SurgeFuzz: Surge-Aware Directed Fuzzing for CPU Designs

Yuichi Sugiyama, Reoma Matsuo, Ryota Shioya
Graduate School of Information Science and Technology, The University of Tokyo, Tokyo, Japan
Email: {sugiyama, matsuo}@rsg.ci.i.u-tokyo.ac.jp, shioya@ci.i.u-tokyo.ac.jp

***Abstract*—Various verification methods have been proposed for bug detection in central processing unit (CPU) designs, yet their effectiveness remains insufficient. We have observed that such CPU bugs often occur in exceptional handling, such as pipeline stalls and flushes. We found that corner cases in such exceptional handling can be effectively verified through situations we term a 'surge'. A surge refers to a situation where events leading to exceptional handling occur frequently over a short period of time. For instance, a surge caused by frequent queue insertions can eventually fill the capacity, triggering exceptional handling such as a pipeline stall. We propose a novel fuzzing method for CPU designs, named SurgeFuzz, that intentionally generates surges. SurgeFuzz mutates input instruction sequences based on annotations to increase the occurrence of surges. This results in a higher density of event occurrences, thereby enabling efficient verification of corner cases in exceptional handling. We evaluated SurgeFuzz on a large processor design and found several unknown hardware bugs that are difficult to find with existing methods.**

***Index Terms*—Fuzzing, Directed fuzzing, CPU verification, RISC-V**

I. Introduction

Bugs in CPU designs can lead to a multitude of serious issues, ranging from erroneous execution results and system freezes to vulnerabilities enabling unauthorized data access [1]. Indeed, numerous bugs have been detected in commercial x86 and ARM CPUs, which include deadlocks from improper hazard handling and corruption of execution results due to flawed memory speculation [2]–[5] These bugs frequently emerge in complex microarchitectural corner cases, rendering them generally hard to detect. Furthermore, unlike software bugs, these are often challenging to rectify post-production.

Although various verification methods for detecting CPU bugs have been proposed [6]–[15], their efficacy in bug detection remains insufficient. This is mainly because CPU designs are generally large and complex, and thus their state space that needs to be explored is very large. Static verification methods employing model checking or symbolic execution [6]–[8] can often result in state space explosion, making them challenging to apply to large CPU designs [1], [16]. Dynamic verification methods, which typically utilize randomly generated instruction sequences, are comparatively easy to apply [9]–[15]. More advanced dynamic verification methods based on *fuzzing* have been studied in recent years [17]–[25]. Fuzzing generally iterates test input generation, utilizing feedback such as coverage during testing, thereby exploring the hardware state space more effectively than traditional dynamic verification methods. However, the state space in CPUs explored by many fuzzing methods is still very large, and these methods have not been able to effectively detect bugs.

Bugs in CPUs, as mentioned above, are known to occur in corner cases during *exceptional handling*, such as stalls or flushes when queues fill up or request conflicts occur [21], [26]–[29]. This is because the logic of such exceptional handling is typically complex and their corner cases are often overlooked, and because such corner cases rarely occur in practice. For example, it is reported that almost all design bugs in the OpenSPARC processor were found in the load/store unit and the trap logic unit, which deal with many such corner cases [26]. Because these corner cases arise from interactions between various CPU modules and pipeline control, it is difficult to find them through simple unit tests.

Developers are often aware that bugs are likely to occur in corner cases during exceptional handling, and it is common for developers to manually create test inputs to verify specific corner cases [27]. However, generating instruction sequences that cause such corner cases is generally challenging and requires significant effort. For example, one of the bugs discovered in this paper is triggered by repeating many replays due to a memory speculation failure while recovering from a branch misprediction. Creating an instruction sequence that artificially causes such a situation requires deep knowledge of microarchitecture and considerable skill. The instruction sequence that causes such corner cases is strongly dependent on the CPU configuration and behavior, and therefore, small changes in the configuration parameters often prevent the intended corner case from occurring. Dynamic verification methods and fuzzing can theoretically generate inputs to test such corner cases. However, since these methods generally generate test inputs by exhaustively exploring the entire state space of the CPU, it is difficult to efficiently generate instruction sequences that induce specific corner cases, and even if they can be generated, it takes a long time.

We found that corner cases in such exceptional handling can be effectively verified through situations that we term a *surge*. A surge refers to a situation when events leading to exceptional handling occur more frequently within a short period than usual. These events include, for example, the insertion of elements into queues or buffers, access conflicts with banks/buses/shared resources, and the occurrence of speculation misses. Each of these events can normally occur and does not particularly cause problems when it occurs individually. However, if it occurs much more frequently than normally expected, it can trigger various exceptional handling. For instance, frequent insertions into a certain request queue can create a state where the queue is filled, which triggers exceptional handling such as pipeline stalls. Developers often overlook these corner cases in exceptional handling, which can often lead to bugs.

From this observation, we propose SurgeFuzz, a fuzzing method that intentionally triggers surges to efficiently verify corner cases in exceptional handling. SurgeFuzz is a type of directed fuzzing, which is based on user instructions [19], [30]–[33]. In SurgeFuzz, the user annotates locations in the RTL code that indicate the occurrence of an event that triggers the targeted exceptional handling. SurgeFuzz observes the state of the annotated locations and mutates input instruction sequences to induce a surge, that is, to increase the temporal density of event occurrences as much as possible. For example, when considering a surge caused by a bank conflict, the user annotates a signal activated by the conflict, and SurgeFuzz mutates inputs so that the temporal density at which the signal is active increases. As a result, SurgeFuzz can efficiently generate instruction sequences that induce surges, effectively verifying corner cases.

In summary, the main contributions of this study are as follows:

- We found that various corner cases can be verified by a surge, which is a situation where events leading to exceptional handling occur frequently.

This work was partially supported by JST PRESTO JPMJPR21P5 and JSPS KAKENHI JP20H04153.

- We propose a directed fuzzing method called SurgeFuzz, which generates surge-inducing programs based on user annotation and efficiently verifies corner cases. SurgeFuzz creates a small state space containing only the registers associated with the annotated signal and then explores this state space in a directed manner to increase the temporal density of event occurrences.
- SurgeFuzz can efficiently and automatically generate test inputs that trigger specific corner cases. This capability is useful not only for finding logical bugs, but also for finding performance bugs and conducting stress tests.
- We evaluated SurgeFuzz on three relatively large-scale RISC-V out-of-order superscalar processor designs. Across 12 different cases, the evaluation results showed that SurgeFuzz induces surges significantly faster than existing methods. As a result, we discovered five previously unknown bugs in these CPU designs.
- We have made our implementation and evaluation environment available at https://github.com/shioya-lab-public/surgefuzz.

II. Background and Motivation

This section provides the background of SurgeFuzz, focusing on fuzzing and the motivation for this study. Other dynamic verification techniques will be discussed in Section VI later.

A. Feedback-driven Fuzzing

Fuzz testing, also known as fuzzing, is generally a software testing technique, and a program that performs fuzzing is called a fuzzer. In general, fuzzing provides various inputs to a target program for finding bugs and vulnerabilities [34]–[36].

