ReFuzz: Reusing Tests for Processor Fuzzing with Contextual Bandits

Chen Chen[†], Zaiyan Xu[†], Mohamadreza Rostami[‡], David Liu[†],
Dileep Kalathil[†], Ahmad-Reza Sadeghi[‡], and Jeyavijayan (JV) Rajendran[†]
[†]Texas A&M University, USA, [‡]Technische Universität Darmstadt, Germany
[†]{chenc, zxu43, david_liu, jv.rajendran}@tamu.edu,
[‡]{mohamadreza.rostami, ahmad.sadeghi}@trust.tu-darmstadt.de

***Abstract*—Processor designs rely on iterative modifications and reuse well-established designs. However, this reuse of prior designs also leads to similar vulnerabilities across multiple processors. As processors grow increasingly complex with iterative modifications, efficiently detecting vulnerabilities from modern processors is critical. Inspired by software fuzzing, hardware fuzzing has recently demonstrated its effectiveness in detecting processor vulnerabilities. Yet, to our best knowledge, existing processor fuzzers fuzz each design individually, lacking the capability to understand known vulnerabilities in prior processors to fine-tune fuzzing to identify similar or new variants of vulnerabilities.**

To address this gap, we present ReFuzz, an adaptive fuzzing framework that leverages *contextual bandit* to reuse highly effective tests from prior processors to fuzz a processor-under-test (PUT) within a given ISA. By intelligently mutating tests that trigger vulnerabilities in prior processors, ReFuzz detects similar and new variants of vulnerabilities in PUTs. ReFuzz uncovered three new security vulnerabilities and two new functional bugs. ReFuzz detected one vulnerability by reusing a test that triggers a known vulnerability in a prior processor. One functional bug exists across three processors that share design modules. The second bug has two variants. Additionally, ReFuzz reuses highly effective tests to enhance efficiency in coverage, achieving an average 511.23× coverage speedup and up to 9.33% more total coverage, compared to existing fuzzers.

I. Introduction

Processors, as the core of computing systems, are crucial not only for performance but also for system security. Over the past 60 years, instruction set architectures (ISAs) have abstracted the functionality of processors independently of their designs. This abstraction makes the main goal of processor designs enhance performance, dependability, energy efficiency, and fast real-time responses through the introduction of new microarchitectures, rather than adding new functionalities or altering input and output spaces [1].

Consequently, processor designs rely on iterative modifications and extensive reuse of well-established designs. For example, *Intel* extends successful processor generations such as *Tiger Lake* into subsequent generations like *Alder Lake* and *Raptor Lake* [2], [3]. Moreover, the *design reuse* strategy is typical in hardware. According to a 2023 worldwide semiconductor survey, over two-thirds of non-memory system-on-chips (SoCs) and integrated circuits (ICs) reuse existing designs [4]. Supporting this trend, hardware programming languages like *Chisel* have been developed to facilitate design reuse [5]. For example, BOOMV3 [6] reuses large portions of codebase from another processor, Rocket Core [7].

While abstraction of functionalities and design reuse reduce workload, they allow vulnerabilities to propagate across processors. For example, RISC-V processors, CVA6 [8], PicoRV32 [9], and Kronos [10], incorrectly raise exceptions for valid FENCE and FENCE.I instructions due to faulty decoding logic [11]–[13]. Similarly, CVA6 and BOOMV3 incorrectly raise exceptions when accessing page table entries that violate physical memory attribute checks [13].

As processor designs continue to grow more complex, efficiently verifying their integrity and security becomes increasingly challenging. Patching these vulnerabilities post-silicon is costly, as the flaws exist physically within the hardware [14], often requiring kernel and microcode updates [15], [16], disabling microarchitectures [17], or even recalling products [18]. Such mitigation degrades performance and significantly impacts vendors' finances and reputation. Detecting vulnerabilities during the pre-silicon stage (i.e., before fabricating the processors) is therefore critical.

Industrial verification flow. Industry applies both direct and random testing to verify processors before fabrication [19]. Direct testing uses the internals of processor designs and known vulnerabilities, such as common vulnerabilities and exposures (CVEs) and common weakness enumerations (CWEs), to manually create directed tests [20]. These tests are usually reused across different designs for vulnerability detection and coverage achievement [21]. Random testing generates instruction sequences to verify processor behaviors. However, direct testing requires deep domain knowledge, while random testing struggles to effectively verify large-scale designs [11].

Processor fuzzing at pre-silicon. Inspired by software fuzzing, processor fuzzing has emerged as an effective approach for detecting vulnerabilities in modern processors [11]–[13], [22]–[26]. Processor fuzzers have proven effective for

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.240118
www.ndss-symposium.org

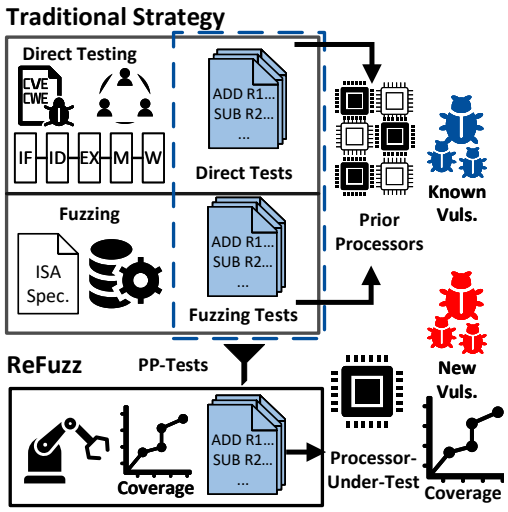

Figure 1. ReFuzz, a novel fuzzing framework that leverages effective tests from prior processors to enhance fuzzing efficiency on processor-under-tests (PUTs). Vuls. means Vulnerabilities.

identifying a wide range of vulnerabilities during the pre-silicon stage, including functional incorrectness [13], time side-channels [27], and speculative vulnerabilities [28]–[31]. Processor fuzzers use a set of seeds to initiate fuzzing, with the effectiveness of a seed directly influencing the efficiency of vulnerability detection and coverage achievement [32]. Existing research treats processor fuzzing as an advanced alternative to random testing, fuzzing each processor individually, generating seeds from scratch, and improving efficiency through advanced seed generation [12], [13], [33]–[37].

To our best knowledge, no hardware fuzzer leverages insights from direct testing by reusing tests[1] from *prior processors* (PP-tests) to guide fuzzing of a processor-under-test (PUT) and enhance coverage and vulnerability detection[2], as shown in Figure 1. Developing such a method is the core contribution of this work and aligns hardware fuzzing with the reuse trend in the hardware development cycle.

Reusing PP-tests. The PP-tests can enhance fuzzing efficiency on PUTs for three reasons: (i) Processors within the same ISA mostly share input and output spaces and functionalities, enabling effective reuse of PP-tests. (ii) Complex functionalities often remain vulnerable across generations. For example, despite the FDIV bug being discovered in *Intel Pentium* processors in 1994 [18], *AMD* still reports bugs of floating point units in its *Zen* family processors in 2023 due to the complexity of floating-point arithmetic [38]. Similarly, processor fuzzers like Cascade [12] report multiple vulnerabilities related to the memory synchronization function in RISC-V processors. This function is challenging to implement while maintaining both memory consistency and pipeline efficiency. (iii) New microarchitectures can introduce variants of known vulnerabilities that exist in prior processors, making PP-tests valuable starting points for uncovering related flaws.

However, we observe that directly executing PP-tests on PUTs fails to detect variants of known vulnerabilities or to explore the new microarchitectures. While hardware fuzzing addresses these limitations, it introduces its own challenges: (i) A fuzzer can over- or under-mutate a PP-test, reducing both the fuzzer's efficiency and effectiveness. The fuzzer must carefully balance between the time to mutate a given test and the need to proceed to the next. (ii) The test effectiveness varies during fuzzing, requiring dynamic evaluation and prioritization.

ReFuzz. To overcome the challenges, we introduce ReFuzz, the first fuzzing framework that leverages contextual bandit (CB) algorithms to adaptively reuse PP-tests as seeds. ReFuzz uses the exploration-exploitation trade-off inherent to CB algorithms to balance between reusing the current test and switching to the next. ReFuzz evaluates coverage increment of a PP-test at different total coverage to precisely prioritize seeds, enhancing both coverage and vulnerability detection.

Overall, the main contributions of this paper are:

- We develop the first framework, ReFuzz, that leverages CB algorithms to guide test reuse from prior processors as seeds for fine-tuning fuzzers on PUTs. Unlike existing approaches that fuzz each PUT independently, ReFuzz exploits ISA abstraction and common design reuse to provide effective tests across processors following the same ISA.
- ReFuzz is agnostic to any processor fuzzers and random testing that require seeds to initiate processes.
- We evaluate ReFuzz on five widely-used and open-sourced RISC-V processors with diverse microarchitectures and achieve an average 511.23× coverage speedup over baseline fuzzers. ReFuzz also outperforms baseline fuzzers and achieves up to 9.33% more total coverage (see Section VI).
- ReFuzz detected three new vulnerabilities and two new functional bugs. ReFuzz detected one vulnerability by reusing a PP-test that triggers a known vulnerability in a prior processor, resulting in a memory deadlock exploitable for denial-of-service attacks. Rocket Core [7], BOOMV3, and BOOMV4 [6] have the same bug due to reusing the same module. BOOMV3 and BOOMV4 share another, and BOOMV4 has more variants due to its new microarchitectures.

II. Background

A. Verification of Processor Design

To manage design complexity, verification of modern processors is conducted at multiple levels. At the unit level, verification targets individual components like decoders or adders. The subsystem level targets integrated groups of modules that perform specific functions like cache coherence. At the architecture level, verification ensures that the processor is compliant with its ISA specifications, which is the primary focus of most hardware fuzzers [39].

To support multi-level verification, both industry and academia employ a comprehensive toolbox of formal and dynamic techniques [40]. **Formal verification**, primarily used

[1]A processor test is a binary executable with a sequence of instructions.

[2]For brevity, we call processors that share the ISA with PUTs as "prior" processors and the tests executed on them as "prior-processor" tests (PP-tests).

at the unit and subsystem levels, relies on predefined assertions to validate functional correctness and security properties. At the architecture level, dynamic verification is more common and includes **random testing**, which generates instruction sequences to explore design behaviors, and **direct testing**, which applies curated test suites or handcrafted tests based on known vulnerabilities (e.g., CVEs, CWEs) [41]. Because many processors share similar verification goals, such as covering corner cases and detecting variants of vulnerabilities, directed tests from prior processors are commonly reused across processor generations to reduce verification effort [21].

B. Hardware Processor Fuzzers

Fuzzing is a dynamic technique that verifies designs through iterative test generation and execution [32]. Processor fuzzers typically produce instruction sequences based on the instruction set architecture (ISA) of the processor-under-test (PUT). A processor fuzzer includes four core components: *seed generator*, *mutation engine*, *feedback engine*, and *vulnerability detector* [22]. The *seed generator* generates an initial set of tests as **seeds**, by randomly selecting instruction opcodes and operands [24]. A fuzzer then simulates or emulates these seeds on the PUT and collects feedback data and output used by the *feedback engine* to guide mutations and the *vulnerability detector* to identify bugs.

The *feedback engine* often employs code coverage as feedback [11], which monitors the amount of hardware logic explored, such as branch statements, finite-state machines (FSMs), and toggled bits. Additionally, hardware fuzzers use customized metrics as feedback, such as control-register coverage [24], which tracks the states of multiplexer signals, and control and status register (CSRs) coverage [23], which monitors the values of CSRs in the PUT. The *vulnerability detector* identifies vulnerabilities using either **assertions**, which check if certain conditions hold true at runtime [42], or **differential testing**, which compares the PUT's outputs against a golden-reference model (GRM) [11]–[13], [24], [36]; mismatches represent potential vulnerabilities in the PUT.

After executing the initial tests, the feedback engine identifies "interesting tests" that reach new coverage points for further exploration of design spaces. These tests guide the *mutation engine* to generate new tests. The mutation engine performs data manipulation, such as bit flips and swaps [11], [43], similar to the strategies used in the most popular software fuzzer, American Fuzzy Lop (AFL) [44]. Mutation operators may mutate instruction operands or alter entire instructions. The *seed generator* and *mutation engine* automate test generation, and the effectiveness of seeds is crucial for determining the efficiency and effectiveness of a fuzzing campaign [32].

C. Bandit Problems

Bandit problems are a class of reinforcement learning problems focused on action selection without modeling future state transitions. The objective is to learn a **policy** that maximizes expected cumulative reward by balancing exploration and exploitation [45].

Contextual bandit (CB) is a type of bandit problem that includes contextual information. Before taking an action, the learner observes the context of the environment. The learner then selects an action and receives a reward only for that action, without feedback on the unchosen alternatives. This feature makes CB algorithms particularly suited for real-world scenarios, where environments are dynamic and involve a large number of actions (e.g., thousands) [45].

In general, CB consists of six core components: (i) agent: the decision maker selecting actions; (ii) environment: the source of context and reward feedback; (iii) set of arms ($\mathcal{A}$): the set of available actions; (iv) context (c): information observed before making a decision; (v) policy (π): the mapping from context to actions; (vi) reward (r): feedback from environment for the chosen action. At each time step t, the agent observes a context c_t from the environment, selects an action $a \in A$ according to policy $\pi(\cdot|c_t)$, and receives a reward $r_t = f(c_t, a)$ from the environment.

