2026 Design, Automation & Test in Europe Conference (DATE 2026)

Fine-Grained Code Analysis for Processor Fuzzing

Ziyue Zheng*, Zhi Qu* and Yangdi Lyu†
Microelectronics Thrust, The Hong Kong University of Science and Technology (Guangzhou)
†Corresponding author: yangdilyu@hkust-gz.edu.cn

***Abstract*—The increasing complexity of modern processor designs has posed significant challenges in achieving comprehensive coverage metrics for functional verification of Register-Transfer Level (RTL) designs. Despite the availability of white-box RTL models, recent advancements in hardware fuzzing have predominantly focused on grey-box methodologies, which lack effective utilization of internal logic and structural information.**

This paper presents a novel approach that addresses this limitation by extracting control flow graphs (CFGs) from processor designs and analyzing the dependencies within these graphs. The analyzed CFGs serve as heuristic information to guide the generation of processor stimuli. By effectively leveraging internal logic information during the simulation of complex processors, this method provides interpretable heuristics for test generation. Experimental results demonstrate the effectiveness of utilizing control flow information derived from processor designs in enhancing the convergence speed of coverage metrics and guiding test sequences towards hard-to-reach states.

I. Introduction

Test generation is a common approach in functional verification used to verify whether a design meets its specifications. However, traditional test generation approaches face considerable challenges when dealing with complex designs. While fully formal methods, such as bounded model checking (BMC) [1]–[3], can theoretically prove design correctness or generate counterexamples, they require vast computational resources that are impractical for large industrial designs. Random simulation methods, on the other hand, suffer from inadequate coverage and inefficiency in exploring the design space, especially when targeting hard-to-reach states.

There is a growing trend in the community towards integrating software-based methodologies, such as fuzzing, into hardware testing procedures. Coverage-guided fuzzing has emerged as a promising test generation technique in the realm of hardware design [4], [5], demonstrating the ability to improve functional coverage and identify vulnerabilities [6], [7]. Coverage-guided fuzzing operates through three main stages: (1) *Corpus Generation*: Generates an initial set of random or crafted inputs (seeds) to form the corpus. (2) *Mutation and Merge*: Seeds are mutated (e.g., bit flips) to form mutated inputs. (3) *Feedback*: Executes and analyzes the mutated inputs. Seeds that provide new coverage or behaviors are added to the origin corpus. This iterative methodology allows for thorough exploration of the design space, capturing crashes for further analysis and uncovering issues that traditional methods may have overlooked.

*Ziyue Zheng and Zhi Qu contributed equally to this work.

This work was supported by the National Natural Science Foundation of China under Grant #62402412.

Prior processor fuzzing efforts struggled to achieve high coverage on large-scale designs due to the lack of fine-grained heuristics. Many of these approaches focused on covering specific registers [5], [6], which limited their effectiveness in complex scenarios. While directed fuzzing methods [8] introduced instance-level distance to guide test generation, they lacked finer-grained information (e.g., statements, branches).

To overcome these limitations, we propose a novel methodology that enables fine-grained, design-aware test generation. Our approach combines static analysis of processor RTL with dynamic feedback-guided fuzzing to systematically explore complex state spaces.

The methodology begins with an automated analysis that extracts control and data dependencies directly from the processor implementation. Unlike previous work that relied on manually identified control-state registers [5], [6], our technique automatically constructs a comprehensive model of the processor's decision logic by analyzing branch conditions and state transitions at the statement level. This analysis operates on a consolidated representation of the design to ensure complete coverage of cross-module dependencies. Following this, we develop a feedback mechanism that evaluates test inputs based on their ability to exercise unexplored control-flow paths. The fuzzing engine prioritizes inputs that demonstrate progress toward covering difficult-to-reach states, as identified through our static analysis.

The key contributions of this work include:

- We present an open-source framework[1] for processor RTL verification. The framework automatically extracts control-flow and data dependencies from the RTL. The analysis is then used to guide targeted processor fuzzing, eliminating the need for manual identification of critical registers or design-specific heuristics.
- A novel runtime feedback mechanism that leverages fine-grained statement and branch coverage metrics to guide test generation, enabling more systematic exploration of processor state spaces.
- Experimental results on open-source RISC-V processors (Rocket Chip and BOOM) demonstrate that our approach achieves higher coverage and bug-finding efficiency compared to traditional fuzzing methods.

