SpecDoctor: Differential Fuzz Testing to Find Transient Execution Vulnerabilities

Jaewon Hur
Seoul National University
hurjaewon@snu.ac.kr

Suhwan Song
Seoul National University
sshkeb96@snu.ac.kr

Sunwoo Kim
Samsung Research
sunwoo28.kim@samsung.com

Byoungyoung Lee*
Seoul National University
byoungyoung@snu.ac.kr

ABSTRACT

Transient execution vulnerabilities have critical security impacts to software systems since those break the fundamental security assumptions guaranteed by the CPU. Detecting these critical vulnerabilities in the RTL development stage is particularly important, as it offers a chance to fix the vulnerability early before reaching the chip manufacturing stage.

This paper proposes SpecDoctor, an automated RTL fuzzer to discover transient execution vulnerabilities in the CPU. To be specific, SpecDoctor designs a fuzzing template, allowing it to test all different scenarios of transient execution vulnerabilities (e.g., Meltdown, Spectre, ForeShadow, etc.) with a single template. Then SpecDoctor performs a multi-phased fuzzing, where each phase is dedicated to solve an individual vulnerability constraint in the RTL context, thereby effectively finding the vulnerabilities.

We implemented and evaluated SpecDoctor on two out-of-order RISC-V CPUs, Boom and NutShell-Argo. During the evaluation, SpecDoctor found transient-execution vulnerabilities which share the similar attack vectors as the previous works. Furthermore, SpecDoctor found two interesting variants which abuse unique attack vectors: *Boombard*, and *Birgus*. *Boombard* exploits an unknown implementation bug in RISC-V Boom, exacerbating it into a critical transient execution vulnerability. *Birgus* launches a Spectre-type attack with a port contention side channel in NutShell CPU, which is constructed using a unique combination of instructions. We reported the vulnerabilities, and both are confirmed by the developers, illustrating the strong practical impact of SpecDoctor.

CCS CONCEPTS

• **Security and privacy** → **Side-channel analysis and countermeasures**;

KEYWORDS

transient-execution vulnerability; fuzzing; differential testing

ACM Reference Format:
Jaewon Hur, Suhwan Song, Sunwoo Kim, and Byoungyoung Lee. 2022. SpecDoctor: Differential Fuzz Testing to Find Transient Execution Vulnerabilities. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22), November 7–11, 2022, Los Angeles, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3548606.3560578

1 INTRODUCTION

Transient execution vulnerabilities are critical vulnerabilities in modern CPUs. Since the first disclosure of Spectre [1] and Meltdown [2], many other transient execution vulnerabilities have been discovered—ForeShadow [3], RIDL [4], FPVI [5], Cacheout [6], etc. While such vulnerabilities critically harm the security of software on the affected CPU, vendors were not able to quickly release the patch as these are rooted in the micro-architecture of the CPU.

The root cause of these vulnerabilities is highly related to the speculative execution, which originally intended to maximize the CPU performance. Specifically, CPU attempts to predict the result of the earlier instruction, then execute the later instruction speculatively under the assumption that the prediction was correct [7]. However, if the prediction was wrong, CPU rollbacks the execution of the later instruction to preserve the execution correctness. Here, the rollbacked instructions are called transient instructions. As this rollback is not visible from the architectural point of view, transient execution does not harm the execution correctness of CPU—it only changes the micro-architectural states. The problem is that various attack techniques have been discovered to learn the traces in the micro-architectural states due to the transient instructions. Since these transient instructions are the artifact of the wrong prediction and thus should not be executed, execution results of transient instructions may include security sensitive data, violating the fundamental security isolation guarantees of CPU.

While there have been several previous approaches to automatically detect these vulnerabilities on the off-the-shelf CPUs (i.e., CPUs that are already manufactured) [8, 9], approaches detecting in the RTL development stage have not gained much attention. Detecting in the RTL development stage is particularly important because it offers a chance to fix the vulnerability early, even before the CPU chip is manufactured and released. Once it is released, it becomes extremely difficult to fix as the vulnerabilities are hardwired in the

*Corresponding author

CCS '22, November 7–11, 2022, Los Angeles, CA, USA.

ACM ISBN 978-1-4503-9450-5/22/11…$15.00
https://doi.org/10.1145/3548606.3560578

chip. Taking x86 as an example, many transient execution vulnerabilities are still not fixed (or partially fixed) due to the irrecoverable nature of the hardware [1, 4–6, 10].

Given the importance of early detection, this paper focuses on how to employ popular fuzz testing to realize RTL-based detection of transient execution vulnerabilities. Compared to traditional fuzzing techniques, we find two unique challenges to perform fuzz testing for automated identification of the transient execution vulnerabilities. First, transient execution vulnerabilities can be launched in various threat models and context settings. To be specific, all software entities (i.e., a user program, a kernel, or an enclave) running on the CPU can be the victim or the attacker, as the vulnerabilities exploit the underlying hardware. Moreover, the attacks can be constructed in a different way depending on who performs the transient execution—e.g., the transient execution in Meltdown [2] is performed from the attacker's side, whereas those in Spectre [1] is performed from the victim's side. Furthermore, various CPU settings such as page table settings render the problem even worse (e.g., ForeShadow [3]). These are all unfamiliar topics in traditional fuzz testing since the traditional fuzzers mainly focus on how to extend the input coverage, while the threat model and setting are fairly fixed once the fuzzing target is determined (e.g., AFL [11] and Syzkaller [12] mutate file inputs or syscall inputs, respectively, which do not test different running environments).

The second challenge is that transient execution vulnerabilities are difficult to detect. As already known, in order to trigger the vulnerabilities, two general violations should be raised: 1) transient execution is performed, and 2) micro-architectural side-channels is constructed. However, we do not know how to explicitly express these violations into programmable constraints to be asserted in the RTL context. More importantly, these two violations should be chained to complete the transient execution attacks, making it even more difficult to be found.

In this paper, we propose SpecDoctor, a tool to automatically find transient execution vulnerabilities in the CPU RTL. SpecDoctor is a full-fledged RTL fuzzer, finding unified transient execution attacks while covering all possible context settings. SpecDoctor solves the aforementioned challenges by following two approaches. First, SpecDoctor implements a common template to configure all the threat models and hardware settings. SpecDoctor provides comprehensive configuration options so that all the transient execution vulnerabilities can be covered on the given template. Second, based on the configured template, SpecDoctor designs a multi-phased fuzzing framework to sequentially solve each violation constraint of transient execution vulnerabilities. In order to detect each violation in the RTL layer, SpecDoctor monitors a single RTL structure to detect all the occurrences of transient executions. Then it implements a differential testing framework to identify micro-architectural timing side-channels while selectively monitoring RTL components. Therefore, SpecDoctor outputs a complete transient execution attack PoCs on the target CPU. Using such attack PoCs, we were able to seamlessly reproduce the vulnerability.

We implemented and evaluated SpecDoctor on two RISC-V CPUs, Boom and NutShell-Argo, which are real-world open-source CPUs implementing out-of-order pipelines. As a result, SpecDoctor found transient execution vulnerabilities on these CPUs, which share the similar attack vectors as the discovered attacks [1–3, 13, 14]. Furthermore, SpecDoctor found two interesting attack variants which exploit previously unknown and unique attack vectors: i) *Boombard* on RISC-V Boom, and ii) *Birgus* on NutShell. All these vulnerabilities were confirmed by the corresponding developers, and the *Boombard* has received a CVE number.

Boombard is a Meltdown-variant in RISC-V Boom. The remarkable aspect of *Boombard* is that it exploits an implementation bug in RISC-V Boom—i.e., the implementation does not follow the specification [15]. According to the RISC-V Boom specification, Boom CPU should not have a side-channel in the branch predictor. However, SpecDoctor found an implementation bug in the branch predictor logic, which fails to follow the specification, and re-interpreted it into a critical transient execution vulnerability. *Boombard* alarms the hardware engineers that they should not entirely rely on a CPU specification [16, 17], but verify their RTL implementation to build a secure CPU.

Birgus is a Spectre-variant in RISC-V NutShell, which constructs a port contention side channel using a previously unknown gadget. We note that the instruction patterns for the port contention side channel have been widely studied, and those are blacklisted for mitigation [10, 18]. In this respect, *Birgus* showcases a new instruction pattern which was not identified by the previous work, implying that the new CPU would still be vulnerable to such an attack. Hence, *Birgus* signifies the security risks of a pattern based hardware verification (e.g., regression testing) and Spectre mitigation. Although SpecDoctor does not produce a complete set of gadgets to be checked, *Birgus* suggests that we need continuous verification to stop the transient execution vulnerabilities.

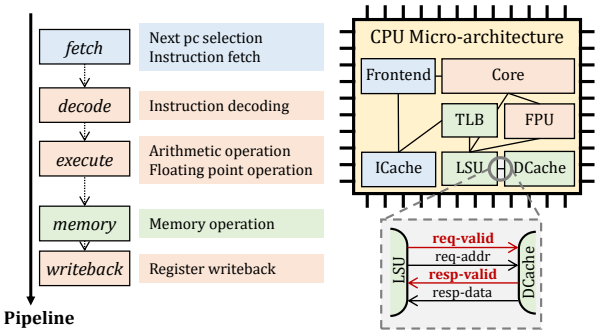
2 BACKGROUND

This section provides background on the transient execution attacks (§2.1) as well as CPU micro-architecture (§2.2), which are necessary to understand SpecDoctor. Then, we briefly describe the basics of fuzzing technique (§2.3).

2.1 Transient Execution Attacks

Transient execution denotes the execution of transient instructions, which change the micro-architectural states but leave no traces in the architectural states. Thus, the transient instructions are not executed from the architectural point of view, it does not harm the execution correctness of the CPU. However, if the attackers can learn the micro-architectural states due to the transient instructions, critical security issues can arise, which is called transient execution attacks. Specifically, transient execution attacks allow the attackers to retrieve the secret data by carefully manipulating the transient execution (e.g., wrong speculation or wrong branch prediction) and then reading the trace of secret left in the micro-architectural states. Almost all of the modern CPUs are known to be vulnerable to the transient execution attacks due to the widely adopted optimization features, such as out-of-order execution and various speculative execution features.

