PORTRUSH: Detect Write Port Contention Side-Channel Vulnerabilities via Hardware Fuzzing

Peihong Lin, Pengfei Wang✉, Lei Zhou, Gen Zhang, Xu Zhou, Wei Xie, Zhiyuan Jiang, Kai Lu✉
National University of Defense Technology
{phlin22, pfwang, zhoulcs, zhanggen, zhouxu, xiewei, jzy, kailu}@nudt.edu.cn

Abstract*—CPU vulnerabilities pose ongoing security challenges in modern CPU architectures. Among the CPU vulnerabilities, write port contention—caused by multiple functional modules simultaneously competing for a limited number of shared write ports—remains insufficiently studied. In this paper, we study write port contention side-channel vulnerabilities in CPUs and propose PORTRUSH, a novel fuzzing framework designed to detect and validate such vulnerabilities at the register-transfer level (RTL). First, PORTRUSH constructs a Write Request Graph (WRG) to statically identify potential write port contention instances by modeling write paths and priority relationships among functional modules that target shared storage elements. Second, within the WRG, PORTRUSH implements a Hierarchical Aggregation and Decoding method to efficiently detect write port contention by monitoring relevant hardware signals across design hierarchies. Third, PORTRUSH employs a Contention-guided Hardware Fuzzing approach to trigger write port contention and automatically combine contention-triggered instruction sequences with transient execution attack patterns, enabling validation of write port contention side-channel vulnerabilities. We evaluate PORTRUSH on three RISC-V CPUs (BOOM, NutShell, and Rocket Core) and demonstrate its effectiveness in identifying and triggering write port contention. Furthermore, we validate that the discovered vulnerabilities can be exploited in realistic write port contention attack scenarios. Based on these vulnerabilities, we present two novel attack vectors: `Birgus-variant`, which exploits contention at the physical register file in the Reorder Buffer, and `MSHRush`, which leverages contention between the `Load/Store Unit` (`LSU`) and `Miss Status Handling Register` (`MSHR`) at the L1 data cache to induce secret-dependent execution delays. We also propose mitigation strategies for CPU developers to prevent such vulnerabilities.

I. INTRODUCTION

The increasing complexity of modern CPU architectures has led to the emergence of various micro-architectural vulnerabilities, including speculative execution attacks [14], [25], [27], [29] and privilege escalation exploits [6], [8], which pose serious threats to both hardware and software security. Among these, cache-based side-channel attacks have been the most widely studied and exploited in practice. Attacks such as FLUSH+RELOAD [21], PRIME+PROBE [31], and Evict+Time [32] leverage subtle differences in cache access times to extract sensitive information from victim processes. As a result, most existing detection and mitigation efforts, including Kernel Page-Table Isolation [20] and various cache partitioning and randomization techniques [18], [38], have focused primarily on defending against cache-based side channels.

In contrast, port contention side-channel attacks, especially those caused by write port contention, have received far less attention from the research community and industry. In modern CPUs, different functional modules (i.e., *write entities*) often share a limited number of write ports to access storage elements such as register files or caches. When multiple entities issue write requests in the same cycle that exceed the number of available write ports, contention occurs. To arbitrate access, most CPUs employ priority-based mechanisms, such as fixed-priority or first-come-first-served policies [1], [5], [43], which can cause lower-priority instructions to be delayed by higher-priority ones, resulting in observable timing differences.

Recent works [2], [12], [23] have shown that timing differences, when combined with transient or speculative execution, can form the basis of powerful side-channel attacks. Specifically, attackers can craft gadgets that deliberately trigger write port contention by manipulating control-flow or data dependencies, causing high-priority instructions to compete with low-priority victim instructions for write port access. If the victim's execution time varies depending on secret-dependent conditions (e.g., whether a specific secret bit matches an attacker-controlled probe), attackers can infer sensitive information based on measured timing differences. Write port contention that can be exploited in this manner is termed ***write port contention side-channel vulnerability***.

Despite their threat, existing vulnerability detection approaches, including both static analysis (e.g., formal verification [17], [22]) and dynamic testing (e.g., hardware fuzzing [13], [16], [24], [28], [35], [39]), do not specifically target write port contention side-channel vulnerabilities. As a result, hidden instances of write port contention may go undetected, and it remains unclear whether and how these cases can be exploited in real-world attacks.

In this paper, we present PORTRUSH, a novel hardware fuzzing framework designed to detect write port contention side-channel vulnerabilities. PORTRUSH combines four essential stages: (i) static identification of potential write port contention instances, (ii) dynamic monitoring of contention

✉ Corresponding author(s).

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.2026.240587
www.ndss-symposium.org

instances, (iii) triggering write port contention through a hardware fuzzing approach, and (iv) automated validation by combining write port contention with transient execution attack patterns to assess the feasibility of real side-channel attacks. Moreover, based on the detection and validation of these vulnerabilities, we propose mitigation strategies for CPU designers.

However, realizing this framework presents several challenges. **(i) Statically identifying potential write port contention is challenging.** Modern RISC-V CPUs [10], [15], [43] feature deeply nested, modular structures with complex arbitration logic, making it difficult to extract all write paths and priority relationships among write entities for comprehensive static analysis. **(ii) Real-time monitoring of write port contention is non-trivial.** Out-of-order execution in modern CPUs [4], [15], [43] decouples programmed instruction order, making it infeasible to monitor which entities issue write requests (i.e., detect *write request behaviors*) based on instruction order. Thus, PORTRUSH must monitor internal signals at runtime, but the large signal space makes it challenging to efficiently identify relevant signals and design scalable, low-overhead detection mechanisms. **(iii) Dynamically triggering and validating write port contention is challenging.** Current dynamic validation approaches, such as hardware fuzzing [23], [24], [39], are not specifically designed to target write port contention. These methods lack contention-aware feedback and do not guide the generation of simultaneous write requests from multiple entities. Moreover, they do not support the automatic combination of triggered contention instances with transient execution attack patterns, thus failing to validate the exploitability of port contention through real side-channel attacks.

PORTRUSH solves the aforementioned challenges by following three approaches. (i) We propose a **Write Request Graph (WRG)** abstraction that models arbitration logic and priority relations in RTL, enabling automated extraction of all write paths and their associated priorities, thereby identifying potential write port contention (§IV-B). (ii) We design a **Hierarchical Aggregation and Decoding** mechanism that collects relevant signals across module boundaries and efficiently reconstructs write request behaviors and contention instances in real time (§IV-C). Write port contention is detected when the number of write requests issued by write entities exceeds the number of available write ports associated with the target storage element. (iii) We develop a **Contention-guided Hardware Fuzzing** approach that automatically triggers write port contention by maximizing write request coverage, actively generating simultaneous requests from multiple entities, and seamlessly combining these with transient execution attack patterns to validate write port contention side-channel attacks (§IV-D).

In summary, we make the following contributions:

- We conduct the study of write port contention side-channel vulnerabilities in CPU microarchitectures. Our work covers static identification, monitoring, triggering, and validation of these vulnerabilities, providing comprehensive understanding of their root causes and impact.
- We design and implement PORTRUSH, a novel fuzzing framework to detect write port contention side-channel vulnerabilities. PORTRUSH is evaluated on three real-world RISC-V CPUs (BOOM [43], NutShell [15], and Rocket Core [5]), identifying 177 distinct instances of potential write port contention and successfully triggering 35 of them. Among these, three instances are verified to be exploitable in conjunction with transient or speculative execution as side-channel attack vectors.
- We validate the feasibility of write port contention side-channel attacks. By combining contention-triggering instruction sequences with transient or speculative execution patterns, we discover three side-channel attack vectors, including two novel variants (`MSHRush` on BOOM and `Birgus-variant` on NutShell) and the known `Spectre-STC` attack on BOOM. Compared to traditional cache-based attacks, write port contention side-channel attacks enable information leakage even in processors with secure or partitioned caches.

II. BACKGROUND

Hardware Fuzzing. Hardware fuzzing is an automated dynamic testing technique designed to explore the functional boundaries of hardware designs and identify potential vulnerabilities by iteratively generating and executing test cases [13], [16], [24], [28], [35], [39]. Typically, the hardware fuzzer generates a set of test cases, known as seeds, which are executable programs composed of instruction sequences. These seeds are then executed on the DUT using hardware simulation tools, such as open-source Verilator [34] or commercial simulators such as Synopsys VCS [9]. During execution, coverage feedback is collected in one of two ways: actively by instrumenting the DUT with dedicated logic to monitor specific behaviors or passively via built-in metrics provided by the simulation tools [24], [28]. Common coverage metrics include the finite state transitions of control registers (i.e., register coverage) [13], [24], multiplexer pattern coverage [28], or accessed register states [35]. These coverage insights guide further test case mutations, such as bit flips or instruction substitutions, strategically driving the testing towards improved hardware design coverage.

To further enhance vulnerability detection, differential testing and assertion-based verification are often integrated into hardware fuzzing workflows. Differential testing identifies vulnerabilities by comparing outputs or states (e.g., register values, memory consistency, exception handling) of the DUT against those of a known-correct ***golden*** reference model [23], [24], [39]. Assertion-based verification encodes expected hardware properties as assertions within the DUT, and violations of these assertions indicate potential vulnerabilities [13], [30]. However, despite the effectiveness of these techniques in detecting hardware vulnerabilities, systematic analysis of hardware fuzzing methods targeting write port contention remains limited. On one hand, current hardware fuzzing techniques lack appropriate methods to monitor write

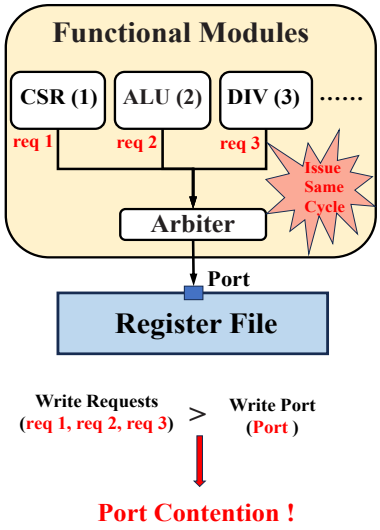

Fig. 1: An illustration of the write port contention

```
attacker:
  rdcycle x10
  call victim
  rdcycle x11
  sub x11, x11, x10
  /* Use timing difference in x11 to infer secret bit */
victim:
  ...
  /* low-priority div is data-dependent on beq branch */
  div x18, x15, x14
  ...
  /* Branch predictor misprediction setup */
  beq x16, x18, L1 /* Branch mispredicted as not taken */
transient:
  /* Begin transient window */
  la x19, secret /* Load address of secret */
  ld x20, 0(x19) /* Load secret value */
  andi x20, x20, 0x1 /* Extract secret bit */
  beqz x20, L1 /* If secret bit = 0, skip alu storm */
alu_storm:
  /* multiple alu instructions within 1 cycle */
  add x24, x15, x14
  ...
L1:
  ...
  ret
```

Listing 1: An example of combining a speculative attack with write port contention to leak secret information

request behaviors, which would provide useful feedback for optimizing mutation strategies. On the other hand, existing assertion-based verification approaches are not suited to detect write port contention side-channel vulnerabilities, as these are essentially non-functional, resource-related concerns rather than functional correctness errors.

Timing Side-channel Attacks. Time side-channel attacks are a powerful form of information leakage that exploits subtle timing differences observed during a system's execution to uncover secret or sensitive data. Typically, these attacks analyze measurable differences in execution time stemming from the internal microarchitectural behavior of CPUs. One prominent category of timing side-channel exploits is cache-based attacks, such as PRIME+PROBE [31], Evict+Time [32], and FLUSH+RELOAD. Classical examples, such as Spectre [27] and Meltdown [29], exploit speculative execution by manipulating cache states to leak sensitive information. More recent studies have revealed timing side channels based on port contention within CPU functional modules. For example, the attack named *SMoTherSpectre* [12] leverages contention on CPU execution ports. By strategically causing a conflict at the hardware execution-resource level during speculative execution, adversaries measure timing differences induced by resource contention to reconstruct sensitive information.