II. Related Work

Recent advancements in hardware fuzzing have introduced various methodologies to address the unique challenges of hardware testing. RFUZZ [4] utilizes mux-toggle coverage

[1]https://github.com/HKUSTGZ-MICS-LYU/FineGrainedFuzz

978-3-9826741-1-7/DATE26/© 2026 EDAA

TABLE I: Comparison of Fuzzing Framework

Design	Hardware	Software	Fine-Grained Code Analysis
RFUZZ [4]	✓		
DifuzzRTL [6]	✓		
TheHuzz [7]	✓		
Hypfuzz [10]	✓		✓✗*
FormalFuzzer [11]	✓		✓✗*
DAFL [14] , [13]		✓	✓
Our Work	✓		✓

*Use a commercial formal tool to assist test generation.

to monitor specific signals in multiplexers, using mutation operator scheduling inspired by the AFL fuzzer [9]. DifuzzRTL [6] employs differential fuzz testing to detect CPU bugs, creating an ISA-level instruction generator and mutator, and adopting a register-coverage aware metric to guide test generation and mutation. Similarly, TheHuzz [7] leverages an ISA-level instruction generator but takes a step further by analyzing intrinsic hardware behaviors in hardware description languages (HDLs) and utilizing coverage metrics to generate targeted assembly-level instructions. Recent efforts, such as Hypfuzz [10] and FormalFuzzer [11], integrate commercial formal tools like JasperGold [12] to explore states that are difficult for random seed generators to reach.

Despite recent advancements, hardware fuzzing has primarily relied on metrics such as mux, toggle, and register coverages, which encounter challenges due to the vast search space and their lack of directness. Table I summarizes various fuzzing techniques, focusing on applications to hardware and software testing, as well as their ability to perform fine-grained code analysis. While much of the previous work in hardware fuzzing has concentrated on coverage metrics without incorporating code analysis, tools like Hypfuzz and FormalFuzzer utilize commercial formal tools that apply a form of code analysis during the solving process, as indicated by the half-checkmark in the table. In Table I, we also highlight DAFL [14] as an example of an advanced software fuzzer that employs control flow and data dependency analysis. However, it is crucial to recognize that software fuzzers cannot be directly used for hardware fuzzing due to fundamental differences in their underlying structures. Our work uniquely applies control flow and data flow analysis to hardware designs, enhancing the effectiveness of hardware fuzzing.

III. Framework

Our proposed framework, as depicted in Figure 1, is divided into three main parts. First, a hierarchical RTL model is converted into a flattened and symbolized design to simplify subsequent analysis. Second, we perform static analysis on the processed RTL to extract the dependency relationships of basic blocks (BBs). This involves constructing a Control Flow Graph (CFG) and determining the dependencies, which are essential for identifying critical paths and understanding the control flow within the design. Finally, we integrate these dependency relationships into a traditional fuzz loop to guide and optimize directed test generation.

A. RTL Preprocessing

The RTL preprocessing transforms the original hardware design into an analysis-ready representation through two key steps: design flattening and RTL symbolization. This transformation enables subsequent fine-grained static analysis while maintaining computational efficiency suitable for large-scale processor designs.

Hierarchical RTL design methodology enhances readability and reusability but complicates static analysis, such as control flow and dependency analysis. Previous work, such as DirectFuzz [8], employs module connectivity graphs for test generation but cannot achieve comprehensive, dependency-aware RTL analysis. To overcome this, our framework flattens all RTL modules [15] to fully preserve dependency information.

Building upon the flattened design, we developed a lightweight framework that systematically converts RTL into symbolic representations at the statement level. This conversion preserves the complete semantic information while enabling direct application of software analysis techniques. The framework operates with high efficiency, as demonstrated by our experimental evaluation in Section IV, and produces the structured input required for the comprehensive static analysis described in Section III-B.

B. Static Analysis

The static analysis aims to guide directed test generation in traditional fuzz loops. The analysis is divided into two steps, referred to as ❸ and ❹ in Figure 1. In the first step, we construct a CFG and perform branch-level code instrumentation. Along with the construction of CFG, our framework can symbolize each condition and statement in the processor design. These conditions and statements can then be utilized for control flow dependency analysis.

1) CFG Construction & Instrumentation: To construct the control flow graph (CFG) and perform code instrumentation, we rely on a parser to analyze the structural information from Verilog code. During the parsing process, we extract expressions for each condition and statement, which will be used later for variable dependency analysis. Since the RTL code has already been flattened, we do not need to handle instance-related information.