In particular, a method known as *feedback-driven fuzzing* repeatedly generates inputs and executes a target program, using feedback based on information gathered during execution. The information provided as feedback is typically based on a *coverage metric*. Coverage is a measure of the degree to which a target program is tested (explored) within its state space. For example, the coverage of executed lines or branch directions in code is generally used.

We explain the general behavior of the feedback-driven fuzzing using Figure 1. (1) The fuzzer selects a *seed* for test inputs from a seed pool. (2) The fuzzer mutates the selected seed to generate a certain number of test inputs. (3) The fuzzer executes a target program using the generated inputs. (4) Based on the test results, the fuzzer decides whether to save the tested input as a new seed in the seed pool. For example, if the coverage of executed lines is used as the metric, an input that increases the coverage will be added to the seed pool as a new seed.

B. Applying Fuzzing to CPU Designs

Similar to fuzzing in software testing, fuzzing in RTL designs, including CPU designs, generates test inputs efficiently and automatically. While in RTL designs the test input is typically a bit vector for hardware input signals [17], [19], in CPU designs the test input is typically a sequence of instructions, which are generated by random mutation/insertion of instructions [18]. Since many aspects other than test input generation are essentially common between CPU designs and RTL designs, the following describes fuzzing in RTL designs.

The major difference between fuzzing in RTL designs and fuzzing in software testing is the coverage metric used for feedback. Coverage metrics in software (e.g., lines, branches, or statements) are not sufficient for RTL designs because such metrics cannot efficiently capture the state of hardware such as finite state machines (FSMs) [18]. Furthermore, while fuzzing in RTL designs can be implemented by measuring the coverage in hardware to achieve fast fuzzing [17], it is difficult to implement coverage metrics in lightweight hardware when those metrics are based on information such as lines and statements.

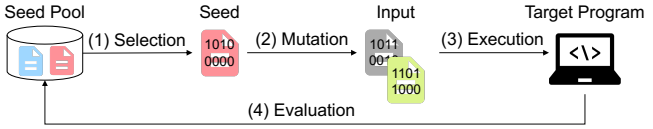

Fig. 1: Overview of feedback-driven fuzzing.

For these reasons, various coverage metrics have been proposed for RTL designs [17], [18]. One of the most naive coverage metrics for capturing hardware state is the set of all register states, but this approach can easily lead to state explosion. Thus, existing methods propose several coverage metrics that can capture hardware states appropriately while reducing the number of states to be explored. For example, RFUZZ [17] uses *the selection signal of a multiplexer*, which determines which input to output, as a coverage metric. DifuzzRTL [18] uses *control registers*, which are registers directly or indirectly connected to the selection signals of multiplexers. The number of signals used for these coverage metrics is much smaller than the number of flip-flops, and thus these existing methods can explore the state space more efficiently than the naive method.

As an alternative approach, DirectFuzz [19] focuses its exploration on specific parts of the state space. DirectFuzz is based on RFUZZ and aims to maximize the coverage of a specific module instance. It analyzes the hierarchical relationship of modules in hardware description language (HDL) code to create a directed graph that represents the relationships between modules. DirectFuzz intensively verifies a target module by preferentially mutating inputs that increase the coverage of modules close to the target module on the graph.

C. Bugs in Exceptional Handling

Existing fuzzing methods fail to sufficiently verify corner cases that arise in various *exceptional handling* scenarios performed by modern CPUs. Modern CPUs have complex structures that include various resources such as buffers, queues, and tables, along with state machines to manage them [37], [38]. Using such units, CPUs execute instructions while performing various buffering, scheduling, speculation, and recovery operations. These units can cause exceptional events, including resource allocation failures due to capacity limits, resource access conflicts, cache misses, and speculation misses. These events trigger exceptional handling, which differs from the normal instruction pipeline processing, such as pipeline flushes and stalls, instruction replays, and queue insertions for waiting.

It is well known that bugs in CPUs typically occur in corner cases of such exceptional handling [26], [28], [29]. This is because the logic of exceptional handling is typically so complex that its corner cases are easily overlooked and because such corner cases rarely occur in usual program execution. Such corner cases arise from the interaction of various modules and pipeline controls in CPUs, making it difficult to detect them through simple unit tests [27]. As a result, bugs such as deadlocks, livelocks, and corruption of execution results frequently occur even in modern commercial CPUs [2]–[5].

D. Challenges in Exceptional Handling Verification

We evaluated the ability of several existing methods to verify exceptional handling. We used a relatively large RISC-V out-of-order superscalar processor, RSD [37], to evaluate the maximum usage of the load queue in the processor. When the load queue reaches its capacity limit, additional requests lead to exceptional handling, such as stalling the pipeline. Such a load queue is known to be a bug-prone area in CPUs [26]. Figure 2 shows the distribution based on

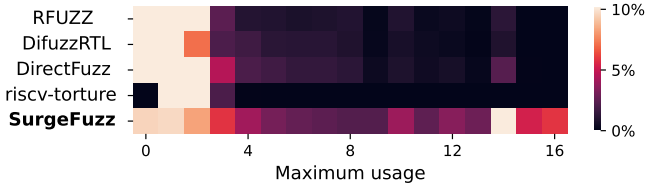

Fig. 2: Distribution of the maximum usage of load queue in RSD [37]. Values greater than 10% are shown in the same color. The existing methods generate few test inputs with high queue usage.

the maximum usage of the load queue with 16 entries. The evaluation environment is based on the ones used in Section V. As shown in the figure, the existing methods often verify the load queue under normal conditions with low usage and they are unable to verify the exceptional handling that occurs when its capacity is fully utilized.

This result is due to the fact that these methods uniformly explore the entire state space of the processor design. For example, RFUZZ/DifuzzRTL uniformly explores the state space consisting of all selection signals/control registers in the design. Since RSD has about $5,000$ bits of control registers, the size of its naive state space is about 2^{5000}. Applying uniform exploration over such a large space is unlikely to reach the corner case where the queue reaches its maximum usage. DirectFuzz can intensively explore a specific part of the state space, but as the evaluation results show, it is not able to efficiently explore the corner case in the load queue. This indicates that a corner case in a queue-like structure cannot be reached by simply specifying a module and intensively exploring it. As a more traditional method of verification, riscv-torture [14] uses randomly generated inputs under certain constraints, but such a method also fails to increase the load queue usage as well as fuzzing.

While it is possible to induce a situation where the capacity of the load queue is fully utilized by intentionally reducing the capacity, such a scenario does not lead to effective verification. The reason for this is that if the configuration of some modules is significantly changed, the behavior of other parts of the CPU may change, and the corner cases to be verified may no longer occur. For example, if the size of a specific queue is drastically reduced, it can become a bottleneck, leading to a significant slowdown of the entire CPU. As a result, bugs such as those that arise when frequent replays occur when the entire queue capacity is used cannot be effectively verified.

Developers often recognize that bugs tend to occur in corner cases during exceptional handling, and it is common for them to manually create test inputs to verify these specific situations [21], [27]. However, generating instruction sequences that cause such corner cases is generally challenging and requires deep knowledge of microarchitecture and considerable skill.