III. Motivation

1) Causes of Write Port Contention: Write port contention arises in modern CPUs when, within a single clock cycle, the number of write requests issued by functional modules (e.g., `CSR`, `ALU`, and `DIV`) exceeds the number of available shared write ports to storage elements such as register files, memory, or caches. This situation is common in practical designs [1], [5], [15], [43], where cost and area constraints often limit the number of write ports. To resolve such contention, designers implement arbitration mechanisms, such as fixed-priority or first-come, first-served (FCFS) schemes, that determine which request is granted access in each cycle. For example, in a

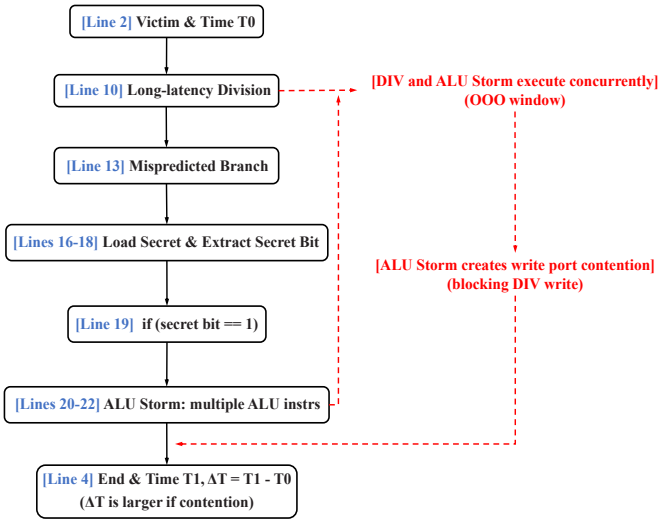

Fig. 2: Motivation example

fixed-priority scheme, each module is assigned a predefined priority, and the highest-priority request is served first, while lower-priority requests are deferred to subsequent cycles.

However, this arbitration introduces timing discrepancies: lower-priority or later-arriving write-eliciting instructions may experience observable delays when they contend for the same port as higher-priority instructions. Crucially, these timing delays are not just a performance artifact—they can be externally observed and exploited by attackers. By carefully orchestrating contention, an attacker can induce measurable timing differences in the execution of victim instructions, thereby constructing side-channel attacks that leak sensitive information via write port contention.

2) Security Risks of Write Port Contention: Write port contention introduces exploitable security risks when an attacker

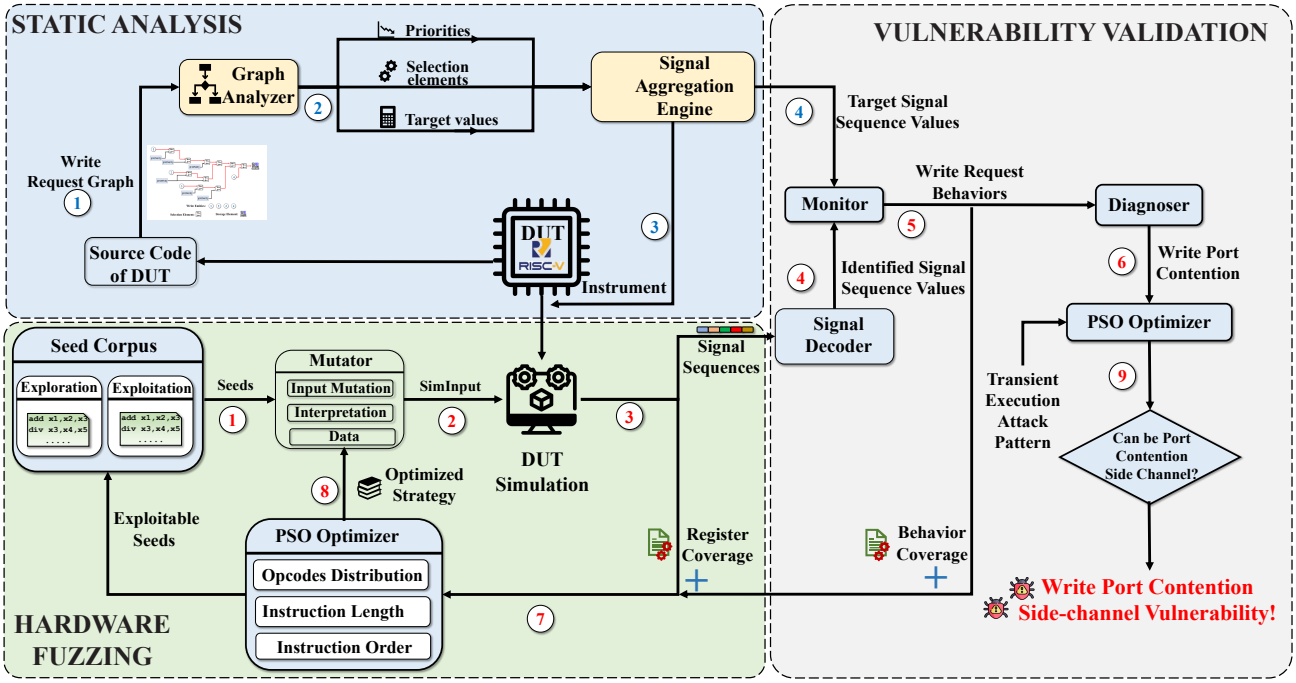

Fig. 3: The overview of PORTRUSH.

can deliberately delay lower-priority instructions through bursts of high-priority instructions, resulting in measurable timing differences. This timing side-channel becomes a practical vulnerability when combined with transient or speculative execution attack patterns, enabling attackers to leak sensitive information that would otherwise be protected by software-level checks.

In a typical attack scenario, the `victim` function contains a long-latency, low-priority division instruction `div` (Line 10) whose result determines the outcome of a conditional branch (Line 13). As shown in Listing 1 and Figure 2, the attacker first poisons the branch predictor to ensure the branch is mispredicted. When the attack is triggered, the CPU speculatively enters a transient window (Lines 14-23) before the true branch outcome is known. During this window, the `victim` function loads the secret (Lines 16-18) and checks whether a specific secret bit matches an attacker-controlled probe value (Line 19). If the condition is met, the code speculatively executes a burst of high-priority `alu` instructions, intentionally issuing multiple write requests to the register file and creating intense write port contention (Lines 20-23). The `div` instruction and the speculative `alu` instructions can overlap in execution due to the out-of-order scheduling capabilities of modern CPUs. Specifically, while `div` is still pending and its result is needed for the branch in Line 13, the processor may speculatively execute instructions in the transient window, such as the *alu_storm* containing multiple `alu` instructions, before actual branch resolution. This behavior allows speculative `alu` instructions to contend for the write port at the same time as the `div` is completing. As a result, significant write port contention occurs only when the secret bit matches the probe value. Although speculative instructions are eventually squashed after branch misprediction is resolved, the contention they cause delays the write-back of the division result. This microarchitectural delay persists and can be detected by the attacker through precise timing measurements of the `victim` function's execution. This case demonstrates that **write port contention, when combined with transient execution attack patterns, creates a novel and exploitable side-channel vulnerability.**

IV. DESIGN OF PORTRUSH

A. A High-Level Overview

PORTRUSH detects write port contention side-channel vulnerabilities by static identification, real-time monitoring, and dynamic fuzzing combined with side-channel attack validation. Specifically, PORTRUSH first extracts the **Write Request Graph** to model arbitration and priority relations and identify potential write port contention instances (§IV-B). It then employs a lightweight **Hierarchical Aggregation and Decoding** mechanism to monitor real-time write request behaviors and write port contention instances (§IV-C). Third, a **Contention-guided Hardware Fuzzing** approach is introduced to automatically trigger write port contention (§IV-D). Finally, write port contention side-channel attacks are constructed by combining write port contention with transient execution. Contention instances that can be exploited in this manner are validated as *write port contention side-channel vulnerabilities*.

As depicted in Figure 3, PORTRUSH comprises three integrated modules: static analysis, hardware fuzzing, and vulnerability validation. Blue numbers indicate steps in the static analysis phase, while red numbers indicate steps in the hardware fuzzing phase.

Static Analysis and Monitoring Instrumentation. The *Graph Analyzer* extracts the WRG, calculates priorities of write entities, and identifies selection elements and their target values required for successful write requests. The *Signal Aggregation Engine* then defines instrumentation rules to aggregate selection signals, mapping each write request to a unique signal sequence (i.e., target signal sequence values). The map containing *target signal sequence values* is stored locally and retrieved during fuzzing to reconstruct and identify write request behaviors.

Hardware Fuzzing. The *Seed Corpus* of fuzzing is organized into two pools: the exploration pool and the exploitation pool. The exploration pool leverages a dictionary derived from the RISC-V ISA [3], mapping valid opcodes to their operand formats and enabling the generation of diverse instruction templates. During exploration, the *Mutator* selects opcodes and fills in operand fields (e.g., `rd`, `rs`, `imm`), without strictly constraining memory or control flow targets, allowing PORTRUSH to explore a wide range of architectural exceptions (e.g., misaligned accesses and out-of-bounds reads and writes). During exploitation, the *Mutator* selects seeds from the exploitation pool to construct new instruction sequences that can trigger write port contention.

Vulnerability Validation and Feedback. During DUT simulation, the signals of selection elements are aggregated and encoded into signal sequences to the *Signal Decoder*. The *Signal Decoder* reconstructs write request behaviors from the signal sequences, and the *Diagnoser* identifies contention instances. The exploitability is assessed by correlating detected contention with transient execution attack patterns, and continuously feeding coverage and contention information back to the PSO optimizer. This feedback loop allows the optimizer to adapt its fuzzing strategies, such as opcode selection, instruction sequence length, and instruction ordering, thereby enabling the automated discovery of write port contention side-channel vulnerabilities.

B. Write Request Static Profiling

We first construct the WRG to model the arbitration and priority relationships of write entities and identify potential write port contention.

1) Write Request Graph Construction: Given a CPU RTL design, PORTRUSH abstracts the WRG from its RTL code. In the WRG, each node represents a hardware element involved in the generation or transmission of write requests, such as functional modules (e.g., `ALU`), selection elements (e.g., multiplexers, arbiters), and storage elements (e.g., registers, caches, and memory). Each edge denotes a possible data or control dependency between these elements, indicating how a write request traverses from its source to its destination. Formally, we define the WRG as follows:

Definition 1 Given the RTL of DUT D, the WRG is defined as $G_w(D) = (V, E)$, where V is the set of elements in D, including write entities, selection elements, and storage elements. E is the set of directed edges (v_i, v_j), where each edge indicates that a write request can propagate from v_i to v_j in the RTL. Each edge may be annotated with its corresponding arbitration or selection condition.

The construction of the WRG includes three steps:

(1) Extracting the graph structure from the DUT. We first extract the necessary structural information directly from the DUT. All elements in the RTL design involved in write operations are identified as nodes in the graph, including 1) functional modules that generate write requests, 2) elements such as wires, registers, ports, and multiplexers that propagate write requests, and 3) storage elements with write ports that receive write requests. Each directed edge in the graph represents a possible data flow or control dependency between these elements.

(2) Identification and simplification of write paths. Starting from each write entity in the DUT, we perform a depth-first search to enumerate all possible paths leading to the storage elements. During this process, only elements directly involved in arbitration, such as selection elements with multiple inputs and a single output (e.g., multiplexers, priority arbiters), are retained in the graph. This results in a simplified representation that highlights the essential arbitration logic, filtering out micro-events or transitions that do not impact write port contention. Formally, a path $path = < (e, s_1), (s_1, s_2), \ldots, (s_n, se) >$ denotes the sequence of nodes and edges from a write entity e to a storage element se, with intermediate nodes s_i representing only selection or arbitration logic.