In our CFG, each process in Verilog is represented separately in a subgraph. The nodes in this graph, known as basic blocks (BBs), contain the relevant statements, as well as their predecessors and successors. It is worth emphasizing that "assign" statements are treated as a distinct type of basic block in our approach. This distinction is made because an "assign" statement is functionally equivalent to a process that continuously assigns values (i.e., an always(*) statement).

For instrumentation, instead of directly printing control flow coverage, we utilize the Verilog Programming Interface (VPI) to read the information during simulation. Due to the limited maximum readable register width in VPI for some simulators,

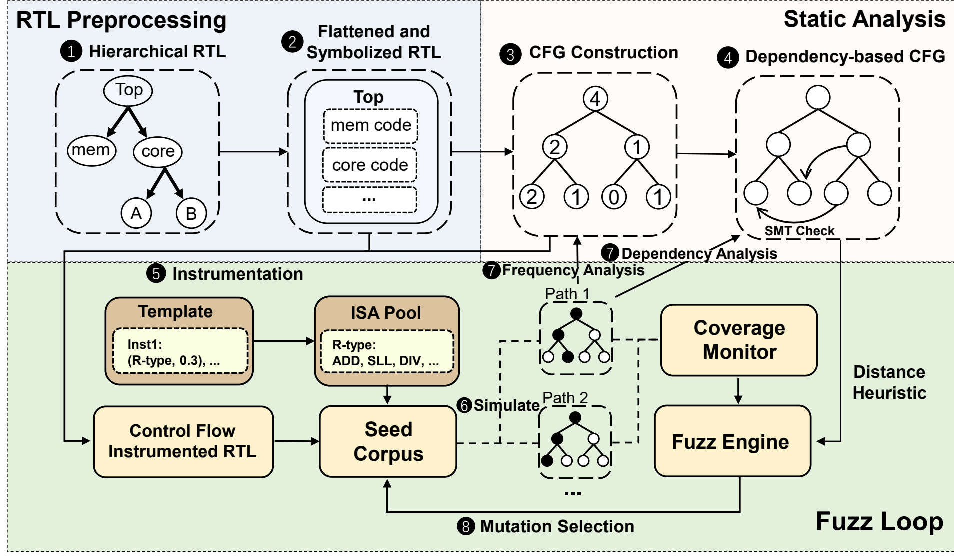


Fig. 1: The overall framework of our work, comprising static analysis and the fuzz loop. We integrate control flow analysis to construct heuristics for the fuzz engine, guiding simulation input selection and mutation.

we create several registers, each with a width of 500, to store whether each basic block has been accessed (excluding "assign" type BBs). A basic block that has been accessed will have the corresponding bit in the register set to 1.

2) Dependency Analysis: The purpose of dependency analysis is to construct precise dependencies between basic blocks. In cycle-based hardware design, statements accessed in the current cycle may depend on statements from several cycles earlier. Dependency analysis aims to uncover these temporal relationships.

The steps for dependency analysis are as follows. First, it iterates over all the guard conditions of branches in the design, denoted as $G = \{g_1, g_2, \dots\}$. For each guard condition g, it identifies all the statements that assign values to each variable within g. Each assignment e is then converted into the single static assignment (SSA) form for satisfiability checking, as shown in Equation 1.

$$Q(i) = \neg g_{i_0} \land e \land g_{i_1} \tag{1}$$

If the check is successful, the basic block containing e is considered a predecessor of the basic block guarded by g. If e is an assignment expression, it will identify all the variables on the right-hand side of e and find all the statements e' that assign values to those variables, making the basic block containing the current expression e a predecessor of the basic block containing e'. This process repeats until no more predecessor relationships can be added.

We utilize the example code shown in Figure 2, which is adapted from a simplified snippet related to illegal instruction handling in the Rocket processor, to illustrate the outcomes of our dependency analysis. The left CFG ① illustrates the CFG without dependency analysis, where each basic block is connected based on control flow alone. The right CFG ② shows the CFG after dependency analysis, with additional predecessor relationships: b_4 is connected to a_3 due to *illegal_insn* assignment. The connections among a_1, a_2, and a_3 are due to the "assign" statements. For example, as a_1 modifies the register *status_isa*, it becomes the predecessor of a_2. And a_1 is connected to b_2 due to the assignment of *reg_misa* in b_2. This refined CFG provides a more accurate representation of the dependencies between basic blocks.