Known Transient Execution Attacks. Various transient execution attacks have been presented from the following aspects: 1) triggering transient execution, 2) accessing secret data, and 3) leaking the data through a micro-architectural side-channel.



(a) Pipeline stages. (b) CPU micro-architecture.

Figure 1: Conceptual pipeline stages and their RTL implementation in the CPU. Modules in CPU continuously exchange signals for executing an instruction.

In order to trigger transient execution, Spectre or Meltdown-type of attacks can be launched. Specifically, Spectre-types manipulate control- and data-flow prediction to trigger the transient execution. For example, Spectre [1] poisons the CPU's branch predictor to trigger transient instructions, and Ret2spec [13] poisons the return address stack. On the other hand, Meltdown-type leverages deferred permission checks such as checking flag bits in a page table entry [2], or a permission to use floating point registers [19].

In order to access secret data, the attackers have to bypass the security boundaries enforced by a software (e.g., bound check) or a hardware (e.g., page permission check). Spectre-type attacks bypass the software enforced security checks by manipulating the control flow through a transient execution. Meanwhile, Meltdown-types bypass the security checks enforced by the hardware primitives. Micro-architectural data sampling (i.e., MDS) attacks can also be classified into Meltdown-type, where the secret data is transiently accessed by the shared buffers in a CPU (e.g., line fill buffer [4]).

Finally, in order to leak the secret data, attackers abuse micro-architectural side-channels. CPUs are known to have various side-channels including caches [20], branch predictors [21], and even an execution port [10]. Based on the vulnerable CPU components, various attack patterns were also proposed to leak the secret in various environments (e.g., flush and reload, prime and probe patterns for D-cache side-channel [22]). Even more side-channels are being continuously disclosed due to the complex and shared nature of the CPU components [18, 23].

2.2 CPU Microarchitecture

CPU micro-architecture is an RTL implementation of the CPU following a predefined instruction set architecture (ISA). Since the CPU serves the entire ISA, ranging from a simple arithmetic operation to the complicated ones such as a virtual memory translation, it is built with a huge number of components and logics.

CPU Pipeline Implementation. CPU serves a given instruction through distinct pipeline stages (e.g., *fetch, decode, execute*, etc.), and the conceptual pipeline stages are materialized into the RTL modules in the CPU. For example, as shown in Figure 1, next pc selection and instruction fetch (i.e., *fetch* in Figure 1-(a)) are handled in the CPU `Frontend` of Figure 1-(b), arithmetic operations (i.e. *execute* in Figure 1-(a)) are performed in the `Core`, and the memory

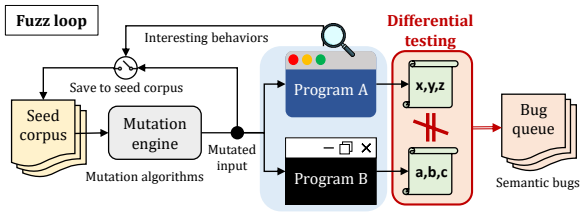


Figure 2: General workflow of differential fuzz testing.

requests (i.e., *memory* in Figure 1-(a)) are served by the load-store unit (i.e., `LSU`) and `DCache`. Thus, an instruction is sequentially processed in each stage while the modules are concurrently working to complete the assigned job (e.g., fetching instruction).

Meanwhile, modules in CPU are interconnected through the wires so that they can exchange the signals for serving the instructions. In particular, modules exchange control signals to indicate the presence of a request or a response. For example in Figure 1-(b), the `LSU` asserts the `req-valid` signal to inform the `DCache` of the memory request, while the `req-addr` carries the requested address. Then, the `DCache` responds to the request by asserting the `resp-valid` signal upon the preparation of the data on the `resp-data` signal. Depending on the location of data (i.e., on L1-, L2-cache, or memory), the assertion of the response would be either delayed or advanced. As such, the communication of the control signal indicates a completion of an assigned pipeline stage.

Out-of-order Execution and Reorder Buffer. Modern CPUs adopt out-of-order execution for high performance. While the instructions should come in effect sequentially from the architectural perspective, the CPU can internally serve the instructions out-of-order. In the out-of-order CPU, instructions which have resolved all the needed operands can be issued immediately so that they can be served earlier. Meanwhile, the CPU maintains the architectural order of the instructions by queuing the fetched instructions in reorder buffer (RoB) sequentially. The RoB guarantees the order by saving the instructions and sequentially retiring them, thus making them visible to the programmer.

However, the RoB cannot retire all the instructions queued into it, since some instructions should not be performed in ISA semantics (i.e., transient instructions). On occurrence of such transient instructions (e.g., branch misprediction), RoB flushes the entries in the queue behind the triggering instruction (e.g., branch instruction) and rolls back the queue tail to the correct point. Thus, RoB works as a single synchronization point for the instructions while the transient instructions are triggered by various parts in the CPU.

2.3 Fuzzing

Fuzzing is arguably the most effective technique for finding bugs in software programs. The key idea behind fuzzing is simply to repeat generating a random input and testing a target program. While iterating the loop, the fuzzer saves the inputs which trigger interesting behaviors (e.g., extend code coverage) and mutates them to further find the corner cases. Starting from the AFL, fuzzers have found huge amounts of bugs from various programs such as a kernel [12], hypervisor [24], or even a CPU [25].

Differential Fuzz Testing. Fuzzer requires a detection mechanism to identify the bug, i.e., abnormal behavior. Though the abnormal behaviors such as memory corruption can easily be detected by memory error detectors (such as ASAN [26]), it is difficult to identify semantic bugs as those do not exhibit apparent abnormal behaviors. Thus, previous works on identifying semantic bugs have leveraged the differential testing, which compares the results of multiple programs with the same purpose (e.g., file system [27], JVM [28]). Since these programs are supposed to return the same results for the same input, we can confirm that it is a bug if they show different input-output relations. Incorporating the differential testing into the fuzzing (i.e., Figure 2), the differential fuzz testing has been widely used to find bugs in programs such as file systems.

3 CHALLENGES IN SPECDOCTOR

Finding transient execution vulnerabilities impose two challenges inherent to the unique characteristics of the vulnerabilities. The first challenge is that the vulnerabilities have various threat models and settings (**C1**). The second is that those have no vulnerability constraints defined in the RTL context (**C2**). In the following, we elaborate these two challenges in turn.

C1. Various Threat Models and Settings. Ever since Spectre and Meltdown have been discovered, many more transient execution vulnerabilities have been discovered under various threat models (i.e., privileges of victim and attacker) and various context settings (i.e., a memory setup, or disabling/enabling SMT) [1, 2, 5, 6]. This suggests that in order to completely detect transient execution vulnerabilities, SpecDoctor should generalize all possible configurations, including threat models, memory settings, and the context of the transient execution.

For instance, let us take the examples of Spectre-like and Meltdown-like vulnerabilities to highlight this challenge. In the case of Spectre-like vulnerabilities, SpecDoctor should be configured as follows: (i) a threat model: the kernel is a victim and the user is an attacker; (ii) a memory setup: a page table should be configured to prohibit the user from accessing kernel's memory; and (iii) a context: the transient execution should be launched from the victim's context. In the case of Meltdown-like vulnerabilities, on the contrary, SpecDoctor should be configured as follows: (i) a threat model: the same as Spectre-like vulnerabilities; (ii) a memory setup: kernel's memory should be protected, but it should be mapped in the same address space of the user; (iii) a context: the transient execution should be launched from the attacker's context.

Furthermore, this challenge becomes even more crucial, as the underlying hardware has a huge number of available configurations (e.g., enclave, SMT, or page table flag settings). For example, Foreshadow [3] found the attack can be launched on the enclaves through the similar mechanism as Meltdown. Medusa [8] introduces a variant of ZombieLoad [29] which exploits a different CPU buffer.

C2. Detecting Vulnerability Constraints in RTL. Memory corruption bugs have a fairly simple vulnerability constraint (i.e., a memory instruction should access a valid memory region), so the conventional fuzzing techniques leverage this simple constraint to detect the bugs [11, 30].

However, transient execution vulnerabilities are semantic vulnerabilities, which do not have explicit constraints in the RTL layer to detect the violations. Worse yet, such vulnerabilities can be launched through diverse instructions with different root causes, which makes it difficult to formulate them into general easy-to-detect constraints. To be specific, previous works [9, 31] roughly sketched the constraints of the transient execution vulnerabilities as follows: (i) attacking instructions are transiently executed; (ii) attacking instructions construct a micro-architectural timing side-channel, thus leaking the secret data. These constraints state the high-level characteristics of transient execution vulnerabilities, but it is unclear how to interpret these constraints from the RTL context.

Taking the example of the constraint (i), the transient execution can be raised by various RTL components—e.g., the branch prediction, load-store page fault, misaligned access instruction, etc. Since each of these causes is implemented in different RTL components, it is unclear how to comprehensively detect the constraint (i). Moreover, the constraint (ii) can be satisfied by various RTL components because there can be various timing side-channels in the CPU—e.g., data and instruction cache, TLB, branch predictor, etc. As mentioned before, it would be challenging for SpecDoctor to detect the violation of constraint (ii) from all different RTL components.

Approaches of SpecDoctor. In order to address aforementioned challenges, we design SpecDoctor based on the following two approaches. In order to address the first challenge (i.e., **C1**), SpecDoctor provides a common template for configuring all the hardware settings (e.g., virtual memory), and it configures the attack by selecting a set of configurations (i.e., threat model, and context of the transient execution). SpecDoctor comprehensively defines configuration options so that entire transient execution vulnerabilities can be covered (more details in §4.1).