III. Observations on Reusing and Mutating Tests

Directly reusing prior-processor tests (PP-tests) often falls short when detecting variants of vulnerabilities or achieving comprehensive coverage on processor-under-tests (PUTs). While PP-tests remain valuable for their original verification purposes, they exhibit inherent limitations in identifying variants of vulnerabilities that manifest uniquely within evolved microarchitectures. In this section, we analyze the limitations of directly using PP-tests to detect variants of vulnerabilities or enhance coverage on PUTs and show how processor fuzzers can overcome these limitations by mutating tests intelligently.

A. Case Study: Detecting Variants of Vulnerabilities

`ReFuzz` detects a new bug affecting both `BOOMV3` and `BOOMV4` [6], where improper updates to the CSRs cause certain instructions observed to increment the committed instruction counter (`minstret`) by two. This behavior violates the RISC-V specification, which mandates that each committed instruction increments `minstret` by one [46]. Accurate accounting of committed instructions is essential for performance profiling [47], bug reproduction [48], and anomaly detection [49].

Shared root cause lies in the interaction between the reorder buffer (`ROB`) and the `CSR` modules. The `ROB` module manages instruction commit logic, while the `CSR` module tracks architectural states, including privilege levels and the number of committed instructions (i.e., `minstret`). In both `BOOMV3` and `BOOMV4`, committing an instruction by the `ROB` module triggers a two-cycle delay to update the `minstret` register in the `CSR` module, as shown by the waveforms in Figure 2. However, the processors are configured to expose architectural states to the software level when an instruction is committed without counting this delay. This makes the CSRs, such as `minstret`, inaccurately represent the processor's architecture states, leading to some instructions failing to increment the `minstret` register, while others increment the register by

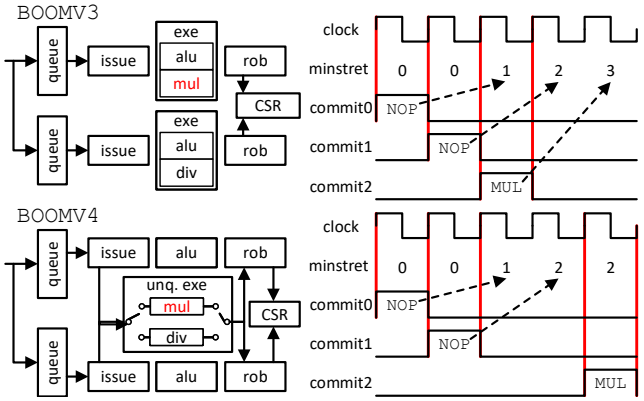

Figure 2. BOOMV3 and BOOMV4 have the same bug due to reusing modules that update the minstret register two cycles after an instruction is committed. BOOMV4 has more variants of the bug due to its new microarchitectures. For example, the MUL instruction will trigger the bug in BOOMV4 but not in BOOMV3. Red lines highlight the clock cycles when minstret is accessed to represent architectural states, while dashed arrows point to the actual commit number for each instruction.

two, BOOMV3 and BOOMV4 reuse both modules, thereby sharing the same bug.

Variants in BOOMV4. However, BOOMV4 includes 10 additional instructions that can trigger this bug due to different microarchitectures. Both BOOMV3 and BOOMV4 are superscalar processors, capable of executing multiple instructions in parallel via multiple issues [1]. In Figure 2, they are configured with two issue slots. In BOOMV3, each issue slot is equipped with an execution unit, including a multiplier for the MUL instruction. In contrast, BOOMV4 uses a unique execution unit across issue slots. To avoid data races, the shared unit implements a static arbiter to serialize access, introducing extra latency when BOOMV4 accesses the multiplier. Consequently, executing the same test (an instruction sequence with two NOP and one MUL) results in MUL incrementing minstret by two on BOOMV4, but by one on BOOMV3. Figure 2 illustrates this discrepancy, and Appendix C lists all instructions observed to trigger this bug. This case highlights that (i) the design reuse strategy can propagate vulnerabilities across processor generations, and (ii) directly reusing tests from prior processors may fail to detect variants of vulnerabilities. For example, BOOMV4 has 10 additional variants of the vulnerability. Properly mutating PP-tests (e.g., by altering instruction opcodes [11]) increases the probability of detecting such variants.

However, effective mutation is non-trivial. Over- or under-mutating a test can reduce a fuzzer's efficiency and effectiveness. Fuzzers must carefully balance how long to mutate a given test versus when to proceed to the next.

Observation #1: Mutating PP-tests is essential for uncovering variants of vulnerabilities in PUTs. However, the effectiveness and efficiency of fuzzing depend on balancing how long to mutate a test before switching to the next one.

Table I
COVERAGE RESULTS OF DIFFERENT TEST REUSE STRATEGIES ON BOOMV4 [6]. "PP" REFERS TO CVA6, Rocket Core, AND BOOMV3.

Strategy	Ave. Total Coverage (%)	Speedup
Fuzzing from Scratch (baseline)	66.66	1.00×
Same Sequence of BOOMV3-Tests	66.47	0.76×
Random Sequence of PP-Tests	66.40	0.74×
Ranked Sequence of BOOMV3-Tests	66.64	1.00×
Ranked Sequence of PP-Tests	66.76	2.30×
ReFuzz	70.58	5.40×

B. Can Reusing PP-tests Enhance Coverage?

Since improving coverage achieved by a fuzzer is critical for detecting vulnerabilities, we evaluate whether simply reusing PP-tests can improve coverage on PUTs.

Evaluation setup. We evaluate coverage achieved by a fuzzer in terms of *total coverage*[3] and *coverage speed*. We use an existing fuzzer [11] to fuzz BOOMV3 (the prior processor) and BOOMV4 (the PUT) [6] by generating 21K tests. The 21K tests from BOOMV3 serve as PP-tests. We define coverage speed as the number of tests required to reach a given total coverage. For example, the baseline fuzzer reaches 66% total coverage after generating 8,548 tests, whereas directly reusing tests from BOOMV3 requires 11,246 tests to reach the same total coverage, resulting in a 0.76× slowdown. Also, executing all PP-tests from BOOMV3 on BOOMV4 resulted in 0.19% less total coverage. Table I summarizes the coverage results of all strategies, with ReFuzz performing the best (see Section V for the details of the evaluation setup).

Inadequate diversity. Directly reusing tests from BOOMV3 does not achieve higher total coverage than baseline on BOOMV4 because the PP-tests cannot explore unique design features of BOOMV4. To address the issue, we collect 21K tests each from two RISC-V processors Rocket Core [7] and CVA6 [8] (totally 63K tests), which are widely used as benchmarks by processor fuzzers [12], [13], [24], [33], [36]. We randomly pick 21K tests from all three processors (i.e., Random tests from procs). The strategy achieves 0.26% less total coverage than the baseline.

In contrast, using the same PP-tests, ReFuzz achieves 3.92% more total coverage compared to the baseline. This highlights that simply reusing PP-tests achieves similar total coverage as fuzzing the PUT directly. However, mutating PP-tests, as ReFuzz does, can help explore unique design features of the PUT, leading to higher total coverage.

Observation #2: Simply reusing PP-tests achieves similar total coverage as fuzzing the PUT directly. Mutating PP-tests achieves more coverage.

Inefficient order of execution. The naive approach for reusing PP-tests is to execute them on the PUT in the identical sequence (as they were executed during the PP fuzzing cam-

[3]Total coverage refers to the cumulative percentage of coverage points reached after executing a given number of tests.

paign). However, this method proves insufficient for enhancing the coverage speed, as it degrades the coverage speed by a factor of $0.76\times$ compared to fuzzing the PUT from scratch, evidenced by our experimental results (see second row in Table I). As expected, results from other PPs with this method demonstrate even worse performance.

A more advanced approach would be to randomly select tests from the PP-tests and execute them on the PUT. However, this method encounters the same inefficiency problem of coverage speed, as demonstrated by our experiments (see third row in Table I). This approach, on average, further degrades the coverage speed by a factor of $0.74\times$. This shows that the order of execution impacts the coverage speed.

To advance the method, we ranked the tests based on their *standalone coverage*[4], while executing on their related PP. For instance, if a PP-test originated from `Rocket Core`, we ranked it based on its standalone coverage from `Rocket Core`. In the fourth row of Table I, we present results for the most closely related PP, `BOOMV3`, to our PUT, `BOOMV4`, which demonstrates that while ranking provides better coverage speed compared to identical sequence and random selection, it still delivers the same performance as the baseline. To further advance this method, we ranked the PP-tests based on their average standalone coverage across all three processors, achieving a $2.30\times$ improvement in coverage speed (see fifth row in Table I). These results suggest that leveraging a broader set of PPs helps identify highly effective tests.

While the aforementioned strategies improve coverage speed, their static nature prevents them from improving total coverage. This observation motivates that a dynamic strategy, while benefiting from effective test reuse and improving coverage speed, could also increase total coverage. We propose the `ReFuzz` framework to address this lack of dynamic strategy. `ReFuzz` improves the coverage speed by $5.40\times$, outperforming all evaluated strategies. The key insight is that a test's effectiveness in increasing coverage varies during the fuzzing process. For instance, a test achieves an average 4.99% *incremental coverage*[5] across all three processors when the total coverage is 55%, but the same test achieves only 0.06% incremental coverage when the total coverage is 70%.

Observation #3: The effectiveness of tests varies during the fuzzing process. The fuzzer needs to dynamically evaluate the effectiveness of tests and determine which test to mutate.

IV. Methodology

In this section, we first discuss why CB is suitable for reusing prior-processor test (PP-tests) with hardware fuzzing and how we model hardware fuzzing as a CB problem. We then describe how we build and train the CB model. We improve the efficiency of the CB model based on the characteristics of fuzzing (we call it adaptive CB), and we further optimize the training tests to enhance the effectiveness of the CB model. Finally, we integrate the trained CB model with two distinct fuzzers to test its effectiveness, demonstrating that the approach is agnostic to any hardware fuzzers.

A. Why Contextual Bandit (CB)?

As shown in III, a naive, greedy approach that prioritizes historically high-performing tests risks converging on local optima. Conversely, an exhaustive evaluation of all prior test cases is computationally prohibitive and lacks the dynamic and adaptive nature. This problem of choosing an action (a test) based on the current state (*coverage context*[6]) to maximize cumulative reward (total coverage) is precisely a **contextual decision-making problem**.

The CB algorithms are suited to this challenge due to three key properties. First, CBs incorporate *context* into their policy. The effectiveness of a test changes as fuzzing progresses; tests for broad exploration at low total coverage (e.g., 50%) are different from those needed for deep, corner-case exploration at high total coverage (e.g., 70%). CBs learn a policy that adapts to the evolving coverage context, enabling **adaptive decision-making** for `ReFuzz` to select the most effective tests at any coverage context. Second, CBs are designed to balance leveraging known effective tests (exploitation) with investigating new ones (exploration). This balance is critical for fuzzing efficiency. `ReFuzz` uses this capability to dynamically adjust its strategy (**exploration-exploitation trade-off**), ensuring it efficiently uses highly effective tests. Third, CBs are computationally lightweight and can handle a large action space list PP-test corpora without the overhead of more complex reinforcement learning models. Their *anytime learning* [45] property means they continuously refine their policy and retain a useful solution even if the training process is interrupted (**scalability and anytime learning**).

B. Modeling Fuzzing as a Contextual Bandit (CB) Problem

To apply CB to fuzzing, we model the interaction between `ReFuzz` and the PUT as a contextual bandit problem. During training, `ReFuzz` learns a CB policy by iteratively exploring PP-tests and observing their effectiveness. The goal is to identify and prioritize tests that maximize total coverage.

Preliminary formulation. The primary components of a bandit problem are: *agent*, *environment*, *set of arms*, *context*, *policy*, and *reward*, as mentioned in Section II-C. `ReFuzz` is the agent that learns the policy (i.e., which test to select first) through interacting with the environment. We let the training environment be the PUT and its verification environment, such as software simulators [50], which provides coverage feedback. We then let n be the total number of training steps.

Definition 1. $\mathcal{A}$ is the finite set of **arms**. It contains all PP-tests. a_t denotes the test selected by `ReFuzz` at time t.

[4]Standalone coverage refers to the percentage of coverage points achieved by executing a single test.

[5]Incremental coverage refers to the percentage of newly reached coverage points compared to total coverage by a test.

[6]Coverage context refers to the point in the fuzzing process when total coverage reaches specific thresholds, such as 55%, 60%, or 65%.

Definition 2. $\mathcal{C} \subseteq [0,1]$ is the finite set of **coverage context** which contains all possible total coverage of a PUT achievable by the fuzzer. We use $c_t \in \mathcal{C}$ to denote the coverage context at time t, where c_t ranges from 0 (covers no point) to 1 (covers 100% coverage points).

Definition 3. $r_t(c_t, a_t)$ denotes the reward at time t after executing the test a_t selected under context c_t. To maximize total coverage, we let the reward represent **coverage increment** after receiving coverage feedback for the selected test from the environment as $r_t(c_t, a_t) = \Delta cov_t(a_t)$, where $\Delta cov_t(a_t) = \{\Delta cov_t \mid \Delta cov_t$ is covered by a_t at t but not $c_t\}$. In training, the coverage increment is the incremental coverage of the selected test a_t.