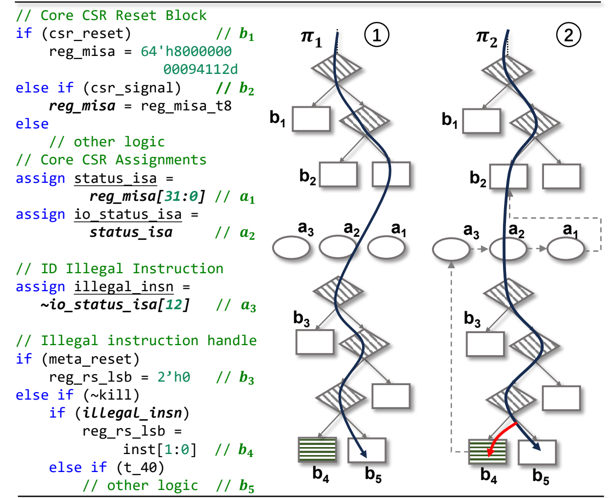


Fig. 2: This figure shows a comparison of control flow graphs (CFGs) derived from the simplified code of illegal instruction handling in the Rocket processor. Circle 1 (left) illustrates the CFG without dependency analysis, while circle 2 (right) displays the CFG after dependency analysis.

C. Fuzz Loop

The Fuzz Loop, shown at the bottom of Figure 1, is designed to maximize functional coverage by iteratively simulating and mutating input seeds. Each *seed* is a specific input that is

Algorithm 1 Seed Corpus Optimization in Fuzz Engine (Time-based Termination)

Input: RTL model $\mathcal{R}$, maximum seeds N, maximum running time MAX_Time, initial seed corpus I
Output: Optimized Seed corpus I'

```
1:  StartTime ← CurrentTime()
2:  I' ← I
3:  while CurrentTime() - StartTime < MAX_Time do
4:    if |I'| < N/10 then
5:      // Generation Stage
6:      I' ← I' ∪ GenerateNewSeed()
7:    else
8:      p ← Random(0, 1)
9:      if p < 0.1 then
10:       // Generation Stage
11:       I' ← I' ∪ GenerateNewSeed()
12:     else if p < 0.55 then
13:       // Mutation Stage
14:       seed ← SelectSeed(I', 1, 0.2)
15:       mutated_seed ← MutateSeed(seed)
16:       I' ← I' ∪ {mutated_seed}
17:     else
18:       // Merging Stage
19:       seed1, seed2 ← SelectSeed(I', 2, 0.2)
20:       merged_seed ← MergeSeeds(seed1, seed2)
21:       I' ← I' ∪ {merged_seed}
22:     end if
23:   end if
24: end while
25: return I'
```

composed of a fixed number of instruction words. The *Seed Corpus* is the set of all inputs, denoted as $I = \{i_1, i_2, \ldots\}$, where the size of I is limited by an upper bound N. The Fuzz Loop consists of two parts: generating an initial seed from the RTL and a template, and optimizing the seed corpus using a fuzz engine based on observed simulation results.

The initial seed corpus is generated from the instrumented RTL and a template. The *Template* refers to the format of the instruction set architecture (ISA). An *ISA Pool* includes various instruction types, such as R-type (e.g., ADD, SLL, DIV) and others. The instructions are constructed from the ISA pool and then assembled into instruction words, representing sequences of binary instructions ready for simulation.

The optimization of the seed corpus in the fuzz engine consists of three different stages: generation, mutation, and merging, which are demonstrated in Algorithm 1. In the *generation* stage, when the seed corpus contains fewer than $N/10$ seeds, the primary goal is to generate new seeds. Once the seed corpus exceeds $N/10$ seeds, the process transitions to a state where there is a 10% probability of generating new seeds, a 45% probability of mutating existing seeds, and a 45% probability of merging existing seeds. The *Mutation* stage involves selecting a seed and altering one of its instruction words, whereas the *merging* stage involves selecting two seeds and combining them to form a new seed.

In order to guide the selection of seeds for subsequent merge and mutation operations, the execution paths of previous seeds are recorded in simulation ❻. The execution paths are then analyzed to compute a heuristic value based on their paths, which is used to prioritize seeds that potentially increase functional coverage.