III. Surge-Aware Fuzzing for CPU Designs

A. Key Idea

We have discovered that many exceptional handling cases in CPUs can be induced by situations we refer to as a surge. A surge refers to a situation where events leading to exceptional handling occur more frequently in a short time than usually expected. For instance, frequent insertions into a request queue can create a state where the queue becomes full, triggering exceptional handling such as pipeline stalls. We consider events such as element insertions into queues or buffers, access conflicts with banks/buses/shared resources, and the occurrence of speculative misses. By intentionally causing these events frequently, we can trigger various types of exceptional handling.

Listing 1: Example of an annotation.

```
1  reg [2:0] in;
2  reg t, c1, c2;
3  reg [3:0] cnt;
4  (* SURGE_FREQ=1 *) wire increment;
5
6  always (@posedge clock) begin
7    t <= in[0] & in[1];
8    if (in[0] == 1) c1 <= in[2];
9    if (in[1] == 1) c2 <= in[2];
10   if (increment == 1) cnt <= cnt + 1;
11 end
12 assign increment = t & in[2];
```

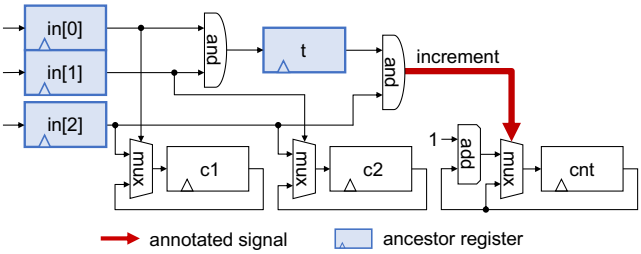

Fig. 3: Circuit synthesized from the code in Listing 1. The red signal represents an annotated signal, and the blue registers are the ancestor registers of that annotated signal.

Based on this insight, we propose SurgeFuzz, a directed fuzzing method that intentionally triggers surges to efficiently verify corner cases in exceptional handling. In SurgeFuzz, users annotate signals in RTL code that indicate the occurrence of events leading to targeted exceptional handling. 1) To limit the space to be explored, SurgeFuzz sets up a small partial state space consisting only of registers related to the annotated signals. 2) Then, when exploring this small state space, SurgeFuzz prioritizes the exploration of areas with a higher temporal density of event occurrences. Consequently, this allows for the automatic generation of test inputs that trigger exceptional handling more efficiently than manual or dynamic verification methods.

B. System Overview

1) The small state space to be explored consists of registers, called *ancestor registers*, selected based on annotations. Users annotate a signal in RTL code that indicates the occurrence of an event, and we refer to the signal as an *annotated signal*. An ancestor register is a register that directly or indirectly determines the value of an annotated signal. For example, Listing 1 shows the annotated HDL code, and Figure 3 shows the circuit synthesized from this HDL code. The increment on line 4 is the signal annotated by (* SURGE_FREQ=1 *), indicating that the goal is to frequently set this signal to 1. The in[0], in[1], in[2], and t are its ancestor registers. This annotation is intended to create test inputs that execute the increment to the cnt register at a high frequency. While these ancestor registers are typically selected from those close to the annotated signal, the detailed selection method is explained in Section IV.

SurgeFuzz explores the state space of ancestor registers to generate test inputs with high coverage. The reason for observing ancestor registers is that when the annotated signal is a 1-bit signal, it can only take on a state of either 0 or 1, resulting in a state space consisting only of the signal that is too small for meaningful exploration. Thus, SurgeFuzz explores the state of the ancestor registers, which determines the future value of the annotated signal.

The state space to be explored consists only of the states of ancestor registers, making it significantly smaller than the entire state

Listing 2: Example of an instrumented code.

```
wire [3:0] coverage;
assign coverage = {in[0], in[1], in[2], t};
```

space of CPUs used in existing methods. For example, in Figure 3, registers c1 and c2 are not ancestor registers, and thus their states are not explored. SurgeFuzz improves the efficiency of the verification by excluding the exploration of the state of registers that are not related to the annotated signal. These unrelated registers are typically abundant in CPUs.

2) SurgeFuzz directionally explores this small state space to achieve objectives given by annotations, such as reaching states where the signal is frequently active (Section III-E). This directed exploration is realized in the seed selection phase (1) of the feedback-driven fuzzing, as shown in Figure 1, by preferentially selecting seeds that yield a higher frequency of the event indicated by the annotation.

C. Annotations

This section describes the details of three annotations in SurgeFuzz.

1) SURGE_FREQ=P instructs, as mentioned in the previous example, a 1-bit signal to be activated (P=1) or deactivated (P=0) at a high frequency within a short period. For example, by annotating a signal that becomes active on branch mispredictions, it is possible to generate programs that frequently induce branch mispredictions.
2) SURGE_CONSEC=P instructs that the duration when a 1-bit signal is continuously active (P=1) or inactive (P=0) should be extended. This annotation is used in scenarios where a specific event not only occurs at a high frequency but also continues for a certain period. For example, in Listing 5 presented in the later evaluation, we generate programs that prevent the commit of a store instruction for a long period by annotating a signal that indicates a blockage of the commits in the CPU.
3) SURGE_COUNT=P instructs to increase (P=MAX) or decrease (P=MIN) the value of a certain signal. Unlike the two annotations above, SURGE_COUNT is used when instructing a part that indirectly indicates the frequent occurrence of events. For example, if the number of elements in a queue is explicitly stored in a register, such a register can be annotated. This generates a program that frequently inserts elements into the queue and utilizes as much of its capacity as possible.

D. Coverage Metric using Ancestor Registers

SurgeFuzz primarily uses the contents of the ancestor registers, concatenated as a bit sequence, as a coverage metric. For example, as shown in Listing 2, a signal to measure coverage is automatically inserted into the HDL code, enabling the fuzzer to observe it. In this example, the selected ancestor registers in[0], in[1], in[2], and t are concatenated and used. The ancestor registers are basically chosen from those close to the annotated signal, with the selection method detailed later in Section IV.

SurgeFuzz measures this bit sequence every cycle to explore the state space of target designs. In accordance with the feedback-driven fuzzing framework (Section II-A), inputs that enhance the coverage of this bit sequence, i.e., triggering new states, are added to the seed pool as new seeds. By focusing solely on the state of the ancestor registers that determine the state of the annotated signal, SurgeFuzz can perform more efficient exploration compared to existing methods that explore the state of registers in the entire CPU design.

E. Directed Exploration using Temporal Density

We employ *power scheduling* [39], [40] to accomplish directed exploration. Power scheduling introduces energy E to each seed in the feedback-driven fuzzing described in Section II-A. This energy determines the probability of the seed being selected from the seed pool. Seeds with higher energy yield more test inputs, thereby facilitating directed exploration. In SurgeFuzz, higher energy is assigned to inputs that more effectively achieve the objective specified by the annotation. For example, in Listing 1, higher energy is assigned to test inputs where the signal increment is set to one more frequently.