To handle the second (i.e., **C2**), SpecDoctor first leverages the key RTL component to detect transient execution, i.e., the constraint (i). This allows SpecDoctor to monitor a single RTL component to capture all transiently executed instructions. SpecDoctor also takes a differential testing approach while selectively monitoring RTL components that can construct a timing side-channel, so that it can detect the constraint (ii). More importantly, SpecDoctor performs a multi-phased fuzzing (more details in §4.2, §4.3, and §4.4), where each phase detects the constraint violation and all phases are sequentially chained together, allowing SpecDoctor to efficiently find transient execution vulnerabilities.

4 DESIGN OF SPECDOCTOR

SpecDoctor is a fully automated CPU RTL fuzzer to find transient execution vulnerabilities. Given the CPU RTL, SpecDoctor automatically tests all the possible configuration options and finds concrete PoCs of transient execution attacks, which can faithfully confirm the vulnerabilities.

Overview. In order to systematically find transient execution vulnerabilities, we design SpecDoctor with four phases as shown in Figure 3. In phase 1, SpecDoctor defines an attack by selecting a set of configuration options such as the threat model and the context of transient execution. Then the base template is configured to

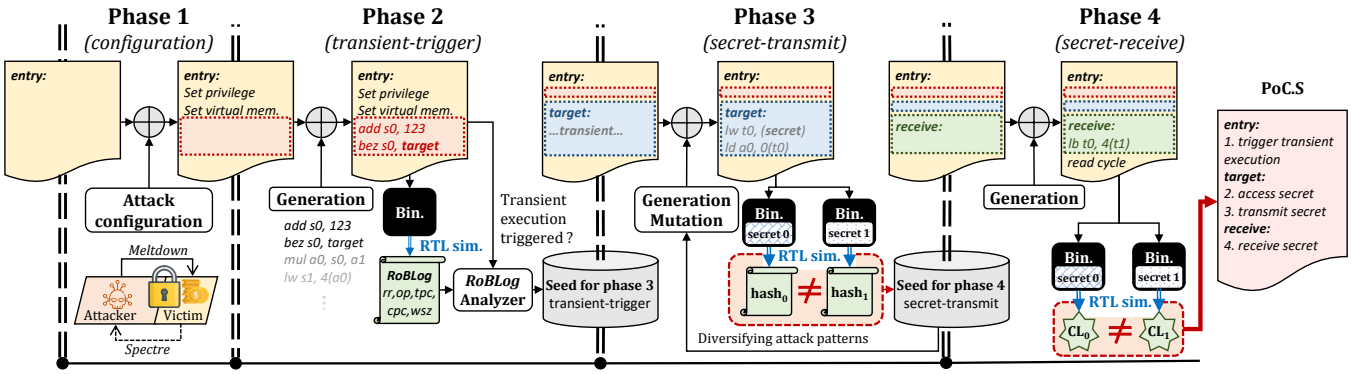
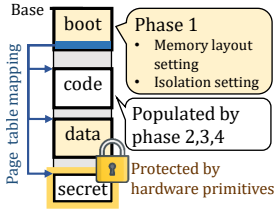


Figure 3: SpecDoctor fuzzer framework for finding transient execution vulnerabilities.

Variables		Options	
Context of transient execution		Attacker, Victim	
Privilege	Victim	M*	S†
	Attacker	S†, U‡	U‡

M*: Machine, S†: Supervisor, U‡: User

(a) Example attack scenarios on RISC-V ISA.



(b) Configured memory layout and hardware protection.

Figure 4: Configurations in SpecDoctor phase 1.

appropriately set the virtual memory and hardware primitives for protecting the secret.

Given the template, SpecDoctor sequentially solves the key vulnerability constraints so that it finally outputs concrete transient execution attack PoCs. In phase 2, SpecDoctor finds instruction sequences which trigger a transient execution. Then phase 3 randomly generates transient instructions based on the testcases from phase 2, so as to find the instructions which transiently access and transmit a secret. Finally in phase 4, SpecDoctor completes the transient execution attacks by finding instructions which can receive the secret transmitted from phase 3.

4.1 Phase 1: Attack Configuration

Since the transient execution attacks can be performed in various environments, SpecDoctor tests all possible attack scenarios by selecting all different attacking context and privileges for the attacker and the victim. The possible options are summarized in Figure 4-(a). In this phase, the key design consideration is to enable SpecDoctor to test all possible attack scenarios while avoiding unnecessarily complex options.

Context of Transient Execution. Transient execution attacks can be performed from two different contexts—either the attacker's context or the victim's context. Since these contexts impact how the attack would be launched (i.e., SpecDoctor has to set the right privilege), SpecDoctor offers the configuration options to specify the context of transient execution. Taking the example of the attacks originating from an attacker's context, Meltdown or MDS are triggered when executing the attacker's code. Taking the example of the victim's context, transient execution is triggered when executing the victim's code.

Privileges of Attacker and Victim. SpecDoctor offers configuration options which specify privileges of the attacker and the victim. In RISC-V, three different privileges are available: machine, supervisor, and user, where each corresponds to the enclave, kernel and user processes. Here, we assume the victim always has the higher privilege than the attacker. This is because the attacker's goal is to retrieve secret data in a higher privilege domain, which is not allowed to be accessed by the privilege design. Different isolation primitives for protecting secret are used depending on the victim's privilege.

Initial Memory Sections. Then, SpecDoctor initializes the memory sections, as it basically runs a program on a bare-metal CPU. SpecDoctor physically allocates four memory sections as in Figure 4-(b): `boot`, `code`, `data`, and `secret` section. In the `boot` section, SpecDoctor places the initial configuration code, such as enabling and initializing the virtual memory layouts as well as setting up the trap vectors. When initializing the virtual memory, SpecDoctor configures the page tables to enable supervisor and user mode programs to run in its own address space. SpecDoctor randomly assigns the flags (i.e., read, write, execute, and user) to each page in memory section so that it can test all possible page table settings. The `code` section is left empty from this phase, as it will be populated later with the fuzzed instructions (§4.2). The `data` section contains random normal data which is randomly loaded or stored in the following phases, and the `secret` section is later populated with secret data for differential testing (§4.3).

Isolation Enforcement on Secret. After setting the memory layout, SpecDoctor protects the secret pages using the hardware primitives. We use different hardware primitives depending on the privilege of the victim.

Following the RISC-V ISA [32], for the machine mode victim, SpecDoctor protects the secret pages using PMP. For the supervisor mode victim, SpecDoctor protects the secret by only removing the user flag in the page table entries of the `secret` section. The page table is also protected from the user mode attackers.

4.2 Phase 2: Triggering Transient Executions

With the initial configuration from phase 1, phase 2 finds instructions which trigger a transient execution through fuzzing—We call such instructions transient-trigger instructions. To this end, SpecDoctor populates the `code` section with random instructions.

Moreover, SpecDoctor designs a monitor to detect the transient execution while running the fuzzed instructions.

Populating Random Instructions. SpecDoctor populates random instructions in following two steps: 1) random control flow graph (CFG) generation, and 2) random opcodes encoding and operands embedding. For the CFG, SpecDoctor randomly constructs a directed graph of basic blocks, then it randomly assigns all sorts of control flow instructions (i.e., branch, direct and indirect jump, call, and return) to connect the basic blocks. Then, each basic block is filled with random instructions which include random opcodes (e.g., `add`) and operands (e.g., `r0, r1`) from the ISA. The maximum number of basic blocks and instructions in each basic block can be configured before running the fuzzer. Then for each generation, SpecDoctor picks those numbers uniformly at random in the range between one and the maximum.

To randomly pick the opcodes and operands, SpecDoctor accepts a dictionary, which lists all the legitimate mappings of the opcodes and operand formats (i.e., RISC-V ISA [32, 33]). Specifically, SpecDoctor randomly picks an opcode from the table (e.g., add immediate opcode `addi`). Then, it randomly embeds valid operand values using the dictionary (e.g., destination register `rd`, source register `rs`, and immediate value `imm`). In most cases, SpecDoctor generates correctly formatted instructions, but it also randomly generates wrong-formatted ones so as to test the case of decoding errors. In particular, the memory and control flow opcodes need a valid target address, but SpecDoctor does not constrain the address so that it can test various architectural exceptions (e.g., load access fault, or misaligned jump). Meanwhile, all the exceptions are caught by the trap vectors registered in phase 1. SpecDoctor also reuses the operand values across the instructions so that the generated instructions micro-architecturally affect each other (e.g., branch instruction resolved long after the fetch due to the preceding instructions). In phase 2, SpecDoctor generates random instructions without coverage feedback.

Detecting Transient Execution. In order to detect transient execution, SpecDoctor monitors internal micro-architectural CPU states. This is because transient execution is not observable outside the CPU (i.e., §2.1), as it is the nature of transient execution. The challenge here is that transient execution can be triggered by various CPU components (e.g., branch predictor, LSU, MMU, etc.), so SpecDoctor may need to monitor all of such micro-architectural behaviors at runtime.

Instead, we leverage the operational characteristic of RoB [7] to minimize the monitoring costs. To be specific, RoB synchronizes all the transient instructions executed in the CPU. Upon the occurrence of transient execution, RoB is always involved to rollback its queue to maintain a correct order of instructions, regardless of the CPU component which triggers the transient execution. In this respect, we manually embed logic into the RoB which monitors and reports the occurrence of a rollback event, thus notifying an occurrence of a transient execution. We note that the RoB is a part of general implementation of an out-of-order CPU, which can be found in major CPUs such as Intel [7].

RoB-Feedback Analysis. The embedded logic continuously monitors RoB rollback events while SpecDoctor runs the random testcases on the RTL simulated CPU. For each rollback event, the logic reports `RoBLog` which contains following information: 1) `rr`, a rollback reason (e.g., branch misprediction), 2) `op`, opcode of the triggering instruction (e.g., `bez`), 3) `tpc`, first pc of the transient instructions (e.g., mispredicted target), 4) `cpc`, correct pc after flushing transient instructions (e.g., correct branch target), and 5) `wsz`, elapsed cycles from the start of transient instructions until the rollback.