Two main heuristics are used in this work. The first one is the *frequency heuristic*, which focuses on the frequency of execution nodes. The second one is the *dependency-aware heuristic*, which considers the dependencies between different basic blocks.

1) Frequency Heuristic: The main goal of the frequency heuristic is to prioritize seeds that can access rare branches. Seeds that access rare branches are more likely to enter hard-to-reach states, thereby activating more unvisited branches and quickly increasing functional coverage.

Assume there are k branches in a processor design, and each branch has a current accumulated access count $C = \{t_1, t_2, \ldots, t_k\}$. The smaller the access count, the harder it is to reach that branch, with unvisited branches being our desired targets.

For all seeds, we can extract the indices of the branches they access. For example, if $seed_i$ accesses branches with indices $P_{seed_i} = \{p_1, p_2, \ldots\}$, where each p_i represents one accessed branch. We define the heuristic value for the exploration potential of this seed as:

$$H_{freq}(seed_i) = \sum_{p \in P_{seed_i}} \frac{1}{C_p} \tag{2}$$

Here, $H_{freq}(seed_i)$ is the heuristic value for $seed_i$, and it is computed as the sum of the reciprocals of the access counts of the branches accessed by $seed_i$. This value helps prioritize seeds that can access branches with lower access counts.

2) Dependency-aware Heuristic: The Dependency-aware Heuristic relies on the refined Control Flow Graph (CFG) formed by the dependency analysis introduced in Section III-B2. Let us use the CFG in Figure 2 as an example. Suppose b_4 is the only currently unvisited basic block, indicated by horizontal lines. Given two paths, π_1 and π_2, we would intuitively choose π_2 as the seed for mutation and merging because it passes through b_2, and b_2 is close to b_4 in the refined CFG.

When designing the Dependency-aware Heuristic, we consider P_{seed_i}, the set of branches accessed by $seed_i$. For each accessed branch $p_i \in P_{seed_i}$, we search for successors of p_i in the refined CFG with a distance less than or equal to 3. If any of these successors include an unvisited branch, we take the reciprocal of the smallest distance to an unvisited branch as the heuristic value for p_i. We then sum all these values together to form the heuristic value for the seed, as expressed in the following formula:

$$H_{dep}(seed_i) = \sum_{p_i \in P_{seed_i}} \left(\min_{d(p_i, u_j \in unvisited)} \frac{1}{d(p_i, u_j)} \right) \tag{3}$$

Here, $H_{dep}(seed_i)$ is the dependency-aware heuristic value for $seed_i$, p_i is an element of P_{seed_i}, and $d(p_i, u_j)$ is the distance from p_i to an unvisited branch u_j.

Algorithm 2 Select Seed with Frequency and Dependency-aware Heuristics

Input: Seed corpus I, number of seeds to select i, probability for Dependency-aware Heuristic p_{dep}
Output: Selected seeds S
1: Initialize empty list S
2: **for** each $seed \in I$ **do**
3: Calculate $H_{freq}(seed)$ as in Equation 2
4: **end for**
5: Use roulette wheel selection based on H_{freq} values to select i seeds, add to S_{freq}
6: Generate a random number $r \in [0, 1]$
7: **if** $r < p_{dep}$ **then**
8: **for** each $seed \in I$ **do**
9: Initialize $H_{dep}(seed) \leftarrow 0$
10: **for** each $p_i \in P_{seed}$ **do**
11: Find successors of p_i with depth ≤ 3
12: Compute the heuristic value as in Equation 3
13: Add the value to $H_{dep}(seed)$
14: **end for**
15: **end for**
16: Use roulette wheel selection based on H_{dep} values to select i seeds, add to S_{dep}
17: **else**
18: $S_{dep} \leftarrow \{\}$
19: **end if**
20: Combine S_{freq} and S_{dep} into $S_{combined}$
21: Randomly select i seeds from $S_{combined}$ and add to S
22: **return** S

Algorithm 2 shows how to select seeds based on both frequency and dependency-aware heuristics. Initially, it calculates the frequency heuristic H_{freq} for each seed in the corpus I to measure its potential to explore rarely accessed branches. Seeds are then selected using roulette wheel selection based on H_{freq} values, forming S_{freq}. A random number r determines whether the dependency-aware heuristic H_{dep} should be applied. Given that calculating H_{dep} involves a significant workload, H_{dep} is only computed if $r < p_{dep}$. This heuristic assesses the seed's potential to reach unvisited branches within a few steps in the CFG. Seeds are selected again using roulette wheel selection based on H_{dep} values, forming S_{dep}. The final step combines S_{freq} and S_{dep} into $S_{combined}$, from which i seeds are randomly chosen to form the output set S. This algorithm balances the exploration of rare branches and strategic CFG traversal, enhancing coverage and effectiveness.

