Sonar: A Hardware Fuzzing Framework to Uncover Contention Side Channels in Processors

Kanqi Zhang
State Key Laboratory of Cyberspace Security Defense, IIE, CAS; UCAS
Beijing, China
zhangkanqi@iie.ac.cn

Peinan Li*
State Key Laboratory of Cyberspace Security Defense, IIE, CAS; UCAS
Beijing, China
lipeinan@iie.ac.cn

Miao Li
State Key Laboratory of Cyberspace Security Defense, IIE, CAS; UCAS
Beijing, China
limiao@iie.ac.cn

Xin Tian
State Key Laboratory of Cyberspace Security Defense, IIE, CAS; UCAS
Beijing, China
tianxin@iie.ac.cn

Zelong Du
State Key Laboratory of Cyberspace Security Defense, IIE, CAS; UCAS
Beijing, China
duzelong@iie.ac.cn

Quanchen Liu
State Key Laboratory of Cyberspace Security Defense, IIE, CAS; UCAS
Beijing, China
liuquanchen@iie.ac.cn

Yongqiang Lyu
Tsinghua University
Beijing, China
luyq@tsinghua.edu.cn

Yu Jiang
Tsinghua university
Beijing, China
jy1989@mail.tsinghua.edu.cn

Dan Meng
State Key Laboratory of Cyberspace Security Defense, IIE, CAS; UCAS
Beijing, China
mengdan@iie.ac.cn

Rui Hou*
State Key Laboratory of Cyberspace Security Defense, IIE, CAS; UCAS
Beijing, China
hourui@iie.ac.cn

Abstract

Contention-based side channels, rooted in resource sharing, have emerged as a significant security threat in modern processors. These side channels allow attackers to leverage timing differences caused by conflicts in execution ports, caches, or interconnects to infer secret information such as cryptographic keys or enclave-resident data. Despite increasing awareness, detecting such channels remains challenging because triggering contentions requires precisely orchestrating specific microarchitectural states, which is often difficult in practice, especially for timing-sensitive contentions.

This paper introduces `Sonar`, the first systematic and automated fuzzing framework designed to uncover contention side channels in processors. Our core idea is to leverage microarchitectural states to guide testcase generation, enabling the precise triggering of microarchitectural events with stringent conditions. `Sonar` is built on the key observation that multiplexers (MUXes) are hotspots for contention, as resource contention frequently involves data routing and signal selection, which are fundamentally implemented by MUXes in circuits. We first identify contention-critical states with side channel risks based on MUXes, and then utilize these runtime states to directly guide testcase generation via fuzzing, progressively approaching and ultimately triggering contentions. Finally, we employ a dual-differential comparison method to efficiently detect contention-induced side channels and simulate attack scenarios to assess their exploitability.

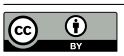
Evaluated on two out-of-order RISC-V processors, `Sonar` uncovers 14 contention side channels, including 11 previously unknown. These results demonstrate the effectiveness of `Sonar` in uncovering potentially exploitable microarchitectural contentions.

ACM Reference Format:
Kanqi Zhang, Peinan Li, Miao Li, Xin Tian, Zelong Du, Quanchen Liu, Yongqiang Lyu, Yu Jiang, Dan Meng, and Rui Hou. 2025. Sonar: A Hardware Fuzzing Framework to Uncover Contention Side Channels in Processors. In *58th IEEE/ACM International Symposium on Microarchitecture (MICRO '25), October 18–22, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3725843.3756136

1 Introduction

Recent studies have revealed significant risks associated with timing side channel attacks in processors. These side channels stem from contention among different requests for the same resources across users, such as execution ports and caches, enabling attackers to observe and exploit timing differences to infer encryption keys[1–4] and even extract private data from trusted execution environments[5–7]. Furthermore, contentions arising from shared components are widespread in processors, introducing security

*Corresponding authors: Peinan Li and Rui Hou.
IIE, CAS: Institute of Information Engineering, Chinese Academy of Sciences.
UCAS: University of Chinese Academy of Sciences.



MICRO '25, Seoul, Republic of Korea

ACM ISBN 979-8-4007-1573-0/25/10
https://doi.org/10.1145/3725843.3756136

risks at various stages of the pipeline[1–4, 8–15]. Although post-silicon defense strategies can mitigate timing side channels, they are insufficient to fully eliminate vulnerabilities[16, 17], and tend to incur substantial implementation complexity[18–20] and performance overheads[21, 22]. Therefore, integrating vulnerability detection into the pre-silicon phase presents a promising solution, as it provides an opportunity to address vulnerabilities during chip design and avoid costly post-silicon repairs.

However, current formal-based[23, 24] and fuzzing-based[25–27] pre-silicon timing side channel detectors are insufficient for targeting contention side channels. On one hand, they rely on random instruction sequences as testcases, which makes it difficult to trigger contention events at the microarchitectural level, especially those with strict timing requirements. On the other hand, they lack contention-related microarchitectural information as feedback, failing to perceive contention-triggering states, which hampers their ability to accurately induce contention. Thus, developing a universal approach that can effectively detect contention side channels remains demanding yet challenging.

Challenge 1: Identifying contention-critical states with side-channel risks in large-scale processor designs. In modern processors, frequent contention and the explosion of the microarchitectural state space greatly complicate the analysis of contentions. The primary challenge is to identify contention-critical microarchitectural states and filter out those do not pose side-channel risks.

Challenge 2: Triggering contention and exposing observable exploitable timing differences. Contention side channels arise from triggering contentions, which result in observable timing differences. However, there exists a gap between random testcase generation and the precise triggering of microarchitectural contention events. So the key challenge is to bridge this gap by effectively leveraging available contention states and maximizing the observability of timing differences.

Challenge 3: Detecting contention side channels and evaluating their exploitability. Contention side channels are identified through timing differences under different secret values. However, not all timing differences stem from contention, and the complexity of processors makes distinguishing them challenging and time-consuming. Therefore, rapid identification of contention side channels is critical. Additionally, assessing their exploitability presents another challenge.

To address the above challenges, we employ fuzzing to iteratively guide the generation of testcases by contention-critical microarchitectural states, thereby approaching and triggering contention to detect potential side channels. In this paper, we propose **`Sonar`, the first systematic and automated pre-silicon fuzzing framework targeting contention side channels**. We observe that contentions in processors often involve operations such as signal selection, data routing, and path switching, which are predominantly managed by multiplexers (MUXes). As a result, MUXes are frequently the hot spot of resource contentions. Therefore, we use MUXes as the basis for identifying contention points and constructing our fuzzing framework `Sonar`. In summary, this paper makes the following contributions:

- **Propose a method to identify and monitor contention-critical microarchitectural states.** We first locate potential contention resources based on MUX, referred to as *contention points*. All states at these points are considered contention-critical. To this end, we propose a general bottom-up tracing method to identify contention-critical states, including inputs (i.e., requests), select signals, and outputs at contention points. Additionally, we incorporate the timing interval between requests (abbr. *reqsIntvl*) as a type of contention state, as it is a critical factor that indicates the triggering state of contention. However, *reqsIntvl* requires instrumentation for dynamic collection. To minimize instrumentation overhead, we filter out states unrelated to inputs that cannot lead to side channels, ensuring testing efficiency without compromising detection capability.
- **Propose a contention-triggering mechanism guided by microarchitectural states.** To detect contention side channels, we design a testcase template with a specialized secret-dependent region. By observing timing differences in other instruction regions affected by contentions under different secret values, we can infer the secrets. To bridge the gap between random instruction sequences and the triggering of timing-sensitive microarchitectural contentions, we utilize the inter-request timing interval as feedback to guide testcase generation, ultimately meeting the timing conditions. Specifically, we collect runtime *reqsIntvl* associated with potential side-channel risks and use it as a metric to retain and select testcases that minimize *reqsIntvl* during fuzzing, thereby progressively approaching the triggering of contentions. Moreover, to accelerate the reduction and convergence of *reqsIntvl*, we introduce an adaptive directed mutation strategy.
- **Design efficient contention side channel detection and analysis method.** We accurately identify side channels by analyzing differences in the relative commit times of each instruction. Furthermore, we explore the root causes of these timing differences related to contention by comparing contention-critical states under different secret values. This dual-differential mechanism enables efficient detection of contention side channels. Additionally, we simulate realistic attack scenarios and design an attack template for evaluating the exploitability of detected side channels.
- **Evaluate `Sonar` in open-source RISC-V processors.** We evaluate `Sonar` on two open-source out-of-order RISC-V processors, BOOM and NutShell, and successfully detect 14 contention side channels, including 11 newly uncovered and 3 previously disclosed. We also assess the exploitability of these newly uncovered side channels.

2 Background

2.1 Contention Side Channels

In modern processors, contention is a prevalent issue due to the limited availability of resources, arising from conflicting access to shared resources by multiple requests. These requests may originate from different cores[28] or the same core. Within the same core, they can come from various threads (e.g., Intel's Simultaneous Multithreading[29]) or even from the same thread[4, 30]. Contention may affect the execution of the corresponding instructions for requests, leading to either blocking or acceleration, ultimately

exposing observable timing variations at the cycle-level. Attackers can exploit these timing discrepancies to launch side channel attacks, potentially compromising sensitive information.

Contention side channels can be categorized into two types depending on their impact on microarchitectural states[31]. The first category is *volatile side channel*, which contention induces transient state changes and leaves no persistent microarchitectural traces. For example, in the port contention scenario, multiple requests might share execution ports, such as INT/FP ports. When these ports become saturated, instruction dispatch is stalled and instruction execution latency is elevated. Detecting such latency potential indicates that port contention may have occurred. If the victim's port usage correlates with secrets, the attacker can deduce secret values via timing differences[2–4]. Components such as cache bank[1, 14, 32], interconnect path[28], and MSHR[11] are also vulnerable to such attacks. The second category is *persistent side channel*, which contention leaves lasting footprints in the microarchitecture. For instance, in the cache contention scenario, incorrect speculative execution may alter the order of memory operations, leading to persistent changes in cache state that can affect the execution of other memory access instructions mapped to the same cacheline. The attacker can measure their own cache access latency and infer the victim's memory access patterns based on timing differences[12, 13, 33–36]. In addition to caches, other persistent storage components, such as way predictor[37], TLB[15, 38], PHT[9], and BTB[39, 40] also can be exploited.

