INSTILLER: Towards Efficient and Realistic RTL Fuzzing

Gen Zhang, Pengfei Wang⊠, Tai Yue, Danjun Liu, Yubei Guo and Kai Lu⊠
National University of Defense Technology

***Abstract*—Bugs exist in hardware, such as CPU. Unlike software bugs, these hardware bugs need to be detected before deployment. Previous fuzzing work in CPU bug detection has several disadvantages, e.g., the length of RTL input instructions keeps growing, and longer inputs are ineffective for fuzzing.**

In this paper, we propose INSTILLER (Instruction Distiller), an RTL fuzzer based on ant colony optimization (ACO). First, to keep the input instruction length short and efficient in fuzzing, it distills input instructions with a variant of ACO (VACO). Next, related work cannot simulate realistic interruptions well in fuzzing, and INSTILLER solves the problem of inserting interruptions and exceptions in generating the inputs. Third, to further improve the fuzzing performance of INSTILLER, we propose hardware-based seed selection and mutation strategies.

We implement a prototype and conduct extensive experiments against state-of-the-art fuzzing work in real-world target CPU cores. In experiments, INSTILLER has 29.4% more coverage than DiFuzzRTL. In addition, 17.0% more mismatches are detected by INSTILLER. With the VACO algorithm, INSTILLER generates 79.3% shorter input instructions than DiFuzzRTL, demonstrating its effectiveness in distilling the input instructions. In addition, the distillation leads to a 6.7% increase in execution speed on average.

***Index Terms*—fuzzing, RTL, hardware security.**

I. INTRODUCTION

CPU bugs are notorious. Besides the well-known Meltdown and Spectre [1], numerous bugs are reported, including the Pentium FDIV bug [2], Broadwell MCE bug [3], and Ryzen segfault bug [4]. All of them can cost the manufacturers millions or billions of dollars in mitigating and repairing the bugs.

Before CPU deployment, the circuits and RTL (register transition level) designs should be thoroughly verified. In software deployment, bugs can be avoided with timely patches. However, in the development of CPU, once deployed, it is nearly impracticable to remove the impact of hardware vulnerabilities. For example, the mitigation of Meltdown and Spectre only focuses on part of the mainstream products, due to the challenging balance among the mitigation itself, performance impact, and implementation complexity [5].

Previous work has made some attempts to detect CPU bugs [6], [7], both in static and dynamic techniques. Among them, verifying CPU with fuzz testing [8] is one of the most promising approaches [9], [10]. However, there are still several drawbacks to these techniques, and they extend to the following challenges.

Challenge 1: growing input length. The basic structure of the input instructions of the CPU consists of an instruction sequence. Interruptions and exceptions can be inserted to simulate the real-world execution of the CPU. Starting from a simple instruction, as the fuzzing process goes on, the length of input tends to increase. The speed of fuzz testing is the key to its success [11]–[16]. Longer input instructions are catastrophic to the execution speed of fuzzing since longer inputs will spend more CPU cycles. More importantly, according to our analysis in the evaluation in Section V-C, coverage does not increase proportionally to input length. Therefore, we should come up with solutions to shorten the input instructions and improve the fuzzing efficiency.

Challenge 2: realistic interruption and exception handling. Interruptions and exceptions are common in the execution of the CPU. Simulating them in testing CPU can cover the corner cases of CPU verification. Previous fuzzing work mentioned considering interruptions in the design [10], but the approach is relatively simple. Exceptions are not simulated, and multiple interruptions and exceptions and their priorities are also not included. Missing these situations cannot simulate the real-world CPU execution and cannot cover all the CPU states.

Challenge 3: fuzzing techniques related to hardware. Fuzzing techniques are initially designed for testing software. Many customized approaches in fuzzing, such as program transformation [17] and process tracing [18], are also tailored for software programs. Especially in the critical steps of fuzzing, such as seed selection and mutation, previous fuzzing work did not combine fuzzing with hardware features well. For example, [10] did not consider seed selection when fuzzing the CPU. Not using hardware-related techniques cannot improve the fuzzing performance in testing CPU RTL.

To address the above challenges, we propose the following techniques.

Input instruction distillation based on a variant of ant colony optimization. To save the CPU cycles and improve fuzzing performance, we need to distill the inputs and shorten the input instruction length. The basic idea of input instruction distillation is to construct a subset of the original input set, which is shorter in length and can maintain the original coverage. Ant colony optimization (ACO) is one of the latest techniques for approximate optimization [19]. The algorithm simulates the routine of an ant colony to search for the shortest path to the target city. We use the idea of ACO to distill input instructions. We model the length of input instructions as the number of ants and the RTL circuits as cities. The algorithm can output the best input instruction and length for the current status, which finishes the task of input instruction distillation. Moreover, we make some changes to the classic ACO and propose a variant of ACO (VACO) to fit the RTL fuzzing scenario.

Simulating realistic interruption and exception handling. First, we include exceptions in fuzzing the CPU, which is not proposed in previous work [10]. Next, we integrate more than one interruption and exception to test the CPU, aiming to simulate the real-world execution scenario of the CPU more comprehensively. Besides, we consider the priorities of different interruptions and exceptions, which can thoroughly fuzz the CPU. The above techniques can better simulate real-world interruption and exception handling than previous work.

Hardware-related seed selection and mutation. We propose new seed selection and mutation strategies in fuzzing the CPU. In seed selection, not only basic fuzzing heuristics are taken into consideration, but also hardware heuristics, e.g., special instructions and registers. For mutation, we propose strategies closely related to hardware, such as insertion or deletion based on the input instruction length. The seed selection and mutation strategies combine fuzzing with hardware characteristics, and they overcome the drawbacks of previous tools.

We implement a prototype INSTILLER (**Inst**ruction Dist**iller**) and conduct extensive evaluation against state-of-the-art fuzzing work. In general, the results show the effectiveness of our proposed techniques. Our tool increases coverage by 29.4%. For input instruction distillation, the length of INSTILLER is 79.3% shorter than DiFuzzRTL. For vulnerability discovery, INSTILLER finds 17.0% more mismatches in the targets. In addition, the input instruction distillation leads to a 6.7% increase in execution speed on average.

In conclusion, we make the following contributions to this paper:

- We propose an input instruction distillation technique, which is based on a variant of ant colony optimization. The distillation can make the inputs shorter and more effective.
- We enable our fuzzer to handle multiple interruptions and exceptions. The priorities of them are also considered. These techniques can simulate realistic interruption and exception handling well.
- We propose hardware-based seed selection and mutation strategies. We use hardware-related heuristics and mutation operations to improve fuzzing performance in the situation of RTL fuzzing.
- We implement a prototype named INSTILLER and conduct extensive experiments. The results show that our tool outperforms previous work and demonstrate the effectiveness of our proposed approaches.

II. Background and Motivation

A. *CPU Design, Interruptions, and Exceptions*

CPU design and verification. ISA (instruction set architecture) is the basis of designing a CPU. Different types of ISA can be implemented on different processors, e.g., Intel (X86) and M1 (ARM). RTL is a real hardware design based on the specific ISA. RTL can be described in hardware description languages (HDL) such as Verilog. Most importantly, CPU RTL should be completely tested before deployment. Not like software, the bugs and vulnerabilities in hardware cannot be easily mitigated with patches. Dynamic techniques are more common in testing RTL [10], which include testing with ISA and RTL simulation. The idea of differential testing in INSTILLER is to compare the results of ISA and RTL simulation and detect hardware bugs.

Input instructions, interruptions, and exceptions. Before execution, specific instructions should be loaded into the CPU. In this paper, the inputs of CPU RTL are formed with different instructions. By interpreting every instruction, the CPU finishes executing the program. Besides normal executions, interruptions and exceptions will raise, e.g., IO interruptions and illegal access exceptions. When testing RTL, interruptions and exceptions should be simulated to thoroughly cover the corner situations. Moreover, multiple interruptions and exceptions and their priorities are common in CPU execution. For example, another high-priority interruption occurs when handling a low-priority one. Previous fuzzing work [9], [10] failed to handle multiple interruptions or exceptions and their priorities. We consider these situations in our paper.

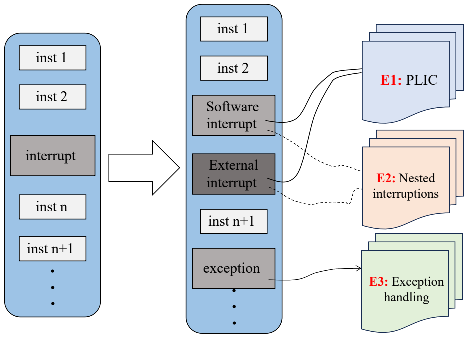


Fig. 1. Example of the effectiveness of multi-interruption and exception hardware fuzzing.

Figure 1 is a motivating example showing how a single-interruption no-exception strategy limits the effectiveness of fuzzing. This is a simplified input sequence containing instructions, two interruptions with different privilege levels, and an exception. There are mainly three aspects where the single-interruption no-exception strategy limits the effectiveness of fuzzing, i.e., coverage. First, platform level interrupt controller (PLIC) is a subsystem that can control the arbitration and distribution of multiple interrupts [20]. The code in this subsystem can only be triggered when there are multiple interruptions in the input sequence. Thus, single-interruption fuzzers fail to reach this coverage, which is denoted as *E1* in Figure 1.

Moreover, the single-interruption no-exception strategy also hinders the fuzzers from covering nested interruption scenarios, which is denoted as *E2*. Handling nested interruptions is enabled in certain IP cores [21]. Single interruption can never reach these scenarios. Therefore, enabling multiple and nested interruptions in CPU fuzzing can reach more coverage.

Third, exception handling is an important subsystem in CPU implementation [22], and there are at least 16 entries for standard exception handling. No-exception fuzzing strategy fails to cover this exception-handling code. Our proposed fuzzing strategy with exceptions in the input sequence can solve this problem, which is denoted as *E3* in Figure 1.

