

DejaVuzz: Disclosing Transient Execution Bugs with Dynamic Swappable Memory and Differential Information Flow Tracking assisted Processor Fuzzing

Jinyan Xu
Zhejiang University
Hangzhou, Zhejiang, China
phantom@zju.edu.cn

Yangye Zhou
Zhejiang University
Hangzhou, Zhejiang, China
zhouyangye@zju.edu.cn

Xingzhi Zhang
Zhejiang University
Hangzhou, Zhejiang, China
xingzhizhang@zju.edu.cn

Yinshuai Li
Southern University of Science and
Technology
Shenzhen, Guangdong, China
liys2022@mail.sustech.edu.cn

Qinhan Tan
Princeton University
Princeton, New Jersey, USA
qinhant@princeton.edu

Yinqian Zhang
Southern University of Science and
Technology
Shenzhen, Guangdong, China
yinqianz@acm.org

Yajin Zhou
Zhejiang University
Hangzhou, Zhejiang, China
yajin_zhou@zju.edu.cn

Rui Chang
Zhejiang University
Hangzhou, Zhejiang, China
crix1021@zju.edu.cn

Wenbo Shen
Zhejiang University
Hangzhou, Zhejiang, China
shenwenbo@zju.edu.cn

Abstract

Transient execution vulnerabilities have emerged as a critical threat to modern processors. Hardware fuzzing testing techniques have recently shown promising results in discovering transient execution bugs in large-scale out-of-order processor designs. However, their poor microarchitectural controllability and observability prevent them from effectively and efficiently detecting transient execution vulnerabilities.

This paper proposes DejaVuzz, a novel pre-silicon stage processor transient execution bug fuzzer. DejaVuzz utilizes two innovative operating primitives: dynamic swappable memory and differential information flow tracking, enabling more effective and efficient transient execution vulnerability detection. The dynamic swappable memory enables the isolation of different instruction streams within the same address space. Leveraging this capability, DejaVuzz generates targeted training for arbitrary transient windows and eliminates ineffective training, enabling efficient triggering of diverse transient windows. The differential information flow tracking aids in observing the propagation of sensitive data across the microarchitecture. Based on taints, DejaVuzz designs the taint coverage matrix to guide mutation and uses taint liveness annotations to identify exploitable leakages. Our evaluation shows that DejaVuzz outperforms the state-of-the-art fuzzer SPECDOCTOR, triggering more comprehensive transient windows with lower training overhead and achieving a 4.7 \times coverage improvement. And DejaVuzz also mitigates control flow over-tainting with acceptable overhead and identifies 5 previously undiscovered transient execution vulnerabilities (with 6 CVEs assigned) on BOOM and XiangShan.

1 Introduction

The recent discovery of transient execution vulnerabilities has unveiled a significant threat to modern processors. These vulnerabilities, such as Spectre [21] and Meltdown [25], exploit speculative execution, a key performance optimization feature, to leak sensitive data through side channels. The ongoing battle between attackers and defenders resembles a continuous cat-and-mouse game. For example, Spectre-V2 [21] promoted the privilege-isolated branch prediction deployment, but follow-up research soon discovered bugs [3, 44, 50] in other speculation components. Similarly, after Fore-shadow [45] was patched, Microarchitectural Data Sampling (MDS) [4, 46] attacks emerged. This arms race not only challenges the efficacy of existing defense mechanisms but also underscores the necessity of a proactive approach to automated transient execution bug detection.

Some efforts [27, 28, 51] have been applied to commodity processors. However, due to the black-box nature of off-the-shelf processors, these approaches rely heavily on template-based generation and fixed side channels, which makes it difficult for them to uncover new vulnerabilities. On the contrary, detection approaches at the pre-silicon stage have yet to be extensively studied. Detecting these vulnerabilities during the Register Transfer Level (RTL) development phase is crucial, as hardware bugs are usually difficult to fix once the design is manufactured. Early detection allows for timely remediation, preventing these bugs from being integrated into production hardware. Therefore, proactive testing and verification at the pre-silicon stage is imperative for ensuring processor microarchitecture security.

Formal verification and fuzzing are commonly used methods for existing processor RTL transient execution bug detection. Although formal approaches can prove security properties exhaustively, limited by the state explosion problem, existing methods [9, 39, 43, 55] solve the scalability problem by modeling processor transient execution behavior at a higher level of abstraction. However, given the complexity of the out-of-order processor design, the microarchitecture implementation details ignored by the model are highly error-prone [18, 53]. Furthermore, the complicated design pre-knowledge and heavy manual efforts required for hardware modeling and security property definition also impede applying formal methods to complex designs.

Recently, processor fuzzing has demonstrated promising results in verifying large-scale complex processor designs [5, 19, 20, 36, 53], and researchers also have begun applying fuzzing to detect transient execution vulnerabilities [11, 12, 18]. INTROSPECTRE [12] and TEESEC [11] use gadget templates to generate Meltdown-type transient execution vulnerabilities and identify leakage by searching for secret values in the microarchitecture logs. SPECDOCTOR [18], on the other hand, employs a multi-phase random instruction generation process and utilizes differential testing to detect sensitive data leakage. However, due to the complexity of the transient execution vulnerabilities, current fuzzing methods are either too limited [11, 12], only capable of identifying specific leakage patterns, or too inefficient [18], taking days to complete the detection, thereby limiting their practical adoption. To effectively and efficiently fuzz transient execution bugs, the following two challenges need to be addressed.

First, only transiently executed instructions are considered effective fuzzing payloads, so the fuzzer needs to efficiently trigger diverse transient windows for fuzzing. However, triggering these transient windows requires deliberate microarchitecture training. Due to significant differences in training patterns among various microarchitecture components, existing approaches generate limited transient windows with high training overhead (§6.2). The inability to generate various transient windows means the microarchitecture cannot be fully explored. Additionally, ineffective training instructions waste simulation time, increasing training overhead and reducing the fuzzing throughput.

Second, the fuzzer needs to perceive the propagation of sensitive data during transient execution to guide mutation and detect leakages. Information flow tracking is a promising solution, but it suffers from the control flow over-tainting problem in complex designs [37]. Due to the lack of effective methods to trace sensitive data, existing fuzzers cannot measure coverage or identify exploitable leakages (§6.3). Lacking coverage metrics means that the quality of stimuli cannot be assessed, leading to inefficient input mutation. Passing unexploitable leakages to subsequent stages not only results in false positives but also makes later phases futile, further misguiding the fuzzing process.

To address the challenges mentioned, we propose DeJaVuzz, an effective and efficient pre-silicon processor fuzzer for transient execution vulnerabilities, powered by two novel operating primitives: dynamic swappable memory and differential information flow tracking. Dynamic swappable memory serves as an isolation primitive, responsible for transparently switching instruction sequences to control the microarchitecture to trigger desired transient execution behaviors. This primitive resolves conflicts between instruction sequences by time-sharing the address space. To increase the diversity of triggered transient windows, DeJaVuzz isolates training and transient instruction sequences to generate arbitrary transient windows, and uses the training derivation strategy to derive targeted training based on transient execution information. To reduce the training overhead, DeJaVuzz isolates each training instruction sequence to explore different training effects, and eliminates ineffective training through the training reduction strategy. Differential information flow tracking acts as the tracing primitive that is responsible for observing microarchitecture state changes caused by sensitive data. This primitive eliminates the control flow over-tainting problem by comparing whether different secrets can produce different selections on the same control signal. With the help of taints, DeJaVuzz designs a taint coverage matrix to evaluate how sensitive data propagates during the transient execution, effectively guiding exploration. Furthermore, DeJaVuzz introduces taint liveness annotations to bind state registers to related taint registers. By using annotated state registers as liveness signals, DeJaVuzz filters out unexploitable taints to reduce false positives.

Overall, this paper makes the following contributions:

- We summarize the challenges of transient execution bug fuzzing in terms of microarchitectural controllability and microarchitectural observability and propose two novel operating primitives: a *dynamic swappable memory* model to resolve address space conflicts for better microarchitectural control, and a *differential information flow tracking* technique to mitigate control flow over-tainting for improved microarchitectural observation.
- Utilizing these two operating primitives, we develop a new processor fuzzing framework named DeJaVuzz, which effectively and efficiently detects transient execution bugs. DeJaVuzz designs training derivation and training reduction strategies atop dynamically swappable memory to efficiently trigger diverse transient windows, and utilizes taints generated by differential information flow tracking to guide fuzzing and identify leakage.
- We evaluate DeJaVuzz on two well-known RISC-V out-of-order processors [54, 57]. Compared to the SOTA fuzzer SPECDOCTOR [18], DeJaVuzz achieves a 4.7× improvement in coverage with more comprehensive transient windows and lower training overhead. DeJaVuzz mitigates control flow over-tainting with acceptable overhead and identifies

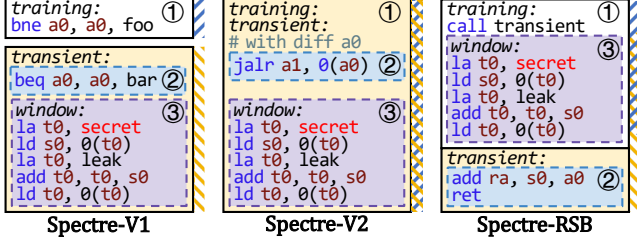


Figure 1. Training and transient execution sections of Spectre-V1, Spectre-V2 and Spectre-RSB. The secret decoding step ④ is omitted.

5 previously unknown transient execution vulnerabilities, all of which are assigned CVE numbers.

To facilitate the community and future research, we publish the source code and experiments of DejaVuzz at <https://github.com/sycuricon/DejaVuzz>.

