient Cache Attacks on AES, and Countermeasures. *Journal of Cryptology*, 2010.

[241] Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri, and Hiroshi Miyauchi. Cryptanalysis of DES implemented on computers with cache. In *CHES*, 2003.

[242] Chris Urmson, Joshua Anhalt, Drew Bagnell, Christopher Baker, Robert Bittner, M N Clark, John Dolan, Dave Duggins, Tugrul Galatali, Chris Geyer, Michele Gittleman, Sam Harbaugh, Martial Hebert, Thomas M Howard, Sascha Kolski, Alonzo Kelly, Maxim Likhachev, Matt McNaughton, Nick Miller, Kevin Peterson, Brian Pilnick, Raj Rajkumar, Paul Rybski, Bryan Salesky, Young-Woo Seo, Sanjiv Singh, Jarrod Snider, Anthony Stentz, Ziv Wolkowicki, Jason Ziglar, Hong Bae, Thomas Brown, Daniel Demitrish, Bakhtiar Litkouhi, Jim Nickolaou, Varsha Sadekar, Wende Zhang, Joshua Struble, Michael Taylor, Michael Darms, and Dave Ferguson. Autonomous Driving in Urban Environments: Boss and the Urban Challenge. *Journal of Field Robotics*, 2008.

[243] U.S. Department of Justice. North Korean Regime-Backed Programmer Charged With Conspiracy to Conduct Multiple Cyber Attacks and Intrusions. `https://www.justice.gov/opa/pr/north-korean-regime-backed-programmer-charged-conspiracy-conduct-multiple-cyber-attacks-and`, 2018. Accessed: May, 2021.

[244] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, Frank Piessens, and Ku Leuven. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. Technical report, 2020.

[245] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling Your Secrets Without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *Usenix Security*, 2017.

[246] Jeroen Van Cleemput, Bjorn De Sutter, and Koen De Bosschere. Adaptive Compiler Strategies for Mitigating Timing Side Channel Attacks. *Transactions on Dependable and Secure Computing*, 2017.

[247] 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*, 2019.

[248] Venkatanathan Varadarajan, Thomas Ristenpart, and Michael Swift. Scheduler-based Defenses against Cross-VM Side-channels. In *Usenix Security*, 2014.

[249] Bhanu C Vattikonda, Sambit Das, and Hovav Shacham. Eliminating Fine Grained Timers in Xen. In *CCSW*, 2011.

[250] Luke Wagner. Mozilla Security Blog: Mitigations landing for new class of timing attack. `https://blog.mozilla.org/security/2018/01/03/mitigations-landing-new-class-timing-attack/`, 2018. Accessed: May, 2021.

[251] Guanhua Wang, Sudipta Chattopadhyay, Arnab Kumar Biswas, Tulika Mitra, and Abhik Roychoudhury. KLEESPECTRE: Detecting Information Leakage through Speculative Cache Attacks via Symbolic Execution. *Transactions on Software Engineering and Methodol*, 2020.

[252] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. oo7: Low-overhead Defense against Spectre Attacks. *CoRR*, 2018.

[253] Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. CacheD: Identifying Cache-Based Timing Channels in Production Software. In *Usenix Security*, 2017.

[254] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. In *CCS*, 2017.

[255] Wubing Wang, Yinqian Zhang, and Zhiqiang Lin. Time and Order: Towards Automatically Identifying Side-Channel Vulnerabilities in Enclave Binaries. In *RAID*, 2019.

[256] Yao Wang, Andrew Ferraiuolo, and G. Edward Suh. Timing channel protection for a shared memory controller. In *HPCA*, 2014.

[257] Yao Wang and G. Edward Suh. Efficient timing channel protection for on-chip networks. In *NoCS*, 2012.

[258] Zhenghong Wang and Ruby B. Lee. Covert and Side Channels Due to Processor Architecture. In *ACSAC*, 2006.

[259] Zhenghong Wang and Ruby B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *ISCA*, 2007.

[260] Hassan M. G. Wassel, Ying Gao, Jason K. Oberg, Ted Huffmire, Ryan Kastner, Frederic T. Chong, and Timothy Sherwood. SurfNoC: A low latency and provably non-interfering approach to secure networks-on-chip. *Computer Architecture News*, 2013.

[261] Samuel Weiser, Andreas Zankl, Raphael Spreitzer, Katja Miller, Stefan Mangard, and Georg Sigl. DATA – Differential Address Trace Analysis: Finding Address-based Side-Channels in Binaries. In *Usenix Security*, 2018.