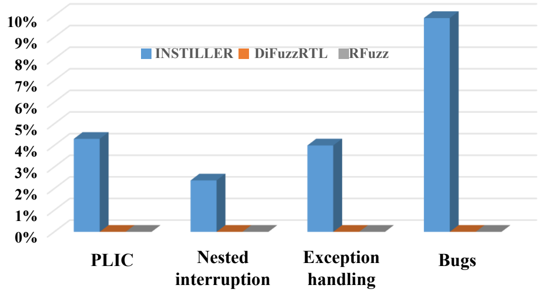


Fig. 2. Percentages of HDL line coverage and bugs of INSTILLER, DiFuzzRTL, and RFuzz related to multiple interruptions and exceptions.

Moreover, a preliminary experiment is conducted to reveal the insight of multiple interruptions and exceptions. We compare INSTILLER with state-of-the-art DiFuzzRTL and RFuzz to collect the coverage increase and bug detection related to interruptions and exceptions in Figure 2. DiFuzzRTL and RFuzz are not designed with multiple interruptions and exceptions. Therefore, their related line coverage and detected bugs are zeros. The percentages of HDL lines covered by INSTILLER are 4.3%, 2.4%, and 4.0% of the overall coverage, in PLIC, nested interruptions, and exception handling, respectively. In total, HDL line coverage related to multiple interruptions and exceptions makes up for over 10% of the coverage by INSTILLER. Besides, bugs triggered by multiple interruptions and exceptions are about 10% of all the bugs. The results in Figure 2 are **quantitative examples** of the motivation in Figure 1.

Therefore, we believe enabling multiple interruptions and exceptions with their priorities is innovative compared with other fuzzers in two aspects. First, the fuzzing strategy in [9], [10] is immature without multiple interruptions and exceptions with their priorities, and they deviate significantly from the actual CPU execution scenario. This is the key difference between our proposed “REALISTIC” strategy and theirs. Second, we have invested sufficient effort in code implementation in this part. For instance, the insertion of multiple interruptions with privilege levels requires us to deal with the *mcause* register and other issues. Therefore, our strategy is creative compared with previous work.

B. Fuzzing and Differential Testing

Fuzzing or fuzz testing [8] is one of the most successful software testing techniques. Coverage-guided grey-box fuzzing (CGF) is a variant of fuzzing, and it is famous for its great balance between effectiveness and efficiency [23]. The basic steps of fuzzing include:

- 1) Given initial seeds;
- 2) Select a seed and mutate the seed to generate input instructions;
- 3) Execute the target program with the instructions;
- 4) If there is new coverage, save the input instruction to the seed pool; If there is a program crash, report and save this crash;
- 5) Continuing to step 2).

Differential testing compares the results of two or more systems, and different results indicate bugs and vulnerabilities [24]. Based on it, differential fuzz testing is proposed to solve more complex bug-discovery problems [10].

In CPU verification, comparing the results of RTL executions with that of a golden model (ISA) is an effective testing technique [10], [25]. Therefore, our work combines the idea of CGF and differential testing to boost the CPU verification process.

C. Ant Colony Optimization

Ant colony optimization was first introduced in the early 1990’s [19]. It is one of the latest techniques for approximate optimization. ACO is used to solve combinatorial optimization problems such as the traveling salesman problem (TSP), which is an optimization algorithm simulating ant foraging behavior. The basic procedure of ACO can be concluded as:

- 1) Initialize the parameters;
- 2) Calculate the probability for every city and every ant;
- 3) Select the best next city to walk for every ant;
- 4) Update the pheromone table after the ants finish walking;
- 5) Continue to step 2) before termination.

In this paper, we use the idea of ACO to distill input instructions. However, classic ACO does not fit RTL testing. The number of ants in ACO is constant, and it is variable in our model. Moreover, in classic ACO, the algorithm chooses the next best city for an ant. In our work, executing an input instruction will cover multiple circuits (cities), and it is choosing the next best cities. Therefore, we propose a variant of classic ACO to handle the above problems.

III. DESIGN

A. Overview

Figure 3 is the overview of INSTILLER. There are mainly three newly-designed infrastructures, including the VACO algorithm, interruption and exception simulation, and seed selection with mutation. First, VACO is capable of distilling the input instructions in RTL fuzzing, which can keep the input short and effective. Next, realistic interruption and exception handling is simulated by our simulation process. Through this kind of simulation, our fuzzing process is closer to the real-world execution of the CPU. In addition, the seed selection and mutation strategies integrate hardware-related features into fuzzing and improve the fuzzing performance.

Algorithm 1 Overview of the procedures of INSTILLER.

```
Require: Initial seeds $S$
while $t < TIME\_OUT$ do
    if $start\_distill == True$ then
        $re = relation\_extract()$
        $len = VACO(re)$
    end if
    $s = seed\_selection(seed)$
    $s' = mutation(s, len)$
    $input = interrupt\_exception(s')$
    $O_I = ISA\_sim(input)$
    $O_R = RTL\_sim(input)$
    $Cross\_check(O_I, O_R)$
end while
Ensure: Bug reports
```

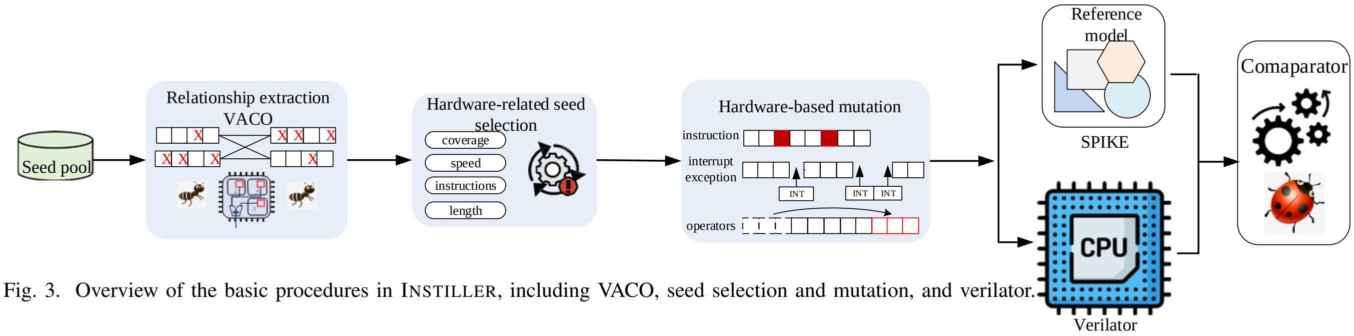


Fig. 3. Overview of the basic procedures in INSTILLER, including VACO, seed selection and mutation, and verilator.

The detailed execution process of INSTILLER is shown in Algorithm 1. Given initial seeds, the fuzzing process is started. Depending on the current coverage status, INSTILLER decides whether the input instruction distillation should be started. Distillation includes relationship extraction and the VACO algorithm. The output of distillation is the most effective input and its length for the current fuzzing status. After seed selection and mutation, the input instructions are inserted with multiple interruptions and exceptions, which are ready for execution. ISA simulation and RTL simulation will be executed, and their results are cross-checked to output bug reports. In general, Figure 4 shows the fuzzing procedure of INSTILLER, and the colored parts are the modification to the basic fuzzing process.

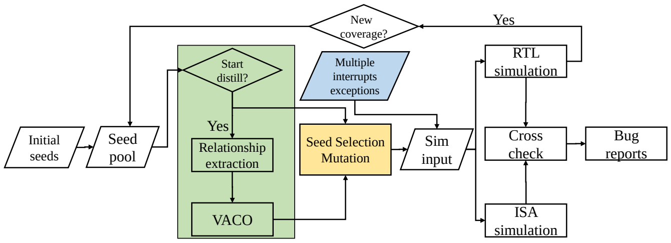


Fig. 4. Fuzzing procedures of INSTILLER, where the colored parts are newly proposed mechanism compared with traditional fuzzing.

B. Input instruction Distillation Based on VACO

In state-of-the-art RTL fuzzing work [10], the length of the input instruction keeps increasing as the fuzzing continues. Long input instructions slow down the fuzzing process and are unfriendly for fuzzing. Therefore, we propose input instruction distillation to keep the inputs short and effective. The distillation includes relationship extraction and the VACO algorithm.

Relationship extraction. In the execution of the CPU, the complex operations are finished by some related instructions and they should be treated as a group. For example, in Figure 5, the three instructions complete an ADD operation. Therefore, it is rational to extract the relationships between instructions. According to our preliminary study [1], the relationships between RTL input instructions can be divided into software relationships and hardware relationships. If there are relationships between instructions, we collect them to form instruction groups. These groups are then used in the VACO algorithm.

```
① la xreg1, symbol                 // load the address to xreg1
② addi xreg1, xreg1, imm6          // xreg1 add imm, store to xreg1
③ amoadd.w xreg0, xreg2, (xreg1) // mem[xreg1] add mem[xreg2], store to mem[xreg0]
```

Fig. 5. Example of the relationships between instructions, where the three instructions all operate on the same register.

Algorithm 2 The relationship extraction algorithm.

```
Require: Software inputs $W_s$, hardware inputs $W_h$, coverage $cov$
$W = sort(W_s, cov)$
$W_D = \emptyset$ /* instruction groups */
for $i \rightarrow W$ do
    for $j \rightarrow W$ do
        if $register_i == register_j$ then
            $W_D = W_D + (i, j)$
        end if
        if $target(i) == j$ then
            $W_D = W_D + (i, j)$
        end if
    end for
end for
for $i \rightarrow W_h$ do
    if $i \Rightarrow clock, iterrupt, privilege, register$ then
        $W_D = W_D + i$
    end if
end for
Ensure: Instruction groups $W_D$
```

Software relationships include data-flow and control-flow relationships. First, we sort all the executed input instructions with coverage. Then, beginning from the input instructions with the most coverage, the instructions are traversed. If two instructions share the same registers, there is a data-flow relationship between them. If one input is the jump target of another, there is a control-flow relationship between them. Instructions that have software relationships are collected to form groups.

Hardware relationships include clock cycles, interruptions, privilege levels, and special registers. For example, if inserting an interruption after an instruction can change the privilege level (priority), we consider there is a hardware relationship. Another example is the non-aligned load and store addresses are exceptional. When the hart time comparator (a memory mapping register named `mtimecmp`) is larger than the real-time counter `mtime`, the clock interruption will be triggered. These instructions together with the hardware events are collected to form groups.