At each time step, ReFuzz observes the current coverage context. It chooses a test, runs the test, and checks how much coverage increment is achieved (reward). It then updates its policy about which tests are most effective at increasing coverage under which coverage context. Thus, the goal is to learn the optimal CB policy that maximizes the sum of collected rewards, i.e., total coverage. Let $g\colon \mathcal{C} \times \mathcal{A} \to \mathcal{C}$ denote the deterministic function that represents the PUT and its verification environment. In particular, given the coverage context of a PUT c_t and the test a_t selected by ReFuzz, $c_{t+1} = g(c_t, a_t)$ is the next coverage context of the PUT. Formally, the mathematical objective of ReFuzz is

$$\max_{\pi} \mathbb{E}\left[\sum_{t=1}^{n} r_t(c_t, a_t) \mid a_t \sim \pi(\cdot \mid c_t), c_{t+1} = g(c_t, a_t)\right].$$

C. Training a Contextual Bandit (CB) Model

Figure 3 shows how ReFuzz applies CB to identify effective tests and takes two inputs: PP-tests and various coverage contexts collected during fuzzing PPs. Following industry practices in creating directed tests for different verification purposes [21], we categorize PP-tests into either **vulnerability tests** that trigger known vulnerabilities or **coverage tests** that improve coverage achieved on one or multiple PPs.

During training, ReFuzz generates two types of *test lists*: *vulnerability list* and multiple *coverage lists*. A test list contains optimal PP-tests identified by the CB model associated with a probability distribution, in which θ_1 corresponds to the probability of $Test_1$ being selected by the CB model for a given coverage context c_1. Each coverage list is tuned to a specific coverage context. Multiple coverage lists help ReFuzz to prioritize tests precisely based on the current coverage context and prevent ReFuzz from selecting the same test across different coverage contexts. For example, the coverage list trained for 50% coverage context contains more tests that explore major design spaces in a PUT. While the coverage list for 70% coverage context contains more tests that explore corner cases (see Section VI-D). The number of coverage list depends on the granularity of coverage context configured during ReFuzz's setup.

However, using PP-tests to train a CB model is non-trivial due to the limitations in the original CB algorithms. To address this, we first use a *test minimizer* to preprocess PP-Tests and remove redundant ones that reach the same coverage points. The minimized tests are then used by the *adaptive CB algorithm*, which drops ineffective tests and learns which tests are most effective at achieving coverage increment under different coverage contexts.

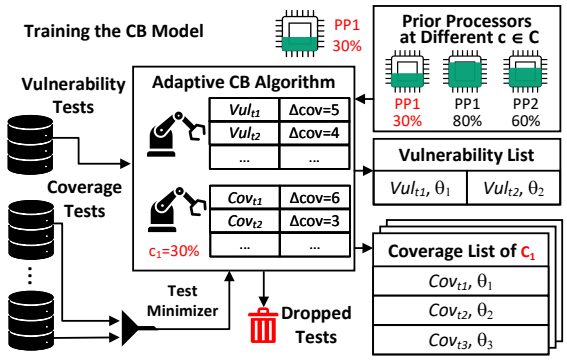

Figure 3. ReFuzz's training stage. C is the set of different coverage contexts. Vul_t is a test in the vulnerability list, and Cov_t is a test in the coverage list.

Figure 4 shows the coverage achievement of the original CB algorithm, the adaptive CB algorithm, and the baseline fuzzer. The original CB algorithm achieves 2.43% more coverage and 4.35× speedup than the baseline. The adaptive CB algorithm achieves 3.92% more coverage and 5.40× speedup[7].

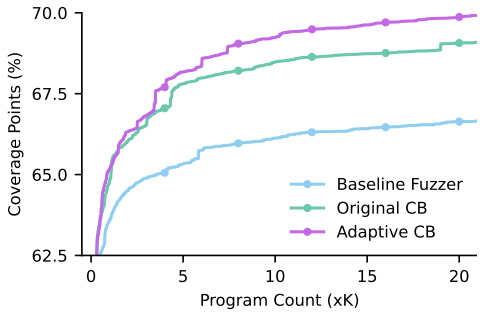

Figure 4. The total coverage achieved by the baseline fuzzer, original CB, and adaptive CB on BOOMV4 [6].

D. Challenges and Solutions

The original CB algorithm achieves limited improvement on the baseline fuzzer due to two major challenges.

Challenge 1. Ineffective tests remain in the lists. The original CB algorithm assumes that all actions remain available forever [45]. However, in fuzzing, some tests consistently achieve zero coverage increment and should not remain in consideration. Moreover, since training a CB model is an iterative process, newly selected tests may outperform existing ones by achieving higher coverage increments. Without a mechanism to eliminate ineffective tests, they remain in the test list, degrading ReFuzz's overall performance.

[7]To make a fair comparison, the results only use PP-tests from the same baseline fuzzer. We collect PP-tests from multiple baseline fuzzers, which achieves substantial improvement (see Section VI-C).

Solution 1. We develop an *adaptive CB algorithm* with an *elimination function*. Unlike conventional elimination strategies [45] that only eliminate suboptimal arms, our approach also aims to explore a broad range of effective tests to enhance fuzzing efficiency. The algorithm trains the model by iteratively identifying and retaining highly effective tests under different coverage contexts. It monitors their effectiveness through moving-average coverage increments and dynamically eliminates ineffective ones. Tests that consistently achieve high coverage increments are promoted to the final model.

Algorithm 1 outlines the adaptive CB algorithm, with the elimination function highlighted in gray. The inputs are the PP-test corpus $\mathcal{A}_{\text{corpus}}$, a coverage context c, the number of arms k, the check window γ, the pre-defined adaptive threshold θ, and the training step n. The outputs are the coverage list $\mathcal{A}$ and the policy π. Since the number of tests (arms) is changing during training, the algorithm uses an auxiliary policy π_{tmp} and a temporary arm set $\mathcal{A}_{\text{tmp}}$ that always contains k arms. For each coverage context c, the algorithm calculates the average coverage increments a test can achieve (Line 8).

Once the test has been selected at least γ times (Line 9), the algorithm evaluates its effectiveness. If the test achieves no coverage increment, it is dropped from $\mathcal{A}_{\text{tmp}}$ with its relevant variables (Line 11), and a new test from the PP-test corpus $\mathcal{A}_{\text{corpus}}$ is added to maintain the number of arms (Lines 12–13). However, if the probability to select the test exceeds the pre-defined adaptive threshold θ (Line 14), it is promoted to the ultimate arm set $\mathcal{A}$ (i.e., the test list) and the final policy π (Lines 15–16). Since the number of the ultimate arms will vary depending on the coverage context and the effectiveness of PP-tests, the final policy needs to be normalized to ensure it remains a valid probability distribution (Line 20).

Challenge 2. Tests may achieve the same coverage points. The CB algorithm evaluates the effectiveness of tests independently, without accounting for overlap in coverage. As a result, it may prioritize multiple tests that reach the same coverage points, even if each individually achieves a high coverage increment. This redundancy reduces exploration diversity and leads to inefficient use of fuzzing resources.

Solution 2. We develop the *test minimizer* to remove redundant tests before training, ensuring computational efficiency while preserving the total coverage achieved by the original test set. Specifically, the test minimizer selects the smallest possible subset of PP-tests that together reach the same total coverage as all PP-tests. For example, if test a covers all the coverage points covered by tests b and c, then tests b and c are removed.

However, identifying such redundancies is computationally challenging. Comparing tests pairwise or exhaustively evaluating all subsets quickly becomes intractable, as the number of combinations grows exponentially with the number of tests. Therefore, inspired by `MINTS` [51], we formulate the task of selecting a minimal subset of tests as an optimization problem and solve it using integer programming. The model of test minimizer is discussed in Appendix A. The model requires a **coverage matrix** as the input, where each row corresponds to a single test and shows which coverage points it reaches. The number of columns represents the total number of coverage points defined by a target coverage metric in a PUT.

Algorithm 1 Adaptive CB algorithm.

1: **Inputs:** $\mathcal{A}_{\textbf{corpus}}, c, k, \gamma, \theta, n$
2: **Outputs:** $\mathcal{A}, \pi$
3: **Initialize:** $\mathcal{A} \leftarrow \emptyset$; $\mathcal{A}_{\text{tmp}} \subset \mathcal{A}_{\text{corpus}}$ with $|\mathcal{A}_{\text{tmp}}| = k$; $\pi_{\text{tmp}}(a|c) = \frac{1}{|\mathcal{A}_{\text{tmp}}|} \quad \forall a \in \mathcal{A}_{\text{tmp}}$; $\pi(a|c) \leftarrow 0 \quad \forall a$; $\hat{r}(a) \leftarrow 0 \quad \forall a \in \mathcal{A}_{\text{tmp}}$; $\#(a) \leftarrow 0 \quad \forall a \in \mathcal{A}_{\text{tmp}}$
4: $\forall a \in \mathcal{A}_{\text{tmp}}$: update $\hat{r}(a)$ and $\#(a)$ once.
5: **for** $t = 1, 2, \ldots, n$ **do**
6: $\quad a_t \sim \pi_{\text{tmp}}(\cdot \mid c)$; $r_t \leftarrow r(c, a_t)$
7: $\quad \pi_{\text{tmp}} \leftarrow \text{UpdatePolicy}(\pi_{\text{tmp}}, c, a_t, r_t)$
8: $\quad \hat{r}(a_t) \leftarrow \frac{\hat{r}(a_t) \times \#(a_t) + r(c, a_t)}{\#(a_t)+1}$; $\#(a_t) \leftarrow \#(a_t) + 1$
9: $\quad$ **if** $\#(a_t) \geq \gamma$ **then**
10: $\quad\quad$ **if** $\hat{r}(a_t) = 0$ **then**
11: $\quad\quad\quad \mathcal{A}_{\text{tmp}} \leftarrow \mathcal{A}_{\text{tmp}} \setminus \{a_t\}$
12: $\quad\quad\quad a' \leftarrow \text{random\_sample}(\mathcal{A}_{\text{corpus}})$
13: $\quad\quad\quad \mathcal{A}_{\text{tmp}} \leftarrow \mathcal{A}_{\text{tmp}} \cup \{a'\}$
14: $\quad\quad$ **else if** $\pi_{\text{tmp}}(a_t \mid c) \geq \theta$ **then**
15: $\quad\quad\quad \mathcal{A} \leftarrow \mathcal{A} \cup \{a_t\}$
16: $\quad\quad\quad \pi(a_t \mid c) \leftarrow \pi_{\text{tmp}}(a_t \mid c)$
17: $\quad\quad\quad \mathcal{A}_{\text{tmp}} \leftarrow \mathcal{A}_{\text{tmp}} \setminus \{a_t\}$
18: $\quad\quad\quad a' \leftarrow \text{random\_sample}(\mathcal{A}_{\text{corpus}})$
19: $\quad\quad\quad \mathcal{A}_{\text{tmp}} \leftarrow \mathcal{A}_{\text{tmp}} \cup \{a'\}$
20: $\pi(a \mid c) \leftarrow \frac{\pi(a|c)}{\sum_{a' \in \mathcal{A}} \pi(a'|c)}$; **return** $\mathcal{A}, \pi$

Constructing the coverage matrix for our test minimizer involves two key requirements. First, we must understand the **hierarchical structure** of the PUT. This is essential for correctly attributing coverage points to their corresponding register-transfer level (RTL) modules, enabling fine-grained analysis of where each test explores within the design. Second, we need to monitor which coverage points within each RTL module are reached by individual tests. This enables us to compile a **row vector** for each test in the coverage matrix, where each entry reflects whether a coverage point was covered.

To address these requirements, we leverage the coverage databases generated by *Synopsys* `VCS` [50], an industry-standard simulator. While `VCS` does not provide APIs for directly accessing the hierarchical structure of the PUT or the status of coverage points, we can extract this information by parsing the internal files of the databases. We then use the information to concatenate row vectors for tests and construct the coverage matrix. The approach ensures stability and remains compatible with a broad range of processor designs. As a result, it facilitates the training of our CB model and `ReFuzz`'s integration with industrial verification flows. The result shows that test minimizer successfully identifies the minimal subset by removing 98.76% redundant tests, reducing the number of tests from $126K$ to $1.5K$ (See Section VI-D).

No elimination is performed on vulnerability tests. Note that the elimination function and the test minimizer are applied

only to coverage tests, which are numerous and highly redundant. In contrast, we assume that vulnerability tests trigger distinct vulnerabilities and explore diverse design spaces. Therefore, no vulnerability tests are dropped during training.

E. Integrating the Trained CB Model with Processor Fuzzers

Environment for testing. To evaluate the generalizability of our CB model, we test it in an environment that differs from the training stage. Given that fuzzers may employ different mutation strategies [11]–[13], the testing *environment* includes both the baseline fuzzer and the PUT. Unlike the training stage, where the model uses only the incremental coverage of each PP-test as the reward, ReFuzz evaluates the cumulative incremental coverage of the PP-test and its mutated variants. Based on the cumulated incremental coverage, the CB model automatically decides whether to continue mutating the current test or proceed to the next one.

Altering the testing environment is standard practice for assessing the generalizability of a CB model. This is analogous to sim-to-real generalization in reinforcement learning [52], where models trained in simulation are tested under altered dynamics, such as changes in gravity or actuator noise. Similarly, testing ReFuzz in an environment with different fuzzing mechanisms provides a more robust assessment of its effectiveness. Empirical results show that ReFuzz is agnostic to fuzzers with distinct mechanisms and outperforms them in both total coverage and coverage speed (see Section VI-C).

