arXiv:2402.03704v2 [cs.CR] 14 Mar 2024

WhisperFuzz: White-Box Fuzzing for Detecting and Locating Timing Vulnerabilities in Processors

Pallavi Borkar[§,*], Chen Chen[†,*], Mohamadreza Rostami[‡], Nikhilesh Singh[§], Rahul Kande[†], Ahmad-Reza Sadeghi[‡], Chester Rebeiro[§], and Jeyavijayan (JV) Rajendran[†]
§*Indian Institute of Technology Madras, India,* [†]*Texas A&M University, USA,*
[‡]*Technische Universität Darmstadt, Germany*
§{cs20d202, nik, chester}@cse.iitm.ac.in,
[†]{chenc, rahulkande, jv.rajendran}@tamu.edu,
[‡]{mohamadreza.rostami, ahmad.sadeghi}@trust.tu-darmstadt.de

Abstract

Timing vulnerabilities in processors have emerged as a potent threat. As processors are the foundation of any computing system, identifying these flaws is imperative. Recently fuzzing techniques, traditionally used for detecting software vulnerabilities, have shown promising results for uncovering vulnerabilities in large-scale hardware designs, such as processors. Researchers have adapted black-box or grey-box fuzzing to detect timing vulnerabilities in processors. However, they cannot identify the locations or root causes of these timing vulnerabilities, nor do they provide coverage feedback to enable the designer's confidence in the processor's security.

To address the deficiencies of the existing fuzzers, we present *WhisperFuzz*—the first white-box fuzzer with static analysis —aiming to detect and locate timing vulnerabilities in processors and evaluate the coverage of microarchitectural timing behaviors. *WhisperFuzz* uses the fundamental nature of processors' timing behaviors, *microarchitectural state transitions*, to localize timing vulnerabilities. *WhisperFuzz* automatically extracts microarchitectural state transitions from a processor design at the register-transfer level (RTL) and instruments the design to monitor the state transitions as coverage. Moreover, *WhisperFuzz* measures the time a design-under-test (DUT) takes to process tests, identifying any minor, abnormal variations that may hint at a timing vulnerability. *WhisperFuzz* detects 12 new timing vulnerabilities across advanced open-sourced RISC-V processors: `BOOM`, `Rocket Core`, and `CVA6`. Eight of these violate the zero latency requirements of the Zkt extension and are considered serious security vulnerabilities. Moreover, *WhisperFuzz* also pinpoints the locations of the new and the existing vulnerabilities.

1 Introduction

The evolution in computer architecture has significantly amplified the complexity of hardware design, especially in modern processors, which are the foundation of today's computing systems. As technology advances, designers integrate more functionalities into hardware, leading to more intricate architectural and microarchitectural features in processors. However, as the complexity of the design increases, so does the number of design regions to verify and protect. Traditional techniques to verify modern processors cannot scale with the number of (new) hardware vulnerabilities discovered. For example, the number of newly detected hardware common vulnerabilities in the National Vulnerability Database (NVD) increased from three in 2012 to 92 in 2022 [5]. Further, as of 2023, MITRE reported 117 hardware-related vulnerability types, known as Common Weakness Enumerations (CWEs) [41]. These rapidly increasing vulnerabilities threaten the security of the expanding digital landscape across different domains necessitating efficient detection strategies [13, 15, 21, 44, 45].

Timing vulnerabilities are of particular concern as they can leak sensitive information, undermining the entire system's security. Well-known attacks such as Spectre [38], Meltdown [40], Foreshadow [61], LVI [62], RIDL [63], ZombieLoad [54], CrossTalk [50], Zenbleed [4], and Retbleed [68] exploit timing vulnerabilities present in a wide range of commercial processors. Multiple variants of these attacks have been shown to subvert security countermeasures implemented to prevent such attacks. Unlike functional vulnerabilities, timing vulnerabilities can manifest in a logically correct implementation, making them hard to detect. Timing vulnerabilities rely on the difference in execution time of the hardware components to leak sensitive information. These vulnerabilities underscore the need for rigorous security analysis in modern processors. Moreover, unlike software flaws, which can be patched post-deployment, fixing hardware vulnerabilities after manufacturing is difficult, as they are physically ingrained into the Silicon. Therefore, detecting vulnerabilities at the pre-Silicon stage is imperative for secure hardware.

Existing timing vulnerability detection strategies for processors use formal methods or fuzzing. Formal methods, such as theorem proving [20], model checking [18], assertion proving [70], and information-flow tracking [33] explore de-

*These authors contributed equally to this work.

sign spaces exhaustively and prove security assertions about hardware. Thus, detecting timing vulnerabilities using formal methods is a rigorous approach to ensure design security [22, 59]. However, these methods are limited by the state explosion problem [10, 17–19]; exhaustively exploring the complex modern hardware is computationally hard [16, 31]. Some approaches aim to handle this scalability issue by modeling hardware at the higher abstraction level and approximating its timing behavior [59]. However, abstracting hardware can lead to over-optimistic results or false positives [16]. Furthermore, these formal approaches require a comprehensive understanding of the designs' security specifications and manually defining properties, an error-prone process [46].

Alternatively, hardware fuzzing has shown its effectiveness in detecting vulnerabilities in large-scale designs [14, 16, 34, 37, 39, 71]. Using fuzzing, Google detected the recent vulnerability on AMD Zen2 processors, *Zenbleed* [4], a speculative execution vulnerability that allows attackers to extract sensitive information through software exploitation [47]. Black-box fuzzing [35, 67] and grey-box fuzzing [51] have been applied to detect timing vulnerabilities in processors. They explore the design spaces by generating different combinations of instructions as inputs and use performance counters to identify potential timing vulnerabilities [35, 51, 67]. While these techniques overcome the scalability issue of formal verification, they suffer from two critical limitations. First, although they successfully find instructions that cause timing vulnerabilities, they rely on confirmation from designers to identify and pinpoint the root cause (locations) [35, 51, 67]. Second, they lack the adequate coverage metric to capture the timing behaviors of the processor. Designers rely on coverage metrics to obtain the necessary confidence before tape-out [27, 36, 58]. Therefore, introducing such metrics to evaluate the progress of fuzzing is typical [69]. We will elaborate on these shortcomings in Section 5.

Our Goals and Contributions. We enhance existing fuzzing strategies to address their limitations by integrating static analysis. This allows us to automatically pinpoint the sources of timing vulnerabilities and compute the coverage of timing behaviors in processor designs. Our fuzzer efficiently explores the design space, detecting timing vulnerabilities, while our novel static approach identifies the root causes and provides timing behavior coverage.

Locating the root cause of timing vulnerabilities and computing timing coverage is non-trivial and poses several challenges: (i) expressing the timing behaviors of processor modules formally is complex, as they do not operate in isolation and can influence each other; (ii) finer measurement of module timing behaviors is needed, which is time-consuming for modern processors with numerous modules [73, 74]; (iii) tracing vulnerabilities to their root causes within the design space is intricate; and (iv) traditional mutation algorithms used in fuzzers are insufficient for detecting timing vulnerabilities due to their reliance on microarchitectural state transitions.

To address these challenges, (i) We have developed the *Micro-Event Graph*, a static program analysis technique that formally expresses module timing behaviors in a processor by extracting microarchitectural state transitions of a design-under-test (DUT) at the register-transfer level (RTL). To efficiently cover the extensive design space, we tailor the technique to generate individual graphs for each RTL module (cf. Section 3.5). (ii) We analyze each RTL module's simulation trace to measure its timing behaviors precisely. To streamline our analysis efforts, we devise a hierarchical strategy based on the characteristics of timing vulnerabilities to prioritize modules for examination (cf. Section 3.6). (iii) We pinpoint the root causes of detected timing vulnerabilities utilizing static analysis techniques and properties of the *Micro-Event Graph*, employing a module-wise strategy to navigate the complex design space. (iv) We have adapted traditional hardware fuzzing methods to efficiently explore a DUT's design spaces and crafted a specialized mutation engine to exploit timing vulnerabilities. Furthermore, we have instrumented graphs into the DUT to monitor module state transitions based on the input (cf. Section 3.8).

In summary, our contributions are:

- We present a novel white-box fuzzer with static analysis, *WhisperFuzz*, for timing vulnerability detection in processors at the RTL. *WhisperFuzz* extracts and monitors microarchitectural state transitions at RTL and measures the timing behaviors of each RTL module to identify timing vulnerabilities. Hardware fuzzing enables *WhisperFuzz* to explore the microarchitectural state space efficiently.

- With static analysis, *WhisperFuzz* will identify the locations/root causes of timing vulnerabilities. Moreover, *WhisperFuzz* introduces a timing coverage metric to help designers evaluate the timing behaviors explored.

- We evaluate the effectiveness of *WhisperFuzz* on three real-world, open-sourced processors from RISC-V instruction set architecture (ISA) – `BOOM` [74], `Rocket Core` [9], and `CVA6` [73], which are widely used as benchmarks in the hardware security community.

- *WhisperFuzz* finds 12 new timing vulnerabilities across all three benchmarks. Eight of them pose serious security vulnerabilities, according to the RISC-V Zkt contract [43]. *WhisperFuzz* also pinpoints the locations of all existing and new vulnerabilities.

2 Background

In this section, we provide a succinct background on hardware fuzzing and microarchitectural timing side channels, which form the basis of *WhisperFuzz*.

2.1 Hardware Fuzzing

Hardware fuzzing is a dynamic verification technique that iteratively generates testing inputs called tests to verify target hardware [16, 34, 39]. A coverage-feedback fuzzer starts by generating an initial set of tests, called **seeds**, randomly using a *seed generator*. When fuzzing processors, these seeds are executable programs with a sequence of instructions [16, 37]. The fuzzer simulates the target hardware with these tests using open-source or commercial hardware simulation tools such as Verilator [56] and Synopsys VCS [3].

During simulation, the fuzzer collects **coverage** information that quantifies the activities caused by the test in the hardware. For instance, the coverage information can be represented as transitions of finite-state machines (FSMs) [37]. Fuzzers either instrument the hardware to add activity monitors [34, 39], or use the existing coverage monitors of the simulation tools [16, 37] to collect this coverage information. Next, it generates new tests automatically by performing bit manipulation operations, called **mutations** on the current tests, increasing coverage. The fuzzer iterates over this cycle of test generation and simulation to verify target hardware till desired coverage is achieved.

Fuzzers use a *vulnerability detector* to detect vulnerabilities using differential testing or hardware assertions. In differential testing, the fuzzer runs a golden reference model along with the target hardware and compares its outputs to detect vulnerabilities [34, 37]. Alternatively, some fuzzers insert hardware assertions, i.e., rules describing the expected behavior, in the target hardware and detect vulnerabilities as violations of these assertions [42]. While existing black-box [35, 67] and grey-box [51] fuzzers can detect timing vulnerabilities in processors, these approaches fail to locate the root causes of these vulnerabilities. Further, these techniques can not evaluate the explored timing behaviors. These limitations delay the mitigation process and hamper the designer's confidence due to absence of a coverage metric.

2.2 Timing Side-Channel Attacks

Timing side-channel attacks exploit measurable variations in the execution time of instructions to glean secret information from victim applications. These timing variations arise from interactions of different operands with the microarchitecture. For instance, [11] proposed a timing attack to recover AES keys by measuring the cache accesses during encryption across different keys.

