SIGFuzz: A Framework for Discovering Microarchitectural Timing Side Channels

Chathura Rajapaksha, Leila Delshadtehrani, Manuel Egele, Ajay Joshi
Department of ECE, Boston University, {chath, delshad, megele, joshi}@bu.edu

***Abstract*—Timing side channels can be inadvertently introduced into processor microarchitecture during the design process, mainly due to optimizations carried out to improve processor performance. These timing side channels have been used in various attacks including transient execution attacks on recent commodity processors. Hence, we need a tool to detect timing side channels during the design process. This paper presents SIGFuzz, a fuzzing-based framework for detecting microarchitectural timing side channels. A designer can use SIGFuzz to detect side channels early in the design flow and mitigate potential vulnerabilities associated with them. SIGFuzz generates a cycle-accurate microarchitectural trace for a program that executes on the target processor, it then uses two trace properties to identify side channels that would have been formed by the program. These two trace properties evaluate the effect of each instruction in the program, on the timing of its prior and later instructions, respectively. SIGFuzz also uses a statistical distribution of execution delays of instructions with the same mnemonic to flag potential side channels that manifest with different operands of an instruction. Furthermore, SIGFuzz automatically groups the detected side channels based on the microarchitectural activity trace (i.e. signature) of the instruction that triggered it. We evaluated SIGFuzz on two real-world open-source processor designs: Rocket and BOOM, and found three new side channels and two known side channels. We present a novel Spectre-style attack on BOOM based on one of the newly detected side channel.**

***Index Terms*—side channels, microarchitecture, hardware fuzzing**

I. Introduction

Microarchitectural side channels have been threatening the security of computing systems for decades [1]. The recent discovery of side-channel based transient-execution attacks such as Spectre [2] and Meltdown [3] has led to a flurry of new side-channel based attacks [4]. These microarchitectural side channels are often introduced due to power, performance and/or area optimizations. For example, Shin et al. [5] showed that data prefetching, a performance optimization technique used in modern processors, introduces side channels that attackers can exploit. Therefore, identifying side channels early in the design flow and mitigating any vulnerabilities associated with them is vital to ensure processor security.

Typically, side channels have been discovered through manual inspection and experiments. For example, Fogh et al. [6] discovered side channels related to `rdseed` and `pause` instructions in Intel CPUs through the processor documentation and experiments. However, manual inspection is not a practical method to identify side channels at design time as it can be quite cumbersome. Moreover, manually checking the interplay between all instruction combinations and their side effects is practically impossible due to the limited ‘time to market’ for any processor. Hence, there is a pressing need for a tool that can detect various types of side channels with reduced manual effort during the design stage of processors.

Several automation methods have been proposed in recent work for detecting side channels in processors. These include dynamic verification (fuzzing) methods [7] as well as formal verification methods [8]. Fuzzing is a popular software testing method [9], which involves running a target software with random or mutated inputs to find bugs. Fuzzing has been recently adapted for hardware testing [7], [10]–[13].

Osiris [7] introduced an automated method to find microarchitectural side channels using a generic three-step model. Covert Shotgun [14] and ABSynthe [15] automated the detection of contention-based side channels that occur when instructions are executed in hyper threads (two logical cores) simultaneously. UPEC [8] introduced an exhaustive formal verification method for detecting side channels in the context of transient execution attacks. AutoCAT [16] introduced a reinforcement learning based framework for automatically generating cache-timing attack sequences for any given cache.

Unfortunately, these prior work either can detect a specific type of side channels or are limited by their applicability. Osiris is limited to detecting side channels formed by three instructions that follow their model, which limits the types of side channels it can detect. Methods proposed by Covert Shotgun and ABSynthe are unique for detecting contention-based side channels and hence, they are not applicable for other classes of side channels (e.g., single core timing side channels). Adapting the UPEC approach involves considerable manual effort for building models and proving properties (e.g.-Microequivalence [8]) for each target processor, which hinders its applicability. AutoCAT specializes in detecting cache timing channels, which limits its capability to find side channels in other microarchitectural units.

In this paper, we propose SIGFuzz[1] , a **generic** fuzzing-based framework for discovering various types of microarchitectural timing side channels. SIGFuzz executes random or mutated assembly tests on the target processor and evaluates the effect of each instruction on the execution time of other instructions in the test to detect timing side channels. Our generated tests contain >100 randomly ordered instructions with random operands (as opposed to just three instructions with constant operands in Osiris). This enables SIGFuzz to detect side channels formed with more than three instructions with different operand values.