Resuming seed generation. Unlike the training stage, ReFuzz does not add new tests to the curated test lists during testing. Thus, when a coverage list becomes empty due to the unique design features of the PUT, ReFuzz allows the fuzzer to resume its native seed generation to continue exploring the design spaces. Additionally, if the total coverage is too low (no match with any coverage context) for the CB algorithm to select tests from the next coverage list, and the current coverage list is empty, ReFuzz will skip directly to the next list, avoiding stalling fuzzing progress. In summary, ReFuzz resumes the fuzzer's own seed generation strategies only after all curated test lists have been exhausted, ensuring efficient reuse of PP-tests while maintaining flexibility to explore unique design features of the PUT.

ReFuzz begins by selecting tests from the vulnerability list and monitors their coverage increment. If a test and its mutated variants fail to improve coverage, it is dropped. Once all tests in the vulnerability list are dropped, ReFuzz switches to coverage lists. ReFuzz tracks the total coverage achieved by vulnerability tests and uses this information to determine the current coverage context and decide the starting coverage list. ReFuzz periodically evaluates each test, removing those that show no further coverage increment, and continues this process until all coverage lists are empty. Appendix H shows the detailed integration process.

F. Putting it all Together

The ReFuzz framework consists of the **training** and the **testing** stages, as shown in Figure 5. In the training stage,

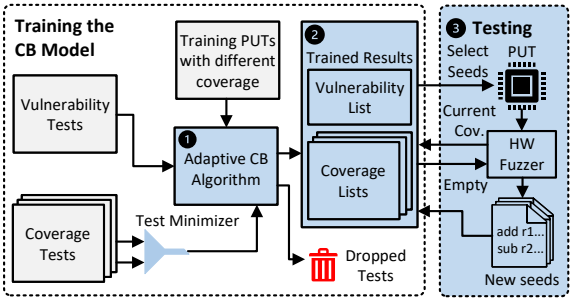

Figure 5. The framework of ReFuzz.

ReFuzz begins by preprocessing a large corpus of tests from prior fuzzing campaigns. To address test redundancy, a test minimizer removes 98.76% of redundant tests, resulting in a lean and effective test corpus for training. Next, ① the adaptive CB algorithm analyzes the performance of minimized tests across prior processors and coverage contexts. This allows ReFuzz to learn which tests are most likely to increase coverage under specific coverage contexts. ② The output is a set of carefully curated test lists: a vulnerability list for known vulnerabilities, and several coverage lists, each tuned to a coverage context. In ③ the testing stage, ReFuzz integrates with an existing processor fuzzer to guide its test selection. The result is a unified framework that combines the speed and targeted nature of directed testing with the explorative power of processor fuzzers.

V. Benchmarks, Training, and Implementation

Fuzzer selection. We select fuzzers to generate tests for training based on two criteria: (i) Do fuzzers implement distinct algorithms that increase the probability of verifying diverse processor functionalities? This diversity helps ReFuzz identify tests that explore a broad range of design spaces. (ii) Are fuzzers' strategies generalizable, allowing them to be applied across different processor designs? This generalizability enables ReFuzz to evaluate the effectiveness of a test on a variety of processors following the same ISA. Based on these criteria, we select TheHuzz and Cascade [12] as our baseline fuzzers. Appendix F includes detailed justifications.

Benchmark selection. While ISAs define the functionalities and interfaces of processors, hardware vulnerabilities often arise from microarchitectures. Therefore, we select processors with diverse microarchitectures as benchmarks.

Table II
Microarchitectural features in benchmark processors.

Processor	OoO	SIMD	Ld/St Forwarding	Speculative Scheduling	Mem. Dep. Predictor
CVA6 [8]	✗	✓	✗	✗	✗
Rocket Core [7]	✗	✗	✗	✗	✗
BOOMV3 [6]	✓	✗	✓	✗	✗
BOOMV4 [6]	✓	✗	✓	✗	✗
RSD [53]	✓	✗	✓	✓	✓

Given that most commercial processors are protected intellectual property (IP) and closed-source, we select bench-

marks from open-source RISC-V processors. Table II summarizes the microarchitectures of each processor. To capture a broad range of microarchitectures, we select three widely-used RISC-V processors by existing processor fuzzers [11]–[13], [33], [36] for training: Rocket Core [7], BOOMV3 [6], and CVA6 [8]. We also include BOOMV4 and RSD [53] to evaluate the effectiveness of ReFuzz. BOOMV4 is the next generation of BOOMV3, enabling assessment of ReFuzz's adaptability to different generations of processors.

Simulation and vulnerability detection. We use *Chipyard* (version 1.13.0, commit 69eba86) [54] as the simulation environment for CVA6 [8], Rocket Core [7], BOOMV3, and BOOMV4 [6], while RSD (commit 7b65f6b) [53] is simulated using VCS [50]. We employ *differential testing*, a commonly used approach in the processor fuzzers, to detect vulnerabilities and bugs, as mentioned in Section II-B. We use Spike (commit de5094a) [55] as the golden-reference model, widely used by existing fuzzers [11], [24], [33], [56]. All experiments are conducted on a 48-core AMD EPYC 7443 processor at 2.6 GHz with 256GB RAM.

Building the PP-test corpus. For vulnerability tests, we collect tests that trigger vulnerabilities detected by two fuzzers on the training processors. TheHuzz detects four vulnerabilities in CVA6, while Cascade detects ten vulnerabilities in CVA6 and two in BOOMV3. For coverage tests, we run both TheHuzz and Cascade to generate 21K tests per training processor, resulting in a total of 126K tests. For a fair comparison in Section III, we use TheHuzz [11] and its coverage tests as the baseline.

A. Training

Preprocessing the PP-test corpus. As mentioned in Section IV-C, we develop a test minimizer to remove redundant tests and implement it using DOcplex [57] from IBM. To construct the coverage matrix, we developed a parser to parse VCS coverage databases (i.e., .vdb). After executing each test, the parser identifies the RTL modules and their associated coverage points, determines whether each point was reached, and encodes this as a binary vector, with the ordering aligned to the traversal sequence of the module hierarchy. This binary vector serves as a row in the coverage matrix. The parser repeats this process for all PP-tests.

Sampling coverage contexts. We sample coverage contexts from the training processors during fuzzing with TheHuzz and Cascade. We obtain coverage feedback using *Synopsys* VCS [50]. Starting at 55% total coverage, we sample a new coverage context at every 5% increment.

Training stage. We implement both the original and adaptive CB algorithms using the MABWiser Python library [58]. To encourage exploration of diverse, high-effective tests, we adopt the ϵ-greedy algorithm for the CB algorithm with an exploration probability (ϵ) of 0.2. We use branch coverage as the target metric due to its importance in uncovering vulnerabilities [33]. ReFuzz can also be configured for other coverage metrics. We observe that branch coverage often starts above 50% and can quickly exceed 60%, but growth slows around 65% across the three training processors. Based on this observation, we define coverage context at $\{55\%, 60\%, 65\%, 70\%\}$. These coverage contexts are also applied during testing to determine appropriate coverage lists.

To maximize effectiveness and avoid test overlap between coverage lists, we begin training ReFuzz from the highest coverage context (e.g., 70%) and use the remaining tests to train the lower contexts. During training, ReFuzz randomly selects one of the training processors to compute the reward of each test for a given coverage context.

Configuring the CB model. According to Algorithm 1, we heuristically configure the number of arms k to 100 and check window γ to 3. ReFuzz is trained at 10000 steps to identify effective tests for each coverage context. A critical parameter in the adaptive CB algorithm is the adaptive threshold θ, which determines test effectiveness. Setting θ too high may result in too few tests being identified (fewer than 10), while setting it too low may include too many, diluting the selection of optimal tests. To balance this tradeoff, we configure θ separately for each coverage context: $\theta = \{55\% : 1.90, 60\% : 1.50, 65\% : 0.90, 70\% : 1.26\}$. The threshold ensures that, by the end of training, each coverage list includes approximately 100 effective tests, aligning with the number of arms k. Appendix B details how we fine-tune the threshold automatically.

Manual efforts and automation. Manual efforts are required to set up existing fuzzers for processors and collect coverage results because fuzzers may not support all benchmarks. This is a general step for all fuzzers. For vulnerability tests, we obtain them directly from artifacts or documented GitHub issues. Once the PP-test corpora and fuzzers are configured, the training and testing processes are fully automatic.

VI. Evaluation

We evaluate ReFuzz comprehensively to answer the following questions:

- **Q1.** Can ReFuzz effectively reuse prior-processor tests (PP-tests) to detect variants of known vulnerabilities on processor-under-tests (PUTs)?
- **Q2.** Is ReFuzz agnostic to existing fuzzers, and can it outperform them on total coverage and coverage speed?
- **Q3.** What are the individual contributions of various ReFuzz optimizations and parameters?

We first discuss the new vulnerabilities and bugs detected by ReFuzz, highlighting their connection to the design reuse strategy. To show the generalization of ReFuzz across different fuzzers, we integrate ReFuzz with TheHuzz [11] and Cascade [12] and evaluate the improvement on total coverage and coverage speed. Each fuzzer generates 21K tests per processor, and we repeat the experiment three times to evaluate the average results. Finally, we analyze how the optimization and configuration impact the efficiency of ReFuzz.

A. New Vulnerabilities and Bugs

ReFuzz detected three new security vulnerabilities and two new functional bugs across multiple processors. Affected commits are mentioned in Section V. **B1** and **B2** lead to incorrect

outputs of performance counters. Bug **B1** exists in Rocket Core [7], BOOMV3, and BOOMV4 [6], all of which share a hardware module. The bug exists across three processors due to **design reuse** and characteristics of the *Chisel* hardware programming language [5]. Bug **B2** affects both BOOMV3 and BOOMV4, with BOOMV4 exhibiting additional variants due to its new microarchitectures, as discussed in Section III. Precise implementation of performance counters is critical for performance analysis, profiling, and debugging [47]. In addition, performance counters are used for malware detection [49]. Therefore, instructions that produce incorrect counter values can reduce detection accuracy and may also serve as side channels for information leakage. A detailed analysis of these two bugs is provided in Appendix C.

The vulnerabilities are severe, with CVSS scores ranging from 7.1 to 8.5 (out of 10) [59], and enable exploit scenarios, such as denial of service (DoS) or unauthorized access. The vendors either acknowledge the new vulnerabilities and bugs, or we have identified their root causes. To answer **Q1**, ReFuzz effectively reuses PP-tests to detect variants of known vulnerabilities on PUTs.

Vulnerability V1 (CVSS score: 7.1). A memory deadlock occurs on the RSD processor when executing the FENCE.I instruction. ReFuzz reuses the vulnerability test that triggers a FENCE.I-based vulnerability in CVA6 [8] to detect this vulnerability. **V1** presents a denial-of-service vector. Any user-mode program capable of executing arbitrary instructions can invoke FENCE.I to cause the memory deadlock of the processor. A potential exploitability context is shown in Appendix D.

Vulnerability V2 (CVSS score: 8.5). RSD executes *illegal* LOAD instructions, leading to DoS attacks or unauthorized access attacks. A malicious attacker can execute instructions to compromise the system's security. This vulnerability is an undocumented hardware feature, CWE-1242 [60].

Vulnerability V3 (CVSS score: 8.5). This vulnerability is similar to **V2** except that the opcode corresponds to the STORE instruction. Both vulnerabilities **V2** and **V3** stem from the same root cause in RSD's instruction decoding logic, as discussed in Appendix E. **V2** can allow malicious programs to probe memory in ways not anticipated by the software stack, potentially enabling information disclosure. **V3** can compromise data integrity in user-mode or shared-memory scenarios, weaken isolation boundaries, and overwrite control data structures managed by higher-privileged components. Appendix D shows their potential exploitability context.

B. Vulnerability Detection Speed

We compared ReFuzz's efficiency against TheHuzz and Cascade on vulnerability detection. Since ReFuzz reuses inputs that trigger known bugs in training processors, we evaluate time-to-bug and required tests to detect new vulnerabilities in testing processors (RSD and BOOMV4), which prevents overfitting. To measure time-to-bug, we use VCS as the simulation tool for all fuzzers. Table III summarizes the results. Because a wide range of instructions can trigger the two new bugs, as shown in Table IV (See Appendix C), their time-to-bug numbers are not informative for speed comparison and were excluded. But they still demonstrate how IP reuse can propagate vulnerabilities and how different microarchitectures can introduce vulnerability variants.

ReFuzz takes an average $6.29\times$ less real time and $5.98\times$ less tests to detect new vulnerabilities. ReFuzz detects **V1** faster than TheHuzz [11] because the seed that triggers another FENCE.I-related vulnerability in CVA6 is in the *vulnerability list* of ReFuzz. As a result, ReFuzz does not need to generate such seeds from scratch. ReFuzz's test speed is similar to Cascade for **V1** because Cascade aims to generate long inputs with diverse instructions, increasing the probability of containing the instructions that can trigger **V1**. However, the time-to-bug is $14.76\times$ faster because the PP-test only contains the instructions that can trigger the **V1**, leading to less simulation time. While users have to identify the instructions that trigger **V1** from Cascade's long inputs. This also shows that reusing effective PP-tests can further enhance the following debugging process.