Over the years, multiple techniques to perform timing attacks have developed such as PRIME+PROBE [48], EVICT+TIME [48], and FLUSH+RELOAD [30]. The key idea of all these attacks is to target a specific shared resource, such as the cache memory, and exploit its data-dependent timing behavior. Such attacks consist of a sequence of instructions which when executed with different data take different execution times. For example, consider the pair of instruction sequences in Listing 1. The two sequences are identical but differ in the data they operate upon. The sequence can be considered a timing vulnerability if the execution of the two sequences results in different execution times.

Listing 1: A pair of instruction sequences that have identical instructions but operate on different data. The instruction sequence is a timing vulnerability if the execution times differ.

```
1 LI   t6, 0x81321          LI   t6, 0x11235
2 LI   a7, 0xFEEEE          LI   a7, 0xFFFFF
3 LD   t4, 0(a7)            LD   t4, 0(a7)
4 ADDI a9, a5, t4           ADDI a9, a5, t4
5 LD   t5, 0(t6)            LD   t5, 0(t6)
```

The objective of our work is to develop *WhisperFuzz*, which identifies such instruction sequences and pairs of input data that result in different execution times. Furthermore, *WhisperFuzz* localizes the root cause for the differing execution times, thereby assisting in mitigation.

3 Methodology

In this section, we first explain the relationship between timing behaviors and transitions of a digital circuit. We introduce the *Micro-Event Graph* that helps capture the microarchitectural transitions at a fine granularity. We use this graph to identify the location of timing vulnerabilities and monitor the timing behaviors covered. Further, we discuss the challenges of extracting the *Micro-Event Graph* from a processor design. Finally, we give an overview of our solutions to these challenges and elaborate on each solution.

3.1 Microarchitectural Transitions and Timing Behaviors

A finite-state machine (FSM) can model a sequential digital circuit. Given an input, the circuit transitions through various states in the FSM to produce the output. Assuming each state in the FSM takes constant time, the sequence of state transitions to generate the output determines the execution time of the circuit for a given input. Thus, we claim: *two different inputs resulting in the same state transition sequences will take the same execution time.*

Consider the case study of a cache composed of multiple cache sets. Each cache set can be represented as an FSM with five states: `LookUp`, `FreeBlock`, `Replace`, `Wait`, and `Ready`, as shown in Figure 1. When a program accesses data, the cache first performs a look-up for the associated memory address in the `LookUp` state. If the data is found in the cache (cache hit), it will be directly transferred to the `Ready` state, and the processor can access the data without going to the main memory. Thus, an input address that is already present in the cache set causes transitions {`LookUp` $\rightarrow$ `Ready`} taking

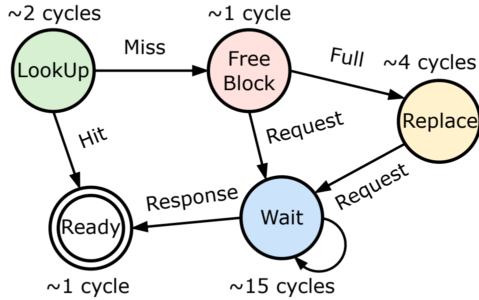


Figure 1: A finite-state machine (FSM) representation of the cache set protocol. Each state is assumed to take a constant time as shown at each node.

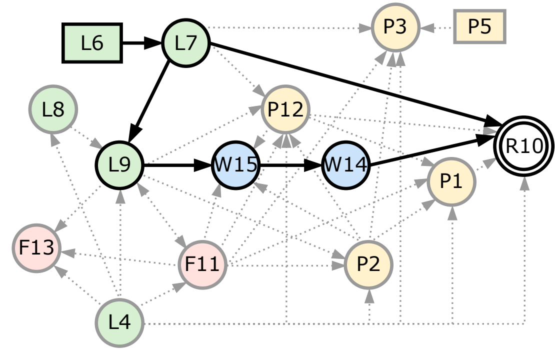


Figure 2: A high level representation of *Micro-Event Graph* (MEG) for cache set protocol represented as an FSM in Figure 1.

three clock cycles in its corresponding implementation. However, if the data is not found (cache miss), the cache set will transfer to the `FreeBlock` state to look for a free cache block to store the data. If a free block exists, the cache will transition to the `Wait` state and request the memory for the corresponding data. In case the cache does not have a free block, it will transfer to the `Replace` state, select a block for eviction based on the replacement policy, and then transfer to the `Wait` state. The cache waits in this state until it receives data from memory. It then transitions to the `Ready` state. Thus, if an input address is not present in the cache set, two FSM state transition sequences (and execution times) are possible: (i) `{LookUp → FreeBlock → Wait → Ready}`, taking 19 clock cycles if the cache has a free block or (ii) `{LookUp → FreeBlock → Replace → Wait → Ready}`, taking 23 cycles if the cache does not have a free block. Assuming each state in the FSM takes a constant execution time, a difference in the execution time of two inputs implies a difference in the sequence of state transitions followed in the FSM. However, in practice, FSM states do not always take a constant execution time. Moreover, an FSM model is abstract and cannot effectively represent complex microarchitectural details in digital circuits. Thus, an FSM representation fails to uncover timing differences arising at the microarchitectural level. Further, it makes localizing the source of timing difference difficult, delaying the mitigation process.

We introduce *Micro-Event Graphs* (MEGs) to overcome these drawbacks. A MEG models a given digital circuit using the register-transfer level (RTL) as a set of events, which we call microarchitectural events. Each microarchitectural event affects the contents of at least one element, such as a wire or a register in the digital circuit. A MEG models a given digital circuit as possible events and dependencies between these events. Each node in the MEG represents an element while a directed edge from a parent node to a child node indicates that an event on the parent element can potentially trigger an event on the child element.

Figure 2 shows a high-level representation of the MEG corresponding to the cache set protocol. The MEG consists of 15 nodes and 115 edges. Each state in the corresponding FSM (refer Figure 1) maps to one or more nodes in the MEG. For example, nodes colored in green (node labels starting with L) correspond to state `LookUp` and those colored in yellow (node labels starting with P) correspond to state `Replace`. Each execution of the cache set protocol can be mapped to at least one path in the MEG shown. Similar to the state transitions in the FSM, but at the finest granularity, an input triggers a sequence of microarchitectural events in the MEG during its execution, each taking a constant time. Thus, one can have the following observations:

P1. If two inputs to the microarchitecture result in the same event transitions in the MEG, then the execution time for the two inputs is the same.

P2. If there is a difference in the execution time of the two inputs, then the sequence of microarchitectural events followed is different for the two executions.

P3. If two inputs to the microarchitecture result in different event transitions in the MEG, then the execution time for the two inputs may differ.

3.2 Detecting and Localizing Timing Vulnerabilities in Processors: A High-Level Overview

The goal of *WhisperFuzz* is to detect and localize timing vulnerabilities in a processor design-under-test (DUT). To detect timing vulnerabilities, we leverage the strength of hardware fuzzing. The fuzzer generates an instruction sequence and at least two corresponding data inputs of the form shown in Listing 1, that take different execution times when applied to the sequence. For each pair of instruction sequences and inputs, we trace the path of events in the MEG. For example, in Figure 2, each path represents an execution corresponding to different addresses given to a `load` instruction. These paths

are then used to localize the root cause of the vulnerability. The root cause is the event prior to the first bifurcation in these two paths.

For example, consider the two highlighted paths in Figure 2. One path traces event sequence { L6, L7, R10}, while the other traces {L6, L7, L9, W15, W14, R10}. Each path corresponds to a different address given to a `load` instruction. Since the two paths are different, they may take different execution times due to **P3** (See Section 3.1), resulting in a timing vulnerability. These paths trace the same events until L7, after which they bifurcate. Hence, an event on node L7 will likely be the vulnerability's root cause. The number of paths covered in the MEG gives a notion of the coverage of timing behaviors of the DUT.

3.3 Challenges

Developing *WhisperFuzz* involves the following challenges.

C1. Generating the MEG. The MEG must capture all possible events and event transitions in a given processor DUT. This is specifically challenging in modern microprocessors due to their complexity and large code bases. We develop an automated strategy to address this challenge that extracts the MEG given the DUT's source code in RTL form. Section 3.5 elaborates on this strategy.

C2. Characterizing timing behavior of each processor module. One way to determine timing behaviors is to input instruction sequences to the DUT and measure the execution time. However, a complete processor design contains thousands of signals, complicating the localization of the vulnerability. An alternate bottom-up approach is to isolate each module in the processor, provide inputs, and measure the timing behavior of the module. However, timing differences detected may not be observable when the module is integrated into the complete processor. Therefore, we follow a two-pronged approach where we first generate instruction sequences for the complete processor to detect timing differences and localize at a module level. We present a *hierarchical analysis strategy* to prioritize the modules to be analyzed, thereby detecting vulnerabilities faster, as discussed in Section 3.6.

C3. Localizing the source of timing vulnerability. The large code space of the processor's DUT makes it challenging to locate the source of a timing vulnerability. It necessitates manual effort and a detailed understanding of the processor's microarchitecture. To address this challenge, we introduce an automated static analysis strategy on the MEG that can identify the root cause of the timing vulnerability within a few seconds. Section 3.7 elaborates on the strategy.

C4. Fuzzing the microarchitectural state space and determining coverage. Hardware fuzzing has shown its effectiveness in detecting vulnerabilities in large-scale designs, such as processors. However, the existing grey-box processor fuzzers [51] are not compatible with timing vulnerability detection. Their mutation algorithms are designed to accelerate coverage increment, but timing vulnerability detection requires sequences of data-dependent instructions that change processors' timing behaviors as seen in Listing 1. Moreover, existing fuzzers lack coverage feedback to demonstrate the timing behaviors covered. Providing such metrics can help designers decide to tape out. To address this challenge, we develop a specialized mutation algorithm that generates data-dependent instructions. We then use paths of each module's MEG as the timing coverage metric and instrument the DUT to provide coverage feedback.

3.4 The *WhisperFuzz* Framework

To address the challenges discussed in Section 3.3, we develop *WhisperFuzz* that comprises of three major modules: *Seed Generation*, *Vulnerability Detection*, and *Vulnerability Localization*, as shown in Figure 3.

Seed Generation. *WhisperFuzz* first uses the Seed Generation module that contains a coverage-feedback fuzzer [14, 16, 34, 37, 39] to explore design spaces guided by the fuzzer's internal code-coverage metric. The fuzzer utilizes the *Input Generator* to generate input instructions, which are then simulated in the fuzzer's internal *DUT Simulation* unit. The *Feedback Engine* calculates the code-coverage metric of this input. Based on this metric, the *Code Coverage Mutator* mutates the input instructions to improve the coverage. The *Input Database* records various fuzzer-generated inputs and the corresponding code-space covered.

Vulnerability Detection. Based on the *Feedback Engine*, *WhisperFuzz* identifies a sequence of instructions that explore new design spaces as seeds and sends them to the Vulnerability Detection module. The *Operands mutator* in the Vulnerability Detection module mutates the data in the instruction sequence to trigger varying timing behaviors (See Section 3.8). These mutated instruction sequences are simulated in the *DUT Simulation* unit of the Vulnerability Detection module. The *Leakage analyzer* then compares simulation traces and detects timing vulnerabilities (See Section 3.6).