We define a power scheduling function that assigns energy to a seed input as follows:

$$p(\mathcal{T}) = score(\mathcal{T})^2 \tag{1}$$

where $\mathcal{T} = t_1, t_2, \cdots t_N$ represents the series of values held by the annotated signal in each cycle of the N-cycle simulation using the seed input, and t_i denotes the value at the i-th cycle. The function $score(\mathcal{T})$ calculates the score for power scheduling from the series of values held by the annotated signal, and is defined as follows according to the type of annotation[1]:

$$score(\mathcal{T}) = \begin{cases} \max_{i=1}^{N-M+1} \sum_{j=i}^{i+M-1} t_j & \text{(FREQ)} \\ \max_{i=1}^{N} c_i & \text{(CONSEC)} \\ \max_{i=1}^{N} t_i & \text{(COUNT)} \end{cases} \tag{2}$$

where M in FREQ indicates the number of cycles in the window measuring the frequency. Therefore, in FREQ, the frequency is measured every M cycles, and the highest frequency throughout the simulation is used as the score. c_i in CONSEC is an auxiliary variable that represents the number of cycles where the signal continuously takes the value 1 up to the i-th cycle, and is computed as follows:

$$c_i = \begin{cases} c_{i-1} + 1 & \text{if } t_i = 1 \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

where $c_0 = 0$.

The function $score(\mathcal{T})$ yields a larger value when a more intense surge is occurring, thereby also increasing the value of the energy function $p(\mathcal{T})$. As a result, seed inputs with higher energy are preferentially used in the exploration. In this paper, we increase energy in a quadratic manner, aiming to allocate more energy to crucial inputs compared to a linear manner.

IV. Details of the Ancestor Register Selection Method

In the following, we describe a simple method of selecting ancestor registers based on the distance from an annotated signal, and then describe an optimized method based on mutual information.

A. Distance-based Selection

A *distance-based algorithm* selects a specified number of ancestor registers in order of distance from an annotated signal. Here, the number of registers between a signal and another register connected through combinational logic is defined as the *register distance*, which is used as a criterion for selection. For example, in Figure 3, the distance between increment and t is zero, the register distance between t and in[0] is one (including itself when the register is the starting point), and the register distance between cnt and in[0] is two.

Suppose we annotate increment using SURGE_FREQ and select three bits for the ancestor registers in total. Initially, the two bits, t and

[1]These formulas assume the case of P=1 or P=MAX. For the cases of P=0 or P=MIN, the formulas should be appropriately inverted.

in[2], which are closest to increment, are selected. Then, both in[0] and in[1] are connected to increment through one register, thus they have the same distance of one. In such situations, one of the registers of the same distance is selected randomly. That is, in addition to t and in[2], either in[0] or in[1] is selected as the ancestor registers.

This distance-based algorithm is implemented by analyzing a directed graph that represents the circuit structure synthesized from the HDL code. In this directed graph, each node corresponds to a register, and the edges represent connections from each register to another upstream register through combinational circuits. For instance, in the case of Figure 3, there are two edges from the node t to the nodes in[0] and in[1]. A breadth-first search is performed on this graph, starting from the annotated signal, to select the ancestor registers in order of their proximity.

B. Mutual Information-based Pruning

As mentioned above, the distance-based algorithm selects the nearest ancestor registers from an annotated signal. However, the method of simply selecting the closest ones may result in selecting *redundant* ancestor registers whose contents change synchronously.

A typical example of such redundant registers is pipeline registers, which are inserted to increase operating frequency. Especially in short paths, pipeline registers may be inserted that are directly connected without any combinational circuits. Such registers are synchronous in content and their measurement is redundant. A more complex example is t, in[0] and in[1] in Figure 3. t is strongly correlated to the values of in[0] and in[1] because the results of the AND operations on in[0] and in[1] are connected. Thus, in[0], in[1], and t are redundant in terms of coverage measurement. Since the total number of bits in the ancestor registers greatly affects the storage capacity required for the fuzzer[2], such redundant registers should be eliminated as much as possible.

We propose a method to improve the efficiency of fuzzing by removing redundant registers in ancestor registers based on *mutual information*. Mutual information in information theory is a measure of the interdependence of two random variables. Mutual information can be regarded as the amount of information that one random variable contains about another random variable. Using this notion of mutual information, we measure the amount of information that a register contains about other registers. If the amount of mutual information is large, then those registers are redundant and one of them can be removed without reducing the quality of the coverage metrics. This allows more states to be represented per the same number of registers.

1) Mutual Information between Registers: In the following, we show how to compute the normalized mutual information between two arbitrary bit-width registers x and y. We simulate a target RTL design for N cycles with some input and sample the values of the two registers in each cycle. Let $\mathcal{X}$ be the set of values observed in register x, and $\mathcal{Y}$ be the set of values observed in register y. In this case, the normalized mutual information is given by:

$$I(x, y, \mathcal{X}, \mathcal{Y}) = H(x, \mathcal{X}) - H(x|y, \mathcal{X}, \mathcal{Y}) \tag{4}$$

where $H(x, \mathcal{X})$ is the entropy of register x and $H(x|y, \mathcal{X}, \mathcal{Y})$ is the conditional entropy of register x given register y. $H(x, \mathcal{X})$ and $H(x|y, \mathcal{X}, \mathcal{Y})$ are given by:

$$H(x, \mathcal{X}) = -\sum_{x_i \in \mathcal{X}} \frac{C(x_i)}{N} \log \frac{C(x_i)}{N} \tag{5}$$

[2]If a total of N bits of ancestor registers are used as coverage indicators, $O(2^N)$ bits of capacity are needed in the fuzzer to record the possible states.

Algorithm 1: Selection of Ancestor Registers using MI

Input: register graph generated from HDL code G, root node representing annotated signal R_s, total number of bits of ancestor registers to be selected T_a, NMI threshold for selection of ancestor registers T_n, sampling data D
Output: set of all selected ancestor registers A

```
1  A ← ∅;
2  queue ← {R_s};
3  visited ← {R_s};
4  while queue ≠ ∅ do
5      u ← Pop(queue);
6      foreach v ∈ GetParents(G, u) do
7          if v ∈ visited then
8              continue
9          if IsValuable(A, D, r) then
10             A ← A ∪ {r};
11             if GetBitSize(A) ≥ T_a then
12                 return A;
13             Push(queue, v);
14 return A;
15
16 Function IsValuable (A, D, r):
17     foreach a ∈ A do
18         if NMI(a, r, D[a], D[r]) ≥ T_n then    // NMI=Eq.7
19             return false;
20     return true;
```

$$H(x|y, \mathcal{X}, \mathcal{Y}) = -\sum_{x_i \in \mathcal{X}, y_j \in \mathcal{Y}} \frac{C(x_i)}{N} \frac{C(y_j|x_i)}{C(x_i)} \log \frac{C(y_j|x_i)}{C(x_i)} \tag{6}$$

where $C(x_i)$ is the amount of value x_i observed in register x, and $C(y_j|x_i)$ is the amount of value y_j observed in register y when value x_i is observed in register x. Since the maximum mutual information depends on the bit widths of x and y, we use the normalized one given by:

$$NMI(x, y, \mathcal{X}, \mathcal{Y}) = \frac{2 \times I(x, y, \mathcal{X}, \mathcal{Y})}{H(x, \mathcal{X}) + H(y, \mathcal{Y})} \tag{7}$$

where $H(y, \mathcal{Y})$ is the entropy of register y, which is obtained similarly to $H(x, \mathcal{X})$.

