HFL: Hardware Fuzzing Loop with Reinforcement Learning

Lichao Wu, Mohamadreza Rostami, Huimin Li, Ahmad-Reza Sadeghi
Technische Universität Darmstadt, Germany
{lichao.wu, mohamadreza.rostami, huimin.li, ahmad.sadeghi}@trust.tu-darmstadt.de

***Abstract*—As hardware systems grow increasingly complex, ensuring their security becomes more critical. This complexity often introduces difficult and costly vulnerabilities to address after fabrication. Traditional verification methods, such as formal and dynamic approaches, encounter limitations in scalability and efficiency when applied to complex hardware designs. While hardware fuzzing presents a promising solution for efficient and effective vulnerability detection, current methods face several challenges, including coverage saturation, long simulation times, and limited vulnerability detection capabilities.**

This paper introduces Hardware Fuzzing Loop (HFL), a novel fuzzing framework designed to address these limitations. We demonstrate that Long Short-Term Memory (LSTM), a machine learning model commonly used in natural language processing, can effectively capture the semantics of test cases and accurately predict hardware coverage. Building on this insight, we leverage reinforcement learning to optimize the test generation strategy dynamically within a hardware fuzzing loop. Our approach utilizes a multi-head LSTM to generate sophisticated RISC-V assembly instruction sequences, along with an LSTM-based predictor that evaluates the quality of these instructions. By dynamically interacting with the hardware, HFL efficiently explores complex instruction sequences with minimal fuzzing iterations, allowing it to uncover hard-to-detect vulnerabilities. We evaluated HFL on three RISC-V cores, and the results show that it achieves higher coverage using fewer than 1% of the test cases required by leading hardware fuzzers, effectively mitigating the issue of coverage saturation. Furthermore, HFL identified all known vulnerabilities in the tested systems and discovered four previously unknown high-severity issues, demonstrating its significant potential in improving hardware security assessments.

***Index Terms*—hardware fuzzing, RISC-V, reinforcement learning, natural language processing**

I. Introduction

Hardware security is critical for ensuring the integrity of computing systems, as vulnerabilities enable attackers to exploit specific flaws or manipulate inherent hardware functionalities, compromising sensitive data and system security. Addressing these vulnerabilities has become a pressing issue in hardware security. However, identifying and mitigating these threats in complex hardware systems remains challenging and resource-intensive. Even after a device undergoes rigorous, expert-driven security evaluations, it may not be entirely vulnerability-free.

Current approaches to mitigating hardware vulnerabilities, such as dynamic verification [1], [2] and formal verification [3]–[7], attempt to detect vulnerability with higher accuracy and lower human efforts. Dynamic verification employs runtime assertions within the hardware to check for security violations. However, it can only detect vulnerabilities post-fabrication, limiting its effectiveness in preemptively securing the design. Formal verification, though more precise, is hindered by significant manual effort and struggles to scale as system complexity grows. These limitations emphasize the need for more advanced and automated methodologies to ensure the security of hardware systems. Hardware fuzzing has become popular recently thanks to its capability to automate the discovery of vulnerabilities in hardware architectures [8]–[13]. Although hardware fuzzing has been successfully applied across multi-core systems with limited human involvement, existing works heavily rely on randomness with limited interaction with the Device Under Test (DUT), which restricts the exploration of untested hardware states, thus fail to uncover deeper vulnerabilities. Besides, hardware fuzzing can be time-consuming, as the DUT must execute many test cases.

We introduce a novel hardware fuzzing method, the Hardware Fuzzing Loop (HFL) to address these shortcomings. Unlike traditional approaches, HFL models the hardware behavior with machine learning techniques. Concretely, we show that a natural language processing technique, Long Short-Term Memory (LSTM) [14], can predict the hardware coverage with high accuracy based on the input instruction sequence, indicating its capability to understand the semantics of the instruction sequence and modeling the DUT behavior when executing these instructions. Based on this foundation, we use LSTM to build the instruction generator and predictor, then optimize models with real-time DUT feedback powered by reinforcement learning (RL). This iterative process guides the instruction generator in producing better test cases that improve overall hardware coverage and vulnerability detection efficiency. Our experimental results show that HFL significantly enhances hardware coverage and identifies more severe and sophisticated vulnerabilities than any existing works, offering a promising solution to the hardware security challenges faced by modern computing systems.