Vulnerability Localization. *WhisperFuzz* invokes the *Preprocessor* in Vulnerability Localization to extract *Micro-Event Graph* of modules (See Section 3.5) and instruments potential timing behaviors based on the *Micro-Event Graph*. For the timing vulnerabilities identified by the *Leakage analyzer*, *WhisperFuzz* invokes the *Diagnozer* to pinpoint the cause of these timing vulnerabilities (See Section 3.7). The *Coverage analyzer* collects simulation traces of all inputs from the Vulnerability Detection module and calculates the timing behaviors covered.

The *WhisperFuzz* framework repeats these steps until there is a timeout, or the fuzzer and *Operand mutator* completely cover the timing behaviors of the DUT.

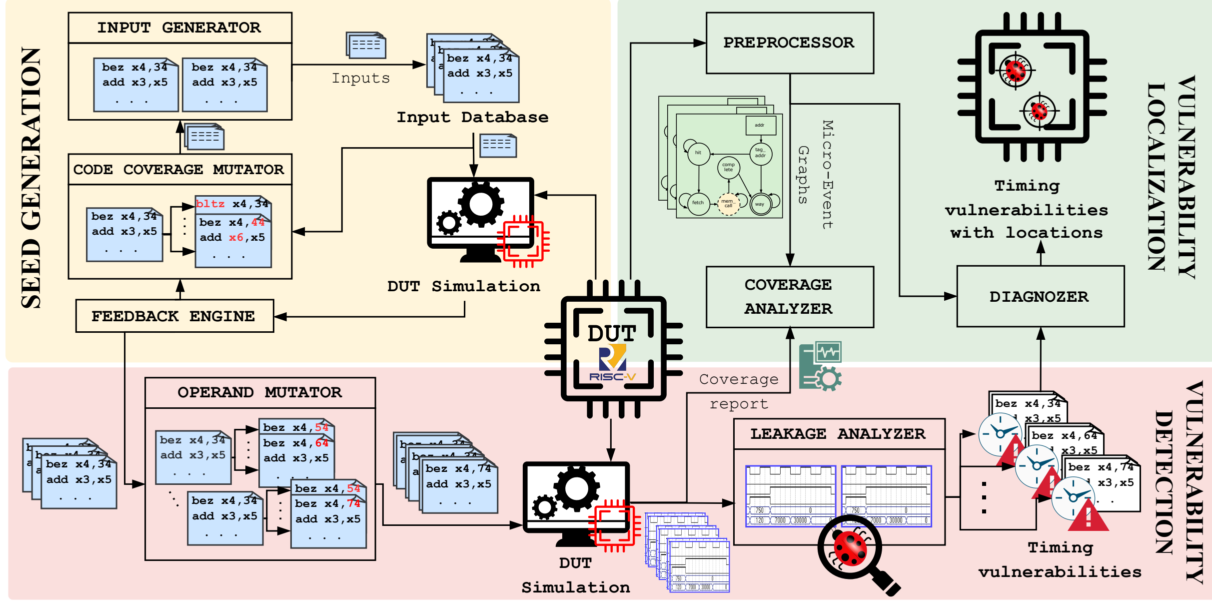


Figure 3: The *WhisperFuzz* framework. It includes three key modules. First, the Seed Generation module internally utilizes a coverage-feedback fuzzer to explore the design space. The generated inputs are recorded in a database. Mutations are performed to improve code coverage. Second, the Vulnerability Detection module uses the generated seed, mutates the instruction operands, and identifies the timing vulnerabilities based on DUT simulations. Finally, the Vulnerability Localization module pinpoints the locations of uncovered vulnerabilities.

3.5 Extracting Micro State Transitions

In this section, we elaborate on the MEG used to address challenge C1. Using a *Preprocessor*, *WhisperFuzz* parses the RTL code of a given design to generate its MEG representation. Each node in the MEG represents an event corresponding to an element in the RTL, while an edge indicates that there exists an event on the parent node that can trigger a change in the value of the child node. Edges between nodes are also annotated with conditions required to trigger the event and the RTL line number which causes the dependency between the event. Nodes in the MEG can be categorized into sequential nodes or combinational nodes corresponding to the sequential or combinational nature of the hardware element it represents [29]. Formally, we define a MEG as:

Definition 1 *Given the RTL of a module D in a DUT, the corresponding micro-event graph is defined as $\mathbb{G}(D) = (S, \Sigma)$, where the nodes are denoted by $S = \{s_1, s_2, \cdots, s_n\} = \{S_q \cup S_c \cup S_I \cup I \cup O\}$ and include all elements in D; S_q and S_c represent the sequential and combinational elements respectively; S_I represent external modules instantiated in D; I and O are the set of inputs and outputs respectively to the module D. The set Σ represents the edges in the graph, $\mathbb{G}$, such that $(s_i, s_j) \in \Sigma$ if a change in s_i may trigger a change in s_j as per the RTL. Each edge is annotated with the condition required for the change to occur.*

Listing 2: Simplified Verilog code of cache set protocol (refer Figure 1).

```
1  input addr; output way;
2  reg hit, full, valid, fetch;
3  wire tag_addr, complete;
4  assign tag_addr = addr[tag_bits];
5  always@(posedge clock) begin
6      if tag_addr == tag
7          way <= index; hit <= 1;
8      if hit == 0 && full != 1 && valid == 1 begin
9          fetch <= 1; temp_way <= index;
10     if fetch == 1:
11         // Call mem_call for fetching block from next level.
12         // Sub-instance sets complete signal
13         mem_call(addr, complete);
14     if complete:
15         way <= temp_way;
16 end
```

Consider the cache set protocol example given in Section 3.1, with its corresponding simplified Verilog code as in Listing 2. A sub-graph of its MEG is shown in Figure 4, denoting input node `addr` with an incoming edge. Due to the `assign` statement on Line 4, a transition in the value of `addr` triggers a change in the value of `tag_addr`. To denote this dependency, the graph contains a directed edge from the node `addr` to `tag_addr`. Similarly, the value of `tag_address` can influence the value of `way` and `hit` if the condition (`tag_address == tag`) on Line 6 holds. Hence, the edges from `tag_address` to `way` and `hit` are annotated with the condition. The sub-instance call on Line 13 corre-

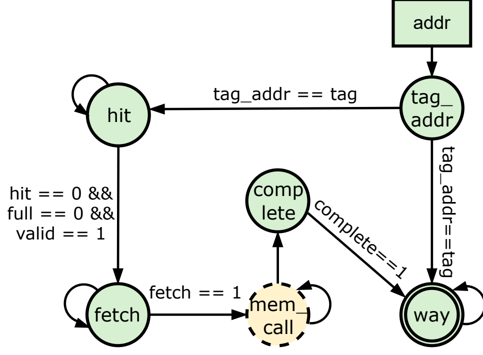


Figure 4: Sub-graph extracted from *Micro-Event Graph* in Figure 2 of the cache set protocol case study. The L6, L7, L9, W15, W14 and R10 nodes from Figure 2 correspond to `addr`, `tag_addr`, `hit`, `mem_call`, `complete`, `way` respectively.

sponds to the dashed node `mem_call` in the MEG. The accept state from node `way` indicates that it references an output signal. Appendix A provides a further discussion on the implementation details of MEG.

The *Preprocessor* parses the RTL description of the design, translating it into the corresponding graphical representation using Algorithm 1. Given an RTL module D, the *Preprocessor* first extracts into $\mathbb{G}(D).X$ where $X = I, O, S, \Sigma,$ or S_I corresponds to the set of inputs, outputs, nodes, edges, and sub-instance nodes respectively (Lines 3 and 4). Finally, the *Preprocessor* iterates over each line in the RTL, identifies destination signals (Line 6) and operand signals (Line 7), and adds an edge between each operand signal and the destination signal (Line 9). The *Diagnozer* (Section 3.7) and *Coverage Analyzer* (Section 3.8) of *WhisperFuzz* utilize this MEG for further analysis. *WhisperFuzz* generates *Micro-Event Graph* for each module individually to counteract the vast design complexity of processors. The comparitively reduced permodule complexity consumes a reasonable computation cost and resource utilization as shown in Table 3.

Micro-Event Path. We define a *Micro-Event Path* (MEP) as a path that starts at the input node and traces connected nodes until it reaches the output node. Given an input, this path traces the sequence of events triggered in the module. Formally,

Definition 2 *A sequence of directed edges,* $P = \langle(s_1,s_2),(s_2,s_3),\cdots,(s_{n-1},s_n)\rangle$ *where* $(s_i,s_j) \in \Sigma, \forall(i,j)$ *and* $s_1 \in I$ *and* $s_n \in O$*, is a Micro-Event Path in* $\mathbb{G}(D)$.

Figure 4 notes two different MEPs from the input node `addr` to the output node `way`. Each path can be mapped to a path in the FSM in Figure 1, but at a finer granularity. In case of a cache hit, the module follows the FSM path {`LookUp` $\rightarrow$ `Ready`} mapped to MEP {`addr` $\rightarrow$ `tag_addr` $\rightarrow$ `way`}. In case of a cache miss, if the cache set has a free cache line, the module follows the FSM path {`LookUp` $\rightarrow$ `FreeBlock` $\rightarrow$ `Wait` $\rightarrow$ `Ready`} mapped to {`addr` $\rightarrow$ `tag_addr` $\rightarrow$ `hit` $\rightarrow$ `fetch` $\rightarrow$ `mem_call` $\rightarrow$ `complete` $\rightarrow$ `way`}. A comparison of these two differing MEPs indicates the RTL wire `tag_addr` as the hardware element responsible for the divergence in the two paths. *WhisperFuzz* utilizes this information to trace the root cause of the vulnerability, as elaborated in Section 3.7.

Algorithm 1: Preprocessor in Vulnerability Localization module of *WhisperFuzz*.

Input : D `// RTL code of module`
Output : $\mathbb{G}$ `// MEG of module`

1 $\mathbb{G}(D).I \leftarrow \phi$; $\mathbb{G}(D).S_I \leftarrow \phi$; $\mathbb{G}(D).O \leftarrow \phi$
2 $\mathbb{G}(D).S \leftarrow \phi$; $\mathbb{G}(D).\Sigma \leftarrow \phi$
3 $\mathbb{G}(D).I$, $\mathbb{G}(D).S_I$, $\mathbb{G}(D).O \leftarrow$ GETINOUTPUTS(D)
4 $\mathbb{G}(D).S \leftarrow$ GETSIGNALS(D)
`/* Iterate RTL code */`
5 **for** ***line*** $\in D$ **do**
`// Get an operand set that can change the value of destination signal`
6 $s_2 \leftarrow$ GETDESTINATION(*line*)
7 $s_1 \leftarrow$ GETOPERANDS(*line*)
8 **for** $s_1 \in S_1$ **do**
`// Add an edge between each operand and the destination signal`
9 $\mathbb{G}(D).\Sigma \leftarrow \mathbb{G}(D).\Sigma \cup \{s_1, s_2, line_no\}$
10 return $\mathbb{G}$

3.6 Characterizing Timing Behaviors

Existing hardware fuzzers detect timing vulnerabilities by measuring the execution time of an entire processor through performance counters [35,51,66,67]. *WhisperFuzz*, however, requires finer information to localize the timing vulnerabilities. Therefore, *WhisperFuzz* computes the execution time taken by each module within the processor. For this purpose, we use simulation of inputs generated by the *Seed Generation* and mutated by the *Operand Mutator*.