However, ReFuzz detected **V2** and **V3** that are not on the *vulnerability list* slower than TheHuzz. This is because ReFuzz prioritizes the PP-tests of known vulnerability as mentioned in Section IV-F, which delays the identification of **V2** and **V3**. Adding the corresponding tests for **V2** and **V3** will improve the speed of vulnerability detection. Moreover, integrating ReFuzz with Cascade can not detect these two vulnerabilities because Cascade constrains the values of the "funct3" field of LOAD and STORE instructions to always be in the valid range. This shows the need to include PP-tests from fuzzers with distinct mechanisms and include mutation processes to enhance design space exploration.

C. Coverage Evaluation

To answer **Q2**, we compare the capability of ReFuzz in achieving coverage with the baseline fuzzers: TheHuzz [11] and Cascade [12]. Unlike the evaluations in Sections III-B and IV-C, we use the PP-tests from both fuzzers to train ReFuzz for comprehensive evaluation. Across two testing processors, BOOMV4 and RSD, ReFuzz achieves a $511.23\times$ coverage speedup on average than the baseline fuzzers. Specifically, ReFuzz generates $511.23\times$ fewer tests to achieve the same total coverage as the baseline fuzzers. For total coverage, ReFuzz achieves an average of 1.89% more total coverage..

Though using CVA6, Rocket Core, and BOOMV3 for training, ReFuzz outperforms both baseline fuzzers in total coverage. On average, ReFuzz achieves 3.48% more total coverage across all five processors, highlighting ReFuzz's ability to reuse highly effective tests to help fuzzers explore more design spaces. Figure 6 shows coverage results.

Comparison with TheHuzz. For testing processors, ReFuzz achieves 5.94% more total coverage and a $1818.27\times$ speedup compared to TheHuzz on BOOMV4 and achieves similar total coverage (68.45%) as TheHuzz (68.54%) on RSD. On the training processors, ReFuzz achieves 6.11% more total coverage on CVA6 and 9.33% more total coverage on Rocket

Table III
SUMMARY OF VULNERABILITY DETECTION SPEED OF THEHUZZ [11], CASCADE [12], AND REFUZZ. N.D. REFERS "NOT DETECTED."

Vulnerability	Processor	CWE	Time (sec)						Tests					
			TheHuzz	ReFuzz+ TheHuzz	Speedup	Cascade	ReFuzz+ Cascade	Speedup	TheHuzz	ReFuzz+ TheHuzz	Speedup	Cascade	ReFuzz+ Cascade	Speedup
V1: FENCE.I causes memory deadlock	RSD	833	137.44	15.73	8.74×	202.78	13.74	14.76×	83.33	4.00	20.83×	4.67	4.00	1.17×
V2: Illegal LOAD can be executed	RSD	1242	478.43	853.65	0.56×	N.D.	N.D.	—	302.33	478.33	0.63×	N.D.	N.D.	—
V3: Illegal STORE can be executed	RSD	1242	596.06	545.18	1.09×	N.D.	N.D.	—	375.00	296.00	1.27×	N.D.	N.D.	—

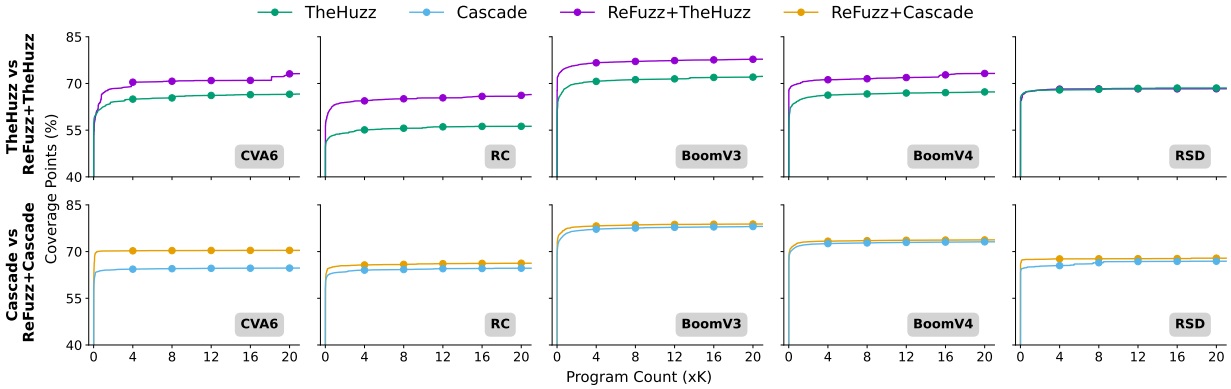

Figure 6. Branch coverage results of TheHuzz [11], Cascade [12], and ReFuzz.

Core. This is because ReFuzz leverages the mutation engines of TheHuzz to mutate tests with complex data- and control-flow logic from Cascade, which helps explore more design spaces than randomly generated seeds.

Comparison with Cascade. For testing processors, ReFuzz achieves 0.63% more total coverage and a 10.90× speedup compared to Cascade on BOOMV4. On RSD, ReFuzz achieves similar total coverage (67.88%) as Cascade does (66.91%) but still achieves a 215.06× coverage speed than Cascade. On the training processors, ReFuzz achieves 1.30% more coverage on Rocket Core. This is because ReFuzz incorporates tests generated by TheHuzz, which helps explore corner cases. ReFuzz achieves limited improvement on total coverage when integrating with Cascade, primarily because Cascade lacks a mutation engine and depends only on the existing tests.

Yet, ReFuzz achieves 5.48% more total coverage and a 1086.98× speedup over Cascade on CVA6, because the FENCE.I will cause CVA6 to hang using Cascade's setup. Since most of Cascade's tests include this instruction, they hang early in execution, preventing subsequent instructions from running and limiting their ability to explore the design spaces of CVA6. ReFuzz eliminates most of these tests during training, highlighting the importance of dropping ineffective tests and evaluating the effectiveness of tests across processors.

Coverage improvement on RSD is limited compared to both baseline fuzzers because RSD is a 32-bit RISC-V processor, whereas all training processors are 64-bit. As a result, many instructions of PP-tests identified during training are illegal or invalid on RSD, reducing the effectiveness of tests and hindering design space exploration. To address this limitation, future iterations of ReFuzz could be trained using a set of 32-bit RISC-V processors.

Evaluating condition coverage. To evaluate the comprehensiveness of ReFuzz, we also use condition coverage as the target metric. Compared to branch coverage, condition coverage is a finer coverage metric that monitors the possible signal combinations in branch statements [33]. The coverage context and the adaptive threshold θ are configured as $\theta = \{45\% : 14.84, 50\% : 11.82, 55\% : 7.81, 60\% : 4.69, 65\% : 2.86\}$. Across two testing processors, BOOMV4 and RSD, ReFuzz achieves an average of 223.57× coverage speed and 2.2% more total coverage. On average, ReFuzz achieves a 279.44× coverage speed and 3.05% more total coverage across all five processors. Appendix G includes more details.

Overall, to answer **Q2**, ReFuzz is agnostic to existing fuzzers and outperforms them in both total coverage and coverage speed. These improvements highlight its ability to generate more effective tests and accelerate the exploration of processor design spaces, making it a robust and scalable solution for enhancing vulnerability detection across a wide range of processors.

D. Evaluating ReFuzz's Optimizations and Parameters

To answer **Q3**, in Sections III-B and IV-C, we have analyzed how ReFuzz leverages the mutation engines of baseline fuzzers to achieve higher total coverage, and how the adaptive CB algorithm helps ReFuzz identify effective tests compared to the original CB algorithm.

In this section, we further examine how different optimizations and parameters affect ReFuzz's efficiency. Specifically, we focus on the impact of the **PP-test corpus** and the number

of **training steps** on the efficiency and effectiveness of the trained CB model. Unlike other parameters such as the number of arms, which make minor adjustments to the configuration of ReFuzz's CB model, the PP-test corpus determines the effectiveness of tests, while the training steps decide whether the model has sufficient time to identify optimal tests.

To make a comprehensive analysis, we prepared three PP-test corpora of different sizes. The **all** corpus includes all PP-tests generated by both fuzzers. The **interesting** corpus includes only interesting PP-tests that achieve incremental coverage during fuzzing the prior processors. The **minimized** corpus consists of PP-tests identified by our test minimizer.

Test minimizer. Figure 7 shows the results of applying the test minimizer to the all corpus to generate the minimized corpus. Compared to the original 21K tests generated by each fuzzer, it significantly reduces the corpus size. For example, it reduces the number of TheHuzz tests for Rocket Core from 21K to 174, achieving a 99.17% reduction rate. On average, the test minimizer takes 944.28 seconds and achieves a 98.76% reduction rate compared to the number of PP-tests in the all corpus and a 65.89% reduction rate compared to the interesting corpus. This substantial reduction highlights the high degree of redundancy among tests and underscores the importance of reusing highly effective tests to improve fuzzing efficiency.

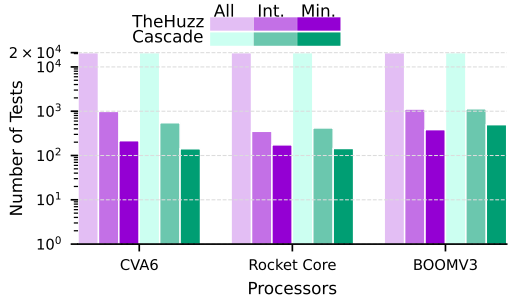

Figure 7. The number of tests the test minimizer reduces from TheHuzz [11] and Cascade [12]. "Int." represents the interesting corpus, and "Min." represents the minimized corpus.

Different training steps and PP-test corpora. CB algorithms support *anytime learning*, meaning that longer training generally yields better results. To study this effect, we vary the number of training steps, $0.5K, 1K, 2K, 5K, 8K$, and $10K$, and test each setting across three PP-test corpora. For every setting, we measure how many effective tests the adaptive CB algorithm identifies and repeat the experiment three times to report average results.

Figure 8 presents the average number of effective tests on the coverage list when the coverage context is 70%. Using the all corpus as an example, we observe that as training steps increase from $0.5K$ to $10K$, the average number of effective tests rises from 4 to 51, demonstrating the *anytime* property of CB algorithms. Because the minimized corpus contains the smallest subset of PP-tests that achieves equivalent total coverage as the all corpus, ReFuzz is able to identify more effective tests from it under the same training steps. For example, at $10K$ training steps, ReFuzz identifies an average

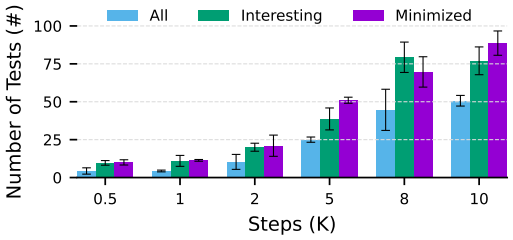

Figure 8. Training ReFuzz's CB model with different training steps.

of 89 effective tests from the minimized corpus, compared to 77 from the interesting corpus and 51 from the all corpus. This result underscores the impact of the test minimizer in improving the training efficiency and the CB model's effectiveness.

Compared to other coverage contexts, identifying the effective tests for the 70% coverage context requires longer training steps, as the remaining uncovered points tend to be corner cases. ReFuzz requires fewer training steps to find effective tests for other coverage contexts.

To answer **Q3**, the test minimizer effectively reduces the size of the PP-test corpus while preserving total coverage, demonstrating its value in eliminating redundant tests. Analysis of different training steps confirms the anytime property of CB algorithms and indicates that highly effective tests identified by the test minimizer can enhance the efficiency of the training process.

VII. Related Work

Seed generation. Traditional approaches, such as those used in DIFUZZRTL [24] and ProcessorFuzz [23], rely on randomly generating instruction sequences [11], [24], [26], [42]. MorFuzz [13] and Cascade [12] increase input diversity by enforcing control- and data-flow constraints during generation. HyPFuzz [33] uses formal verification to synthesize inputs that reach hard-to-reach design spaces but scale poorly with complex designs. Machine learning-based approaches [35], [61], such as HFL [37] and GenHuzz [36], explore the use of large language models or reinforcement learning to generate instruction streams with semantic and structural dependencies. Despite promising early results, these methods often suffer from hallucinations or invalid tests, limiting their practicality.

Seed schedule. MABFuzz [56] leverages multi-armed bandit approaches and coverage feedback to identify an optimal schedule of seeds for each PUT. However, it is incompatible with non-feedback-based fuzzers like Cascade and relies on the baseline fuzzer's seed-generation strategy. ReFuzz discards fuzzers' internal strategies and intelligently uses effective PP-tests to enhance fuzzing efficiency.

Coverage metrics. Existing processor fuzzers either rely on industry-standard code coverage metrics [11], [33], [35]–[37] or define custom hardware-specific metrics [13], [23], [24], [43]. Standard coverages are broadly compatible with existing design flows and simulation environments widely used in industry [62], [63], making them the most practical option for testing production-grade designs. Custom coverage metrics

attempt to capture more internal behaviors, such as multiplexer activity [43], register toggling [24], or instruction semantics [13] but face integration barriers to industry verification flows. ReFuzz adopts industry-standard code coverage and ensures simple integration with industry verification flows.