2) Ancestor Register Selection using Mutual Information: Algorithm 1 presents the process for selecting ancestor registers using the previously mentioned normalized mutual information. Similarly to the distance-based method, this algorithm performs a breadth-first search on a graph representing the register connectivity. That is, it selects ancestor registers through a breadth-first search using the annotated signal as the starting point (lines 4-13). The difference from the distance-based algorithm is that when a new candidate register is found, a decision is made whether to add it to the list of already selected registers based on the amount of mutual information (line 9). At this time, the normalized mutual information is calculated between the registers already selected as ancestors and candidates (lines 16-20). If one or more of the obtained normalized mutual information values are larger than a given threshold, the candidate register is not inserted into the selected list. This improves coverage metrics by selecting more valuable registers and removing redundant ones.

V. Evaluation

A. Evaluation Setup

We evaluated SurgeFuzz and the existing state-of-the-art fuzzers for RTL designs: RFUZZ [17], DifuzzRTL [18], and DirectFuzz [19].

In our evaluation, the same parameters were basically used except for coverage metrics and power scheduling. The basic fuzzing algorithms, such as input generation and seed selection, were implemented based on those of DifuzzRTL and are common to all evaluated fuzzers. For the DirectFuzz evaluation, we selected the modules annotated in the SurgeFuzz evaluation as target modules. We also evaluated riscv-torture [14], a random generator widely used to verify various RISC-V CPU designs, to compare traditional dynamic verification techniques with SurgeFuzz.

We evaluated these methods on a machine with two AMD EPYC 7713 64-core processors and 1TB of memory. We implemented 1) automatic ancestor register identification (Section III) by analyzing the dependency graph generated from RTL code, and 2) code instrumentation for coverage recording on Yosys [41] version 0.21. We also used Verilator [42] version 4.216, a simulation environment supporting SystemVerilog, to simulate the RTL code.

We used RSD [37], NaxRiscv [43], and Boom [38], which are RISC-V out-of-order superscalar processors, for our evaluation. These CPUs are relatively large and complex and are capable of dynamically scheduling many instructions, including memory accesses, and executing them out-of-order.

B. Evaluated Cases

To evaluate the effect of introducing SurgeFuzz, we created twelve different cases for adding annotations to the CPU designs, as listed in Table I. We searched through the targeted designs using keywords such as "conflict" and "busy" to identify logic associated with surges and added annotations to them.

In cases A1 through A4, we added the SURGE_FREQ annotations to signals that become active when an event that leads to exceptional handling occurs to generate surges. In A1, we annotated a signal that becomes active when an exception occurs in RSD, as shown in Listing 3. In A2, we annotated a signal that becomes active when a branch prediction miss occurs in NaxRiscv, as shown in Listing 4. In A3, we annotated a signal that becomes active when a way conflict in a data cache occurs in NaxRiscv, and in A4, we annotated a signal that becomes active when a load queue becomes full in BOOM.

In cases A5 through A7, we added the SURGE_CONSEC annotation to signals that become active when certain events are being blocked, with the goal of increasing the number of cycles in which the events are continuously blocked. In A5, we added the annotation to a signal indicating that the issue of instructions is blocked. In A6, we added the annotation to a signal indicating that there is no space in the miss status handling register (MSHR), which is responsible for managing missed accesses to caches. In A7, as shown in Listing 5, we newly defined the sc_blocked signal to indicate that the commit of a store instruction is blocked, and we added the annotation to the signal. As demonstrated in this example, if there is no signal that represents a desired case, we can define a new signal and annotate it to verify various cases.

In cases A8 through A12, we added the SURGE_COUNT annotations to registers that manage the usage of a queue, with the goal of increasing the usage of the queue. Specifically, we added annotations to registers in the load queue (LQ), store queue (SQ), and replay queue (RQ) in RSD and in the LQ and SQ in NaxRiscv. The LQ/SQ are units for scheduling speculative accesses to memory on these processors, and the RQ is a unit that manages instructions replayed in the event of a speculation miss. The number of entries in the LQ and SQ is 16, and the number of entries in the replay queue is 20. For example, in A8, we annotated curCount, a register that manages the current usage of the LQ, as shown in Listing 6.

TABLE I: Summary of evaluated cases.

ID	Type	CPU	Brief description
A1	FREQ	RSD	Frequent exceptions.
A2	FREQ	NaxRiscv	Frequent branch prediction misses.
A3	FREQ	NaxRiscv	Frequent DataCache way conflict.
A4	FREQ	BOOM	Frequently fill the load queue.
A5	CONSEC	RSD	Stall the scheduler longer.
A6	CONSEC	RSD	Keep MSHR busy longer.
A7	CONSEC	BOOM	Block store commits longer.
A8	COUNT	RSD	High load queue usage.
A9	COUNT	RSD	High store queue usage.
A10	COUNT	RSD	High replay queue usage.
A11	COUNT	NaxRiscv	High load queue usage.
A12	COUNT	NaxRiscv	High store queue usage.

Listing 3: Annotated code snippet for case A1.

```
(* SURGE_FREQ=1 *) wire exceptionDetected;
```

Listing 4: Annotated code snippet for scenario A2.

```
branchMissEvent := RegNext(reschedule.valid &&
    reschedule.reason === ScheduleReason.BRANCH)
    init(False) addAttribute("SURGE_FREQ=1")
```

Listing 5: Annotated code snippet for case A7.

```
val sc_blocked = can_fire_store_commit &
    !will_fire_store_commit
annotate(new ChiselAnnotation{
  override def toFirrtl = AttributeAnnotation(
    sc_blocked.toTarget, "SURGE_CONSEC=1")
})
```

Listing 6: Annotated code snippet for case A8.

```
(* SURGE_COUNT="MAX" *) [4:0] reg curCount;
```

C. Efficiency of Surge Generation

We initially evaluated the impact of surge generation for the evaluated cases shown in Table I. We ran 30 instances for each method to take into account the random nature of fuzzing and evaluated the average of those results. Since the simulation time varies with each CPU design, the evaluation duration varies from 1 hour, 4 hours, or 24 hours depending on the CPU design used in each case.

Figure 4 shows the transition of the score defined by Equation 2 over the execution time. It demonstrates that **SurgeFuzz** can generate surge-inducing programs more effectively than the existing methods. In most cases, **SurgeFuzz** exhibited a rapid increase in its score within the first few tens of minutes, consistently maintaining a significantly higher level than that of the existing methods until the end. In contrast, the existing methods showed slower score growth, with some cases reaching a saturation point in their growth early on. In particular, the score of **riscv-torture** changed very little throughout the execution time, because it can only generate inputs according to pre-defined constraints.

Next, we evaluated the effectiveness of the following two methods used by SurgeFuzz: (a) the register selection method based on mutual information described in Section IV-B and (b) the power scheduling strategy described in Section III-E. (1) **SurgeFuzz w/o mi** represents a model that disables the mutual information-based pruning and employs the naive distance-based selection method explained in Section IV-A. (2) **SurgeFuzz w/o ps** represents a model that disables the power scheduling strategy.