For each simulation of the DUT, the *Simulator* generates a set of simulation traces corresponding to each signal within the DUT. A simulation trace is a time series that records all transitions of a signal during an execution. The *Leakage Analyzer* then selects the subset of interesting traces that can lead to a vulnerability. These traces are snipped at the last clock cycle at which any signal within a selected module instance toggles. The duration of this constrained trace is then computed as the execution time of the module instance. The *Leakage analyzer* analyses the timing behaviors of each module instance in a DUT to identify the module instance with timing vulnerabilities.

Hierarchical leakage analysis. Modern processors, however, consist of hundreds of module instances with various inter-module dependencies [9, 73, 74]. Analyzing all their timing behaviors is time-consuming and computationally expensive. Hence, we require a staggered approach for leakage analysis, which prioritizes modules based on their dependencies. A different module-specific execution time indicates that the source of the vulnerability either originates in the current module or a module lower in the hierarchy. For example, the memory control unit (MCU) controls the data accesses in the cache. Upon storing the data at a memory address, the cache sends the `Ready` signal to the MCU. Thus, if a timing difference is in the cache delaying the `Ready` signal, the MCU will also observe the timing difference.

The *Leakage Analyzer* implements a hierarchical method taking a bottom-up approach. In this approach, we map the dependencies between modules and place these modules into various hierarchical levels such that the lower-level modules are sub-instances of a higher-level module. The hierarchical leakage analysis is then carried out incrementally from the lowest module to the highest module. For example, in `BOOM` [74] there are 431 module instances, when placed hierarchically they constitute 10 levels. Thus, `BOOM` analysis starts from the tenth level and moves upwards until the top module. This approach also ensures the detection of timing vulnerabilities in all lower levels before detection in a higher level module instance.

Once the *Leakage analyzer* identifies a module instance with execution time differences caused by a pair of inputs, it sends the module instance and the corresponding simulation traces to *Diagnozer* to pinpoint the location of timing differences further.

3.7 Localizing the Source of Timing Vulnerabilities

For a pair of inputs exhibiting a difference in execution time in a particular module instance, the *Diagnozer* localizes the source of the timing difference within the design RTL. The *Diagnozer* takes as input two sets of simulation traces, each corresponding to an execution of the DUT with different data inputs. It operates in two phases: (i) identifying the element causing the divergence and (ii) mapping the cause in the RTL source code of the processor.

Identifying the element causing the divergence. For the given pair of inputs, the *Leakage Analyzer* generates a set of simulation traces corresponding to each input, ST_1 and ST_2 for the module under examination, D. If each trace in the two sets are exactly identical, then the execution time for the two inputs are equal. On the other hand, if the execution time of the two inputs are different, there exists a subset of the traces within the two sets that differ. In this subset of simulation traces, the trace which first deviates, corresponds to the combinational

Algorithm 2: The Diagnozer in *WhisperFuzz* to locate the timing vulnerabilities in the DUT.

Input : (ST_1, ST_2) `// Pair of simulation traces`
1 $\mathbb{G}(D)$ `// MEG of module under examination`
Output : (V_s, L) `// Set of signals identified as the cause of the timing difference and corresponding line numbers in RTL,`
2 $V_s \leftarrow \phi$; $temp_V_s \leftarrow \phi$; $L \leftarrow \phi$
`/* Phase 1 */`
3 **for** *every clock cycle (clk) in* ST_1 **do**
4   **for** $s \in \mathbb{G}(D) \cdot S$ **do**
5   **if** $s \in ST_1[clk] \neq s \in ST_2[clk]$ **then**
6             $temp_V_s \leftarrow temp_V_s \cup \{s\}$
7   **if** $temp_V_s$ *is not empty* **then**
8   break
`/* Phase 2 */`
9 **for** $temp_V_s$ *is not empty* **do**
10     $new_V_s \leftarrow \phi$
11   **for** $s \in temp_V_s$ **do**
12   **for** $(s, s_{child}) \in \mathbb{G}(D) \cdot \Sigma$ **do**
13   **if** s_{child} *is sequential* **then**
14                 $V_s \leftarrow V_s \cup \{s_{child}\}$
15                 $L \leftarrow L \cup \{line_no\}$`// RTL line number`
16   **else**
17                 $new_V_s \leftarrow new_V_s \cup \{s_{child}\}$
18     $temp_V_s = new_V_s$
19 return (V_s, L)

element that instigates the timing difference. Algorithm 2 shows the process of the *Diagnozer*. In the first phase, the *Diagnozer* iterates through every clock cycle of simulation traces and finds the first signals which differ in the two sets of traces (Lines 6,7). It creates the list of signals $temp_V_s$ whose traces first differ between ST_1 and ST_2.

Mapping the cause in the source code. In a module, a timing difference occurs because some hardware elements take varying clock cycles based on the input. The Sequential element influences the number of clock pulses for execution. Hence, to trace the source of the timing difference within the module, the *Diagnozer* traces dependencies from the identified combinational signals to the sequential nodes in the MEG.
In Algorithm 2, this tracing is done by a Breadth First Search to identify the subsequent sequential elements which originate from each signal in $temp_V_s$. This is stored in the list V_s along with line numbers where the corresponding event on the sequential element is found in the RTL of D.

3.8 Fuzzing Microarchitectural State Space

WhisperFuzz introduces two components *Operand mutator* and *Coverage analyzer* to generate inputs for timing vulnerability detection and to monitor timing behaviors explored.

Operand mutator consists of specialized mutation algorithms to generate inputs for detecting timing vulnerabilities in processors. Detecting vulnerabilities requires changing the DUT's timing behaviors by triggering different microarchitectural state transitions [35, 51, 67]. Moreover, a valid timing side-channel is data/memory-dependent [8, 24, 48, 64, 72]. Therefore, we constrain *Operand mutator* to mutate only the segments of a test that will change the memory or data values.

Processor fuzzers use sequences of instructions to verify DUTs, as mentioned in Section 2.1. An instruction contains `opcode` and `operand` fields [32, 52]; both of which cause the microarchitectural state transitions. The `opcode` fields determine the instruction's operation. Meanwhile, the *operand* fields provide the source and destination registers (i.e., general-purpose registers (GPRs) and control and status registers (CSRs)) and immediate values/memory addresses. Some instructions contain immediate memory addresses only (e.g., branch operation instructions). To generate data/memory-dependent inputs, we constrain *Operand mutator* to mutate only the immediate values/memory addresses of instructions uniformly at random [32, 55]. To further increase the mutation space, we assign random values to GPRs and valid memory addresses during the processor's initialization.

However, replacing the original mutation algorithms, *Coverage Mutator*, with *Operand mutator* reduces the fuzzer's efficiency in exploring the design space for timing vulnerabilities. This is because the mutator will not change the `opcode` and register-operands of instructions. Therefore, to maintain the efficacy of design space exploration and guarantee the effectiveness of timing vulnerability detection simultaneously, we use *Coverage mutator* to explore design spaces and use *Operand mutator* to exploit time side-channels near the design spaces explored. Any tests achieving new code coverage will be identified as seeds for *Operand mutator*. For each seed, the *Operand mutator* will generate multiple data-dependent inputs, aiming to trigger the microarchitectural state transitions that will cause timing differences compared to the seed.

Coverage analyzer monitors timing behaviors covered by mapping microarchitectural state transitions to executed paths. For a given DUT, various paths exist between the input nodes and output nodes of the corresponding MEG. For an input value to the DUT, tracing the execution path followed and mapping it to the MEP poses a challenge.

To address this challenge, we utilize SystemVerilog Assertion (SVA) properties known as `cover` properties [29]. If an `cover` property evaluates to true for a given input, the DUT enters a stage during simulation when the property holds. Each graphical path is converted to a `cover` property using the annotated edge conditions and the timing behavior of the node. The timing behavior of a node depends upon the type of node defined. Algorithm 3 shows the process of generating the conditions of a MEP, $P \in \mathbb{G}(D)$, as the expression of a `cover` property. As combinational nodes are modeled after combinational logic, events occurring on these nodes complete instantaneously (line 4), while those occurring on sequential nodes complete on the next clock edge (line 6). If the value of a signal is from other modules, including input signals and signals connected with subinstances, we assume the events will *eventually* happen (line 8). Appendix B shows the `cover` properties for MEPs in the example cache set.

Properties representing all possible paths in the MEG corresponding to the module are instrumented in the RTL. The *DUT Simulation* unit of the Vulnerability Detection module takes as input this modified RTL. The *Coverage Analyzer* utilizes the results of the resultant assertion report for calculating the coverage of the various graphical paths.

Algorithm 3: Coverage analyzer

```
   Input  : P // Micro-Event Path
1  G(D) // MEG of module D
   Output : Condition // Assertion Property
            corresponding to Micro-Event Path P
2  Condition ← ⟨⟩ // Empty Sequence
3  for (s_i1, s_i2) ∈ P do
4     if (s_i1, s_i2) has branch then
5        Condition ← Condition||⟨branch⟩
6     if s_i2 ∈ S_q then
7        Condition ← Condition||⟨1 cycle⟩
8     if s_si1 ∈ S_I then
9        Condition ← Condition||⟨eventually⟩
10 return Condition
```

4 Evaluation

We evaluate *WhisperFuzz* on three most advanced open-sourced processors based on RISC-V [52] instruction set architecture (ISA). We first demonstrate the new vulnerabilities detected by *WhisperFuzz* and provide statistical analysis to prove the existence of the timing side channels. We then leverage the power of *Micro-Event Graph* of *WhisperFuzz* to identify the root causes of these vulnerabilities and evaluate the efficiency of our framework, as shown in Table 2. Finally, we evaluate the timing behaviors covered by fuzzing.

4.1 Evaluation Setup

Benchmark selection. Most commercial processors are protected intellectual properties without available source code. Thus, we pick the three large (in terms of the number of gates) and widely-used open-sourced processors: `Rocket Core` [9],

BOOM [74], and CVA6 [73] from the RISC-V ISA. Most recent hardware security tools are evaluated using these processors [16, 34, 37]. The CVA6 and BOOM are more complex compared to the Rocket Core. They possess advanced microarchitectural features such as out-of-order execution and support single instruction-multiple data (SIMD) execution.

Evaluation environment. We use the industry-standard tool, Synopsys *VCS* [3] for DUT simulation. We convert timing behaviors of RTL modules into SystemVerilog Assertion (SVA) cover properties and instrument them into DUTs. We use *Chipyard* [7] environment for the processors. We collect the coverage report and simulation traces from *VCS* to analyze the timing coverage and timing behaviors, respectively.

Fuzzing setup. We use *HyPFuzz* [16] to generate the seeds for *Operand mutator*. Other processor fuzzers that generate sequences of instructions as inputs are also compatible [14, 26, 34, 37, 39, 57, 71]. *HyPFuzz* is a state-of-the-art hardware fuzzer that combines fuzzing and formal tools to maximize coverage and speed up design exploration. *HyPFuzz* is also compatible with various coverage metrics. We use a combination of branch, condition, and FSM metrics for code coverage. Branch and condition metrics monitor the combinational logic of DUTs. The FSM metric monitors the sequential logic of DUTs [3, 16, 37]. Therefore, any new points covered by inputs represent at least one new microarchitectural state transition triggered. We collect these inputs as seeds and use the *Operand mutator* to generate 200 data-dependent inputs for each seed. We ran the entire fuzzing process for 72 hours, and repeated it thrice to collect coverage results.