SIGFuzz generates cycle-accurate microarchitectural traces[2] of tests and checks if the traces satisfied/violated two trace properties to determine whether any instruction in the test triggered a side channel. These two trace properties state whether the instruction in question affected the commit time of prior

[1]Available at `https://github.com/bu-icsg/SIGFuzz`
[2]A trace that represents the microarchitectural state of the processor in each clock cycle.

and later instructions in the program, respectively. SIGFuzz also determines a statistical distribution of delays of instructions with the same mnemonic to flag potential timing side channels. More details about this process are provided in Section III. Furthermore, SIGFuzz features an automated method to group similar side channels, based on the mnemonic and the microarchitectural activity trace, i.e., the *signature* of the instruction that triggered it. At the end of a fuzzing session, SIGFuzz automatically generates a report of potential side channels that were detected.

We evaluated SIGFuzz by running it on two commonly used RISC-V open-source processors, namely Rocket [17] and BOOM [18]. SIGFuzz found both known and new side channels in both processors. Additionally, we present a novel Spectre-style attack on BOOM based on one of our newly detected side channels. In summary, we make the following contributions:

1) We present a **generic** approach for discovering microarchitectural timing side channels formed by a given program. The generic nature of our approach enables us to discover side channels in a broader scope compared to prior work. Based on this approach, we develop SIGFuzz, a fuzzing framework for discovering microarchitectural timing side channels.
2) We discovered three new side channels in two open-source RISC-V processors, Rocket and BOOM. Two of the newly detected side channels are common to both Rocket and BOOM, while the third one is unique to BOOM.
3) We present a novel Spectre-style attack based on a newly detected side channel.

II. Related Work

There have been several recent work that automate the detection of side channels. Osiris [7] introduces an automated method to discover microarchitectural side channels using a generic three-step model, which iteratively checks all three-instruction combinations in an ISA while keeping the operands constant. Covert Shotgun [14] and ABSynthe [15] automate the detection of contention-based side channels across two hyperthreads. Covert Shotgun runs a set of hand-picked instructions on two hyperthreads of a CPU simultaneously and measures any observable contention from one instruction on the other. ABSynthe improves on Covert Shotgun by automatically finding the sequence of instructions that maximize the information leakage for a given processor and using it to synthesize cross-thread attacks. AutoCAT [16] introduces a reinforcement learning based framework for automatically generating cache-timing attack sequences for any given cache. UPEC [8] proposes a formal verification-based method to exhaustively detect transient execution side channels in processors at design time.

The aforementioned approaches are, however, either limited to discovering a specific type of side channels or limited in their practical applicability. For example, Osiris is limited to detecting side channels that are formed by three instructions with constant operands. In contrast, SIGFuzz is capable of discovering side channels formed by any generic instruction sequence with any number of instructions and different operands. Similarly, AutoCAT is limited to detecting timing channels in the cache while SIGFuzz introduces a more generic method that can discover timing side channels formed in any microarchitectural unit of a processor. The methods proposed by Covert Shotgun and ABSynthe are not applicable beyond contention-based side channels between hyperthreads. In contrast, SIGFuzz detects timing-based side channels that manifest in a single hardware thread. Unlike SIGFuzz, UPEC guarantees exhaustiveness when detecting transient execution side channels. However, SIGFuzz can be applied to a target with less manual effort without developing models (microequivalence) or customization, which is required for UPEC.

Several other fuzzing methods have been introduced recently for finding bugs in processors [10]–[12]. These methods use an ISA simulator as a golden model, where any mismatch with the golden model is considered a potential bug. However, ISA simulators only model the architectural behavior, leaving the microarchitectural behaviors of the target processor unchecked. Therefore, these methods are not capable of detecting microarchitectural side channels and are not comparable with SIGFuzz.

III. SIGFuzz

Figure 1 shows the overview of the SIGFuzz fuzzing framework. At a high level, SIGFuzz uses a coverage-guided fuzzing engine to generate tests. It then mutates these tests and executes them on the target processor. Subsequently, SIGFuzz relies on two trace properties to identify any side channels that could form by the tests. SIGFuzz then extracts the signature of each committed instruction and stores it in a bin database. These signatures are then post-processed to detect potential timing side channels. In the rest of this section, we first discuss the motivation for the design of SIGFuzz, then discuss the different steps of SIGFuzz in detail.