These two types of contention side channels impose different timing requirements on requests. The root cause of volatile side channels is the limited bandwidth of shared resources, which typically necessitates multiple requests contending for the same resource simultaneously. This imposes strict timing constraints, making such side channels difficult to be detected in practice. In contrast, to ensure that the subsequent requests are influenced by the persistent traces left by the preceding request, persistent side channels require requests to sequentially access the same microarchitectural storage unit (e.g., a cacheline or TLB entry), thereby imposing constraints on the data similarity between the requests. Current researches on the detection of contention side channels primarily focus on persistent side channels[41–43], lacking a systematic approach, especially for volatile side channels with strict timing requirements.

2.2 Processor Fuzzing

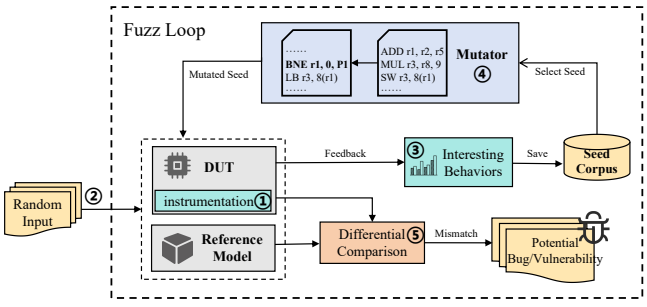


Figure 1: General workflow of processor fuzzing.

Recent studies have applied the fuzz testing technique from software to processor verification[25–27, 44–54]. The five key phases of fuzzing are illustrated in Figure 1. ①Instrumentation: Monitoring logic is embedded into the DUT (Design Under Test) to collect runtime information (e.g., control register coverage[44]) during test execution, providing feedback for testcase quality evaluation. ②Seed Generation: The fuzzer generates massive random testcases as initial seeds for fuzzing during initialization, populating a seed corpus. ③Seed Retention: High-quality testcases are dynamically retained in the seed corpus for subsequent mutation. ④Seed Mutation: As a key mechanism for testcase diversification, the fuzzer mutates selected seeds from the corpus to generate new testcases. ⑤Vulnerability Detection: Vulnerabilities are detected through differential analysis against reference models or expected outputs to identify hardware behavioral deviations.

Current research on processor fuzzing primarily focuses on detecting functional errors (e.g., Difuzzrtl[44], Cascade[47] and etc[45, 46, 48–53, 55–60]) and timing side channels (e.g., SIGFuzz[26], WhisperFuzz[27], SpecDoctor[25]). Existing fuzzers generally employ randomized instruction sequences as testcases to reach novel execution paths or microarchitectural states. However, when targeting microarchitectural events, such as contention in this paper, the role of randomized instruction sequences and uncontrolled mutation is limited. Because these microarchitectural events typically require specific microarchitectural states, such as strict timing requirements of requests. Random instructions at the architectural level often fail to accurately reach the required microarchitectural level states, resulting in a gap. Therefore, it is essential to design an effective mechanism to bridge this gap and detect side-channel risks triggered by the target events.

3 Threat Model

This paper focuses on uncovering exploitable side channels in processors caused by contention over shared resources. We assume that attackers can construct volatile or persistent contention side channels on various processor components. These channels can be constructed within a single core or dual-core architectures and can cross privilege levels or hardware threads. Additionally, attackers are assumed to extract secret information by leveraging cycle-level variations in instruction execution time. Sub-cycle level timing differences within circuits, which are generally unobservable at the architectural level, are beyond the scope of this work.

4 Design Overview

Contentions are prevalent in processors and may induce security risks such as side channel vulnerabilities. Once a chip is fabricated, patching such vulnerabilities becomes exceptionally challenging. Consequently, it is critical to proactively detect contention side channels during the register transfer level (RTL) design phase.

In this paper, we propose `Sonar`, the first systematic and automated pre-silicon fuzzing framework to detect contention side channels in processors. `Sonar` can be divided into three main components, as illustrated in Figure 2. *First*, we locate contention points in processors based on MUXes, identify and monitor contention-critical microarchitectural states at these points. To minimize instrumentation overhead and ensure testing efficiency, we filter out

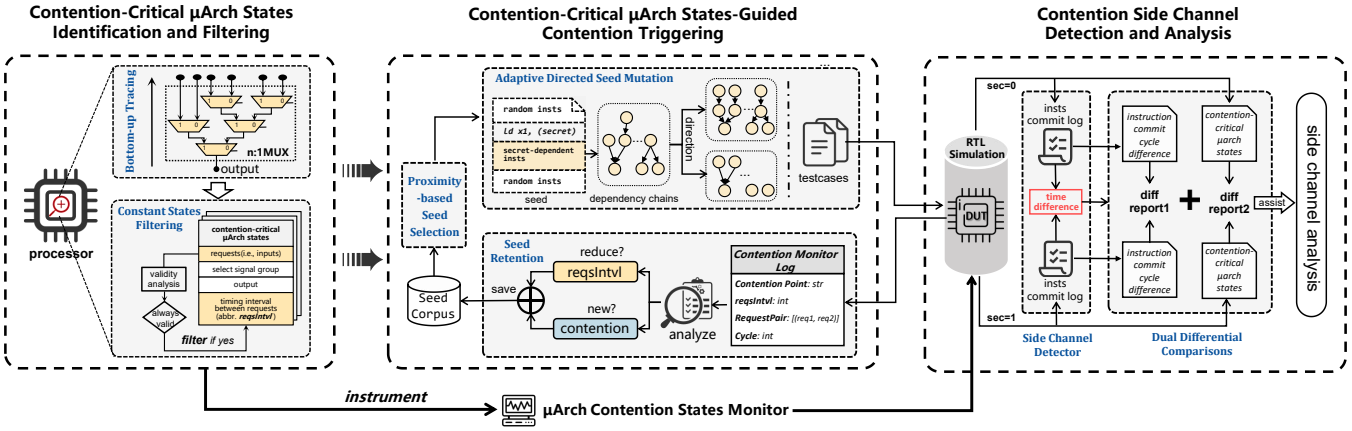


Figure 2: Design overview of Sonar.

states that do not pose side channel risks. *Then*, we design a specialized testcase template to support the detection of contention side channels under both single-core and dual-core scenarios. During the execution of testcases, we monitor and collect contention states that are related to side channels as feedback to guide the triggering of contention. Specifically, we gradually reduce the timing interval between requests to trigger timing-sensitive volatile contentions. We also propose an adaptive directed mutation strategy to further accelerate the reduction of request intervals. In addition, while monitoring timing conditions, we also increase the data similarity between requests by mutation to facilitate the exploration of persistent contention. *Finally*, to detect side channels caused by triggered contentions, we exhaustively inspect potentially affected instructions in the testcases and determine whether timing differences exist under different secret values. Upon observing timing differences, we leverage our dual-differential comparison method to pinpoint contention-related causes and uncover contention side channels. Subsequently, we employ our designed Meltdown-like attack templates to evaluate their exploitability.

5 Contention-Critical Microarchitectural States Identification and Filtering

All microarchitectural states at contention points are considered contention-critical. To find contention-critical microarchitectural states, we design a general method to locate contention points and recognize states on these points. These states will facilitate the triggering of contentions during fuzzing. However, the numerous identified states require extensive instrumentation for collection, which can reduce testing efficiency. Note that some states are input-independent and do not expose side-channel risks. Therefore, we filter these states to ensure testing efficiency.

5.1 Contention States Identification via Bottom-Up Tracing

Resource contentions often involve operations such as signal selection, data routing, and path switching. In circuits, although many combinational logic structures can be used to implement these operations, the fundamental implementation typically relies on multiplexer (MUX) [61]. Therefore, we use MUX as the general basis for locating *contention points*. MUX is a hardware element to select one input from multiple inputs as the output, with the select signal determining the source of the output. The simplest MUX is a 2:1 MUX (i.e., *output = mux(sel, tval, fval)*), which receives only two inputs. When the select signal *sel* is 1, *tval* is selected as the output; otherwise, *fval* is selected. When there are more than two inputs, an n:1 MUX can be implemented by cascading multiple 2:1 MUXes.

Leveraging the cascading property of MUX, we design a bottom-up tracing method starting from the output of an n:1 MUX to identify all requests and select signals involved at the contention point. Specifically, for each 2:1 MUX in the circuit, if its tval or fval is connected to the output of another 2:1 MUX, continue recursive tracing; otherwise, stop tracing. Through this method, an n:1 MUX cascading tree can be constructed, and all leaf nodes in the tree represent requests at the contention point. As illustrated in Figure 3, through bottom-up tracing, we start at the MUX output and identify all requests and selection signals at the contention point ldq_stq_idx.

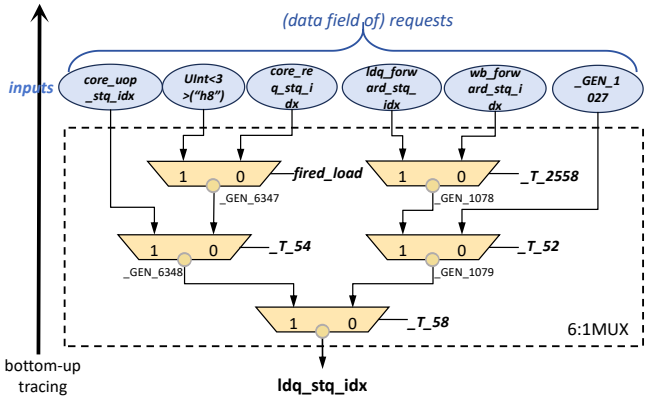


Figure 3: An example of using the bottom-up tracing method to identify all requests, select signals, and outputs at contention points.

