Lost and Found in Speculation: Hybrid Speculative Vulnerability Detection

Mohamadreza Rostami[†], Shaza Zeitouni[†], Rahul Kande[‡], Chen Chen[‡], Pouya Mahmoody[†], Jeyavijayan (JV) Rajendran[‡], Ahmad-Reza Sadeghi[†]
[†]Technical University of Darmstadt, Darmstadt, Hessen, Germany [‡]Texas A&M University, College Station, Texas, USA

Abstract

Microarchitectural attacks represent a challenging and persistent threat to modern processors, exploiting inherent design vulnerabilities in processors to leak sensitive information or compromise systems. Of particular concern is the susceptibility of Speculative Execution, a fundamental part of performance enhancement, to such attacks.
We introduce *Specure*, a novel pre-silicon verification method composing hardware fuzzing with Information Flow Tracking (IFT) to address speculative execution leakages. Integrating IFT enables two significant and non-trivial enhancements over the existing fuzzing approaches: i) automatic detection of microarchitectural information leakages vulnerabilities without golden model and ii) a novel *Leakage Path* coverage metric for efficient vulnerability detection. *Specure* identifies previously overlooked speculative execution vulnerabilities on the RISC-V BOOM processor and explores the vulnerability search space 6.45× faster than existing fuzzing techniques. Moreover, *Specure* detected known vulnerabilities 20× faster.

CCS Concepts: • **Security and privacy** → **Security in hardware**.

Keywords: Hardware Fuzzing, Speculative Execution Vulnerability

ACM Reference Format:
Mohamadreza Rostami[†], Shaza Zeitouni[†], Rahul Kande[‡], Chen Chen[‡], Pouya Mahmoody[†], Jeyavijayan (JV) Rajendran[‡], Ahmad-Reza Sadeghi[†]. 2024. Lost and Found in Speculation: Hybrid Speculative Vulnerability Detection. In *61st ACM/IEEE Design Automation Conference (DAC '24), June 23–27, 2024, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3649329.3658469

1 Introduction

Hardware is the foundation of computing systems, and as such, insecure hardware exposes all critical systems to risk. In particular, attacks based on microarchitectural design and implementation flaws in processors constitute a severe threat in exploiting the hardware to extract sensitive data or compromise the whole computing system [2, 3, 8, 10, 17, 18, 20]. As designs advance and attackers evolve, we encounter increasingly sophisticated microarchitectural attacks [2, 3]. A crucial form of microarchitectural attacks exploits *Speculative Execution*, a fundamental part of processors' performance enhancement [2, 3, 17, 20]. Speculative execution technique predicts instruction outcomes before completion, reducing memory latency and mitigating pipeline stalls. However, speculative execution can lead to incorrect path forecasts, necessitating a rollback of changes made by transient instructions. Despite the rollback, some effects will still persist, serving as the root cause of speculative vulnerabilities. These vulnerabilities have been exploited to compromise systems' security by leveraging different channels to leak information from the microarchitectural layer to the architectural layer. These channels include direct channels such as architectural registers (e.g., Advanced Vector Extensions (AVX) registers [3], EFLAGS register [20], and Memory-Mapped-Input/Output (MMIO) interfaces [17]), as well as timing side/-covert channels [2, 8]. Researchers and manufacturers are investigating new defenses and countermeasures to protect against these attacks. However, unlike software vulnerabilities that can be patched digitally, hardware patches often require physical modifications or replacements. Thus, detecting such vulnerabilities before fabrication is imperative.
However, existing approaches for automated detection of speculative execution vulnerabilities suffer from various deficiencies: They (i) are often restricted to known attack vectors [11, 15, 16], hence, missing opportunities for detecting unknown vulnerabilities and their underlying causes, (ii) focus on specific microarchitectural elements rather than covering the entire processor-under-test (PUT) [16], (iii) suffer from state explosion [14], (iv) require additional hardware components [11], or (v) require a golden reference model [19]. These limitations reduce flexibility and increase manual effort and hardware overhead. Nonetheless, a more comprehensive and

Email: [†]{mohamadreza.rostami,shaza.zeitouni,pouya.mahmoody,ahmad.sadeghi}@trust.tu-darmstadt.de [‡]{rahulkande,chenc,jv.rajendran}@tamu.edu

scalable technique for detecting hardware vulnerabilities is crucial to meet the field's evolving needs.