Then, SpecDoctor analyzes the `RoBLog` and saves the corresponding testcase based on two criteria. First, the testcase may suggest a new attack vector (i.e., a different combination of `rr` and `op`). SpecDoctor saves the testcase since itself can be a variant of a transient execution vulnerability. Second, if the testcase has an already found attack vector, SpecDoctor prioritizes a testcase with larger transient window size (i.e., `wsz`), because it improves the chance of success for the attack [9].

4.3 Phase 3: Accessing and Transmitting Secret

Given the transient-trigger instructions found by phase 2, phase 3 finds the instructions which are transiently executed, but access and transmit the secret through a micro-architectural timing side-channel.

The basic idea is to monitor CPU's micro-architectural states and check if the states are changed depending on the secret. The challenge executing this idea is that it is costly to monitor entire micro-architectural states. Furthermore, monitoring the entire micro-architectural states would incur large false-positives, as a large portion of CPU components are only used for conveying the data, not affecting the execution time of instructions. Thus, SpecDoctor designs two sub-steps for this phase 3, combining static and dynamic analyses. First, SpecDoctor performs the static analysis to find candidate RTL components (which we call timing-change components) that may have potentials to trigger timing side-channels. Next, SpecDoctor performs the dynamic analysis, particularly a different testing. Monitoring only the timing-change components at runtime, this testing finds concrete transient instructions which transmit a secret through the timing-change components (i.e., secret-transmit instructions).

Statically Finding Timing-change Components. Given the CPU RTL source code, SpecDoctor compiler finds all the timing-change components which can potentially be used as a timing side-channel. In order to statically identify such components, SpecDoctor relies on the insight that the control signals (i.e., `valid` signals in §2.2) can change the instruction execution time. To be specific, since the `valid` signals indicate the presence of a request or a response, the assertion timings of such signals affect the module's following behaviors and affect the instruction execution time. Thus, we assume that the stateful components (i.e., registers and memories) wired into the high-level module's `valid` signals can be used as a timing side-channel, as their states can induce timing difference.

For example, Figure 5 illustrates a part of simplified logic inside a `DCache` of Boom CPU [34]. The `DCache` has two memory components, `tag array` and `data array`. The `tag array` stores high bits of a memory address, while the `data array` stores the memory data. When a memory read/write request comes from the `LSU` through the `req_addr` (i.e., a requested address, ①), `DCache` first compares `req_addr` with the bits stored in the `tag array` (②). If the address matches, `DCache` responds to the `LSU` by 1) asserting the

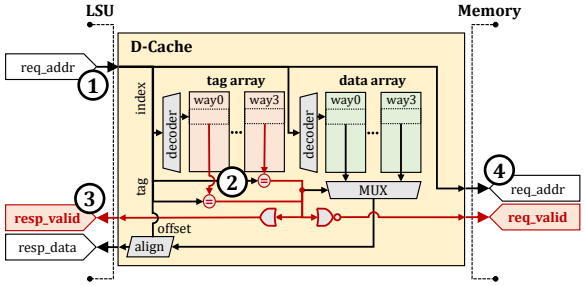
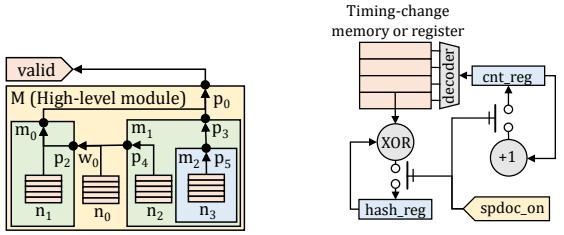


Figure 5: Simplified logic of `DCache` in RISC-V Boom CPU. `tag array` is indirectly wired into the `valid` signals.



(a) High-level module containing four timing-change components.

(b) Hash computation for summarizing the timing-change component's state.

Figure 6: Data-flow analysis and instrumentation by SpecDoctor compiler.

`resp_valid` signal, 2) locating the corresponding data value stored in `data array`, and 3) sending the located data through `resp_data` (③). On the other hand, if the address does not match, `DCache` sends a request to the `Memory` first (④), and later responds to the `LSU` after fetching the requested data. Thus, depending on whether the `tag array` contains a matching address or not, it would lead to the `DCache` timing side-channel by affecting the `resp_valid` signal, while the `data array` does not.

Based on this insight, SpecDoctor designs a backward data-flow analysis to identify all the timing-change components which are wired into the high-level module's `valid` signals. Given a high-level module and its `valid` ports, the analysis returns all the timing-change components using the graphs parsed from the RTL source code. More specifically, the analysis visits entire modules under the high-level module, where SpecDoctor performs the backward slicing for each module. Taking Figure 6-(a) as an example, given the high-level module (i.e., M) which has child modules (i.e., m_0, m_1, and m_2), the analysis returns all the stateful components (i.e., n_0, n_1, n_2, and n_3) wired into the `valid` port (i.e., p_0).

Dynamically Finding Secret-transmit Instructions. After statically identifying the timing-change components, SpecDoctor dynamically finds secret-transmit instructions which transmit a secret through the timing-change components. To be specific, SpecDoctor randomly generates transient instructions on top of the testcases from phase 2, using the algorithm explained in §4.2. Then it runs RTL simulations to detect whether the instructions change the micro-architectural states of the timing-change components depending on a secret value. Since the states of the timing-change components can affect the execution time of instructions, their states could possibly be observed later to infer the transmitted secret value.

To this end, SpecDoctor designs a differential testing framework, which compares the status of the timing-change components after running two RTL simulations with the same transient instructions but with different secret values (i.e., data of `secret` section). Note that only the `secret` section is different while both RTL simulations and all the memory sections (i.e., `boot`, `code`, `data`) are initialized to be the same. Thus, SpecDoctor can confirm that the different states originate from the different secret data. In other words, SpecDoctor can determine if a secret is transmitted by checking if a given testcase results in different micro-architectural states of the timing-change components.

In order to observe the micro-architectural states of the timing-change components, SpecDoctor automatically instruments each component and provides an interface (e.g., memory-mapped `spdoc_addr`) to observe their states. To be specific, SpecDoctor instruments a logic as shown in Figure 6-(b), which prints out the summary of micro-architectural states (i.e., `hash_reg`) when a dedicated signal (i.e., `spdoc_on`) is asserted. Upon the assertion of the `spdoc_on` signal, all the memory lines (or registers) of the timing-change component are iteratively hashed into `hash_reg`, and the final `hash_reg` value is printed out, which summarizes the entire state of the component. Then, we modified the RTL simulation testbench to assert the `spdoc_on` signal when a testcase indicates it through the given interface (e.g., store to `spdoc_addr`).

In this way, SpecDoctor observes the micro-architectural states of the components after running randomly generated transient instructions. For example, given the testcase and corresponding `RoBLog` as shown in the left part of Figure 7, SpecDoctor first embeds randomly generated instructions at the start pc of the transient instructions (i.e., `tpc` in `RoBLog`) so that they are transiently executed (①). Then, SpecDoctor embeds the instructions for observing the micro-architectural states (i.e., hash-measure instructions which store to `spdoc_addr`) at the correct pc after flushing the transient instructions (i.e., `cpc`, ②). Since the instructions at the correct pc are executed right after flushing the transient instructions, SpecDoctor can observe the micro-architectural states right after the transient execution. Note that embedding the instructions between `tpc` and `cpc` does not affect triggering the transient execution, because the transient execution is only triggered by the preceding instructions.

Mutating Random Instructions. Unlike the phase 2, SpecDoctor employs a unique fuzzing feedback in this phase to find various secret-transmit instructions. To be specific, SpecDoctor maintains a corpus which saves the testcases triggering the state differences, and randomly mutates one of them to generate a new testcase. For mutating the testcase, SpecDoctor applies per-instruction mutation algorithm, which randomly replaces, removes, or appends an instruction in the given testcase. The random instructions are constructed as explained in §4.2 (i.e., opcode encoding and operand embedding). Meanwhile, SpecDoctor only mutates the instructions in the transiently executed part (i.e., `tpc` in `RoBLog`) so that the mutation does not affect the earlier phases (e.g., triggering the transient execution).

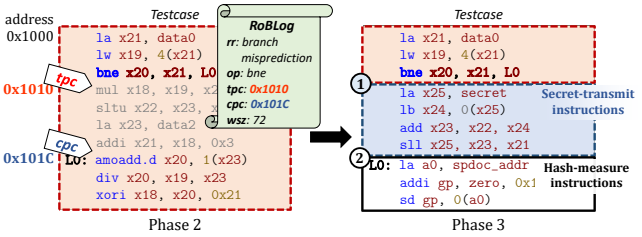


Figure 7: Testcase with RoBLog from SpecDoctor phase 2 (left), and secret-transmit, hash-measure instructions generated from phase 3 (right). rr: RoB rollback reason, op: opcode of triggering instruction, tpc: start pc of transient instructions, cpc: correct pc after flushing transient instructions, and wsz: transient window size.

4.4 Phase 4: Receiving Secret

Given the secret-transmit instructions, phase 4 finds secret-receive instructions, which receive the secret through timing side-channels. To be specific, we design yet another differential testing, which executes a testcase twice with different secret and compares CPU cycles taken. When running these two testcases, all inputs are initialized to be the same (i.e., instructions and memory sections) except the `secret` section. In this way, SpecDoctor ensures that the cycle differences of the executed instructions are only depending on the secret data.

Finding Secret-receive Instructions. In order to find secret-receive instructions, SpecDoctor embeds randomly generated instructions in the `receive` part of the code section, and measures the cycle count before and after executing the instructions. The testcase template has the `receive` part to be executed after all the instructions from phase 1 to 3, so that the secret-receive instructions are executed after the states of the timing-change components are changed by the secret-transmit instructions. Also, the instructions are executed only with the attacker's privilege, as the secret reception is only performed on the attacker's side. Then, SpecDoctor compares the difference of the measured cycle counts (i.e., instruction execution time) between the RTL simulations and determines the secret is received if those are not the same.

Eventually, the entire assembly file which 1) triggers a transient execution, 2) transiently accesses and transmits a secret value, and 3) receives it is saved as a PoC of a transient execution attack.