At contention points, when multiple requests arrive simultaneously, the MUX prioritizes one request for processing while others must wait, leading to processing delays that may create a volatile side channel. In contrast, when requests arrive sequentially and occupy the same request path, earlier requests may leave traces in the data path that can be probed by subsequent requests, forming a persistent side channel. In both scenarios, whether requests arrive simultaneously or sequentially can be determined by observing the cycle intervals between requests. As such, *in addition to the inherent inputs (i.e., requests), select signals, and outputs at contention points, the timing intervals between requests (abbr. reqsIntvl) are also important contention-critical states.*

To collect these contention-critical states, we can directly monitor inherent circuit states such as requests, outputs, and select signals. However, for *reqsIntvls* that are not inherently present in the circuit, instrumentation is required. Requests in the processor typically consist of a data field and a validity field, and they are only processed when the validity field indicates they are valid. Therefore, the inserted code is used to dynamically record the timing intervals between any two distinct valid requests, as well as the intervals between two consecutive valid requests from the same source.

5.2 Filter Out Contention States Without Side-Channel Risks

The goal of Sonar is to identify cases where different secret inputs induce distinct microarchitectural states, enabling information leakage through timing differences. However, among the contention-critical states identified via bottom-up tracing, some states are independent of secrets. For example, a 2:1 MUX is used to select one output from two constants according to the select signal. The requests and *reqsIntvl* at this contention point remain unchanged regardless of the secrets, as the requests are constants and the validity field of the constant is always considered valid. These contention-critical states do not introduce timing differences, thereby posing no risk of information leakage. Consequently, monitoring these states is unnecessary, especially for the *reqsIntvl* that requires instrumentation. Our experiments reveal that, on average, 31% of the contention points identified through bottom-up tracing fall into this category. Filtering out these constant states that pose no side-channel risk is crucial for reducing instrumentation overhead and ensuring testing efficiency.

If states are constant, they are not associated with side-channel risks, and we filter out such states accordingly. It is straightforward to distinguish whether requests, select signals and the outputs are constants due to the fixed format of constants in the circuit. However, determining whether *reqsIntvl* is constant through static analysis is challenging, as it is a runtime-collected state. Fortunately, we observe that when requests at a contention point are input-dependent, they are typically marked by a valid signal. Building on this observation, during bottom-up tracing, we check whether each request has a valid signal. If no valid signals are present for any request at a contention point, we consider the requests to be valid during all clock cycles, and the *reqsIntvl* is always a constant 0. Dynamic monitoring of the *reqsIntvl* at these points is meaningless.

To achieve this goal, we design the request validity determination algorithm, as delineated in Algorithm 1. According to circuit programming specifications, the validity and data fields of a request typically share the same prefix, which indicates that they belong to the same request. For example, in the ROB module of BOOM [62], an instruction commit request includes a validity field io_commit_valid and a data field io_commit_uops_inst. Both fields share the same prefix io_commit. Therefore, pattern matching can first be used to identify validity signals sharing the same prefix as a request's data field (line 3 in Algorithm 1). If no match is found, we trace back to the data field's source signals, whose validity may indirectly indicate the request's validity. If validity fields for these sources are found, the request's validity is the bitwise AND of all source validities (line 4-7 in Algorithm 1). If no validity field can be determined by the above methods, the request is considered constantly valid.

Algorithm 1: Request Validity Determination Logic

```
   Input: 𝒟, data field of request.
   Output: 𝒱, valid field of request.
1  function Main(𝒟):
2      𝒱 ← null;
       /* Look for a validity signal with the same prefix as 𝒟        */
3      𝒱 ← PrefixMatchedValid(𝒟);
4      if 𝒱 is null then
5          Srcs ← FindSrcs(𝒟);
6          foreach src ∈ Srcs do
7              𝒱 ← 𝒱 ∧ Main(src);
8          end
9      end
10     return 𝒱;
```

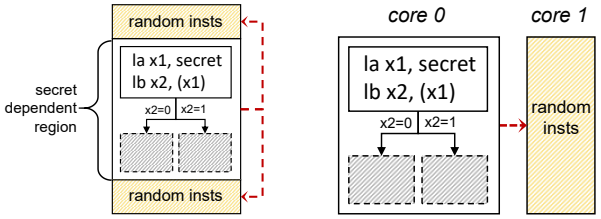
6 Microarchitectural States-Guided Contention Triggering

In typical hardware fuzzers, potential bugs are often uncovered through the generation of random instruction sequences. However, many side channels originate from specific changes in microarchitectural states, which are difficult to induce accurately using random instruction generation and mutation. Without monitoring contention states, it is challenging to use a random method to guide multiple requests to converge at a contention point, even within the same cycle. Therefore, we propose a method to observe microarchitectural states and leverage these states to guide testcase generation, effectively triggering contention.

6.1 Testcase Template and Contention States Monitoring

Sonar is designed to effectively identify contention side channels that can leak secret information. This requires that the testcases meet the following criteria: ① Induce microarchitectural contention state variations under different secret values; ② Support for the detection of volatile and persistent side channels in both single-core and dual-core scenarios. To this end, we design a testcase template as shown in Figure 4. For the single-core scenario, as illustrated in Figure 4a, any instruction preceding or following the secret-dependent instruction region may contend with the secret-dependent instructions, potentially affecting their execution time.

For the dual-core scenario, as shown in Figure 4b, the victim executes secret-dependent instructions on one core, while the attacker on the other core frequently executes instructions that may contend with the victim. Depending on the secret value, the presence or absence of contention between the victim and attacker causes variations in the attacker's execution time. These timing differences allow the attacker to infer the secret value.



(a) Single Core (b) Dual-Core

Figure 4: The testcase template in Sonar. The regions indicated by dashed arrow may exhibit observable timing differences due to contention.

It is worth noting that not all contentions have the potential to leak sensitive information and pose a side-channel risk, and only contention involving secret-dependent requests can. We refer to such risky contention as *secret-dependent contention*, while the opposite is secret-independent contention. For example, the contentions occurring in the random instruction region of the template shown in Figure 4 are unlikely to be exploited to infer secrets, as these instructions are not directly related to the secret. Triggering non-risky contention may block the possibility of reaching risky secret-dependent contention at the same contention point, as the process of approaching contention at that point almost halts once contention occurs. Thus, only states associated with risky contention are used to guide the generation of testcases. To accurately monitor the states of secret-dependent contention during testcase execution, we restrict the monitoring window to the clock cycles during which secret-dependent instructions are in-flight, rather than spanning the entire execution cycle of the testcase. In detail, the monitoring window is defined as the clock period starting with the first secret-dependent instruction entering the Reorder Buffer (ROB) and ending with the last one being committed. Within this window, in-flight secret-dependent instructions may induce contentions. Outside this window, the probability of secret-dependent contentions is low.

6.2 States-Guided Testcase Generation for Triggering Contentions

The quality of the seed corpus directly impacts the depth, breadth, and efficiency of fuzzing. Therefore, a significant challenge in fuzzing is designing reasonable seed retention and selection strategies to maintain the quality of the seed corpus. In addition, mutation is an important method for generating diverse new testcases. Although random mutation can explore more execution paths, it is also likely to disrupt the critical structures in the testcases that may affect contention points. Hence, more targeted mutation strategy should also be developed to guide and approach contentions.

6.2.1 Gradual Approach to Trigger Volatile Contentions. Volatile contentions require requests to contend for resources simultaneously, which makes it challenging to generate testcases that meet such strict timing requirements through random generation. Sonar addresses this challenge by employing a gradual approach strategy to achieve the goal: ① Retain the testcase as a seed if it can reduce the *reqsIntvl* at any contention point; ② Prioritize seeds with the smallest *reqsIntvl* for mutation; ③ Guide the mutation of seeds based on the trend of decreasing *reqsIntvl* until it reaches zero, thereby triggering volatile contention.

Seed Retention and Selection: Within the monitoring window, we collect the *reqsIntvl* between any two valid requests at contention points, which may originate from different levels of the MUX tree. The minimum *reqsIntvl* among all request pairs serves as a measure of contention at a contention point and forms the basis for seed retention and selection. If a testcase can reduce the minimum *reqsIntvl* at any contention point, we add it to the seed corpus. During seed selection, we prioritize seeds that are closer to triggering contention. Specifically, contention points with smaller minimum *reqsIntvl* (but not zero) are more likely to be selected as targets for further narrowing. Among the testcases that achieve the same minimum *reqsIntvl* at the target contention point, one is randomly selected for mutation.

Interval-guided Directed Seed Mutation: To make the mutation results more controllable and converging, we propose an adaptive directed mutation strategy. This strategy leverages the effect of the previous mutation to guide the next mutation direction, thereby accelerating the reduction of *reqsIntvl*: if the previous mutation reduces the minimum request interval, maintain the previous mutation direction; otherwise, change the mutation direction. Minimizing the *reqsIntvl* essentially means that the cycles at which requests are valid get closer. There are two key factors that influence the validity of a request: the operand parsing time of instructions and the availability of microarchitectural resources. Fortunately, instruction-level mutations provide a direct and effective way to control operand parsing time by adjusting the dependency chain length, as this time is closely related to the chain length. By increasing or decreasing the dependency chain length, we can regulate the operand resolution time and, in turn, indirectly control the effective timing of requests. We observe that inserting or removing instructions only at the head of the dependency chain has a monotonic and uniform impact on the parsing time of all downstream instructions in the chain: adding instructions at the head delays the parsing time of downstream instructions, while removing them advances it. This monotonicity and continuity provide a clear basis for mutation directions. By determining whether the previous mutation reduced the *reqsIntvl*, we can assess the effectiveness of the mutation direction and dynamically adjust it. This adaptive adjustment strategy enables a faster reduction of the *reqsIntvl*, allowing us to quickly approach or trigger contention.