2 Background

2.1 Transient Execution Vulnerabilities

As shown in Figure 1, the process of exploiting a transient execution bug can be divided into the following 4 attack steps: ① training the target microarchitecture, ② triggering a transient window through the trained state, ③ accessing sensitive data and encoding it into a side channel, and ④ subsequently decoding the secret from the side channel.

However, different types of transient windows exhibit highly varied training patterns. For Spectre-V1, the training section (blue stripe) and the transient execution section (yellow stripe) are independent, which means these two sections can be generated independently as long as the branch instructions have the same address offset. However, this is not always true for other transient execution bugs such as Spectre-V2 and Spectre-RSB [22, 26]. The Spectre-V2 attack requires different arguments (*a0*) to switch between training and exploiting the Branch Target Buffer (BTB) with the same code. And the Spectre-RSB attack requires tempting the processor to speculatively return to a corrupt address by training the Return Stack Buffer (RSB). As seen in the last two types, complex transient windows are mixed with the training section. Triggering such complex transient windows is challenging, as the stimulus generator must carefully handle the semantics of training and transient execution to ensure the window section is executed transiently as expected. Otherwise, non-speculative execution of the window section during training could lead to false positives.

2.2 Hardware Dynamic Information Flow Tracking

Information Flow Tracking (IFT) has been widely deployed at all levels of hardware abstraction to understand how information flows through a system [15, 24, 41, 48]. Hardware

dynamic IFT, known as taint tracking, can dynamically verify information flow properties during the runtime. This is achieved by marking sensitive state elements with taints at the circuit level and propagating the taints based on the operations on sensitive data. There are three instrumentation levels for the hardware dynamic IFT mechanism: gate level [42], RTL level [2], and cell level [37]. Figure 2 shows how hardware dynamic IFT is implemented in hardware. The dynamic IFT instrumentation generates a shadow circuit based on the original circuit, all registers in the original circuit are copied to store taints, and the combinational logic gates are replaced with the corresponding taint propagation policy implementation. The taint propagation policies are a set of rules that are responsible for tainting outputs that are affected by tainted inputs. Policies 1 and 2 are the state-of-the-art taint propagation policies [2, 37] for the AND and MUX cells, respectively. By using shadow circuits, dynamic IFT provides the ability to observe the information flow of the design without affecting the original functionality.

$$O_{AND}^t = (A \& B^t) | (B \& A^t) | (A^t \& B^t) \quad (1)$$

$$O_{MUX}^t = (S ? B^t : A^t) | (\underline{S^t ? (A^t B)} | (A^t | B^t) : 0) \quad (2)$$

Taints generated by the direct computation of input taints and signals, like in Policy 1, are referred to as data taints. In Policy 2, in addition to selecting data taints via the selection signal *S*, the underlined component produces control taints due to the conditional selection semantics of the multiplexer. Unlike data taints, which are only impacted by the actually executed code, control taints also consider changes occurring on unselected branches (i.e., the $A^t B$ term). Thus, once taints propagate to the control flow, it can easily lead to over-tainting [35, 37]. Since taint propagation policies only generate taints without eliminating them, more registers become tainted as the circuit executes, making it increasingly difficult to identify target information flows precisely.

According to our evaluation (§6.3), the state-of-the-art hardware dynamic IFT mechanism CELLIFT [37] suffers from the control flow over-tainting problem. Next, we use the Reorder Buffer (RoB) module of BOOM [57] in Figure 2 as an example to explain how the taint explosion occurs during the RoB rollback. The third RoB entry updates its opcode field register *rob_3_uopc* with the new opcode *enq_uopc* when a valid micro-operation is enqueued (*enq_valid* is high) and the tail pointer points to the third entry (*rob_tail_idx* is equal to 3). Before the RoB rollback, instructions using tainted sensitive data as operands in step ③ write back and taint the RoB state register. When the RoB rolls back, the movement of the tail pointer causes *rob_tail_idx* to be tainted. Since the frontend also uses the RoB index to maintain state, *enq_valid* is tainted. According to Policy 1, both inputs are tainted (the comparison result of the Equal cell is also tainted due to the tainted *rob_tail_idx*), causing the MUX selection signals to be marked as tainted. Furthermore, based on Policy 2, the register *rob_3_uopc* is also marked as

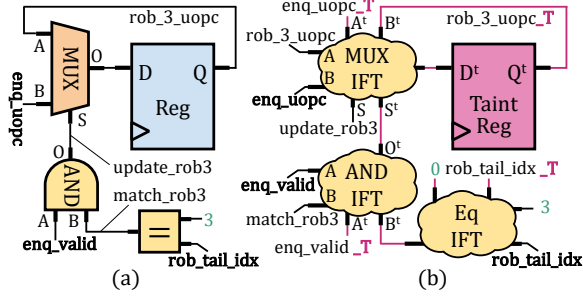


Figure 2. Hardware dynamic information flow tracking instrumentation. a) is the example circuit from the BOOM RoB module, b) is the corresponding IFT shadow circuit.

tainted due to the different input data. All 736 RoB entry field registers have a similar update logic. Therefore, they are all suddenly tainted when the RoB rolls back.

2.3 Processor Fuzzing for Transient Execution Bugs

Processor fuzzing has been employed to detect various bugs, including functional bugs [19, 20, 36, 53], transient execution bugs [11, 12, 18], and side-channel bugs [33]. Although bugs are characterized differently, existing fuzzers generally follow a similar workflow consisting of three main steps.

First, the input generator generates instruction sequences as stimuli either based on constraints [36, 53] or through random generation [19, 20, 33]. As discussed in §2.1, a transient execution attack involves multiple steps. Thus, existing fuzzers strategically divide the generation into multiple phases. For instance, INTROSPECTRE and TEESEC complement the main gadget with the preceding helper gadgets depending on whether the target memory access paths are met in the software execution model. SPECDOCTOR sequentially progresses through the transient-trigger, secret-transmit, and secret-receive phases to generate a complete stimulus. During each phase, additional instructions are randomly appended to those generated in the previous phase until specific goals are met. The goals of each phase are to trigger a RoB rollback, generate differences in microarchitecture, and cause differences in execution cycles.

Second, the fuzzer uses an RTL simulator to convert the Design Under Test (DUT) into a software model and then uses the model to execute the generated instruction sequences. During simulation, the fuzzer leverages instrumentation to measure coverage to guide mutations. Existing fuzzers define several coverage metrics to reflect the general processor behavior, such as mux toggle coverage [23], control register coverage [19, 53], or hardware behavior coverage [20]. However, transient execution vulnerabilities focus more on propagating sensitive data within the microarchitecture. Therefore, existing general processor behavior coverage metrics are unsuitable for transient execution vulnerabilities.

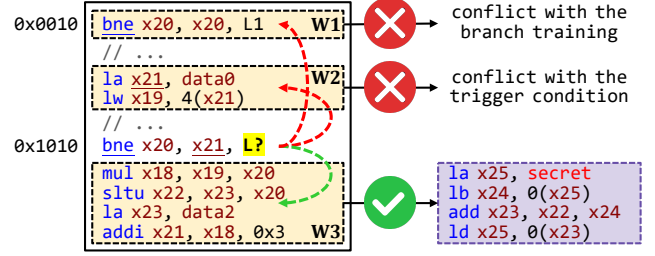


Figure 3. Assuming the branch instruction at 0x1010 can trigger transient windows at different addresses by using different branch targets L?, only transient windows that do not conflict with training instructions can be exploited.

Third, the fuzzer analyzes the microarchitecture to determine if any bug exists. Unlike the functional bugs that can be detected using co-simulation [19, 53], transient execution vulnerabilities require detailed microarchitecture analysis. For example, INTROSPECTRE and TEESEC dump the microarchitecture at each cycle and then assess whether leakage has occurred based on the presence of the secret values in the log. SPECDOCTOR observes execution behavior by hashing the final state of the timing components after transient execution and evaluates leakage by comparing the consistency of the hash values between different variants.

3 Operating Primitives

In this section, we first analyze the challenges of transient execution fuzzing based on the key capabilities required by a fuzzer and identify their root causes. Next, we present the design of two novel operating primitives and explain how they address the root causes. For the challenges, we use designs based on the primitives to address them in §4.

3.1 Challenges and Root Causes

The task of a transient execution bug fuzzer is to generate instruction sequences that trigger transient windows and encode secrets into the microarchitecture, and then determine whether the encoded states can leak the secrets. To achieve this, a competent fuzzer must possess two key capabilities. First, it must effectively train the microarchitecture to trigger diverse transient windows, since we are only interested in transiently executed behaviors. Second, it must accurately track the propagation of sensitive data, as we only focus on microarchitecture changes caused by secrets. Based on this observation, we define these two capabilities as microarchitectural controllability and observability, respectively.

Microarchitectural Controllability [8, 30, 49] refers to the ability of a fuzzer to efficiently manipulate microarchitecture to trigger desired transient execution behaviors. Existing fuzzers generate transient windows using template-based [11, 12] or random-based [18] methods. While they can

successfully trigger transient windows, they fail to address the following two challenges.

C1-1. Limited Transient Window. Template-based methods are limited to specific transient window templates, while random-based methods also fail to generate arbitrary transient windows. As shown by W3 in Figure 3, SPECDOCTOR randomly generates training instructions and replaces the RoB squashed instructions with the secret encoding instructions to exploit. However, when the RoB squashed instructions are mixed with the training instructions (i.e., complex transient windows in §2.1), replacing them may invalidate transient execution. For example, replacing branch training can prevent the predictor from reaching the desired prediction state (W1), while replacing the assignment to the condition comparison register x21 could change the branch outcome (W2). For this reason, SPECDOCTOR discards all transient windows containing backward jumps. As a result, existing fuzzers are limited to exploring only a restricted subset of transient windows.