(3) Normalization of multi-input selection elements. In RTL designs, write arbitration is typically implemented using selection elements such as multi-input priority arbiters, multiplexers, or priority encoders. To facilitate priority extraction and analysis, each multi-input selection element is normalized by decomposing it into a cascade of two-input selection elements. Given a selection element with inputs $I_1, I_2, \ldots, I_k$ $(k \geq 3)$ and a priority order $I_1 > I_2 > \ldots > I_k$, we construct a cascade of $k-1$ two-input selection elements. At each stage i, the selection element receives I_i and the previous stage's output O_{i-1}, producing output O_i, thereby ensuring that higher-priority inputs are selected first. The selection logic is defined as follows:

- The first stage output is $O_1 = I_1$.
- For stage i $(2 \leq i \leq k)$, the output is:

$$O_i = \begin{cases} I_i, & \text{if } I_i = \text{True} \\ O_{i-1}, & \text{otherwise} \end{cases} \tag{1}$$

After this transformation, the path of a write request from a write entity to the write port can be represented as: $WP =< (e, s_1''), (s_1'', s_2''), \ldots, (s_{t-1}'', s_t''), (s_t'', se) >$, where e is the write entity, se is the storage element, and each s_i'' denotes a two-input selection element. The union of all such nodes and edges forms the WRG, as illustrated in Figure 4.

2) Priority Calculation: In the WRG, as outlined in Formula 1, a write entity attains higher priority if its write request is arbitrated by a selection element closer to the write port. Therefore, we first assign priorities to all selection elements in the WRG based on their shortest path distances to the write

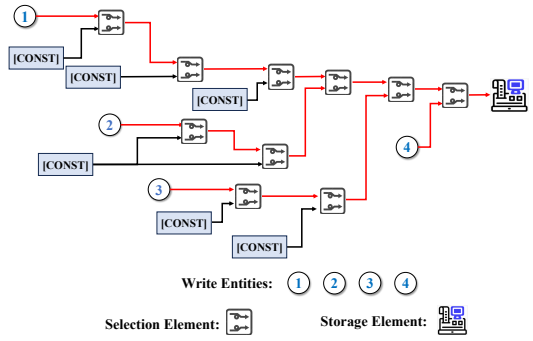

Fig. 4: An illustration of the Write Request Graph

port, and then use these priorities to determine the priorities of the write entities.

By treating each edge traversal as a unit distance, the priority of a selection element s is defined as:

$$priority(s) = \max D - d(s,\ se) \tag{2}$$

Where $d(s,\ se)$ is the shortest path length from the selection element s to the storage element se, and $\max D$ is the maximum such distance among all selection elements. Thus, selection elements closer to the write port are assigned higher priorities.

Subsequently, the priority of a write entity e is the average priority of all selection elements along its write path:

$$priority(e) = \frac{\sum_{s_i \in S} priority(s_i)}{|S|} \tag{3}$$

Where S denotes the set of all selection elements along e's write path, and $|S|$ denotes the number of selection elements in S. This approach ensures that write entities traversing higher-priority selection elements are assigned higher priorities.

3) Identification of Potential Write Port Contention: For a given storage element se, if the number of write entities N_e simultaneously issuing write requests to the storage element exceeds the available number of write ports N_{port}, it indicates the presence of potential write port contention instances. The number of such potential contention instances is calculated as:

$$C\_Num(se) = \sum_{k=N_{port}+1}^{N_e} \binom{N_e}{k} \tag{4}$$

Where $C\_Num(se)$ denotes the total number of potential write port contention instances that may occur at se. According to Formula 4, when the number of entities attempting to issue write requests exceeds the number of available write ports, the number of potential contention instances is determined by the combinations of these write entities.

C. Monitoring Write Port Contention

Given the WRG, PORTRUSH employs a **Hierarchical Aggregation and Decoding** method to monitor write port contention.The key idea is to monitor the signals of selection

Algorithm 1: Monitoring write port contention

Input: WRG of the DUT
Output: Potential Write Port Contention Instances

1 $\mathcal{M} \leftarrow$ `Prioritize_Modules`(DUT)
2 **foreach** $D \in \mathcal{M}$ **do**
3 $seq_D, map_1 \leftarrow$ `Collect_Signals`(D)
4 $\mathcal{E} \leftarrow$ `Prioritize_Entities`(D)
5 $\mathcal{U} \leftarrow$ `Get_SubModules`(D), $seq'_D = seq_D$
6 **foreach** $D_i \in \mathcal{U}$ **do**
7 $seq'_i \leftarrow$ `Collect_SubSequences` (D_i)
8 $seq'_D, map_2 \leftarrow Concatenate(seq'_D, seq'_i)$
9 **foreach** $e_i \in \mathcal{E}$ **do**
10 $Tseq_i \leftarrow$ `Construct_Target_Seq`(map_1, map_2)
11 `Update`$((e_i, Tseq_i))$
12 $seq_t \leftarrow seq_{D_t}$ // D_t is the top-level module
13 **foreach** $D \in \mathcal{M}$ **do**
14 $seq_D \leftarrow$ `Extract_SubSequence`(seq_t), $W_D \leftarrow \emptyset$
15 **foreach** $e_i \in \mathcal{E}$ **do**
16 **if** `Match_Target_Values`$(Tseq_i, seq_D)$ **then**
17 $W_D \leftarrow W_D \cup e_i$ // Record write request
18 $seq_D \leftarrow seq_D \setminus S_i$ // Remove bits
19 **if** $|W_D| > N_{port}$ **then**
20 `Identify_Contention` (W_D)

elements across RTL modules and determine whether the number of write requests simultaneously issued by write entities exceeds the number of write ports of the target storage element. The main challenge lies in efficiently collecting and integrating signals from nested modules into the top-level module and accurately identifying write request behaviors from the aggregated signal sequence. PORTRUSH addresses these challenges in three steps: 1) Statically identify the expected signal values of all selection elements along each write entity's write path when its request is accepted by the write port. 2) Collect and hierarchically aggregate these signals from all relevant selection elements into the top-level RTL module. 3) Design a **Signal Decoder** to interpret the aggregated signal sequence, thereby identifying write request behaviors and monitoring write port contention. The complete detection algorithm is presented in Algorithm 1.

1) Static Analysis for Instrumentation: Given the WRG of the DUT, PORTRUSH first statically prioritizes RTL modules according to their hierarchical dependencies, reflecting the layered structure typical in RISC-V CPUs, where modules are instantiated within one another (Line 1). This ensures that lower-level modules are processed before their parent modules.

For each RTL module D in the prioritized list $\mathcal{M}$, PORTRUSH identifies all relevant selection elements in the WRG and their associated signals. Specifically, it determines the necessary signal conditions, referred to as *target values*, that must be satisfied for each write entity to issue a valid write request. This results in a signal sequence seq_D for each module and a mapping map_1 that associates each write entity e_i with its dependent selection signals and their required values (Line 3):

$$map1 = \{(e_i, sm_i) | e_i \in \mathcal{E}, sm_i = (s_j, v_j) | s_j \in S_i, v_j \in V_i\} \tag{5}$$

Where $\mathcal{E}$ is the set of write entities in module D, S_i is the set of selection elements along the write path of e_i, and V_i is the set of target values for these signals. This mapping enables precise identification of the conditions under which write requests are triggered, forming the basis for subsequent decoding and contention detection.

Subsequently, PORTRUSH enumerates and prioritizes the write entities in D according to their write priorities, ensuring that entity-specific triggering conditions are accurately captured (Line 4). The set of all sub-modules instantiated within D is denoted as $\mathcal{U}$, and the local signal sequence seq_D is initialized for hierarchical aggregation (Line 5). Signal sequences are aggregated in a bottom-up manner: for each sub-module D_i in $\mathcal{U}$, its signal sequence seq'_i is concatenated with seq'_D to form the new aggregated sequence (Lines 6-8). During each concatenation, a global offset mapping map_2 is maintained, recording the position of each monitored signal within seq'_D. The offset for each signal is computed as:

$$offset_i = \sum_{k=1}^{i-1} |seq_k| + i - 1 \tag{6}$$

$$map_2 = \{(s_i, offset_i) |\ s_i \in S,\ offset_i \in \mathbb{N}, 1 \le i \le n\} \tag{7}$$

Where $offset_i$ is the offset of element i in seq'_D, seq_k is the signal sequence of the k-th sub-instance, and S is the set of elements.

Upon completion of hierarchical aggregation, for each write entity e_i in $\mathcal{E}$, PORTRUSH constructs a global target signal mapping $Tseq_i$ that associates each relevant selection signal with its required value at the corresponding global offset (Lines 9-11):

$$Tseq_i = \{(offset_j, v_j) \mid (s_j, v_j) \in map_1(e_i),\ offset_j = map_2(s_j)\} \tag{8}$$

Where $map_1(e_i)$ provides the monitored signal-value pairs for e_i, and $map_2(s_j)$ gives the global offset of s_j in the concatenated sequence. $Tseq_i$ thus specifies the target value assignment for each relevant signal in e_i.

After all modules are processed, the aggregated sequence seq_t for the top-level module D_t is constructed, representing the unified runtime state of all monitored signals (Line 12).

2) Decoding Aggregated Sequence and Monitoring Write Port Contention: With the aggregated signal sequence, PORTRUSH designs a **Signal Decoder** to identify write request behaviors and monitor write port contention as follows:

(1) **Signal subsequence extraction.** For each module D, the decoder uses map_2 to extract the relevant signal subsequence, ensuring accurate tracking of signal paths for all write entities (Line 14).

(2) **Decoding and comparison.** Write entities are processed in priority order (Line 15). For each entity e_i, the decoder retrieves $Tseq_i$ and compares the target values against the corresponding bits in seq_D (Line 16). If all values match, e_i is identified as issuing a write request in the current cycle. Moreover, once a write entity is matched, the associated signal bits (e.g., s_i at $offset_i$ in map_2) are excluded from further comparisons for the remaining lower-priority write entities (Lines 17-18). This ensures that signal bits already claimed by higher-priority entities are not incorrectly attributed to others, reflecting the arbitration mechanism in RTL designs.

(3) **Contention monitoring.** The decoder records each write request. If the number of write entities issuing write requests to the same storage element exceeds the number of available write ports in the same cycle, write port contention is triggered (Lines 19-20).

Steps (1)—(3) are repeated for all modules, ensuring that all write request behaviors and potential contentions are accurately identified.

D. Hardware Fuzzing for Triggering Write Port Contention

We propose an efficient hardware fuzzing framework for triggering write port contention, driven by a PSO optimizer and proceeding in two phases. We first formalize seed generation and PSO optimization. Then, we optimize mutation strategies by tuning mutation parameters to improve register coverage and write request behavior coverage (**Phase 1**). Finally, we minimize and reorder instruction sequences to trigger write port contention efficiently (**Phase 2**).

1) Formalization of Seed Generation and PSO Optimization: Let $\mathcal{X}$ denote the instruction space, the instruction set is $I = \{i_k \mid k \in [1, N_{op}]\}$, where each instruction i_k is defined by a tuple (op_k, D_k, w_k), representing the opcode, operand domain, and weight, respectively. The probability of generating opcode op_k is:

$$P(op = op_k) = \frac{w_k}{\sum_{m=1}^{N_{op}} w_m} \tag{9}$$

Where N_{op} is the number of opcode types in the RISC-V architecture. The mutator is modeled as a stochastic process $M : \Theta \to \mathcal{X}$, with parameter space $\Theta = [l_{\min}, l_{\max}] \times \mathbb{R}_+^{N_{op}}$, where:

- $N_{\text{inst}} \in [l_{\min}, l_{\max}]$ is the instruction sequence length, uniformly sampled from the interval $[l_{\min}, l_{\max}]$.
- $\vec{\theta} = [w_1, \ldots, w_{N_{op}}] \in \mathbb{R}_+^{N_{op}}$ is the opcode weight vector, controlling $P(op = op_k)$.
- The output space is $\mathcal{X} = \bigcup_{n=l_{\min}}^{l_{\max}} I^n$, representing the set of all instruction sequences with length in $[l_{\min}, l_{\max}]$.

Based on this formalization, PORTRUSH samples opcode sequences $OP = [op_1, op_2, \ldots, op_n]$ according to $\vec{\theta}$, and randomly generates operands $data_k$ within each D_k. To emulate hardware timing, interrupts $\delta_t \sim \mathcal{N}(\mu_t, \sigma_t^2)$ are randomly inserted between cycles.