6.2.2 Enhance Data Similarity to Investigate Persistent Contentions. Persistent contentions require at least two consecutive requests on the same request path. This necessitates observing at least one valid request within the monitoring window, ensuring that contention involves secret-dependent instructions. The other valid request may occur either within or outside the monitoring window. Besides,

triggering persistent contentions imposes specific requirements on the data fields of requests. Based on existing research, the data fields typically need to be similar or identical to map to the same storage unit. Therefore, in addition to the previously mentioned directed mutation ensuring at least one request appears within the monitoring window, we specifically adjust the operands of the same type of instruction during instruction mutation to enhance the data similarity between requests.

7 Contention Side Channels Detection and Analysis

To detect contention side channels, merely triggering contention is insufficient. Contentions that lead to side channels must exhibit observable timing differences under different secret values, which our detection mechanism relies upon. Therefore, we first identify the timing differences induced by side channels. Then we automatically locate the potential root causes of the contention that may lead to these timing differences. After detecting contention side channels, we design attack templates to analyze and explore their exploitability.

7.1 Identification of Side Channels

To comprehensively detect side channels exposed by testcases, we consider all regions potentially affected by contentions. These regions include the instruction regions pointed to by the dashed arrows in Figure 4, covering both single-core and dual-core scenarios. Examining the execution time of the whole region may lead to missed side channels due to the cancellation of timing effects. To address this, we adopt a finer-grained, instruction-level detection method by analyzing the execution time of each instruction under different secret values. We observe that requests affected by contentions are directly reflected in the commit time of their corresponding instructions. Thus, we detect side channels by comparing whether the commit times of the same instruction differ under varying secret values.

However, detecting side channels solely by comparing instruction commit time differences is inaccurate. Instruction commit time differences may result directly from side channels or indirectly from delays in preceding instructions. For example, in the top table of Figure 5, the commit times of the div and mul instructions differ under different secret values. However, only the div instruction is genuinely affected by the side channel, causing a 1-cycle delay when the secret value is 1. The mul instruction itself is unaffected but is delayed because it must wait for the div instruction to complete commit, resulting in the same delay. In practice, if timing differences are solely caused by the in-order commit mechanism, the relative commit time differences between adjacent instructions remain unchanged under different secret values. Based on this observation, we propose the *commit cycle difference (CCD)* metric as an indirect measure of instruction execution time. By comparing whether the CCD changes with secret values, we can identify instructions genuinely affected by side channels. This approach effectively filters out timing differences unrelated to side channels, preventing wasted time and interference with the subsequent analysis.

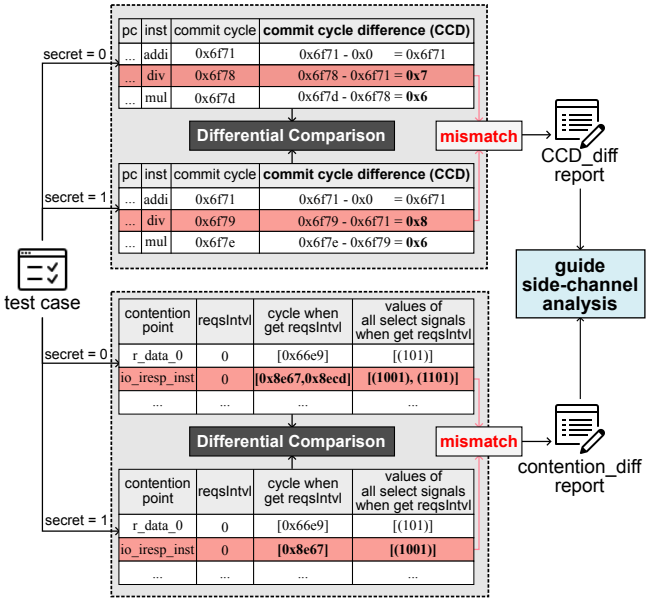


Figure 5: Accurate contention side channel detection based on dual-differential comparison.

7.2 Justification of Contention Side Channels

Even after identifying instructions genuinely affected by side channels, attributing timing differences to specific resource contentions is still challenging and laborious. Because instructions often pass through complex pipeline paths with multiple contention points, exhaustive inspection is inefficient. The essence of contention side channels is that different secrets trigger variations in microarchitectural contention states, which have observable impacts on instruction execution. In other words, only contention points with state differences under different secret values can lead to contention side channels. Thus, we focus on discrepencies in contention states, as shown in the table at the bottom of Figure 5. During testcase execution, we collect contention-critical microarchitectural states within the monitoring window at each contention point and perform differential comparisons to identify state deviations, generating a report that documents these discrepancies.

Ultimately, the CCD differential comparison pinpoints the timing differences directly affected by side channels, while the differential comparison of contention states records the specific contentions likely responsible for those timing differences. Together, these two reports enable rapid identification and justification of contention side channels.

7.3 Exploitability Analysis

After detecting contention side channels, we further investigate their exploitability. Based on the testcase template in Figure 4, we extend it into a Meltdown-like [63] attack template, as shown in Listing 1. At line 6, privileged secret data is accessed illegally, inducing an access fault exception. However, due to lazy exception handling, the processor continues to execute subsequent instructions. During this period, instructions related to the secret may

cause contention with other instructions. Under different secret values, the execution time may vary. By leveraging this characteristic, attackers can infer secrets accordingly. For example, at line 6, *access(secret_addr)* reads kernel secret data bit by bit. If *secret[i]* is 0, no contention occurs with the instruction at *line 5*; if *secret[i]* is 1, contention occurs, resulting in longer execution time. Thus, longer execution time is inferred as *secret[i] = 1*, while shorter time is inferred as *secret[i] = 0*.

Listing 1: Meltdown-style attack template for contention side channels exploitability analysis.

```
1  for (int i = 0; i < secret_bits; i++) {
2      for (int t = 0; t < train_times; t++) {
3          int t1 = rdcycle();
4          operand = delay_inst();  // Delay execution of affected
               instructions
5          affected_inst(operand)
6          access(secret_addr); // access secret to trigger transient
               execution
7          if (extract_bit(secret, i))
8              contentionPath(); // Encode '1' via contention
9          else
10             no_contentionPath(); // Encode '0'
11         int timediff = rdcycle() - t1;
12     }
13     Secret[i] = (timediff > threshold) ? 1 : 0;
14 }
```

Additionally, to increase the likelihood of resource contention between the instructions at line 5 and line 6 within the transient window, we insert a block of computation instructions at line 4 to delay the operand resolution time of the affected instructions. Using this attack template, we can conduct a targeted analysis of the identified contention side channels to assess their exploitability.

8 Evaluation

We implemented Sonar, which includes identification and monitoring of contention-critical microarchitectural states, guided fuzzing based on these states, and accurate side channel detection and analysis. In this section, we present the evaluation of Sonar on open-source RISC-V processors.

8.1 Evaluation Setup

Configuration of DUTs. We evaluate Sonar on two open-source out-of-order RISC-V processors with varying complexity: **BOOM**[62] and **NutShell**[64]. Table 1 summarizes their key configuration parameters.

Fuzzing Environment: We compile BOOM and NutShell using the open-source RTL simulator Verilator [65] and integrate BOOM with memory systems and peripherals via Chipyard [66]. All evaluation experiments are conducted on a machine equipped with an Intel Xeon Platinum 8380 CPU running at 2.30 GHz, with 80 CPU cores and 962 GiB of RAM.

8.2 Contention States Identification and Filtering

We choose FIRRTL[67], a circuit intermediate representation, for processor analysis. It serves as a transitional layer between the high-level design in Chisel[68] and low-level implementation such as Verilog, preserving rich structural details of the design. FIRRTL enables implementation-independent analysis and circuit modifications that can be easily extended to other designs.

Table 1: Key parameters of BOOM and NutShell.

Feature	BOOM	NutShell
Supported ISA	RV64GC	RV64 IMAC/Zicsr/Zifencei
Privilege	U/S/M	U/S/M
Pipeline Stages	10	9
Fetch Width	8	2
Fetch Buffer	24	8
BrPred	uBTB+BTB+TAGE	BTB+PHT
Int/Fp PhyRegs	100/96	32/-
Mem/Fp/Int Func	1/1/3	1/-/2
ROB Entry	96	32
Ld/St Queue	24/24	-/8
I/DCache	32/32KB	32/32KB
L1 MSHR	2	-
L2 Cache	512 KB	128 KB
Bus Protocol	TileLink	SimpleBus+AXI4

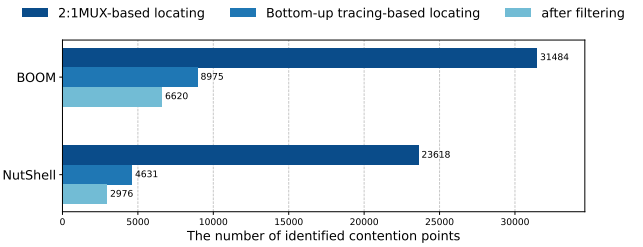


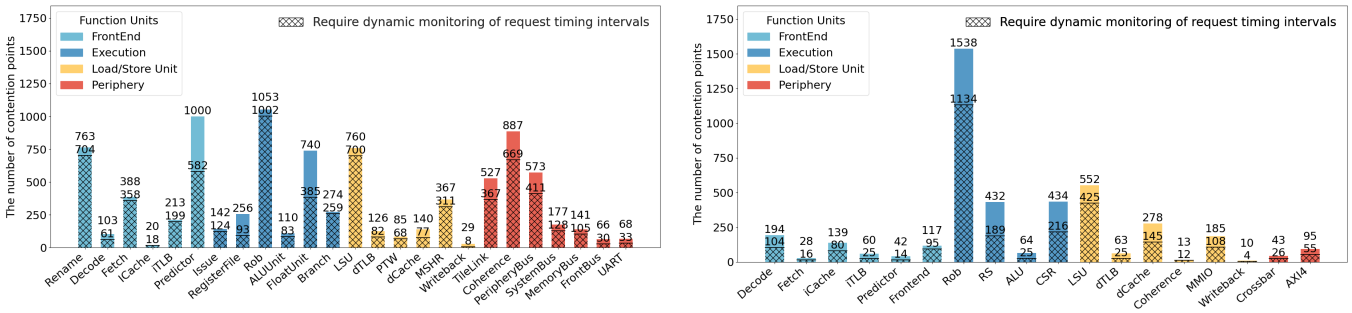
Figure 6: The number of identified contention points with different strategies.