C1-2. Inefficient Training. Making the fuzzer recognize the microarchitecture changes caused by randomly generated instructions and subsequently exploit them is exceptionally challenging. INTROSPECTRE and TEESEC use a manual software execution model to assist in setting up the required microarchitecture but cannot train states beyond the model. SPECDOCTOR also has difficulty assembling matched training-exploitation instruction pairs because meaningless random training instructions often occupy the required addresses. Unutilized microarchitecture training instructions not only reduce the fuzzing throughput but also diminish the training effectiveness due to potential conflicts.

The root cause of the above challenges is the address space conflict. Since the fuzzer cannot predict training effectiveness or transient window locations, instructions are hardly placed at the desired address. For example, training instructions may occupy addresses needed for transient windows, and different training instructions cannot be tested at the same address. This makes it difficult for existing fuzzers to arrange instructions linearly to trigger the desired transient execution behaviors.

Microarchitectural Observability [14, 29, 56] concerns the ability of a fuzzer to monitor and measure the effects of sensitive data on the microarchitecture. Despite having complete access to processor internal states, existing fuzzers fail to track how sensitive data propagates through the microarchitecture, leading to two challenges.

C2-1. Feedback Gap. Prior work ignores the coverage matrix and thus fails to provide feedback for input mutation, leading to blind and random input mutation. This problem is caused by the lack of ability to track the propagation process of sensitive data. INTROSPECTRE and TEESEC cannot capture secrets after arithmetic operations due to the use of value matching. SPECDOCTOR only computes the hash of the final state, and the compressed execution process prevents

capturing the different propagation paths during execution. The missing coverage matrix leaves a gap between input mutation and execution, making it difficult for the fuzzer to explore all possible transient behaviors efficiently.

C2-2. Imprecise Oracle. Buffers are extensively used in processor microarchitecture to improve performance and typically include state registers to indicate the validity of the current data. For example, the Line Fill Buffer (LFB) in BOOM is managed by the Miss Status Holding Register (MSHR). Once the cache line refill is completed, MSHR switches its state register to invalid to indicate that the data in the LFB is outdated instead of clearing the LFB. Existing work has incorrectly considered this scenario as vulnerable, as INTROSPECTRE and TEESEC would match the sensitive data remaining in the LFB. It would also cause SPECDOCTOR to generate different hashes. Due to the imprecise oracles, existing fuzzers pass these false positives to subsequent steps, resulting in meaningless execution.

The root cause of the above challenges is the lack of a mechanism to track state changes caused by sensitive data. Without the ability to observe the information flow of sensitive data, existing fuzzers are unable to measure coverage based on the distribution of encoded sensitive data or query state registers to identify exploitable leakages.

3.2 Dynamic Swappable Memory

Instead of using scalability-limited templates to solve the address space conflict, the core insight of DejaVuzz is that address space can be time-shared by different semantics. Figure 4 shows how scheduling instruction sequences within the same address space enables triggering complex transient windows that could not be generated in Figure 3. During simulation, we first load training instruction sequence (1) or (2) into memory to train the predictor. After training, we flush the memory and load transient instruction sequence (3) to trigger the backward transient window at 0x1010. For the training instruction sequence, since the full address space is available, we do not need to use similar addresses like 0x0010 to train the predictor. Instead, we can directly place a branch training instruction at 0x1010. Additionally, we can explore different training effects, such as using sequence (1) to train the prediction as untaken or sequence (2) to train it as taken. For the transient instruction sequence, since training instructions are not in this sequence, W1 type conflicts are avoided, and conflicted register assignments can be moved to other available addresses (e.g., 0x0) to resolve W2 type conflicts. As shown in sequence (3), after setting up the registers, DUT can directly jump to 0x1010 to trigger the transient window without any conflicts. Besides generating arbitrary transient windows, we can also identify effective training by trying different training instruction sequences. For example, by trying combinations (1)(3) and (2)(3), we can find that only (2) contributes to triggering the transient window. Thus, switching

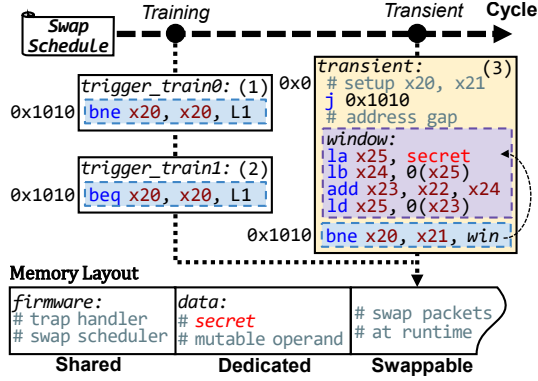


Figure 4. Using swapMem to trigger transient execution.

instruction sequences on demand at different stages effectively resolves address space conflicts, allowing the fuzzer to effectively control the microarchitecture to trigger desired transient execution behaviors.

However, implementing the above switching process with assembly instructions can pollute memory-related training states. To address this, we propose the dynamic swappable memory (swapMem), enabling transparent instruction sequence switching. Since side channel bugs require multiple DUT instances with different secrets to detect behavioral differences, the swapMem is specifically designed for this scenario. As shown at the bottom of Figure 4, the swapMem consists of three regions. The shared region is shared across multiple DUT instances and contains the essential execution environment, including state initialization, trap handling, and runtime instruction sequence scheduling. To facilitate modifying secrets, each DUT has a dedicated region for storing sensitive data and mutable operands. The swappable region is used to hold instruction sequences with different semantics. Each DUT can load the required instruction sequence into the swappable region at runtime according to the swap schedule. Typically, DejaVuzz first executes all training instruction sequences on the DUT, then updates sensitive data permissions, and finally executes the transient instruction sequence. Once a sequence is completed, an exception is triggered, and then the trap handler flushes the instruction cache and loads the next sequence into the swappable region. After swapping the new sequence, the DUT jumps to its entry and continues execution.

The swapMem enhances microarchitectural controllability as the isolation primitive, resolving address space conflicts. In §4.1, we will discuss how to design instruction sequence generation strategies based on swapMem to trigger diverse windows and optimize training overhead.

Table 1. The control taint propagation policies of diffIFT.

Cell Type	Propagation Policy
Multiplexer	$(S?B^t:A^t) (S^t \& S_{diff}?(A^tB^t) (A^t B^t):0)$
Comparison Cell	$O_{diff} \& (A^t B^t)$
Register with En	$(En?D^t:Q^t) ((En^t \& En_{diff}?(D^tQ^t) (D^t Q^t):0)$
Memory Read	$mem^t[addr] \{WIDTH\{addr_{diff}\}\}$
Memory Write	$(Wen?Wdata^t:mem^t[addr]) \{WIDTH\{Wen_{diff} (addr_{diff} \& Wen)\}\}$

3.3 Differential Information Flow Tracking

DejaVuzz intends to employ the information flow tracking technique to identify state changes caused by secrets. However, as discussed in §2.2, the control flow over-tainting problem makes it impossible to identify the propagation of sensitive data. Thus, we propose differential information flow tracking (diffIFT) to mitigate the over-tainting problem.

When fuzzing transient execution vulnerabilities, we can assume that leakage occurs when executing a given instruction sequence using different secrets produces different behaviors. However, Policy 2 considers arbitrary input differences rather than differences caused by secrets. Therefore, a core insight of DejaVuzz is that if no secret can influence the value of a control signal, then even if it is tainted, it should be ignored, as it cannot select an alternative path. However, it is extremely expensive to precisely compute all potential values of each control signal in the out-of-order processor for all input secrets at each cycle [16]. Inspired by the multi-variant execution [7, 31, 34], DejaVuzz approximates the solution with concrete values from multiple variants. To be specific, DejaVuzz creates a differential testing test-bench to determine if sensitive data can produce different values of a control signal by executing the same instructions on two identical DUTs with different secrets. Table 1 lists the updated control taint propagation rules for all supported control flow cells. The overall policies are similar to CELLIFT, except the control taints only propagate when cross-instance comparison signals are high. The highlighted signals with the *diff* subscript represent cross-instance comparison signals. Take the multiplexer as an example, when diffIFT encounters a multiplexer whose selection signal S is tainted, diffIFT checks whether the selection signals are consistent between the variants (i.e., $S_{diff} = S_{DUT_1} \wedge S_{DUT_2}$). If there is a difference, it indicates that sensitive data can generate different selections, and diffIFT, therefore, performs control taint propagation. Otherwise, diffIFT only considers data taint propagation. We instrument the DUT at the RTL IR level and thus support word-level cells and non-flattened memories. Additionally, the data taint propagation policies for data flow cells in diffIFT are consistent with CELLIFT.

It is worth noting that diffIFT is an underapproximation of information flow since it uses concrete values. If a secret pair happens to produce the same value on a secret-dependent

control signal, a false negative will occur. When this happens, data taints still propagate accurately, but control taints are suppressed due to identical control signals. Therefore, DejaVuzz generates secrets for the variant DUT by flipping each bit of the original secret to avoid using identical values. Besides, by leveraging the dedicated region in swapMem, DejaVuzz can directly load different secret pairs to mitigate false negatives without regenerating the input.

The diffIFT serves as the tracing primitive to enhance microarchitectural observability. With the help of taints, DejaVuzz is able to observe sensitive data and its derived values across the microarchitecture. In §4.2 and §4.3, we will explain how to use taint to compute coverage and identify leakages.