Our main contributions are:

1) We introduce a multi-head LSTM-based instruction generator capable of producing RISC-V instructions. Our generator efficiently generates instruction by allocating a model head to each instruction component. A deep understanding of the instruction sequence semantics allows the model to make more flexible and intelligent adjustments based on the DUT.
2) We introduce the hardware fuzzing loop (HFL) framework, a reinforcement learning-based scheme that fine-tunes the instruction generator in real time. The hardware

fuzzing loop is controlled by instruction mask and reset modules and guided by DUT feedback. The proposed framework encourages the instruction generator to generate high-coverage test cases while avoiding saturation.

3) To overcome the expensive and slow hardware simulation, We introduced a hardware coverage predictor to provide fast and reliable feedback for the generator without using hardware simulation for all of fuzzing testcases. We validate HFL on three RISC-V cores, RocketChip [15], Boom [16], and CVA6 [17], demonstrating its superiority over state-of-the-art fuzzing frameworks in hardware coverage and vulnerability detection.
4) Our approach identified a significant number of mismatches and revealed previously undiscovered vulnerabilities. Furthermore, our framework successfully detects all bugs and vulnerabilities identified by current hardware fuzzers and discovers *four* novel vulnerabilities with severe security consequences, which the CVA6 [17] developers have confirmed. We have requested CVE identifiers for these new vulnerabilities; They will be reported as soon as we receive them from MITRE [18].

The remainder of this paper is organized as follows. Section II provides essential background information. Section III reviews related work in the field. In Section IV, we present a high-level overview of the proposed HFL, followed by implementation details in Section V. Section VI benchmarks HFL against other fuzzing tools. Section VII discusses the vulnerabilities and bugs discovered by our approach. Finally, Section VIII summarizes our framework and outlines future research directions.

II. Preliminaries

A. Hardware Fuzzing

Hardware fuzzing is a promising technique for testing complex hardware designs thanks to its relatively low deployment costs and ability to expedite the verification process. The fuzzing process begins with generating random yet valid input stimuli. Mutation algorithms are then applied to modify these inputs, allowing the fuzzer to explore new states and behaviors of the DUT. The modified stimuli are fed into the DUT, and its behavior is continuously monitored. To identify potential security flaws or bugs, the DUT's output is compared to the expected results produced by a Golden Reference Model (GRM), which simulates the correct behavior of the hardware. Discrepancies between the DUT and GRM outputs signal potential bugs or vulnerabilities in the hardware. An overview of existing hardware fuzzing methods is presented in Section III.

B. Machine Learning

Machine learning is fundamental for solving complex problems in various domains, from pattern recognition to decision-making in dynamic environments. Natural language processing (NLP) is an important sub-branch of machine learning that allows computer programs to understand human language as it's spoken and written. Long short-term memory (LSTM) is one of the most widely used NLP models. It is a specialized form of recurrent neural network (RNN) that excels in capturing long-term dependencies in data sequences [14]. LSTM uses input, forget, and output gates to control the information flow, allowing the network to remember or discard data as needed, thus is well-suited for applications like language modeling [19] and time-series predictions [20].
Reinforcement Learning (RL) [21] is a machine learning paradigm that optimizes sequential decision-making. Instead of learning from labeled data, an RL agent interacts with its environment, choosing actions that maximize a cumulative reward over time. The agent learns through trial and error, receiving feedback through rewards or penalties, and continuously refines its decision-making policy. This allows the agent to adapt to complex environments, making RL particularly effective in robotics [22], game-playing [23], and automated control systems [24].