Our Contributions. We present a novel pre-silicon verification method, *Specure*, effectively addressing critical aspects of speculative execution vulnerability detection. *Specure*, a hardware-agnostic and non-invasive solution, detects speculative execution vulnerabilities in real-world size processors without necessitating additional hardware, modifications, or the use of golden reference models. By composing hardware fuzzing and Information Flow Tracking (IFT), our approach efficiently identifies and quantifies direct channels within the processor design, revealing potential information leakage pathways and pinpointing their root cause in the hardware design. Overcoming challenges in coherent integration, we develop a strategy to express these channels and establish a novel coverage feedback mechanism for the fuzzer. Introduced coverage metric differs significantly from previous methods that solely desired to increase code coverage, blindly relying on luck to trigger vulnerabilities, thereby ensuring a comprehensive exploration of processor design and potential vulnerabilities. Our key contributions include:

Automated detection of direct channels and their root causes: *Specure* leverages IFT to visualize the direct channel leakages between the processor's microarchitectural and architectural components. This is done by deriving the processor's Information Flow Graph (IFG) and extracting pathways from microarchitectural to architectural registers.

Novel coverage metric. We propose a unique *Leakage Path* (LP) coverage for speculative execution leakages. LP coverage provides two advantages over traditional code coverage-based fuzzers: (i) it identifies new and unknown vulnerabilities due to its fine-grained IFT-based guidance, and (ii) it discovers vulnerabilities more efficiently than other fuzzers.

Implementation. We implement *Specure* using both open-source and commercial RTL simulators. To demonstrate its effectiveness, we use BOOM [12], the most advanced open-source out-of-order processor, as the PUT.

Evaluation. We validate our approach on BOOM by coverage analysis and vulnerability detection. With the novel coverage metric, *Specure* explores the vulnerability search space 6.45× and detects known vulnerabilities 20× faster than existing fuzzing techniques. We further validate our approach to detect recently discovered vulnerabilities [3, 20] that remain undetectable by current methods. For that, we successfully reproduced the speculative execution vulnerabilities [3, 20] in x86 architecture on BOOM.

As we will explain in detail in the evaluation section (§ 4), recent critical speculative execution vulnerabilities on commercial processors are derived from advanced microarchitectural features [2, 3, 17, 18, 20]. It is essential to stress that due to the absence of these features in open-source RISC-V processors, we emulate the behaviors of two recent and critical vulnerabilities on BOOM. This demonstration aims to highlight the effectiveness of our hybrid fuzzer and the novel coverage metric in discovering and analyzing the root causes of such vulnerabilities.

2 Background & Related Work

Direct Channels for Speculative Execution Attacks. Direct channels enable attackers to leak microarchitectural information directly to the architecture level without relying on timing or power side channels, which require precise measurements and side-channel interfaces [2, 3, 17, 20]. Zhang et al. [20] showcased the exploitation of vulnerable `umonitor` and `umwait` instructions on recent Intel processors. These instructions optimize power states without interrupts, allowing information leakage during transient execution. *Zenbleed* [3] exploits zeroing registers optimization in AMD CPUs. When calling the zeroing instruction `vzeroupper` in the mispredicted speculative window, it will zero the upper bits of the register vector and deallocate dependencies of the vector and can allocated by the victims thread. *Zenbleed* was detected by fuzzing, showing the effectiveness of fuzzing in detecting vulnerabilities in large-scale designs.