4.2 Detecting Novel Side Channels

This section will discuss 12 new timing vulnerabilities found by *WhisperFuzz*. Furthermore, Appendix C contains the proof of concept code for mentioned vulnerabilities.

DIVUW + REM in BOOM [74]. The side channel under consideration pertains to the consecutive execution of the DIVUW and REM instructions. In Figure 5d, the operational characteristics of this side channel are graphically depicted across various operand executions. Our analysis revealed median discrepancies of 48 cycles, 35 cycles, and 83 cycles between the divisor equal to 0, 1, and greater than 1 distributions respectively, with a maximum deviation of 101 cycles observed when deliberately selecting operands to maximize the disparity. Establishing a threshold for the median separation facilitates a successful discrimination rate of 100% when distinguishing between binary states 0 and 1.

DIVUW in CVA6 [73]. The side channel pertains to the execution of DIVUW instruction. Figure 5a illustrates the operational characteristics of this side channel across multiple operand executions. Notably, our analysis has unveiled median discrepancies of 14 cycles, 39 cycles, and 54 cycles among the divisor equal to 0, 1, and greater than 1 distributions respectively. Furthermore, we have observed a maximum deviation of 56 cycles when deliberately selecting divide by zero to maximize the disparity. Establishing a threshold for the median separation enables the successful discrimination of binary states 0 and 1 with a 100% accuracy rate. However, our findings indicate that attackers leverage this channel to transmit three states rather than the intended two states by defining two thresholds, achieving a success rate of 91%.

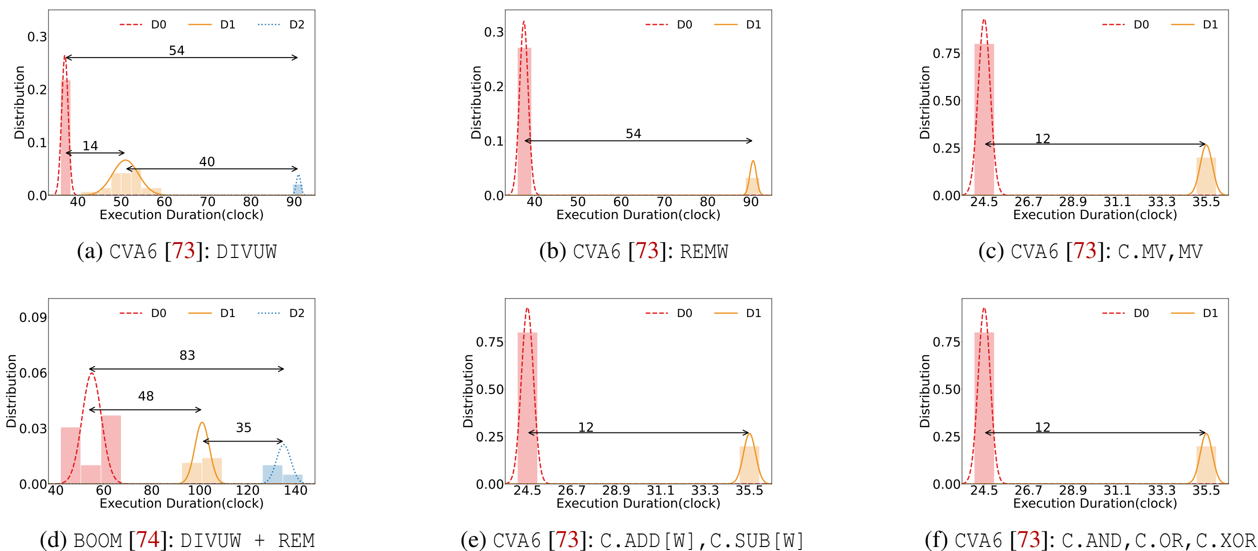
REMW in CVA6 [73]. *WhisperFuzz* detected this novel side channel when fuzzing CVA6 with REMW instruction. Figure 5b shows the operational characteristics of this side channel across multiple instances of operand executions. We demonstrate a median difference of 54 cycles between the divisor equal to 0 and greater than 0 timing behavior distributions. Additionally, when we select zero as a divisor in the REMW instruction to maximize the timing difference, the maximum deviation is 56 cycles. Attackers can use this channel by establishing a threshold for the median separation to transmit binary states 0 and 1 with 100% accuracy.

C.ADD[W], C.SUB[W], C.AND, C.OR, C.XOR, and [C].MV in CVA6 [73]. *WhisperFuzz* detected multiple novel side channels that occur when executing compressed RISC-V instructions, i.e. C.ADD[W], C.SUB[W], C.AND, C.OR, C.XOR, and [C.]MV. RISC-V Zkt contract has classified these instructions as serious security vulnerabilities if the instructions are data dependent [2]. The MV instruction has the same timing behavior as its compressed version, i.e., C.MV and causes a timing channel. Figures 5e, 5f, and 5c, show the operational characteristics of these side channels across multiple instances of operand executions. We demonstrate a median difference of 12 cycles between the second operand equal to 0 and greater than 0 timing behavior distributions. Selecting a specific value of zero as the operands of these instructions results in 12 more cycles compared to any other operand values. Establishing a threshold for the median separation allows for the successful discrimination between binary states 0 and 1 with an accuracy rate of 100%.

4.3 Redetecting Known Side Channels

DIV in BOOM [74]. *WhisperFuzz* successfully generated test cases that exposed timing side-channel vulnerabilities related to division instructions (DIV), a vulnerability disclosed in *SIGFuzz* [51] for the BOOM processor. *WhisperFuzz* generated multiple test cases featuring the DIV instruction, and by systematically mutating the input values during the fuzzing process, it revealed variations in the number of clock cycles required for the DIV instruction to complete its operation. Further investigation elucidated that when the divisor was bigger than the dividend, the division unit necessitated more time to conclude the division process.

SC in Rocket Core [9] and BOOM [74]. *WhisperFuzz* also identified a timing side-channel associated with Store-Conditional operations, effectively diagnosing timing dispari-



(a) CVA6 [73]: DIVUW (b) CVA6 [73]: REMW (c) CVA6 [73]: C.MV,MV

(d) BOOM [74]: DIVUW + REM (e) CVA6 [73]: C.ADD[W],C.SUB[W] (f) CVA6 [73]: C.AND,C.OR,C.XOR

Figure 5: Timing behaviour of detected novel side-channels.

Listing 3: Source location of **DIVUW** in CVA6 [73].

```
13512    state_q <= state_d;
13513    op_a_q <= op_a_d;
13514    op_b_q <= op_b_d;
13515    res_q <= res_d;
13516    cnt_q <= cnt_d;
13517    id_q <= id_d;
13518    rem_sel_q <= rem_sel_d;
13519    comp_inv_q <= comp_inv_d;
13520    res_inv_q <= res_inv_d;
13521    op_b_zero_q <= op_b_zero_d;
13522    div_res_zero_q <= div_res_zero_d;
```

ties resulting from the presence of the dirty bit in the data cache implementation. The vulnerability was discovered in *SIGFuzz* [51] for the Rocket Core and BOOM processors. The test cases feature a SC instruction containing at least one subsequent load instruction. Through the mutation of these test cases, we were able to detect timing discrepancies when the load/store module attempted to access an address not present in the cache. Subsequently, the *Diagnozer* of *WhisperFuzz* pinpointed the root cause of this timing difference, attributing it to the dirty bit that was set by the Store-Conditional instruction for a cache line.

4.4 Pinpointing the Locations of Side Channels

We apply *Diagnozer* (See Section 3.7) to identify the root causes of detected timing vulnerabilities (See Sections 4.3, 4.2). We describe, in detail, the *Diagnozer* results for two novel and two known vulnerabilities. The results for all detected vulnerabilities are summarized in Table 2.

DIVUW in CVA6 **[73].** In CVA6, given the instruction sequence in Listing 7, the *Diagnozer* identifies the source of the vulnerability in module serdiv in the lines highlighted in Listing 3. The first phase of the *Diagnozer* identifies 27 signals as the instigating signals, while the second phase pinpoints that these signals change the values of 8 sequential signals.

Compressed Instructions in CVA6 [73]. Given two trace files corresponding to the instruction sequence in Listing 9 the first phase of the *Diagnozer* identifies the sources of these vulnerabilities in the ALU module. The exact RTL lines are as highlighted in Listing 14. Though the ALU module does not have a sequential component, the effects of its outputs are propagated to the inputs of the other modules thereby influencing further execution.

DIVUW + REM in BOOM [74] and Rocket Core [9]. Given two trace files corresponding to the instruction sequence in Listing 6, the *Diagnozer* identifies the source of this vulnerability in BOOM [74] and Rocket Core [9] at different lines in module MulDiv. The vulnerability is localized to the sequential elements divisor, state, negout in both processors (refer Table 2).

Division by zero in BOOM [74], Rocket Core [9], CVA6 [73]. Given two trace files corresponding to the instruction sequence in Listing 6 with the divisor set to 0, the *Diagnozer* identifies the sources of this vulnerability in BOOM, Rocket Core, CVA6 as shown in Table 2. Though the root cause is localized to the same module in Rocket Core and BOOM, the pinpointed lines differ. While in CVA6, the divide by zero vulnerability is localized to module serdiv.

Hence, though the same timing vulnerability can affect multiple DUTs, due to the differences within the microarchitectural design, the source of the vulnerability in the RTL code differs. Furthermore, consider the two division-related

vulnerabilities detected in CVA6 [73]. Although these vulnerabilities affect the same module (serdiv), the source of the vulnerabilities within the module differs. Hence, the automated localization of vulnerability sources performed by *WhisperFuzz* is beneficial and efficient.

4.5 Coverage Analysis

We use *Coverage analyzer* to monitor timing behaviors explored by inputs from *Operand mutator*. *Coverage analyzer* converts paths of *Micro-Event Graph* of an RTL module into cover properties and instrument these properties in to DUT. After simulation, *Coverage analyzer* collects assertion results to calculate the timing behaviors covered (See Section 3.8). Figure 6 shows the coverage achieved in various modules of the three benchmarks. Due to the space limitation, we only show the coverage of 10 modules with the highest total number of timing coverage points. On average, *WhisperFuzz* achieves 39.57%, 33.16%, and 20.20% coverage on CVA6 [73], Rocket Core [9], and BOOM [74], respectively.

Since we emphasize covering timing behaviors, our *Operand mutator* generates 200 inputs for each seed generated by the coverage-feedback fuzzer. The overall coverage of *WhisperFuzz* can be increased by using more seeds from coverage-feedback fuzzer and running the fuzzer for more time. Also, many coverage points are left uncovered due to the multiple configurations of DUT. For example, CVA6 has parameters to configure its float-point unit to support operations from 8-bit to 128-bit [73]. *WhisperFuzz* fuzzes CVA6 in its default configuration where CVA6 uses 64-bit operations. Hence, all other operations' timing behaviors are uncoverable. Rocket Core's floating-point unit [9] and BOOM's branch predictor [74] have similar configurations. Therefore, these configurations reduce the overall coverage.