A. Motivation

Typically, a group of instructions is required to form a side channel. If any instruction from the group affects the execution time of a prior or later instruction in the group, it indicates the existence of a potential side channel

Example 1: In the contention-based side channel used in the Spectre-STC [8] attack, arithmetic and logic instructions cause additional delays in prior division instructions due to the contention at the register file write port.

Example 2: In the Flush+Reload [19] cache side-channel, flush instruction affects the timing of a later memory access instruction that *reloads* the data.

Motivated by the above observations, we came up with two trace properties to detect the two types of side channels shown in examples 1 and 2. (Property 1 and 2 in III-B4). Furthermore, identifying whether a particular instruction had any effect on the timing of another instruction is challenging. To address this challenge, we propose a differential method based on the `nop` instruction. We claim that the effect of an instruction (say K) on the timing of other instructions in the test can be accurately identified by replacing the instruction K with a `nop` and comparing the commit times of each instruction when using K and when using `nop` in place of K.

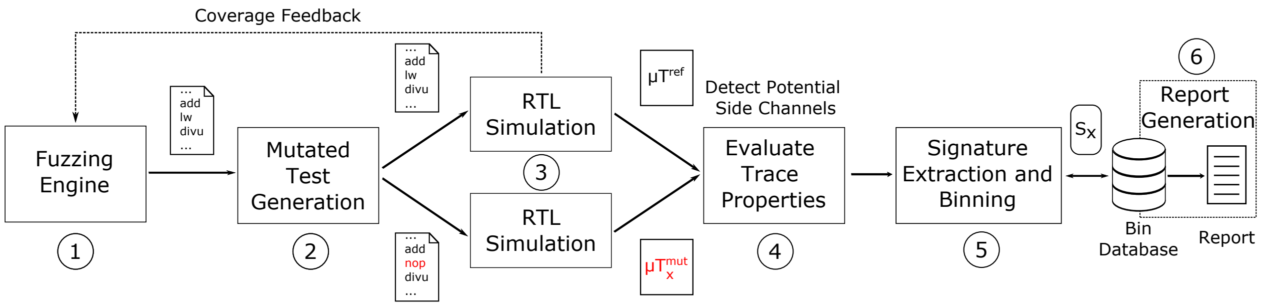


Fig. 1. SIGFuzz framework: SIGFuzz generates cycle-accurate microarchitectural traces (μT^{ref} and μT_x^{mut}) for the reference test and a mutated test. These traces are used to evaluate the trace properties, detect potential side channels and extract the signature (S_x) of the instruction in question (`lw` in the example shown in the figure). Signatures and information about the detected side channels are stored in the bin database, from which a report is generated at the end of the fuzzing session.

Additionally, an instruction by itself can manifest as a side channel depending on its operands.

Example 3: If a division instruction takes a longer time to execute when the divisor is greater than the dividend compared to its usual execution time, an attacker can exploit this timing difference to leak information about the data processed by the division operation (we demonstrate this in Section V).

With the goal of detecting this type of side channels, we define a metric called commit time difference (CTD) that represents the execution time of an instruction (see Section III-B4).

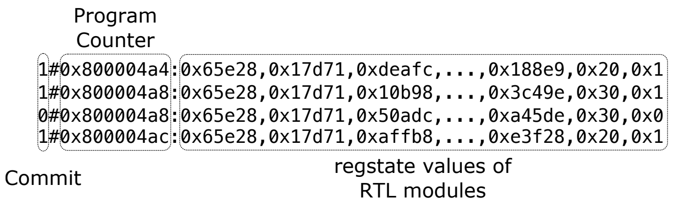
B. SIGFuzz Framework

1) Fuzzing Engine: The fuzzing engine generates random assembly tests and passes them to the next step. It receives coverage information for a test from step ③. If a test increased the coverage of the design, that test is marked as 'interesting' and kept for further mutations.

2) Mutated Test Generation: SIGFuzz takes a test provided by the fuzzing engine and mutates it to create more tests. The mutation involves replacing each instruction with a *nop* instruction, one at a time. We refer to the original test provided by the fuzzing engine as reference test T^{ref} and a test that SIGFuzz generates by replacing the x^{th} instruction as mutated test T_x^{mut}.