Identification and Distribution of Contention Points. MUXes are hotspots for contention. However, simply treating every 2:1 MUX as a contention point would greatly overestimate the true count, because many 2:1 MUXes only partially contribute to contention. Our MUX-based bottom-up method can accurately identify true contention points, including those implemented through cascaded MUXes. Compared to the 2:1 MUX-based method, our MUX-based bottom-up tracing method significantly reduces the number of identified points: from 31,484 to 8,975 (71.5% reduction) on BOOM, from 23,618 to 4,631 (80.4% reduction) on NutShell, as shown in Figure 6. The distribution and amount of identified contention points are shown in Figure 7. We find that contention points are mainly concentrated in the frontend, ROB, LSU, and peripheral bus across these two processors.

Filtering States Without Side-Channel Risk: Figure 7 also shows the number of dynamically monitored contention points for the timing interval between requests after implementing the filtering method described in §5.2. On BOOM, the number of contention points is reduced from 8,975 to 6620 (26.2% reduction); on NutShell, from 4,631 to 2976 (35.7% reduction).

8.3 Comprehensive Evaluation of Sonar

8.3.1 ***Overhead Analysis of Instrumentation.*** To collect runtime contention states, we instrument the DUTs by adding monitoring logic to the source code through custom transformation passes



(a) Contention points distribution in *BOOM*. **(b) Contention points distribution in *NutShell*.**

Figure 7: Distribution and number of contention points in BOOM and NutShell: before vs. after filtering out states without side-channel risk.

at the FIRRTL stage. As shown in Table 2, the instrumentation introduces compilation overheads of 43% and 45% for BOOM and NutShell, respectively, while causing simulation speed slowdowns of 26% and 38%. The Verilog code sizes for BOOM and NutShell are 550k and 230k lines, with instrumentation accounting for 14% and 20% of the total code, respectively. Nevertheless, the overall fuzzing speed remains acceptable. Notably, the instrumentation does not affect the functional behavior of the DUTs, thus the execution results of testcases are consistent before and after the instrumentation.

Table 2: Instrumentation overhead of Sonar. Numbers in parentheses indicate the percentage overheads compared to the original.

DUT	Contention points	Compile time(s)	#New verilog	Simulation speed(Hz)*	Fuzzing speed(/hour)
NutShell	4631	470(45%)	34k(20%)	28k(38%)	7596
BOOM	8975	314(43%)	67k(14%)	5.3k(26%)	239

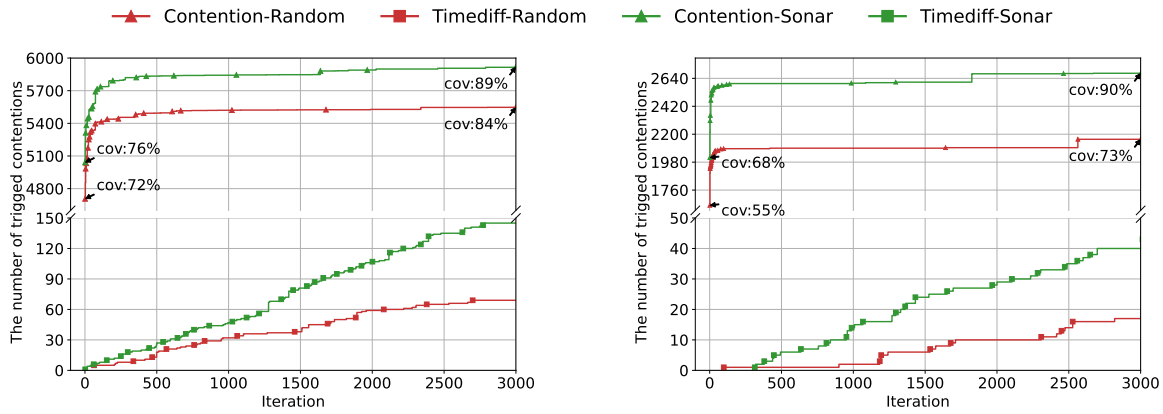
*Simulation speed is measured as cycles per second.

8.3.2 ***Detection Capability Evaluation of Sonar.*** To evaluate the vulnerability detection capability of Sonar, we conduct 3000 iterations of testing on BOOM and NutShell. We also compare Sonaragainst random testing (i.e., Sonar without any guidance) along two dimensions: cumulative contention coverage and the number of timing differences that reflect secret values. The results presented in Figure 8 convey the following information: ① A large number of contentions are triggered in the early stage. Analysis reveals that most are dominated by a single signal, meaning only one request has a valid signal at these contention points. Moreover, we also find that the valid signal also happens to be the request itself, as illustrated in Figure 9. Since these valid signals are easily asserted in the circuit, many contentions are triggered at the outset of testing. ② New contentions tend to appear in clusters, as a single contention event may involve multiple data selections and thus map to several contention points. ③ Sonar consistently outperforms random testing on both processors, with an average 117% increase in triggered contention points and over 210% increase in observed timing differences. ④ Although thousands of contentions are triggered, only 2.4%–7.2% result in observed timing differences. This is because timing differences cannot be observed if contention under different secret values affects instructions identically, or if the affected instructions fall outside the observable region defined by the template. As the number of iterations increases, the number of timing differences continues to grow linearly, suggesting that more risky contentions are likely to be uncovered in the future.

8.3.3 ***Breakdown Experiments.*** We conduct experiments to evaluate the effectiveness of the proposed fuzzing strategies for triggering contentions: (a) seed retention, (b) seed selection (which inherently includes retention), and (c) seed mutation (which essentially includes both retention and selection). As shown in Figure 10, during the early stages of testing, the advantages of these strategies are not pronounced. However, their benefits become evident as testing progresses. The results show that our proposed directed seed mutation strategy can effectively enhance the ability to trigger contentions.

8.3.4 ***Comparison with SpecDoctor.*** We compare Sonar with SpecDoctor[25], an open-source pre-silicon side channel fuzzer, in terms of their ability to trigger contentions. As shown in Figure 11, under the same iterations, Sonar triggers 2.13× more new contention points than SpecDoctor, showing stronger contention triggering ability and greater potential for side channel discovery. Moreover, SpecDoctor's instrumentation has $O(n^2)$ time complexity, where n is the number of FIRRTL statements in a module, making it impractical for large-scale designs. While Sonar's instrumentation runs in $O(n)$ time, offering greater scalability for complex designs.

8.3.5 ***Effectiveness of Dual Differential Comparison-Assisted Side Channel Analysis.*** In practice, our dual differential comparison significantly speeds up the debugging of timing differences. For example, when identifying execution port contention side channel on BOOM, this method reduces debugging time from one hour to just ten minutes, greatly enhancing contention side-channel analysis efficiency.



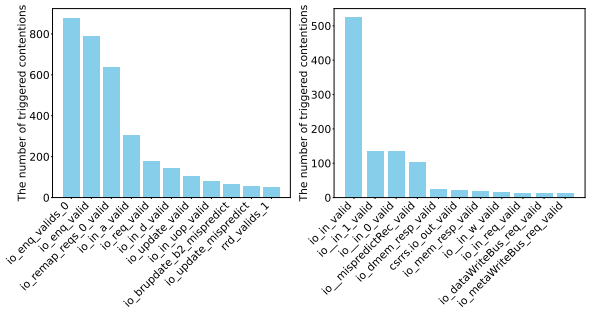
(a) BOOM. (b) NutShell.

Figure 8: Comparison of the number of triggered contentions and timing differences on BOOM and NutShell using Sonar and random testing.

Table 3: List of contention-based side channels found by Sonar.

Processor	Shared Resource	#	Description of Contention Side Channels	New?	Time Difference	Accuracy
BOOM	TileLink†	S1	The younger *ICache read* instruction blocks the older *DCache read/writeback* instruction due to TileLink D-Channel contention.	✓	40 cycles	>99%
		S2	The younger *ICache read* instruction blocks the older *ICache read/writeback* instruction due to TileLink D-Channel contention.	✓	32-37 cycles	>99%
		S3	Due to TileLink D-Channel contention, the younger *DCache read* instruction blocks the older *ICache read/writeback* instruction.	✓	1-38 cycles	>99%
		S4	Due to TileLink D-Channel contention, the younger *DCache read* instruction blocks the older *DCache read/writeback* instruction.	✓	9 cycles	>99%
	MSHR	S5	The younger load instruction occupies an MSHR and blocks the older one because their addresses have the same set index but different tags.	✓	40 cycles	>99%
	LineBuffer	S6	When a younger and an older load instruction access the read linebuffer simultaneously, the younger one is prioritized, delaying the older one.	✓	9 cycles	>99%
		S7	When a younger and an older store instruction access the write linebuffer simultaneously, the younger one is prioritized, delaying the older one.	✓	2-8 cycles	>96%
	EXE Unit	S8	When requests from alu, imul, and div simultaneously contend for the response port of the execution unit, the request from alu is prioritized, while others are delayed.	✗	1-11 cycles	-
	Div Unit	S9	The younger division instruction blocks the older one by entering the execution unit first.	✗	57-70 cycles	-
	L1 DCache	S10	The younger store conditional instruction writes data to cache and marks it dirty regardless of success, delaying older instructions accessing the same cacheline due to the required cache writeback.	✗	12-31 cycles	-
		S11	The younger and older instructions access the same cacheline, with the younger instruction executes first, causing the older instruction to hit in the cache and thus be executed faster.	✓	59 cycles	>99%
		S12	The younger load instruction loads data into the cache and evicts a cacheline that is needed by the older load instruction, causing the older instruction to be delayed.	✓	18 cycles	>94%
Nutshell	MDU	S13	Multiplication and division instructions share the Multiply-Divide Unit (MDU), which is non-pipelined. When a younger multiplication instruction occupies the MDU, the older division instruction is blocked.	✓	4-63 cycles	<2%
	L1 ICache	S14	Contention on the shared read/write port of the L1 ICache can delay instruction fetches.	✓	8 cycles	<2%

†Side channels due to contention on TileLink can also be observed in the dual-core scenario.