IV. Experiment

In this paper, the flattened RTL process was implemented using a modified version of FlattenRTL [15]. The CFG construction, instrumentation, and dependency analysis were performed using Pyverilog [19] and the SMT-solver Z3 [20]. The simulation was conducted using cocotb [22] with Verilator.

The experiments were conducted on a high-performance computing cluster with Intel Xeon Gold 6348 CPUs. The benchmarks are two open-source RISC-V processor designs, i.e., Rocket [17] and Boom [18], with Spike [21] serving as the golden reference model. Table II shows the number of branches and lines of code in Rocket and Boom. Boom is more complex, with significantly more branches and lines of code than Rocket.

TABLE II: Benchmark Characteristics

Benchmark	Flattened	Total Branches	Total Lines
Rocket		7675	61201
Rocket*	✓	15400	111373
Boom		38523	202433
Boom*	✓	54018	345525

*The flattened benchmarks used in our paper

A. RTL Preprocessing and Static Analysis Time Evaluation

To obtain fine-grained RTL information, we developed a scalable tool combining RTL preprocessing (flattening and symbolic construction) with static analysis for CFG generation. Our tool can handle industrial-scale processors efficiently, as shown in Table III.

For Rocket, preprocessing took 7.88 seconds and static analysis took 154.84 seconds. For the more sophisticated Boom processor, preprocessing took 16.72 seconds, and static analysis required 688.61 seconds. These results demonstrate that our toolchain efficiently processes industrial-scale RTL designs. Notably, our static analysis completes in minutes, making it practical for integration with verification workflows. While fuzzing typically requires hours or days to achieve sufficient coverage, our approach provides critical insights that can significantly improve fuzzing efficiency by guiding test generation toward more promising paths.

TABLE III: RTL Preprocessing and Static Analysis Time for Rocket-Chip and Boom

Benchmark	Preprocessing Time (s)	Static Analysis Time (s)
Rocket	7.88	154.84
Boom	16.72	688.61

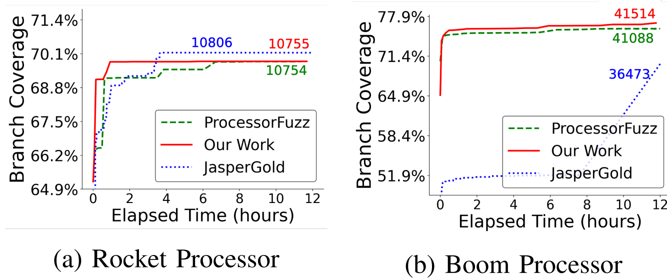
B. Coverage Evaluation

For coverage evaluation, we performed a 12-hour test generation twice. The results for time and coverage were averaged over iterations. Figure 3 shows the branch coverage over time for both the Rocket and Boom Processors. The x-axis represents the elapsed time in hours, while the y-axis indicates the branch coverage percentage. These graphs compare the performance of our framework with that of other methodologies.

The branch coverage for the Rocket Processor is depicted in Figure 3(a). The graph illustrates the comparative performance of three different methodologies: ProcessorFuzz, our work, and JasperGold [12]. JasperGold is a highly powerful commercial

formal verification tool that leverages formal methods to exhaustively verify the design. To generate tests to cover all branches using JasperGold, we constructed negated properties for all branches. While JasperGold ultimately covered more branches, our work achieves high coverage within 4 hours, reaching approximately 70.1%, surpassing both ProcessorFuzz and JasperGold. The final covered branch counts for JasperGold, ProcessorFuzz, and our work are 10806, 10754, and 10755, respectively.

The branch coverage for the Boom is shown in Figure 3(b). Our work achieves 76.9% branch coverage, covering 41,514 branches, while ProcessorFuzz reaches 76.1% coverage with 41,088 branches. JasperGold, however, increases coverage very slowly, ultimately reaching 36,473 branches. The slower coverage increase in JasperGold is due to the limitations of formal methods when dealing with the state explosion problem, making it less suitable for large models like the Boom Processor. This further demonstrates the efficiency of our approach in achieving higher branch coverage more rapidly compared to previous approaches.