Mutation optimization with PSO. While the PSO optimizer adopts phase-specific strategies for distinct mutation operators, its fundamental objective is to guide mutation strategies toward global optima—specifically, configurations that maximize the triggering and exploitation of write port contention.

We choose PSO for its superior search capability in high-dimensional continuous parameter spaces, which aligns with our need to simultaneously optimize multiple mutation parameters (e.g., opcode weights, sequence lengths, and interrupt timings). PSO efficiently converges toward global optima

through collaborative exploration and exploitation by a particle swarm. Each particle represents a candidate configuration and updates its search trajectory by tracking personal best and global best positions. This mechanism enables PSO to balance exploration (discovering diverse write request patterns) and exploitation (reinforcing effective configurations) in Phase 1, while fine-tuning sequence minimization parameters in Phase 2. Moreover, PSO's stochasticity and adaptivity make it robust to noise, which is critical for non-deterministic behavior in hardware fuzzing. In contrast to gradient-based methods or MAB models [40], [41] commonly used in fuzzing, PSO does not require differentiability of the objective function, making it suitable for our coverage-based discrete feedback metrics.

As detailed in Algorithm 2, a swarm of K particles is initialized, with each particle $p_i = (x_i, v_i)$ encoding a candidate mutation strategy x_i and its corresponding velocity v_i. Here, $x_i \in \mathbb{R}^d$ represents the particle's position in the parameter space, where d is the dimensionality of the mutation strategy (e.g., opcode weights $\vec{\theta}$ and sequence length bounds). The velocity $v_i \in \mathbb{R}^d$ determines the direction and magnitude of parameter updates in subsequent iterations. Particle initialization is randomized to ensure sufficient diversity in the search space (Line 1). During each iteration, the mutator utilizes the parameters of each particle to generate a batch of seeds $\mathcal{X}_i$ for RTL simulation (Line 5). To enhance optimization efficacy, distinct reward functions are designed for each fuzzing phase. Following simulation, PORTRUSH evaluates the reward associated with each particle (Line 6). Particles update their personal bests when improvements are observed, and the global optimum is concurrently tracked (Lines 7–9). Subsequently, standard PSO update equations are applied, enabling adaptive adjustment of both positions and velocities in relation to individual and global bests (Lines 10–12). The settings of parameters in Line 11 follow established PSO practice to balance exploration and exploitation: population size population size $K \in [30, 50]$, iterations $T \in [100, 200]$, inertia weight $\omega \in [0.4, 0.9]$ (gradually decreasing), acceleration coefficients $c_1, c_2 \in [1.5, 2.0]$, and random factors $r_1, r_2 \sim U(0, 1)$. This iterative process proceeds for a predetermined number of rounds, progressively refining mutation parameters to maximize both the coverage and exploitability of write port contention events as potential side-channel attack vectors.

2) Phase 1: The goal of PORTRUSH in Phase 1 is to optimize the instruction sequence length, opcode types, and opcode selection probabilities to maximize the register coverage and write request behavior coverage. Therefore, we use $x_i = \langle N_{inst}, \vec{\theta}, \mu_t, \sigma_t \rangle$ to encode the mutator parameters and timing variations, while v_i is the rate and direction of change for each parameter. The reward function in Phase 1 is designed based on coverage feedback collected during simulation and used to compute a reward for each particle (Line 6):

$$R(x_i) = \alpha \underbrace{\frac{|S_{new}|}{|S_{total}|}}_{\text{Coverage}} + \beta \underbrace{\log\left(1 + \sum_{e \in \mathcal{E}} \mathbb{I}_{[e \notin H]}\right)}_{\text{Novelty}} - \gamma \underbrace{\frac{1}{2|\mathcal{E}|^2 \bar{e}} \sum_{i=1}^{|\mathcal{E}|} \sum_{j=1}^{|\mathcal{E}|} |e_i - e_j|}_{\text{Gini penalty}} \quad (10)$$

Algorithm 2: Optimizing Fuzzing Mutation with PSO

Input: Number of particles K, maximum iterations T, inertia coefficient ω
Output: Optimal solution x^*

1 Initialize particle swarm p_i with random parameters
2 Set iteration counter $t \leftarrow 0$
3 **while** $t < T$ **do**
4 **foreach** *particle* p_i **do**
5 Generate seeds $\mathcal{X}_i = \text{Mutator}(p_i)$ for simulation
6 Calculate reward $R(x_i)$
7 **if** $R(x_i) > R(b_i)$ **then**
8 Update particle's personal best position $b_i \leftarrow x_i$
9 Identify global best position $g = \arg\max_i R(b_i)$
10 **foreach** *particle* p_i **do**
11 Update velocity: $v_i \leftarrow \omega v_i + c_1 r_1 (b_i - x_i) + c_2 r_2 (g - x_i)$
12 Update position: $x_i \leftarrow x_i + v_i$
13 Increment iteration counter $t \leftarrow t + 1$
14 **return** global best particle $x^* = g$

Where α, β, and γ are weights for register coverage, novelty of write request behaviors, and Gini penalty of write request behaviors, respectively. To simplify computation, we set the values of α, β, and γ according to different DUT to normalize their corresponding three terms, ensuring all values fall within $[0, 1]$. The coverage term $\frac{|S_{new}|}{|S_{total}|}$ measures the proportion of new register coverage relative to total register coverage, reflecting the effectiveness of exploring new areas. The novelty term rewards the coverage of previously unseen write request behaviors (i.e., *write request behavior coverage*), with the logarithmic function scaling the reward as more novel behaviors are discovered. In this context, $\mathcal{E} = \{e_1, \ldots, e_m\}$ denotes the set of all write entities, and H is the subset that has already generated write requests. The indicator function is defined as:

$$\mathbb{I}_{[e \notin H]} = \begin{cases} 0 & \text{if } e \text{ } is \text{ } in \text{ } H \\ 1 & \text{otherwise} \end{cases} \quad (11)$$

The Gini penalty term promotes balanced exploration among all write entities by discouraging uneven distribution of write requests. Given this encoding and reward function, the PSO optimizer iteratively updates $\langle N_{inst}, \vec{\theta}, \mu_t, \sigma_t \rangle$ to maximize both coverage metrics throughout Phase 1.

3) Phase 2: In Phase 2, seeds from the exploitation pool are first minimized to retain only instructions essential for triggering the observed write request behaviors. Subsequently, the instructions within each minimized seed are reordered to maximize the likelihood that write requests from distinct entities coincide, thereby triggering write port contention.

Seed minimization is performed via an iterative binary search process. Starting from the original instruction sequence I_N, the sequence is recursively divided into two halves, $I_{[0,\ \frac{N}{2}-1]}$ and $I_{[\frac{N}{2},\ N-1]}$, and each half is simulated to verify whether the targeted write request behaviors are preserved. If either half maintains the required coverage, minimization continues recursively on that half. Otherwise, the current sequence is identified as minimal with respect to the desired behaviors. All minimal sequences are subsequently grouped by

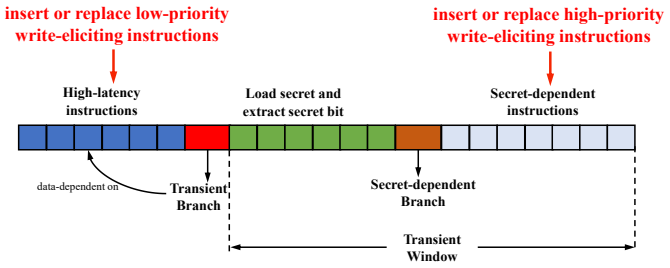

Fig. 5: Instruction sequence construction to validate write port contention vulnerabilities in the side-channel attack

write entity, ensuring that each group targets the same storage element.

Upon obtaining minimal instruction sequences, those within the same group are concatenated, and their register, memory, and address usage is optimized to minimize dependencies that could prevent simultaneous write requests. The concatenated instruction sequence is then subjected to a PSO-based permutation search, where each particle encodes a permutation $x_i = [k_0, k_1, k_2, k_3, \ldots, k_{M-1}]$, with k_j denoting the position of the $j-th$ instruction in the concatenated sequence of length M. The reward function is defined as:

$$R(x_i) = - \min_{e_1, e_2 \in \mathcal{E},\ e_1 \neq e_2} \left| \text{cycle}_{e_1} - \text{cycle}_{e_2} \right| \qquad (12)$$

Where $\mathcal{E}$ is the set of write entities in the current sequence, and $cycle_e$ denotes the clock cycle at which entity e issues a write request. This formulation promotes instruction orderings that minimize the interval between write requests from different entities, thereby enhancing the probability of triggering write port contention. The PSO optimizer then efficiently explores the permutation space based on this encoding and reward function.

E. Write Port Contention Side-channel Validation

To validate write port contention side channels in realistic attack scenarios, we combine contention-triggering instruction sequences with transient execution attack patterns. Transient execution, such as that enabled by speculative out-of-order execution, creates a transient window. During this window, instructions are executed speculatively. Although their architectural effects are eventually squashed, their microarchitectural side effects, such as resource contention, remain observable.

As shown in Figure 5, we leverage this property by carefully arranging the instruction sequence. **Low-priority write-eliciting instructions are placed within hard-to-resolve (e.g., high-latency) instructions** immediately before a subsequent transient branch. Meanwhile, **high-priority write-eliciting instructions are inserted into secret-dependent instructions** inside the transient window (i.e., after the transient and secret-dependent branches, in the speculative path). This placement ensures that when the CPU speculatively executes the transient window, high-priority instructions contend for write ports with the preceding low-priority instructions. This contention occurs even though the speculative results will eventually be discarded. The induced contention manifests as measurable timing differences or microarchitectural state changes, forming the basis of the side-channel attack.

To automate the generation of such attack patterns, we use the PSO optimizer to jointly determine (i) how many high- and low-priority instructions to insert or replace, (ii) their exact positions within the instruction sequence, and (iii) whether the resulting sequence successfully exposes a write port contention side-channel attack. Each candidate sequence encodes specific insertion and replacement operations and is evaluated using the following three criteria.

Transient execution verification. We combine SpecDoctor [23] to monitor whether transient execution is successfully triggered. Specifically, we leverage lightweight logic added to the CPU's Reorder Buffer (RoB) to detect rollback events. Each rollback event is logged with relevant information, including the cause of rollback, the opcode of the triggering instruction, the program counters for both the transient and correct paths, and the duration of the transient window. This mechanism enables us to accurately monitor whether and when transient windows are opened during testing.

Write port contention monitoring. To determine whether the generated instruction sequence triggers write port contention between low- and high-priority instructions, PortRush dynamically collects signal sequences that reflect write port requests from instructions of different priorities. By interpreting these signal sequences, PortRush can identify cycles in which both high- and low-priority instructions issue write requests simultaneously, thereby confirming the occurrence of write port contention and its potential as a microarchitectural side channel.

Timing side-channel observation. To assess the presence and effectiveness of a side channel, we measure the execution time of the `victim` function under different candidate sequences and secret values. Specifically, we examine whether significant timing differences arise when low- or high-priority instructions are inserted or replaced, and whether flipping the secret value (e.g., from 1 to 0 or vice versa) leads to corresponding changes in execution time. If both conditions are satisfied, the observed timing variations indicate that a practical and exploitable side channel exists. All timing differences are recorded to support reward calculation.

Based on the above criteria, the PSO optimizer evaluates each candidate using the following reward function:

$$R(x_i) = \mathbb{I}_{\text{transient}(x_i)} + \mathbb{I}_{\text{contention}(x_i)} + \Delta t(x_i) \qquad (13)$$

Where $\mathbb{I}_{\text{transient}(x_i)}$ is an indicator function that equals 1 if transient execution is successfully triggered in candidate sequence x_i, and 0 otherwise. $\mathbb{I}_{\text{contention}(x_i)}$ indicates whether write port contention is induced, and $\Delta t(x_i)$ measures the timing difference observed in the `victim` function. Based on Formula 13, instruction sequences that keep the transient window open, trigger write port contention, and cause observable timing differences in the `victim` function receive higher rewards. The optimizer thus automatically and heuristically searches for the optimal number and placement of low- and

high-priority instructions, ultimately identifying instruction sequences that expose exploitable side channels.