4.6 Exploitability of Detected Vulnerabilities

In this section, we discuss the potential exploitations of the vulnerabilities detected by *WhisperFuzz* as presented in Section 4. Such vulnerabilities can be exploited for information leakage across diverse scenarios as described below.

Covert Channels. A timing covert channel breaks the process isolation guarantees provided by the hardware. A *sender* process can perform operations influencing the execution time of a *receiver* process, which infers a bit value based on this observed timing. For instance, the DIVUW-based vulnerability detected by *WhisperFuzz* in CVA6 [73] can be employed to design a covert channel based on the timing differences. However, realizing such a covert channel requires the communicating processes to execute on the physical core using hyperthreading features which are unavailable on our evaluation processors [9, 73, 74].

Speculative Execution Attacks. Such attacks happen in out-of-order processors when, during the rolled back of speculatively executed instructions, processor leave their footprints on the micro-architectural components such as the cache. This has been exploited in several popular attacks [1, 38, 40, 51]. An attacker can formulate a similar attack with the vulnerabilities found by *WhisperFuzz*. For instance, with speculative execution support on CVA6, a combination of the load and time-dependent instructions(any instruction that is detected by *WhisperFuzz*, See Section 4.2) can utilized to encode sensitive data into the cache, which a cache timing attack can then glean. However, the current state of the evaluated processors is limited to non-speculative execution.

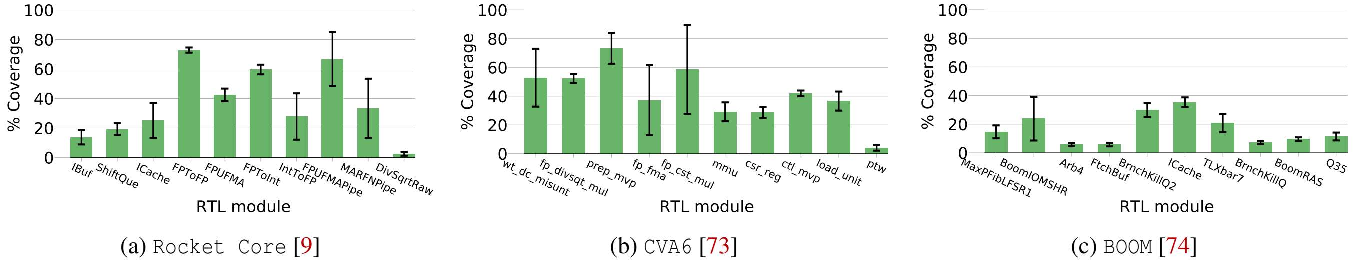
Attacking Library Implementations. An adversary can exploit the timing differences based on the operand values discovered by *WhisperFuzz* to glean sensitive information from popular libraries in cryptography or machine learning domains. For such an attack, the library implementation is required to have the same instruction flow, e.g., DIVUW followed by REM for BOOM [74] or C.ADD for CVA6 [73] with the operand dependent on a secret value.

Table 1: Comparison with prior works on timing vulnerability detection on processors with *WhisperFuzz*. (*N.A.: Not applicable, TSC: Timing side channel.*)

Paper	Manual effort	Scalable	Design source	Timing vulnerability	Coverage	Root cause analysis
***UPEC* [23]**	✓	✗	RTL	Covert Channels	N.A.	✗
***Fadiheh et al.* [22]**	✓	✗	RTL	Covert Channels	N.A.	✗
***Checkmate* [59]**	✓	N.A.	Abstract model	Cache TSC	N.A.	✗
***Osiris* [67]**	✗	✓	Black-box	Eviction-based TSC	N.A.	✗
***ABSynthe* [28]**	✗	✓	Black-box	Contention-based TSC	N.A.	✗
***PLUMBER* [35]**	✓	✓	Black-box	Variants of cache TSC	N.A.	✗
***SIGFuzz* [51]**	✗	✓	RTL	TSC	✗	✗
WhisperFuzz	✗	✓	RTL	TSC	✓	✓

5 Related Work

Existing state-of-the-art techniques for timing side-channel vulnerability detection primarily employ formal approaches [22, 23, 25, 59, 65] and fuzzing [28, 35, 51, 67] techniques. However, these approaches still exhibit critical shortcomings. In contrast to *WhisperFuzz*, these approaches fail to pinpoint the root causes of the detected timing vulnerabilities without manual efforts that take a long time. Thus, the mitigation based on these techniques is coarse-grained rendering them inefficient in terms of the computational resources in DUT. Further, the coverage metrics used by these solutions, such as hardware performance counters or code coverage [35, 51, 67] do not capture the timing behaviors of the DUT, resulting in uncertainty prior to tape-out [27, 36, 58]. In this section, we



(a) `Rocket Core` [9] (b) `CVA6` [73] (c) `BOOM` [74]

Figure 6: Timing coverage of RTL modules in `CVA6` [73], `Rocket Core` [9], and `BOOM` [74]. The black line indicates the variation in coverage offered.

discuss these perform a comparative analysis with *WhisperFuzz*, as illustrated in Table 1.

Formal approaches for timing vulnerability detection. ***UPEC*** [22,23] is a white-box approach to detect side channels in RISC-V RTL designs using SAT-based bounded model-checking. However, such an approach is not scalable to complex processor designs. Alternatively, ***Checkmate*** [59] employs *micro-happens-before* graphs to analyze transient execution vulnerabilities and timing side channels. It detects patterns within these graphs to assess the susceptibility of architectural models to timing side-channel threats. In contrast to our methodology, *Checkmate* relies on matching patterns of vulnerable instructions, while *WhisperFuzz* is semantically oriented and automatic.

Fuzzing-based approaches for timing vulnerability detection. ***Osiris*** [67] is a black-box fuzzer that identifies timing vulnerabilities in commercial processors by brute-forcing different combinations of instruction sequences. However, to reduce the search space, it limits the instruction sequence length to one, leaving vulnerabilities requiring multiple instructions [49] or specific operands [48] to trigger undetected. ***ABSynthe*** [28] and ***PLUMBER*** [35] identify combinations of instructions that trigger microarchitectural timing side-channel leakages by deriving a *leakage template*. However, *PLUMBER* requires manual efforts to specify mutation algorithms and potential behaviors of a DUT to generate this template. Further, it is limited to the existing cache module and cannot locate them in the DUT.

SIGFuzz [51] is a grey-box fuzzer that detects the existence of timing vulnerabilities in processors at the RTL. It generates combinations of instructions to identify cycle-accurate microarchitectural timing side-channels. However, replacing instructions can create additional architectural differences, such as differences in general-purpose registers, resulting in a high rate of false positives. Further, ***SIGFuzz*** suffers from limitations in pinpointing vulnerability locations and coverage metrics as the black-box approaches.

WhisperFuzz addresses these limitations of existing works by providing a novel white-box fuzzer with static analysis to detect and pinpoint timing vulnerabilities in executed test-cases in processors enabling fine-grained mitigations. *WhisperFuzz* is scalable to complex designs and end-to-end automated with a specialized coverage metric for timing behaviors.

6 Discussion

Use of *Code coverage mutator* and *Operand mutator*. *WhisperFuzz* employs both the *Code coverage mutator* and the *Operand mutator* to explore the design space and generate data-dependent inputs, respectively. In our experiments, we set up the use of these two mutators heuristically. The determination of their utilization is an optimization problem that can aid the efficacy of vulnerability detection. We can model the probability of *Operand mutator* covering a timing behavior [53, 75]. When this probability falls below a threshold, the *Code coverage mutator* can be called to generate new seeds. However, such an analysis is beyond the scope of this paper.

Port scanning vulnerabilities. Contention for a port in the architecture can cause execution delays, enabling attackers to create a high-resolution time side-channel by scanning for port contention [6, 12]. However, these attacks depend on the high bandwidth of shared resources and, primarily, the simultaneous multithreading architecture [6, 12, 60]. The current open-sourced benchmarks lack such advanced architectures [9, 73, 74], and hence detecting port scanning is outside the scope of *WhisperFuzz*.

7 Conclusion

Recent hardware fuzzers have showcased their potential to identify timing vulnerabilities in intricate designs, such as processors. However, the existing black-box or grey-box fuzzing approaches fall short in pinpointing the precise location or root cause of timing vulnerabilities. Further, these approaches lack the necessary coverage feedback mechanisms for the exploration of timing behaviors. Addressing these gaps, we develop *WhisperFuzz*, the first approach that combines white-box fuzzing with static analysis. Its primary objectives are not only to accurately determine the locations of timing vulnerabilities but also to evaluate the timing behaviors. *WhisperFuzz* has successfully detected 12 new timing vulnerabilities and all previously known ones in open-source processors. Moreover, it pinpoints the root causes of these vulnerabilities. This

opens up novel avenues in vulnerability detection and timely mitigation in processors.

8 Acknowledgement

Our research work was partially funded by Intel's Scalable Assurance Program, Deutsche Forschungsgemeinschaft (DFG) – SFB 1119 – 236615297, the European Union under Horizon Europe Programme – Grant Agreement 101070537 – CrossCon, the European Research Council under the ERC Programme - Grant 101055025 - HYDRANOS, the US Office of Naval Research (ONR Award #N00014-18-1-2058), the Lockheed Martin Corporation, and the Centre for Hardware Security Entrepreneurship Research and Development (C-HERD) project, Ministry of Electronics and Information Technology (MEiTY), Government of India. This work does not in any way constitute an Intel endorsement of a product or supplier. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect those of Intel, the European Union, the European Research Council, Lockheed Martin Corporation, the US Government, or the Indian Government.

References

[1] BOOM Speculative Attacks. https://github.com/riscv-boom/boom-attacks, 2019. Last accessed on 10/01/2023.

[2] Zkt "Constant Time" Instruction List. https://github.com/rvkrypto/riscv-zkt-list/blob/main/zkt-list.adoc, 2021. Last accessed on 10/01/2023.

[3] Synopsys VCS. https://www.synopsys.com/verification/simulation/vcs.html, 2022. Last accessed on 10/01/2023.

[4] Cross-Process Information Leak. https://www.amd.com/en/resources/product-security/bulletin/amd-sb-7008.html, 2023. Last accessed on 09/28/2023.

[5] National Vulnerability Database. https://nvd.nist.gov/vuln/search, 2023. Last accessed on 09/28/2023.

[6] A. C. Aldaya, B. B. Brumley, et al. Port Contention for Fun and Profit. *IEEE Symposium on Security and Privacy*, 2019.

[7] A. Amid, D. Biancolin, et al. Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs. *IEEE Micro*, 40(4):10–21, 2020.

[8] M. Andrysco, D. Kohlbrenner, et al. On Subnormal Floating Point and Abnormal Timing. *IEEE Symposium on Security and Privacy*, 2015.

[9] K. Asanović, R. Avizienis, et al. The Rocket Chip Generator. (UCB/EECS-2016-17), Apr 2016.

[10] C. Baier and J.-P. Katoen. Principles of Model Checking. 2008.

[11] D. J. Bernstein. Cache-timing attacks on AES. 2005.

[12] A. Bhattacharyya, A. Sandulescu, et al. Smotherspectre: Exploiting Speculative Execution Through Port Contention. *ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[13] R. Bloem, B. Gigerl, et al. Power Contracts: Provably Complete Power Leakage Models for Processors. *ACM SIGSAC Conference on Computer and Communications Security*, pages 381–395, 2022.