Fuzzing or fuzz testing provisions test inputs to the software/hardware under test to uncover faults or vulnerabilities that traditional testing methods may miss [7]. The fuzzer generates random, malformed, or unusual inputs to test how the program handles them. The initial set of test inputs, *seeds*, can be automatically generated or manually crafted by verification engineers. During each fuzzing round, the fuzzer mutates the optimal test inputs from the preceding round using operations like *bit/byte flipping*, *swapping*, *deleting*, or *cloning* to generate new inputs. Fuzzing has gained prominence in the security community due to its automation, cost-effectiveness, and scalability [11, 13, 16, 19].

Fuzzing for Speculative Vulnerabilities. Specdoctor [11] introduced a pre-silicon fuzzer that utilizes differential fuzzing with varied secret values to detect speculative execution vulnerabilities. It identifies potentially vulnerable modules based on known attacks and monitors them for discrepancies during fuzzing. However, Specdoctor's restricted selection criteria of known potentially vulnerable modules limits its ability to confirm exploitability and overlooks vulnerabilities. On the other hand, Introspectre's [15] approach to detecting Meltdown-like vulnerabilities demands significant manual effort for preparing Meltdown gadgets, i.e., code snippets. The fuzzer is only responsible for randomly selecting gadgets, assigning random values to parameters within gadgets, and connecting gadgets to make a test case. Note that both works [11, 15] have been evaluated using BOOM. In [16], a post-silicon fuzzer to detect speculative vulnerabilities in x86 CPUs is presented. To identify vulnerabilities, [16] determines whether the fuzzer can induce identical architectural states with differing data cache states. This work is limited to detecting Spectre-like vulnerabilities and capturing timing leakage only in the L1 Data cache.

Information Flow Tracking (IFT) ensures sensitive data is not leaked or improperly accessed by monitoring the flow of information within a system. IFT tags data with labels that indicate their security level and tracks these tags as data moves through different components. Based on security specifications, IFT can inform if sensitive data leaks to insecure destinations [1]. Solt et al. [9] presented a dynamic IFT tool for detecting Spectre and Meltdown attacks. However, this approach has runtime overhead and requires changes to the underlying processor.

3 Design

This section introduces our hybrid approach that combines fuzzing and IFT to identify speculative execution vulnerabilities in the pre-silicon. We focus in this work on the detection of direct leakage channels. A speculative execution attack requires first leaking information to microarchitectural components within the speculative window and then retrieval of leaked information from the microarchitectural layer to the architectural layer. To detect vulnerabilities triggered by speculative execution, *Specure* entails two phases. In the Offline Phase (§ 3.1), *Specure* employs static IFT to identify potential leakage locations and paths within the PUT that could directly leak information from microarchitectural registers to architectural registers. In the Online Phase (§ 3.2), *Specure* employs fuzzing to generate inputs that trigger speculative execution. Then, *Specure* detects leakage points within the PUT using a hyper-property aligned with the speculative execution vulnerabilities previously introduced definition.

3.1 Offline Phase

This phase leverages IFT to extract information flows within the PUT. The process is performed statically, using the processor's RTL model, and in two steps, as shown in Figure 1. In **Step 1**, the information flow graph (IFG) is extracted from the PUT's RTL model. IFG is a directed graph that accurately represents the information flow within the hardware RTL model expressed as $IFG = (R, F)$. R represents the set of all signals in the PUT. F represents the connections between signals defined as $F \subseteq \{(v, u) \mid v, u \in R \textit{ and } v \neq u\}$, where (v, u) represents an edge or a connection between vertex v, the source signal, and vertex u, the destination.