V. Implementation

Static Analysis. To construct the WRG, extract the priorities of write entities, identify potential write port contention, and perform instrumentation for real-time monitoring of write port contention, we implemented a custom pass in the FIRRTL [11] compiler. First, we use `sbt` [7] to compile the Chisel source code into an intermediate FIRRTL representation for static analysis, and then further compile the FIRRTL representation into Verilog files. The implementation consists of two parts: (i) a module (800+ LoC in Scala) for RTL graph analysis, WRG construction, extraction of write entity priorities, and identification of potential write port contention; and (ii) a module (200+ LoC in Scala) for instrumenting wires and registers to monitor signals of selection elements within the WRG.

Hardware Fuzzing. We implemented a hardware fuzzing framework to generate instruction sequences that trigger write port contention. The framework is implemented in Python (600+ LoC) and includes: (i) a **Signal Decoder** to interpret aggregated signal sequences; and (ii) a **Diagnoser** to detect write port contention; and (iii) a **PSO Optimizer** to improve write request behavior coverage, trigger write port contention, and combine contention-triggered instruction sequences with transient execution attack patterns for vulnerability validation.

VI. Evaluation

To evaluate PortRush, we conducted extensive experiments to answer the following research questions:

- **RQ1:** Can PortRush detect write port contention?
- **RQ2:** How does PortRush perform in terms of coverage?
- **RQ3:** Can write port contention triggered by PortRush cause real security vulnerabilities?
- **RQ4:** How can write port contention be exploited to construct side-channel attacks?

A. Evaluation Setup

Benchmarks. Most commercial processors are protected intellectual property without publicly available source code. Therefore, we selected three large and widely used open-source RISC-V CPUs: Rocket Core [5], BOOM [43], and NutShell [15]. These RTL cores are popular in the research community and have been used in state-of-the-art studies [13], [24], [39], [42]. Among them, BOOM and NutShell are more complex than Rocket Core, featuring advanced microarchitectural capabilities such as superscalar and out-of-order execution.

Evaluation environment. All experiments were conducted using software-based simulation within a Docker container on a 64-bit Ubuntu system. The host machine was equipped with 80 CPU cores (Intel Xeon(R) Gold 6133 @ 2.50GHz). For each RTL design, we performed five independent fuzzing runs. After each run, we collected and analyzed the register coverage, write request behavior coverage, and detected write port contention instances to comprehensively evaluate the effectiveness of our approach.

TABLE I: Potential write port contention statically identified

Target	Storage Element	N_{port}	Write Entity	Cont-Inst
BOOM	BoomDataArray_array_0_0	1	LSU, BoomMSHRFile	1
BOOM	BoomDataArray_array_1_0	1	LSU, BoomMSHRFile	1
BOOM	BoomDataArray_array_2_0	1	LSU, BoomMSHRFile	1
BOOM	BoomDataArray_array_3_0	1	LSU, BoomMSHRFile	1
BOOM	BoomMSHRFile_lb	1	BoomMSHR, BoomMSHR_1	1
BOOM	L1MetadataArray_tag_array	1	BoomMSHRFile, BoomProbeUnit	2
BOOM	RegisterFileSynthesizable_1_regfile	2	Port 0: MemAddrCalcUnit, FPUUnit Port 1: CSRFile, ALUUnit, PipelinedMulUnit, DivUnit	1 11
BOOM	RegisterFileSynthesizable_regfile	2	Port 0: ALUUnit, PipelinedMulUnit, DivUnit Port 1: FPUUnit, FDivSqrtUnit	4 1
NutShell	Backend_ooo_T	2	CSR, MOU, LSU, ALU Multiplier, Divider, BRU	42
NutShell	MDU	1	Divider, Multiplier	1
NutShell	ROB_prf	2	CSR, MOU, LSU, ALU Multiplier, Divider, BRU	42
Core	FPU_regfile	1	CSRFile, FPUFMAPipe, FPUFMAPipe_1, MulAddRecFNPipe, MulAddRecFNPipe_1, DivSqrtRecFN_small, DivSqrtRecFN_small_1	57
Core	Rocket__T_815	1	MulDiv, CSR, FPU, ALU	11

B. Identifying and Triggering Write Port Contention (RQ1)

Detecting write port contention in our evaluation consists of two stages. First, PortRush performs static analysis to identify potential write port contention by pinpointing architectural scenarios where multiple instructions may simultaneously attempt to access shared write ports. Then, PortRush employs hardware fuzzing to automatically generate instruction sequences that trigger actual write port contention instances, thereby validating the existence and impact of such contention in the target microarchitecture.

1) Static Identification of Potential Write Port Contention: In the static analysis phase, PortRush identifies storage elements along with their associated write ports and write entities. If the number of write entities exceeds the number of write ports for a given storage element, that element is flagged as potentially susceptible to write port contention. The results of this static analysis on Rocket Core, BOOM, and NutShell are presented in Table I. The first column (**Target**) lists the evaluated CPUs, while the second (**Storage Element**) specifies the relevant storage elements. For instance, `BoomDataArray_array_0_0` refers to a sub-array within the BoomDataArray module. The third column (N_{port}) indicates the number of write ports associated with each storage element, and the fourth column (**Write Entity**) enumerates the functional modules capable of issuing write requests to that element. The fifth column (**Cont-Inst**) reports the number of distinct contention instances possible for each storage element's write ports. For example, BOOM's `RegisterFileSynthesizable_1_regfile` features two write ports. `Port0` is shared by `MemAddrCalcUnit` and `FPUUnit`, resulting in only one contention instance. In contrast, `Port1` is shared by `CSRFile`, `ALUUnit`, `PipelinedMulUnit`, and `DivUnit`, allowing for 11 distinct potential contention instances, arising from all possible pairs,

TABLE II: Write port contention triggered by fuzzing

Target	Storage Element	N_{port}	Write Entity	Cont-Insts
BOOM	BoomDataArray_array_0_0	1	LSU, BoomMSHRFile	1
BOOM	BoomDataArray_array_1_0	1	LSU, BoomMSHRFile	1
BOOM	BoomDataArray_array_2_0	1	LSU, BoomMSHRFile	1
BOOM	BoomDataArray_array_3_0	1	LSU, BoomMSHRFile	1
BOOM	BoomMSHRFile_lb	1	BoomMSHR, BoomMSHR_1	1
BOOM	L1MetadataArray_tag_array	1	BoomMSHRFile, BoomProbeUnit	2
BOOM	RegisterFileSynthesizable_1_regfile	2	Port 0: MemAddrCalcUnit, FPUUnit Port 1: CSRFile, ALUUnit, PipelinedMulUnit, DivUnit	1 7
BOOM	RegisterFileSynthesizable_regfile	2	Port 0: ALUUnit, PipelinedMulUnit, DivUnit Port 1: FPUUnit, FDivSqrtUnit	4 1
NutShell	Backend_ooo_T	2	CSR, MOU, LSU, ALU Multiplier, Divider, BRU	7
NutShell	MDU	1	Divider, Multiplier	1
NutShell	ROB_prf	2	CSR, MOU, LSU, ALU Multiplier, Divider BRU	7

TABLE III: Comparison with DiFuzzRTL on register coverage

Targets	PORTRUSH	DiFuzzRTL	p-value
Rocket Core	86,385 (-6.27%)	92,169	$1.02 * 10^{-4}$
BOOM	407,661 (-7.03%)	438,519	$3.16 * 10^{-4}$
Average	494,046 (-6.90%)	530,668	$2.09 * 10^{-4}$

TABLE IV: Comparison with DiFuzzRTL on WPC coverage

Targets	Potential Cont. Inst.	PORTRUSH Triggered	Coverage (%)	DiFuzzRTL Triggered	Coverage (%)
BOOM	24	20	83.3	5	20.8
NutShell	85	15	17.6	1	1.2
Total	109	35	32.1	6	5.5

triples, and quadruple combinations of write entities. The potential write port contention detected by PORTRUSH not only guides the subsequent hardware fuzzing process but also assists hardware designers and researchers in localizing and understanding the root causes of possible write port contention side-channel vulnerabilities.

As shown in Table I, there are 13 storage elements across BOOM, NutShell, and Rocket Core that are susceptible to write port contention, with a total of 177 distinct instances of potential port contention identified. Each instance can be triggered by carefully crafted instruction sequences, which may be exploited to construct side-channel attacks.

2) Triggering Write Port Contention through Fuzzing: The results of write port contention triggered by fuzzing are presented in Table II. As shown, a total of 11 storage elements across BOOM and NutShell exhibited write port contention, with 35 distinct contention instances successfully triggered. This number is noticeably lower than the total number of potential write port contention instances for two reasons. First, some potential write port contention instances require three or more write entities to simultaneously issue write requests. Although our hardware fuzzing approach leverages a PSO optimizer to optimize instruction ordering, such complex scenarios are difficult to trigger within the limited 24-hour testing window. Second, due to Rocket Core's in-order, single-issue architecture, the identified potential port contention instances cannot be activated in practice. Nevertheless, since some superscalar, out-of-order RISC-V CPUs may be developed based on Rocket Core, the potential write port contention detected in Rocket Core could propagate to advanced SoC designs, thereby introducing side-channel security risks.

C. Evaluation on Coverage (RQ2)

Beyond detecting write port contention, PORTRUSH preserves comprehensive design space exploration capabilities by leveraging the register coverage metric from DiFuzzRTL.

We adopt this coverage metric for two reasons. First, it captures PORTRUSH's ability to cover a broad range of distinct write paths. Triggering write port contention instances requires executing multiple such paths, and their number is directly correlated with coverage, making it a meaningful indicator of the fuzzer's ability to reach contention conditions. Second, PORTRUSH and DiFuzzRTL share methodological similarities, as both derive coverage from MUX signals for coverage analysis (DiFuzzRTL) or port contention monitoring (PORTRUSH). Comparing coverage with DiFuzzRTL thus demonstrates that PORTRUSH's port contention monitoring mechanism, coupled with the PSO optimizer, does not introduce significant performance overhead that would reduce coverage. To assess PORTRUSH's effectiveness in uncovering functional vulnerabilities, such as deviations between DUT execution and RISC-V ISA specifications, we directly compared PORTRUSH and DiFuzzRTL under identical register coverage conditions. As DiFuzzRTL does not support software simulation for NutShell, our evaluation focuses on Rocket Core and BOOM, as summarized in Table III.

Across five repeated fuzzing trials, PORTRUSH's average register coverage on Rocket Core was 6.27% lower than DiFuzzRTL, and 7.03% lower on BOOM. On average, PORTRUSH exhibited only a 6.90% reduction in register coverage across both targets, suggesting that the additional instrumentation and PSO-based optimizations in our hardware fuzzing approach do not significantly degrade the ability to explore the DUT's design space.

To more accurately assess PORTRUSH's performance in uncovering write port contention side-channel vulnerabilities, we introduce the metric of **write port contention coverage (WPC coverage)**, defined as the proportion of statically identified potential write port contention instances that are dynamically triggered during fuzzing. As Table IV shows, PORTRUSH achieves a WPC coverage of 83.3% on BOOM and 17.6% on NutShell, with an overall coverage of 32.1%. In contrast, DiFuzzRTL only achieves 20.8% and 1.2% on the two CPUs, respectively, with an overall coverage of 5.5%. These results demonstrate that, while maintaining competitive register coverage, PORTRUSH is significantly more effective than DiFuzzRTL in exercising and exposing write port contention side-channel vulnerabilities in modern CPUs.

D. Write Port Contention Side-channel Vulnerability (RQ3)

By automatically combining discovered write port contention with known transient execution attack patterns, we discovered three vulnerabilities that can be exploited to construct write port contention side-channel attacks. Among them, two are new vulnerabilities, `Birgus-variant` and `MSHRush`, while the third is the known `Spectre-STC` vulnerability. The two newly discovered vulnerabilities have been reported and assigned CVE identifiers.