(a) BOOM. (b) NutShell.

Figure 9: Dominance of single valid signals in contentions triggered by the first 20 testcases.

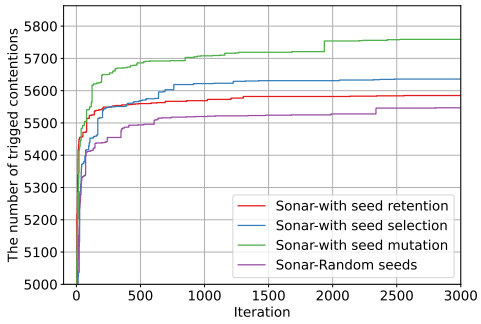


Figure 10: Effectiveness of each strategy of triggering contentions in Sonar on BOOM.

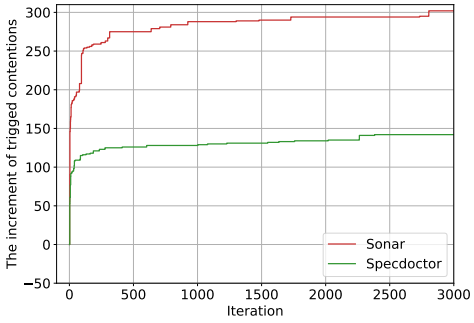


Figure 11: Comparison of Sonar and SpecDoctor in terms of the number of newly triggered contentions on BOOM.

8.4 Side Channels Uncovered by Sonar

Sonardetected **12** contention side channels on BOOM and **2** on NutShell, **11** of which are newly discovered. Table 3 provides a brief description of these side channels and their resulting timing differences in instruction execution. Based on the shared resources, we categorize these side channels into 6 types. Next, we describe these side channels in detail and highlight their novelty.

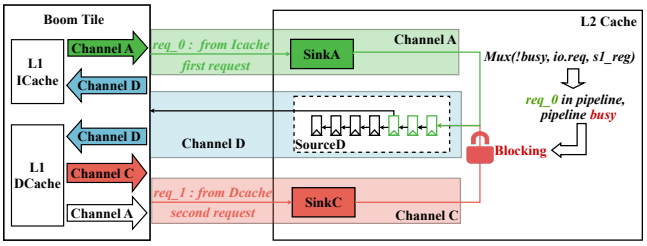


Figure 12: Mechanism of side channels S1-S4, which triggers contentions on TileLink D-Channel.

A. TileLink Contention (S1-S4)

BOOM employs TileLink [69] as its on-chip interconnect protocol. Data transfers between the L1 and L2 caches are routed through the D-Channel. A cache read request occupies the D-Channel for 8 cycles to return a cacheline of data, while a cache writeback request takes only 1 cycle. As shown in Figure 12, a younger instruction's iCache read request *req_0* blocks an older instruction's DCache writeback request *req_1*, delaying its response due to channel occupancy (S1). A TileLink-based contention side channel arises whenever at least one of the requests is a cache read that occupies the D-Channel for multiple cycles, blocking the other request. Cache read requests may originate from either the L1 ICache or DCache. Consequently, similar side channels are also observed in other instruction patterns, as described by S2–S4 in Table 3. In practice, timing differences may be further amplified by subsequent instructions, often exceeding 8 cycles.

Unlike Intel's On-chip InterConnect mechanism [70, 71], contention in TileLink does not stem from complex multi-channel arbitration or resources shared across cores. Instead, it can be triggered by intra-thread timing conflicts within the instruction pipeline, without relying on cross-core or LLC-slice sharing.

B. MSHR Contention (S5)

MSHRs are used to track and manage outstanding cache miss requests. There are two management modes in BOOM: *pri*, for new requests or those unable to reuse existing MSHRs; and *sec*, for requests that attempt to reuse an existing MSHR. Consider two cache miss requests , *req_0* and *req_1*, that share the same setidx (the address field identifying a cache set) but differ in tag. The first request *req_0* occupies an MSHR for a cache miss. When *req_1* arrives, MSHRs attempt to process it via the *sec* mode, as there is an ongoing *req_0* mapped to the same cache set. However, the tag mismatch indicates they do not share the same cacheline, so reuse fails. As a result, *req_1* must wait until *req_0* completes before being accepted in *pri* mode, as depicted in Figure 13. According to our observation, this delay can cause a stall of 40 cycles.

Unlike Speculative Interference Attacks[11], which attribute blocking to the full MSHRs occupancy, we uncover a distinct case in BOOM: requests can be blocked even when MSHRs are available, due to the setidx match but tag mismatch. We first identify and characterize this scenario, and refer to it as false sharing path blocking.

C. Read/Write LineBuffer Contention (S6-S7)

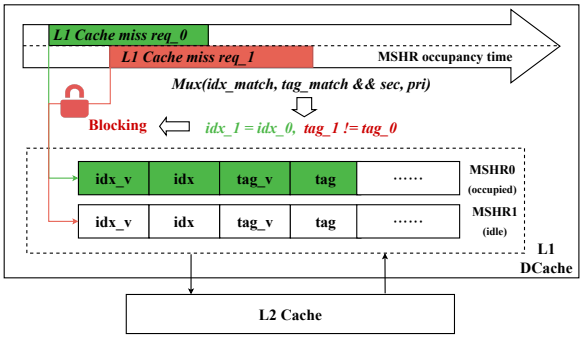


Figure 13: Mechanism of triggering contention side channel S5 on MSHR.

The LineBuffer is designed to minimize direct memory interactions per access. In BOOM, the read and write LineBuffers are used to temporarily hold data read from the L2 Cache and data to be written back to the L2 Cache, respectively. However, when two read/write requests are issued to the read/write LineBuffer simultaneously (i.e., the same clock cycle), only one can be serviced at a time, causing the other request to be stalled for one cycle. This results in delayed instruction commit times.

Unlike RIDL[72], which exploits the property that a LineBuffer entry may contain data from previous loads, our finding reveal that the timing characteristics of simultaneous accesses to the read/write LineBuffer can also lead to side-channel risks.

D. L1 DCache Contention (S10-S12)

Within a single hardware thread, two load instructions access the same L1 DCache line, with the younger instruction executing first. When the younger instruction encounters a cache miss, the older instruction will benefit from the resolved miss and thus executes faster. Without the younger instruction executing first, the older one has to handle the cache miss by itself, resulting in longer execution time (S11). Similarly, when the younger load instruction hits an L1 DCache miss while the DCache is full. A cacheline must be evicted after fetching the data from memory. Unfortunately, the evicted line is exactly the one the older load instruction needs, resulting in an extra cache miss. The older load must spend more time handling the cache miss (S12). Intuitively, S11 and S12 resemble attacks such as Flush+Reload[34] and Prime+Probe[12]. The difference is that S11 and S12 do not rely on multithreading. Sonar unveils that cache-based side channel attacks can be carried out on a single hardware thread.

In addition, Sonar also find the persistent side channel S10 caused by store conditional instructions mapped to the same cacheline , which was first revealed by SIGFuzz [26].

E. L1 ICache Contention (S14)

During instruction fetch, the ICache in NutShell does not support simultaneous reads and writes. When read or write requests contend for the ICache port, some requests are delayed, resulting in timing differences in instruction execution. The measured delay is approximately 8 cycles in our testing.

F. Execution Unit Contention (S8, S9, S13)

The Multiply-Divide Unit (MDU) in Nutshell is a non-pipelined execution unit that handles both multiplication and division instructions. When a younger instruction occupies the MDU ahead of an older one, the older instruction is forced to wait until the MDU becomes available. This contention for the MDU causes delays in committing the older instruction, forming the side channel S13. Contention can occur between any multiplication or division instructions. The observed delay ranges from 4 to 63 cycles, depending on the execution times of various instructions.

Moreover, Sonar can also discover the execution unit writeback port contention side channel S8 in BOOM in as little as 10 minutes, which was first found by UPEC [23]. While Specdoctor [25] revealed execution unit contention caused by division instructions in NutShell, Sonar also can detect the same side channel S9 in BOOM.

8.5 Exploitation of Unveiled Contention Side Channels

Based on the exploitation template in §7.3, we evaluate the exploitation of new contention side channels listed in Table 3. On BOOM, we successfully construct Meltdown-like exploitation PoC (proof-of-concept) for S1–S7 and S11–S12. After executing each PoC 1,000 times, the results show that the inferred accuracy for a consecutive 128-bit key exceeds 99% in all cases. Notably, S7 achieves slightly lower accuracy due to its observable timing difference being only 2–8 cycles, which is more susceptible to noise interference. Additionally, for S12, the random nature of cache eviction leads to a low probability for triggering the contention scenario, resulting in a lower success rate. We also try to construct PoC attacks for the two side channels S13 and S14 on NutShell. However, the secret accuracy rate is below 2%. We attribute this mainly to NutShell's earlier exception detection in the pipeline, which causes exceptions to be handled before the contention side channel is established. Nonetheless, we believe there remains potential for further exploitation and plan to explore alternative attack methods beyond Meltdown to disclose sensitive information in future work.

8.6 Mitigation Strategies

Both volatile and persistent side channel attacks rely on timing measurements. Therefore, the most direct mitigation against contention-based side channels is to restrict access to clock registers, preventing attackers from obtaining precise timing information and thus reducing the accuracy of information leakage[73].

Many volatile side channel attacks rely on simultaneous multithreading (SMT); thus, disabling SMT can effectively mitigate such attacks. Besides, SecSMT[74] proposes partitioning pipeline resources to eliminate resource contention between SMT threads. In addition, obfuscated execution[75, 76], which increases execution path uncertainty, can further reduce attack success rate.