III. Related Work

State-of-the-art fuzzing frameworks primarily focus on fuzzing SoCs, CPUs, and standalone IP blocks. One notable approach is RFuzz [25], the first FPGA emulation-based hardware fuzzing technique, which generates inputs using multiplexer control signals combined with American Fuzzy Lop (AFL)-based [26] mutation functions. Similarly, Li et al. proposed a new metric, Full Multiplexer Toggle Coverage, to enhance hardware coverage performance [27]. In contrast, Trippel et al. [28] introduced a novel technique for fuzzing hardware-like software by fuzzing the hardware simulation binary instead of directly porting software fuzzers to hardware designs. More recent developments in hardware fuzzing have taken a domain-specific approach, eliminating the need to translate hardware into software. For instance, TheHuzz [8] simulates the RTL design of a processor using instruction binaries within Synopsys VCS [29], tracking code coverage through multiple metrics such as branch, condition, toggle, finite state machine (FSM), and functional coverage. DifuzzRTL [30] similarly generates instructions while gathering control register coverage to guide the fuzzing process. Building on this work, MorFuzz [31] enhances coverage by 4.4 times compared to DifuzzRTL by generating fuzzing seeds based on syntax and semantics while also using runtime feedback to mutate instructions. ProcessorFuzz [32] represents another concurrent effort, generating instructions and gathering coverage data from control and status registers. Meanwhile, HyPFuzz [9] leverages formal verification tools to guide the fuzzer into hard-to-reach design spaces, increasing coverage and exposing more vulnerabilities. SoC Fuzzer [10] takes a security-driven approach, guiding the fuzzing process based on security properties and a generic cost function to detect vulnerabilities in the DUT. Cascade [11] focuses on optimizing instruction execution efficiency by constructing long instruction sequences and eliminating control flow influences. It conducts the fuzzing process at the program level without relying on mutation strategies for guidance. However, despite their innovations, these methods still struggle to handle the complexity of input semantics and provide sufficient feedback from the DUT, which limits their overall effectiveness. Reinforcement learning has been applied across numerous domains and applications. In the context of hardware fuzzing, ChatFuzz [33] is the only known approach that uses reinforcement learning to guide the generation of fuzzing test cases. However, ChatFuzz generates binary-based test cases that suffer from weaker inter-instruction semantic connections compared to assembly-level tests, constraining the generator's ability to produce more effective and diverse test cases, and limiting overall performance.

IV. Our Hardware Fuzzing Loop (HFL)

This work proposes a novel hardware fuzzing framework, Hardware Fuzzing Loop (HFL), that integrates the NLP model's ability to process sequential data with reinforcement learning's (RL) capacity to optimize decision-making through interaction. This combination allows HFL to understands the semantics of the instruction sequence and generates better test cases based on feedback from the hardware coverage. We demonstrate the semantic-interpretation capability of LSTM in Section IV-C.

A high-level overview of HFL is presented in Fig. 1. HFL starts with test case generation, which is then post-processed (by blue blocks) and sent to the DUT. The reward assignment module processes the hardware coverage feedback returned by the DUT and updates the instruction generator based on the rewards received. Simultaneously, bug and vulnerability detection is conducted by comparing the execution traces between the DUT and the Golden Reference Model (GRM). Each component of this process is explained in detail in the following sections.

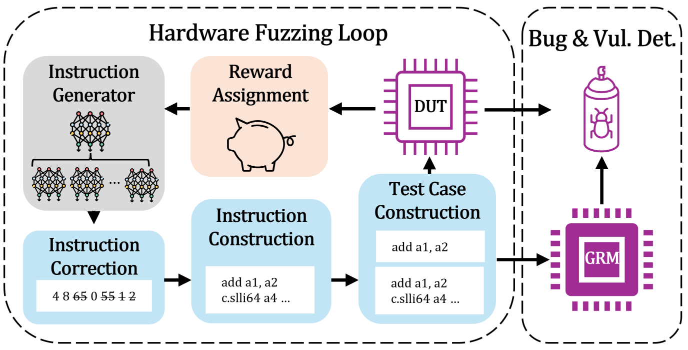


Fig. 1: A demonstration of the hardware-guided reinforcement learning framework.

A. *Test Case Generation and Reward Assignment*

The test case generation process comprises two main components: the instruction generator and post-processing modules. The instruction generator is designed to generate random instructions while continuously learning from experience to improve future test cases. To achieve this, we first analyze various RISC-V instructions, including its extensions. We observe that an instruction can involve up to four registers (e.g., `fnmsub.d fs4, fs9, ft5, fs9`), while some instructions may also include immediate values (e.g., `li t5, -84`) or addresses (e.g., `csrw 0x453, ra`).