(a) Rocket Processor (b) Boom Processor

Fig. 3: Branch coverage over time for the Rocket and Boom Processors.

C. Evaluation of Dependency-aware Heuristic

To evaluate the effectiveness of our dependency-aware heuristic, we conducted an ablation study by introducing an additional experiment that uses only the frequency heuristic. In this experiment, we removed the dependency-aware heuristic optimization and retained only the frequency heuristic.

Figure 4 shows the comparison of branch coverage in Boom Processor over time using ProcessorFuzz, the frequency heuristic (Freq), and the combined frequency and dependency-aware heuristic (Freq+Dep). The results indicate that the combined heuristic (Freq+Dep) achieves higher branch coverage than both the frequency-only heuristic and ProcessorFuzz. Specifically, after 12 hours, the branch coverage for Freq+Dep reaches 76.9% (41,514 branches), while the frequency-only heuristic (Freq) achieves 76.2% (41,135 branches), and ProcessorFuzz reaches 76.1% (41,088 branches). The results show that the dependency-aware heuristic is able to guide tests to hard-to-reach states by constructing dependency relationships, which help direct the tests towards unexplored branches.

D. Performance Evaluation of Bugs Discovery

To evaluate the bug discovery performance of our method, we conduct a 24-hour fuzzing comparison of our framework with ProcessorFuzz [5]. During this experiment, we identified

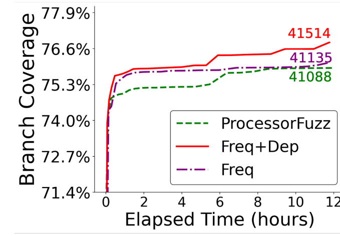


Fig. 4: Comparison of branch coverage in Boom over time using ProcessorFuzz, frequency heuristic (Freq), and combined frequency and dependency-aware heuristic (Freq+Dep).

the following mismatches between the RTL simulation and the golden model:

Bug 1: When `frm` contains an invalid value, floating-point instructions with dynamic rounding mode do not raise an illegal instruction exception.

Bug 2: The result of the `fsqrt.s` instruction with various `rm` fields differs between the processor and Spike.

Bug 3: After an invalid `fdiv` instruction, the 'invalid operation' flag in `fcsr` is not set.

Bug 4: The FS bit is incorrectly set to dirty in `mstatus/sstatus` when it is not needed.

Bug 5: A mismatch in `mstatus` occurs due to the MPRV implementation update from priv-1.12.

Table IV presents a summary of the time taken to identify each bug using our framework compared to ProcessorFuzz [5]. Our approach finds Bug 1 and Bug 4 on Boom and Rocket up to 5-15 times faster than ProcessorFuzz. Additionally, our framework successfully uncovers Bug 4 on Boom and Bug 5, which were not detected by ProcessorFuzz within the 24-hour timeframe. This demonstrates both higher efficiency and broader bug coverage of our framework.

TABLE IV: 24 Hours Fuzzing Bug discovery times comparison in seconds

Bug	Benchmark	Our work (s)	ProcessorFuzz [5] (s)
Bug 1	Boom	4725.28	14133.02
Bug 2	Boom	273.00	149.28
Bug 3	Boom	10049.05	1138.64
Bug 4	Rocket	2681.92	42574.42
	Boom	482.97	NA
Bug 5	Boom	137.44	NA

V. Conclusion

In this paper, we presented a novel fuzzing methodology that combines automated RTL analysis with feedback-guided test generation for complex processors. Our approach extracts control-flow and data dependencies without manual intervention and uses fine-grained coverage feedback to systematically explore processor state spaces. Experimental results on the Rocket Chip and BOOM demonstrate that our method not only achieves superior and more efficient coverage compared to traditional fuzzing techniques but also effectively identifies more bugs within a testing budget.

References

[1] D. K. and M. Purandare, "Ebmc: The enhanced bounded model checker," http://www.cprover.org/ebmc, 2022.

[2] R. Mukherjee, D. Kroening, and T. Melham, "Hardware verification using software analyzers," in *IEEE Computer Society Annual Symposium on VLSI*, 2015.

[3] A. Gargantini and C. Heitmeyer, "Using model checking to generate tests from requirements specifications," *SIGSOFT Softw. Eng. Notes*, 1999.