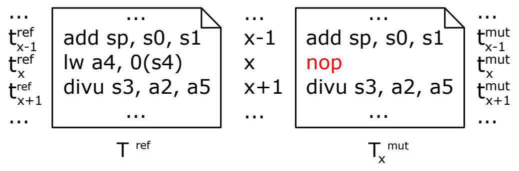
3) RTL Simulation and Microarchitectural Trace Generation : SIGFuzz then executes the reference test and the mutated test in the target processor through two separate RTL simulations. These RTL simulations generate microarchitectural traces of the tests (μT^{ref} and μT_x^{mut}). SIGFuzz generates these traces from the RTL simulation of the target processor for two purposes. First, to check violation of trace properties and detect potential side channels formed by the tests, which require a cycle-accurate commit trace. Second, to extract the microarchitectural activity trace i.e. signature of an instruction, which requires a mechanism to represent the microarchitectural state of the processor at each clock cycle.

To satisfy the first need, we instrumented the RTL design of the target processor to print commit and program counter RTL signals in each clock cycle and included it in the microarchitectural trace. The commit signal is set to 1 whenever an instruction is committed. To satisfy the second need, SIGFuzz uses `regstate` representation of an RTL module used in the register-coverage metric introduced by DifuzzRTL [11]. DifuzzRTL identifies all control signals that drive the muxes in a module in an RTL design of a processor and hashes them together to


Program Counter
Commit
regstate values of RTL modules

```
1#0x800004a4:0x65e28,0x17d71,0xdeafc,...,0x188e9,0x20,0x1
1#0x800004a8:0x65e28,0x17d71,0x10b98,...,0x3c49e,0x30,0x1
0#0x800004a8:0x65e28,0x17d71,0x50adc,...,0xa45de,0x30,0x0
1#0x800004ac:0x65e28,0x17d71,0xaffb8,...,0xe3f28,0x20,0x1
```

Fig. 2. Part of a microarchitectural trace generated by SIGFuzz. Each line in the microarchitectural trace represents a clock cycle during the test execution in the processor. The leftmost value in the trace is the commit signal, which indicates whether the program counter in the same row is committed in that clock cycle. The program counter located in the second column does not carry any meaning if the commit signal is zero. Each comma-separated value after the ":" sign represents the `regstate` value of an RTL module in the target processor.



```
...
add sp, s0, s1
lw a4, 0(s4)
divu s3, a2, a5
...
```

```
...
add sp, s0, s1
nop
divu s3, a2, a5
...
```

Fig. 3. A minimal example of a reference test (T^{ref}) and a mutated test generated from that (T^{mut}). $x-1$, x, $x+1$ represent the instruction number in both tests. t_{x-1}^{ref}, t_x^{ref}, t_{x+1}^{ref} and t_{x-1}^{mut}, t_x^{mut}, t_{x+1}^{mut} represent the time in clock cycles when each instruction was committed in T^{ref} and T^{mut}, respectively.

create `regstate` for that module. Therefore, `regstate` represents the microarchitectural state of an RTL module in a given clock cycle. To enable `regstate` for all modules, we instrumented the RTL design of the processor with register-coverage and used `regstate` values of each RTL module to generate a microarchitectural activity in each clock cycle. Figure 2 shows an example microarchitectural trace generated by SIGFuzz.

4) Evaluate Trace Properties: Using the differential method mentioned in III-A, we define two trace properties to evaluate the effect of one instruction on others. We use the example tests shown in Figure 3 to describe these.

Property 1. This property checks whether an instruction x affects the commit time of an earlier instruction in the test. It can be formally expressed as follows:

$$\forall n < x \; : \; t_n^{ref} = t_n^{mut}$$

where t_n^{ref} and t_n^{mut} correspond to the commit time of the n^{th} instruction in the reference test and the mutated test.

Property 2. This property checks whether an instruction x affects the commit time of later instructions in the test. It can be formally expressed as follows:

$$\forall n > x \; : \; t_n^{ref} - t_x^{ref} = t_n^{mut} - t_x^{mut}$$

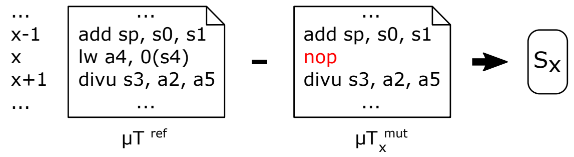


Fig. 4. Signature extraction process for instruction x (`lw a4,0(s4)` in the figure). The signature is extracted from the bitwise subtraction operation between the microarchitectural trace of the reference test (μT_{ref}) and that of the mutated test (μT_x). $x-1$, x, $x+1$ represent the instruction numbers in both tests.