4 The DejaVuzz Framework

In this section, we demonstrate how DejaVuzz builds on operating primitives to address the challenges in §3.1, enabling effective and efficient transient execution bug fuzzing.

Overview. As shown in Figure 5, the workflow of DejaVuzz consists of three phases. The first two phases focus on triggering and exploring transient execution, while the final phase is responsible for detecting leakage. DejaVuzz leverages swapMem to isolate different instruction sequences within the same address space. In Phase 1, DejaVuzz derives targeted training for diverse transient windows and evaluates each training to eliminate ineffective training. In Phase 2, DejaVuzz completes the transient window and attempts to encode sensitive data into the microarchitecture. During simulation, DejaVuzz uses diffIFT to track sensitive data propagation and collects taint as coverage to guide exploration. In Phase 3, DejaVuzz first checks transient window constant time execution violations. If no timing differences are detected, it further uses taint liveness annotations to check whether secrets encoded into the microarchitecture can be exploited. Finally, DejaVuzz reports test cases that violate transient window constant time execution or contain exploitable taints as potential bugs.

4.1 Phase 1: Transient Window Triggering

Phase 1 focuses on triggering diverse transient windows with minimal overhead. For challenge C1-1, DejaVuzz uses swapMem to isolate transient execution from training to generate arbitrary transient windows, and employs the training derivation strategy (§4.1.1) to generate targeted training. For challenge C1-2, DejaVuzz further isolates each training and applies the training reduction strategy (§4.1.2) to identify and eliminate ineffective training.

4.1.1 Step 1.1: Trigger Generation. While swapMem resolves address space conflicts, allowing DejaVuzz to generate arbitrary transient windows, effective training is still required to trigger them. To train the required microarchitecture components for triggering transient windows, DejaVuzz employs the training derivation strategy. It first randomly

generates a transient window and then derives targeted training based on the expected transient window.

Trigger Instruction Generation. In this step, DejaVuzz only generates the trigger section of the transient packet (❶). The transient packet refers to the instruction sequence that triggers a transient window and transiently accesses and encodes sensitive data (i.e., transient instruction sequence (3) in Figure 4). DejaVuzz first randomly generates trigger instructions based on the trigger type from the seed. The trigger instructions supported by DejaVuzz cover the entire basic instruction set, including sequential execution instructions (e.g., integer or floating-point arithmetic operations, valid memory accesses), control transfer instructions (e.g., branches, indirect jumps, and returns), and instructions that may trigger architectural exceptions (e.g., illegal instructions, memory access violations). In the example shown in Figure 5, suppose DejaVuzz plans to trigger a transient window caused by a return address misprediction. Next, DejaVuzz generates a dummy transient window filled with nop instructions (❷). For sequential execution instructions and exceptions, the transient window is placed immediately after the trigger instruction by default. For control transfer instructions, DejaVuzz randomly selects whether to place the transient window after the trigger instruction. Finally, DejaVuzz uses an ISA simulator to compute the operands required to trigger the transient window and generate the related register initialization instructions. Therefore, DejaVuzz covers transient windows triggered by all instruction types, effectively enhancing transient window diversity.

Trigger Training Derivation. DejaVuzz uses the transient execution information in transient packets to randomly generate multiple trigger training packets (❸). The trigger training packet refers to the instruction sequence used for training microarchitecture to trigger the transient window (i.e., training instruction sequences (1) and (2) in Figure 4). For each trigger training packet, DejaVuzz first generates a random training instruction, and then inserts nop instructions to align it with the trigger instruction in the transient packet. In the example, we generate three trigger training packets, with the training instructions all placed at the same address (i.e., 0x4) as the trigger instruction ret. Next, DejaVuzz further adjusts the control flow of the training instruction if the training instruction is a control transfer instruction. To be specific, DejaVuzz adjusts the control flow of the training instruction to match the control flow of the generated transient window, enhancing the training effectiveness for control flow prediction. For example, DejaVuzz adjusts the caller address in packet trigger_train_0 to ensure that the return address matches the start address of the transient window (i.e., 0x8). By deriving training from transient execution information, DejaVuzz not only generates diverse transient windows but also produces targeted training, ensuring the fuzzer can more effectively control the microarchitecture to trigger desired transient execution behaviors.

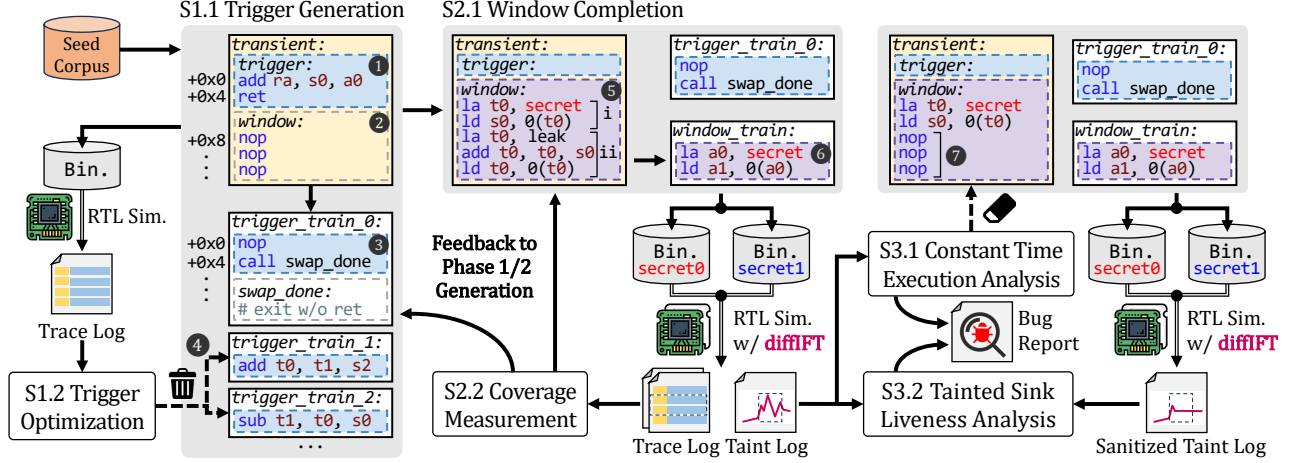


Figure 5. DejaVuzz fuzzing workflow for finding transient execution vulnerabilities, taking Spectre-RSB for example.

4.1.2 Step 1.2: Trigger Optimization. After generating the trigger training packets, DejaVuzz evaluates which packets are helpful in triggering transient windows. Leveraging swapMem, DejaVuzz employs the training reduction strategy that identifies and discards ineffective trigger training packets without affecting transient window triggering, thereby reducing training overhead.

Transient Execution Evaluation. DejaVuzz packages these packets together with a swap schedule, which schedules them in the order of trigger training packets first and then the transient packet. After the RTL simulation, DejaVuzz analyzes the RoB IO events from the trace log. If the number of enqueued instructions within the transient window exceeds the number of its committed instructions, it indicates that the transient window has been successfully triggered.

Training Reduction. Although trigger training packets are derived from the transient packet for targeted training, not all training contributes to triggering the transient window. Fortunately, since each training instruction is isolated in its packet, DejaVuzz can identify ineffective packets by removing one at a time and re-simulating the remaining packets to see if the transient window still triggers (④). If removing a trigger training packet does not affect transient window triggering, it will be permanently discarded from the swap schedule. Otherwise, the packet is necessary, and DejaVuzz will keep it in the swap schedule. DejaVuzz evaluates each trigger training packet in the order of the swap schedule. This process repeats until only necessary trigger training packets remain or none are available. It is obvious that integer arithmetic operations do not contribute to return address prediction. Therefore, in the example, DejaVuzz finds that discarding `trigger_train_1` and `trigger_train_2` does not affect the triggering of the transient window, and finally removes them. By discarding ineffective trigger training packets, DejaVuzz is able to trigger transient windows with minimal training overhead.

4.2 Phase 2: Transient Execution Exploration

DejaVuzz explores which microarchitectures can be used to encode secrets during this phase. DejaVuzz uses taints as the coverage to guide the exploration (§4.2.2), effectively addressing challenge C2-1.

4.2.1 Step 2.1: Window Completion. DejaVuzz replaces the dummy transient window with real payloads and generates a complete test case.

Transient Window Completion. DejaVuzz generates two blocks in the window section (⑤): (i) the secret access block and (ii) the secret encoding block. In the secret access block, besides fixed instructions to access sensitive data, it also randomly masks the high-order bits of the address to attempt to cover MDS-type bugs. In the secret encoding block, DejaVuzz randomly generates instructions that depend on secrets in order to propagate secrets across the microarchitecture.

Window Training Derivation. Similar to trigger training packets, DejaVuzz also derives window training packets for the secret access block (⑥). The window training packet is the training instruction sequence used to train memory-related states used by the transient window. In the example, DejaVuzz attempts to warm up sensitive data into the processor’s internal buffers in advance, such as data cache and load buffer. The generated window training packets are scheduled before the trigger training packets in the swap schedule to avoid invalidating the transient window.

4.2.2 Step 2.2: Coverage Measurement. DejaVuzz performs RTL simulation using the diffIFT instrumented DUTs and measures coverage from the taint log to guide subsequent stimulus generation.

Taint Coverage. DejaVuzz introduces the first secret sensitive coverage matrix designed for transient execution vulnerability fuzzing. The taint coverage treats the total number of taints within a local range as an independent coverage point. To be specific, DejaVuzz inserts a new register array bitmap

into each RTL module. During each clock cycle, DejaVuzz uses the number of tainted registers within the module as the index and writes 1 to the corresponding slot in the bitmap. After the transient execution, DejaVuzz checks the value of each slot in the bitmap. If a slot's value is 1, it indicates that the corresponding number of taints has been explored within the module, and DejaVuzz records the index of such a slot and its module name as a tuple. Finally, DejaVuzz evaluates input exploration based on the total number of collected (module, index) tuples.