To build an instruction generator capable of handling all types of RISC-V instructions, including the RISC-V ISA extensions, we introduce a multi-head LSTM model to manage these diverse instruction formats. Specifically, the LSTM functions as a feature extractor, processing instruction sequences to capture semantic information, which is then routed to the appropriate heads. The instruction generator comprises seven heads, each responsible for generating a different component of instruction: one for the opcode, four for the operand registers, one for the immediate data, and one for the address. Each head varies in output size according to the vocabulary of the specific element it predicts (e.g., 241 opcodes and 32 registers).

The instruction generator produces outputs based on the highest prediction probability from each head. However, since not all outputs are necessary for every instruction, we implement an instruction correction module to ensure the validity of the generated assembly instructions. This module begins by identifying the opcode, which defines the instruction format and determines which outputs are required to construct the instruction. The module generates two outcomes: 1) the selected instruction outputs, decoded into a valid instruction, and 2) an instruction mask, indicating which heads were used and which were not. This instruction mask is later utilized to fine-tune the instruction generator, as discussed in Section IV-B.

The selected outputs are decoded to form a valid instruction. Our instruction generation scheme ensures the correctness of generated instructions. Once sufficient instructions have been generated, the test construction module assembles the test case. Specifically, with N instructions, each test case includes one additional instruction, building on the previous test case. This method allows for a precise evaluation of the contribution of each instruction regarding the hardware coverage. Also, it ensures that later instructions do not overwrite the previous architectural state, which could trigger a bug. As a result, our approach detects bugs using significantly fewer test cases than previous methods, presented in Section VI and VII.

Once the DUT executes the instruction sequence, the hardware coverage is returned and serves as the basis for reward assignment, as defined in Eq. (1):

$$R = \alpha \times \text{hardware\_coverage} + r_{bonus}, \tag{1}$$

where r_{bonus} denotes the bonus reward (r_{bonus}), which is granted if the generated test case achieves the highest hardware coverage observed so far. r_{bonus} and hardware_coverage drive the generator to produce test cases that cover more hardware states, thereby increasing the likelihood of bug detection. The settings of α and r_{bonus} are detailed in Section V-B.

B. *Model Update with Hardware Feedback*

The instruction generator can produce random RISC-V instruction sequences but cannot reach certain hardware coverage requiring multiple instruction combinations. To address this issue, we integrate the instruction generator with DUT using Proximal Policy Optimization (PPO) [34], a well-known RL scheme. The framework is shown in Fig. 2.

During each iteration, the instruction generator creates a test case composed of RISC-V instructions. The predictor evaluates the generated test case; the advantage $\hat{A}_t$ is estimated to measures how well the new instruction sequence S_{t+1} performed relative to the baseline S_t:

$$\hat{A}_t = R_t + \gamma V(S_{t+1}) - V(S_t), \tag{2}$$

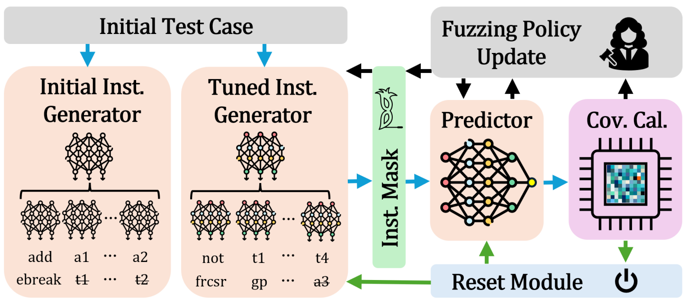


Fig. 2: Reinforcement learning framework for model updates.

where $V(S_t)$ is the predicted value of S_t; γ is the discount factor. The predictor parameterized by ϕ is trained by minimizing the mean squared error between predicted and observed returns, as shown in Eq. (3):

$$L^{\text{pred}}(\phi) = \mathbb{E}_t[(V(S_t;\phi) - (R_t + \gamma V(S_{t+1};\phi)))^2]. \tag{3}$$