With the goal of detecting side channels that manifest in a single instruction (Example 3 in III-A), we define a new metric CTD, which represents the difference in the commit time of x^{th} instruction in T_{ref} and the *nop* instruction (that replaced the x^{th} instruction) in T_{mut}, i.e. CTD = $t_x^{ref} - t_x^{mut}$.

SIGFuzz checks if any of the instructions in the reference and mutated tests have violated property 1 or property 2 using the microarchitectural traces generated in step ③. A violation of either property indicates a potential side channel. This information is relayed to ⑤. SIGFuzz also calculates CTD using microarchitectural traces and passes it on to step ⑤, where it gets stored in the bin database along with the signature of the x^{th} instruction.

5) Signature Extraction and Binning: Due to the random nature of tests generated by a fuzzing engine, the same side channel can be formed multiple times during a fuzzing session. To avoid analyzing the same side channel repetitively, SIGFuzz introduces a grouping method based on the microarchitectural activity trace, i.e., the signature of the *trigger* instruction. This is based on the idea that if two instructions triggered two side channels and the signatures of the two instructions are the same, the side channel they triggered should be the same.

Extracting the signature. The signature of an instruction is extracted by applying the same differential method mentioned in section III-A on microarchitectural traces. To explain the process of signature extraction, let's call the signature of the x^{th} instruction as S_x. S_x is extracted by performing a bitwise subtraction operation between μT^{ref} and μT_x^{mut} from the beginning of two logs till the commit line of x in μT_{ref}. This process is illustrated in Figure 4. In a scenario where x takes more clock cycles to commit than the `nop` instruction (CTD $>$ 0), additional lines in μT_x are included directly in the signature without the subtraction operation.

Bin Database (BDB). SIGFuzz maintains a bin database for storing the signatures captured during a fuzzing session. In BDB, signatures are first sorted based on their instruction mnemonic, and for each mnemonic we have one or more *bins*, where each *bin* corresponds to a group of similar signatures. After extracting a signature, SIGFuzz checks the bins under the instruction mnemonic for a similar signature. If a similar signature is found in a bin, the extracted signature is added to the same bin. The process for checking similarity is described later in this section. If the extracted signature is different from the existing signature(s) for that instruction mnemonic, a new bin is created with the new signature. In summary, the bin database contains bins for each mnemonic, each bin representing a unique microarchitectural behavior of an instruction.

Comparing signatures. All signatures added to a bin are averaged to create a 'composite signature' that represents all signatures in the bin. Averaging is also done individually for each bit in the signature. To check the similarity of a new signature with the 'composite signature' of a bin, SIGFuzz first checks whether the new signature and the 'composite signature' have the same length. If the length of the two signatures is the same, we create a binary matrix for each signature where each column represents how a mux control signal in the processor changed during the lifetime of an instruction. Then, SIGFuzz calculates the Normalized Hamming Distance (NHD) between corresponding columns of two matrices and generates a list of NHD values, one for each column. The maximum of these NHD values is used as an inversely related metric for the similarity of two signatures. For example, if the maximum NHD between the columns of two signatures is 0.2 (out of 1), that relates to a 80% similarity between the two signatures. SIGFuzz then uses a predefined threshold of 0.3 with the maximum NHD value to classify whether two signatures are similar or not. We decided the threshold value manually by checking the NHD values between the signatures of instructions that have the same microarchitectural behaviors.

If step ④ indicated a property violation by an instruction, the bin containing the signature of that instruction is marked in BDB with the property it violated. After SIGFuzz executes all mutated tests in step ①, it takes another test from the fuzzing engine and continues the same process.

6) Report Generation: Once the fuzzing session ends after a user-specified time, SIGFuzz automatically generates a detailed report from the bin database, with bins that relate to side-channel behaviors flagged. During this process, SIGFuzz creates a histogram of CTD for each instruction mnemonic. That is used to generate a Gaussian Kernel Density Estimate (KDE) function, in which maxima and minima are used to identify the number of clusters in the distribution. The number of clusters for each mnemonic is included in the report and mnemonics with more than one cluster are marked as having potential side-channel behaviors. We show an example on how clusters in CTD histogram relate to side channels in Section IV.