Algorithm 2 shows the process of the relationship extraction procedure. This approach outputs the instruction groups to VACO.

The VACO algorithm. As we discussed above, by simulating the behaviors of the ant colony, ACO is a classic optimization algorithm to find the shortest path between cities.

[1] This study is conducted by investigating the RISC-V Instruction Set Manual Volume I and II [21], [22].

This article has been accepted for publication in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. This is the author's version which has not been fully edit
content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2024.3360318

It is an iterative algorithm, and it outputs the best solution when the iteration is done. To adopt ACO in our scenario, we need to solve three problems: 1) When to start the algorithm? 2) How to model the factors in fuzzing into the algorithm? 3) When to stop the process?

1) The algorithm is invoked when there is a continuous average coverage decrease. Average coverage is coverage divided by the input instruction length. When this indicator decreases, the input instruction length is too long, and the input is not effective in finding new coverage.

2) We model the length of the input instructions as the number of ants, and the RTL circuits as the cities in ACO. The scale of RTL [2] (number of cities) is n, and the current length of input instruction (number of ants) is m. In each iteration, for every ant i and every path j, we calculate the pheromone table as:

$$pher_j = (1-\rho) * pher_j + \sum_{i=1}^{m} \Delta pher_j^i \tag{1}$$

In Equation 1, ρ means the evaporation rate of pheromones, which is a tunable parameter. The definition of $\Delta pher_j^i$ is:

$$\Delta pher_j^i = \begin{cases} \frac{1}{length_i} & ant\ i\ traverses\ path\ j \\ 0 & otherwise \end{cases} \tag{2}$$

Then, a probability table is calculated by:

$$p_j^i = \frac{pher_j * h_j}{\sum\limits_{k\, not\, traversed\, by\, i}^{n} pher_k * h_k} \tag{3}$$

This equation uses coverage as heuristics (h_j), which is proportional to the coverage of path j. This is where we use the instruction groups. Every instruction group stands for an ant. Executing the instructions in a group is the process of an ant walking through the cities. The best candidate for the next ant is selected with probability table p_j^i. Therefore, by appending the best group in this iteration to the current input instructions, the length of input will increase by one after each iteration of the algorithm. When the algorithm is terminated, the length of the input instructions is the most effective length for the current fuzzing status.

3) The algorithm will iterate until there is an average coverage increase compared with that before starting the algorithm.

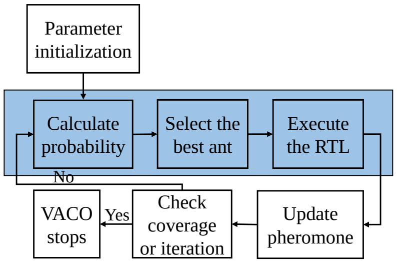


Fig. 6. Basic workflow of VACO, where the colored parts are newly designed compared with ACO.

As Figure 6 shows, the basic workflow is an iterative algorithm. Our proposed algorithm is a variant of the classic ACO. The colored parts in the figure are the major differences between VACO and ACO: 1) The number of ants in ACO is constant, while it is variable in VACO. When the algorithm is terminated, the generated length is the best input instruction length. 2) In VACO, the probability calculation is based on the coverage of the fuzzing process. However, the transition probability in ACO is related to the distance between cities. 3) The classic ACO does not contain the process of executing the ant in the RTL. Therefore, introducing the execution into VACO inevitably causes performance overhead. We integrate the RTL execution in VACO into the main fuzzing loop to reduce the performance overhead.

Algorithm 3 The VACO algorithm.

```
Require: Number of iterations $Max\_iter$, evaporation rate $\rho$, RTL scale $n$, input
    length $m$, group $W_D$
 1: while $iteration < Max\_iter$ do
 2:     $pher[: n] = [1...1]$
 3:     $heuristics[: iteration] = coverage$
 4:     $p = Equation\,3$
 5:     $id = roulette\_wheel\_selection(p)$
 6:     $coverage' = exe(id)$ /* Executing this input instruction results in coverage'
    */
 7:     if $average\ coverage\ increased$ then
 8:         $break$
 9:     end if
10:     $pher = (1-\rho) * pher + \Sigma\Delta pher$
11: end while
12: $m = m + len(id)$
Ensure: The most effective input instruction length $m$
```

In conclusion, Algorithm 3 shows the process of the VACO algorithm. By receiving the parameters, VACO iterates and finally outputs the best input instruction and length in the current fuzzing status. The length is shorter and the input instruction is more effective in finding new coverage. Therefore, the input distillation is finished. In Line 5, roulette wheel is a classic selection algorithm [26].

C. Realistic Interruption and Exception Handling

First, previous work has no exception inserted in the input instructions. We include exceptions in the RTL inputs to solve this problem. Next, in real-world applications, multiple interruptions and exceptions are common during CPU execution. We consider this situation in RTL testing and enable multiple interruptions and exceptions together with their respective handling strategies. In our design, multiple interruptions and exceptions are inserted in the input instructions for simulation. Every interruption and exception is analyzed to extract the relative information, e.g., the address of the interruption and the cause of the exception.

Figure 1 shows the difference between multiple interruptions and exceptions and single ones. Several instructions `inst n` constitute a complete input. Interruptions and exceptions can be inserted between the instructions. The input on the right with multiple interruptions or exceptions is the one in INSTILLER. This input can better simulate real-world CPU execution.

Moreover, interruptions and exceptions have specific priorities. For example, in RISC-V, the priority of interruptions in machine mode is higher than that in supervisor mode [22]. We consider different interruptions and exceptions with different

[2] In this paper, we use the coverage definition of DiFuzzRTL. Therefore, the scale of RTL means the number of control registers [10].

priorities in fuzzing. In a situation where interruptions and exceptions with higher priority are triggered when a lower-priority one is being handled, more RTL states can be covered than testing without priorities.

In conclusion, in the process of fuzzing the RTL, we include both multiple interruptions and exceptions along with their priorities to test the target RTL more thoroughly.

D. Seed Selection

Previous work in RTL fuzzing ignores the importance of seed selection and hardware features in improving the performance of fuzzing. In seed selection, we focus on the normalized heuristics of input instructions to make decisions. Normalized heuristics means the heuristics score of an input divided by its length. The reason for using normalized heuristics is that one important perspective of our work is to distill seeds and generate shorter input instructions. A higher normalized heuristics score indicates the input instruction is more potential for fuzzing, and its length is relatively short at the same time.

In INSTILLER, we use the following heuristics to select seeds. 1) Basic heuristics. It includes coverage increase (`cov`) and execution speed (`speed`). These metrics of heuristics are commonly seen in fuzzing tools [27]–[29]. We use these metrics to score seeds in this paper. Specifically, coverage is the most important metric in coverage-guided fuzzing. Therefore, it is used in our tool. Then, execution speed is also a crucial factor in fuzzing [11]–[16], and we integrate this metric in our heuristics.

2) RTL heuristics. This category contains metrics related to RTL hardware, including the number of load or store instructions (`ld_st`), floating point instructions (`fp`), and jump instructions (`jp`). According to our study of recent bugs in real-world cores [30], [31], e.g., Boom and Rocket, we find out that these instructions are the cause of multiple bugs. Therefore, we use these metrics in scoring the input instructions.

In conclusion, our heuristics calculation can be summarized as

$$heuristics = \frac{\omega * cov * speed + ld\_st * fp * jp}{len} \tag{4}$$

In this equation, ω is the proportion of basic metrics in the heuristics, which controls the weight between basic and RTL metrics. Based on this heuristics, we select input instructions with the highest score in the seed pool.

E. Mutation

In the design of DiFuzzRTL, the mutator can only add instructions to the inputs in mutation [10]. In other words, the mutation operation will keep the length of input instructions increasing. However, in common fuzzing tools such as AFL, it is widely acknowledged that fuzzers should be equipped with various effective mutation strategies [23], [32], [33], e.g., “dictionary” (replacing part of the input instructions with tokens[3]) and “splice” (splicing two inputs to generate one) in AFL.

We use several mutation strategies in INSTILLER to improve fuzzing. 1) The mutation strategy of DiFuzzRTL. The paper indicates its strategy is effective in fuzzing [10]. Therefore, we retain it in INSTILLER. 2) Dictionary. In input instruction distillation, we get the distilled inputs, in which we can extract dictionary tokens to guide mutation. These tokens are effective in fuzzing and short in length. By replacing part of the original input instruction with these tokens, this “dictionary” mutation strategy is completed. 3) Insertion. We randomly insert new instructions to the inputs. 4) Deletion. Part of the input instruction is deleted, and the length of the input will decrease.

These four types of mutation strategies are similar to the mutation in binary fuzzing, but they are different. The input instruction of binary fuzzing can be encoded to sequences of “1”s and “0”s. The mutation on the sequence can be treated as mutating a simple string to enumerate all cases. Therefore, “bitflip” (flipping “1” to “0”, or “0” to “1”) in AFL is highly effective in discovering new paths [32]. However, if there are illegal instructions, randomly mutating the input instructions of RTL can be often meaningless. Therefore, we use several techniques in the design of mutation strategies. For example, when inserting instructions, we use previously-used ones with higher probability and newly-generated ones with lower probability. Another example is in deletion, when deleting one instruction, the related instruction will also be deleted, e.g., a jump instruction and its target.

Furthermore, these four types of mutation in INSTILLER are invoked in different situations. They are chosen with different probability, which is calculated by weighted metrics. Every strategy is chosen with a different probability. If the condition is satisfied, the respective mutation strategy is selected. Otherwise, INSTILLER uses the original mutation. This can be concluded as

$$mutation = \begin{cases} \texttt{dictionary} & coverage\ decreases \\ \texttt{insertion} & len < l \\ \texttt{deletion} & len > l \\ \texttt{basic} & otherwise \end{cases} \tag{5}$$

IV. IMPLEMENTATION

We implement INSTILLER based on DiFuzzRTL. In general, the RTL instrumentation is finished with FIRRTL . We use Spike for ISA simulation, and cocotb for RTL simulation. The main fuzzing loop is implemented in Python.