Listing 1. Example RTL code: a *top* module with two D-FFs.

```
module D_FF(input d, input clk, output q);
  reg q;
  always @(posedge clk)
    q <=  d;
endmodule
module top(input clk, input i, output o);
  reg q1;
  D_FF df1(.d(i), .clk(clk), .q(q1));
  D_FF df2(.d(q1), .clk(clk), .q(o));
endmodule
```

Listing 1 shows a simple RTL example, a *top* module that implements a two clock-cycle delay using two D-flip-flops (D-FFs). In the *IFG*, R is defined as:

$$R = \{top.q1,\ top.clk,\ top.i,\ top.o,\ top.df1.d,\ top.df1.q,\ top.df1.clk,\ top.df2.d,\ top.df2.clk,\ top.df2.q\},$$

and F is defined as:

$$F = \{(top.clk,\ top.df1.clk), (top.clk,\ top.df2.clk), (top.i,\ top.df1.d), (top.df1.d,\ top.df1.q), (top.df1.q,\ top.q1), (top.q1,\ top.df2.d), (top.df2.d,\ top.df2.q), (top.df2.q,\ top.o)\}.$$

In **Step 2**, the PUT's IFG is used to extract all potential direct leakage channels (*PDLC*) by extracting all paths from every microarchitectural register to all architectural registers. A direct leakage channel can be visualized in the IFG as a chain of edges starting from a microarchitectural register and ending in an architectural register. *Specure* first identifies and labels all architectural registers in the set of all signals R obtained in Step 1. To distinguish the architectural registers from the microarchitectural registers in R, we parsed the latest privileged and unprivileged RISC-V instruction set architecture (ISA) specification [4] and automatically extracted programmer-accessible registers, including memory-mapped registers. We apply the skewed-aware join approach to reduce the computation complexity of path extraction. We reverse all paths in the IFG and search for routes that start from architectural registers to all microarchitectural registers, This approach reduces the algorithm's complexity from $O(V^2)$ to $O(V)$, where V is the number of all registers in the PUT (i.e., $|R|$). The output of Offline Phase includes the PUT's IFG and the *PDLC* list.

3.2 Online Phase

The processing steps of *Specure*'s Online Phase consist of five components as shown in Figure 1. In the following, we provide a detailed description of each component.

Hardware Fuzzer aims to effectively explore the search space for potential vulnerabilities and cover as many items in the PDLC list as possible. In the scope of this work, *search space* pertains to the collection of all possible combinations of instructions that cause the PUT to enter a state where it leaks information during the speculative execution window. To boost vulnerability detection, we integrated special input seeds into the fuzzer alongside random seeds. The special seeds have transient execution windows covering scenarios like branch misprediction, branch target injection, and return stack buffer manipulation. Incorporating such seeds into the initial list accelerates the discovery of transient execution leaks, thus, enabling the fuzzer to faster trigger vulnerabilities compared to relying solely on random seeds.

Microarchitecture Visualizer receives test inputs from the Hardware Fuzzer and simulates them on the PUT. Then, it extracts the typical code coverage metrics (toggle, branch,

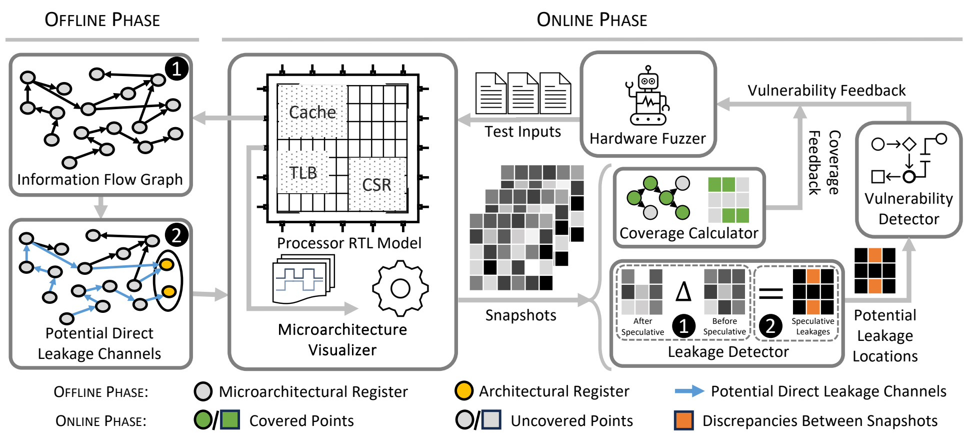

Figure 1. Overview of *Specure*. The Offline Phase leverages IFT technique to generate the *IFG* and *PDLC* of an RTL model. The Online Phase leverages them to identify the existence and pinpoint the locations of speculative vulnerabilities.