Using the advantage $\hat{A}_t$, the instruction generator's parameters θ are updated by maximizing the clipped surrogate objective $L^{\text{gen}}(\theta)$:

$$\begin{aligned} L^{\text{gen}}(\theta) =& \mathbb{E}_t[\min(\frac{\pi_\theta(A_t|S_t)}{\pi_{\theta_{\text{old}}}(A_t|S_t)}\hat{A}_t, \\ & \text{clip}(\frac{\pi_\theta(A_t|S_t)}{\pi_{\theta_{\text{old}}}(A_t|S_t)}, 1-\epsilon, 1+\epsilon)\hat{A}_t)], \end{aligned} \tag{4}$$

where S_t and A_t denote the current instruction sequence and newly generated instruction, respectively; the fuzzing policy, represented by the instruction generator, is denoted by π_θ. The clipping threshold ϵ ensures the tuned instruction generator does not deviate significantly from the initial instruction generator $\pi_{\theta_{\text{old}}}$, ensuring learning stability. Following this RL scheme, the instruction generator evolves to maximize hardware coverage and strives for the r_{bonus} (Eq. (1)), which is awarded for achieving the highest coverage in comparison to previous tests. Following Eq. (4), actions with higher advantage estimates $\hat{A}_t$ are favored in future iterations, while those with lower $\hat{A}_t$ are deprioritized. Although this approach is standard in RL, it introduces a significant limitation in hardware fuzzing, which we call *the curse of exploitation*. Firstly, the probability distribution of opcodes, operands, and immediate values skews toward more common opcodes (e.g., `sub`) has a higher chance to be selected and get a reward than more complex ones (e.g., `fcvt.d.lu`). Aligned with the issue reported in [35], this leads to less frequent instructions disappearing from the generated instruction sequence, causing the fuzzer to generate repetitive instructions. A second issue stems from the fuzzing policy π. RL's goal of optimizing instruction combinations results in a stable policy π^*, which maximizes rewards but reduces exploration. The fuzzing policy gradually becomes biased toward actions that previously yielded high rewards, limiting its ability to explore new instruction combinations.

To overcome these issues, we introduce a two-part solution that involves instruction masking and dynamic fuzzing control.

Instruction Mask: Each head of the instruction generator predicts different instruction components, such as opcodes, operands, immediate values, and addresses. We introduce an instruction mask to balance the training of the instruction generator's heads, ensuring that only the active heads are updated in the current instruction. This prevents overfitting to specific instructions and encourages the generator to learn effectively across all instruction types. By balancing the updates to each head, the fuzzer maintains diversity in its instruction generation, which helps avoid repetitive instruction patterns and promotes greater hardware coverage.

Reset Module: We implement a reset module by tracking the maximum hardware coverage $C_{\max}$. If the coverage stagnates, we reinitialize the parameters θ and ϕ of both the instruction generator and predictor. This control mechanism forces the fuzzer to explore new action spaces, preventing it from being stuck in local optima. The predictor reset ensures it rewards newly discovered instruction combinations instead of reinforcing previously successful actions.

C. Case Study: Hardware Coverage Prediction with LSTM

Following the fuzzing scheme outlined before, the instruction generator determines the next instruction that maximizes hardware coverage based on the instructions executed previously. This requires the generator to comprehend the semantics of the input instruction sequence. However, gaining this understanding typically involves training with feedback from hardware simulations, which is both time-intensive and costly due to licensing restrictions. To address these limitations, we use LSTM to predict hardware coverage based on input assembly test cases. This model, the *hardware coverage predictor*, is evaluated using 830 000 test cases generated for the RocketChip [15]. The hardware coverage for each test case is represented as a bit string, where each bit corresponds to a coverage point.

We tokenize and encode the instruction sequence for the LSTM. For hardware coverage, which serves as the labels, we considered three hardware coverage metrics in this case study: conditional, line, and FSM coverage. We exclude *dead coverage points*—those that are always covered or never covered by any test case. Including these points would be redundant, as their outcomes are constant. Notably, more than 70% of the coverage points were identified as dead, and removing them significantly reduced the model's computational complexity. We trained the LSTM model for 200 epochs. An early stopping mechanism was implemented to speed up training: training was terminated if the model's accuracy did not improve for ten consecutive epochs. The dataset was split 90/10 for training and validation, respectively. The validation accuracy for each coverage point is shown in Fig. 3.

The hardware coverage predictor achieved high validation accuracy, with average accuracies of 94%, 94%, and 97% for conditional, line, and FSM coverage, respectively. We observed accuracy drops for specific coverage points, such as between indices 160 and 195 in the line coverage prediction (the middle graph), which we attribute to extreme class imbalances. In these cases, the model becomes biased toward specific predictions. Following Fig. 3, we conclude that the LSTM model effectively interprets the semantics of input test cases and predicts the corresponding hardware coverage. In addition, it demonstrates the model's ability to approximate the behavior of the DUT, providing a solid foundation of HFL.