In total, the implementation of INSTILLER can be divided into three parts. 1) The input instruction distillation process, which includes relationship extraction and VACO. We add this part in `Fuzzer.py`, and there are about 500 lines of code in total. Additionally, we modify the main fuzzing loop to interact with VACO, including the starting and terminating conditions. 2) Interruptions and exceptions. We integrate multiple interruptions and exceptions into INSTILLER, as well as their priorities. `signature_checker.py` is modified, where we focus on the special registers for the interruption and exception handling, e.g., `scause` and `mcause`. This part contains about 550 lines of code. 3) Seed selection and mutation strategies.

[3] Token means an item in the dictionary.

We implement the heuristics calculation and four mutation strategies in `mutator.py`, which include about 700 lines of code. Both seed selection and mutation are integrated into the original process, which will cause no additional steps.

V. Evaluation

In our evaluation, we answer the following research questions:

- **RQ1.** Can INSTILLER increase code coverage and shorten input instruction length?
- **RQ2.** Does INSTILLER have better vulnerability discovery ability than state-of-the-art RTL fuzzers?
- **RQ3.** What is the performance increase in execution speed of INSTILLER?
- **RQ4.** How does the VACO algorithm perform?
- **RQ5.** How do the techniques on interruptions and exceptions perform?
- **RQ6.** How do the seed selection and mutation strategies perform?

A. Setup

TABLE I
Information of the target CPUs, including ISA, number of pipelines, instruction width, and release time

Targets	ISA	Pipeline	Width	Year
mor1kx	OpenRISC	6-stage	32-bit	2013
or1200	OpenRISC	5-stage	32-bit	2000
Boom	RISC-V	4-stage	32-bit	2017
Rocket	RISC-V	5-stage	32-bit	2016

Targets. In our experiments, the target RTL designs include mor1kx [34], or1200 [35], Boom [30], and Rocket [31]. These are popular RTL cores, and state-of-the-art papers used them in the experiments [10], [25]. Therefore, including these RTL designs in our evaluation demonstrates persuasiveness and representativeness. The detailed information is listed in Table I.

Compared tools. To demonstrate the performance of INSTILLER, we compare it with DiFuzzRTL. DiFuzzRTL is one of the state-of-the-art CPU testing tools. We will compare INSTILLER with DiFuzzRTL in different aspects such as coverage[4].

Metrics. We use coverage, the length of input instructions, the number of mismatches, and execution speed as the metrics in our evaluation. For every target CPU, we repeat the 24-hour fuzzing 10 times. In addition, we calculate the p values and $\hat{A}_{12}$ values of all the experiment results to eliminate the effect of randomness in fuzzing [36]. If the $p < 0.05$ and $\hat{A}_{12} > 0.5$, then this specific comparison shows a statistically significant difference.

B. Evaluation on Coverage

Table II is the coverage results of INSTILLER and DiFuzzRTL. According to the table, INSTILLER outperforms DiFuzzRTL in all the targets, and all the coverage increase is more than 11%. In all the comparisons with DiFuzzRTL, the p values are less than 0.05, and the $\hat{A}_{12}$ values are greater than 0.5. The results indicate statistically significant differences. In the comparison in or1200, INSTILLER has 36.5% more coverage than DiFuzzRTL, which is the greatest difference among all the results. On average, INSTILLER reaches 29.4% more coverage, and this result also has a statistically significant difference.

TABLE II
Evaluation on coverage of Instiller and DiFuzzRTL, where the value in the bracket denotes the increase or decrease compared with the competitors

Targets	INSTILLER	DiFuzzRTL	p value	$\hat{A}_{12}$
mor1kx	201288.7(+32.6%)	151840.5	$5.11*10^{-4}$	1.0
or1200	269504.8(**+36.5%**)	197503.9	$6.84*10^{-5}$	1.0
Boom	547433.5(+28.8%)	425183.7	$9.13*10^{-5}$	1.0
Rocket	101489.6(+11.9%)	90715.3	$8.98*10^{-5}$	1.0
Average	279929.15(+29.4%)	216310.85	$1.90*10^{-4}$	1.0

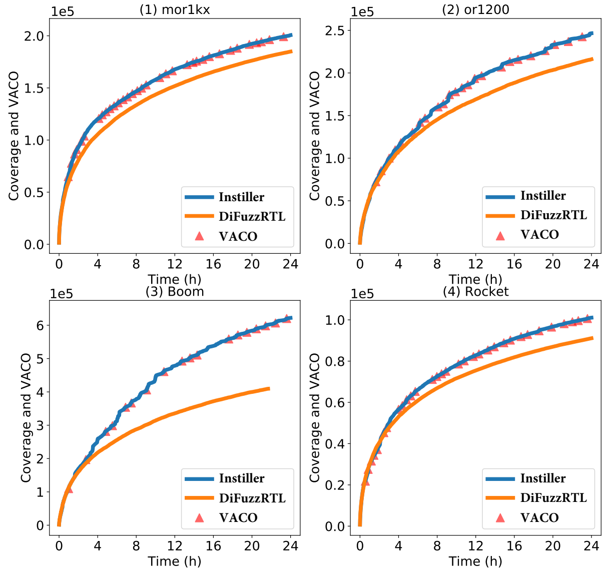


Fig. 7. Coverage and the effectiveness of VACO over time, where the X-axis is the time, and the Y-axis is the coverage.

Figure 7 shows the coverage growing over time of INSTILLER and DiFuzzRTL. We record the results within 24 hours of fuzzing. According to the figure, in all the targets, the coverage of INSTILLER is greater, and the growth of coverage in INSTILLER is faster than DiFuzzRTL.

The results in Table II and Figure 7 demonstrate the effectiveness of input instruction distillation. The internal reason is that we distill inputs based on the coverage performance of input instructions, e.g., Line 1 in Algorithm 2 sorts the inputs based on coverage.

In conclusion, according to these experiment results, INSTILLER has better coverage exploration ability than DiFuzzRTL.

C. Evaluation on Input Instruction Length

Table III shows the input instruction length during fuzzing of INSTILLER and DiFuzzRTL, respectively. In all the target

[4] The source code of TheHuzz [25] is not available. Therefore, it is not evaluated in this paper.

TABLE III
EVALUATION ON INPUT INSTRUCTION LENGTH OF INSTILLER AND DIFUZZRTL, WHERE THE VALUE IN THE BRACKET DENOTES THE INCREASE OR DECREASE COMPARED WITH THE COMPETITORS

Targets	INSTILLER	DiFuzzRTL	p value	$\hat{A}_{12}$
mor1kx	451.39(-77.6%)	2018.21	$9.56*10^{-5}$	1.0
or1200	385.63(-80.6%)	1986.70	$7.55*10^{-5}$	1.0
Boom	486.75(-74.1%)	1882.80	$9.13*10^{-5}$	1.0
Rocket	420.67(**-83.6%**)	2557.67	$8.98*10^{-5}$	1.0
Average	436.11(-79.3%)	2111.34	$8.80*10^{-5}$	1.0

TABLE IV
NUMBER OF MISMATCHES OF INSTILLER AND DIFUZZRTL, WHERE THE VALUE IN THE BRACKET DENOTES THE INCREASE OR DECREASE COMPARED WITH THE COMPETITORS

Targets	INSTILLER	DiFuzzRTL	p value	$\hat{A}_{12}$
mor1kx	110.1(-3.9%)	120.9	0.98	0.2
or1200	598.3(+6.7%)	560.9	0.001	1.0
Boom	5546.0(**+18.8%**)	4666.6	0.01	1.0
Rocket	33.3(+14.0%)	29.2	0.06	0.7
Average	1573.9(+17.0%)	1344.4	0.26	0.73

CPU cores, INSTILLER shortens the length of input instructions, and all the decrease is more than 74%. Especially in the Rocket core, the decrease of length reaches the maximum of 83.6%. In addition, all the comparisons with DiFuzzRTL have statistically significant differences. The average input instruction length of INSTILLER is 79.3% shorter than DiFuzzRTL. These results indicate the effectiveness of VACO, which significantly shortens the input instruction length.

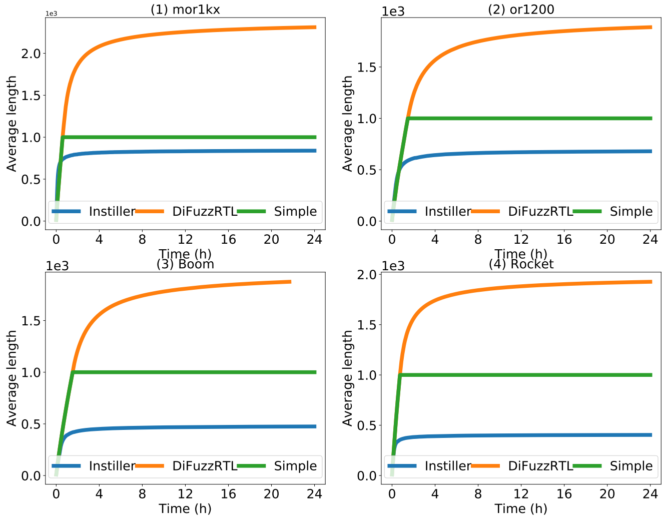


Fig. 8. Input instruction length over time, where the X-axis is the time, and the Y-axis is the average length.

Figure 8 is the result of input instruction length over time. In the 24 hours, compared with DiFuzzRTL, INSTILLER shortens the length.

The results in this section show a significant difference between input instruction length in INSTILLER and DiFuzzRTL. Decreasing the input length is the main focus of this work, and the mechanism in VACO contributes to the results. Moreover, the results of DiFuzzRTL in Table II and Figure 8 demonstrate the statement in Section I that coverage does not increase proportionally to input length.