For persistent side channel attacks, resource isolation is an effective mitigation strategy to block attack vectors. For example, Catalyst[77] proposes cache isolation by allocating separate cache resources to different applications, preventing information leakage through shared components. Randomization techniques, such as

random indexing[78, 79] and random padding[80], introduce uncertainty in data placement and access patterns, making it harder for attackers to infer sensitive information.

9 Related Work

In recent years, several fuzzing-based side channel detection works have emerged at the pre-silicon stage. For example, SpecDoctor[25] designs a multi-stage fuzzing template that detects timing side channels in transient scenarios by analyzing execution time differences after secret-dependent data transfers. WhisperFuzz[27] integrates static analysis to detect and localize timing vulnerabilities, and SIGFuzz[26] identifies side channels by monitoring the impact of other instructions on commit timing. However, these approaches fundamentally rely on randomly generated and mutated testcases, lacking feedback and guidance from contention-related states, which makes it difficult to effectively trigger complex or strict contentions. By comparison, Sonar automatically identifies, collects, and leverages contention-critical microarchitectural states to guide fuzzing, enabling targeted testcase mutation and more effective triggering of contention.

Formal methods like UPEC[23] have also been designed to detect side channels, but they require custom models, resulting in high manual costs and limited scalability to large, complex designs. Due to the linear complexity of the contention states identification, Sonar can be applied to large-scale and complex designs, offering end-to-end capability for discovering complex vulnerabilities.

Additionally, some works focus on post-silicon timing side channel discovery. Revizor[54] targets the L1 DCache by detecting information leakage through checking consistency of hardware traces under the same contract trace. Osiris[81] detects side channels using a triple-instruction model. ABSynthe[82] proposes an automated method for contention side channels, but it is designed for post-silicon stage and relies on real hardware. In comparison, Sonar aims to uncover contention side channels caused by arbitrary resources and instruction sequence lengths at pre-silicon stage, enabling earlier vulnerability detection and mitigation.

10 Conclusion

In this paper, we present Sonar, the first automated pre-silicon fuzzing framework for detecting contention-based side channels in processors. Sonar systematically identifies contention-critical microarchitectural states and leverages them to guide the generation of testcases, efficiently triggering contention. This approach bridges the gap between randomized testcases at the architectural level and the precise triggering of microarchitectural events. Furthermore, the proposed dual-differential comparison mechanism can facilitate rapid identification of contention-based side channels. Evaluation on two real-world RISC-V processors BOOM and NutShell reveals 14 contention side channels, 11 of which are previously unknown, spanning multiple pipeline stages and components. We also further analyze their exploitability using our designed Meltdown-style attack template.

Acknowledgments

We sincerely appreciate the constructive feedback from the anonymous reviewers, which helped to further enhance the manuscript. We are also grateful to all co-authors for their efforts and contributions. Moreover, we would like to express our special thanks to the XiangShan team for their valuable support of this work, especially Prof. Yungang Bao and Miaomiao Yuan from the Institute of Computing Technology, Chinese Academy of Sciences (ICT, CAS), and Prof. Dan Tang from the Beijing Institute of Open Source Chip (BOSC). Besides, this work is supported by the Joint Funds of the National Natural Science Foundation of China under Grant No.U24A6009, the National Science Fund for Distinguished Young Scholars under Grant No.62125208 and the Young Scientists Fund of the National Natural Science Foundation of China under Grant No.62202467.

References

[1] Yuval Yarom, Daniel Genkin, and Nadia Heninger. 2017. CacheBleed: a timing attack on OpenSSL constant-time RSA. *Journal of Cryptographic Engineering* 7 (2017), 99–112.

[2] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. 2019. Smotherspectre: exploiting speculative execution through port contention. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 785–800.

[3] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. 2019. Port contention for fun and profit. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 870–887.

[4] Jacob Fustos, Michael Bechtel, and Heechul Yun. 2020. SpectreRewind: Leaking secrets to past instructions. In *Proceedings of the 4th ACM Workshop on Attacks and Solutions in Hardware Security*. 117–126.

[5] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 557–574. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/lee-sangho

[6] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software Grand Exposure: SGX Cache Attacks Are Practical. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*. USENIX Association, Vancouver, BC. https://www.usenix.org/conference/woot17/workshop-program/presentation/brasser

[7] Tianlin Huo, Xiaoni Meng, Wenhao Wang, Chunliang Hao, Pei Zhao, Jian Zhai, and Mingshu Li. 2019. Bluethunder: A 2-level Directional Predictor Based Side-Channel Attack against SGX. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2020, 1 (Nov. 2019), 321–347. https://doi.org/10.13154/tches.v2020.i1.321-347

[8] Yun Chen, Lingfeng Pei, and Trevor E Carlson. 2023. AfterImage: Leaking control flow data and tracking load operations via the hardware prefetcher. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 16–32.

[9] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2015. Covert channels through branch predictors: a feasibility study. In *Proceedings of the fourth workshop on hardware and architectural support for security and privacy*. 1–8.

[10] Stefan Gast, Jonas Juffinger, Martin Schwarzl, Gururaj Saileshwar, Andreas Kogler, Simone Franza, Markus Köstl, and Daniel Gruss. 2023. Squip: Exploiting the scheduler queue contention side channel. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2256–2272.

[11] Mohammad Behnia, Prateek Sahu, Riccardo Paccagnella, Jiyong Yu, Zirui Neil Zhao, Xiang Zou, Thomas Unterluggauer, Josep Torrellas, Carlos Rozas, Adam Morrison, et al. 2021. Speculative interference attacks: Breaking invisible speculation schemes. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 1046–1060.

[12] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache attacks and countermeasures: the case of AES. In *Topics in Cryptology–CT-RSA 2006: The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2005. Proceedings*. Springer, 1–20.

[13] Li-Chung Chiang and Shih-Wei Li. 2025. Reload+Reload: Exploiting Cache and Memory Contention Side Channel on AMD SEV. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 1014–1027. https://doi.org/10.1145/3676641.3716017

[14] Ahmad Moghimi, Jan Wichelmann, Thomas Eisenbarth, and Berk Sunar. 2019. Memjam: A false dependency attack against constant-time crypto implementations. *International Journal of Parallel Programming* 47 (2019), 538–570.

[15] Ralf Hund, Carsten Willems, and Thorsten Holz. 2013. Practical timing side channel attacks against kernel space ASLR. In *2013 IEEE Symposium on Security and Privacy*. IEEE, 191–205.

[16] Thomas Bourgeat, Ilia Lebedev, Andrew Wright, Sizhuo Zhang, Arvind, and Srinivas Devadas. 2019. MI6: Secure Enclaves in a Speculative Out-of-Order Processor. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 42–56. https://doi.org/10.1145/3352460.3358310

[17] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. 2018. DAWG: A defense against cache timing attacks in speculative execution processors. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 974–987.

[18] Sam Ainsworth and Timothy M. Jones. 2020. MuonTrap: preventing cross-domain spectre-like attacks by capturing speculative state. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture* (Virtual Event) *(ISCA '20)*. IEEE Press, 132–144. https://doi.org/10.1109/ISCA45697.2020.00022

[19] Khaled N. Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2019. SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*. 1–6.

[20] Sam Ainsworth. 2021. GhostMinion: A Strictness-Ordered Cache System for Spectre Mitigation. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 592–606. https://doi.org/10.1145/3466752.3480074

[21] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W. Fletcher. 2019. Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 954–968. https://doi.org/10.1145/3352460.3358274

[22] Kevin Loughlin, Ian Neal, Jiacheng Ma, Elisa Tsai, Ofir Weisse, Satish Narayanasamy, and Baris Kasikci. 2021. {DOLMA}: Securing speculation with the principle of transient {Non-Observability}. In *30th USENIX Security Symposium (USENIX Security 21)*. 1397–1414.

[23] Mohammad Rahmani Fadiheh, Alex Wezel, Johannes Müller, Jörg Bormann, Sayak Ray, Jason M Fung, Subhasish Mitra, Dominik Stoffel, and Wolfgang Kunz. 2022. An exhaustive approach to detecting transient execution side channels in RTL designs of processors. *IEEE Trans. Comput.* 72, 1 (2022), 222–235.

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

[25] Jaewon Hur, Suhwan Song, Sunwoo Kim, and Byoungyoung Lee. 2022. SpecDoctor: Differential fuzz testing to find transient execution vulnerabilities. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 1473–1487.

[26] Chathura Rajapaksha, Leila Delshadtehrani, Manuel Egele, and Ajay Joshi. 2023. SIGFuzz: A framework for discovering microarchitectural timing side channels. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1–6.

[27] Pallavi Borkar, Chen Chen, Mohamadreza Rostami, Nikhilesh Singh, Rahul Kande, Ahmad-Reza Sadeghi, Chester Rebeiro, and Jeyavijayan Rajendran. 2024. Whisperfuzz: White-box fuzzing for detecting and locating timing vulnerabilities in processors. *arXiv preprint arXiv:2402.03704* (2024).

[28] Riccardo Paccagnella, Licheng Luo, and Christopher W Fletcher. 2021. Lord of the ring (s): Side channel attacks on the CPU On-Chip ring interconnect are practical. In *30th USENIX Security Symposium (USENIX Security 21)*. 645–662.

[29] Dean M Tullsen, Susan J Eggers, and Henry M Levy. 1995. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd annual international symposium on Computer architecture*. 392–403.

[30] Thomas Rokicki, Clémentine Maurice, and Michael Schwarz. 2022. CPU port contention without SMT. In *European Symposium on Research in Computer Security*. Springer, 209–228.

[31] Jiliang Zhang, Congcong Chen, Jinhua Cui, and Keqin Li. 2024. Timing Side-channel Attacks and Countermeasures in CPU Microarchitectures. *ACM Comput. Surv.* 56, 7, Article 178 (April 2024), 40 pages. https://doi.org/10.1145/3645109

[32] Zhen Hang Jiang and Yunsi Fei. 2017. A novel cache bank timing attack. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 139–146.