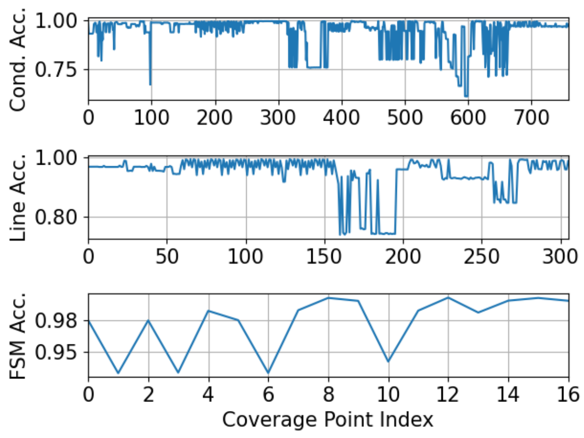


Fig. 3: Validation accuracy of hardware coverage predictor.

V. Implementation

A. Instruction Generator and Predictor

This work leverages LSTM for its ability to learn long-term dependencies in input sequences and forms the core of the instruction generator and the predictor. The instruction generator uses a two-layer LSTM with a hidden size of 256. The LSTM's output embeddings are fed into seven separate heads, each consisting of a hidden layer with 32 features. This multi-task learning approach allows the instruction generator to share features across different heads, improving robustness. We reuse the same LSTM model for the predictor with an additional output layer for the predictor, allowing it to understand the semantics of the input and, thus, score instructions effectively. The learning rate for both the instruction generator and predictor is set to 1e-4. While it is possible to use a single model with two outputs for both instruction generator and predictor, such a shared structure can lead to interference between tasks, potentially reducing performance [36].

B. Fuzzing Process

HFL leverages feedback from the DUT to fine-tune the instruction generator and predictor, resulting in improved test cases. The process begins with test case generation and reward assignment, as outlined in Section IV-A. In Eq. (1), the weight for hardware coverage (α) is set to 0.2, while R_{bonus} is 0.4. This balance ensures both exploration and exploitation are properly managed, with equal contribution from hardware coverage and bonus rewards. To further stabilize training, we normalize the rewards: this adjustment sharpens gradients for positive rewards and softens them for negative ones, promoting faster learning and smoother training convergence. γ and ϵ are set to 0.1 and 0.2, respectively.

The generated test cases are executed on the GRM and DUT, yielding two outputs: hardware coverage for training the generator and predictor and the architectural state for differential testing. Differential testing is a commonly used hardware fuzzing technique that compares execution traces between the RTL model of the DUT and the ISA GRM [8], [11], [33], [37]. However, these comparisons frequently result in numerous mismatches, duplicates, or false positives [37]. To address these, we aligned the device tree and boot ROM code for both the RTL models [15]–[17] and Spike [38], ensuring consistent initialization between them. To mitigate duplicate mismatches, we developed a signature extraction algorithm that generates a unique signature for each mismatch based on opcode, register values, and exception causes. By making the signature register independent, we ensure that different manifestations of the same bug (with varying sequences of register) are treated as a single issue, thus reducing duplicate reports, and minimizing the manual effort to detect bugs and vulnerabilities.

VI. Coverage Benchmark

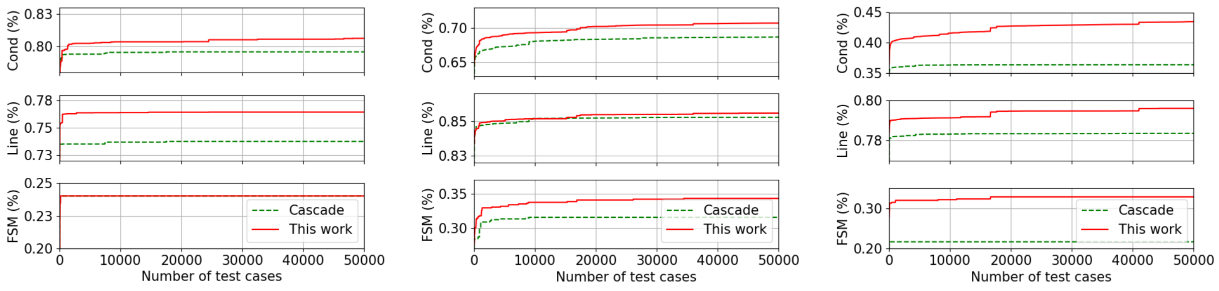
We benchmark HFL against four state-of-the-art fuzzers: DifuzzRTL [30], TheHuzz [8], ChatFuzz [33], and Cascade [11]. We perform a comprehensive coverage analysis using condition, line, and finite-state machine (FSM) metrics across three RISC-V cores: RocketChip [15], Boom [16], and CVA6 [17]. For the other fuzzers, due to space constraints, we focused on RocketChip using condition coverage metrics.