- Therefore, based on the results of coverage and input instruction length, we can answer RQ1: INSTILLER *can increase code coverage and shorten input instruction length at the same time.*

D. Evaluation on Vulnerability Detection

Table IV shows the number of mismatches of differential testing. In the design of INSTILLER and DiFuzzRTL, if the output of ISA simulation is different from RTL execution, i.e., a mismatch, a potential bug is detected. It is rational to use the number of mismatches to demonstrate the vulnerability detection ability of the fuzzing tools. In the table, except for mor1kx, INSTILLER outperforms DiFuzzRTL in all the targets. The greatest improvement is in Boom core, with an increase of 18.8%. On average, INSTILLER also outperforms DiFuzzRTL in vulnerability discovery, and it detects 16.9% more mismatches.

Furthermore, we manually investigate why DiFuzzRTL outperforms INSTILLER in mor1kx. In Table II, INSTILLER has more coverage. Our investigation shows the fuzzers cover different parts of mor1kx. Moreover, by looking into the code of mor1kx, we find out that the hardware circuits covered by INSTILLER cannot trigger as many mismatches as DiFuzzRTL. These factors lead to the results in Table IV.

- Therefore, we can answer RQ2: INSTILLER *has better vulnerability detection ability than DiFuzzRTL.*

E. Evaluation on Execution Speed

TABLE V
EXECUTION SPEED PER SECOND OF INSTILLER AND DIFUZZRTL, WHERE THE VALUES IN THE BRACKET DENOTE THE INCREASE OR DECREASE COMPARED WITH THE COMPETITORS

Targets	INSTILLER	DiFuzzRTL	p value	$\hat{A}_{12}$
mor1kx	0.27(+8.0%)	0.25	0.99	0.15
or1200	0.40(+5.3%)	0.38	0.98	0.21
Boom	0.25(+4.1%)	0.24	0.99	0.10
Rocket	0.36(**+9.1%**)	0.33	0.99	0.20
Average	0.32(+6.7%)	0.30	0.99	0.17

Table V shows the execution speed of INSTILLER and DiFuzzRTL. Execution speed is the result of dividing the number of executions by the time. In general, INSTILLER runs faster than DiFuzzRTL. The greatest difference is in Rocket, which is 9.1%. On average, the performance increase in INSTILLER is 6.7%, which shows the effectiveness of distilled input instructions. Shorter inputs require fewer CPU cycles to execute.

In Section V-C, the input instruction length of INSTILLER is 79.3% shorter on average. Intuitively, the performance increase should have been more than 6.7%. We investigate the cause of this result. First, there is no direct correspondence between the input length and execution speed of fuzzing. Having 79.3% short length does not mean executing 79.3% faster. Second, there is a performance overhead in the process of relationship extraction and VACO. The relationship extraction

This article has been accepted for publication in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. This is the author's version which has not been fully edi
content may change prior to final publication. Citation information: DOI 10.1109/TCAD.2024.3360318

enumerates all the executed input instructions, and VACO is also an iterative process. Therefore, the performance overhead is unavoidable to finish these processes.

TABLE VI
PERFORMANCE OVERHEAD OF RELATIONSHIP EXTRACTION AND VACO, WHERE THE VALUES IN THE BRACKET DENOTE THE INCREASE OR DECREASE COMPARED WITH THE COMPETITORS

	INSTILLER	INSTILLER^{-R}	INSTILLER^{-V}	INSTILLER^{-RV}
Speed	0.32	0.34(+6.3%)	0.29(-9.4%)	0.30(-6.3%)

[1] INSTILLER^{-R} denotes INSTILLER without relationship extraction.
[2] INSTILLER^{-V} denotes INSTILLER without VACO.
[3] INSTILLER^{-RV} denotes INSTILLER without relationship extraction and VACO.

In addition, we conduct extra experiments to show their specific overhead. Table VI shows the respective average speed of four different configurations of INSTILLER. When relationship extraction is disabled, the fuzzer can run 6.3% faster than the original INSTILLER. However, investigating coverage data shows that this configuration has less coverage than INSTILLER. Therefore, disabling relationship extraction has an effect on speed and coverage. If there is no VACO in INSTILLER, the execution speed decreases by 9.4%. The reason is that disabling VACO means input distillation is disabled, which cannot utilize shorter input instructions. Besides, the execution of relationship extraction in INSTILLER^{-V} has overhead. The average speed of INSTILLER^{-RV} is the same as DiFuzzRTL in Table V, which means other approaches of INSTILLER, e.g., seed selection and mutation, cause negligible performance overhead.

- We can answer RQ3: *The performance increase of* INSTILLER *is 6.7% on average.*

F. Results of VACO

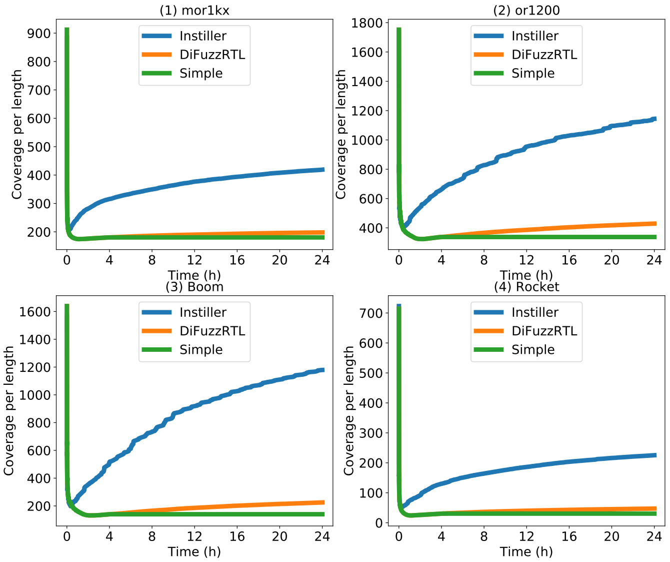


Fig. 9. Coverage divided by input instruction length over time, where the X-axis denotes time, and the Y-axis denotes coverage divided by length.

Figure 9 shows the results of coverage divided by input instruction length over time of INSTILLER and DiFuzzRTL. Both of the lines experience a high start in the figure. The reason is that the first input instruction length is always "1", and executing the first input brings about a 700 to 1800 coverage increase. Regardless of the high start, INSTILLER has higher coverage per length than DiFuzzRTL. This result indicates the input instructions in INSTILLER are more effective in discovering new coverage. The mechanisms of relationship extraction and VACO ensure this result.

Figure 7 shows the coverage results of INSTILLER and DiFuzzRTL, and how VACO affects coverage in the 24 hours. The red circles in the figure indicate the enabling of VACO. Every time VACO is enabled, the coverage increases more rapidly than DiFuzzRTL.

TABLE VII
RESULTS OF COVERAGE AND INPUT INSTRUCTION LENGTH OF DIFFERENT CONFIGURATIONS OF INSTILLER, WHERE THE VALUES IN THE BRACKET DENOTE THE INCREASE OR DECREASE COMPARED WITH THE COMPETITORS

	INSTILLER	INSTILLER^{-R}	INSTILLER^{-V}	INSTILLER^{-RV}
Coverage	279929.2	248663.5(-11.2%)	255439.2(-8.7%)	217197.5(-22.4%)
Length	436.1	498.4(+14.3%)	2185.9(+401.2%)	2254.8(+417.0%)

[1] INSTILLER^{-R} denotes INSTILLER without relationship extraction.
[2] INSTILLER^{-V} denotes INSTILLER without VACO.
[3] INSTILLER^{-RV} denotes INSTILLER without relationship extraction and VACO.

Table VII shows the results of coverage and input instruction length of four different configurations of INSTILLER. For coverage, disabling relationship extraction and VACO will have a negative effect. With a 22.4% decrease, INSTILLER^{-RV} has the least coverage, which is almost the same as the result of DiFuzzRTL in Table II. For input instruction length, the VACO algorithm has the most effect. Disabling VACO alone causes a 417.0% length increase, which is also close to the result of DiFuzzRTL in Table III.

Besides, we compare VACO with a simpler method which sets an upper limit for the length and discards seeds exceeding this limit [5]. The results are shown in Figure 8 and Figure 9. The upper limit is set to 1,000 in our evaluation. Figure 8 denotes the average length over time of this simple strategy. In this figure, the average length of Simple grows as the fuzzing process continues. When the length reaches the limit, it stays unchanged until the end of fuzzing.

Moreover, as shown in Figure 9, the result of Simple experiences a high start in the beginning, which is similar to INSTILLER and DiFuzzRTL. However, as the fuzzing process continues, the coverage per length of Simple stays at a low value compared with INSTILLER, and the value remains unchanged several hours after the start of fuzzing. Though simply setting an upper limit for the length and discarding seeds exceeding this limit can control the length of the input sequence, there is no benefit in improving coverage. Our VACO algorithm surpasses the basic upper-bound-limiting strategy by controlling the input length and increasing coverage at the same time.

Furthermore, we conduct experiments comparing the bug-finding performance between INSTILLER and INSTILLER without VACO (denoted as INSTILLER^{-V}). As shown in Table

[5] "Simple" strategy is the version of INSTILLER that replaces VACO with an upper-bound-limiting strategy.

VIII, Instiller outperforms Instiller^{-V} in all the target CPUs. On average, Instiller has 12.5% more mismatches, demonstrating better vulnerability discovery ability of Instiller. The reason behind these results is mainly due to the coverage of the fuzzing process. As shown in Table VII, Instiller^{-V} has 8.7% less coverage than Instiller. Covering more code ensures Instiller to find more bugs.

TABLE VIII
Number of mismatches of Instiller and Instiller without VACO (denoted as Instiller^{-V}), where the values in the bracket denote the increase or decrease compared with the competitors

Targets	Instiller	Instiller^{-V}	p value	$\hat{A}_{12}$
mor1kx	110.1(+7.4%)	102.5	0.04	0.7
or1200	598.3(+6.5%)	561.7	0.002	1.0
Boom	5546.0(**+13.3%**)	4897.1	0.01	1.0
Rocket	33.3(+10.3%)	30.2	0.045	1.0
Average	1573.9(+12.5%)	1397.9	0.022	0.93

- Therefore, we can answer RQ4: *Experiment results show the effectiveness of VACO, together with relationship extraction, which can increase code coverage, shorten input instruction length, and find more bugs at the same time.*

G. Results of Multiple Interruptions and Exceptions

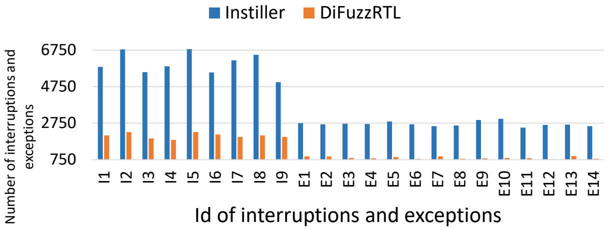


Fig. 10. Number of interruptions and exceptions of Instiller and DiFuzzRTL, where the X-axis denotes ID, and the Y-axis denotes the number.