By leveraging these vulnerabilities, we constructed write port contention side-channel attacks on both BOOM and NutShell processors. The performance of the three attack vectors is presented in Table V, where the **Error Rate** denotes the accuracy of secret leakage, and the **Leakage Rate** represents the speed at which secret bits are leaked by the attack sequences generated by PortRush. As shown in Table V, all three attack variants exhibit error rates below 10% and achieve leakage rates exceeding 15 Kb/s. An error rate below 10% enables PortRush to accurately infer the value of each secret bit using a majority voting scheme over repeated experiments. Specifically, for each secret bit, we perform the value inference five times, record the number of times the bit is inferred as 0 or 1, and select the value with the higher count as the final result. This majority voting approach, combined with a low per-trial error rate, ensures that each secret bit can be accurately recovered, as the probability of incorrect inference across multiple trials is significantly reduced. These results demonstrate that write port contention side-channel vulnerabilities can serve as a reliable attack vector when combined with transient execution attack patterns, enabling effective leakage of secret values. Such write port contention side-channel attack vectors are different from traditional cache-based attacks. They exploit competition for limited write ports in shared CPU structures rather than cache occupancy or timing, allowing information leakage even in processors with secure or partitioned caches.

E. Case Study of the `MSHRush` Attack (RQ4)

In this section, we analyze the newly discovered write port contention side-channel vulnerability `MSHRush` in detail, and further explain how the new attack variant exploits the contention vulnerability to leak secret information. Analysis of `Birgus-variant` and `Spectre-STC` is also provided in §A and §B of the Appendix.

`MSHRush` is a Spectre-type attack targeting RISC-V BOOM that is unique in exploiting write port contention at the L1 data cache (`BoomDataArray_array`) rather than the register file. The code snippet is shown in Listing 2, and the attack flow is illustrated in Figure 6.

TABLE V: The performance of the three attack variants

Variant	Error Rate	Leakage Rate	New variant?
`Birgus-variant`	8.7%	24Kb/s	✓
`MSHRush`	4.5%	15Kb/s	✓
`Spectre-STC`	4.8%	19Kb/s	×

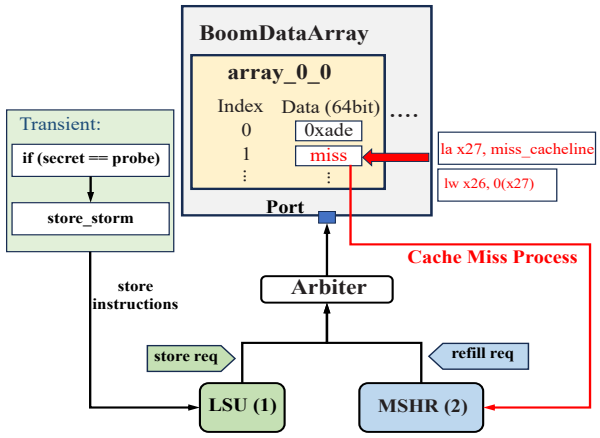

Fig. 6: The illustration of `MSHRush` attack

```
1  attacker:
2    rdcycle x10
3    call victim
4    rdcycle x11
5    sub x11, x11, x10
6    /* Use timing difference in x11 to infer secret bit */
7  victim:
8    la x27, miss_cacheline /* Address misses in cache */
9    lwu x25, 0(x27) /* Cache miss and MSHR allocation */
10   lwu x26, 0(x25) /* Load wait for refill completion */
11   ... /* More instructions dependent on x25 and x26 */
12   beqz x30, L1 /* Branch depends on refill result */
13 transient:
14   la x21, secret
15   lbu x22, 0(x21) /* Load secret byte */
16   andi x22, x22, 0x1 /* Extract secret bit */
17   beqz x22, L1 /* If secret bit = 0, skip store storm */
18   la x23, store_array
19   li x24, 64
20 store_storm:
21   sw x0, 0(x23) /* Store hits to saturate write port */
22   addi x24, x24, -1
23   bnez x24, store_storm
24 L1:
25   ...
26   ret
```

Listing 2: `MSHRush` in BOOM.

Write port contention of BOOM in DCache. In BOOM, the L1 data cache (`BoomDataArray_array`) provides only a single write port, shared by both the `Load/Store Unit` (`LSU`) and `Miss Status Handling Register` (`MSHR`). When `LSU` and `MSHR` simultaneously issue write requests, the `LSU` is given higher priority, resulting in delayed `MSHR` refills. An attacker can saturate the `store` pipeline so that the `LSU` issues continuous write requests, monopolizing the write port and prolonging `MSHR` miss recovery.

Technical details of `MSHRush`. Similar to Spectre-type attacks, `MSHRush` exploits transient execution to encode secret information into microarchitectural state, write port contention timing. The attack creates a measurable timing difference in the `victim` function's execution that depends on secret data. The attack proceeds in four phases:

(i) Establishing the transient window (Lines 8–12). The attacker first ensures a cache miss by having the `victim` load data from a deliberately evicted cache line (Line 9). This cache miss triggers an `MSHR` allocation to refill the missing data from

memory. The attacker then chains subsequent `load` instructions (Lines 10–11) and a conditional branch `beqz` (Line 12) that depend on the missed data. Since these instructions cannot resolve until the slow memory access completes, they create a dependency chain. Meanwhile, the branch predictor has been trained to mispredict this branch, causing the processor to speculatively execute the instructions following the branch, which forms the transient window where attack operations will occur.

(ii) Secret-dependent transient behavior (Lines 13–19). Inside the transient window, the attack loads the secret value and extracts its least significant bit (Lines 14–16). Crucially, the attack then takes different execution paths based on this secret bit: If the secret bit is 1, the attack enters a tight loop (Lines 18–23) that repeatedly executes `store` instructions. This `store_storm` saturates the `LSU` pipeline, causing it to continuously issue write requests to the write port. Since `LSU` has higher priority than `MSHR`, this creates sustained write port contention that blocks the `MSHR` refill operation. If the secret bit is 0, the `store_storm` is skipped entirely, leaving the write port available. The `MSHR` can complete its refill operation without interference.

(iii) Microarchitectural state persistence. When the original cache miss finally completes, the branch misprediction is detected and all speculatively executed instructions are squashed. For instance, the architectural effects (register writes, etc.) are discarded. However, the microarchitectural timing effects persist. The write port contention caused by the `store_storm` has already delayed the `MSHR` refill. This delay is observable because it affects when the dependency chain (Lines 11–13) can finally resolve and the `victim` function can complete.

(iv) Timing-based secret extraction. The attacker measures the total execution time of the `victim` function. When the secret bit is 1, the write port contention delays the `MSHR` refill, resulting in longer execution time. When the secret bit is 0, the refill completes faster, resulting in shorter execution time. By repeatedly invoking the `victim` function and measuring these timing differences, the attacker can reliably infer the value of each secret bit, effectively leaking the entire secret one bit at a time.

We evaluated the attack by setting the secret to different values (e.g., `0xdeadbeef`) and repeatedly testing the leakage accuracy. Figure 7 shows the distribution of execution cycles for secret values 1 and 0. The x-axis represents clock cycles, while the y-axis indicates the number of times the `victim` function executed in each cycle during the side-channel attack. For example, when inferring a 32-bit secret (e.g., `0xdeadbeef`) with 5 repetitions per bit, there are 160 total measurements. These execution times may be distributed across different clock cycles. The red line shows the distribution when the inferred secret bit is 0, the blue line corresponds to when it is 1, and some overlap exists due to noise, such as microarchitectural optimizations and RTL simulation variability.

We observe that when the secret is 0, the execution duration

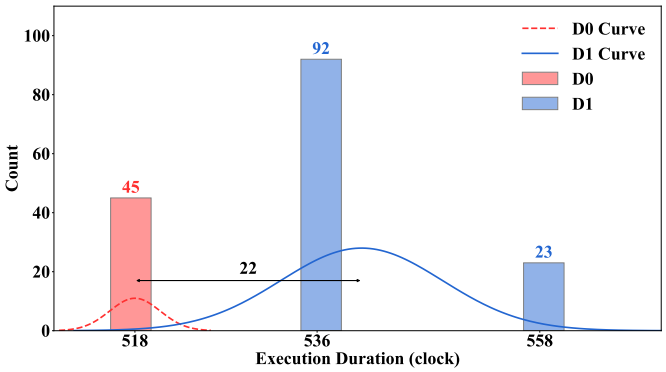

Fig. 7: Distribution of `victim` function execution cycles for secret value equals 0 (D0) and 1 (D1) in `MSHRush`

of `victim` is 518 cycles. When the secret is 1, the execution duration of `victim` can be either 536 or 558 cycles. Thus, the expected execution duration, calculated as the sum of each cycle value multiplied by its probability, is 518 cycles for secret 0 and 540 cycles for secret 1—a difference of 22 cycles. This 22-cycle increase demonstrates that write port contention—triggered when the secret bit is 1—significantly delays the execution of `victim`. The substantial and consistent timing difference between the two secret values demonstrates the effectiveness of our approach in inducing measurable timing side channels via port contention.

VII. Discussion

Root causes of write port contention. We present PortRush, a systematic hardware fuzzing framework for detecting and analyzing write port contention side-channel vulnerabilities at the RTL in modern CPUs. PortRush can systematically identify potential write port contention, trigger such contention by fuzzing, and validate the vulnerability of write port contention in real side-channel attacks. This highlights the framework's effectiveness compared to traditional manual analysis and hardware fuzzing methods. Through extensive evaluation, PortRush uncovers not only new attack variants but also provides deep insights into the architectural factors that make write port contention exploitable. Specifically, our study identifies root causes of write port contention in modern out-of-order RISC-V CPUs. First, write port contention often arises when the number of write entities exceeds the number of available write ports, as many designs deliberately limit the number of ports to balance design complexity, power consumption, and performance. Second, multi-issue out-of-order execution increases the likelihood of simultaneous write requests, making it feasible for attackers to manipulate dependencies and induce contention. Third, the widespread use of fixed-priority arbitration schemes, such as static priority arbiters, can be exploited to favor certain write requests consistently. Finally, short-latency instructions can preempt longer-latency instructions by completing and issuing write requests within a single cycle, thereby amplifying contention and side-channel risk.

Challenges in mitigating write port contention. Mitigating write port contention side-channel vulnerabilities presents several challenges. While architectural optimizations such as fair or round-robin arbitration can be introduced at critical points to distribute access more evenly, these approaches are not without drawbacks. For instance, in the `MSHR` scenario of BOOM, fair arbitration assigns equal priority and bandwidth to `store` and `MSHR` refill requests, which can cause frequent interference and negatively impact overall CPU performance. Attackers may also exploit these mechanisms by flooding the system with `MSHR` refill requests, thereby disrupting normal `store` operations and inferring secret values through timing analysis. Furthermore, increasing the number of write ports, while potentially reducing contention, introduces additional design complexity, area, and power consumption. Thus, CPU designers must carefully balance performance, complexity, and security when considering such mitigation strategies.

Optimal architectural defenses. Effective mitigation requires a combination of targeted architectural enhancements. First, scaling the number of write ports to match the level of concurrent demand can minimize contention windows and structural hazards, reducing the risk of side-channel exploitation. Second, prioritizing long-latency instructions during arbitration helps prevent critical operations from being persistently delayed by short-latency requests, thereby limiting observable timing variations. Third, implementing advanced arbitration mechanisms, such as round-robin or resource-aware policies, at contention-prone points can further distribute access equitably and prevent privilege escalation attacks. Collectively, these measures address both performance and security, making them robust solutions for mitigating write port contention side-channel vulnerabilities in modern CPU microarchitectures.

VIII. Related Work

Hardware Fuzzing. Prior work in hardware fuzzing, such as RFUZZ [28], DifuzzRTL [24], MorFuzz [39], RISCVuzz [35] and TheHuzz [26], has established coverage-guided and differential fuzzing as effective techniques for exposing functional bugs and undocumented instructions at the register-transfer level (RTL) of CPUs. These frameworks utilize various strategies, including cycle-level input generation and automated test harnesses, to maximize hardware coverage and identify behavioral discrepancies between the DUT and golden reference models. However, these approaches are primarily oriented toward functional validation and do not provide targeted mechanisms to systematically detect or analyze microarchitectural security vulnerabilities, such as write port contention. As a result, there remains a critical need for specialized fuzzing frameworks that can uncover security-sensitive contention vulnerabilities at the RTL.