As illustrated in Fig. 4, HFL consistently outperforms Cascade across all cores and metrics, except for FSM coverage on RocketChip, where both fuzzers perform identically. A key advantage of HFL is its ability to continue increasing coverage with additional test cases (e.g., Figs. 4b and 4c) while the cascade reaches a plateau early. This reflects the ability of HFL to combine instructions and explore specific coverage points intelligently. Furthermore, when the fuzzer is stuck in a local optimum, the reset module triggers, promoting further exploration of untested hardware states.

HFL also achieves superior coverage compared to its counterparts on all the fuzzers on RocketChip using condition coverage metrics. The result plot is discarded due to space constraints. Aligned with CASCADE results shown in Fig. 4, the considered fuzzer saturates early, indicating the low quality of generated test cases. Even after generating up to 100K test cases, HFL maintains its coverage advantage, achieving comparable results to other fuzzers with less than 1% of the test cases. This shows the efficiency of HFL in exploring various hardware states, ultimately leading to better bug detection.

VII. Vulnerability Detection

V1: Cache Line Modification Leading to Processor Crash. In CVA6, modifying a cache line that contains the currently executing instruction can cause a processor crash. This vulnerability occurs when an adversary executes a store instruction that targets memory within the same cache line as the active instruction, disrupting cache coherency during a write-back operation. HFL generates a test case to trigger this issue by modifying the memory address at `0x800011FF`, which shares a cache line with the executing instruction. If the store is directed to a different cache line (e.g., `0x80001200`), the processor functions normally. The proof of concept code is shown in Listing 1.



(a) RocketChip (b) Boom (c) CVA6

Fig. 4: Coverage Benchmark between HFL and Cascade

Listing 1: Proof of concept code for triggering a crash via cache line modification

```
asm volatile("li t1, 0x800011FF");
asm volatile("li t0, 0x000000013");
asm volatile("sw t0, (t1)");
```

This behavior introduces a severe risk as it causes undefined behavior, leading to a processor crash and potential denial of service (DoS). Code running on the CVA6 core could be exploited by an attacker to intentionally trigger this vulnerability, leading to system instability (`CWE-1281`).

V2: Delay in Enforcing PMP Rules Leading to Unauthorized Access. In CVA6 cores, there is a delay in enforcing Physical Memory Protection (PMP) rules. When configuring PMP, the protection does not immediately apply to the next instruction but is delayed by 128 bits (addresses ending in four zero bits). This delay is caused by a microarchitectural issue in CVA6, where PMP rules fail to cover the next 128 bits after a rule is set. As a result, an attacker can access memory regions before the protection takes effect (CWE-1220). HFL generates a test case where PMP configuration should prevent access to a restricted address. However, the CVA6 core allows memory access for the next 128 bits, shown in Listing 2. This flaw introduces a security risk: An attacker can read data in PMP-protected regions, such as the Secure Boot ROM, before the protection is enforced.

Listing 2: Proof of concept code for delay in PMP enforcement

```
// t1 points to address of "li t3, 0x88" >> 2
asm volatile ("csrw pmpaddr0, t1");
asm volatile ("li t0, 0x89 "); // Read only
asm volatile ("csrw pmpcfg0, t0 ");
asm volatile ("li t3, 0x88"); // Should throw
    ↪ exception here!!
```

V3: Jump to Misaligned Address Fails to Trigger Exception. Jumps to misaligned instruction addresses without compressed extension support should trigger a misaligned fetch exception. However, in some versions of CVA6, attempting to jump to a misaligned address fails to raise this exception, allowing the program to continue execution without error (CWE-1281).