Since the trends in these evaluation results did not vary significantly across all the cases A1-A12, we only show the results for

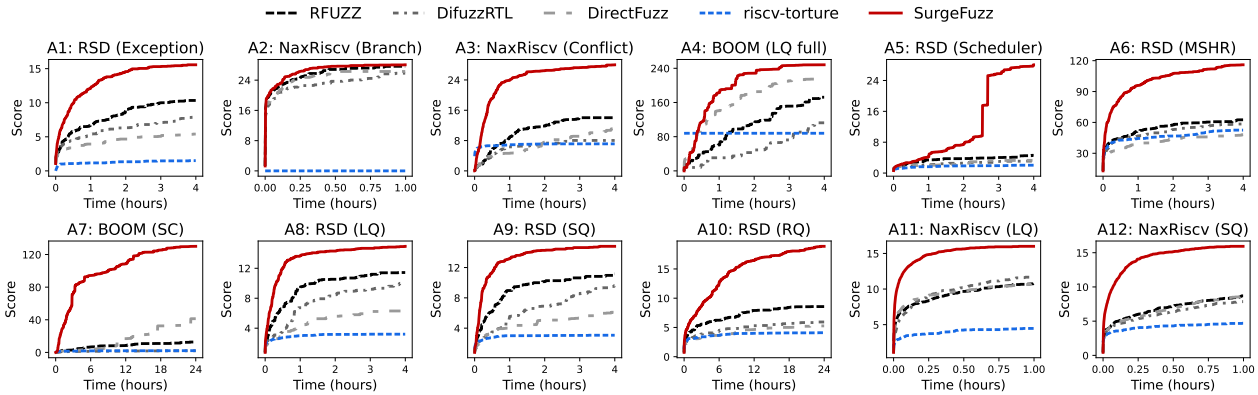

Fig. 4: Transition of scores indicating surge intensity. The x- and y-axes represent the elapsed time and the average of the maximum value of the score achieved up to that point over 30 runs, respectively. The higher the line, the faster the surge-inducing programs can be generated.

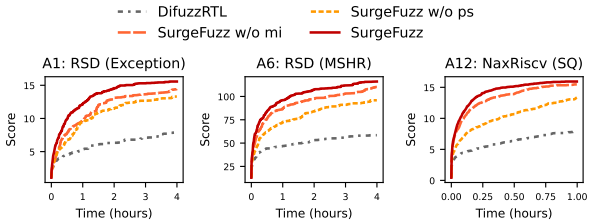

Fig. 5: Transition of scores in the three variants of SurgeFuzz and DifuzzRTL. This figure is interpreted in the same way as Figure 4.

several representative cases. Figure 5 shows the evaluation results for the two variants of SurgeFuzz, as well as for **SurgeFuzz** and **DifuzzRTL**. All these results show that **SurgeFuzz** performs better than **SurgeFuzz w/o mi** and **SurgeFuzz w/o ps**. The performance degradation is larger for **SurgeFuzz w/o ps** than for **SurgeFuzz w/o mi**, indicating that the power scheduling contributes significantly to performance. Despite being identical except for the coverage metrics, **SurgeFuzz w/o ps** exhibits superior performance to **DifuzzRTL**. This indicates that the proposed coverage metric with a small state space works effectively.

D. Discovered Bugs

We found six bugs, listed in Table II, from RSD and Boom using SurgeFuzz and the annotations listed in Table I. These bugs were detected by triggering assertions intended to detect issues such as deadlocks that were originally embedded in the target HDL code. Among them, bugs B1 to B5 discovered by RSD were previously unknown. RSD has been verified using tests specifically developed for it and various public test suites for RISC-V, but such tests did not find these bugs. These bugs were not known at the time of our evaluation, and we created the annotation without any knowledge of the location of these bugs.

As indicated in Table II, these bugs are very complex and can only be detected with inputs that satisfy complex constraints. In particular, bug B5 occurs only under a very limited condition[3]. It is difficult and expensive for developers to manually create test cases for such complex constraints. This result shows that SurgeFuzz can detect hard-to-find bugs with little effort by intentionally inducing surges based on annotations.

[3]This bug can be triggered if an instruction, sharing the same ROB index as its preceding instruction, is flushed during a period where instruction scheduling is stalled. This stall is typically due to consecutive recovery operations following the issue of a long-latency instruction, such as Load or Multiply/Divide.

E. Efficiency of Bug Detection

We evaluated how quickly and how likely the six bugs listed in Table II can be found by SurgeFuzz compared to existing methods. Figure 6 shows the time distribution to find these bugs. We ran each method for 24 hours and plotted the time at which we first found each bug. If we did not find a bug within 24 hours, we do not plot the time. Since fuzzing results are stochastic, we ran 30 instances of each method and plotted all the results. Note that, riscv-torture could not find any bugs and is therefore not shown in the figure.

Figure 6 shows that SurgeFuzz can detect various bugs faster and with higher probability than the existing methods. When comparing the bug detection times for B1 and B5, the median detection time for SurgeFuzz is less than that of the existing methods. These results indicate that SurgeFuzz can quickly detect bugs B1 and B5. Furthermore, when comparing the probability of bug detection for bugs B2, B3, B4, and B6, SurgeFuzz is significantly higher than existing methods. For instance, the existing methods can hardly detect bugs B2 and B4, while SurgeFuzz can detect these bugs at an average probability of 42% and 27%, respectively. The results for SurgeFuzz are independent of the annotation location and show the same tendency overall.

In addition, while the existing methods were unable to detect bug B6 at all, SurgeFuzz was able to detect it. Specifically, SurgeFuzz with case A7 detected this bug with a higher probability than that with case A5. This result implies that there is a strong correlation between the scenario in A7 and the cause of bug B6. Although case A4 uses a very simple annotation, it was able to detect bug B6. This result implies that it is possible to detect complex bugs using simple annotations without deep knowledge of targeted designs.

VI. Related Work

Dynamic verification methods are widely used in CPU design verification due to their ease of application [9]–[15], [44]. For instance, in the verification of RISC-V CPUs, there are tools such as riscv-torture [14] and riscv-dv [15], which automatically generate programs for test inputs. These tools randomly generate programs based on manually created constraints, without utilizing runtime information as typical fuzzing methods do. This makes it difficult to generate programs with complex behavior and to adequately verify complex designs. In addition, a technique known as coverage directed test generation has been proposed, which uses coverage feedback

TABLE II: Summary of bugs detected by SurgeFuzz. The ✓in the "New Bug" column indicates that the bugs were unknown before applying SurgeFuzz. Thus, bugs B1 to B5 were first discovered by SurgeFuzz.

ID	CPU	Description of bugs	New Bug
B1	RSD	If an exception is caused during a recovery process from a speculation miss, the exception may not be processed.	✓
B2	RSD	When the ROB is full, a speculation miss in the first instruction in the ROB may cause some units to be incorrectly recovered.	✓
B3	RSD	Data may be incorrectly forwarded from the store queue when a load instruction that accesses an invalid area is executed.	✓
B4	RSD	If an instruction is flushed immediately after it is replayed from the replay queue, the associated resources in the MSHR may not be released.	✓
B5	RSD	If recovery processes are performed continuously, an instruction may not wake up correctly in the scheduler.	✓
B6	BOOM	If wakeups of the load instruction are consecutive, a deadlock may occur in a commit of the store instruction.	