Vulnerability detection. Most processor fuzzers are designed to detect functional bugs through differential tests [11], [23], [24], [33], [35], [36], [61]. Orthogonal efforts target microarchitectural vulnerabilities, including speculative execution attacks [28]–[31] and timing side-channels [25], [27], leveraging techniques like information flow tracking (IFT) [64], [65]. These works differ in their threat models and detection methods but share the same underlying need for effective test generation. Since ReFuzz focuses on optimizing test reuse, it is agnostic to the target vulnerability and can be integrated with a wide range of detection strategies.

Overall, ReFuzz fills a key gap in prior work, which treats each processor independently and generates seeds from scratch, ignoring useful information from earlier testing efforts. ReFuzz introduces a CB-based test reuse framework that adaptively selects and mutates tests from prior processors for the current PUT. This approach improves coverage and increases the likelihood of uncovering cross-generational vulnerabilities and their variants.

VIII. Discussion

Integrating ReFuzz with industry verification pipelines. ReFuzz naturally fits into the continuous integration and regression testing pipeline for IP vendors due to its *anytime learning* property, allowing ReFuzz to continuously identify and reuse effective PP-tests while keeping the trained model within a reasonable size. Moreover, besides *Synopsys* VCS [50], ReFuzz is compatible with coverage metrics from other industrial simulators, such as *Cadence* Xcelium [66] and *Siemens* Questa One Sim [67], which also generate coverage files similar to VCS's coverage database.

Future challenges. To further strengthen ReFuzz's impact on the semiconductor development cycle, we identify three key challenges: **(i) Generalizability.** In practice, hardware designs vary in ISA extensions (e.g., 32-bit vs. 64-bit RISC-V, or cross-ISA settings such as RISC-V and ARM) and verification platforms (e.g., FPGA-prototyping, black-box, and hybrid flows). **(ii) Sparsity.** Reuse may be constrained when vendors have limited historical designs or employ heavily customized architectures. And **(iii) Scalability.** The reliance on integer programming to build the test minimizer may raise scalability concerns when applied to large industrial test suites.

Future directions. For generalizability, ReFuzz can evolve in two directions. First, it can support more contexts, such as ISA extensions or even cross-ISA scenarios, so that it can reuse more precise tests, similar to selecting ISA extensions when compiling binaries. Second, an abstract CB model could use microarchitectural features, functionalities, and ISA extensions as context, along with mappings between instructions and the features they exercise. Under a given context, the model could identify the relevant features and their interactions and then generate appropriate inputs. Finally, ReFuzz can adapt to verification platforms with different observability levels by using any quantitative metric that defines 0% and 100% coverage. As long as coverage increments can be computed (Definition 3), ReFuzz can operate on that platform.

For sparsity, when historical design data are limited, vendors can enrich their test suites by incorporating tests from other ISAs and translate them to the target ISA. For designs with highly customized or sparse architectures, integrating additional mutation engines into ReFuzz can help explore architecture-specific features.

Finally, for scalability, combining integer linear programming (ILP), clustering, and approximation algorithms can tackle the challenge. One strategy is to partition the overall suite into smaller subsets and build a hierarchical approach. The approaches perform ILP on each subset, merge the resulting minimal suites, and then run ILP again to obtain a global minimal set. This approach, coupled with parallelization, can substantially improve efficiency. If scalability issues persist even after clustering, approximation algorithms, such as greedy heuristics, can provide near-optimal solutions with significantly lower overhead.

IX. Conclusion

Design reuse leads to vulnerabilities that propagate across processor generations. ReFuzz proposes the first test reuse hardware fuzzing framework that reuse and mutate effective tests from prior processors to enhance fuzzing on processors-under-test, leveraging contextual bandit algorithms. ReFuzz is also agnostic to processor fuzzers or any dynamic techniques that require seeds to initiate their processes. ReFuzz uncovered three new vulnerabilities. One was triggered by the same test that triggered vulnerabilities in a prior processor. It detects two functional bugs across three processors due to shared designs. Evaluations show that ReFuzz achieves an average 511.23× coverage speed and 1.89% more total coverage over baseline fuzzers. These results establish ReFuzz as a practical solution for processor security, especially for companies like *Intel* and *AMD*, which have effective tests from a broad range of prior processors.

Acknowledgment

We thank Stephen Muttathil for his contribution to Cascade implementation. Our research work was partially funded by Intel's Scalable Assurance Program, ONR Award #N00014-18-1-2058, the Lockheed Martin Corporation, NSF-Grant 2344914, SRC-3308.001, the SCALE Fellowship Program (M2402515), DFG-SFB 1119-236615297, the European Union under Horizon Europe Programme-Grant Agreement 101070537-CrossCon, NSF-DFG-Grant 538883423, and the European Research Council under the ERC Programme-Grant 101055025-HYDRANOS. This work does not in any way constitute an Intel endorsement of a product or supplier. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors.

X. Ethics Considerations

We consider the paper without potentially negative outcomes from tangible harms and violations of human rights. We consider the full spectrum of stakeholders and the paper, including no harms related to exposing the identity of research subjects, such as the individual identities, groups, and organizations, and the behaviors, communications, or relationships associated with such identification. **Respect for Persons.** The research targets open-sourced hardware designs and does not involve natural persons, live systems, or certain data that identifies them. Thus, no harm is caused to human subjects, non-subjects, and information and communication technology users, such as disruption of access, loss of privacy, or unreasonable constraints on protected speech or activities. **Beneficence.** To minimize the relevant harmful impacts, we contacted the vendors of open-sourced benchmarks as soon as our strategy detected the vulnerabilities. Moreover, based on our current knowledge, the vulnerabilities exist in an open-source benchmark, which has not been used for any commercial purposes. We discuss the root causes of vulnerabilities in the paper to help vendors and potential users mitigate the vulnerabilities. **Justice: Fairness and Equity.** We select benchmarks that are widely used in existing papers. We select existing techniques as baselines based on their distinct and unique features. We evaluate the performance of our technique and baselines on the same platform for a fair comparison.

References

[1] D. A. Patterson and J. L. Hennessy, "Computer Organization and Design ARM edition: the Hardware Software Interface," *Morgan Kaufmann*, 2016.

[2] "Intel® 64 and IA-32 Architectures Software Developer Manuals," https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html, last accessed on 11/05/2025.

[3] "Intel® Core™ Processor Family," https://www.intel.com/content/www/us/en/products/details/processors.html, last accessed on 11/05/2025.

[4] I. Manage, "Design Data Management Worldwide Trends – Survey Report," https://www.icmanage.com/design-data-management-worldwide-trends-survey-report/, 2013, last accessed on 11/20/2025.

[5] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing Hardware in a Scala Embedded Language," *Design Automation Conference*, 2012.

[6] J. Zhao, B. Korpan, A. Gonzalez, and K. Asanovic, "SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine," *Fourth Workshop on Computer Architecture Research with RISC-V*, May 2020.

[7] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman, "The Rocket Chip Generator," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17, Apr 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[8] F. Zaruba and L. Benini, "The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology," *IEEE Transactions on Very Large Scale Integration Systems*, 2019.

[9] YosysHQ, "PicoRV32 - A Size-Optimized RISC-V CPU," https://github.com/YosysHQ/picorv32, 2019, Last accessed on 07/07/2025.

[10] SonalPinto and F. Solt, "Kronos RISC-V," https://github.com/YosysHQ/picorv32, 2019, Last accessed on 07/07/2025.

[11] R. Kande, A. Crump, G. Persyn, P. Jauernig, A.-R. Sadeghi, A. Tyagi, and J. Rajendran, "TheHuzz: Instruction Fuzzing of Processors Using Golden-Reference Models for Finding Software-Exploitable Vulnerabilities," *USENIX Security Symposium*, 2022.

[12] F. Solt, K. Ceesay-Seitz, and K. Razavi, "Cascade: CPU Fuzzing via Intricate Program Generation," *USENIX Security Symposium*, 2024.

[13] J. Xu, Y. Liu, S. He, H. Lin, Y. Zhou, and C. Wang, "Morfuzz: Fuzzing processor via runtime instruction morphing enhanced synchronizable co-simulation," *USENIX Security Symposium*, 2023.

[14] M. Ender, A. Moradi, and C. Paar, "The Unpatchable Silicon: a Full Break of the Bitstream Encryption of Xilinx 7-Series FPGAs," *USENIX Security Symposium*, 2020.

[15] "USN-3597-2: Linux kernel (HWE) vulnerabilities," https://ubuntu.com/security/notices/USN-3597-2, last accessed on 11/05/2025.

[16] "USN-3531-3: intel-microcode update," https://ubuntu.com/security/notices/USN-3531-3, last accessed on 11/05/2025.

[17] "KB4457951: Windows guidance to protect against speculative execution side-channel vulnerabilities," https://support.microsoft.com/en-us/topic/kb4457951-windows-guidance-to-protect-against-speculative-execution-side-channel-vulnerabilities-ae9b7bcd-e8e9-7304-2c40-f047a0ab3385, last accessed on 11/05/2025.

[18] "Intel Annual Report, 1994," https://www.intel.com/content/www/us/en/history/history-1994-annual-report.html, last accessed on 11/05/2025.

[19] B. Bailey, "Practical Processor Verification," https://semiengineering.com/practical-processor-verification/, 2020, Last accessed on 07/14/2025.

[20] "User Stories," https://cwe.mitre.org/about/user_stories.html, Last accessed on 07/14/2025.

[21] S. Ye, "Best Practices for a Reusable Verification Environment," https://www.eetimes.com/best-practices-for-a-reusable-verification-environment, 2004, Last accessed on 07/08/2025.

[22] M. Rostami, C. Chen, R. Kande, H. Li, J. Rajendran, and A.-R. Sadeghi, "Fuzzerfly Effect: Hardware Fuzzing for Memory Safety," *IEEE Security & Privacy*, 2024.

[23] S. Canakci, C. Rajapaksha, L. Delshadtehrani, A. Nataraja, M. B. Taylor, M. Egele, and A. Joshi, "ProcessorFuzz: Processor Fuzzing with Control and Status Registers Guidance," *IEEE International Symposium on Hardware Oriented Security and Trust*, 2023.

[24] J. Hur, S. Song, D. Kwon, E. Baek, J. Kim, and B. Lee, "DIFUZZRTL: Differential Fuzz Testing to Find CPU Bugs," *IEEE Symposium on Security and Privacy*, 2021.

[25] C. Rajapaksha, L. Delshadtehrani, M. Egele, and A. Joshi, "SIGFuzz: A Framework for Discovering Microarchitectural Timing Side Channels," *Design, Automation & Test in Europe Conference & Exhibition*, 2023.

[26] Y. Sugiyama, R. Matsuo, and R. Shioya, "SurgeFuzz: Surge-Aware Directed Fuzzing for CPU Designs," *IEEE/ACM International Conference on Computer Aided Design*, 2023.

[27] P. Borkar, C. Chen, M. Rostami, N. Singh, R. Kande, A.-R. Sadeghi, C. Rebeiro, and J. Rajendran, "Whisperfuzz: White-box Fuzzing for Detecting and Locating Timing Vulnerabilities in Processors," *USENIX Security Symposium*, 2024.

[28] J. Hur, S. Song, S. Kim, and B. Lee, "SpecDoctor: Differential Fuzz Testing to Find Transient Execution Vulnerabilities," *ACM SIGSAC Conference on Computer and Communications Security*, 2022.

[29] M. Rostami, S. Zeitouni, R. Kande, C. Chen, P. Mahmoody, J. R. Rajendran, and A.-R. Sadeghi, "Lost and Found in Speculation: Hybrid Speculative Vulnerability Detection," *ACM/IEEE Design Automation Conference*, 2024.

[30] A. de Faveri Tron, R. Isemann, H. Ragab, C. Giuffrida, K. von Gleissenthall, and H. Bos, "Phantom Trails: Practical Pre-Silicon Discovery of Transient Data Leaks," *USENIX Security Symposium*, 2025.

[31] M. Ghaniyoun, K. Barber, Y. Zhang, and R. Teodorescu, "IntroSpectre: A Pre-Silicon Framework for Discovery and Analysis of Transient Execution Vulnerabilities," *International Symposium on Computer Architecture*, 2021.

[32] X. Zhu, S. Wen, S. Camtepe, and Y. Xiang, "Fuzzing: A Survey For Roadmap," *ACM Computing Surveys*, 2022.

[33] C. Chen, R. Kande, N. Nguyen, F. Andersen, A. Tyagi, A.-R. Sadeghi, and J. Rajendran, "HyPFuzz: Formal-Assisted Processor Fuzzing," *USENIX Security Symposium*, 2023.

[34] C. Chen, V. Gohil, R. Kande, A.-R. Sadeghi, and J. Rajendran, "PSOFuzz: Fuzzing Processors with Particle Swarm Optimization," *IEEE/ACM International Conference on Computer Aided Design*, 2023.

[35] M. Rostami, M. Chilese, S. Zeitouni, R. Kande, J. Rajendran, and A.-R. Sadeghi, "Beyond Random Inputs: A Novel ML-Based Hardware Fuzzing," *Design, Automation & Test in Europe Conference & Exhibition*, 2024.