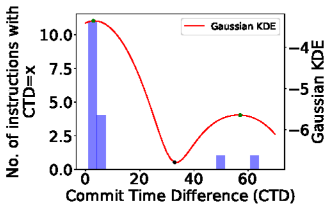
IV. Side Channel Detection

In this section, we first describe our experimental setup and then present an example of how finding clusters in the histogram of CTDs led us to discover a new side channel. Next, we present the new and known side channels discovered by SIGFuzz.

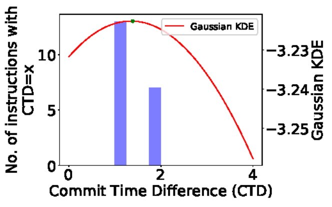
A. Experimental Setup

We implemented and evaluated SIGFuzz on two commonly used open-source RISC-V processors, Rocket [17] and BOOM [18]. We used DifuzzRTL [11] as the fuzzing engine and register coverage [11] as feedback in the implementation of SIGFuzz. We used the Verilator RTL simulator to run RTL simulations for both processors. All results were obtained by running SIGFuzz on server nodes with Intel Xeon E5-2670 CPUs and CentOS Linux 7 as the operating system.

We ran five 4-hour fuzzing sessions with SIGFuzz for each processor and collected the automatically generated reports. We went through the instructions and bins that were flagged as potential side channels in these reports and found three new side channels and two known side channels [8], [20]. Two of these



(a) `div`



(b) `fnmadd.s`

Fig. 5. CTD histograms and Gaussian Kernel Density Estimate (KDE) for (a) `div` and (b) `fnmadd.s` instructions on BOOM processor.

side channels were common for both Rocket and BOOM while the third one was unique to BOOM. Rocket and BOOM having the same side channels is not surprising as they share the implementations of several functional units. Both known and new side channels that SIGFuzz detected are listed in Table I. In a 4-hour fuzzing session, on average, the number of reference and mutated tests executed for the Rocket core were 337 and 3284, respectively, while they were 209 and 1851 for the BOOM core.

B. Discovering Side Channels Through CTD Histogram

Here, we use CTD histogram and Gaussian Kernel Density Estimate (KDE) of two example instructions to explain how we can use clusters to identify side channels. Figure 5 shows the historgrams for `div` and `fnmadd.s` instructions. As shown in Figure 5a, the `div` instruction has two CTD clusters, and the KDE has two maxima and one minima due to the two CTD clusters. In contrast, KDE of `fnmadd.s` instruction (Figure 5b) has only one maxima due to a single cluster. We manually investigated the root cause for `div` instructions having two CTD clusters. We observed that in the left cluster of the `div` instruction dividend is greater than the divisor, while the dividend is smaller than the divisor in the right cluster. We confirmed that this behavior is common to both Rocket and BOOM. This led us to conclude the existence of a timing side channel in the division unit of both Rocket and BOOM processors (Table I, No. 4). For `fnmadd.s` mnemonic, we only observed one cluster. This led us to conclude that `fnmadd.s` does not have a side channel behavior.

C. New Side Channels Discovered by SIGFuzz

1. Store Conditional (SC) Side Channel. This side channel was triggered by store conditional instructions and was identified due to a violation of property 2. SC instructions in RISC-V ISA store a value to a given address if a reservation made by a prior load-reserve instruction to the same address still exists. We observed that, regardless of a reservation, a SC instruction causes an additional delay in a later load instruction if the load instruction is accessing an address that maps to the same cache line accessed by the SC instruction. A further investigation revealed that SC instructions bring data to cache and mark them as 'dirty', regardless of whether an actual store was performed or not. The processor thus takes additional time to first write the 'dirty' data accessed by the SC instruction, delaying the execution of the load instruction. Both BOOM and Rocket have write-back cache implementations, which keep the 'dirty' data in the cache until it is evicted.

2. Division Unit Side Channel. This side channel was discovered through the clusters that SIGFuzz identified in CTD histogram of division instructions as explained in IV-B. Division instructions take additional 78 cycles in BOOM to finish when the divisor $>$ dividend compared to when the divisor $<$ dividend. We demonstrate the exploitability of this side channel in Section V by presenting a novel Spectre-style attack.

3. Load Side Channel. We discovered this side channel through the violation of property 1 when fuzzing the BOOM processor using SIGFuzz. We observed that a `ld` instruction gets delayed by 8 cycles when there is a `ld` instruction later in the program, even when there are no data dependencies between the two. BOOM developers confirmed this as an unexpected behavior.