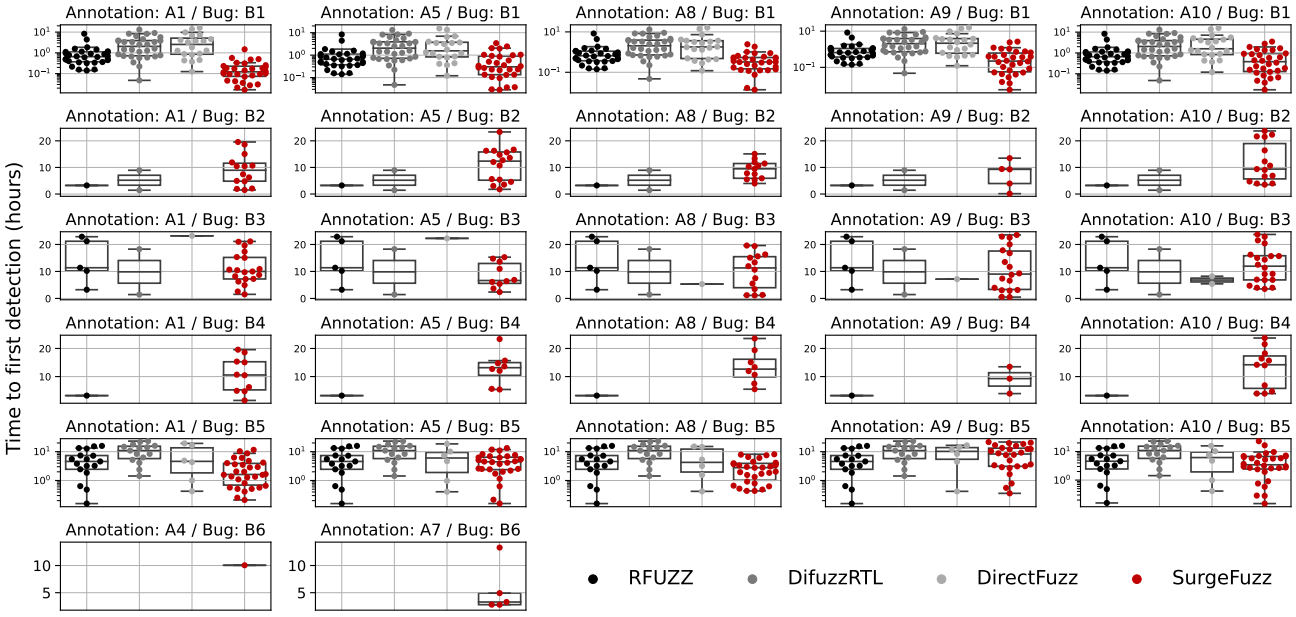

Fig. 6: Distribution of detection times for the six bugs shown in Table II for each method. In each plot, lower points indicate that the bug was found faster, and more points indicate that the bug was found with a higher probability. Each row represents the results for bugs B1 to B6 in order, and each column shows the results for different annotations. Since RFUZZ and DifuzzRTL do not use annotations, the same data are plotted across each row. Because riscv-torture was unable to find any bugs at all, it is omitted from this figure. Since each fuzzer executed 30 instances, up to 30 points can be plotted. For example, the plot for "Annotation: A1 / Bug: B4" represents the results for bug B4 listed in Table II when using annotation A1 as shown in Table I. In this plot, RFUZZ and SurgeFuzz have 1 or 10 points plotted respectively, indicating a probability of detecting a bug of 1/30 and 1/3 respectively. On the other hand, DifuzzRTL and DirectFuzz do not have any points plotted, indicating no bug detection at all.

to update the constraints of the test generator at runtime [9]–[11]. However, in most cases, this technique requires a deep understanding of each target design [19], and thus, it cannot be easily applied.

Similar to our approach, techniques that leverage human knowledge and assistance to identify hard-to-find bugs in CPU designs have also been proposed [11], [21]. For example, StressTest [11] generates inputs that effectively stress the points of interest using a Markov model. While most existing fuzzing methods focus on input generation, Logic Fuzzer [21] randomizes the internals of CPU designs themselves. Logic Fuzzer randomizes the state or control signal of each test target without affecting its logical functionality and aims to explore states that are difficult to reach with normal input. However, such techniques often require significant domain knowledge and user effort, and thus the setup cost is expensive. On the other hand, SurgeFuzz only requires simple annotations and can be relatively easily applied, even without knowledge of target CPUs.

In the field of software fuzzing, there exist methods that target hard-to-test corner cases and rare states, similar to our approach [39], [45]–[47]. AFLFast [39] and FairFuzz [45] automatically focus on less frequently executed paths or edges, thereby increasing the chances of uncovering bugs that other fuzzing methods might overlook. On the other hand, IJON [46] and FuzzFactory [47] focus more directly on specific targets, such as corner cases and rare states, based on human instructions. As mentioned above, these software fuzzing methods are difficult to apply directly to CPU designs. Furthermore, SurgeFuzz differs significantly from these methods in that it focuses on and annotates surges, which are temporal phenomena.

VII. Conclusion

While significant advancements have been made in the field of dynamic verification for CPU designs, the challenge of effectively verifying exceptional handling remains unresolved. To tackle this problem, we proposed a new directed fuzzing method called SurgeFuzz. SurgeFuzz employs annotations to induce 'surges' - situations where events related to exceptional handling occur at a high density - enabling more efficient verification of potential bug-prone areas in CPU designs. Using a relatively large and complex RISC-V CPU as a test bench, we demonstrated that SurgeFuzz, solely with simple annotations, can induce surges and uncover bugs more efficiently compared to existing methods.

References

[1] G. Dessouky, D. Gens, P. Haney, G. Persyn, A. K. Kanuparthi, H. Khattri, J. M. Fung, A. Sadeghi, and J. Rajendran, "Hardfails: Insights into software-exploitable hardware bugs," in *USENIX Security Symposium (USENIX Security)*, 2019, pp. 213–230.

[2] Intel, "Intel® Core™ x-series processor family - specification update."

[3] Freescale Semiconductor, "Chip errata for the i. MX 6Dual/6Quad."

[4] AMD, "Revision guide for AMD family 17h models 00h-0fh processors."

[5] AMD, "Revision guide for AMD family 19h models 00h-0fh processors."

[6] R. Kaivola, R. Ghughal, N. Narasimhan, A. Telfer, J. Whittemore, S. Pandav, A. Slobodová, C. Taylor, V. A. Frolov, E. Reeber, and A. Naik, "Replacing testing with formal verification in Intel Core i7 processor execution engine validation," in *International Conference on Computer Aided Verification (CAV)*, 2009, pp. 414–429.

[7] A. Ahmed, F. Farahmandi, and P. Mishra, "Directed test generation using concolic testing on RTL models," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2018, pp. 1538–1543.

[8] R. Zhang, C. Deutschbein, P. Huang, and C. Sturton, "End-to-end automated exploit generation for validating the security of processor designs," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 815–827.

[9] S. Fine and A. Ziv, "Coverage directed test generation for functional verification using Bayesian networks," in *ACM/IEEE Design Automation Conference (DAC)*, 2003, pp. 286–291.