We collect 9 interruptions (*I1 - I9*) and 14 exceptions (*E1 - E14*) from [22]. In this part of the evaluation, we insert interruptions and exceptions into the input instructions with 50% probability. Moreover, we set the limit of interrupts and exceptions to three, respectively, to show the effectiveness of multiple ones. However, this does not mean Instiller can only support three interruptions or exceptions. The limit can be configured as needed.

Figure 10 shows the number of interruptions and exceptions of Instiller and DiFuzzRTL. The number of interruptions and exceptions of Instiller are more than DiFuzzRTL. Multiple ones are inserted into the input instructions, and the chance of triggering new states and bugs is higher in Instiller.

In addition, we compare Instiller with DiFuzzRTL by considering the priorities of interruptions and exceptions. In the design of RISC-V, the interruptions and exceptions have fixed priorities. For example, “I2” is higher than “I0”, and “E3” is higher than “E0”. Therefore, different combinations of them indicate different combinations of priorities. In this part, we define *interruption state transition (IST)* and *exception state transition (EST)* to describe different combinations of interruptions and exceptions. For example, there are “I3” and “I7”, and “E2” in an input instruction. The IST of this input instruction is 1 ((3 $\ll$ 1) XOR 7), and EST is 4 ((2 $\ll$ 1) XOR 0). Here, we use $(ID_1 \ll 1)$ XOR $ID_2)$ as a hash method to represent IST or EST. In our definition, the more different values of IST or EST, the more different states a fuzzer can reach.

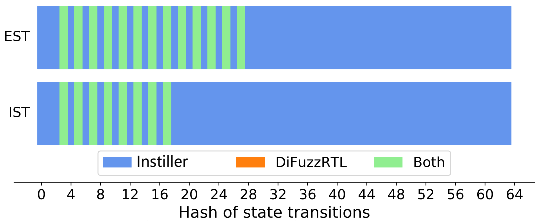


Fig. 11. Distribution of IST and EST of Instiller and DiFuzzRTL.

Figure 11 shows the distribution of state transitions of interruptions and exceptions. The IST of Instiller ranges from 0 to 64, while that of DiFuzzRTL ranges from 2 to 18. The EST result is similar. The states Instiller can reach are more than DiFuzzRTL. This result indicates that by considering multiple interruptions and exceptions with their combinations, Instiller can trigger more states in fuzzing, and therefore, it has a higher chance to discover bugs. Note that in the figure, the result of “DiFuzzRTL” is covered by “Both”, so we cannot see the result of it.

TABLE IX
Number of mismatches of Instiller and Instiller without multiple interruptions and exceptions (denoted as Instiller^{-IE}), where the values in the bracket denote the increase or decrease compared with the competitors

Targets	Instiller	Instiller^{-IE}	p value	$\hat{A}_{12}$
mor1kx	110.1(+2.8%)	107.1	0.015	0.85
or1200	598.3(+3.7%)	577.2	0.01	0.9
Boom	5546.0(+2.5%)	5411.8	0.03	1.0
Rocket	33.3(**+7.1%**)	31.1	0.02	1.0
Average	1573.9(+2.6%)	1531.8	0.019	0.94

Moreover, Table IX compares the bug-finding performance between Instiller and Instiller without multiple interruptions and exceptions (denoted as Instiller^{-IE}). As the table illustrates, Instiller surpasses Instiller^{-IE} in all the CPUs, reaching up to 7.1% in Rocket. On average, Instiller also outperforms the competitor by 2.6%. Besides, all the p values are less than 0.05, and all the $\hat{A}_{12}$ values are greater than 0.5, indicating all the results have significant differences.

We investigate the source code and find out that some mismatches in Table IX reside in the PLIC, nested interruption handling, and exception handling of CPU implementations, as listed in Section II-A and Figure 1. For example, we witness a store page fault exception handling in Rocket core using Instiller, while we cannot reproduce it with other fuzzers or Instiller^{-IE}. This result verifies our investigation in Section II-A that enabling multiple interruptions and exceptions is effective to CPU fuzzing.

- Therefore, we can answer RQ5: *The techniques of multiple interruptions and exceptions are effective.*

H. Effectiveness of Seed Selection and Mutation

Table X shows the results of coverage and length of different configurations of Instiller. By using Equation 4, the fuzzing

TABLE X
COVERAGE AND LENGTH COMPARISON OF INSTILLER WITH OTHER CONFIGURATIONS, WHERE THE VALUES IN THE BRACKET DENOTE THE INCREASE OR DECREASE COMPARED WITH THE COMPETITORS

Targets	INSTILLER	INSTILLER^{-S}	INSTILLER^{-M}	INSTILLER^{-SM}
Coverage	279929.15	267754.12(-4.3%)	259643.88(-7.2%)	255386.9(-8.8%)
Length	436.11	467.08(+7.1%)	455.86(+4.5%)	478.1(+9.6%)

[1] INSTILLER^{-S} denotes INSTILLER without seed selection strategy.
[2] INSTILLER^{-M} denotes INSTILLER without mutation strategy.
[3] INSTILLER^{-SM} denotes INSTILLER without seed selection and mutation strategies.

process is guided towards increasing coverage. Therefore, INSTILLER has 4.3% more coverage than INSTILLER^{-S}. Similarly, Equation 5 uses dictionary mutation to mitigate coverage decrease, and INSTILLER^{-M} has 7.2% less coverage when the mutation strategy is disabled. The coverage of INSTILLER^{-SM} is more than DiFuzzRTL in Table II. This result indicates the effectiveness of other techniques in INSTILLER, e.g., the VACO algorithm.

For input instruction length, the result of INSTILLER^{-S} is 7.1% greater than INSTILLER. In Equation 4, we use heuristics divided by length, aiming to select the shortest input instruction. Therefore, the seed selection strategy also helps shorten input instruction length. In our mutation strategy, the length of the input instruction is controlled by insertion and deletion. The result of INSTILLER^{-M} is 4.5% greater than INSTILLER, indicating the effectiveness of the mutation strategy. INSTILLER^{-SM} has a shorter length than DiFuzzRTL in Table III, proving the huge effect of distillation on input instruction length.

Moreover, we conduct experiments to compare INSTILLER with INSTILLER without seed selection and mutation (denoted as INSTILLER^{-SM}), which is illustrated in Table XI. INSTILLER outperforms INSTILLER^{-SM} in all the tested CPUs. Especially in Rocket, the performance difference reaches the maximum of 7.8%. On average, INSTILLER surpasses INSTILLER^{-SM} by 3.0%, and all the differences are significant according to the p values and $\hat{A}_{12}$ values. Referring to Equation 4 and Equation 5, we add *cov* in the heuristics in seed selection and mutation. Covering more parts of the code contributes to the bug-finding performance of INSTILLER.

TABLE XI
NUMBER OF MISMATCHES OF INSTILLER AND INSTILLER WITHOUT SEED SELECTION AND MUTATION (DENOTED AS INSTILLER^{-SM}), WHERE THE VALUES IN THE BRACKET DENOTE THE INCREASE OR DECREASE COMPARED WITH THE COMPETITORS

Targets	INSTILLER	INSTILLER^{-SM}	p value	$\hat{A}_{12}$
mor1kx	110.1(+5.6%)	104.3	0.002	1.0
or1200	598.3(+5.1%)	569.3	0.01	1.0
Boom	5546.0(+2.7%)	5399.3	0.001	1.0
Rocket	33.3(**+7.8%**)	30.9	0.015	1.0
Average	1573.9(+3.0%)	1525.95	0.007	1.0

- Therefore, we can answer RQ6: *The seed selection and mutation strategies are also effective in increasing coverage, shortening input length, and detecting bugs.*

I. Comparison with Other Hardware Fuzzers

Despite using differential testing, there are other types of hardware fuzzers. In this part, we compare INSTILLER with Hardware Fuzzing Pipeline (HFP) [37] and RFuzz [9]. Both HFP and RFuzz detect bugs without differential testing. There are eight experiment targets, including FFTSmall, Sodor1Stage, Sodor3Stage, Sodor5Stage, TLI2C, TLPWM, TLSPI, and TLUART. These targets are collected from the experiments of HFP and RFuzz. We use HDL line coverage as the coverage metric to show the results. As Figure 12 shows, INSTILLER has more HDL line coverage than HFP and RFuzz. The internal reason for this result is that we propose instruction distillation based on coverage in INSTILLER. The distilled input instructions are short in length and beneficial to coverage. Results in Section V-B have similar conclusions that INSTILLER has more coverage than DiFuzzRTL.

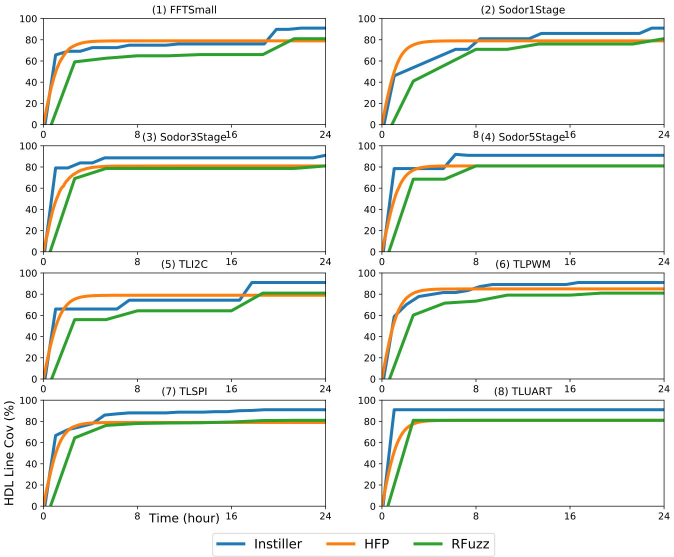


Fig. 12. HDL line coverage of INSTILLER, HFP, and RFuzz over time, where the X-axis denotes time, and the Y-axis denotes coverage.