Table 1. The start and end cycles of Misspeculated Windows.

ID	Start	End	Instruction	Instruction(Readable)
1	34594	34625	FBEC52E3	BGE S8, T5, 0x800025B0
2	89991	90121	FB6F42E3	BLT T5, S6, 0x800025A0
...	...	...	...	...
M	45322	45348	FBAC5CE3	BGE S8, S10, 0x800025C0

finite-state machine (FSM), etc.), execution traces, and waveforms that show PUT's signal values for each simulation clock cycle. Using the waveforms, snapshots are generated from the microarchitectural and architectural signals. Each snapshot corresponds to a single clock cycle and contains the values of the PUT's signals, i.e., state, at that clock cycle. The snapshot represents the PUT's states.

Leakage Detector identifies potential information leakage locations within speculative windows in two steps:

In **Step 1**, the start and end of each speculative window are defined. This can be done by tracing speculative execution indicators, such as the processor's Re-order Buffer (RoB), which ensures the in-order commitment of executed instructions in an out-of-order processor. For example, the RoB's in-queue of BOOM receives multiple micro-operations. Each, equivalent to an instruction in BOOM, contains a signal, `unsafe`, indicating whether this micro-operation starts a speculative window. To find the end cycle of that speculative window, RoB receives signals, such as `brupdate`, from the branch predictor to confirm the (mis)prediction. Using these signals, which can be easily extracted from all snapshots generated by the Microarchitecture Visualizer, we could detect each speculative window's start and end clock cycles. We use this information and maintain a table, called *Misspeculation Table* (*MST*), that keeps the start and end clock cycles and the related instruction for each misspeculated window, as illustrated in Table 1. Inputs generated by the Hardware Fuzzer could have zero or more speculative windows.

In **Step 2**, the discrepancies between the snapshots corresponding to the start and end of each speculative window are computed. These discrepancies represent potential information leakage locations, shown in orange in Figure 1. Note that while the microarchitectural state changes due to the execution of the speculative window, not all of these changes indicate the existence of information leakage. Therefore, potential information leakage locations are forwarded to the Vulnerability Detector for detecting direct-channel leakage.

Vulnerability Detector detects direct-channel leakage, i.e., changes in the architectural state due to the execution of a misspeculated window. By cross-referencing the altered architectural registers with the PDLC list, we can determine the microarchitectural registers responsible for leaking information to these architectural registers, thereby pinpointing the root cause of the vulnerability. This approach effectively eliminates the need for a golden model, as we leverage the difference snapshot to identify mismatches due to speculative execution. If the architectural state has not changed during the speculative window, this only implies that the corresponding input caused no direct leakage from the microarchitectural to the architectural layer. However, it does not negate the existence of side-channel leakage, which is out of the scope of this work. Vulnerability Detector provides feedback on the presence of vulnerabilities to guide Hardware Fuzzer generating new inputs.

Coverage Calculator receives the snapshots and the PDLC list (§ 3.1) to compute our novel *Leakage Path* (LP) coverage. The *LP* metric aims to guide Hardware Fuzzer to further explore potential direct leakage channels during speculative execution, thereby increasing the chances of triggering speculative execution vulnerabilities. It computes the *LP* coverage based on the number of times the PLDC signals toggled during the speculative window.

4 Evaluation Results

We evaluated our approach using the most intricate, open-source RISC-V out-of-order processor, BOOM [12]. We used

Chipyard [6] as the simulation environment for BOOM. We conducted experiments on a 32-core, 2.6 GHz Intel Xeon processor with 512 GB RAM running Linux-based Cent OS.