The taint coverage has two key properties. The first is locality, as coverage is measured at the module level, reflecting the propagation of sensitive data across different hierarchies. The second is position-insensitive, which helps filter out redundant encoding. For example, when sensitive data is encoded in different slots of the cache data array, the coverage points generated by the cache module are the same.

Coverage Feedback. Once all packets are ready, DejaVuzz packages them into two swappable stimuli with different secrets for diffIFT. After simulation, DejaVuzz first determines the cycle range of the transient window by analyzing RoB IO events from the trace log, then checks taint changes in the transient window from the taint log. If taints increase, it indicates that sensitive data has been successfully propagated, and DejaVuzz further measures the taint coverage from the taint log. If the coverage increase is less than the average increase or sensitive data is not propagated, DejaVuzz mutates the seed to regenerate the window section. If the results after multiple attempts still show low coverage growth, DejaVuzz will discard the seed and return to Phase 1.

4.3 Phase 3: Transient Leakage Analysis

In this phase, DejaVuzz analyzes whether the final state can leak sensitive data. For challenge C2-2, DejaVuzz uses taint liveness annotations to filter out unexploitable taints in the final analysis phase (§4.3.2).

4.3.1 Step 3.1: Constant Time Execution Analysis. For test cases that successfully access and propagate sensitive data, DejaVuzz further analyzes whether leakage occurred. It first compares the execution time of the transient window between DUTs. If inconsistent, it indicates that sensitive data may have caused timing side channels, such as port contention, during the transient window. DejaVuzz directly reports these test cases as potential vulnerabilities.

Encode Sanitization. Although test cases with transient window constant time execution cannot directly leak secrets through the timing side channel, the encoded sensitive data may still be leaked via other side channels. Since accessing sensitive data during training also generates taints, we need to distinguish the taints caused by the secret encoding block before further analyzing whether the encoded sensitive data can be exploited. Therefore, DejaVuzz replaces the secret encoding block in the transient packet with nop instructions

(7) and re-runs the simulation. By comparing the sanitized taint log with the original taint log, DejaVuzz can identify the taints generated by the secret encoding block.

4.3.2 Step 3.2: Tainted Sink Liveness Analysis. The taints produced by diffIFT only indicate reachability. As the LFB example in §3.1, not all encoded secrets are exploitable. Therefore, DejaVuzz further analyzes taint liveness to determine whether the tainted sinks can be exploited.

Taint Liveness Annotation. Inspired by selective data protection [1, 32, 52], DejaVuzz uses annotations to bind taint registers to their corresponding state registers. Developers can annotate the registers with the `liveness_mask` custom attribute [6, 40] to declare their state registers. Taking LFB as an example, the `mshr_valid_vec` signal comes from the state register in MSHR, and the `lb` register is the data buffer in LFB. Line 4 shows the annotation. During diffIFT instrumentation, DejaVuzz automatically connects the liveness signal `mshr_valid_vec` to the taint register of `lb`.

```

1 wire mshrs_0_valid, mshrs_1_valid;
2 wire [15:0] mshr_valid_vec =
3     {8{mshrs_1_valid}, 8{mshrs_0_valid}};
4 (* liveness_mask = "mshr_valid_vec" *)
5 reg [63:0] lb [15:0];
6
7 BoomMSHR mshrs_0 (.io_mshr_valid(mshrs_0_valid));
8 BoomMSHR mshrs_1 (.io_mshr_valid(mshrs_1_valid));

```

However, since the implementation of the state registers is coupled with the microarchitecture, developers may be unable to reference them directly. To accommodate various implementation, we design the liveness signal interface as a generic vector, with each bit representing whether the corresponding slot in the taint register array is valid. For example, the lower 8 entries of `lb` are managed by `mshrs_0`, while the upper 8 entries are managed by `mshrs_1`. We can construct the liveness signal as shown in lines 2-3. DejaVuzz currently requires developers to manually convert state registers into liveness signal vectors. Table 2 shows the manual effort required for annotation and patching. By default, DejaVuzz treats all register arrays (including those registers generated by `Vec` in Chisel) as potential sinks, and developers can customize sinks as needed. Finally, DejaVuzz identifies the target sinks from the encoded secrets obtained in the previous step and reports tainted sinks with valid liveness signals as potential vulnerabilities.

5 Implementation

The implementation consists of 1) a testharness generator responsible for instrumenting RTL source code and integrating two DUTs into a testbench containing `swapMem`, and 2) the fuzzing pipeline illustrated in Figure 5.

Testharness Generator. We implement the `swapMem` atop the Starship SoC generator [38], with ~300 LoC Python for

Table 2. Summary of the cores used for evaluation.

Feature	BOOM	XiangShan
Configuration	SmallBOOM	MinimalConfig
ISA	RV64GC	RV64GC
Verilog LoC	171K	893K
Annotation LoC	212	592

swapMem RTL model generation and ~500 LoC DPI-C for swapMem runtime. The diffIFT instrumentation adds new passes in the Yosys synthesizer to insert taint cells for taint propagation, involving ~1KLoC C++. The taint cell library of diffIFT is implemented in Verilog, which also uses ~1KLoC. **Fuzzing Pipeline.** The fuzzing pipeline consists of ~6500 LoC Python and ~180 LoC RISC-V assemble, which includes stimulus generation and fuzzing management. DejaVuzz uses seeds to generate stimuli, which contain configurations for trigger instructions and transient windows, as well as entropy for the random instruction generator. The generator supports the RV64GC instruction set and covers common transient window types. The fuzzing manager employs a multi-threaded design, allowing multiple RTL simulation instances to run in parallel.

6 Evaluation

We evaluate DejaVuzz by answering the following questions:

- **RQ 1.** How effective and efficient is DejaVuzz in triggering diverse transient windows? (§6.2)
- **RQ 2.** How well does DejaVuzz trace sensitive data, improve coverage, and identify leakages? (§6.3)
- **RQ 3.** Can DejaVuzz uncover previously unknown transient execution bugs in real-world processors? (§6.4)

6.1 Experimental Setup

All experiments are conducted on a machine with dual AMD EPYC 9334 processors featuring 64 cores and 512GB of RAM. We use the industry-standard RTL simulator Synopsys VCS for RTL simulation. Limited by the number of licenses, we only used a maximum of 16 threads in the experiments.

We evaluate DejaVuzz on BOOM [57] and XiangShan [54], two well-known out-of-order processors that are actively maintained in the RISC-V community. BOOM is the third generation of the Berkeley out-of-order machine and is widely evaluated in related academic work [9, 11, 12, 18, 37, 39]. XiangShan is currently the most high-performance open-source RISC-V core and thus has a more complex architecture. Their configurations are summarized in Table 2.

Since INTROSPECTRE and TEESEC only focus on Meltdown-type vulnerabilities and their released artifacts do not include a complete fuzzing framework, we only compare DejaVuzz with SPECDOCTOR. Due to the complex manual patching of the DUT required by SPECDOCTOR, we only compare the BOOM supported by both.

6.2 Microarchitectural Controllability Evaluation

We collect 2,500 transient windows separately and summarize their types and training overhead in Table 3. The Training Overhead (TO) refers to the number of training instructions generated to trigger transient windows. Since DejaVuzz uses nop instructions to align training instructions with trigger instructions, we also compute the Effective Training Overhead (ETO) by excluding the padding nop instructions. For misprediction-type transient windows, since predictors have default prediction states, we exclude transient windows that require no training to trigger.

The results show that SPECDOCTOR can only cover 4 types of transient windows on BOOM and requires about 125 instructions for training. Instead, DejaVuzz can trigger all types of transient windows with minimal overhead. Notably, the training reduction strategy successfully identifies the necessary training packets for triggering the transient window. Therefore, DejaVuzz can trigger exception-type transient windows with zero overhead and use a few training instructions (excluding nop instructions) to trigger misprediction-type windows. To show the effectiveness of the training derivation strategy, we introduce the DejaVuzz* variant. DejaVuzz* still uses swapMem, but its training packets consist of random instructions instead of deriving from transient execution information. Due to the training reduction strategy, both DejaVuzz* and DejaVuzz have zero training overhead for exception-type transient windows. However, since random training fails to align trigger instructions and match transient execution flows, DejaVuzz* cannot trigger indirect jump misprediction on XiangShan. For the other misprediction-type transient window, DejaVuzz* incurs higher training overhead due to the lack of targeted training. These results demonstrate that DejaVuzz can effectively and efficiently trigger more diverse transient windows.

6.3 Microarchitectural Observability Evaluation

Micro-benchmark. We first evaluate the overhead of diffIFT instrumentation at compile and runtime, using the state-of-the-art information flow tracking technique CELLIFT as a reference. The compilation duration includes Chisel elaboration, Yosys instrumentation, and VCS synthesis. For runtime overhead, we manually implement a benchmark covering common transient execution vulnerability test cases and record simulation times. Table 4 shows the results, indicating that the overhead of diffIFT is acceptable compared to CELLIFT. Since CELLIFT instruments at the cell level, it requires flattening all memory, resulting in a significantly increased compilation time. In contrast, diffIFT instruments at the RTL IR level, achieving faster instrumentation. Figure 6 further shows the changes in the taint sum over cycles when executing the benchmark on BOOM. The result proves that CELLIFT does suffer from taint explosion. Once all registers are tainted, CELLIFT loses the ability to track secrets, and

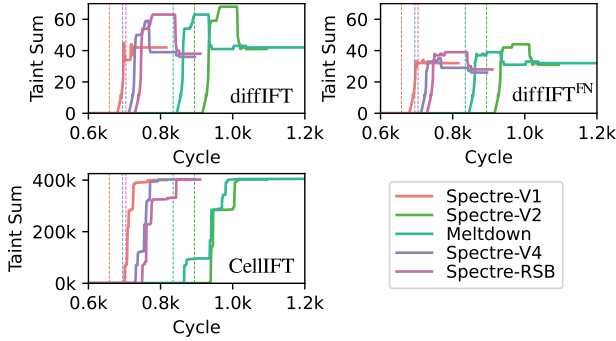
Table 3. Training overhead for different types of transient windows.

