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-
sign hardware and prove security assertions about hardware. Thus, detecting timing vulnerabilities using formal methods is a rigorous approach to ensure design security [22, 58]. 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 [58]. 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 scalibility 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, 57]. 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 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 location/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) – Boot [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 Zk 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
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.

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.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>LI t6, 0x81321 LI t6, 0x11235</td>
</tr>
<tr>
<td>2</td>
<td>LI a7, 0xFFEEE LI a7, 0xFFFFF</td>
</tr>
<tr>
<td>3</td>
<td>LD t4, 0(a7) LD t4, 0(a7)</td>
</tr>
<tr>
<td>4</td>
<td>ADDI a9, a5, t4 ADDI a9, a5, t4</td>
</tr>
<tr>
<td>5</td>
<td>LD t5, 0(t6) LD t5, 0(t6)</td>
</tr>
</tbody>
</table>

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 micro-architecture. 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.

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 → Ready} taking 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 fusing. 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 instruc-
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 Operands mutator completely cover the timing behaviors of the DUT.

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 \( \mathcal{G}(D) = (S, \Sigma) \), where the nodes are denoted by \( S = \{s_1, s_2, \ldots, s_n\} = \{S_i \cup S_j \cup I \cup O\} \) and include all elements in D: \( S_i \) and \( S_j \) 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 \( \Sigma \) represents the edges in the graph, \( \mathcal{G} \), 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.

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

```
input addr; output way;
reg hit, full, valid, fetch;
wire tag_addr, complete;
assign tag_addr = addr[15:8];
always@(posedge clock) begin
    if tag_addr == tag
        way <= index; // if hit == 0 &&
    if hit == 0 && // full == 0 &
        valid == 1
            fetch <= 1; // Call mem_call for fetching block from next level.
            mem_call(addr, complete);
        if complete;
        way <= temp_way;
end
```
Algorithm 1: Preprocessor in Vulnerability Localization module of WhisperFuzz.

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D$ // RTL code of module</td>
<td>$G$ // MEG of module</td>
</tr>
</tbody>
</table>

1. $G(D).I \leftarrow \emptyset$; $G(D).S_I \leftarrow \emptyset$; $G(D).O \leftarrow \emptyset$
2. $G(D).S \leftarrow \emptyset$; $G(D).S_{\Sigma} \leftarrow \emptyset$
3. $G(D).I$, $G(D).S_I$, $G(D).O \leftarrow \text{GETINOUTPUTS}(D)$
4. $G(D).S \leftarrow \text{GETSIGNALS}(D)$
5. /* Iterate RTL code */
6. for line $\in D$ do
7.    $s_2 \leftarrow \text{GETDESTINATION}(\text{line})$
8.    $s_1 \leftarrow \text{GETOPERANDS}(\text{line})$
9.    for $s_1 \in S_I$ do
10.      $G(D).S \leftarrow G(D).S \cup \{s_1, s_2, \text{line_no}\}$
11. return $G$

Definition 2 A sequence of directed edges, $P = \{(s_1, s_2), (s_2, s_3), \ldots, (s_{n-1}, s_n)\}$ 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 $G(D)$. Figure 4 notes two different MEPs from the input node $\text{addr}$ to the output node $\text{way}$. 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 ($\text{LookUp} \rightarrow \text{Ready}$) mapped to MEP ($\text{addr} \rightarrow \text{tag_addr} \rightarrow \text{way}$). In case of a cache miss, if the cache set has a free cache line, the module follows the FSM path ($\text{LookUp} \rightarrow \text{FreeBlock} \rightarrow \text{Wait} \rightarrow \text{Ready}$) mapped to ($\text{addr} \rightarrow \text{tag_addr} \rightarrow \text{hit} \rightarrow \text{fetch} \rightarrow \text{mem_call} \rightarrow \text{complete} \rightarrow \text{way}$). A comparison of these two different MEPs indicates the RTL wire $\text{tag_addr}$ 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.