D. Known Side Channels Discovered by SIGFuzz

4. ORC Attack Side Channel [20]. SIGFuzz identified this side channel due to a violation of property 2 by store instructions. A load instruction is delayed if it was mapped to the same cache line as a pending store operation that was executed earlier.

5. Spectre-STC Side Channel [8]. SIGFuzz identified this side channel due to a violation of property 1 by floating-point instructions in BOOM. Spectre-STC uses a contention-based side channel, which delays the execution of division instructions by ALU and multiplication instructions that are after the division instruction in the program code. The root cause of this behavior is BOOM giving priority to ALU and multiplication instructions over division instruction when contention is created at the register file write port. SIGFuzz identified a scenario where an additional delay was introduced to an earlier division instruction when a floating-point instruction was replaced by a `nop`. In BOOM, `nop` instruction is decoded as a `addi` instruction (which is an ALU instruction) with all operand values set to zero. Therefore, we concluded that this is the same side channel used in Spectre-STC.

V. Side Channel Exploitation

In this section, we demonstrate a novel Spectre-style attack based on the division unit side channel detected by SIGFuzz. The Spectre attack exploits the transiently executed instructions after a control or data flow misprediction. During the transient window created by a misprediction, Spectre leaks otherwise inaccessible data by encoding it in the cache. Later an attacker can use a cache covert channel to access this data.

Figure 6 shows the minimum proof-of-concept code that can be used to illustrate the Spectre-style attack we propose. The first part of the code (line 1 to 11) uses a set of floating-point division operations to delay resolving the branch at line 13. This delay will make the processor execute instructions inside the if condition speculatively, given that the branch predictor predicts the branch at line 13 as taken. In this attack, we assume that the attacker can control the values of `idx` and `probe_val` and knows the address of the secret he wants to access. To leak information about a secret stored in the memory, attacker can set `idx` as `idx` = (address of the secret - address of `array1`). This will result in reading the secret value during the transient window created by the branch on line 13 and using it as

TABLE I
SIDE CHANNELS DISCOVERED BY SIGFUZZ IN ROCKET AND BOOM PROCESSORS.

No	Processor	Description of the side channel	New/Known?	Flagged property or CTD
1	Rocket, BOOM	Store conditional (sc) instructions bring data to cache and mark them as dirty regardless of the store conditional success. This delays the later memory accesses that map to the same cache line because data needs to be written back.	New	Property 2
2	Rocket, BOOM	In a division instruction, if the divisor is larger than the dividend or equal to zero, the instruction takes longer time to execute compared to other scenarios.	New	CTD
3	BOOM	If a load instruction is followed by another load instruction, the earlier load gets delayed.	New	Property 1
4	Rocket	Side channel used in ORC attack [20]. If there is a pending store to a particular cache line, loads that map to the same cache line are delayed till the store is done.	Known	Property 2
5	BOOM	Side channel used in Spectre-STC [8] attack. Division instructions get delayed due to the contention created at the registerfile write port by later ALU and multiplication instructions.	Known	Property 1

```
 1 array1_sz =  array1_sz << 3;
 2 asm("fcvt.s.lu      fa4, %[in]\n"
 3     "fcvt.s.lu      fa5, %[inout]\n"
 4     "fdiv.s fa5, fa5, fa4\n"
 5     "fdiv.s fa5, fa5, fa4\n"
 6     "fdiv.s fa5, fa5, fa4\n"
 7     "fcvt.lu.s      %[out], fa5, rtz\n"
 8     : [out] "=r" (array1_sz)
 9     : [inout] "r" (array1_sz),
10       [in] "r" (dummy)
11     : "fa4", "fa5");
12
13 if (idx < array1_sz){
14     asm("div        x1, %[dvd], %[dvs]\n"
15         "slli       x1, x1, 6\n"
16         "add        x1, x1, %[daddr]\n"
17         "lw         %[out], (x1)\n"
18         : [out] "=r" (dummy)
19         : [dvd] "r" (array1[idx]),
20           [dvs] "r" (probe_val),
21           [daddr] "r" (&dummyaddr)
22         : "x1" );
23 }
```

Fig. 6. Minimal RISC-V code for Spectre-style attack based on the side channel in the division instruction.

the dividend of the division instruction on line 14. Division instruction at line 14 uses the `probe_val` as the divisor.