[33] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+Flush: A Fast and Stealthy Cache Attack. In *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721* (San Sebastián, Spain) *(DIMVA 2016)*. Springer-Verlag, Berlin, Heidelberg, 279–299. https://doi.org/10.1007/978-3-319-40667-1_14

[34] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: a high resolution, low noise, L3 cache side-channel attack. In *Proceedings of the 23rd USENIX Conference on Security Symposium* (San Diego, CA) *(SEC'14)*. USENIX Association, USA, 719–732.

[35] Samira Briongos, Pedro Malagón, José M Moya, and Thomas Eisenbarth. 2020. {RELOAD+ REFRESH}: Abusing Cache Replacement Policies to Perform Stealthy Cache Attacks. In *29th USENIX Security Symposium (USENIX Security 20)*. 1967–1984.

[36] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. 2021. Prime+ Scope: Overcoming the observer effect for high-precision cache contention attacks. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2906–2920.

[37] Moritz Lipp, Vedad Hadžić, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. 2020. Take a way: Exploring the security implications of AMD's cache way predictors. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. 813–825.

[38] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2018. Translation leak-aside buffer: Defeating cache side-channel protections with {TLB} attacks. In *27th USENIX Security Symposium (USENIX Security 18)*. 955–972.

[39] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. 2018. Branchscope: A new side-channel attack on directional branch predictor. *ACM SIGPLAN Notices* 53, 2 (2018), 693–707.

[40] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2016. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–13.

[41] Tianwei Zhang, Yinqian Zhang, and Ruby B Lee. 2016. Cloudradar: A real-time side-channel attack detection system in clouds. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 118–140.

[42] Austin Harris, Shijia Wei, Prateek Sahu, Pranav Kumar, Todd Austin, and Mohit Tiwari. 2019. Cyclone: Detecting contention-based cache information leaks through cyclic interference. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 57–72.

[43] Jinhua Cui, Yiyun Yin, Congcong Chen, and Jiliang Zhang. 2023. Spoiler-Alert: Detecting Spoiler Attacks Using a Cuckoo Filter. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 1–6. https://doi.org/10.23919/DATE56975.2023.10137180

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

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

[46] Jinyan Xu, Yiyuan Liu, Sirui He, Haoran Lin, Yajin Zhou, and Cong Wang. 2023. {MorFuzz}: Fuzzing processor via runtime instruction morphing enhanced synchronizable co-simulation. In *32nd USENIX Security Symposium (USENIX Security 23)*. 1307–1324.

[47] Flavien Solt, Katharina Ceesay-Seitz, and Kaveh Razavi. 2024. Cascade: CPU fuzzing via intricate program generation. In *Proc. 33rd USENIX Secur. Symp.* 1–18.

[48] Chen Chen, Rahul Kande, Nathan Nguyen, Flemming Andersen, Aakash Tyagi, Ahmad-Reza Sadeghi, and Jeyavijayan Rajendran. 2023. {HyPFuzz}:{Formal-Assisted} Processor Fuzzing. In *32nd USENIX Security Symposium (USENIX Security 23)*. 1361–1378.

[49] Fabian Thomas, Lorenz Hetterich, Ruiyi Zhang, Daniel Weber, Lukas Gerlach, and Michael Schwarz. 2024. RISCVuzz: Discovering Architectural CPU Vulnerabilities via Differential Hardware Fuzzing.

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

[51] Sujit Kumar Muduli, Gourav Takhar, and Pramod Subramanyan. 2020. Hyperfuzzing for soc security validation. In *Proceedings of the 39th International Conference on Computer-Aided Design*. 1–9.

[52] Sadullah Canakci, Leila Delshadtehrani, Furkan Eris, Michael Bedford Taylor, Manuel Egele, and Ajay Joshi. 2021. Directfuzz: Automated test generation for rtl designs using directed graybox fuzzing. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 529–534.

[53] Yuichi Sugiyama, Reoma Matsuo, and Ryota Shioya. 2023. SurgeFuzz: Surge-Aware Directed Fuzzing for CPU Designs. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 1–9.

[54] Oleksii Oleksenko, Christof Fetzer, Boris Köpf, and Mark Silberstein. 2022. Revizor: Testing black-box CPUs against speculation contracts. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 226–239.

[55] Chen Chen, Vasudev Gohil, Rahul Kande, Ahmad-Reza Sadeghi, and Jeyavijayan Rajendran. 2023. PSOFuzz: Fuzzing processors with particle swarm optimization. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*.

IEEE, 1–9.
[56] Dian-Lun Lin, Yanqing Zhang, Haoxing Ren, Brucek Khailany, Shih-Hsin Wang, and Tsung-Wei Huang. 2023. Genfuzz: Gpu-accelerated hardware fuzzing using genetic algorithm with multiple inputs. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.
[57] Sadullah Canakci, Chathura Rajapaksha, Leila Delshadtehrani, Anoop Nataraja, Michael Bedford Taylor, Manuel Egele, and Ajay Joshi. 2023. Processorfuzz: Processor fuzzing with control and status registers guidance. In *2023 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 1–12.
[58] Gen Zhang, Pengfei Wang, Tai Yue, Danjun Liu, Yubei Guo, and Kai Lu. 2024. INSTILLER: Towards Efficient and Realistic RTL Fuzzing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2024).
[59] Timothy Trippel, Kang G Shin, Alex Chernyakhovsky, Garret Kelly, Dominic Rizzo, and Matthew Hicks. 2022. Fuzzing hardware like software. In *31st USENIX Security Symposium (USENIX Security 22)*. 3237–3254.
[60] Vasudev Gohil, Rahul Kande, Chen Chen, Ahmad-Reza Sadeghi, and Jeyavijayan Rajendran. 2024. Mabfuzz: Multi-armed bandit algorithms for fuzzing processors. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1–6.
[61] Sarah Harris and David Harris. 2021. *Digital Design and Computer Architecture, RISC-V Edition*. Morgan Kaufmann.
[62] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. 2020. Sonicboom: The 3rd generation berkeley out-of-order machine. In *Fourth Workshop on Computer Architecture Research with RISC-V*, Vol. 5. 1–7.
[63] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, Mike Hamburg, and Raoul Strackx. 2020. Meltdown: reading kernel memory from user space. *Commun. ACM* 63, 6 (May 2020), 46–56. https://doi.org/10.1145/3357033
[64] OSCPU. 2024. NutShell. https://github.com/OSCPU/NutShell. Accessed: November 19, 2024.
[65] Stephen Wilson. 2024. Verilator: Open-source SystemVerilog simulator. https://github.com/verilator/verilator. Accessed: 2024-11-16.
[66] UC Berkeley Architecture Research Group. 2024. Chipyard: Integrated Design, Simulation, and Implementation Framework. https://github.com/ucb-bar/chipyard. Accessed: 2024-11-16.
[67] Chips Alliance. 2024. FIRRTL: Flexible Intermediate Representation for RTL. https://github.com/chipsalliance/firrtl. Accessed: 2024-10-17.
[68] Chisel Developers. 2024. Chisel: Constructing Hardware in a Scala Embedded Language. https://www.chisel-lang.org/. Accessed: 2024-10-17.
[69] SiFive. 2021. *TileLink Specification*. https://starfivetech.com/uploads/tilelink_spec_1.8.1.pdf Version 1.8.1.
[70] Riccardo Paccagnella, Licheng Luo, and Christopher W Fletcher. 2021. Lord of the ring (s): Side channel attacks on the {CPU}{On-Chip} ring interconnect are practical. In *30th USENIX Security Symposium (USENIX Security 21)*. 645–662.
[71] Miles Dai, Riccardo Paccagnella, Miguel Gomez-Garcia, John McCalpin, and Mengjia Yan. 2022. Don't mesh around:{Side-Channel} attacks and mitigations on mesh interconnects. In *31st USENIX Security Symposium (USENIX Security 22)*. 2857–2874.
[72] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2019. RIDL: Rogue In-Flight Data Load. In *2019 IEEE Symposium on Security and Privacy (SP)*. 88–105. https://doi.org/10.1109/SP.2019.00087
[73] Robert Martin, John Demme, and Simha Sethumadhavan. 2012. Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. *ACM SIGARCH computer architecture news* 40, 3 (2012), 118–129.
[74] Mohammadkazem Taram, Xida Ren, Ashish Venkat, and Dean Tullsen. 2022. {SecSMT}: Securing {SMT} processors against {Contention-Based} covert channels. In *31st USENIX Security Symposium (USENIX Security 22)*. 3165–3182.
[75] Ashay Rane, Calvin Lin, and Mohit Tiwari. 2015. Raccoon: Closing digital {Side-Channels} through obfuscated execution. In *24th USENIX Security Symposium (USENIX Security 15)*. 431–446.
[76] Jan Wichelmann, Anja Rabich, Anna Pätschke, and Thomas Eisenbarth. 2024. Obelix: Mitigating side-channels through dynamic obfuscation. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 4182–4199.
[77] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. 2016. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *2016 IEEE international symposium on high performance computer architecture (HPCA)*. IEEE, 406–418.
[78] Moinuddin K Qureshi. 2018. CEASER: Mitigating conflict-based cache attacks via encrypted-address and remapping. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 775–787.
[79] Moinuddin K Qureshi. 2019. New attacks and defense for encrypted-address cache. In *Proceedings of the 46th International Symposium on Computer Architecture*. 360–371.
[80] Fangfei Liu and Ruby B Lee. 2014. Random fill cache architecture. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 203–215.
[81] Daniel Weber, Ahmad Ibrahim, Hamed Nemati, Michael Schwarz, and Christian Rossow. 2021. Osiris: Automated discovery of microarchitectural side channels. In *30th USENIX Security Symposium (USENIX Security 21)*. 1415–1432.
[82] Ben Gras, Cristiano Giuffrida, Michael Kurth, Herbert Bos, and Kaveh Razavi. 2020. ABSynthe: Automatic Blackbox Side-channel Synthesis on Commodity Microarchitectures.. In *NDSS*.