5 IMPLEMENTATION

SpecDoctor implementation consists of 1) an RTL logic for monitoring RoB rollback events, 2) an RTL compiler for detecting and instrumenting timing-change components, and 3) a fuzzer framework to find transient execution vulnerabilities. The current implementation of SpecDoctor is applied to two CPUs, RISC-V Boom [34] and NutShell [35] CPUs, but the design of SpecDoctor can be applied to other out-of-order CPUs (see more in [36, 37]).

Rollback Monitoring Logic. We manually embedded a rollback monitoring logic in RISC-V Boom and NutShell with 160 and 79 LoC of `Chisel` [38], respectively. To be specific, the implementation contains a logic for detecting the rollback, reading the rollback reason (`rr`), calculating the transient window size (`wsz`), and remembering the `RoBLog` related information (i.e., `op`, `tpc`, and `cpc`). It is worth noting that the monitoring logic is transparent, and thus it does not change the original behavior of the target CPUs.

RTL Compiler for Timing-change Components. In order to detect and instrument timing-change components, we implemented a pass in `FIRRTL` [39] compiler which is used for compiling `Chisel` source code into `Verilog` files. The implementation contains i) 1,000 LoC in `Scala`, performing the static analysis to find the timing-change components and ii) 500 LoC in `Scala`, instrumenting the components.

Dynamic Fuzzer Framework. We implemented all the fuzzing phases with 2,500 LoC in `Python`, which include generating random testcases, running RTL simulations, and analyzing the simulation results. Then we modified the RTL simulation testbench (i.e., `Chipyard` [40] for RISC-V Boom, and `NutShell-SoC` [35] for RISC-V NutShell) for asserting the `spdoc_on` signal.

6 EVALUATION

In this section, we evaluate SpecDoctor from various aspects of the design. To this end, we answer the following research questions:

(1) Can SpecDoctor find various instruction sequences triggering a transient execution? (§6.2)
(2) Can the SpecDoctor compiler find various timing-change components with a reasonable overhead? (§6.3)
(3) Does the multi-phased design of SpecDoctor help quickly finding entire transient execution attacks? (§6.4)
(4) Can SpecDoctor find real-world transient execution vulnerabilities? (§6.5)

6.1 Evaluation Setup

We evaluated SpecDoctor on two open-source out-of-order RISC-V CPUs: Boom [34] and NutShell [35].

RISC-V Boom. Boom implements out-of-order pipelines with 20k lines of `Chisel`, and it has comparable instructions per cycle (IPC) performance to the ARM cortex-A72 core [41]. Boom supports RV64G ISA [33] which includes all privilege levels (i.e., machine, supervisor, and user) and the CPU contains various RTL modules (e.g., issue queue, RoB, and branch predictors).

RISC-V NutShell. NutShell is a configurable RISC-V CPU which implements out-of-order pipelines with 13k lines of `Chisel`, and it also supports RV64IMAC ISA [33] with all privilege levels. Since NutShell does not implement PMP [32] used for protecting an enclave, we did not test the attack scenario with the enclave level victim on this CPU.

Fuzz Testing Environment. We evaluated SpecDoctor using the software RTL simulator, `Verilator`. In order to run the bare-metal binary on the CPU, we slightly modified the corresponding simulation frameworks (i.e., `Chipyard` [40] for Boom and `NutShell-SoC` [35] for NutShell). For the instruction generation, we set the maximum number of basic blocks as 7, and the maximum number of instructions per each basic block as 10. All the fuzzing experiments were carried out on a machine of Intel Xeon Gold 6209 with 40 CPU cores and 512GB RAM, which runs Ubuntu 18.04 LTS. We ran SpecDoctor for more than one week for the evaluation.

6.2 Triggering Transient Executions

In phase 2, SpecDoctor found various transient-trigger instructions, and the results are summarized in Figure 1.

Table 1: Transient executions in RISC-V BOOM and NutShell discovered by SpecDoctor phase 2. rr: RoB rollback reason, op: opcode of triggering instruction, cpu: found cpu which contains the corresponding transient-trigger instructions (i.e., boom, nutshell, and both).

Type	rr		op	cpu
Arch.	PMP violation	Load/Store	ld, st	boom
	VM violation	Load/Store	ld, st	both
	Misaligned access	Load/Store	ld, st	both
Micro.	Control flow misprediction		j, br	both
	Load-store bypass violation		ld	boom

Table 2: Timing-change components found by SpecDoctor in both RISC-V Boom and NutShell. The table partially shows the entire results due to a space limit.

CPU pipeline	Operation	Timing-change component
Frontend	Branch prediction	bim, btb
	Return address prediction	ras
	Virtual memory translation	tlb-tag
	Instruction fetch	icache-tag
Backend	Architectural registers	registers, checkpoint
	Register renaming	rename-map
	Configuration registers	csr
	Instruction issuing	issue-queue
	Virtual memory translation	tlb-tag
	Data fetch	dcache-tag

Types of Transient Execution. We classify the found transient-trigger instructions based on two general root causes: 1) an architectural type, which are defined in the ISA, and 2) a micro-architectural type, which are not defined in the ISA and caused by a CPU's micro-architectural implementation.

In the architectural type, SpecDoctor found that all load-store instructions trigger a transient execution if the instruction raises an exception. However, exceptions related to instruction fetch and decode do not trigger a transient execution. This is expected results considering the design of general CPU pipelines—while the exception condition for load-store instructions are noticed late in the pipeline, that of fetch and decodes instructions are noticed early.

In the micro-architectural type, SpecDoctor found two sources of a transient execution in RISC-V Boom: 1) control-flow misprediction, and 2) load-store bypass violation. Interestingly, atomic memory instructions also trigger an RoB rollback, but current RISC-V Boom implementation does not transiently executes the instructions after the atomic memory instruction (i.e., transient window size is 0). In RISC-V NutShell, SpecDoctor only found that a control-flow misprediction triggers a transient execution.

6.3 Finding Potential Side-Channels

SpecDoctor statically finds timing-change components which are connected to the valid signals. As explained in §4.3, timing-change components can affect the instruction execution time, and they can potentially be used as a timing side-channel.

Discovered Timing-change Components. SpecDoctor found 334, and 101 timing-change components in RISC-V Boom and NutShell, where some of commonly found components are shown in Figure 2. We summarize only a portion of the results due to the space limit.

Table 3: Static and dynamic overheads of SpecDoctor. Numbers in parentheses show the overheads against the original.

Project	Timing-change components	Compilation time (s)	Verilog LoC	Simulation speed (Hz)*	Fuzzing speed (/hour)
Boom	334	311.2 (93%)	479 k (5%)	7.25 k (0.01%)	269
NutShell	101	248.2 (370%)	173 k (23%)	52.6 k (0.01%)	9230

* simulation speed is measured as cycles per second.

In the CPU frontend, SpecDoctor identified several components for predicting the control flow (e.g., BIM, RAS) as timing-change components. Those components are used to select next fetched pc, and it affects the instruction execution time. While SpecDoctor found only one-level branch predictor in RISC-V NutShell, it identified multi-level branch predictors in Boom such as FAMicroBTBBranchPredictor, and TageTable, all of which can affect instruction execution time. In addition to the control flow prediction, SpecDoctor identified timing-change components related to the virtual address translation and fetching instructions.

In the CPU backend, SpecDoctor found timing-change components in register files, control status registers, instruction issue queue, arithmetic unit, etc. The register files are detected as they affect the arithmetic computation, branch resolution, or else. As in the CPU frontend, SpecDoctor detected components related to the virtual address translation, and fetching data. Similar to the branch predictor case, RISC-V Boom was found to have additional level of virtual memory translation cache, called PTW, which also affects memory stage completion time.

Note that the found results contain false-positives as not all the components connected to the valid signals construct a timing side-channel. SpecDoctor finds true timing side-channels through the dynamic differential testing in the following phases.

Instrumentation Overhead. We summarize the statistics of the static analysis and instrumentation in Figure 3. SpecDoctor found 334, and 101 timing-change components in RISC-V Boom and NutShell, and the RTL compilation time including SpecDoctor static analysis were about 311.2 s and 248.2 s. The resulting Verilog files after SpecDoctor instrumentation have 479 k and 173 k LoC for each, which contains 15 % instrumentation overhead on overage. However, the instrumented logic has almost no effect on the RTL simulation speed, as the instrumented logic remains inactive for most of the time, and it works only when monitoring the states of the timing-change components. In terms of the fuzzing speed, RISC-V Boom was slower than NutShell due to the larger simulation complexity and testbench initialization time.

6.4 Multi-phased Fuzzing

SpecDoctor designs a multi-phased fuzzer which finds transient-trigger instructions in phase 2, secret-transmit instructions in phase 3, and secret-receive instructions in phase 4. Thus, we evaluate whether the multi-phased design helps quickly finding various transient execution vulnerabilities.

We implemented two vanilla versions of SpecDoctor: 1) vanilla1 totally randomly generates the entire instructions without any phase, and 2) vanilla2 randomly generates secret-transmit and

Table 4: Comparison study of finding transient execution vulnerabilities. All the attacks have accessed secret through a load instruction.

Transient execution attack			Elapsed CPU time (h)		
context*	transient execution	channel†	vanilla1	vanilla2	SpDoc‡
attacker	arch(store-page-fault)	DCache	*Fail*	*Fail*	46.1
	arch(load-page-fault)	DCache	*Fail*	*Fail*	34.7
	micro(branch-misprediction)	DCache	*Fail*	*Fail*	26.9
victim	micro(branch-misprediction)	DCache	*Fail*	151.6	30.6
	micro(branch-misprediction)	ICache	*Fail*	*Fail*	31.2
	micro(branch-misprediction)	NBDTLB	*Fail*	*Fail*	40.6

* Context of transient execution, † Timing side-channel, ‡ SpecDoctor

Table 5: Variants of transient execution vulnerabilities found by SpecDoctor.