If the secret value is greater than the `probe_val`, division instruction takes less time to execute, allowing the load instruction at line 17 to execute during the transient window, which accesses a dummy address accessible to the attacker. If the secret value is less than the `probe_val`, division instruction takes more time to execute, resulting in the load instruction not executing within the transient window. If the load instruction is executed, it brings the data at `dummyaddr` to the cache. Also, note that a dependence between the division result and the load instruction is added so that the load instruction is not speculatively executed while the processor calculates the division result. Therefore, after the code sequence given in Figure 6 is executed, an attacker can check whether the data at `dummyaddr` is cached or not and deduce whether the secret value was greater than the `probe_val` or not. The attacker can use different values for `probe_val` iteratively and figure out the secret value (through binary search).

VI. CONCLUSION

We propose SIGFuzz, a fuzzing-based framework for detecting microarchitectural timing side channels at design time. SIGFuzz determines variations in instruction execution times to identify timing-based side channels. We implemented and evaluated SIGFuzz on two open-source RISC-V processors, Rocket and BOOM. SIGFuzz discovered three new and two known side channels in these processors, demonstrating its effectiveness and practicality. We presented a novel Spectre-style attack based on a new side channel that SIGFuzz detected, showing the importance of identifying side channels early in the design process to ensure processor security.

ACKNOWLEDGMENT

Parts of this work are funded by Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under agreement number FA8650-18-2-7856, and by NSF Award 1916393. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

REFERENCES

[1] Q. Ge *et al.*, "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware," *Journal of Cryptographic Engineering*, vol. 8, no. 1, pp. 1–27, 2018.
[2] P. Kocher *et al.*, "Spectre attacks: Exploiting speculative execution," in *IEEE SP*, 2019, pp. 1–19.
[3] M. Lipp *et al.*, "Meltdown: Reading kernel memory from user space," in *USENIX Security*, 2018, pp. 973–990.
[4] X. Lou *et al.*, "A survey of microarchitectural side-channel vulnerabilities, attacks, and defenses in cryptography," *ACM Comput. Surv.*, vol. 54, no. 6, 2021.
[5] Y. Shin *et al.*, "Unveiling hardware-based data prefetcher, a hidden source of information leakage," in *CCS*, 2018, p. 131145.
[6] A. Fogh, "Two covert channels," https://cyber.wtf/2016/08/, 2016.
[7] D. Weber *et al.*, "Osiris: Automated discovery of microarchitectural side channels," in *USENIX Security*, 2021, pp. 1415–1432.
[8] M. R. Fadiheh *et al.*, "An exhaustive approach to detecting transient execution side channels in rtl designs of processors," *IEEE Transactions on Computers*, pp. 1–1, 2022.
[9] Google, "Oss-fuzz: Continuous fuzzing for open source software," https://github.com/google/oss-fuzz, 2016.
[10] S. Canakci *et al.*, "Processorfuzz: Guiding processor fuzzing using control and status registers," 2022. [Online]. Available: https://arxiv.org/abs/2209.01789
[11] J. Hur *et al.*, "Difuzzrtl: Differential fuzz testing to find cpu bugs," in *IEEE SP*, 2021, pp. 1286–1303.
[12] R. Kande *et al.*, "TheHuzz: Instruction fuzzing of processors using Golden-Reference models for finding Software-Exploitable vulnerabilities," in *USENIX Security*, 2022, pp. 3219–3236.
[13] S. Canakci *et al.*, "Directfuzz: Automated test generation for rtl designs using directed graybox fuzzing," in *DAC*, 2021, pp. 529–534.
[14] A. Fogh, "Covert shotgun: automatically finding smt covert channels," https://cyber.wtf/2016/09/27/covert-shotgun/, 2016.
[15] B. Gras *et al.*, "ABSynthe: Automatic Blackbox Side-channel Synthesis on Commodity Microarchitectures," in *NDSS*, 2020.
[16] M. Luo *et al.*, "Autocat: Reinforcement learning for automated exploration of cache timing-channel attacks," 2022.
[17] "The rocket chip generator," https://github.com/chipsalliance/rocket-chip.
[18] "Boom: Berkeley out-of-order machine," https://github.com/riscv-boom/riscv-boom.
[19] Y. Yarom *et al.*, "FLUSH+RELOAD: A high resolution, low noise, L3 cache Side-Channel attack," in *USENIX Security*, 2014, pp. 719–732.
[20] M. R. Fadiheh *et al.*, "Processor hardware security vulnerabilities and their detection by unique program execution checking," in *DATE*, 2019, pp. 994–999.