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 assign statement on Line 4, a transition in the value of $\text{addr}$ triggers a change in the value of $\text{tag_addr}$. To denote this dependency, the graph contains a directed edge from the node $\text{addr}$ to $\text{tag_addr}$. Similarly, the value of $\text{tag_address}$ can influence the value of $\text{way}$ and $\text{hit}$ if the condition ($\text{tag_address} == \text{tag}$) on Line 6 holds. Hence, the edges from $\text{tag_address}$ to $\text{way}$ and $\text{hit}$ are annotated with the condition. The sub-instance call on Line 13 corresponds to the dashed node $\text{mem_call}$ in the MEG. The accept state from node $\text{way}$ indicates that it references an output signal. 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 $G(D).X$ 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 comparatively reduced per-module 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,
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, ST1 and ST2 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 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_Vs whose traces first differ between ST1 and ST2.

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_Vs. This is stored in the list Vf 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, 25, 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 con-

---

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

```
Input : (ST1, ST2) // Pair of simulation traces
G(D) // MEG of module under examination
Output : (Vf, L) // Set of signals identified as the cause of the timing difference and corresponding line numbers in RTL,

1. Vf ← φ; temp_Vs ← φ; L ← φ
/* Phase 1 */
2. for every clock cycle (clk) in ST1 do
3.   for s ∈ G(D) · S do
4.     if s ∈ ST1[clk] ≠ s ∈ ST2[clk] then
5.       temp_Vs ← temp_Vs ∪ {s}
6.     if temp_Vs is not empty then
7.       break
/* Phase 2 */
8. for temp_Vs is not empty do
9.   new_Vs ← φ
10.  for s ∈ temp_Vs do
11.    for (s, s_child) ∈ G(D) · Σ do
12.      if s_child is sequential then
13.        Vf ← Vf ∪ {s_child}
14.        L ← L ∪ {line_no} // RTL line number
15.      else
16.        new_Vs ← new_Vs ∪ {s_child}
17.    end
18.  end
19.  temp_Vs = new_Vs
20. return (Vf, L)
```
tains 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 mutation 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, \( P \in G(D) \), 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 A 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.

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 gener-

<table>
<thead>
<tr>
<th>Algorithm 3: Coverage analyzer</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong>: ( P ) // Micro-Event Path</td>
</tr>
<tr>
<td>( G(D) ) // MEG of module ( D )</td>
</tr>
<tr>
<td><strong>Output</strong>: Condition // Assertion Property corresponding to Micro-Event Path ( P )</td>
</tr>
<tr>
<td>1. ( \text{Condition} \leftarrow {} ) // Empty Sequence</td>
</tr>
<tr>
<td>2. ( \text{for} \ (s_{i1}, s_{i2}) \in P \text{ do} )</td>
</tr>
<tr>
<td>3. ( \text{if} \ (s_{i1}, s_{i2}) \text{ has branch then} )</td>
</tr>
<tr>
<td>4. ( \text{Condition} \leftarrow \text{Condition}</td>
</tr>
<tr>
<td>5. ( \text{if} \ s_{i2} \in S_q \text{ then} )</td>
</tr>
<tr>
<td>6. ( \text{Condition} \leftarrow \text{Condition}</td>
</tr>
<tr>
<td>7. ( \text{if} \ s_{i1} \in S_f \text{ then} )</td>
</tr>
<tr>
<td>8. ( \text{Condition} \leftarrow \text{Condition}</td>
</tr>
<tr>
<td>9. return Condition</td>
</tr>
</tbody>
</table>
ate sequences of instructions as inputs are also compatible [14, 24, 26, 34, 37, 39, 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 combinatorial 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 B 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 \texttt{MV} instruction has the same timing behavior as its compressed version, i.e., \texttt{CMV} 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].** \textit{WhisperFuzz} successfully generated test cases that exposed timing side-channel vulnerabilities related to division instructions (DIV), a vulnerability disclosed in \textit{SIGFuzz} [51] for the BOOM processor. \textit{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].** \textit{WhisperFuzz} also identified a timing side-channel associated with Store-Conditional operations, effectively diagnosing timing disparities resulting from the presence of the dirty bit in the data cache implementation. The vulnerability was discovered in \textit{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 \textit{Diagnozer} of \textit{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 \textit{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 \textit{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 6, the \textit{Diagnozer} identifies the source of the vulnerability in module \texttt{serdiv} in the lines highlighted in Listing 3. The first phase of the \textit{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 8 the first phase of the \textit{Diagnozer} identifies the sources of these vulnerabilities in the ALU module. The exact RTL lines are as highlighted in Listing 13. 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 5, the \textit{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 5 with the divisor set to 0, the \textit{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 \texttt{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 (\texttt{serdiv}), the source of the vulnerabilities within the module differs. Hence, the automated localization of vulnerability sources performed by \textit{WhisperFuzz} is beneficial and efficient.