Project	Configuration*	Transient execution†	Timing side-channel‡
Boom	K⇒E (attacker)	pmp/vm-fault	bim, faubtb, tlb, ptw, dcache
	K⇒E (victim)	control-flow	faubtb, ras, i/dcache
		mem-bypass	faubtb, ras, i/dcache
	U⇒K (attacker)	vm-fault	faubtb, ptw, dcache
	U⇒K (victim)	control-flow	faubtb, ras, tlb, ptw, i/dcache
		mem-bypass	faubtb, ras, tlb, ptw, i/dcache
	U⇒E (victim)	control-flow	faubtb, ras, i/dcache
		mem-bypass	faubtb, ras, i/dcache
NutShell	U⇒K (victim)	control-flow	pht, ras, tlb, rs, i/dcache

*<**attacker**>⇒<**victim**> (**Context of transient execution**), **E**: enclave, **K**: kernel, **U**: user
†**control-flow**: control-flow misprediction, **mem-bypass**: load-store bypass violation
‡**faubtb**: 1st level btb, **ras**: return address stack, **ptw**: 2nd level tlb in RISC-V Boom
‡**pht**: pattern history table, **rs**: reservation station in RISC-V NutShell

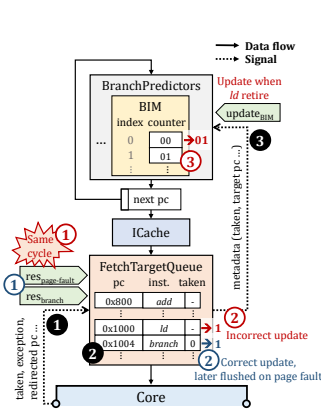
secret-receive instructions on top of the testcases from phase 2, but it does not detect secret transmissions (i.e., phase 3). Thus, all the `vanilla1` and `vanilla2` find transient execution vulnerabilities by comparing CPU cycles as explained in phase 4, while never, or partially solving the vulnerability constraints. Then, we summarized the elapsed CPU time to find the entire vulnerabilities with three different fuzzers (i.e., `vanilla1`, `vanilla2`, and SpecDoctor).

We ran the fuzzers on RISC-V Boom, and summarizes the comparison results as shown in Figure 4. Overall, the results show that the phased fuzzer design clearly helps finding transient execution vulnerabilities. SpecDoctor has found six Meltdown and Spectre-type vulnerabilities within almost 50 CPU hours. On the other hand, `vanilla1` without any phases did not find any vulnerabilities. `vanilla2` found only one attack (i.e., basic Spectre attack) after 151 CPU hours, which was already found by SpecDoctor in 31 hours.

6.5 Found Transient Execution Vulnerabilities

In this section, we evaluate the unified transient execution vulnerabilities found by SpecDoctor. SpecDoctor has found several transient execution vulnerabilities in both RISC-V Boom, and NutShell, including two new vulnerabilities which are explained in §7. In RISC-V Boom, SpecDoctor found the PoCs for Spectre [14], Spectre variant 4 [14], ret2Spec [13], Meltdown [2], and Foreshadow [3]. On the other hand, in RISC-V NutShell, SpecDoctor found Spectre, and ret2Spec. Meltdown-type vulnerabilities were not found in NutShell, as the memory stage of NutShell does not speculatively return data before checking the privilege.

Furthermore, SpecDoctor newly found several attack variants which combine different attack vectors in different attack configurations. While we did not report these vulnerabilities as each attack vector is not new, we want to note that the analysis results would



(a) Mechanism of triggering *Boombard* bug.

```
1  trigger:
2      /* Load from invalid page
3         for triggering page fault */
4      la x18, addr_page_fault
5      lw x19, 0(x18)
6  transient:
7      /* Transiently load secret */
8      la x20, secret
9      ld x21, 0(x20)
10     /* Dummy instructions
11        to simultaneously raise
12        page fault resolution signal and
13        branch resolution signal. */
14     andi x21, x21, 0x1
15     andi x21, x21, 0x1
16     /* Secret dependent branch
17        to transmit secret value */
18     beqz x21, L0
19     nop
20 L0:
21 ...
22 receive:
23     ...
24     /* Receiving secret */
25     bnez x10, L1
26     nop
27 L1:
```

(b) Code snippet of *Boombard*.

Figure 8: *Boombard* transient execution vulnerability on RISC-V Boom.

be helpful for the CPU designers to comprehensively mitigate the vulnerabilities. We categorized the found attack variants as shown in Figure 5. Specifically, RISC-V Boom was vulnerable to more various attacks (e.g., various timing side-channels) as it implements more complex CPU pipeline for optimizing the performance.

7 FINDINGS OF SPECDOCTOR

SpecDoctor found two interesting variants of transient execution vulnerabilities in RISC-V Boom and NutShell, which exploit previously unknown attack vectors. We explain the details of the attacks in the following.

7.1 *Boombard* on RISC-V Boom

Boombard is a variant of Meltdown-type attack found in RISC-V Boom. Different from previously known transient execution vulnerabilities, *Boombard* first showcases that an implementation bug, which disagrees with the specification, can lead to a transient execution vulnerability. To be specific, `BIMBranchPredictor` in Boom should not have the timing side-channel according to its specification—i.e., the specification states that it should not be updated by transient instructions [15] and thus it should not leak the secret through a transient execution. However, SpecDoctor was able to find a transient instruction pattern which unexpectedly updates the `BIMBranchPredictor` (i.e., the *Boombard* bug), and transiently leaks the secret. We reported this vulnerability to the developers and it was assigned with a CVE number, showing the critical impact.

Boombard implicates that in order to detect transient execution vulnerabilities, one should not entirely rely on a CPU specification to model its behavior [16, 17]. Instead, one should always analyze its implementation as well, because the implementation may render its specification in a different way as found by *Boombard*.

Details of *Boombard* Bug. Boom has an internal `BIMBranchPredictor` update logic as shown in Figure 8-(a). According to its specification, `BIMBranchPredictor` only updates its state with the retired branch instruction, and thus transient branch instructions cannot impact `BIMBranchPredictor` [15]. Such an update logic is implemented

with the help of FetchTargetQueue, which maintains the information of the fetched instructions: i) pc and inst, which records the instruction itself; and ii) taken, which records if a branch instruction is resolved to be taken or not. Then, FetchTargetQueue updates the information depending on two signals from the Core: i) a branch resolution signal, which notifies that a certain branch instruction is resolved to be taken or not; and ii) a page fault signal, which notifies that a certain instruction raised a page fault.

For example, when a branch instruction is fetched and then the branch resolution signal is sent from the Core (❶ in Figure 8-(a)), the taken column in the FetchTargetQueue is first updated (❷). Then, when the corresponding branch instruction successfully retires without any preceding page fault, BIMBranchPredictor is finally updated with the taken value.

However, when the page faulting instruction (e.g., load to invalid address) comes before the branch instruction, Boom does not update BIMBranchPredictor since the branch instruction will not retire. In this case, the Core sends two signals to the FetchTargetQueue individually (①), i.e., the page fault signal and the branch resolution signal. Thus, the order of the signals can be changed depending on the instructions as follow: 1) branch first, then page fault, 2) page fault first, then branch, and 3) branch and page fault at the same cycle. In the first case, the corresponding FetchTargetQueue entry of the branch instruction is updated as before (②). However, the entry is not written back to the BIMBranchPredictor and is flushed due to the page fault signal later. In the second case, the branch resolution signal is ignored since the corresponding entry would have been already flushed.

The problem arises in the third case. When the branch resolution signal and the page fault signal arrive at the same cycle (①), Boom incorrectly updates the taken value of the page faulting instruction (②), not the branch instruction. Thus, the updated taken value was finally written back to the BIMBranchPredictor (③) even if the branch instruction is flushed. SpecDoctor was able to find this subtle bug since it fuzzes transient instructions while monitoring the BIMBranchPredictor in phase 3. While this bug is not a security bug as it only degrades the prediction performance, SpecDoctor found that the bug can eventually lead to a transient execution attack.

Details of *Boombard*. Using the *Boombard* bug found by SpecDoctor, we manually constructed the exploit (i.e., *Boombard*). As initially configured by SpecDoctor phase 1, the threat model of the attack is a malicious kernel with a supervisor mode attacking a victim enclave with a machine mode [42]. Also, the *Boombard* attack is a Meltdown-type attack such that the attacker reads the secret beyond the isolation boundary enforced by the hardware (i.e., PMP [32]).

As shown in the code snippet in Figure 8-(b), we places following instructions in order: a page faulting load lw (i.e., line 5), secret load ld (i.e., line 9), dummy instructions andi (i.e., line 14-15), and secret dependent branch beqz (i.e., line 18). Then, we placed another branch bnez (i.e., line 25) at the address conflicting with the page faulting load lw (i.e., address using the same index in BIMBranchPredictor). Thus, the page faulting load lw is used to trigger a transient execution and transmit a secret value through the *Boombard* bug, and the conflicting branch bnez is used to receive the secret value. The dummy instructions in line 14-15 are used to raise the page fault and the branch resolution signals at the same cycle, so as to trigger the *Boombard* bug.

```
1  attacker:
2      rdcycle x10
3      call victim
4      rdcycle x11
5      /* Retrieve secret by measuring execution time */
6      sub x11, x11, x10
7  ...
8  victim:
9      ...
10     /* First division instruction */
11     div x18, x16, x17
12
13     /* Branch instruction depending on the first division */
14     beqz x18, L0
15     /* Branch is mispredicted to be not-taken */
16 transient:
17     /* Secret load */
18     la x19, secret
19     ld x20, 0(x19)
20     /* Second division instruction */
21     div x21, x21, x20
22 L0:
23     ...
24     ret
```

Figure 9: Code snippet of *Birgus*.

The exploit works in following steps: 1) resetting BIMBranchPredictor and other CPU states, 2) performing transient execution and transmitting the secret, and 3) receiving the secret. First, we reset the BIMBranchPredictor entry of conflicting branch bnez to always taken. Then, we reset other CPU states such as DCache or TLB so that the page fault and the branch resolution signals are deterministically asserted later.