[14] C. Chen, V. Gohil, et al. PSOFuzz: Fuzzing Processors with Particle Swarm Optimization. *arXiv preprint arXiv:2307.14480*, 2023.

[15] C. Chen, R. Kande, et al. Trusting the Trust Anchor: Towards Detecting Cross-Layer Vulnerabilities with Hardware Fuzzing. pages 1379–1383, 2022.

[16] C. Chen, R. Kande, et al. HyPFuzz: Formal-Assisted Processor Fuzzing. *USENIX Security Symposium*, pages 1361–1378, August 2023.

[17] E. Clarke, O. Grumberg, et al. Progress on the State Explosion Problem in Model Checking. *Informatics*, pages 176–194, 2001.

[18] E. M. Clarke, T. A. Henzinger, et al. Handbook of Model Checking. 10, 2018.

[19] E. M. Clarke, W. Klieber, et al. Model Checking and the State Explosion Problem. *LASER Summer School on Software Engineering*, pages 1–30, 2011.

[20] D. Cyrluk, S. Rajan, et al. Effective Theorem Proving for Hardware Verification. *International Conference on Theorem Provers in Circuit Design*, 1994.

[21] G. Dessouky, D. Gens, et al. HardFails: Insights into Software-Exploitable Hardware Bugs. *USENIX Security Symposium*, pages 213–230, 2019.

[22] M. R. Fadiheh, J. Müller, et al. A Formal Approach for Detecting Vulnerabilities to Transient Execution Attacks in Out-of-Order Processors. *ACM/IEEE Design Automation Conference*, pages 1–6, 2020.

[23] M. R. Fadiheh, D. Stoffel, et al. Processor Hardware Security Vulnerabilities and Their Detection by Unique Program Execution Checking. *IEEE Design, Automation & Test in Europe Conference & Exhibition*, 2019.

[24] Q. Ge, Y. Yarom, et al. A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware. *Journal of Cryptographic Engineering*, 2018.

[25] K. v. Gleissenthall, R. G. Kıcı, et al. IODINE: Verifying Constant-Time Execution of Hardware. *USENIX Security Symposium*, pages 1411–1428, 2019.

[26] V. Gohil, R. Kande, et al. MABFuzz: Multi-Armed Bandit Algorithms for Fuzzing Processors. *arXiv preprint arXiv:2311.14594*, 2023.

[27] R. Gopinath, C. Jensen, et al. Code Coverage for Suite Evaluation by Developers. *ACM/IEEE International Conference on Software Engineering*, pages 72–82, 2014.

[28] B. Gras, C. Giuffrida, et al. ABSynthe: Automatic Blackbox Side-channel Synthesis on Commodity Microarchitectures. *NDSS*, 2020.

[29] S. L. W. Group. IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. *IEEE Std 1800-2017*, 2018.

[30] D. Gruss, C. Maurice, et al. Flush+ Flush: a fast and stealthy cache attack. *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 279–299, 2016.

[31] X. Guo, R. G. Dutta, et al. Scalable SoC Trust Verification using Integrated Theorem Proving and Model Checking. pages 124–129, 2016.

[32] J. L. Hennessy and D. A. Patterson. Computer Architecture: A Quantitative Approach. 2011.

[33] W. Hu, A. Ardeshiricham, et al. Hardware Information Flow Tracking. *ACM Computing Surveys*, 2021.

[34] J. Hur, S. Song, et al. DIFUZZRTL: Differential Fuzz Testing to Find CPU Bugs. *IEEE Symposium on Security and Privacy*, pages 1286–1303, 2021.

[35] A. Ibrahim, H. Nemati, et al. Microarchitectural Leakage Templates and Their Application to Cache-Based Side Channels. *ACM SIGSAC Conference on Computer and Communications Security*, 2022.

[36] M. Ivanković, G. Petrović, et al. Code Coverage at Google. *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 955–963, 2019.

[37] R. Kande, A. Crump, et al. TheHuzz: Instruction Fuzzing of Processors Using Golden-Reference Models for Finding Software-Exploitable Vulnerabilities. *USENIX Security Symposium*, pages 3219–3236, 2022.

[38] P. Kocher, J. Horn, et al. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[39] K. Laeufer, J. Koenig, et al. RFUZZ: Coverage-Directed Fuzz Testing of RTL on FPGAs. *IEEE International Conference on Computer-Aided Design*, 2018.

[40] M. Lipp, M. Schwarz, et al. Meltdown: Reading Kernel Memory from User Space. *USENIX Security*, 2018.

[41] MITRE. CWE VIEW: Hardware Design. `https://cwe.mitre.org/data/definitions/1194.html`, 2019. Last accessed on 09/28/2023.

[42] S. K. Muduli, G. Takhar, et al. HyperFuzzing for SoC Security Validation. *ACM/IEEE International Conference on Computer-Aided Design*, pages 1–9, 2020.

[43] M.-J. O. Saarinen. riscv-zkt-list. `https://github.com/rvkrypto/riscv-zkt-list`, 2021. Last accessed on 10/16/2023.

[44] O. Oleksenko, C. Fetzer, et al. Revizor: Testing Black-Box CPUs Against Speculation Contracts. *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 226–239, 2022.

[45] O. Oleksenko, M. Guarnieri, et al. Hide and Seek with Spectres: Efficient Discovery of Speculative Information Leaks with Random Testing. pages 1737–1752, 2023.

[46] M. Orenes-Vera, A. Manocha, et al. AutoSVA: Democratizing Formal Verification of RTL Module Interactions. *ACM/IEEE Design Automation Conference*, pages 535–540, 2021.

[47] T. Ormandy and D. Moghimi. Downfall and Zenbleed: Googlers helping secure the ecosystem. `https://security.googleblog.com/2023/08/downfall-and-zenbleed-googlers-helping.html`, 2023. Last accessed on 09/28/2023.

[48] D. A. Osvik, A. Shamir, et al. Cache Attacks and Countermeasures: The Case of AES. *The Cryptographers' Track at the RSA Conference.*, 2006.

[49] C. Percival. Cache Missing for Fun and Profit, 2005.

[50] H. Ragab, A. Milburn, et al. Crosstalk: Speculative Data Leaks Across Cores Are Real. *IEEE Symposium on Security and Privacy*, 2021.

[51] C. Rajapaksha, L. Delshadtehrani, et al. SIGFuzz: A Framework for Discovering Microarchitectural Timing Side Channels. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2023.

[52] RISC-V. RISC-V Webpage. https://riscv.org/, 2023. Last accessed on 10/01/2023.

[53] C. P. Robert. Monte Carlo Methods in Statistics. *arXiv:0909.0389*, 2009.

[54] M. Schwarz, M. Lipp, et al. ZombieLoad: Cross-privilege-boundary data sampling. *ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[55] J. P. Shen and M. H. Lipasti. Modern Processor Design: Fundamentals of Superscalar Processors. 2013.

[56] W. Snyder. Verilator. https://www.veripool.org/wiki/verilator, 2023. Last accessed on 10/01/2023.

[57] F. Solt, K. Ceesay-Seitz, et al. Cascade: CPU Fuzzing via Intricate Program Generation. *USENIX Security Symposium*, 2024.

[58] Synopsys. Accelerating Verification Shift Left with Intelligent Coverage Optimization. https://www.synopsys.com/cgi-bin/verification/dsdla/pdfr1.cgi?file=ico-wp.pdf, 2022. Last accessed on 02/18/2023.

[59] C. Trippel, D. Lustig, et al. Checkmate: Automated Synthesis of Hardware Exploits and Security Litmus Tests. *IEEE International Symposium on Microarchitecture*, 2018.

[60] D. M. Tullsen, S. J. Eggers, et al. Simultaneous Multithreading: Maximizing On-Chip Parallelism. *ACM International Symposium on Computer Architecture*, 1995.

[61] J. Van Bulck, M. Minkin, et al. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. *USENIX Security Symposium*, 2018.

[62] J. Van Bulck, D. Moghimi, et al. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. *IEEE Symposium on Security and Privacy*, 2020.

[63] S. van Schaik, A. Milburn, et al. RIDL: Rogue in-flight data load. *IEEE Symposium on Security and Privacy*, 2019.

[64] Y. Wang, A. Ferraiuolo, et al. Timing Channel Protection for a Shared Memory Controller. 2014.

[65] Z. Wang, G. Mohr, et al. Specification and Verification of Side-Channel Security for Open-Source Processors via Leakage Contracts. *ACM SIGSAC Conference on Computer and Communications Security*, pages 2128–2142, 2023.

[66] V. M. Weaver, D. Terpstra, et al. Non-Determinism and Overcount on Modern Hardware Performance Counter Implementations. *IEEE International Symposium on Performance Analysis of Systems and Software*, pages 215–224, 2013.

[67] D. Weber, A. Ibrahim, et al. Osiris: Automated Discovery of Microarchitectural Side Channels. *30th USENIX Security Symposium*, pages 1–18, 2021.

[68] J. Wikner and K. Razavi. RETBLEED: Arbitrary speculative code execution with return instructions. *USENIX Security Symposium*, 2022.

[69] B. Wile, J. Goss, et al. Comprehensive Functional Verification: The Complete Industry Cycle. 2005.

[70] H. Witharana, Y. Lyu, et al. A Survey on Assertion-based Hardware Verification. *ACM Computing Surveys*, 2022.

[71] J. Xu, Y. Liu, et al. MorFuzz: Fuzzing Processor via Runtime Instruction Morphing enhanced Synchronizable Co-simulation. 2023.

[72] T. Yavuz, F. Fowze, et al. Encider: detecting timing and cache side channels in sgx enclaves and cryptographic apis. *IEEE Transactions on Dependable and Secure Computing*, 20(2):1577–1595, 2022.

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

[74] J. Zhao, B. Korpan, et al. SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine. *4th Workshop on Computer Architecture Research with RISC-V*, 2020.

[75] L. Zhao, Y. Duan, et al. Send Hardest Problems My Way: Probabilistic path prioritization for hybrid fuzzing. *NDSS*, 2019.

Listing 4: Verilog code of multi-ways cache set protocol.

```
input addr; output way;
reg hit, full, valid[cache_ways], fetch;
wire tag_addr, complete;
assign tag_addr = addr[tag_bits];
always@(posedge clock) begin
    if tag_addr == tag[0]
        way <= index[0], hit <= 1;
    else if tag_addr == tag[1]
        way <= index[1], hit <= 1;
    if hit == 0 && full != 1 begin
        if valid[0] == 1
            fetch <= 1; temp_way <= index;
        ...
    end
    if fetch == 1
        // Call mem_call for fetching block from next level.
        // Sub-instance sets complete signal
        mem_call(addr, complete);
    if complete:
        way <= temp_way
end
```

Appendix

A Implementation details of *Micro-Event Graph*

Section 3.5 explains the concept of a *Micro-Event Graph* (MEG). However, when implementing the strategy for real RTL designs, the syntaxes of hardware description languages (HDLs), such as Verilog and SystemVerilog [29], introduce more implementation challenges. Based on the example shown in Listing 2, we created a more complicated cache set with multiple ways as shown in Listing 4. We use the cache set as a case study to explain the implementation details of MEGs. Figure 7 shows the sub-graph of the protocol.