J. Real-world Vulnerabilities

To prove the ability to detect real-world bugs of INSTILLER, we conduct the following evaluation. We collect 12 real-world bugs from the GitHub issue pages of the tested CPUs [30], [31], [34], [35]. As shown in Table XII, the target CPU, bug ID, description, and reproductivity of INSTILLER are listed. In total, INSTILLER can reproduce 8 out of the 12 real-world bugs. All of the detected bugs are produced by the bug-reporting mechanism of INSTILLER, and then they are confirmed by practitioners. This result demonstrates that INSTILLER has the ability to detect real-world vulnerabilities.

VI. DISCUSSION

A. Golden Reference Models (GRM)

INSTILLER and other hardware fuzzers [10], [25] rely on GRMs to detect hardware bugs. The validation of many commercial CPUs largely depends on the availability of GRMs. There are many industrial large-scale simulators using GRMs, such as Intel x86 Archsim, AMD x86 Simnow, and ARM Cortex Neoverse. Thus, the availability of GRM is not a constraint for INSTILLER. Moreover, the GRM itself is highly

TABLE XII
INFORMATION OF REAL-WORLD BUGS AND THE DETECTION ABILITY OF INSTILLER

Targets	ID	Description	Reproducible
Boom	V1	Instruction count is inaccurate when minstret is written by software.	✔
Boom	V2	Static rounding is ignored for fdiv.s and fsqrt.s.	✔
Boom	V3	Floating point instruction which has invalid rm field does not raise exception.	✔
Boom	V4	FS bits in mstatus register is set after fle.d instruction.	✗
mor1kx	V5	Incorrect implementation of the carry flag generation.	✗
mor1kx	V6	Missing access checking for privileged register.	✔
mor1kx	V7	eear register not saving instruction virtual address when illegal instruction exception.	✔
mor1kx	V8	l.fl1, l.ff1 instruction decoding fails.	✔
or1200	V9	Incorrect forwarding logic for the GPR0.	✗
or1200	V10	Incomplete update logic of overflow bit formsb & mac instructions.	✔
or1200	V11	Incorrect generation of overflow flag.	✔
Rocket	V12	EBREAK does not increase instruction count.	✗

unlikely buggy. They have been carefully designed, which are developed with strict version control, and have undergone extensive testing. Therefore, both in availability and validity, using GRMs in INSTILLER to detect bugs is not a concern.

B. Different Coverage Metrics

In this paper, we use control register coverage mentioned in [10]. However, there are other coverage metrics in hardware fuzzing. In RFuzz [9], mux control coverage is utilized. TheHuzz [25] considers multiple coverage metrics, including branch coverage, condition coverage, FSM coverage, etc. Moreover, software coverage is directly used on the translated model of the hardware RTL in [37]. Although coverage metrics of hardware fuzzing are not the research scope of this paper, we still conducted preliminary evaluations on the metrics. Generally speaking, control register coverage performs the best among the metrics in the evaluation. Therefore, we utilize it in this paper.

C. Hyper-parameters

There are three hyper-parameters in the design of INSTILLER, including the tunable parameter in Equation 1, the proportion of basic metrics in Equation 4, and the probability of each mutation operator in Equation 5. For Equation 1, the parameter ρ will affect the strength of mutual influence between the ants, which is related to the global search ability and convergence speed of the algorithm. We compare the three recommended configurations in [38], including 0.02, 0.1, and 0.5. Table XIII shows the comparison of different configurations of ρ. The execution speed of these configurations is similar, and 0.5 reaches the maximum in coverage and mismatches. Therefore, we choose 0.5 as the configuration of ρ in Equation 1.

TABLE XIII
EVALUATION ON PARAMETER ρ IN EQUATION 1 ABOUT COVERAGE, SPEED, AND MISMATCHES

ρ	**0.02**	**0.1**	**0.5**
Coverage	2265493.18	261107.05	**279929.15**
Speed	0.27	0.26	**0.27**
Mismatches	1501.8	1511.6	**1571.9**

Moreover, we conduct experiments to discuss the parameter w in Equation 4. This parameter controls the balance between the basic metrics and the RTL metrics. The basic metrics are related to coverage and execution speed, and the RTL metrics are related to CPU bugs.

TABLE XIV
EVALUATION ON PARAMETER w IN EQUATION 4 ABOUT COVERAGE, SPEED, AND MISMATCHES

w	**0.1**	**0.5**	**1.0**	**2.0**	**10.0**
Coverage	254329.35	257754.12	268953.90	279929.15	281876.22
Speed	0.24	0.25	0.25	0.27	0.31
Mismatches	1655.9	1641.5	1590.1	1571.9	1105.8

Table XIV shows how w in Equation 4 influences the results of coverage, execution speed, and the number of mismatches. As w increases, coverage and speed also increase, and the number of mismatches decreases. We select 2.0 as the configuration of w. This value keeps the balance between coverage, speed, and the number of mismatches.

Besides, we conduct experiments to discuss the length parameter l that determines the choices of the mutation operators in Equation 5. If the length of the current fuzzing exceeds l, the “deletion” operator will be chosen. Otherwise, “insertion” is selected. Table XV shows the comparison of three different configurations of l. This parameter can affect the input length of fuzzing. We choose 400 as the configuration in our implementation, as it keeps a balance between coverage, speed, and mismatches, compared with other configurations.

TABLE XV
EVALUATION ON PARAMETER l IN EQUATION 5 ABOUT COVERAGE, SPEED, AND MISMATCHES

l	**100**	**400**	**1000**
Coverage	235617.60	279929.15	251265.47
Speed	0.33	0.27	0.20
Mismatches	1211.7	1571.9	1562.3

D. Power Schedule

Power schedule is not mentioned in this paper, which is commonly seen in fuzzing tools [23], [39]. If we treat fuzzing as an optimization problem, there are many methods to reach the desired optimum, including seed selection and mutation, coverage metrics, power schedule, etc. The key mechanism of this paper is the VACO algorithm, which shortens the input length, increases coverage, and keeps the input sequence efficient in fuzzing the hardware. As the evaluation shows, VACO accomplishes these tasks with its internal design. Besides, there are other approaches in INSTILLER that make it more realistic in hardware fuzzing, including hardware-based seed selection and mutation. The above components contribute to the “efficient and realistic” fuzzing of INSTILLER. However, we are not claiming that the power schedule is not important in fuzzing. It is not the research focus of INSTILLER, so it is not included in our paper. Besides, we are preparing to utilize the power schedule in hardware fuzzing and leave this as future work.

E. Remaining Challenges and Future Improvements

Remaining challenges. The simulation of RTL execution is relatively slow compared with binary fuzzing, e.g., AFL.

Binary fuzzing can reach the speed of thousands of executions per second, and RTL fuzzing can only reach one execution per several seconds. It is still a challenge to speed up the execution speed of RTL simulation.

In software instrumentation, LLVM and Clang greatly reduce the workload of practitioners. However, the instrumentation of CPU RTL requires FIRRTL. It is a time-consuming process to instrument the code.

Future improvements. In the future, multiple ISAs can be added to INSTILLER, such as ARM ad X86. It would be more attractive for manufacturers to invest in fuzzing these commercial ISAs. However, unlike RISC-V, it is relatively difficult for researchers to get access to these resources to conduct research.

We also plan to propose a new coverage mechanism, aiming to solve different types of bugs in fuzzing the CPU. For example, for bugs related to side channels, such as Meltdown, the coverage mechanism should focus on the branch predictions in the RTL.

VII. RELATED WORK

A. RTL Verification and Testing

RTL testing is a popular field in research, even in capture-the-flag competitions [40]. RFuzz utilizes mux coverage [9] to fuzz CPU. DiFuzzRTL applies differential fuzz testing [10], which is a pioneer in this field. TheHuzz uses different coverage metrics [25] and conducts experiments to show their performance. SpecDoctor is an automated RTL fuzzer to find transient execution vulnerabilities [41]. GenFuzz is a GPU-accelerated hardware fuzzer with a genetic algorithm and multiple inputs [42]. Cascade uses asymmetric ISA pre-simulation to construct RISC-V programs [43]. INSTILLER is different from them, and we aim to shorten input instruction length and increase coverage.

HyperFuzzing converts hardware into software [44], which is a different approach to fuzz RTL. Hardware Fuzzing Pipeline also translates hardware to software to fuzz CPU [37]. Our tool does not need to translate hardware to software. These two tools are different from INSTILLER.

B. Coverage-guided Grey-box Fuzzing

CGF becomes prevalent since the release of AFL [23]. It discovers numerous bugs with its well-designed mechanism. The key to CGF is the coverage feedback to the fuzzer. AFLFast [27] is another milestone of CGF, which improves the power schedule (the number of executions on a seed). MOPT [32] automatically selects mutation operators using Particle Swarm Optimization (PSO), which is a promising direction in fuzzing research. CollAFL proposes a coverage-sensitive fuzzing solution [45], which uses a finely designed coverage metric to effectively avoid path collisions in fuzzing. ovAFLow [46] deals with the problem of taint input bytes. It utilizes a lightweight fuzzing-based taint inference to guide the mutation strategy. INSTILLER is different from them, which utilizes the idea of CGF combined with differential testing to detect CPU bugs. More importantly, the goal of INSTILLER is to distill input instructions for more efficient hardware fuzzing, and it is different from the above fuzzers.