### 4.5 Coverage Analysis

We use \textit{Coverage analyzer} to monitor timing behaviors explored by inputs from \textit{Operand mutator}. \textit{Coverage analyzer} converts paths of Micro-Event Graph of an RTL module into cover properties and instrument these properties in to DUT. After simulation, \textit{Coverage analyzer} collects assertion results to calculate the timing behaviors covered (See Sec-

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

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>L13512</td>
<td>state_q &lt;= state_d;</td>
</tr>
<tr>
<td>L13513</td>
<td>op_a_q &lt;= op_a_d;</td>
</tr>
<tr>
<td>L13514</td>
<td>op_b_q &lt;= op_b_d;</td>
</tr>
<tr>
<td>L13515</td>
<td>res_q &lt;= res_d;</td>
</tr>
<tr>
<td>L13516</td>
<td>cnt_q &lt;= cnt_d;</td>
</tr>
<tr>
<td>L13517</td>
<td>id_q &lt;= id_d;</td>
</tr>
<tr>
<td>L13518</td>
<td>rem_sel_q &lt;= rem_sel_d;</td>
</tr>
<tr>
<td>L13519</td>
<td>comp_inv_q &lt;= comp_inv_d;</td>
</tr>
<tr>
<td>L13520</td>
<td>res_inv_q &lt;= res_inv_d;</td>
</tr>
<tr>
<td>L13521</td>
<td>op_b_zero_q &lt;= op_b_zero_d;</td>
</tr>
<tr>
<td>L13522</td>
<td>div_res_zero_q &lt;= div_res_zero_d;</td>
</tr>
</tbody>
</table>
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.

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 un-coverable. 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 speculative 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 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.

5 Related Work

Existing state-of-the-art techniques for timing side-channel vulnerability detection primarily employ formal approaches [22, 23, 58, 60, 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, 57]. In this section, we 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 [58] 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 testcases 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. 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 primarily depend on the simultaneous multithreading architecture [6, 12, 59] which current open-sourced benchmarks [9, 73, 74] lack, 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. However, the existing black-box or grey-box fuzzing approaches fall short in pinpointing the root cause of timing vulnerabilities and 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 and pinpointed their root causes. 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


Appendix

A SVA cover properties for case study

Listing 4 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 4: SVA cover properties for case study.

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

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

B 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 5 shows the code snippets of DIVUW + REM side-channel on BOOM [74]. Listing 6 shows the code snippets of DIVUW side-channel on CVA6 [73]. Listing 7 shows the code snippets of REMW side-channel on CVA6 [73]. Listing 8 shows the code snippets of compressed instruction-based side-channels on CVA6 [73].

Listing 5: DIVUW + REM side-channel proof of concept code snippets on BOOM [74].

```verilog
LI a4, 3
// 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 6: DIVUW side-channel proof of concept code snippets on CVA6 [73].

```verilog
LI a3, 0
// If a3=0 and a4=0 takes 36 cycles
2 // Else on average takes 37ns
3 LI a4, 1023
4 MV a3, a4
5 // Same for 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].
```

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

```verilog
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 8: 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].

```verilog
LI a3, 0
// If a3=0 and a4=0 takes 36 cycles
2 // Else on average takes 37ns
3 LI a4, 1023
4 MV a3, a4
5 // Same for C.ADD[W], C.SUB[W], C.AND, C.OR, C.XOR, and C.MV.
```

C Locations of side channels

This section shows the locations of timing vulnerabilities identified by the Diagnozer (See Section 3.7). Listing 9 shows the location of Division by zero in BOOM [74]. Listing 10 shows the location of Division by zero in Rocket Core [9]. Listing 11 shows the location of Division by zero in CVA6 [73]. Listing 12 shows the location of DIVUW+REM in BOOM. Listing 13 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 9: Source location of Division by zero in BOOM [74].

```verilog
230133 if (eOut_1) begin // @Multiplier.scala 153:19
230134 if (eOut_1) begin // @Multiplier.scala 153:19
230135 end else begin
230136 if (divby0 && eOut_4) begin
230137 negOut <= 1'd0; // @Multiplier.scala 158:38
230138 end
230139 ...
230140 if (start) begin
230141 remainder <= [[0:0], 0_16];
```