4.1 Offline Phase Evaluation

IFG. We employed Pyverilog [5] to parse BOOM's Verilog code into an abstract syntax tree (AST) and generate its IFG. This step took around 9 minutes for BOOM and is required once per PUT. The resulting $IFG(R, F)$ includes 162, 631 signals in R and 428, 245 connections in F.

PDLC. As outlined in § 3.1, we extracted BOOM's architectural registers using the RISC-V ISA documentation. Subsequently, we employed the Depth-First Search (DFS) algorithm to determine all paths from BOOM's microarchitectural to architectural registers in around 3 minutes. The total number of potential direct leakage channels is 9, 048.

4.2 Online Phase Evaluation

Specure effectively detected two of the most recent direct speculative execution vulnerabilities [3, 20], which cannot be discovered by state-of-the-art hardware fuzzers [11, 13, 16, 19] because of two main advantages of *Specure*: (1) a novel vulnerability detection mechanism that employs fine-grained microarchitectural snapshots of the PUT to trace dynamic information flows and spot misbehaviors, i.e., information leakage, due to speculative execution. Owing to this feature, *Specure* is more accurate than existing fuzzers that rely on comparing execution traces with a golden-reference model [13, 16, 19] or differential fuzzing [11], (2) a novel fine-grained coverage metric, *Leakage Path* (LP), which guides the fuzzer toward potential information leakage channels utilizing static IFT, in contrast to general code coverage feedback [11, 13, 16, 19].

Since (M)WAIT [20] and Zenbleed [3] attacks exploit advanced optimization features, sleep on monitor address and zeroing registers optimization, which are not yet implemented in RISC-V ISA, we first discuss how we emulated their behaviors in BOOM. Then, we compare the effectiveness of *Specure* with state-of-the-art hardware fuzzers in vulnerability detection [11, 19]. Unlike the requirements imposed by *SpecDoctor* [11], *Specure* requires no PUT instrumentation or hardware modifications, thereby introducing no additional simulation overhead. However, *Specure* still incurs a runtime overhead of 82% higher than *TheHuzz* [19] due to snapshots processing and coverage metric computation. Finally, we demonstrate the efficacy of our coverage metric.

Emulating `(M)WAIT` [20]. We introduced a minor modification to BOOM to emulate the (M)WAIT vulnerability. The original (M)WAIT attack [20] requires the `umonitor` instruction to invoke a monitor on a memory address, and the `umwait` instruction sets the core to sleep. The core wakes up when changes occur at the pertinent memory location or after a predetermined amount of time. To maintain generality and avoid unnecessary complexities, we extend the functionality of the RISC-V ISA without introducing new non-standard instructions. We opted to incorporate three new CSR registers that provide the necessary functionality for emulating the (M)WAIT vulnerability. The new CSRs are `mwait_en`, `monitor_addr`, and `mwait_timer`. To emulate (M)WAIT vulnerability behavior on a single-core BOOM, the user sets the `monitor_addr` for monitoring. Activating the waiting behavior involves initiating the timer by writing one to `mwait_en`. If changes occur in the monitored memory (`monitor_addr`), the `mwait_timer` is set to zero. If the timer reaches zero, it is set to one. We modified BOOM's data cache to turn off the timer not only with memory location but also with corresponding cache line changes to implement the root cause of the attack.

Table 2. *Specure*'s vulnerability detection effectiveness compared to prior works (*"e.m." indicates emulated*).

Paper	Spectre v1	Spectre v2	(M)WAIT e.m.	Zenbleed e.m.
[11]	✓	✓	✗	✗
[14]	✓	✓	✗	✗
Specure	✓	✓	✓	✓

Emulating `Zenbleed` [3]. To emulate Zenbleed [3] in a single-core environment, we simplified the vulnerability constraints while maintaining its generality and root cause, which is the change of a general-purpose register within a mispredicted speculative window that remains after closing the window. To avoid adding non-standard instructions, we introduced a new CSR register, `zenbleed_en`. We modified the `Rename Stage` of the BOOM pipeline by manipulating the `maptable` rollback mechanism to prevent the rollback of `Register File` changes when `zenbleed_en` is set to a non-zero value.