C. Optimization in Fuzzing

As a classical optimization technique, ACO is used in fuzzing. ACOFuzz uses ACO to allocate energy [47], which controls the power schedule of fuzzing. AFL-ant proposes a seed screening technique based on ACO [48]. This technique chooses the best seed with ACO. RGF concentrates on new code with an ACO-based power schedule [49]. Therefore, it is a directed fuzzer. In INSTILLER, based on the characteristics of RTL fuzzing, we propose a variant of ACO to distill input instructions. Our VACO is different from the classic ACO in the above-related work. INSTILLER and these ACO-based fuzzers focus on different aspects of the fuzzing process.

Other optimization techniques can also be adopted in fuzzing. For example, EcoFuzz [28] uses a variant of multi-arm bandit (MAB) to assign energy in fuzzing. The power schedule is altered according to the fuzzing status. MobFuzz [29] solves the problem of multi-objective optimization with a method called multi-player MAB. The optimization technique in INSTILLER is ACO, which is different from the MAB algorithm, and we make modifications to it according to the situation of RTL fuzzing.

D. Seed Selection and Mutation

Seed selection and mutation strategies can improve fuzzing efficiency. TortoiseFuzz uses three metrics to select and prioritize seeds [50]. FairFuzz selects seeds that are chosen less frequently [33]. MemLock uses the amount of memory consumption to detect memory consumption bugs [51]. FuzzFactory proposes waypoints to guide seed selection and improve fuzzing [52]. Waypoints can be memory consumption, algorithmic complexity, etc. MoonShine [53] optimizes OS fuzzer seed selection with trace distillation. It can distill seeds from system call traces of real-world programs while preserving the dependencies across the system calls. [54] compares six seed selection approaches, concluding that fuzzing highly relies on the initial seed corpus. However, none of them is related to the heuristics of hardware, e.g., the number of jump instructions. INSTILLER uses hardware-related strategies, which improves the RTL fuzzing efficiency.

VIII. CONCLUSION

In this paper, we conclude three challenges in RTL fuzzing in previous work, which include increasing input instruction length, no realistic interruption or exception, and no hardware-related seed selection and mutation. To solve these challenges, we propose input instruction relationship extraction, together with input instruction distillation based on a variant of ant colony optimization. Moreover, we enable our fuzzer to include multiple interruptions and exceptions to cover more RTL states. We also propose hardware-related seed selection and mutation strategies to improve the fuzzing performance. In addition, we implement a prototype INSTILLER and conduct extensive experiments against state-of-the-art fuzzing work. The results show our tool outperforms previous work in coverage, input instruction length, and vulnerability discovery, which demonstrate the effectiveness of our proposed techniques.

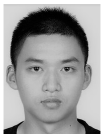
Acknowledgment

Thanks to the reviewers' valuable comments and helpful suggestions. This work is supported by the Natural Science Foundation of China (61902412, 61902416), the Research Project of National University of Defense Technology (ZK20-17, ZK20-09, and ZK23-14), the Natural Science Foundation of Hunan Province of China (2021JJ40692).

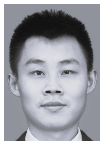
References

[1] Meltdown and Spectre. https://meltdownattack.com/.
[2] The Pentium FDIV Bug. http://scihi.org/the-pentium-fdiv-bug/.
[3] Machine check exception on Broadwell.
[4] Ryzen Segfault Bug. https://Ryzen_Processor_Segfault_Bug.html.
[5] Intel Analysis of Speculative Execution Side Channels.
[6] Dinos Moundanos and et al. Abstraction techniques for validation coverage analysis and test generation. *ToC*, 1998.
[7] Michael Kantrowitz and et al. I'm done simulating; now what? verification coverage analysis and correctness checking of the decchip 21164 alpha microprocessor. In *DAC 1996*. IEEE.
[8] Barton P Miller and et al. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 1990.
[9] Kevin Laeufer and et al. Rfuzz: Coverage-directed fuzz testing of rtl on fpgas. In *ICCAD 2018*. IEEE.
[10] Jaewon Hur and et al. Difuzzrtl: Differential fuzz testing to find cpu bugs. In *SP 2021*. IEEE.
[11] Chijin Zhou and et al. Zeror: Speed up fuzzing with coverage-sensitive tracing and scheduling. In *ASE 2020*.
[12] Stefan Nagy and et al. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In *SP 219*. IEEE.
[13] Cao Zhang and et al. Instrcr: Lightweight instrumentation optimization based on coverage-guided fuzz testing. In *CCET 2019*. IEEE.
[14] Mingzhe Wang and et al. {RIFF}: Reduced instruction footprint for {Coverage-Guided} fuzzing. In *USENIX ATC 2021*.
[15] Mingzhe Wang and et al. Odin: on-demand instrumentation with on-the-fly recompilation. In *PLDI 2022*.
[16] Shaohua Li and et al. Accelerating fuzzing through prefix-guided execution. *PACMPL 2023*.
[17] H. Peng and et al. T-fuzz: Fuzzing by program transformation. In *SP 2018*.
[18] Sergej Schumilo and et al. {kAFL}:{Hardware-Assisted} feedback fuzzing for {OS} kernels. In *USENIX Security 2017*.
[19] Christian Blum. Ant colony optimization: Introduction and recent trends. *Physics of Life reviews*, 2005.
[20] RISC-V PLIC Spec. https://riscv-plic-1.0.0_rc6.pdf.
[21] The RISC-V Instruction Set Manual Volume II. https://riscv-privileged-20190608-1.pdf.
[22] Andrew Waterman and et al. The risc-v instruction set manual. *Volume I: User-Level ISA', version*, 2014.
[23] American fuzzy lop. https://github.com/google/AFL.
[24] William M McKeeman. Differential testing for software. *Digital Technical Journal*, 1998.
[25] Rahul Kande and et al. {TheHuzz}: Instruction fuzzing of processors using {Golden-Reference} models for finding {Software-Exploitable} vulnerabilities. In *USENIX Security 2022*.
[26] Adam Lipowski and et al. Roulette-wheel selection via stochastic acceptance. *Physica A: Statistical Mechanics and its Applications*, 2012.
[27] Marcel Böhme and et al. Coverage-based greybox fuzzing as markov chain. In *CCS 2016*.
[28] Tai Yue and et al. {EcoFuzz}: Adaptive {Energy-Saving} greybox fuzzing as a variant of the adversarial {Multi-Armed} bandit. In *Usenix Security 2020*.
[29] G Zhang and et al. Mobfuzz: Adaptive multi-objective optimization in gray-box fuzzing. In *NDSS 2022*.
[30] Boom. https://github.com/riscv-boom/riscv-boom.
[31] Rocket chip generator. https://github.com/chipsalliance/rocket-chip.
[32] Chenyang Lyu and et al. Mopt: Optimized mutation scheduling for fuzzers.
[33] Caroline Lemieux and et al. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *ASE 2018*.
[34] mor1kx. https://github.com/openrisc/mor1kx.
[35] OpenRISC 1200 implementation. https://github.com/openrisc/or1200.
[36] George Klees and et al. Evaluating fuzz testing. In *CCS 2018*.
[37] Timothy Trippel and et al. Fuzzing hardware like software. In *USENIX Security 2022*.
[38] Marco Dorigo and et al. Ant colony optimization. *IEEE computational intelligence magazine*, 2006.
[39] Andrea Fioraldi and et al. {AFL++}: Combining incremental steps of fuzzing research. In *WOOT 2020*.
[40] Chen Chen and et al. Trusting the trust anchor: towards detecting cross-layer vulnerabilities with hardware fuzzing. In *DAC 2022*.
[41] Jaewon Hur and et al. Specdoctor: Differential fuzz testing to find transient execution vulnerabilities. In *CCS 2022*.
[42] Dian-Lun Lin and et al. Genfuzz: Gpu-accelerated hardware fuzzing using genetic algorithm with multiple inputs. In *DAC 2023*.
[43] Flavien Solt and et al. Cascade: Cpu fuzzing via intricate program generation.
[44] Sujit Kumar Muduli and et al. Hyperfuzzing for soc security validation. In *ICCAD 2020*.
[45] Shuitao Gan and et al. Collafl: Path sensitive fuzzing. In *SP 2018*. IEEE.
[46] Gen Zhang and et al. ovaflow: Detecting memory corruption bugs with fuzzing-based taint inference. *JCST*, 2022.
[47] You Wu and et al. Acofuzz: Adaptive energy allocation for greybox fuzzing. In *ICSTW 2022*. IEEE.
[48] Bowen Sun and et al. Greybox fuzzing based on ant colony algorithm. In *ICAINA 2020*. Springer.
[49] Xiaogang Zhu and et al. Regression greybox fuzzing. In *CCS 2021*.
[50] Yanhao Wang and et al. Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization. In *NDSS*, 2020.
[51] Cheng Wen and et al. Memlock: Memory usage guided fuzzing. In *ICSE 2020*.
[52] Rohan Padhye and et al. Fuzzfactory: Domain-specific fuzzing with waypoints. *Proc. ACM Program. Lang.*, 2019.
[53] Shankara Pailoor and et al. {MoonShine}: Optimizing {OS} fuzzer seed selection with trace distillation. In *USENIX Security 2018*.
[54] Adrian Herrera and et al. Seed selection for successful fuzzing. In *ISSTA 2021*.

Gen Zhang received his Ph.D. degree in computer science and technology in 2022 from NUDT. His research interests include fuzzing and testing.

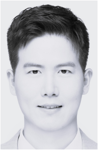


Pengfei Wang received his B.S., M.S., and Ph.D. degrees in computer science and technology, in 2011, 2013, and 2018, respectively, from NUDT. His research interests include operating system and software testing.



Tai Yue received his B.S. and M.S. degrees in computer science and technology, in 2017 and 2019 from Nanjing University and NUDT. His research interests include fuzzing and testing.

Danjun Liu received his Ph.D. degree in computer science and technology in 2023 from NUDT. He is now an assistant professor in NUDT. His research interests include operating systems and parallel computing.



Yubei Guo received her master degree in computer science and technology in 2020, from Hunan University. Her research interests include security.

Kai Lu received his B.S. degree and Ph.D. degree in 1995 and 1999, respectively, both in computer science and technology from NUDT. He is now a professor in NUDT. His research interests include operating systems, parallel computing, and security.