After resetting the CPU, we trigger the transient execution with the page faulting load lw instruction. Then, the sophisticated transient instructions (i.e., secret load ld, dummy andi, and branch beqz) are transiently executed so that the *Boombard* bug is triggered. Therefore, the *Boombard* bug incorrectly updates the BIMBranchPredictor entry of the page faulting load lw, which is also conflicting with the branch bnez—Without the bug, BIMBranchPredictor should not be updated. The BIMBranchPredictor entry would be updated to either taken, or not-taken depending on the secret (i.e., the value of the register x21 of line 18). Thus, we finally receive the secret value by measuring the branch prediction result of the conflicting branch bnez instruction. The exploit successfully retrieved the secret data with an error rate under 5% and 19Kb/s of leakage rates assuming a CPU with 2GHz of the clock rate.

7.2 *Birgus* on RISC-V NutShell

Birgus is a Spectre-type variant found in RISC-V NutShell. The interesting aspect of *Birgus* is that it exploits a previously unknown gadget to construct a port contention side channel. The gadgets which construct a port contention side channel have been widely studied [10, 18]. However, *Birgus* was able to find an instruction pattern in RISC-V NutShell, which is different from the previously studied gadgets. This implies that the CPU verification relying on certain known patterns can be incomplete and we need a continuous verification to stop the transient execution vulnerabilities.

Details of *Birgus*. We provide the *Birgus* exploit as shown in Figure 9. The key insight of the attack is that the instruction under a mispredicted branch (i.e., line 21) can contend with the instruction that has already come before the branch (i.e., line 11). On top of that, the attacker can retrieve the secret value by observing the

induced side effect of the contention (i.e., an execution time of the `victim` function). Especially, *Birgus* exploits two implementation features of RISC-V NutShell: 1) port contention on division unit, and 2) variable latency of the division depending on the input value. As a result, the attacker is able to retrieve the secret by measuring the variable CPU cycles induced by the contention and different division latency on the secret values (i.e., line 2-6).

The vulnerable instructions (i.e., line 11-22 of the `victim`) contain two main parts: i) the first division instruction `div` (i.e., line 11) and the branch instruction `beqz` which depends on the `div` (i.e., line 14), and after the `beqz`, ii) the secret load instruction `ld` (i.e., line 19) and the second division instruction `div` which uses the loaded secret (i.e., line 21). Here, the branch `beqz` is mispredicted and the second part is transiently executed.

Since NutShell has only one division unit, the single division unit handles both the first `div` instruction and the second `div` instruction. This leads to the port contention on the division unit. In most cases, the first `div` instruction is handled earlier than the second `div`, but the second `div` can also be handled earlier if its dependency is resolved earlier than the first one. In this case, the computation of the first `div` is delayed until the second `div` is completed, even if the second `div` is a transient instruction. Meanwhile, the delayed cycle is affected by the secret value, as the second `div` computes on the secret (i.e., `x20`). For example, if the secret value is 0, the second `div` is completed in 1 cycle, otherwise it takes at most 64 cycles.

The problem is that the branch `beqz` depends on the result of the first `div` (i.e., line 11-14). Thus, the resolution of `beqz` is also delayed until the completion of the second, and then the first `div`. This makes the resolution time of the `beqz` depends on the secret value. Finally, the entire execution time of the function `victim` is affected, which enables the attacker to retrieve the secret value by observing the cycle differences. The exploit successfully retrieved the secret with the error rate under 1% and 216Kb/s of leakage rate.

8 DISCUSSIONS

Generalizing to Other Transient Execution Attacks. While we think the template based approach of SpecDoctor is general enough to capture all the transient execution vulnerabilities, the implementation should cover various CPU configurations and features. In the case of SpecDoctor's implementation, the current SpecDoctor cannot cover following types: 1) transient execution due to a memory-ordering violation [5], 2) MDS [4, 6, 29], and 3) self-modifying code [5].

The template needs to be extended to cover these types. The first two types (i.e., memory-ordering violation and MDS) can be covered by handling multi-threaded execution. MDS types would need a special catering, because the secret data should be transiently accessed through a temporary shared buffer by the attacker's thread. Thus, SpecDoctor needs an individual thread, victim and attacker threads, each of which continuously fills and probes these shared buffers, respectively. To cover the self-modifying code, the template and instruction generation of SpecDoctor need to be extended to include a specific memory region which can be repeatedly updated and executed.

In order to generalize the template, SpecDoctor needs to employ both the random instruction generation and the knowledge on previous attack patterns. As the transient execution vulnerabilities need complex configurations of the CPU, a pure random instruction generation may not easily construct such configurations. On the other hand, naive implementation which solely relies on the previous attack patterns cannot explore all the possible attack surfaces. Thus, the random instruction generation and the knowledge on the previous attack patterns should be properly balanced together to construct valid CPU configurations while further testing previously unknown configurations.

Positioning of SpecDoctor against Static Approaches. SpecDoctor employs a hybrid approach by statically analyzing the code for the timing side channels and dynamically fuzzing the instructions to find concrete PoCs. Static and dynamic approaches in SpecDoctor complement each other in two aspects: 1) reducing the search space, and 2) avoiding the false-positives.

a static approach of SpecDoctor reduces the search space of the fuzzer by identifying and instrumenting only the timing-change components. As the CPU contains enormous number of components (e.g., RISC-V Boom has 36,000 registers), it helps SpecDoctor minimize the searching effort and quickly find the instructions constructing a timing side channel.

On the other hand, a dynamic approach avoids the false positives as it finds concrete instructions through fuzzing. As the static analysis may produce false-positive cases (i.e., incorrectly identified timing-change components), SpecDoctor filters out such cases through the fuzzing, and faithfully confirms the identified vulnerabilities.

Comparing SpecDoctor against Regression Testing. We think SpecDoctor has two benefits over regression testing while developing a mitigation: 1) testing the soundness of the mitigation, and 2) assisting a root-cause analysis. First, SpecDoctor can provide better soundness than the regression testing. As the transient execution vulnerabilities are triggered by various components in the CPU, a mitigation attempt may not take account a certain component and thus it is still vulnerable to a variant. In this case, SpecDoctor has potentials to detect such a bypass case—as shown in §7, SpecDoctor is able to find similar variants if the original attack is provided. However, regression testing only checks the previously known patterns, and would not be able to find the variants.

Furthermore, SpecDoctor can assist developers to triage the root-cause of vulnerabilities. Understanding the root-cause of vulnerabilities requires non-trivial manual efforts as they originate from various parts of the CPU. SpecDoctor can help this process by providing information on the internal details of the CPU (e.g., engaged timing-changed components). For example, as shown in §6.5, SpecDoctor helps quickly identifying the root-causes of the found vulnerabilities. On the contrary, regression testing does not provide such information as it only performs end-to-end testing of the known patterns.

9 RELATED WORK

Transient Execution Attacks. Spectre [1] and Meltdown [2] were the first transient execution attacks. Since then, researchers have discovered numerous mechanisms leading to transient execution attacks. In the Spectre domain, several attacks exploited the return address stack to launch the transient executions [13, 44].

Table 6: Comparison of SpecDoctor versus related works.

	RTL verification	Attack type*	Available privileges	Can detect new† tran-trig	sec-tx	sec-rx	Disadvantages compared to SpecDoctor
UPEC [16]	○	M/S	All	△	△	○	Requires manually specified processor model
IntroSpectre [31]	○	M	All	×	△	×	High false-positive, not considering secret leakage
Revizor [43]	×	M/S	Partial (no-enclave)	△	△	×	Only D-cache side-channel, no in-depth analysis
Osiris [23]	×	-	-	×	×	○	Finds only side-channels
Transynther [8]	×	M	Partial (no-enclave)	×	○	×	Only Meltdown-type, D-cache side-channel
SpeechMiner [9]	×	M/S	Partial (no-enclave)	×	×	×	Cannot find new attacks
Absynthe [18]	×	-	-	×	×	△	Find only port contention side-channels
SpecDoctor	○	M/S	All	○	○	○	-

*M: Meltdown-type (including MDS), S: Spectre-type

†tran-trig: transient-trigger, sec-tx: secret-transmit, sec-rx: secret-receive

Jann found a Spectre variant using a speculative store bypass for a data flow misprediction [14]. Ragab et al. [5] discovered a speculation in FPU and memory ordering as a new source of the data flow misprediction. SmotherSpectre [10] introduced a new side-channel, i.e., port contention for leaking secret through the Spectre attack.

On the other hand, in the Meltdown domain, Julian exploited lazy floating point register handling in Intel CPUs [19]. ForeShadow [3] showed that the attack is even possible against the Intel SGX enclaves. RIDL [4] and ZombieLoad [29] opened MDS attacks by discovering that the shared buffers in CPU (e.g., a line fill buffer) can leak security sensitive data. Fallout [45] exploits the vulnerabilities in the store buffer. Van et al. [6] leaked the secret data while writing back the cache lines. CrossTalk [46] proved that the MDS attack is also available on the shared buffer outside the CPU.

Fuzz Testing. Since the introduction of AFL [11], Fuzzing has been widely used in the software community. Many existing fuzzers target user programs [30, 47–50]. Nowadays, fuzzer frameworks have been developed to verify kernel [12, 51], hypervisor [24], and even the CPU RTLs [25]. On top of the fuzzers, Petsios et al. [52] introduced the first differential fuzz testing, which finds semantic bugs in software programs. Especially, Nilizadeh [53] introduced DifFuzz which finds side-channels through differential fuzz testing, but it was not applied to CPU's micro-architectural side-channels.

Automated Approach to Find Transient Execution Attacks. Many previous works have tried to automatically find transient execution vulnerabilities in the CPU. We summarize the related works in Figure 6 as well as comparing those with SpecDoctor. UPEC [16] and IntroSpectre [31] are RTL-based techniques to find transient execution vulnerabilities. UPEC [16] performs a static analysis with a bounded model checking. However, UPEC requires non-trivial manual efforts as it is based on a manually specified processor model for each RTL implementation. On the contrary, SpecDoctor can work on any RTL implementation as long as the baseline implementation language (e.g., Chisel) is the same. IntroSpectre [31] performs a dynamic analysis to find potential MDS type vulnerabilities in a CPU RTL. Specifically, IntroSpectre finds shared buffers in the CPU which temporarily store the security sensitive data. However, IntroSpectre suffers from high false-positives as it does not find the way to transiently leak a secret from the found buffers, which is a key requirement of the transient execution attacks. SpecDoctor does not suffer from such false-positives as it finds unified transient execution vulnerabilities.