[10] G. Squillero, "Microgp-an evolutionary assembly program generator," *Genetic Programming and Evolvable Machines*, vol. 6, no. 3, pp. 247–263, 2005.

[11] I. Wagner, V. Bertacco, and T. M. Austin, "Stresstest: an automatic approach to test generation via activity monitors," in *ACM/IEEE Design Automation Conference (DAC)*, 2005, pp. 783–788.

[12] J. Wang, H. Li, T. Lv, T. Wang, X. Li, and S. Kundu, "Abstraction-guided simulation using Markov analysis for functional verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 2, pp. 285–297, 2016.

[13] V. Herdt, D. Große, E. Jentzsch, and R. Drechsler, "Efficient cross-level testing for processor verification: A RISC-V case-study," in *Forum on Specification & Design Languages (FDL)*, 2020, pp. 1–7.

[14] "riscv-torture," https://github.com/ucb-bar/riscv-torture.

[15] "riscv-dv," https://github.com/chipsalliance/riscv-dv.

[16] Y. Lyu and P. Mishra, "Scalable concolic testing of RTL models," *IEEE Transactions on Computers*, vol. 70, no. 7, pp. 979–991, 2021.

[17] K. Laeufer, J. Koenig, D. Kim, J. Bachrach, and K. Sen, "RFUZZ: coverage-directed fuzz testing of RTL on FPGAs," in *ACM/IEEE International Conference on Computer-Aided Design (ICCAD)*, 2018, pp. 1–8.

[18] J. Hur, S. Song, D. Kwon, E. Baek, J. Kim, and B. Lee, "DifuzzRTL: Differential fuzz testing to find CPU bugs," in *IEEE Symposium on Security and Privacy (S&P)*, 2021, pp. 1286–1303.

[19] S. Canakci, L. Delshadtehrani, F. Eris, M. B. Taylor, M. Egele, and A. Joshi, "DirectFuzz: Automated test generation for RTL designs using directed graybox fuzzing," in *ACM/IEEE Design Automation Conference (DAC)*, 2021, pp. 529–534.

[20] S. K. Muduli, G. Takhar, and P. Subramanyan, "HyperFuzzing for SoC security validation," in *ACM/IEEE International Conference on Computer-Aided Design (ICCAD)*, 2020, pp. 1–9.

[21] N. Kabylkas, T. Thorn, S. Srinath, P. Xekalakis, and J. Renau, "Effective processor verification with logic fuzzer enhanced co-simulation," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2021, pp. 667–678.

[22] T. Trippel, K. G. Shin, A. Chernyakhovsky, G. Kelly, D. Rizzo, and M. Hicks, "Fuzzing hardware like software," in *USENIX Security Symposium (USENIX Security)*, 2022, pp. 3237–3254.

[23] R. Kande, A. Crump, G. Persyn, P. Jauernig, A. Sadeghi, A. Tyagi, and J. Rajendran, "TheHuzz: Instruction fuzzing of processors using golden-reference models for finding software-exploitable vulnerabilities," in *USENIX Security Symposium (USENIX Security)*, 2022, pp. 3219–3236.

[24] C. Chen, R. Kande, N. Nguyen, F. Andersen, A. Tyagi, A. Sadeghi, and J. Rajendran, "HyPFuzz: Formal-assisted processor fuzzing," in *USENIX Security Symposium (USENIX Security)*, 2023, pp. 1361–1378.

[25] J. Xu, Y. Liu, S. He, H. Lin, Y. Zhou, and C. Wang, "MorFuzz: Fuzzing processor via runtime instruction morphing enhanced synchronizable co-simulation," in *USENIX Security Symposium (USENIX Security)*, 2023, pp. 1307–1324.

[26] K. Constantinides, O. Mutlu, and T. M. Austin, "Online design bug detection: RTL analysis, flexible mechanisms, and evaluation," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2008, pp. 282–293.

[27] R. C. Ho, C. H. Yang, M. Horowitz, and D. L. Dill, "Architecture validation for processors," in *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 1995, pp. 404–413.

[28] B. Bailey, "When bugs escape," 2018. [Online]. Available: https://semiengineering.com/when-bugs-escape

[29] H. Arbel, "Bug escapes and the definition of done," 2021. [Online]. Available: https://semiengineering.com/bug-escapes-and-the-definition-of-done

[30] M. Böhme, V. Pham, M. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017, pp. 2329–2344.

[31] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu, "Hawkeye: Towards a desired directed grey-box fuzzer," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018, pp. 2095–2108.

[32] S. Österlund, K. Razavi, H. Bos, and C. Giuffrida, "Parmesan: Sanitizer-guided greybox fuzzing," in *USENIX Security Symposium (USENIX Security)*, 2020, pp. 2289–2306.

[33] H. Huang, Y. Guo, Q. Shi, P. Yao, R. Wu, and C. Zhang, "BEACON: directed grey-box fuzzing with provable path pruning," in *IEEE Symposium on Security and Privacy (S&P)*, 2022, pp. 36–50.

[34] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang, "Fuzzing: State of the art," *IEEE Transactions on Reliability*, vol. 67, no. 3, pp. 1199–1218, 2018.

[35] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2312–2331, 2021.

[36] X. Zhu, S. Wen, S. Camtepe, and Y. Xiang, "Fuzzing: A survey for roadmap," *ACM Computing Surveys*, vol. 54, no. 11s, pp. 1–36, 2022.

[37] S. Mashimo, K. Inoue, R. Shioya, A. Fujita, R. Matsuo, S. Akaki, A. Fukuda, T. Koizumi, J. Kadomoto, H. Irie, and M. Goshima, "An open source FPGA-optimized out-of-order RISC-V soft processor," in *IEEE International Conference on Field-Programmable Technology (FPT)*, 2019, pp. 63–71.

[38] J. Zhao, B. Korpan, A. Gonzalez, and K. Asanovic, "SonicBOOM: The 3rd generation Berkeley out-of-order machine," vol. 5, 2020.

[39] M. Böhme, V. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as Markov chain," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016, pp. 1032–1043.

[40] M. Böhme, V. J. M. Manès, and S. K. Cha, "Boosting fuzzer efficiency: an information theoretic perspective," in *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2020, pp. 678–689.

[41] "Yosys," https://yosyshq.net/yosys.

[42] "Verilator," https://www.veripool.org/verilator.

[43] "NaxRiscv," https://github.com/SpinalHDL/NaxRiscv.

[44] B. Bentley, "Validating the Intel Pentium 4 microprocessor," in *ACM/IEEE Design Automation Conference (DAC)*, 2001, pp. 244–248.

[45] C. Lemieux and K. Sen, "FairFuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2018, pp. 475–485.

[46] C. Aschermann, S. Schumilo, A. Abbasi, and T. Holz, "IJON: Exploring deep state spaces via fuzzing," in *IEEE Symposium on Security and Privacy (S&P)*, 2020, pp. 1597–1612.

[47] R. Padhye, C. Lemieux, K. Sen, L. Simon, and H. Vijayakumar, "FuzzFactory: Domain-specific fuzzing with waypoints," *ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–29, 2019.