Processor	Fuzzer	Load/Store Access Fault TO (ETO)	Load/Store Page Fault TO (ETO)	Load/Store Misalign TO (ETO)	Illegal Instruction TO (ETO)	Memory Disambiguation TO (ETO)	Branch Misprediction TO (ETO)	Indirect Jump Misprediction TO (ETO)	Return Address Misprediction TO (ETO)
BOOM	DejaVuzz	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)	✗	0.0 (0.0)	86.4 (3.8)	85.7 (2.8)	85.6 (2.7)
	DejaVuzz*	1.3	0.1	1.6	✗	0.2	102.2	169.5	89.5
	SPECDOCTOR	✗	126.6	✗	✗	113.5	125.5	122.5	✗
XiangShan	DejaVuzz	0.1 (0.0)	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)	0.0 (0.0)	83.9 (2.8)	90.1 (2.9)	88.7 (2.9)
	DejaVuzz*	0.0	0.0	0.0	0.0	0.4	101.0	✗	97.0

✗ indicates that the corresponding type of transient window failed to trigger.

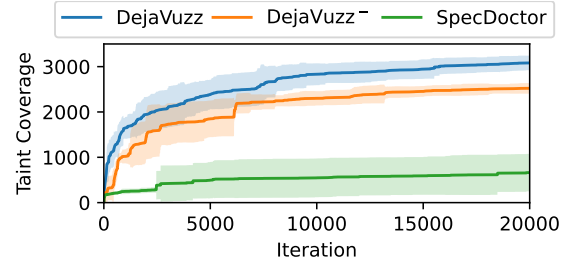
Table 4. Overhead of differential information flow tracking.

Time (s)	BOOM			XiangShan		
	Base	CELLIFT	diffIFT	Base	CELLIFT	diffIFT
Compile	122	2856	268	638		1781
Simulation	Spectre-V1	2.0	152.2	4.8	4.0	17.5
	Spectre-V2	2.1	152.4	5.7	4.5	Timeout
	Meltdown	2.1	152.6	5.6	4.7	after 8h
	Spectre-V4	2.0	152.2	4.9	4.3	17.9
	Spectre-RSB	2.0	152.0	4.8	4.3	17.9

**Figure 6.** Taints during executing each test case. The dotted vertical line represents the start of the transient execution.

the simulation speed is severely degraded. By eliminating control taints caused by identical control signals, diffIFT effectively mitigates control flow over-tainting. Even with two DUTs instantiated in the testbench, the runtime overhead of DejaVuzz is still acceptable.

And to understand the impact of false negatives, we also introduce the diffIFT^{FN} variant in Figure 6. In the diffIFT^{FN} variant, the two DUT instances in the testbench use the same secret to ensure all control signals are identical, representing the worst-case scenario of false negatives. After the transient window is triggered, the taint gradually increases as the secret is loaded into registers. However, since all control signals are the same, diffIFT^{FN} fails to propagate control taints during the process of encoding sensitive data, causing the taints to stop increasing. Finally, the remaining taints are data taints carried by residual secrets in multiple caches and buffers. Therefore, when false negatives occur, data taints still propagate accurately, but control taints are suppressed due to identical control signals.

**Figure 7.** Taint coverage for 5 trials over 20,000 iterations.

Coverage Evaluation. Next, we evaluate the efficiency of microarchitecture exploration. Figure 7 illustrates the growth trend of taint coverage on BOOM. Each experiment is repeated 5 times, and the shaded area represents the 95% confidence interval. To avoid the impact of simulation performance differences between different RTL simulators, we replay the phase 3 test cases generated by SPECDOCTOR in our environment to obtain comparable results and use the number of iterations as the x-axis. The y-axis represents the number of taint coverage points defined in §4.2.2. Due to the lack of feedback on the sensitive data propagation process, SPECDOCTOR only performs random mutations on test cases that can produce different state hashes, limiting its ability to effectively guide fuzzing. With the help of taints, DejaVuzz can guide mutation more effectively, ultimately exploring 4.7× more coverage than SPECDOCTOR. Moreover, DejaVuzz achieves the same saturation coverage as SPECDOCTOR in just 118 iterations. DejaVuzz⁻ is used to demonstrate the effectiveness of using diffIFT as coverage. Instead of using taint coverage, it randomly updates the secret encoding block or regenerates a new transient window for each round. The result shows that DejaVuzz achieves a 22% coverage improvement over DejaVuzz⁻ and achieves the same coverage in 7,200 iterations that DejaVuzz⁻ requires 20,000 iterations to reach. The coverage difference between them demonstrates that using taints as coverage enables more efficient microarchitecture exploration.

Liveness Evaluation. We also found an interesting phenomenon that SPECDOCTOR did not report any vulnerabilities during the coverage evaluation. According to SPECDOCTOR’s design, its phase 3 identified a total of 75 test cases that could

encode sensitive data into the timing components and generate different state hashes. And in its phase 4, SPECDOCTOR attempts to generate random instructions to decode secrets from those timing components. Unfortunately, SPECDOCTOR spent nearly a week executing 100,000 iterations without finding any vulnerabilities. We use taint liveness annotations to analyze all 75 test cases, and find that only 17 of them are real leakages, while the rest are false positives. Most false positives are caused by secrets that fail to be encoded into the microarchitecture but still remain in the data cache. An exception is an invalid test case that executes the transient window during the training. Limited by poor microarchitectural observability, SPECDOCTOR spends a significant amount of time futilely generating random instructions to decode unexploitable false positives. To further validate the effectiveness of taint liveness annotations, we re-execute the test cases using a DejaVuzz variant without taint liveness annotations. Only 21 test cases are correctly identified, while the remaining 54 cases are misclassified due to residual invalid taints in physical registers or RoB. This highlights the effectiveness of taint liveness annotations. With the help of the liveness signals, DejaVuzz can identify exploitable leakages without resorting to inefficient and nondeterministic random decode instruction generation.

6.4 Bugs Found in Real-World Processors

Note that the coverage is only used to evaluate exploration, higher coverage does not guarantee more bugs. Therefore, we also compared the bugs found during the liveness evaluation. Table 5 categorizes all transient execution vulnerabilities discovered by DejaVuzz based on the attack type, transient window type, and exploited timing component. In comparison, SPECDOCTOR can only encode sensitive data into the dcache or trigger lsu port contention. Regarding first bug detection time, SPECDOCTOR takes several days, whereas DejaVuzz detects the first bug in an average of about 10 minutes with 16 threads. Similar to existing work [9, 12, 39], DejaVuzz can cover all trigger variations of known transient execution vulnerabilities, such as replacing the transient window triggered by a page fault in the Meltdown vulnerability with one triggered by unaligned memory access. Additionally, DejaVuzz discovers 5 previously undiscovered transient execution vulnerabilities.

B1. MeltDown-Sampling (CVE-2024-44594) is a hybrid vulnerability of Meltdown and MDS on XiangShan, allowing attackers to sample controllable targets using illegal addresses within a transient window. DejaVuzz generates illegal addresses (e.g., 0x8000...80004000) through the secret access blocks with masks. Due to inconsistent wire widths, when the illegal address is sent to the load unit from the pipeline, the high-bit mask is implicitly truncated. Thus, attackers can sample the secret located at 0x80004000.

B2. Phantom-RSB (CVE-2024-44591) is a vulnerability on BOOM that allows transiently executed instructions to

Table 5. Summary of discovered transient execution bugs.

Processor	Attack Type	Transient Window ¹	Encoded Timing ² Component
BOOM	Meltdown	mem-excp	i/dcache, (l2)tlb, lsu
		mispred, mem-disamb	i/dcache, (l2)tlb
	Spectre	mem-excp	i/dcache, (fau)btb, ras, loop, lsu, fpu
		mispred, mem-disamb	i/dcache, ras, loop, lsu, fpu
XiangShan	Meltdown	mem-exp, mispred, illegal, mem-disamb	i/dcache
	Spectre	mem-excp, mispred, illegal, mem-disamb	i/dcache, lsu, fpu

¹ **mem-excp**: load/store misalign, load/store access/page fault exceptions; **mispredict**: control-flow misprediction; **illegal**: illegal instruction exception; **mem-disamb**: memory disambiguate.