[4] K. Laeufer, J. Koenig, D. Kim, J. Bachrach, and K. Sen, "RFUZZ: Coverage-Directed Fuzz Testing of RTL on FPGAs," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2018, pp. 1-8.

[5] S. Canakci, et al., "ProcessorFuzz: Processor Fuzzing with Control and Status Registers Guidance," in *2023 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, San Jose, CA, USA, 2023, pp. 1-12. doi: 10.1109/HOST55118.2023.10133714.

[6] 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 (SP)*, 2021, pp. .

[7] R. Kande, A. Crump, G. Persyn, P. Jauernig, A.-R. Sadeghi, A. Tyagi, and J. Rajendran, "TheHuzz: Instruction fuzzing of processors using Golden-Reference models for finding Software-Exploitable vulnerabilities," in *USENIX Security Symposium*, 2022, pp. 3219–3236.

[8] 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 *58th ACM/IEEE Design Automation Conference (DAC)*, 2021, pp. 529-534.

[9] lcamtuf, "American Fuzzy Lop (AFL) Fuzzer," 2014, Last accessed on 05/21/2023. [Online]. Available: http://lcamtuf.coredump.cx/afl/technical

[10] C. Chen, R. Kande, N. Nguyen, F. Andersen, A. Tyagi, A.-R. Sadeghi, and J. Rajendran, "HyPFuzz: Formal-Assisted Processor Fuzzing," in *USENIX*, 2023.

[11] N. F. Dipu et al., "FormalFuzzer: Formal Verification Assisted Fuzz Testing for SoC Vulnerability Detection," in *ASPDAC '24*, Incheon, Republic of Korea, 2024, pp. 355–361. [Online]. Available: https://doi.org/10.1109/ASP-DAC58780.2024.10473911

[12] Jasper RTL Apps, "Jasper RTL Apps," Cadence, 2024. [Online]. Available: https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform.html.

[13] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed Greybox Fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2329–2344.

[14] T. E. Kim, J. Choi, K. Heo, and S. K. Cha, "DAFL: Directed Grey-box Fuzzing guided by Data Dependency," in *32nd USENIX Security Symposium (USENIX Security 23)*, Anaheim, CA, 2023, pp. 4931–4948. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/kim-tae-eun

[15] X. Meng, Z. Zheng, and Y. Lyu, "FlattenRTL: An Open Source Tool for Flattening Verilog Module at RTL Level," in *International Symposium of EDA (ISEDA)*, 2024.

[16] C. Chen, V. Gohil, R. Kande, A.-R. Sadeghi, and J. Rajendran, "PSOFuzz: Fuzzing Processors with Particle Swarm Optimization," in *ICCAD*, 2023.

[17] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman, "The Rocket Chip Generator," Technical Report UCB/EECS-2016-17, EECS Dept., Univ. of California, Berkeley, 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[18] C. Celio, D. A. Patterson, and K. Asanović, "The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor," Technical Report UCB/EECS-2015-167, EECS Dept., Univ. of California, Berkeley, 2015. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-167.html

[19] S. Takamaeda-Yamazaki, "Pyverilog: A Python-Based Hardware Design Processing Toolkit for Verilog HDL," in *Applied Reconfigurable Computing*, K. Sano, D. Soudris, M. Hübner, and P. C. Diniz, Eds. Cham: Springer International Publishing, 2015, pp. 451-460.

[20] L. De Moura and N. Bjørner, "Z3: an efficient SMT solver," in *Proc. Theory and Practice of Software, 14th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, Berlin, Heidelberg: Springer-Verlag, 2008, pp. 337-340.

[21] R.-V. Software, "Spike RISC-V ISA Simulator," in *GitHub Repository*, 2019. [Online]. Available: https://github.com/riscv-software-src/riscv-isa-sim

[22] "cocotb: Coroutine based co-simulation library for writing VHDL and Verilog testbenches in Python," in *GitHub Repository*, 2013–2024. [Online]. Available: https://github.com/cocotb/cocotb

[23] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," *Proceedings of CAV*, pp. 24-40, 2010.

[24] M. Mann, A. Irfan, F. Lonsing, et al., "Pono: A Flexible and Extensible SMT-Based Model Checker," *CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II*

[25] A. Goel and K.A. Sakallah, "AVR: abstractly verifying reachability," *Proceedings of TACAS*, pp. 413-422, 2020.