Note that emulating the (M)WAIT [20] and Zenbleed [3] attacks on a single-core open-source RISC-V processor, inherently lacking advanced optimization features like register zeroing and sleep on monitor address, is a proof of concept for *Specure*'s efficacy of identifying direct speculative execution vulnerabilities. Our emulation did not dismiss the original vulnerabilities' generality and root cause, i.e., architecture-visible changes during speculative execution. Moreover, following modifications, we built and tested BOOM to confirm its functionality correctness.

Detecting Emulated (M)WAIT and Zenbleed. To demonstrate the effectiveness of *Specure*, we compared it with *SpecDoctor* [11], the state-of-the-art hardware pre-silicon fuzzer for detecting speculative execution vulnerabilities. We initialized both fuzzers in the same condition to run for 24 hours. After approximately 14 hours and 4.5 hours, *Specure* triggered the emulated (M)WAIT and Zenbleed vulnerabilities, respectively. *Specure* detected the direct information leakage path as the root cause. The root cause report showed a direct leakage path between the data cache and `mwait_timer` CSR register for (M)WAIT and a direct leakage path between

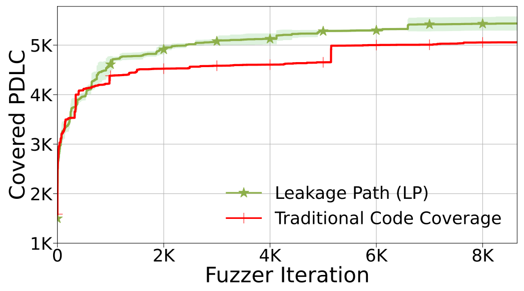

Figure 2. Coverage analysis of traditional code coverage, and *Specure*'s novel coverage metric.

`zenbleed_en` CSR register and general purpose register file and rename module for `Zenbleed`. These two vulnerabilities are categorized as CWE-1342. SpecDoctor [11] practically could not detect these vulnerabilities within 24 hours. However, considering its vulnerability detection mechanism, which relies on differential fuzzing with varied secrets and monitors the hash values of instrumented components for mismatches, it cannot detect these vulnerabilities because: (1) limitation in instrumented microarchitectural components, (2) generating random inputs without using a fine-grained coverage metric, and (3) varied secret-based vulnerability detection will overlook information leakages that do not directly reflect the secret value.

Detecting Spectre Vulnerabilities. We evaluate *Specure*'s capability in detecting known speculative execution vulnerabilities, i.e., Spectre variant 1 and 2 [18]. For this test, we added a data cache to the PDLC list to be monitored by Vulnerability Detector. To assess *Specure*'s performance, we compared it again against *SpecDoctor* [11], which reported that it found Spectre vulnerability in 31 hours. However, *Specure* detected this vulnerability 20× faster than state-of-the-art, after 1.5 hours and 49 minutes, without and with initial seeds with speculative window, respectively.

Coverage Analysis. We performed a comparative evaluation of novel *LP* coverage against traditional coverage metrics [19] in terms of vulnerability detection efficiency, i.e., PDLC activation. In this experiment, we ran *Specure* with two different coverage metrics as feedback: (1) novel *Leakage Path* (LP) and (2) traditional *Code Coverage* (e.g. FSM, toggle, branch, and condition). Figure 2 presents `BOOM`'s covered PDLCs during fuzzing using traditional *code coverage* and *LP* coverage metric. Each experiment was repeated three times, and the plot represents the mean value of all experiments. Figure 2 shows that the number of explored PDLCs achieved by fuzzer guided by code coverage lags in worst cases by 10.2% behind the fuzzer guided by *LP* coverage. Moreover, fuzzer guided by *LP* coverage achieves the same coverage value with only 798 iterations, compared to the 5149 iterations required by code coverage guided fuzzer. This indicates that by employing *LP*, *Specure* explores the search space for speculative execution vulnerabilities 6.45× faster than traditional code coverage metrics.