All the rest do not target RTL implementations. Revizor [43] applied a model-based relational testing to find transient execution vulnerabilities in blackbox CPUs. While Revizor provides concrete backgrounds for detecting the vulnerabilities, it cannot find new side-channels due to the blackbox nature of the targets. SpecDoctor can find comprehensive transient execution vulnerabilities including new side-channels thanks to the in-depth analysis on the RTL. Osiris [23] finds 1-bit micro-architectural side-channels but it does not discuss how to trigger a transient execution or exfiltrate the secret, which are all the key requirements of transient execution attacks. Transynther [8] introduced a framework to find new Meltdown type vulnerabilities by mutating the instructions in known attacks such as Meltdown, and RIDL.

10 CONCLUSION

This paper proposes SpecDoctor, an automated RTL fuzzer to find transient execution vulnerabilities. SpecDoctor introduces a configurable template and multi-phased fuzzer design to efficiently find various transient execution vulnerabilities. SpecDoctor proves its practical impact by finding transient execution vulnerabilities on two RISC-V CPUs (i.e., Boom and NutShell). Moreover, SpecDoctor found *Boombard*, which exploits a new implementation bug in RISC-V Boom, and *Birgus*, which introduces a previously unknown gadget to construct port contention side channel in NutShell. Especially, *Boombard* was assigned with a CVE number, demonstrating the strong implication of SpecDoctor on the security community.

11 ACKNOWLEDGEMENT

We would like to thank anonymous reviewers for their insightful comments. This work was partially supported by Supreme Prosecutor's Office of the Republic of Korea grant funded by Ministry of Science and ICT (No.1275000160), Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (No.2020-0-01840, Analysis on technique of accessing and acquiring user data in smartphone), and National Research Foundation (NRF) of Korea grant funded by the Korean government MSIT (NRF-2019R1C1C1006095). This work was also supported by SAMSUNG Research, Samsung Electronics Co., Ltd, under the title "Utilizing fuzz testing to verify chipset and firmware security". The Institute of Engineering Research (IOER) and Automation and Systems Research Institute (ASRI) at Seoul National University provided research facilities for this work.

REFERENCES

[1] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[2] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. Meltdown: Reading kernel memory from user space. In *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, August 2018.

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

[4] 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 *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[5] Hany Ragab, Enrico Barberis, Herbert Bos, and Cristiano Giuffrida. Rage against the machine clear: A systematic analysis of machine clears and their implications for transient execution attacks. In *Proceedings of the 30th USENIX Security Symposium (Security)*, Online, August 2021.

[6] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. Cacheout: Leaking data on intel cpus via cache evictions. In *Proceedings of the 42st IEEE Symposium on Security and Privacy (Oakland)*, Online, May 2020.

[7] Antonio Gonzalez, Fernando Latorre, and Grigorios Magklis. Processor microarchitecture: An implementation perspective. *Synthesis Lectures on Computer Architecture*, 5(1):1–116, 2010.

[8] Daniel Moghimi, Moritz Lipp, Berk Sunar, and Michael Schwarz. Medusa: Microarchitectural data leakage via automated attack synthesis. In *Proceedings of the 29th USENIX Security Symposium (Security)*, Boston, MA, August 2020.

[9] Yuan Xiao, Yinqian Zhang, and Radu Teodorescu. Speechminer: A framework for investigating and measuring speculative execution vulnerabilities. February 2020.

[10] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. Smotherspectre: exploiting speculative execution through port contention. In *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, London, UK, November 2019.

[11] M. Zalewski. American fuzzy lop. http://lcamtuf.coredump.cx/afl/.

[12] Dmitry Vyukov. Syzkaller: an unsupervised, coverage-guided kernel fuzzer, 2019.

[13] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative execution using return stack buffers. In *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, ON, Canada, October 2018.

[14] Jann Horn. Google project zero. speculative execution, variant 4: speculative store bypass. https://bugs.chromium.org/p/project-zero/issues/detail?id=1272.

[15] Risc-v boom's documentation. https://docs.boom-core.org/en/latest/index.html.

[16] Mohammad Rahmani Fadiheh, Johannes Müller, Raik Brinkmann, Subhasish Mitra, Dominik Stoffel, and Wolfgang Kunz. A formal approach for detecting vulnerabilities to transient execution attacks in out-of-order processors. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020.

[17] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. Hardware-software contracts for secure speculation. In *Proceedings of the 42st IEEE Symposium on Security and Privacy (Oakland)*, Online, May 2020.

[18] Ben Gras, Cristiano Giuffrida, Michael Kurth, Herbert Bos, and Kaveh Razavi. Absynthe: Automatic blackbox side-channel synthesis on commodity microarchitectures. In *Proceedings of the 2020 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2020.

[19] Julian Stecklina and Thomas Prescher. Lazyfp: Leaking fpu register state using microarchitectural side-channels. *arXiv preprint arXiv:1806.07480*, 2018.

[20] Yuval Yarom and Katrina Falkner. Flush+ reload: A high resolution, low noise, l3 cache side-channel attack. In *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, August 2014.

[21] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. Branchscope: A new side-channel attack on directional branch predictor. March 2018.

[22] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[23] Daniel Weber, Ahmad Ibrahim, Hamed Nemati, Michael Schwarz, and Christian Rossow. Osiris: Automated discovery of microarchitectural side channels. August 2021.

[24] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. Hyper-cube: High-dimensional hypervisor fuzzing. In *Proceedings of the 2020 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2020.

[25] Jaewon Hur, Suhwan Song, Dongup Kwon, Eunjin Baek, Jangwoo Kim, and Byoungyoung Lee. Difuzzrtl: Differential fuzz testing to find cpu bugs. In *Proceedings of the 42st IEEE Symposium on Security and Privacy (Oakland)*, Online, May 2020.

[26] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*, Boston, MA, June 2012.

[27] Changwoo Min, Sanidhya Kashyap, Byoungyoung Lee, Chengyu Song, and Taesoo Kim. Cross-checking semantic correctness: The case of finding file system bugs. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, October 2015.

[28] Yuting Chen, Ting Su, and Zhendong Su. Deep differential testing of jvm implementations. In *Proceedings of the 41th International Conference on Software Engineering (ICSE)*, Montreal, Canada, May 2019.

[29] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. Zombieload: Cross-privilege-boundary data sampling. In *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, London, UK, November 2019.

[30] Suhwan Song, Chengyu Song, Yeongjin Jang, and Byoungyoung Lee. Crfuzz: fuzzing multi-purpose programs through input validation. In *Proceedings of the 25th European Software Engineering Conference (ESEC) / 28st ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, Online, November 2020.

[31] Moein Ghaniyoun, Kristin Barber, Yinqian Zhang, and Radu Teodorescu. Introspectre: A pre-silicon framework for discovery and analysis of transient execution vulnerabilities. In *Proceedings of the 48st ACM/IEEE International Symposium on Computer Architecture (ISCA)*, Online, June 2021.

[32] Risc-v isa manual (privileged). https://riscv.org/specifications/privileged-isa/.

[33] Risc-v isa manual (unprivileged). https://riscv.org/specifications/unprivileged-isa/.

[34] Boom: Berkeley out-of-order machine. https://github.com/riscv-boom/riscv-boom.

[35] Nutshell, risc-v cpu developed by oscpu team. https://github.com/OSCPU/NutShell.

[36] Riscyoo: Risc-v out-of-order processors. https://github.com/csail-csg/riscy-OOO.

[37] The lizard core. https://github.com/cornell-brg/lizard.

[38] Chisel 3: A modern hardware design language. https://github.com/freechipsproject/chisel3.

[39] Firrtl:flexible intermediate representation for rtl. https://github.com/freechipsproject/FIRRTL.

[40] Chipyard, an agile risc-v soc design framework with in-order cores, out-of-order cores, accelerators, and more. https://github.com/ucb-bar/chipyard.

[41] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. Sonicboom: The 3rd generation berkeley out-of-order machine. In *Fourth Workshop on Computer Architecture Research with RISC-V*, 2020.

[42] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the 13th European Conference on Computer Systems (EuroSys)*, Crete, Greece, April 2020.

[43] Oleksii Oleksenko, Christof Fetzer, Boris Köpf, and Mark Silberstein. Revizor: Fuzzing for leaks in black-box cpus. *arXiv preprint arXiv:2105.06872*, 2021.

[44] Esmaeil Mohammadian Koruyeh, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *Proceedings of the 13th USENIX Workshop on Offensive Technologies (WOOT)*, Baltimore, MD, August 2019.

[45] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, et al. Fallout: Leaking data on meltdown-resistant cpus. In *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, London, UK, November 2019.

[46] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Crosstalk: Speculative data leaks across cores are real. In *Proceedings of the 42st IEEE Symposium on Security and Privacy (Oakland)*, Online, May 2020.

[47] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, October 2016.

[48] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[49] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing. In *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, August 2018.

[50] Caroline Lemieux and Koushik Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Montpellier, France, September 2018.

[51] Dae R Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. Razzer: Finding kernel race bugs through fuzzing. In *Proceedings of the*

40th IEEE Symposium on Security and Privacy (Oakland), San Francisco, CA, May 2019.

[52] Theofilos Petsios, Adrian Tang, Salvatore Stolfo, Angelos D Keromytis, and Suman Jana. Nezha: Efficient domain-independent differential testing. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2017.

[53] Shirin Nilizadeh, Yannic Noller, and Corina S Pasareanu. Diffuzz: differential fuzzing for side-channel analysis. In *Proceedings of the 41th International Conference on Software Engineering (ICSE)*, Montreal, Canada, May 2019.