Side-Channel and Transient Execution Vulnerability Detection. A substantial body of research has focused on detecting microarchitectural side-channels and transient execution vulnerabilities, employing both formal verification and fuzzing-based methodologies. Formal approaches, exemplified by UPEC [19] and Checkmate [36], use static analysis and bounded model checking to identify timing side-channels or transient execution bugs in RTL designs. Fuzzing-based techniques, such as Osiris [37], SIGFuzz [33], and WhisperFuzz [13], generate instruction sequences to trigger and analyze timing anomalies, aiming to uncover security-relevant timing behaviors. While these methods have advanced the detection of generic timing side-channels and some transient execution vulnerabilities, they often struggle to pinpoint root causes at the level of specific microarchitectural resources, such as write ports, and may require substantial manual effort for vulnerability analysis and mitigation. This highlights the need for automated, resource-aware detection frameworks capable of fine-grained vulnerability analysis.

Port Contention Side-Channels and Speculative Execution Attacks. Recent studies have demonstrated that port contention—arising from competition for shared execution resources—can serve as a powerful side channel in speculative and transient execution attacks. SMoTherSpectre [12] exploits execution port contention in simultaneous multithreading (SMT) processors by flooding specific execution ports to infer secret-dependent instruction types, while PortSmash [2] demonstrates cross-core port contention attacks on Intel's Skylake and Kaby Lake architectures by monitoring timing variations when concurrent threads compete for execution ports. Both attacks operate at the instruction execution level, focusing on exploiting contention in functional execution ports. In contrast, PortRush targets *write port contention*—a distinct microarchitectural bottleneck that occurs when multiple instructions or functional modules simultaneously attempt to write results to shared storage elements such as register files or cache arrays. While SMoTherSpectre and PortSmash rely on software-level profiling and manual attack construction to detect and exploit execution port contention, PortRush provides an automated, RTL-level approach that identifies potential write port contention through static analysis of hardware designs and automatically generates instruction sequences to trigger and validate these vulnerabilities during the hardware design phase.

IX. Conclusion

CPU vulnerabilities continue to present significant security challenges in modern architectures, with write port contention remaining an underexplored threat. In this paper, we introduced PortRush, the hardware fuzzing framework designed for the detection and validation of write port contention side-channel vulnerabilities at the RTL. By abstracting the Write Request Graph, employing a hierarchical aggregation and decoding method, and leveraging a contention-guided fuzzing approach powered by Particle Swarm Optimization, PortRush enables efficient detection and triggering of write port contention. Our evaluation on three RISC-V CPUs, BOOM, NutShell, and Rocket Core, demonstrates the effectiveness of PortRush in uncovering previously unknown vulnerabilities. Notably, PortRush discovered two novel attack variants (`Birgus-variant` and `MSHRush`) and reproduced the existing `Spectre-STC` attack on BOOM. These findings

highlight the security risks posed by write port contention and underscore the need for systematic hardware-level analysis to mitigate such side-channel threats in future CPU designs.

Ethical Considerations

Our research on write port contention side-channel vulnerabilities might bring ethical concerns regarding experiment environments and vulnerability disclosures. We have carefully and proactively considered these research ethics throughout this project. First, our research and experiments were conducted in a local environment (e.g., servers) and did not involve other persons or other live systems. Therefore, it would not have an impact on other parties or live services. Second, we attest that any newly discovered vulnerabilities were responsibly disclosed to CVE to ensure proper mitigation. We did not publicly disclose the vulnerabilities to the broad audience to avoid any potential harm or public exploitations.

Acknowledgments

The authors would like to thank the anonymous reviewers for their insightful suggestions on our paper. This work is carried out with the support of the Hunan Provincial Key Laboratory of Intelligent and Parallel Analysis for Software Security, and is partially supported by the National Natural Science Foundation China (62272472, 62306328, 62302050, 62372121, 62402509), the Science and Technology Innovation Program of Hunan Province (2024RC3136), the Innovative Research Group Project of National Natural Science Foundation of China (62421002), the National University of Defense Technology Research Project (ZK23-14), the Natural Science Foundation of Hunan Province (2021JJ40692, 2025JJ40053), and the Research Project of Key Laboratory (WDZC20245250105).

References

[1] Mor1kx, 2025. https://github.com/openrisc/mor1kx.

[2] Portsmash, 2025. https://github.com/bbbrumley/portsmash.

[3] Risc-v isa manual (privileged), 2025. https://riscv.org/specifications/privileged-isa/.

[4] Riscyoo: Risc-v out-of-order processors, 2025. https://github.com/csail-csg/riscy-OOO.

[5] Rocket core, 2025. https://github.com/chipsalliance/rocket-chip.

[6] Ryzenfallen, 2025. https://github.com/depletionmode/Ryzenfallen.

[7] scala-sbt, 2025. https://www.scala-sbt.org/.

[8] Side channel vulnerabilities: Microarchitectural data sampling and transactional asynchronous abort, 2025. https://www.intel.com/content/www/us/en/architecture-and-technology/mds.html.

[9] Synopsys vcs, 2025. https://www.synopsys.com/verification/simulation/vcs.html.

[10] Xiang shan, 2025. https://github.com/OpenXiangShan/XiangShan.

[11] Chisel 3. A modern hardware design language, 2025. https://github.com/freechipsproject/chisel3.

[12] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. Smotherspectre: Exploiting speculative execution through port contention. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 785–800, New York, NY, USA, 2019. Association for Computing Machinery.

[13] Pallavi Borkar, Chen Chen, Mohamadreza Rostami, Nikhilesh Singh, Rahul Kande, Ahmad-Reza Sadeghi, Chester Rebeiro, and Jeyavijayan Rajendran. WhisperFuzz: White-Box fuzzing for detecting and locating timing vulnerabilities in processors. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 5377–5394, Philadelphia, PA, August 2024. USENIX Association.

[14] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution. In *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, MD, August 2019. USENIX Association.

[15] RISC-V CPU Developed by OSCPU Team. Nutshell, 2025. https://github.com/OSCPU/NutShell.

[16] Chen Chen, Rahul Kande, Nathan Nguyen, Flemming Andersen, Aakash Tyagi, Ahmad-Reza Sadeghi, and Jeyavijayan Rajendran. HyPFuzz: Formal-Assisted processor fuzzing. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1361–1378, Anaheim, CA, August 2023. USENIX Association.

[17] D. Cyrluk, S. Rajan, N. Shankar, and M. K. Srivas. Effective theorem proving for hardware verification. In Ramayya Kumar and Thomas Kropf, editors, *Theorem Provers in Circuit Design*, pages 203–222, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[18] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *ACM Trans. Archit. Code Optim.*, 8(4), January 2012.

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

[20] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. Kaslr is dead: Long live kaslr. In Eric Bodden, Mathias Payer, and Elias Athanasopoulos, editors, *Engineering Secure Software and Systems*, pages 161–176, Cham, 2017. Springer International Publishing.

[21] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+flush: A fast and stealthy cache attack. In Juan Caballero, Urko Zurutuza, and Ricardo J. Rodríguez, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 279–299, Cham, 2016. Springer International Publishing.

[22] J. L. Hennessy and D. A. Patterson. Computer architecture: A quantitative approach. 2011.

[23] Jaewon Hur, Suhwan Song, Sunwoo Kim, and Byoungyoung Lee. Specdoctor: Differential fuzz testing to find transient execution vulnerabilities. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS'22, page 1473–1487, New York, NY, USA, 2022. Association for Computing Machinery.

[24] Jaewon Hur, Suhwan Song, Dongup Kwon, Eunjin Baek, Jangwoo Kim, and Byoungyoung Lee. Difuzzrtl: Differential fuzz testing to find cpu bugs. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1286–1303, 2021.

[25] Saad Islam, Ahmad Moghimi, Ida Bruhns, Moritz Krebbel, Berk Gulmezoglu, Thomas Eisenbarth, and Berk Sunar. Spoiler: Speculative load hazards boost rowhammer and cache attacks. In *28th USENIX Security Symposium (USENIX Security 19)*, SANTA CLARA, CA, August 2019. USENIX Association.

[26] Rahul Kande, Addison Crump, Garrett Persyn, Patrick Jauernig, Ahmad-Reza Sadeghi, Aakash Tyagi, and Jeyavijayan Rajendran. TheHuzz: Instruction fuzzing of processors using Golden-Reference models for finding Software-Exploitable vulnerabilities. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3219–3236, Boston, MA, August 2022. USENIX Association.

[27] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19, 2019.

[28] Kevin Laeufer, Jack Koenig, Donggyu Kim, Jonathan Bachrach, and Koushik Sen. Rfuzz: Coverage-directed fuzz testing of rtl on fpgas. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, page 1–8. IEEE Press, 2018.

[29] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 973–990, Baltimore, MD, August 2018. USENIX Association.

[30] Sujit Kumar Muduli, Gourav Takhar, and Pramod Subramanyan. Hyperfuzzing for soc security validation. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9, 2020.
[31] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of aes. In David Pointcheval, editor, *Topics in Cryptology – CT-RSA 2006*, pages 1–20, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.
[32] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of aes. CT-RSA'06, page 1–20, Berlin, Heidelberg, 2006. Springer-Verlag.
[33] Chathura Rajapaksha, Leila Delshadtehrani, Manuel Egele, and Ajay Joshi. Sigfuzz: A framework for discovering microarchitectural timing side channels. In *2023 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 1–6, 2023.
[34] W. Snyder. Verilator, 2025. https://www.veripool.org/wiki/verilator.
[35] Fabian Thomas, Lorenz Hetterich, Ruiyi Zhang, Daniel Weber, Lukas Gerlach, and Michael Schwarz. Riscvuzz: Discovering architectural cpu vulnerabilities via differential hardware fuzzing, 2024. ghostwriteattack.com.
[36] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. Checkmate: Automated synthesis of hardware exploits and security litmus tests. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 947–960, 2018.
[37] Daniel Weber, Ahmad Ibrahim, Hamed Nemati, Michael Schwarz, and Christian Rossow. Osiris: Automated discovery of microarchitectural side channels. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1415–1432. USENIX Association, August 2021.
[38] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. Scattercache: thwarting cache attacks via cache set randomization. In *Proceedings of the 28th USENIX Conference on Security Symposium*, SEC'19, page 675–692, USA, 2019. USENIX Association.
[39] Jinyan Xu, Yiyuan Liu, Sirui He, Haoran Lin, Yajin Zhou, and Cong Wang. MorFuzz: Fuzzing processor via runtime instruction morphing enhanced synchronizable co-simulation. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1307–1324, Anaheim, CA, August 2023. USENIX Association.
[40] Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. EcoFuzz: Adaptive Energy-Saving greybox fuzzing as a variant of the adversarial Multi-Armed bandit. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2307–2324. USENIX Association, August 2020.
[41] Gen Zhang, Pengfei Wang, Tai Yue, Xiangdong Kong, Shan Huang, Xu Zhou, and Kai Lu. Mobfuzz: Adaptive multi-objective optimization in gray-box fuzzing. 2022.
[42] Gen Zhang, Pengfei Wang, Tai Yue, Danjun Liu, Yubei Guo, and Kai Lu. Instiller: Toward efficient and realistic rtl fuzzing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 43(7):2177–2190, 2024.
[43] J. Zhao, B. Korpan, and others. Sonicboom: The 3rd generation berkeley out-of-order machine. *4th Workshop on Computer Architecture Research with RISC-V*, 2020.

APPENDIX

A. `Birgus-variant` in NutShell

We present `Birgus-variant`, a novel Spectre-type attack discovered in NutShell that leverages previously unknown write port contention to establish a new microarchitectural side channel. Unlike prior port contention attacks [2], [12], [23], Birgus-variant leverages contention at the `Reorder Buffer`'s physical register file (`ROB_prf`) to leak secrets through timing. The code snippet is shown in Listing 3, and the attack flow is illustrated in Figure 9.