[36] L. Wu, M. Rostami, H. Li, J. Rajendran, and A.-R. Sadeghi, "GenHuzz: An Efficient Generative Hardware Fuzzer," *USENIX Security Symposium*, 2025.
[37] L. Wu, M. Rostami, H. Li, and A.-R. Sadeghi, "HFL: Hardware Fuzzing Loop with Reinforcement Learning," *Design, Automation & Test in Europe Conference*, 2025.
[38] A. M. Inc., "Revision Guide for AMD Family 17h Models 00h-0Fh Processors," https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/revision-guides/55449.pdf, Last accessed on 01/21/2025.
[39] "Verification Stages," https://www.chipverify.com/verification/verification-stages, Last accessed on 07/14/2025.
[40] S. Davidmann, "RISC-V Processor Verification Requires the Full Toolbox," https://www.tessolve.com/wp-content/uploads/2025/07/Simon-Davidmann-2.pdf, 2025, Last accessed on 07/17/2025.
[41] K. McDermott, "Getting Started with RISC-V Verification," https://riscv.org/blog/2020/05/getting-started-with-risc-v-verification/, 2020, Last accessed on 07/14/2025.
[42] T. Trippel, K. G. Shin, A. Chernyakhovsky, G. Kelly, D. Rizzo, and M. Hicks, "Fuzzing Hardware Like Software," *USENIX Security Symposium*, 2022.
[43] K. Laeufer, J. Koenig, D. Kim, J. Bachrach, and K. Sen, "RFUZZ: Coverage-Directed Fuzz Testing of RTL on FPGAs," *IEEE/ACM International Conference on Computer-Aided Design*, 2018.
[44] lcamtuf, "American Fuzzy Lop Fuzzer." 2014, Last accessed on 11/18/2025. [Online]. Available: {https://github.com/google/AFL}
[45] T. Lattimore and C. Szepesvári, "Bandit algorithms," *Cambridge University Press*, 2020.
[46] RISC-V, "RISC-V Specifications," https://riscv.org/technical/specifications/, 2025, last accessed on 11/19/2025.
[47] J. Demme and S. Sethumadhavan, "Rapid identification of architectural bottlenecks via precise event counting," *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, 2011.
[48] V. Weaver and S. McKee, "Can hardware performance counters be trusted?" *Proceedings of the IEEE International Symposium on Workload Characterization*, 2008.
[49] X. Wang, C. Konstantinou, M. Maniatakos, and R. Karri, "Confirm: Detecting firmware modifications in embedded systems using hardware performance counters," *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2015.
[50] "VCS," https://www.synopsys.com/verification/simulation/vcs.html, 2025, Last accessed on 11/19/2025.
[51] H.-Y. Hsu and A. Orso, "MINTS: A General Framework and Tool for Supporting Test-Suite Minimization," *International Conference on Software Engineering*, 2009.
[52] K. Panaganti, Z. Xu, D. Kalathil, and M. Ghavamzadeh, "Robust Reinforcement Learning using Offline Data," *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.
[53] S. Mashimo, A. Fujita, R. Matsuo, S. Akaki, A. Fukuda, T. Koizumi, J. Kadomoto, H. Irie, M. Goshima, K. Inoue *et al.*, "An Open Source FPGA-Optimized Out-of-Order RISC-V Soft Processor," *International Conference on Field-Programmable Technology*, 2019.
[54] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb, S. Karandikar, H. Liew, A. Magyar, H. Mao, A. Ou, N. Pemberton, P. Rigge, C. Schmidt, J. Wright, J. Zhao, Y. S. Shao, K. Asanović, and B. Nikolić, "Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs," *IEEE Micro*, 2020.
[55] RISC-V, "SPIKE Source Code," https://github.com/riscv-software-src/riscv-isa-sim, 2023, last accessed on 11/22/2025.
[56] V. Gohil, R. Kande, C. Chen, A.-R. Sadeghi, and J. Rajendran, "MABfuzz: Multi-Armed Bandit Algorithms for Fuzzing Processors," *Design, Automation & Test in Europe Conference & Exhibition*, 2024.
[57] IBM, "IBM Decision Optimization for Watson Studio," https://www.ibm.com/products/decision-optimization-for-watson-studio, Last accessed on 01/05/2025.
[58] E. Strong, B. Kleynhans, and S. Kadioglu, "MABWiser: Parallelizable Contextual Multi-armed Bandits," *Int. J. Artif. Intell. Tools*, 2021.
[59] "Common Vulnerability Scoring System Calculator," https://nvd.nist.gov/vuln-metrics/cvss/v3-calculator, 2019, Last accessed on 08/02/2025.
[60] MITRE, "Hardware Design CWEs," https://cwe.mitre.org/data/definitions/1194.html, 2019, Last accessed on 11/08/2025.
[61] R. Götz, C. Sendner, N. Ruck, M. Rostami, A. Dmitrienko, and A.-R. Sadeghi, "RLFuzz: Accelerating Hardware Fuzzing with Deep Reinforcement Learning," *IEEE International Symposium on Hardware Oriented Security and Trust*, 2025.
[62] "Companies using Synopsys VCS," https://enlyft.com/tech/products/synopsys-vcs, 2025.
[63] "List of companies using Synopsys VCS," https://theirstack.com/en/technology/synopsys-vcs, 2025.
[64] W. Hu, A. Ardeshiricham, and R. Kastner, "Hardware Information Flow Tracking," *ACM Computing Surveys*, 2021.
[65] F. Solt, B. Gras, and K. Razavi, "$CellIFT$: Leveraging Cells for Scalable and Precise Dynamic Information Flow Tracking in RTL," *USENIX Security Symposium*, 2022.
[66] "Xcelium Logic Simulator," https://www.cadence.com/en_US/home/tools/system-design-and-verification/simulation-and-testbench-verification/xcelium-simulator.html, 2025, Last accessed on 11/19/2025.
[67] "Questa One Sim," https://eda.sw.siemens.com/en-US/ic/questa-one/simulation/questa-one-sim/, 2025, Last accessed on 11/19/2025.
[68] J. Hong, B. Kveton, M. Zaheer, and M. Ghavamzadeh, "Hierarchical Bayesian Bandits," *International Conference on Artificial Intelligence and Statistics*, 2022.
[69] A. Piziali, "Functional Verification Coverage Measurement and Analysis," *Springer*, 2008.
[70] H. Foster, "FPGA Language and Library Trends," https://blogs.sw.siemens.com/verificationhorizons/2022/11/21/part-6-the-2022-wilson-research-group-functional-verification-study/, 2022.

APPENDIX

A Optimization Model

This section shows the optimization model for the test minimizer as mentioned in Section IV-C.

Consider the following notations:

1) $\mathbf{T}$: coverage matrix $\mathbf{T} := [t_{ij}]$, where t_{ij} is a binary indicator that represents whether the coverage point j has been covered (1) or not (0) in test i.
2) $\mathbf{L_j}$: minimum coverage constraint for coverage point j.
3) $\mathbf{U_j}$: maximum coverage constraint for coverage point j.
4) $\mathbf{x_i}$: $x_i \in 0, 1, \forall i$, a decision variable indicating if test i is in the minimal subset.

The model is then formulated as:

1) **Objective Function**: Identify the minimal subset of tests that maintains the coverage equivalent: **minimize** $\sum_i x_i$.
2) **Constraint 1**: Ignore coverage point j if it is always covered in all tests: $\sum_i t_{ij} x_i = 0$.
3) **Constraint 2**: The constraint for the rest of the coverage points to identify the minimal subset of tests: $L_j \leq \sum_i t_{ij} x_i \leq U_j$, where If at least one test cover point j, we let $L_j = 1$ and $U_j = \infty$. If no tests cover point j, we let $L_j = U_j = 0$.

B Fine-Tune Adaptive Thresholds Automatically

The configuration of the adaptive threshold θ is automated, as shown in Algorithm 2. The goal is to identify approximately the same number of tests, defined by the number of arms k, for each coverage list. The inputs are the PP-test corpus $\mathcal{A}_{\text{corpus}}$, coverage contexts $\mathcal{C}$, the number of arms k, the check window γ, the training step n, and the tolerance factor f. The output is the fine-tuned adaptive thresholds θ.

The algorithm first calculates the acceptable interval for the number of tests (Line 3). Then, for each coverage context, it starts the binary search from 0.00, assuming all tests are effective, and 100.00 means a test can cover all points of a

Algorithm 2 Fine-Tune Adaptive Thresholds.

Inputs: $\mathcal{A}_{\textbf{corpus}}, \mathcal{C}, k, \gamma, n, f$
Outputs: θ
$\theta \leftarrow \emptyset$, $k_{\text{upper}} = (1+f) * k$, $k_{\text{lower}} = (1-f) * k$
for all $c \in \mathcal{C}$ in reverse order **do**
  $\ell \leftarrow 0.00,\ h \leftarrow 100.00,\ \mathtt{t_c} \leftarrow 0.00$
  for $i = 1$ to 14 **do** ▷ Binary search
    $m \leftarrow (\ell + h)/2$
    $\mathcal{A} \leftarrow \textsc{AdapCB}(\mathcal{A}_{\text{corpus}}, c, k, \gamma, m, n)$
    if $|\mathcal{A}| > k_{\text{upper}}$ **then** $\ell \leftarrow m$
    else if $|\mathcal{A}| < k_{\text{lower}}$ **then** $h \leftarrow m$
    else $\mathtt{t_c} \leftarrow m$; **break**
  $\theta \leftarrow \theta \cup \{\mathtt{t_c}\}$
  $\mathcal{A} \leftarrow \textsc{AdapCB}(c, t_c, \mathcal{A}_{\text{corpus}}, n)$
  $\mathcal{A}_{\text{corpus}} \leftarrow \mathcal{A}_{\text{corpus}} \setminus \mathcal{A}$
return θ

training processor (Lines 6 to 11). Given the threshold range with a precision of 0.01, the binary search requires at most 14 iterations. To avoid duplicate tests across coverage lists, the algorithm runs our adaptive CB algorithm once, removes the chosen tests from the PP-test corpus (Lines 13 and 14), and then fine-tunes the threshold for the next coverage context.

Note that some tests may not have coverage results during fine-tuning. To avoid repeated simulations, we simulate the test once and reuse its coverage result. For experimental purposes, we simulate all 1557 tests in parallel after test minimization using 10 threads. The same number of threads is used during evaluation. The total simulation time is approximately 1.14 hours in real time. Algorithm 2 then took around 1.30 hours to identify the thresholds. This fine-tuning stage illustrates the necessity of test minimization and can be further accelerated with additional threads or more advanced parallelization.

To automatically fine-tune more parameters, such as coverage contexts and check window γ, future work can consider involving more advanced algorithms, such as hierarchical contextual bandits [68].

C Details on Detected Bugs

Bug B1. in `Rocket Core` [7], `BOOMV3`, and `BOOMV4` [6], the `ECALL` and `EBREAK` instructions incorrectly increase the value of the `minstret` register, violating the RISC-V specification. According to the RISC-V ISA, these exception-generating instructions should not contribute to the instruction retirement count, and should leave `minstret` unchanged. This bug misinterprets performance counter values and constitutes a violation of expected behavior (CWE-440) [60].

The bug exists across three processors due to **design reuse** and characteristics of the *Chisel* hardware programming language [5]. Specifically, `Rocket Core` has a module called `CSR`, responsible for updating the values of control and status registers (CSRs), including `minstret`. `BOOMV3` and `BOOMV4` use the same module, thereby propagating the same bug. This bug also shows that the widely applied IP reuse strategy in hardware design can potentially propagate vulnerabilities to multiple processors.

Bug B2. In `BOOMV3`, and `BOOMV4` [6], multiple instructions fail to correctly increase the value of `minstret` register. For example, when executing the `MUL` instruction in `BOOMV4`, `minstret` register increments twice instead of once as expected. However, the same bug does not exist in `BOOMV3` and `Rocket Core`, as discussed in Section III. Though three processors use the same `CSR` module, they can include bug variants due to different microarchitectures. Table IV shows the common instructions that can trigger this bug in `BOOMV3` and `BOOMV4` and the additional instructions that can trigger this bug only in `BOOMV4`.

Table IV
INSTRUCTIONS OBSERVED TO INCREMENT THE `MINSTRET` TWICE.

On Both `BOOMV3` and `BOOMV4`	
CSR Instructions	`CSRRC`, `CSRRCI`, `CSRRS`, `CSRRSI`, `CSRRW`, `CSRRWI`
Memory Synchronization	`FENCE`, `FENCE.I`
Load & Store	`LB`, `LBU`, `LH`, `LHU`, `LW`, `LWU`, `LD`, `SB`, `SH`, `SW`, `SD`
Load-Reserved & Store-Conditional	`LR.W`, `LR.D`, `SC.W`, `SC.D`
On `BOOMV4` Only	
Branch	`JALR`, `BGE`, `BLT`, `BLTU`, `BNE`
Multiplication	`MUL`, `MULH`, `MULHSU`, `MULHU`, `MULW`

D Potential Exploitability Contexts of `RSD` Vulnerabilities

V1 presents a denial-of-service vector. Any user-mode program capable of executing arbitrary instructions can intentionally invoke `FENCE.I` to cause the memory deadlock of the processor. This makes the vulnerability practically exploitable on systems where untrusted or partially trusted code is allowed, which interrupts critical tasks, disrupts real-time workloads, and forces a hardware reset. A potential exploitability context is shown in Appendix D. Attackers can exploit **V1** as shown in Listing 1. This exploit demonstrates that issuing a program with `FENCE.I` instruction on the vulnerable `RSD` processor can trigger internal deadlock. The program executes `FENCE.I` directly in user mode without requiring any privileged operations, allowing an attacker to reliably induce a denial-of-service attack on the system.