5 Conclusion

In this paper, we proposed *Specure*, a pre-silicon verification approach that combines hardware fuzzing and IFT to detect speculative execution vulnerabilities in processors. *Specure* provided a comprehensive approach for addressing the challenges of integerating hardware fuzzing and IFT. By leveraging the novel *Leakage Path* coverage metric, *Specure* offers an efficient hardware-agnostic solution for speculative information leakage detection. In doing so, *Specure* contributes to advancing proactive hardware security verification.

References

[1] H. Wei et al. 2021. Hardware Information Flow Tracking. *Comput. Surveys.*
[2] Daniel Moghimi. 2023. Downfall: Exploiting Speculative Data Gathering. In *USENIX Security.*
[3] T. Ormandy. 2023. Zenbleed. https://bit.ly/zenbleed.
[4] RISC-V. 2021. The RISC-V Instruction Set Manual Volume I, II.
[5] S. Takamaeda-Yamazaki. 2015. Pyverilog: A python-based hardware design processing toolkit for verilog hdl. In *ARC.* Springer.
[6] A. Amid *et al.* 2020. Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs. *IEEE Micro.*
[7] A. Fioraldi *et al.* 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *USENIX Workshop on Offensive Technologies.*
[8] B. Gras *et al.* 2018. Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks. In *USENIX Security.*
[9] F. Solt *et al.* 2022. CellIFT: Leveraging Cells for Scalable and Precise Dynamic Information Flow Tracking in RTL. *USENIX Security.*
[10] G. Dessouky *et al.* 2019. HardFails: Insights into Software-Exploitable Hardware Bugs. *USENIX Security.*
[11] J. Hur *et al.* 2022. SpecDoctor: Differential Fuzz Testing to Find Transient Execution Vulnerabilities. In *ACM SIGSAC CCS.* ACM.
[12] K. Asanovic *et al.* 2015. *The Berkeley Out-of-Order Machine (BOOM): An industry-competitive, synthesizable, parameterized RISC-V processor.* Technical Report.
[13] K. Laeufer *et al.* 2018. RFUZZ: coverage-directed fuzz testing of RTL on FPGAs. In *IEEE/ACM ICCAD.*
[14] M. Fadiheh *et al.* 2022. An exhaustive approach to detecting transient execution side channels in RTL designs of processors. *IEEE Trans. Comput.*
[15] M. Ghaniyoun *et al.* 2021. INTROSPECTRE: A Pre-Silicon Framework for Discovery and Analysis of Transient Execution Vulnerabilities. *ACM/IEEE Annual International Symposium on Computer Architecture.*
[16] O. Oleksenko *et al.* 2023. Hide and Seek with Spectres: Efficient discovery of speculative information leaks with random testing. *IEEE Symposium on Security and Privacy.*
[17] P. Borrello *et al.* 2022. ÆPIC Leak: Architecturally Leaking Uninitialized Data from the Microarchitecture. In *USENIX Security.*
[18] P. Kocher *et al.* 2019. Spectre Attacks: Exploiting Speculative Execution. In *IEEE Symposium on Security and Privacy.*
[19] R. Kande *et al.* 2022. TheHuzz: Instruction Fuzzing of Processors Using Golden-Reference Models for Finding Software-Exploitable Vulnerabilities. *USENIX Security.*
[20] R. Zhang *et al.* 2023. (M)WAIT for It: Bridging the Gap between Microarchitectural and Architectural Side Channels. In *USENIX Security.*

Acknowledgment. This work was partially funded by Intel's Scalable Assurance Program, Deutsche Forschungsgemeinschaft (DFG) – SFB 1119 – 236615297, the European Union under Horizon Europe Programme – Grant Agreement 101070537 – CrossCon, the European Research Council under the ERC Programme - Grant 101055025 - HYDRANOS, the US Office of Naval Research (#N00014-18-1-2058), and the Lockheed Martin Corporation.