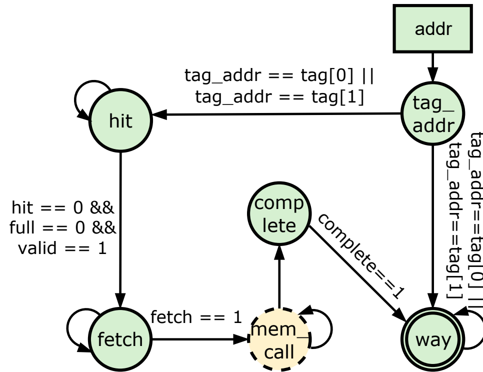


Figure 7: Sub-graph extracted from *Micro-Event Graph* in Listing 4 of the multi-ways cache set protocol.

Nested branch statements. Except for the fundamental branch statements of HDLs, such as `if/else` and ternary (`?`), a register-transfer level (RTL) module also contains nested branch statements. Since all branch statements must be satisfied to trigger the operations under the most nested branch statement, we annotate an edge from each signal in those branch statements to the destination signals of the operations. For example, from line 10 to 12, nested branch statements `hit == 0 && full` $\neq$ `1` and `valid[0] == 1` drive the assignments of the `fetch` signal. Therefore, in Figure 7, the condition on the edge (`hit,fetch`) is the conjunction of `hit == 0 && full` $\neq$ `1` and `valid[0] == 1`.

Path Explosion: In certain cases, an event on one component is capable of triggering an event on another component under multiple different conditions. For example, from line 6 to 9, since the cache set has multiple ways, both condition `tag_address == tag[0]` and `tag_addr = tag[1]` can represent a cache hit. This results in the existence of four edges from node `tag_addr` to `hit` and `way`. Each is annotated with one condition between the two nodes. The existence of multiple edges between the same pair of nodes causes a path explosion during timing coverage instrumentation as mentioned in Section 3.8. Consequently, the number of SystemVerilog Assertion (SVA) `cover` properties in the module increases exponentially. To solve this challenge, we create a disjunction for different conditions on edges joining the same pair of nodes in a path. Figure 7 shows the disjunction on edge (`tag_addr,hit`) and (`tag_addr,way`).

B SVA `cover` properties for case study

Listing 5 shows the `cover` property for the graphical paths in the *Micro-Event Graph* of the cache case study as shown in Listing 2. Section 3.5 mentions two *Micro-Event Paths* from its input `addr` to its output `way`: {`addr` → `tag_addr` → `way`} and {`addr` → `tag_addr` → `hit` → `fetch` → `mem_call` → `complete` → `way`} Among these paths, `hit`, `full`, `fetch`, `way` are sequential nodes; the events occurring on these nodes complete on the next clock edge. The value of the `complete` signal is driven by `mem_call`, a subinstance.

Listing 5: SVA `cover` properties for case study.

```
property p1; //{addr, tag_addr, way}
@(posedge clock) (tag_address == tag);
endproperty

property p2; //{addr, tag_addr, hit, fetch, mem_call,
    ↪ complete, way}
@(posedge clock) !(tag_address == tag) ##1 (hit == 0 &&
    ↪ full != 1 && valid[0] == 1) |-> s_eventually (
    ↪ complete == 1);
endproperty
```

C Proof of concept codes for triggering detected vulnerabilities

This section shows the proof of concept code snippets on triggering various timing vulnerabilities in different RISC-V processors. Listing 6 shows the code snippets of `DIVUW + REM` side-channel on `BOOM` [74]. Listing 7 shows the code snippets of `DIVUW` side-channel on `CVA6` [73]. Listing 8 shows the

Table 2: Summary of results generated by *WhisperFuzz* across different processors. Along with the detected vulnerability, we identify the specific lines in the RTL and trace the signals. We also note the time taken for detecting the various vulnerabilities.

Processor	Vulnerability	Source Module	RTL Lines	Phase 1 Results	Phase 2 Results	Seed generation(s)	Input generation ($*10^3$ s)	Leakage Analyzer ($*10^4$s)
CVA6	DIVUW	serdiv	230115, 230117, 230166, 230169, 230173	Multiple signals	Multiple Signals	54.00	10.70	13.19
CVA6	Divide by zero	serdiv	13522, 13514, 13512	Multiple signals	div_res_zero_q, op_b_q, state_q, cnt_q	3.67	0.61	2.00
BOOM	Divide by zero	MulDiv	230134, 230136, 230148, 230175	Multiple Signals	neg_out, count, state, remainder	3.67	0.66	2.27
Rocket Core	Divide by zero	MulDiv	209671, 209673, 209703, 209725	io_req_bits_in2, _divisor_T	neg_out, divisor, state	3.67	0.63	1.65
Rocket Core	DIVUW	MulDiv	230115, 230117, 230166, 230173	io_req_bits_in2, _divisor_T	_divisor_T, divisor	3.67	0.53	1.71
BOOM	DIVUW	MulDiv	230115, 230117, 230166, 230173	io_req_bits_in2, _divisor_T	_divisor_T, divisor	3.67	0.53	2.34
BOOM	SC	BoomWritebackUnit	180738, 180747, 180756, 180765, 180774, 180783, 180792, 180801	io_data_resp	All wb_buffer	146.00	29.15	34.10
CVA6	Compressed Instructions	ALU	3757	adder_z_flag	Multiple Signals	124	24.77	1.77

code snippets of REMW side-channel on CVA6 [73]. Listing 9 shows the code snippets of compressed instruction-based side-channels on CVA6 [73].

Listing 6: DIVUW + REM side-channel proof of concept code snippets on BOOM [74].

```
1 LI       a3, 291
2 LI       a7, -1954
3 LI       s10, 201
4 LI       t6, 477
5 // If t6=477, a7=-1954, s10=256, a3=291 takes
      ↪ 42ns
6 // If t6=1  , a7=-1954, s10=201, a3=291 takes
      ↪ 143ns
7 DIVUW    t5, a7, t6
8 REM      t3, s10, a3
```

Listing 7: DIVUW side-channel proof of concept code snippets on CVA6 [73].

```
1 LI       a4, 3
2 // If a4=0 takes 92ns
3 // If a4=3 takes 36ns
4 LI       a7, 1333
5 // If a4=33 a7=-1333 takes 54ns
6 DIVUW    t3, a7, a4
```

Listing 8: REMW side-channel proof of concept code snippets on CVA6 [73].

```
1 LI      a4, 3
2 // If a4=0 takes 92ns
3 // Else on average takes 37ns
4 LI      a7, 1333
5 REMW    t3, a7, a4
```

Listing 9: C.ADD[W], C.SUB[W], C.AND, C.OR, C.XOR, and [C.]MV side-channel proof of concept code snippets on CVA6 [73].

```
1 LI    a3, 0
2 // If a3=0 \& a4=0 takes 36 cycles
3 // Else on takes 24 cycles
4 LI    a4, 1023
5 MV    a3, a4
6 // Same for C.ADD[W], C.SUB[W], C.AND, C.OR, C
      ↪ .XOR, and C.MV
```

D Locations of side channels

This section shows the locations of timing vulnerabilities identified by the *Diagnozer* (See Section 3.7). Listing 10 shows the location of *Division by zero* in BOOM [74]. Listing 11 shows the location of *Division by zero* in Rocket Core [9]. Listing 12 shows the location of *Division by zero* in CVA6 [73]. Listing 13 shows the location of DIVUW+REM in BOOM. Listing 14 shows the location of **compressed instructions** and **MV** in CVA6; nine vulnerabilities share the same root cause. The results show that the *Diagnozer* of *WhisperFuzz* can successfully identify the location of timing vulnerabilities in processors.

Listing 10: Source location of ***Division by zero*** in `BOOM` [74].

```
230133 if (eOut_1 == 1) begin // @[Multiplier.scala 153:19]
230134     count <= {{1'd0}, eOutPos};
230135 end else begin
230136     count <=  _count_T_1; // @[Multiplier.scala 143:11]
230137 end
230138 ...
230139 if (divby0 &  _eOut_T_4) begin
230140     neg_out <= 1'h0; // @[Multiplier.scala 158:38]
   ... end
230147 ...
230148 end else if (state == 3'h3) begin
230149     remainder <= {{1'd0}, _GEN_16};
230150 end
```

Listing 11: Source location of ***Division by zero*** in `Rocket Core` [9].

```
209670 end else if (lhs_sign |rhs_sign) begin // @[Multiplier.
           ↪ scala 164:36]
209671     state <= 3'h1;
209672 end else begin
209673     state <= 3'h3;
209674 end
   ... ...
209702 end else begin
209703     neg_out <= lhs_sign != rhs_sign;
   ... ...
209724 if ( _T_38) begin // @[Multiplier.scala 163:24]
209725     divisor <=  _divisor_T; // @[Multiplier.scala 169:13]
```

Listing 12: Source location of ***Division by zero*** in `CVA6` [73].

```
13512     state_q <= state_d;
13513     op_a_q <= op_a_d;
13514     op_b_q <= op_b_d;
13515     res_q <= res_d;
13516     cnt_q <= cnt_d;
13517     id_q <= id_d;
13518     rem_sel_q <= rem_sel_d;
13519     comp_inv_q <= comp_inv_d;
13520     res_inv_q <= res_inv_d;
13521     op_b_zero_q <= op_b_zero_d;
13522     div_res_zero_q <= div_res_zero_d;
```

Listing 13: Source location of **`DIVUW+REM`** in `BOOM` [74].

```
230112 if (cmdMul) begin // @[Multiplier.scala 164:17]
230113     state <= 3'h2;
230114 end else if (lhs_sign |rhs_sign) begin // @[Multiplier.
           ↪ scala 164:36]
230115     state <= 3'h1;
230116 end else begin
230117     state <= 3'h3;
   ... ...
230165 if ( _T_38) begin // @[Multiplier.scala 163:24]
230166     divisor <=  _divisor_T; // @[Multiplier.scala 169:13]
230167 end else if (state == 3'h1) begin // @[Multiplier.scala
           ↪  91:57]
230168     if (divisor[63]) begin // @[Multiplier.scala 95:25]
230169         divisor <= subtractor; // @[Multiplier.scala 96:15]
230170     end
230171 end
230172 if ( _T_38) begin // @[Multiplier.scala 163:24]
230173     remainder <= {{66'd0}, lhs_in}; // @[Multiplier.scala
               ↪ 170:15]
```

Listing 14: Source location of **compressed instructions** and **`MV`** in `CVA6` [73].

```
3754 assign adder_z_flag = ~|adder_result;
3755
3756 // get the right branch comparison result
3757 always_comb begin :branch_resolve
   5     // set comparison by default
   6     alu_branch_res_o = 1'b1;
   7     case (fu_data_i.operator)
   8         EQ: alu_branch_res_o = adder_z_flag;
```

Table 3: MEG and SVA overhead statistics for `BOOM` [74].

Module	MEG		SVA coverage points (per module)
	Time taken(s)	Space consumed(kB)	
MaxPFibLFSR1	0.32	5.2	302
BoomIOMSHR	0.34	23.1	157
BoomWbUnit	0.32	20.6	3372
FetchBuffer	0.52	403.2	1458
BrnchKillQ2	0.35	22.7	1066
ICache	0.13	14.9	3869
TLXbar_7	0.32	13.1	187
BrnchKillQ	0.49	106.1	2777
BoomRAS	0.35	17.0	125
Queue_35	0.36	20.7	124