Write port contention in `ROB_prf` of NutShell. NutShell, a dual-issue out-of-order RISC-V core, provisions two write ports on `ROB_prf` to support concurrent write requests from functional modules (e.g., `LSU`, `Divider`, and `ALU`, etc.) However, as shown in Table I, up to seven modules may simultaneously issue write requests. Contention occurs when more than two requests arrive in a single cycle, as shown in Figure 8. Arbitration delays lower-priority modules (e.g., `ALU`), introducing measurable timing variations. By exploiting the variable latency of division instructions, an attacker can induce and measure contention-dependent delays to infer secret data.

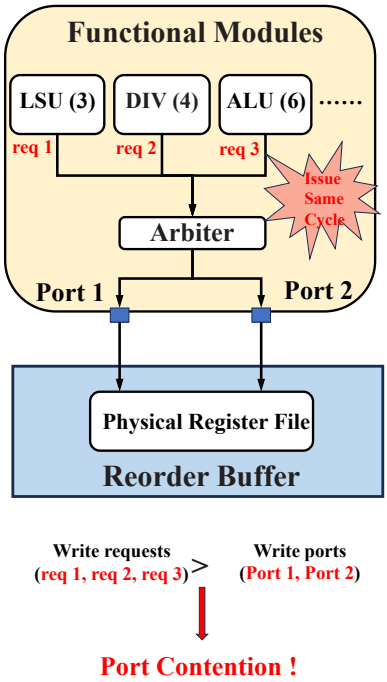

Fig. 8: The illustration of port contention in `ROB_prf`

```
attacker:
  rdcycle x10
  call victim
  rdcycle x11
  sub x11, x11, x10
  /* Use timing difference in x11 to infer secret bit */
victim:
  /*Prepare cache addresses and divisor before attack*/
  li x14, 1 /* divisor for div_lsu_storm */
  la x25, hit-cacheline /*Pre-faulted, always cache-hit*/
  la x15, miss-cacheline /*Pre-faulted, always
      ↪cache-miss*/
  /* multiple alu instructions depend on x15 */
  lwu x16, 0(x15) /* Slow load from miss-cacheline */
  add x17, x16, x14 /* Stall until load completes */
  sub x18, x17, x14
  .... /* More instructions dependent on x15 and x18 */
  /* Branch predictor misprediction setup */
  beqz x18, L1 /*Branch mispredicted as not taken*/
transient:
  /* Begin transient window */
  la x19, secret /* Load address of secret */
  ld x20, 0(x19) /* Load secret value */
  div x21, x21, x20 /*Secret-dependent division timing*/
div_lsu_storm:
  /* multiple div and lsu instructions within 1 cycle */
  lwu x24, 0(x25)
  div x24, x21, x14
  lwu x24, 0(x25)
  ...
L1:
  ...
  ret
```

Listing 3: Code snippet of `Birgus-variant` in Nutshell

Technical details of `Birgus-variant`. Similar to Spectre-type attacks, `Birgus-variant` exploits transient execution attack patterns to encode secret information into microarchitectural timing differences. The key insight is to exploit the delay of lower-priority `alu` write requests caused by simultaneous higher-priority `load` and `division` instructions. By measuring execution time differences in the `victim` function, the attacker can infer secret data. The attack proceeds in four phases:

(i) Establishing the transient window (Lines 9–18). The attacker first prepares cache addresses and divisors (Lines 9–11) to align `div` and `load` instructions, ensuring they can complete in one cycle and issue write requests simultaneously in the later `div_lsu_storm`. Multiple `alu` instructions (Lines 14–16) are made data-dependent on a cache-miss `load` instruction (Line 13), causing them to stall until the slow memory access completes. The branch instruction `beqz` (Line 18) depends on these `alu` results. This branch has been trained to mispredict, causing the processor to speculatively execute multiple instructions in the transient window (Lines 20–29).

(ii) Secret-dependent transient behavior (Lines 20–29). Inside the transient window, the attack includes secret-dependent `load` and `div` operations (Lines 21–23). Crucially, the execution timing differs based on the secret bit. If the secret bit is 0, the `div` instruction completes in one cycle, and the dependent `div_lsu_storm` instructions also finish promptly. This avoids write port contention with the stalled `alu` instructions. If the secret bit is 1, the `div` instruction takes 16 cycles. Therefore, `div` and `load` instructions in `div_lsu_storm` are delayed by 15 cycles and execute concurrently with the `alu` instruction waiting on the `load` dependency. This creates simultaneous write requests to the two write ports of `ROB_prf`, causing write port contention that further delays the lower-priority `alu` instructions.

(iii) Microarchitectural state persistence. When the original cache miss completes, the branch misprediction is detected and all speculatively executed instructions are squashed—their architectural effects are discarded. However, the microarchitectural timing effects persist. The write port contention caused when the secret bit is 1 has already delayed the `alu` instructions. Since the `beqz` instruction branch resolution depends on the completion of these `alu` instructions, the total execution time of the `victim` function becomes secret-dependent and remains observable.

(iv) Timing-based secret extraction. The attacker measures the total execution time of the `victim` function to infer the secret bit. We evaluated the attack by setting the secret to different values (e.g., `0xdeadbeef`) and repeatedly testing the leakage accuracy. As shown in Figure 10, the distribution of execution cycles differs significantly for secret values 1 and 0. The x-axis represents clock cycles, while the y-axis indicates the number of times the `victim` function executed in each cycle during the side-channel attack. For example, when inferring a 32-bit secret (e.g., `0xdeadbeef`) with 5 repetitions per bit, there are 160 total measurements distributed across different clock cycle values. The red line shows the distribution when the inferred secret bit is 0, and the blue line corresponds to when it is 1.

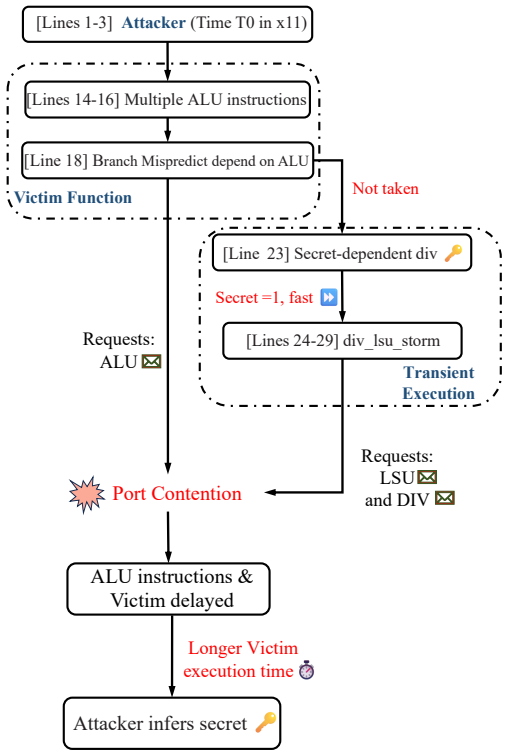

Fig. 9: The illustration of `Birgus-variant`

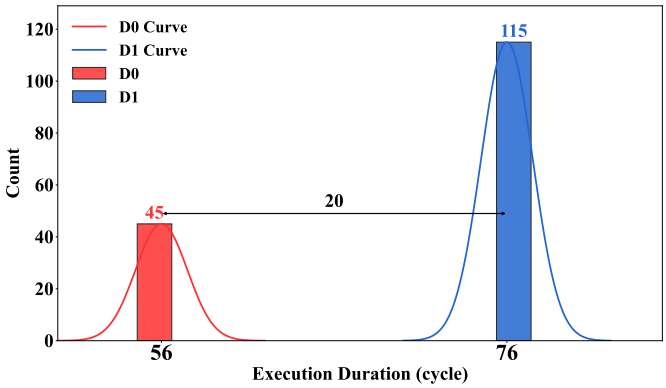

Fig. 10: Distribution of `victim` function execution cycles for secret value equals 0 (D0) and 1 (D1) in `Birgus-variant`

We observe that the execution time cluster is around 55 cycles when the secret bit is 0, and around 75 cycles when the bit is 1. This 20-cycle (36.6%) increase demonstrates that write port contention—triggered when the secret bit is 1—significantly delays the execution of `victim` function. The substantial and consistent timing gap between the two cases demonstrates the effectiveness of our approach in inducing measurable timing side channels via port contention, enabling accurate leakage of secret values despite microarchitectural noise and simulation variability.

B. *Spectre-STC in BOOM*

Spectre-STC is a Spectre-type attack previously identified in BOOM [13], [33]. This attack exploits a critical microarchitectural feature of BOOM: the integer register file is equipped with a single write port shared among multiple functional modules, including DIV, MUL, and ALU. When DIV, MUL, and ALU simultaneously issue write requests, a strict arbitration policy prioritizes ALU and MUL units over the latency-sensitive DIV unit.

The code snippet is shown in Listing 4, and the attack flow is illustrated in Figure 11, The attack proceeds as follows: i) Insert the data-dependent and low-priority division instruction (Line 10). ii) Train the mispredicted branch to open the transient window (Line 13). iii) Load the secret and extract the secret bit (Lines 16-18), and set the secret-dependent branch (Line 19). iv) Insert multiple alu instructions after the secret-dependent branch (alu_storm), which will compete with the division on the same write port. This contention only occurs for $secret = 1$, causing extra delay for the division's completion. After the branch resolves, speculative instructions are squashed, but the timing impact persists and can be measured by the attacker in register $x11$.

Leveraging the write port contention side-channel vulnerability found in Spectre-STC, PortRush automatically generated the attack vector. We evaluated the attack by setting the secret to different values (e.g., 0xdeadbeef) and repeatedly testing the leakage accuracy. Our experimental results demonstrate that this attack achieves secret recovery with an error rate below 5% and a bit rate of 19 Kb/s, highlighting the effectiveness of combining transient execution with microarchitectural write port contention on modern out-of-order cores.

```
1  attacker:
2    rdcycle x10
3    call victim
4    rdcycle x11
5    sub x11, x11, x10
6    /* Use timing difference in x11 to infer secret bit */
7  victim:
8    ...
9    /* low-priority div is data-dependent on beq branch */
10   div x18, x15, x14
11   ...
12   /* Branch predictor misprediction setup */
13   beq x16, x18, L1 /* Branch mispredicted as not taken */
14 transient:
15   /* Begin transient window */
16   la x19, secret /* Load address of secret */
17   ld x20, 0(x19) /* Load secret value */
18   andi x21, x21, 0x1 /* Extract secret bit */
19   beqz x21, L1 /* If secret bit = 0, skip alu storm */
20 alu_storm:
21   /* multiple alu instructions within 1 cycle */
22   alu x24, x15, x14
23   ...
24 L1:
25   ...
26   ret
```

Listing 4: Spectre-STC in BOOM

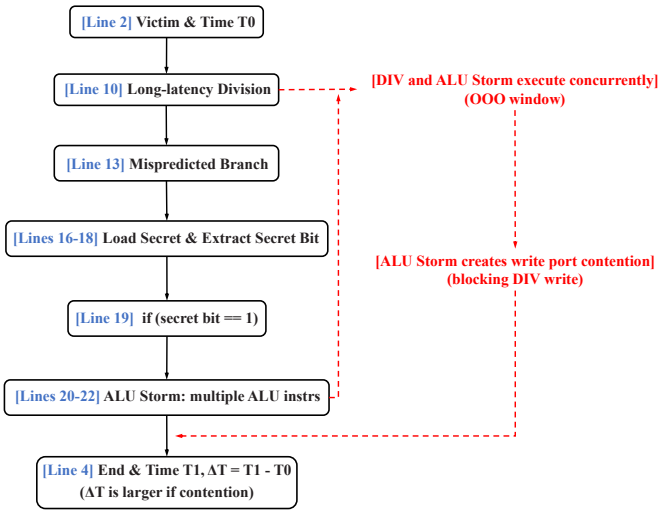

Fig. 11: The illustration of Spectre-STC attack