[262] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F Wenisch, and Yuval Yarom. Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution. 2018.

[263] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas F. Wenisch, and Baris Kasikci. NDA: Preventing Speculative Execution Attacks at Their Source. In *MICRO*, 2019.

[264] Jan Wichelmann, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. MicroWalk: A Framework for Finding Side Channels in Binaries. *CoRR*, 2018.

[265] Stephen B. Wicker. The Ethics of Zero-Day Exploits: The NSA Meets the Trolley Car. *Communications ACM*, 2020.

[266] Nils Wistoff, Moritz Schneider, Frank K. Gürkaynak, Luca Benini, and Gernot Heiser. Prevention of Microarchitectural Covert Channels on an Open-Source 64-bit RISC-V Core. *CoRR*, abs/2005.02193, 2020.

[267] Jonathan Woodruff, Alexandre Joannou, Peter Rugg, Hongyan Xia, James Clarke, Hesham Almatary, Prashanth Mundkur, Robert Norton-Wright, Brian Campbell, Simon Moore, and Peter Sewell. TestRIG: Framework for testing RISC-V processors with Random Instruction Generation. `https://github.com/CTSRD-CHERI/TestRIG`, 2018. Accessed: May, 2021.

[268] Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. Eliminating Timing Side-Channel Leaks using Program Repair. Technical report, 2018.

[269] Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the hyper-space: High-bandwidth and reliable covert channel attacks inside the cloud. In *Usenix Security*, 2012.

[270] Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud. In *Transactions on Networking*, 2015.

[271] Yuan Xiao, Yinqian Zhang, and Radu Teodorescu. SpeechMiner: A Framework for Investigating and Measuring Speculative Execution Vulnerabilities. In *NDSS*, 2020.

[272] Wenjie Xiong and Jakub Szefer. Leaking Information Through Cache LRU States. Technical report, 2019.

[273] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *S&P*, 2015.

[274] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher W Fletcher, and Josep Torrellas. InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy. In *MICRO*, 2018.

[275] Mengjia Yan, Bhargava Gopireddy, Thomas Shull, and Josep Torrellas. Secure Hierarchy-Aware Cache Replacement Policy (SHARP): Defending Against Cache-Based Side Channel Attacks. In *ISCA*, 2017.

[276] Yuval Yarom and Katrina Falkner. Flush+Reload: A High Resolution, Low Noise, L3 Cache Side-channel Attack. In *Usenix Security*, 2014.

[277] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W. Fletcher. Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data. In *MICRO*, 2019.

[278] Andreas Zankl, Johann Heyszl, and Georg Sigl. Automated Detection of Instruction Cache Leaks in Modular Exponentiation Software. In *CARDIS*, 2016.

[279] Kim Zetter. *Countdown to Zero Day: Stuxnet and the launch of the world's first digital weapon*. 2014.

[280] Kim Zetter. Inside the Cunning, Unprecedented Hack of Ukraine's Power Grid. `https://www.wired.com/2016/03/inside-cunning-unprecedented-hack-ukraines-power-grid/`, 2016. Accessed: May, 2021.

[281] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. A hardware design language for timing-sensitive information-flow security. In *ASPLOS*, 2015.

[282] Rui Zhang, Calvin Deutschbein, Peng Huang, and Cynthia Sturton. End-to-End Automated Exploit Generation for Validating the Security of Processor Designs. In *MICRO*, 2018.

[283] Rui Zhang, Xiaojun Su, Jianping Wang, Cong Wang, Wenyin Liu, and Rynson W.H. Lau. On Mitigating the Risk of Cross-VM Covert Channels in a Public Cloud. *Transactions on Parallel and Distributed Systems*, 2015.

[284] Tianwei Zhang, Yinqian Zhang, and Ruby B. Lee. CloudRadar: A Real-Time Side-Channel Attack Detection System in Clouds. In *RAID*, 2016.

[285] Xiaokuan Zhang, Yuan Xiao, and Yinqian Zhang. Return-Oriented Flush-Reload Side Channels on ARM and Their Implications for Android Devices. In *CCS*, 2016.

[286] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM Side Channels and Their Use to Extract Private Keys. In *CCS*, 2012.

[287] Yinqian Zhang and Michael K Reiter. Düppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *CCS*, 2013.