Listing 10: Location of Division by zero in Rocket Core [9].

```verilog
230133 if (eOut_1) begin // @Multiplier.scala 153:19
230134 if (eOut_1) begin // @Multiplier.scala 153:19
230135 end else begin
230136 if (eOut_1) begin // @Multiplier.scala 153:19
230137 end else if (state == 3'b1) begin
230138 ...
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.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Vulnerability</th>
<th>Source Module</th>
<th>RTL Lines</th>
<th>Phase 1 Results</th>
<th>Phase 2 Results</th>
<th>Seed generation(s)</th>
<th>Input generation (+10's)</th>
<th>Leakage Analyzer (+10's)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CVA6</td>
<td>DIVUW + REM</td>
<td>serdiv</td>
<td>230115, 230117, 230166, 230169, 230173</td>
<td>Multiple signals</td>
<td>Multiple Signals</td>
<td>54.00</td>
<td>10.70</td>
<td>13.19</td>
</tr>
<tr>
<td>CVA6</td>
<td>Divide by zero</td>
<td>serdiv</td>
<td>13522, 13514, 13512</td>
<td>Multiple signals</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BOOM</td>
<td>Divide by zero</td>
<td>MulDiv</td>
<td>230134, 230136, 230148, 230175</td>
<td></td>
<td>Multiple Signals</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rocket Core</td>
<td>Divide by zero</td>
<td>MulDiv</td>
<td>209671, 209673, 209903, 209725</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BOOM</td>
<td>DIVUW</td>
<td>MulDiv</td>
<td>230115, 230117, 230166, 230173</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BOOM</td>
<td>SC</td>
<td>BoompWritebackUnit</td>
<td>180738, 180747, 180756, 180765, 180774, 180783, 180792, 180801</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Module</th>
<th>MEG</th>
<th>SVA coverage points (per module)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Time taken(s)</td>
</tr>
<tr>
<td>MaxPConfLFSR1</td>
<td>0.32</td>
<td>5.2</td>
</tr>
<tr>
<td>BoomIOMSHR</td>
<td>0.34</td>
<td>23.1</td>
</tr>
<tr>
<td>BoomWBUnit</td>
<td>0.32</td>
<td>20.6</td>
</tr>
<tr>
<td>FetchBuffer</td>
<td>0.52</td>
<td>403.2</td>
</tr>
<tr>
<td>BrmchKillQ2</td>
<td>0.35</td>
<td>22.7</td>
</tr>
<tr>
<td>ICache</td>
<td>0.13</td>
<td>14.9</td>
</tr>
<tr>
<td>TLXbar_7</td>
<td>0.32</td>
<td>13.1</td>
</tr>
<tr>
<td>BrmchKillQ</td>
<td>0.49</td>
<td>106.1</td>
</tr>
<tr>
<td>BoomRAS</td>
<td>0.35</td>
<td>17.0</td>
</tr>
<tr>
<td>Queue_35</td>
<td>0.36</td>
<td>20.7</td>
</tr>
</tbody>
</table>

Listing 11: 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_d <= rem_sel_d;
13519 comp_inv_q <= comp_inv_d;
13520 res_inv_q <= res_inv_d;
13521 op_b_zero_d <= op_b_zero_d;
13522 div_res_zero_d <= div_res_zero_d;

Listing 12: Source location of *DIVUW + REM* in BOOM [74].

230112 if (condhu) begin // @Multiplier.scala 164:17
230113 state <= 'h2;
230114 end else if (lhs_sign & rhs_sign) begin // @Multiplier.
230115 // scalar 164:36
230116 state <= 'h4;
230117 end else begin
230118 state <= 'h6;
230119 ... ...
230165 if (_7_38) begin // @Multiplier.scala 163:24
230166 divisor <= _divisor_T; // @Multiplier.scala 169:13
230167 end else if (state == 'h6) begin // @Multiplier.scala
230168 // 163:24
230169 ... ...
230173 remainder <= [[1:0], lhs_in]; // @Multiplier.scala
230174 // 170:15

Listing 13: Source location of *compressed instructions* and *MV* in CVA6 [73].

3754 // actual adder
3755 assign adder_result_ext_o = $unsigned(adders_in_a) +
3756 $unsigned(adders_in_b);
3757 assign adder_result = adder_result_ext_o[riscli::XLEN:];
3758 assign adder_o_flag = $adder_result;