² **lsu**: load unit contention; **fpu**: floating-point unit contention; **fau**: first level branch target buffer; **ras**: return address stack; **loop**: loop branch predictor.

update RSB. As shown in the code below, an attacker can corrupt the RSB based on sensitive data. Although BOOM implements a mitigation that restores the Top-Of-Stack (TOS) pointer and the return address in the top entry after mispredictions (line 11), DejaVuzz discovers that BOOM does not restore entries below the TOS pointer (line 10). After the RSB is corrupted, the attacker can leak the secret by measuring the execution time of the ret instruction.

```

1 beq a0, a0, foo # Predicting the branch untaken, now TOS→X
2 la t0, secret # Loading secret
3 ld s0, 0(t0)
4 andi s0, s0, 0x1 # If secret=1, ra=addr of line6, a valid
5 sub s0, x0, s0 # addr; else ra=0, an illegal addr
6 auipc ra, 0 # Following code requires ra has a valid
7 and ra, ra, s0 # addr, illegal addr will be blocked
8 jalr x0, 12(ra) # Return to next, TOS→X-1
9 jalr x0, 16(ra) # Return to next, TOS→X-2
10 jalr ra, 20(ra) # Call to next, overwrite X-1
11 jalr ra, 24(ra) # Call to next, overwrite X

```

B3. Phantom-BTB (CVE-2024-44590) is a vulnerability similar to *Boombard* [18], where BOOM updates the BTB for exceptions under certain conditions. The following code illustrates the details. Due to a race condition bug in BOOM, when an indirect jump misprediction coincides with an exception commit, BOOM misinterprets the exception as an indirect jump and uses the prediction correction for the mispredicted indirect jump (line 12) to update the BTB entry (line 1) of the instruction that triggered the exception.

```

1 lw t0, 1(x0) # Triggering a misalign exception
2 la t0, secret # Loading secret
3 ld s0, 0(t0)
4 andi s0, s0, 0x1 # If secret=1, ra=addr of line6, a valid
5 sub s0, x0, s0 # addr; else ra=0, an illegal addr
6 auipc ra, 0 # Following code requires ra has a valid
7 and ra, ra, s0 # addr, illegal addr will be blocked
8 jalr x0, 12(ra)

```

```

9 nop           # Padding nop to make the final
10 # ...        # misprediction commit with the
11 nop          # exception in the same cycle
12 jalr x0, 12(ra) # Misprediction

```

B4. Spectre-Refetch (CVE-2024-44592, CVE-2024-44593) is a variant of Spectre-Rewind [10] discovered on both BOOM and XiangShan. DejaVuzz found that the instruction address can also be a resource to cause port contention. Specifically, placing the secret dependent branch at an address that triggers instruction cache miss causes the processor to preempt the fetch component during transient execution. This allows attackers to infer the secret by measuring the execution time of the first instruction after the transient window.

B5. Spectre-Reload (CVE-2024-44595) is another variant of Spectre-Rewind on XiangShan. DejaVuzz found that the load pipeline and load queue contend on the load write-back port of the memory access component. By replacing the floating-point division instructions in the secret-dependent branch of Spectre-Rewind with cache-hitting load instructions, attackers can detect increased latency in cache-missing loads before the transient window.

All of the above vulnerabilities can be exploited to leak sensitive data. B1 can directly leak secrets across privilege boundaries, while B2–B5 require access permission for sensitive data to trigger. We disclosed identified bugs by sending bug reports to respective communities in accordance with the security policies listed for the associated project. According to the maintainers, all vulnerabilities in XiangShan have been fixed, while bugs in BOOM will be retained for future research. Therefore, we recommend against using the BOOM processor in security-critical environments.

7 Discussion and Limitation

Precision Trade-off. Implementing precise IFT is inevitably expensive since it is an NP-complete problem [17]. Although diffIFT can mitigate false positives caused by control flow over-tainting, it also introduces false negatives due to the inability to exhaustively compare all secrets. In practice, DejaVuzz, as a dynamic verification solution, can mitigate false negatives by repeatedly attempting different secret pairs.

Training Preference. Some predictors may require longer training patterns. For instance, in the case of branch mispredictions triggered by branch instructions, training a loop predictor to trigger requires a much longer training instruction sequence compared to training a local branch history table to trigger. Therefore, due to the training reduction strategy, DejaVuzz prefers to choose the least costly training instruction sequence.

Stimulus Migration. The stimuli generated by DejaVuzz only work on swapMem. Fortunately, developers usually only need simulation waveform files to pinpoint bugs. If the stimuli must be migrated to a standard memory model (e.g., for writing general-purpose exploitations), careful manual stitching of the packets is required.

Manual Annotation. Since the state registers are coupled to the implementation, they and their bound taints may reside in different pipeline stages or even across modules. Limited by the loss of semantic information during the design synthesis to RTL, DejaVuzz currently relies on manual taint liveness annotations. We leave the automatic taint liveness annotation (such as using type-safe hardware description languages or large language models) for future work.

8 Related Work

Processor Fuzzing. Encouraged by the promising results of processor fuzzing on functional bugs [19, 20, 36, 53], several approaches have applied processor fuzzing to transient execution vulnerabilities. INTROSPECTRE [12] and TEESEC [11] use manually crafted gadgets to generate Meltdown-type vulnerabilities and detect leakages by analyzing processor runtime logs. SPECDOCTOR [18] generates stimuli for transient execution attacks in multiple phases and determines bugs by observing the final execution time. However, these approaches have the following main limitations. First, they linearly generate transient windows or randomly combine instructions for training, resulting in limited diversity and efficiency in triggering transient windows. Second, they can only analyze shallow information from the microarchitecture, making it impossible to provide feedback on the propagation of sensitive data or identify exploitable leakages. To solve these limitations, DejaVuzz uses swapMem to generate and optimize training instructions to trigger diverse transient windows efficiently, and it employs differential information flow tracking to trace sensitive data to provide coverage feedback and detect exploitable leakages.

Black-box Microarchitecture Fuzzing. Commercial processors lack interfaces for obtaining fine-grained internal state information, leading to limited fuzzing exploration space. Most of the existing black-box fuzzers, such as SpeechMiner [51] and Transynther [27], rely on domain knowledge and can only detect vulnerability variants within a limited template scope. Revizor [28, 29] introduces the model-based relational testing approach that generates random instructions to trigger contract violations. However, due to the limited microarchitectural controllability, they cannot even cover some known vulnerabilities that require simple training. Integrating swapMem (e.g., through DMA) can provide better control over the microarchitecture, facilitating deeper testing of black-box processors.

Formal Verification. By rigorously defining speculative contracts [13], ideally, formal verification can catch all transient execution bugs or prove security. However, in practice, today’s formal verification tools usually suffer from limited scalability and cannot be directly applied to complex out-of-order processors. To bypass this limitation, optimized verification schemes [9, 39, 43, 47] verify abstract models of out-of-order processors. However, the efficacy of such formal

checks depends on the precision of the models (e.g., both B2-B4 escape previous formal analyses on BOOM). DejaVuzz can be used as a complement to formal verification to verify implementation details that are ignored by the models.

9 Conclusion

In this paper, we presented DejaVuzz, a novel pre-silicon processor fuzzer designed to detect transient execution vulnerabilities effectively and efficiently. DejaVuzz introduces two innovative operating primitives to enhance microarchitectural controllability and observability. By leveraging dynamic swappable memory and differential information flow tracking, DejaVuzz efficiently triggers diverse transient windows, effectively guides mutation, and identifies exploitable leakages. We evaluated DejaVuzz on two well-known RISC-V out-of-order processors and achieved up to 4.7× improvement in coverage compared to the state-of-the-art fuzzer SPECDOCTOR. Moreover, DejaVuzz identified 5 new transient execution vulnerabilities (with 6 CVEs assigned), showing its effectiveness in detecting previously unknown bugs.