Listing 1. `FENCE.I` causes memory deadlock on `RSD` [53].

```
int main(void) {
    // Execute FENCE.I instruction
    asm volatile ("fence.i" :::"memory");
    return 0;
}
// RSD outputs on the debug console:
// Deadlock detected
// ...
// Deadlock detected
// MSHR deadlock detected
```

V2 can allow malicious programs to probe memory in ways not anticipated by the software stack, potentially enabling information disclosure. While this behavior alone does not inherently grant privilege escalation, it can weaken memory-safety assumptions in system software, including bounds-checking frameworks that rely on predictable exception semantics. **V3** can compromise data integrity in user-mode or shared-memory scenarios, weaken isolation boundaries, and potentially overwrite control data structures managed by higher-privileged components.

Attackers can exploit **V2** and **V3** as shown in Listing 2. The attacker uses the same illegal load/store encoding (funct3 = 111) to perform unintended sign-extended byte operations. While the expected behavior is for the processor to throw exceptions, `RSD` internally sign-extends the accessed byte and completes the operation as though it were a legal instruction. Because the extension semantics differ from the ISA specification, the resulting value in the destination register or memory location may deviate significantly from the expected result. This behavior enables data corruption during program execution and breaks data integrity.

Listing 2. `RSD` [53] executes *illegal* `LOAD` or `STORE` instructions.

```
volatile uint32_t victim_mem = 0x12345678;
int main(void) {
    uint32_t tmp = 0;
    // Execute STORE instruction with illegal funct3
    asm volatile (
        ".word 0x01DE7023 // illegal STORE (funct3 =
            ↪ 111), rs2 = x29 (t4), rs1 = x28 (t3)"
        :
        :"r"(tmp), "m"(victim_mem)
    );
    // Execute LOAD instruction with illegal funct3
    asm volatile (
        ".word 0x000E7E83 // illegal LOAD (funct3 = 111
            ↪ ), rd = x29 (t4), rs1 = x28 (t3)"
        :"=r"(tmp)
        :"0"(tmp), "m"(victim_mem)
    );
    return 0;}
```

E Root Cause of Illegal `LOAD`&`STORE` Instructions in `RSD`

Listing 3 shows the root cause of vulnerabilities **V2** and **V3** in `RSD`'s decoding logic [53]. The `case` statement decides the data width based on the value of "funct3". However, since `RSD` has a `default` statement, it identifies any unrecognized funct3 values as the byte data width with sign-extended (Lines 725–727). The mitigation can add illegal instruction exception for invalid values.

Listing 3. The root cause of illegal load and store instructions.

```
704 case(funct3)
705     MEM_FUNCT3_SIGNED_BYTE :begin //LB, SB
706         memAccessMode.isSigned = TRUE;
707         memAccessMode.size = MEM_ACCESS_SIZE_BYTE;
708     end
709     MEM_FUNCT3_SIGNED_HALF_WORD :begin //LH, SH
710         memAccessMode.isSigned = TRUE;
711         memAccessMode.size = MEM_ACCESS_SIZE_HALF_WORD;
712     end
... ...
725     default :begin
726         memAccessMode.isSigned = TRUE;
727         memAccessMode.size = MEM_ACCESS_SIZE_BYTE;
... ...
```

F Fuzzer Selection

Existing hardware fuzzers fall into two categories: feedback-based and non-feedback-based fuzzers [22]. Feedback-based fuzzers typically use code coverage [11], [33], [36], [37] or customized metrics [13], [23], [24], [43] to guide exploration of design spaces. In contrast, non-feedback-based fuzzers [12] constrain input spaces based on ISA and processor design routines, introducing random behaviors, such as the random selection of operands and opcodes, to explore a broad range of design spaces.

Feedback-based fuzzers have demonstrated effectiveness in detecting vulnerabilities, but they face compatibility issues with processor designs. For example, `RFUZZ` [43] and `DIFUZZRTL` [24] customize coverage metrics for the toggling of multiplexer signals and control registers, respectively. However, the instrumentation requires designs to be written in *Chisel* [5], while over 80% of hardware designs in industry use *Verilog* or *SystemVerilog*, which are not supported by these fuzzers [69], [70]. `ProcessorFuzz` monitors control and status registers (CSRs) as coverage feedback, but such registers are outputs and do not directly reflect the explored design spaces. As a result, deploying feedback-based fuzzers with customized coverage metrics is incompatible with most processor designs. Nonetheless, since software simulators with code coverage metrics are widely used in semiconductor companies [62], [63], fuzzers using code coverage metrics are compatible with a broader range of processor designs that share the same ISA [11], [13], [36].

Finally, we select the feedback-based fuzzer, `TheHuzz` [11] and the non-feedback-based fuzzer `Cascade` [12] as the baseline fuzzers due to their distinct mechanisms and generalizability. `TheHuzz` uses industrial standard code coverage metrics [50] and deploys AFL-like mutators, similar to those used in software fuzzers [44], which mutate seeds with limited or no knowledge of ISAs. It generates tests that contain 20 instructions, which may limit its ability to detect vulnerabilities that require longer instructions, but enables it to explore corner cases due to its randomness. In contrast, `Cascade` [12] constrains the instruction space based on ISA and generates tests of long instruction sequences with complex data- and control-flows. However, `Cascade` may over-constrain the randomness of instruction generation, missing corner cases.

G Evaluation on Condition Coverage Metric

We evaluated `ReFuzz`'s generalized improvement using *condition* coverage, including both coverage speed and total coverage improvement based on the number of tests (Program Count). Figure 9 shows condition coverage results similar to Figure 6 for branch coverage. The coverage context and the adaptive threshold are configured as $\theta = \{45\% : 14.84, 50\% : 11.82, 55\% : 7.81, 60\% : 4.69, 65\% : 2.86\}$. `ReFuzz` outperforms both `TheHuzz` and `Cascade` on the condition coverage, showing its comprehensiveness. Across two testing processors, `BOOMV4` and `RSD`, `ReFuzz` achieves an average of 223.57× coverage speed and 2.2% more total coverage. On average, `ReFuzz` achieves a 279.44× coverage speed

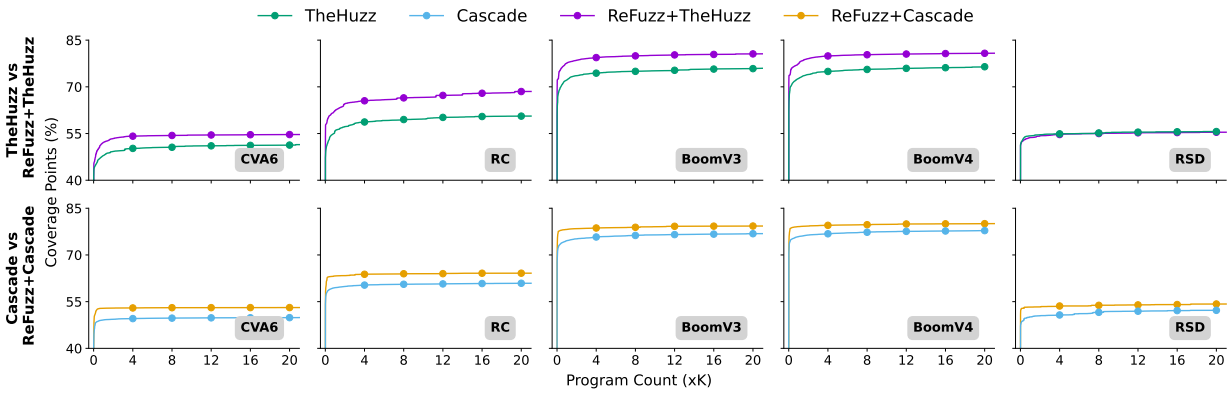

Figure 9. Condition coverage results of `TheHuzz` [11], `Cascade` [12], and `ReFuzz`.

and 3.05% more total coverage across all five processors. The result demonstrates the comprehensiveness of `ReFuzz` on different coverage metrics.

H Integrating the CB Model with Processor Fuzzers

Algorithm 3 shows the integration process of `ReFuzz`'s trained contextual bandit (CB) model with processor fuzzers. The inputs are the trained vulnerability list $\mathcal{A}_v$ and coverage lists $\mathcal{A}_c$ with their learned policies π_v and π_c. it also uses coverage contexts $\mathcal{C}$, the check window γ, and the maximal fuzzing iteration m. The outputs are the achieved total coverage tot_cov. `ReFuzz` starts from the vulnerability list $\mathcal{A}_v$ (Line 4). In `FuzzVulList`, it samples a test a_i from $\mathcal{A}_v$ according to the policy π_v (Line 9). To ensure effectiveness of tests, `ReFuzz` evaluates each selected test a_i every γ fuzzing iterations, checking whether it yields a coverage increment $r_i(tot\_cov, a_i)$ (Line 31) by measuring the cumulative incremental coverage from its mutated variants $f_{cov}(a_i)$. If no coverage increment, the test is dropped (Lines 32–33). Otherwise, `ReFuzz` resets the tracking record # and continues to monitor the coverage increment (Line 35). Once the list is exhausted (Lines 12–13), `FuzzVulList` returns the current total coverage tot_cov (Line 14).

`ReFuzz` then swtiches to coverage lists through `FuzzCovList` (Line 5). `ReFuzz` selects tests based on the cumulative total coverage from the coverage lists $\mathcal{A}_c = \{\mathcal{A}_c^1, \ldots, \mathcal{A}_c^{|\mathcal{C}|}\}$, where $\mathcal{C}$ represents the configured coverage contexts (Line 5). `ReFuzz` continuously monitors the total coverage tot_cov achieved by a processor fuzzer and selects tests according to the policy $\pi(\cdot \mid c)$, corresponding to different intervals of coverage contexts (Lines 16-23). For example, if $tot\_cov < c_1$, it samples a test a_i from the coverage list $\mathcal{A}_c^1$ using the policy $\pi_c(\cdot \mid c_1)$ (Lines 17–18); if $c_1 \leq tot\_cov < c_2$, it samples from $\mathcal{A}_c^2$ using the policy $\pi_c(\cdot \mid c_2)$ (Lines 19–20), and so on for the remaining intervals. This allows `ReFuzz` to adapt its selection of tests to the current total coverage. If all curated tests are exhausted, `ReFuzz` selects new tests generated by the processor fuzzer to explore the unique design features of the PUT (Line 26). After completing at most m iterations, `FuzzCovList` returns the final total coverage tot_cov (Line 29).

Algorithm 3 Integrating the CB Model with Processor Fuzzers

1: **Inputs:** $\mathcal{A}_v, \pi_v, \mathcal{A}_c, \pi_c, \mathcal{C}, \gamma, m$
2: **Outputs:** tot_cov
3: **function** MAIN
4: $\quad tot\_cov \leftarrow$ FUZZVULLIST$(\mathcal{A}_v, \pi_v, \gamma)$
5: $\quad tot\_cov \leftarrow$ FUZZCOVLIST$(\mathcal{A}_c, \pi_c, \mathcal{C}, \gamma, tot_cov, m)$
6: $\quad$ **return** tot_cov
7: **function** FUZZVULLIST($\mathcal{A}_v$, π_v, γ)
8: $\quad$ **while** true **do**
9: $\quad\quad a_i \sim \pi_v(\cdot)$
10: $\quad\quad$ DROPTEST$(a_i, \#(a_i), r(tot_cov, a_i), f_{\text{cov}}(a_i), \mathcal{A}_v, \pi_v, \gamma)$
11: $\quad\quad tot\_cov \leftarrow tot\_cov + f_{cov}(a_i)$
12: $\quad\quad$ **if** $|\mathcal{A}_v| = 0$ **then**
13: $\quad\quad\quad$ **break**
14: $\quad$ **return** tot_cov
15: **function** FUZZCOVLIST($\mathcal{A}_c$, π_c, $\mathcal{C}$, γ, tot_cov, m)
16: $\quad$ **for** $i = 1, 2, \ldots, m$ **do**
17: $\quad\quad$ **if** $tot\_cov < c_1$ **then**
18: $\quad\quad\quad a_i \sim \pi_c(\cdot \mid c_1)$, $a_i \in \mathcal{A}_c^1$
19: $\quad\quad$ **else if** $c_1 \leq tot\_cov < c_2$ **then**
20: $\quad\quad\quad a_i \sim \pi_c(\cdot \mid c_2)$, $a_i \in \mathcal{A}_c^2$
$\quad$ ...
25: $\quad\quad$ **else if** no arm a_i for tot_cov **then**
26: $\quad\quad\quad$ Let fuzzer generate seed a_i
27: $\quad\quad tot\_cov \leftarrow tot\_cov + f_{cov}(a_i)$
28: $\quad\quad$ DROPTEST$(a_i, \#(a_i), r(tot_cov, a_i), f_{\text{cov}}(a_i), \mathcal{A}_c, \pi_c, \gamma)$
29: $\quad$ **return** tot_cov
30: **function** DROPTEST(a, #, r, f_{cov}, $\mathcal{A}$, π, γ)
31: $\quad \# \leftarrow \# + 1$; $r \leftarrow r + f_{\text{cov}}$
32: $\quad$ **if** $\# \geq \gamma$ and $r = 0$ **then**
33: $\quad\quad \mathcal{A} \leftarrow \mathcal{A} \setminus \{a\}$
34: $\quad$ **else**
35: $\quad\quad r \leftarrow 0$; $\# \leftarrow 0$
36: $\quad$ **return** #, r, $\mathcal{A}$, π