References

- [1] Salman Ahmed, Hans Liljestrand, Hani Jamjoom, Matthew Hicks, N Asokan, and Danfeng Daphne Yao. Not all data are created equal: Data and pointer prioritization for scalable protection against {Data-Oriented} attacks. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1433–1450, 2023.
- [2] Armaiti Ardeshiricham, Wei Hu, Joshua Marxen, and Ryan Kastner. Register transfer level information flow tracking for provably secure hardware design. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 1691–1696. IEEE, 2017.
- [3] Enrico Barberis, Pietro Frigo, Marius Muench, Herbert Bos, and Cristiano Giuffrida. Branch history injection: On the effectiveness of hardware mitigations against Cross-Privilege spectre-v2 attacks. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 971–988, Boston, MA, August 2022. USENIX Association.
- [4] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking data on meltdown-resistant cpus. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 769–784, 2019.
- [5] Chen Chen, Rahul Kande, Nathan Nguyen, Flemming Andersen, Aakash Tyagi, Ahmad-Reza Sadeghi, and Jeyavijayan Rajendran. {HyPFuzz}: {Formal-Assisted} processor fuzzing. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1361–1378, 2023.
- [6] Design Automation Standards Committee et al. Ieee standard vhdl language reference manual. *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)*, pages 1–640, 2009.
- [7] Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser. N-variant systems: A secretless framework for security through diversity. In *USENIX Security Symposium*, volume 114, page 114, 2006.
- [8] Catherine Easdon, Michael Schwarz, Martin Schwarzl, and Daniel Gruss. Rapid prototyping for microarchitectural attacks. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3861–3877, 2022.
- [9] Mohammad Rahmani Fadiheh, Alex Wezel, Johannes Müller, Jörg Bormann, Sayak Ray, Jason M Fung, Subhasish Mitra, Dominik Stoffel, and Wolfgang Kunz. An exhaustive approach to detecting transient execution side channels in rtl designs of processors. *IEEE Transactions on Computers*, 72(1):222–235, 2022.
- [10] Jacob Fustos, Michael Bechtel, and Heechul Yun. Spectrerewind: Leaking secrets to past instructions. In *Proceedings of the 4th ACM Workshop on Attacks and Solutions in Hardware Security*, pages 117–126, 2020.
- [11] Moein Ghaniyoun, Kristin Barber, Yuan Xiao, Yinqian Zhang, and Radu Teodorescu. Teesec: Pre-silicon vulnerability discovery for trusted execution environments. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pages 1–15, 2023.
- [12] Moein Ghaniyoun, Kristin Barber, Yinqian Zhang, and Radu Teodorescu. Introspectre: A pre-silicon framework for discovery and analysis of transient execution vulnerabilities. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 874–887. IEEE, 2021.
- [13] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. Hardware-software contracts for secure speculation. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1868–1883. IEEE, 2021.
- [14] Jana Hofmann, Emanuele Vannacci, Cédric Fournet, Boris Köpf, and Oleksii Oleksenko. Speculation at fault: Modeling and testing microarchitectural leakage of {CPU} exceptions. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 7143–7160, 2023.
- [15] Wei Hu, Armaiti Ardeshiricham, and Ryan Kastner. Hardware information flow tracking. *ACM Computing Surveys (CSUR)*, 54(4):1–39, 2021.
- [16] Wei Hu, Jason Oberg, Ali Irturk, Mohit Tiwari, Timothy Sherwood, Dejun Mu, and Ryan Kastner. Theoretical fundamentals of gate level information flow tracking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(8):1128–1140, 2011.
- [17] Wei Hu, Jason Oberg, Ali Irturk, Mohit Tiwari, Timothy Sherwood, Dejun Mu, and Ryan Kastner. On the complexity of generating gate level information flow tracking logic. *IEEE Transactions on Information Forensics and Security*, 7(3):1067–1080, 2012.
- [18] Jaewon Hur, Suhwan Song, Sunwoo Kim, and Byoungyoung Lee. Specdoctor: Differential fuzz testing to find transient execution vulnerabilities. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 1473–1487, 2022.
- [19] Jaewon Hur, Suhwan Song, Dongup Kwon, Eunjin Baek, Jangwoo Kim, and Byoungyoung Lee. Difuzzrtl: Differential fuzz testing to find cpu bugs. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1286–1303. IEEE, 2021.
- [20] Rahul Kande, Addison Crump, Garrett Persyn, Patrick Jauernig, Ahmad-Reza Sadeghi, Aakash Tyagi, and Jeyavijayan Rajendran. {TheHuzz}: Instruction fuzzing of processors using {Golden-Reference} models for finding {Software-Exploitable} vulnerabilities. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3219–3236, 2022.
- [21] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *Communications of the ACM*, 63(7):93–101, 2020.
- [22] Esmaeil Mohammadian Koruyeh, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *12th USENIX Workshop on Offensive Technologies (WOOT 18)*, 2018.
- [23] Kevin Laefer, Jack Koenig, Donggyu Kim, Jonathan Bachrach, and Koushik Sen. Rfuzz: Coverage-directed fuzz testing of rtl on fpgas. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2018.
- [24] Xun Li, Mohit Tiwari, Jason K Oberg, Vineeth Kashyap, Frederic T Chong, Timothy Sherwood, and Ben Hardekopf. Caisson: a hardware description language for secure information flow. *ACM Sigplan Notices*, 46(6):109–120, 2011.

- [25] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, Mike Hamburg, and Raoul Strackx. Meltdown: Reading kernel memory from user space. *Communications of the ACM*, 63(6):46–56, 2020.
- [26] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative execution using return stack buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2109–2122, 2018.
- [27] Daniel Moghimi, Moritz Lipp, Berk Sunar, and Michael Schwarz. Medusa: Microarchitectural data leakage via automated attack synthesis. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1427–1444, 2020.
- [28] Oleksii Oleksenko, Christof Fetzer, Boris Köpf, and Mark Silberstein. Revizor: Testing black-box cpus against speculation contracts. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 226–239, 2022.
- [29] Oleksii Oleksenko, Marco Guarnieri, Boris Köpf, and Mark Silberstein. Hide and seek with spectres: Efficient discovery of speculative information leaks with random testing. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1737–1752. IEEE, 2023.
- [30] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. {SpecFuzz}: Bringing spectre-type vulnerabilities to the surface. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1481–1498, 2020.
- [31] Sebastian Österlund, Koen Koning, Pierre Olivier, Antonio Barbalace, Herbert Bos, and Cristiano Giuffrida. kmvx: Detecting kernel information leaks with multi-variant execution. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 559–572, 2019.
- [32] Tapti Palit, Jarin Firose Moon, Fabian Monrose, and Michalis Polychronakis. Dynpta: Combining static and dynamic analysis for practical selective data protection. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1919–1937. IEEE, 2021.
- [33] Chathura Rajapaksha, Leila Delshadtehrani, Manuel Egele, and Ajay Joshi. Sigfuzz: A framework for discovering microarchitectural timing side channels. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2023.
- [34] Babak Salamat, Todd Jackson, Andreas Gal, and Michael Franz. Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 33–46, 2009.
- [35] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *2010 IEEE symposium on Security and privacy*, pages 317–331. IEEE, 2010.
- [36] Flavien Solt, Katharina Ceesay-Seitz, and Kaveh Razavi. Cascade: Cpu fuzzing via intricate program generation. In *Proc. 33rd USENIX Secur. Symp.*, pages 1–18, 2024.
- [37] Flavien Solt, Ben Gras, and Kaveh Razavi. {CellIFT}: Leveraging cells for scalable and precise dynamic information flow tracking in {RTL}. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2549–2566, 2022.
- [38] Sycuricon. Starship SoC Generator. <https://github.com/sycuricon/starship>.
- [39] Qinhan Tan, Yuheng Yang, Thomas Bourgeat, Sharad Malik, and Mengjia Yan. Rtl verification for secure speculation using contract shadow logic. *arXiv preprint arXiv:2407.12232*, 2024.
- [40] Donald Thomas and Philip Moorby. *The Verilog® hardware description language*. Springer Science & Business Media, 2008.
- [41] Mohit Tiwari, Jason K Oberg, Xun Li, Jonathan Valamehr, Timothy Levin, Ben Hardekopf, Ryan Kastner, Frederic T Chong, and Timothy Sherwood. Crafting a usable microkernel, processor, and i/o system with strict and provable information flow security. *ACM SIGARCH Computer Architecture News*, 39(3):189–200, 2011.
- [42] Mohit Tiwari, Hassan MG Wassel, Bitu Mazloom, Shashidhar Mysore, Frederic T Chong, and Timothy Sherwood. Complete information flow tracking from the gates up. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, pages 109–120, 2009.
- [43] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. Checkmate: Automated synthesis of hardware exploits and security litmus tests. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 947–960. IEEE, 2018.
- [44] Daniël Trujillo, Johannes Wikner, and Kaveh Razavi. Inception: Exposing new attack surfaces with training in transient execution. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 7303–7320, 2023.
- [45] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Security Symposium*. USENIX Association, August 2018.
- [46] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue in-flight data load. In *S&P*, May 2019.
- [47] Zilong Wang, Gideon Mohr, Klaus von Gleissenthall, Jan Reineke, and Marco Guarnieri. Specification and verification of side-channel security for open-source processors via leakage contracts. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 2128–2142, 2023.
- [48] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas F Wenisch, and Baris Kasikci. Nda: Preventing speculative execution attacks at their source. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 572–586, 2019.
- [49] Sander Wiebing, Alvis de Faveri Tron, Herbert Bos, and Cristiano Giuffrida. Inspectre gadget: Inspecting the residual attack surface of cross-privilege spectre v2. In *USENIX Security*, 2024.
- [50] Johannes Wikner, Daniël Trujillo, and Kaveh Razavi. Phantom: Exploiting decoder-detectable mispredictions. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 49–61, 2023.
- [51] Yuan Xiao, Yinqian Zhang, and Radu Teodorescu. SPEECHMINER: A framework for investigating and measuring speculative execution vulnerabilities. In *27th Annual Network and Distributed System Security Symposium*, 2020.
- [52] Jinyan Xu, Haoran Lin, Ziqi Yuan, Wenbo Shen, Yajin Zhou, Rui Chang, Lei Wu, and Kui Ren. Regvault: hardware assisted selective data randomization for operating system kernels. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pages 715–720, 2022.
- [53] Jinyan Xu, Yiyuan Liu, Sirui He, Haoran Lin, Yajin Zhou, and Cong Wang. {MorFuzz}: Fuzzing processor via runtime instruction morphing enhanced synchronizable co-simulation. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1307–1324, 2023.
- [54] Yinan Xu, Zihao Yu, Dan Tang, Guokai Chen, Lu Chen, Lingrui Gou, Yue Jin, Qianruo Li, Xin Li, ZuoJun Li, Jiawei Lin, Tong Liu, Zhigang Liu, Jiazhan Tan, Huaqiang Wang, Huizhe Wang, Kaifan Wang, Chuanqi Zhang, Fawang Zhang, Linjuan Zhang, Zifei Zhang, Yangyang Zhao, Yaoyang Zhou, Yike Zhou, Jiangrui Zou, Ye Cai, Dandan Huan, Zusong Li, Jiye Zhao, Zihao Chen, Wei He, Qiyuan Qian, Xingwu Liu, Sa Wang, Kan Shi, Ninghui Sun, and Yungang Bao. Towards Developing High Performance RISC-V Processors Using Agile Methodology. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1178–1199, 2022.
- [55] Yuheng Yang, Thomas Bourgeat, Stella Lau, and Mengjia Yan. Pensieve: Microarchitectural modeling for security evaluation. In *Proceedings of*

- the 50th Annual International Symposium on Computer Architecture*, pages 1–15, 2023.
- [56] Ruiyi Zhang, Taehyun Kim, Daniel Weber, and Michael Schwarz. ({M} WAIT} for it: Bridging the gap between microarchitectural and architectural side channels. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 7267–7284, 2023.
- [57] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. Sonicboom: The 3rd generation berkeley out-of-order machine. May 2020.