V4: Invalid NaN Inputs in Single-Precision Floating-Point FEQ Leading to Missing Exception Flag. According to the RISC-V ISA specification, executing Single-Precision Floating-Point FEQ with invalid NaN-boxed inputs—where the most significant 32 bits are not set to 1—should perform a quiet comparison. In this scenario, the invalid operation exception flag (NV flag in fcsr) should only be set if either input is a signaling NaN. However, in the CVA6 implementation, this flag is not triggered, resulting in incorrect handling of invalid NaN inputs. However, in CVA6, this flag is not set, leading to incorrect behavior in handling invalid NaN inputs (CWE-1281).

VIII. Conclusion

This work presents HFL, a powerful and highly efficient hardware fuzzer for hardware security testing. HFL integrates a fuzzer powered by natural language processing techniques. By dynamically interacting with the Device Under Test via reinforcement learning, HFL achieves a more extensive coverage of hardware state than existing works. HFL not only detected all previously known bugs on three RISC-V cores but also uncovered four new vulnerabilities, three of which are classified as high-severity at responsible disclosure. These vulnerabilities, which require complex sequences of instructions, are challenging to find with traditional fuzzing techniques. However, HFL excels at generating these complex instruction combinations by effectively understanding how different instructions interact. HFL delivers a highly efficient fuzzing process, making it a powerful tool for hardware security testing.

Acknowledgment

Our research work was partially funded by Intel's Scalable Assurance Program, Deutsche Forschungsgemeinschaft (DFG) – SFB 1119 – 236615297, the European Union under Horizon Europe Programme – Grant Agreement 101070537 – CrossCon, and the European Research Council under the ERC Programme - Grant 101055025 - HYDRANOS. This work does not in any way constitute an Intel endorsement of a product or supplier. Any opinions, findings, conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect those of Intel, the European Union, and the European Research Council.

References

[1] I. Wagner and V. Bertacco, "Engineering trust with semantic guardians," in *2007 Design, Automation & Test in Europe Conference & Exhibition*. IEEE, 2007, pp. 1–6.

[2] M. Hicks, C. Sturton, S. T. King, and J. M. Smith, "Specs: A lightweight runtime mechanism for protecting software from security-critical processor bugs," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2015, pp. 517–529.

[3] S. R. Sarangi, A. Tiwari, and J. Torrellas, "Phoenix: Detecting and recovering from permanent processor design bugs with programmable hardware," in *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. IEEE, 2006, pp. 26–37.

[4] C. Deutschbein and C. Sturton, "Mining security critical linear temporal logic specifications for processors," in *2018 19th International Workshop on Microprocessor and SOC Test and Verification (MTV)*. IEEE, 2018, pp. 18–23.

[5] B. Wile, J. Goss, and W. Roesner, *Comprehensive functional verification: The complete industry cycle*. Morgan Kaufmann, 2005.

[6] G. Dessouky, D. Gens, P. Haney, G. Persyn, A. Kanuparthi, H. Khattri, J. M. Fung, A.-R. Sadeghi, and J. Rajendran, "{HardFails}: insights into {software-exploitable} hardware bugs," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 213–230.

[7] E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani, "Model checking and the state explosion problem," in *LASER Summer School on Software Engineering*. Springer, 2011, pp. 1–30.

[8] 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 *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3219–3236.

[9] C. Chen, R. Kande, N. Nguyen, F. Andersen, A. Tyagi, A.-R. Sadeghi, and J. Rajendran, "{HyPFuzz}:{Formal-Assisted} processor fuzzing," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 1361–1378.

[10] M. M. Hossain, A. Vafaei, K. Z. Azar, F. Rahman, F. Farahmandi, and M. Tehranipoor, "Socfuzzer: Soc vulnerability detection using cost function enabled fuzz testing," in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2023, pp. 1–6.

[11] F. Solt, K. Ceesay-Seitz, and K. Razavi, "Cascade: Cpu fuzzing via intricate program generation," in *Proc. 33rd USENIX Secur. Symp*, 2024, pp. 1–18.

[12] P. Borkar, C. Chen, M. Rostami, N. Singh, R. Kande, A.-R. Sadeghi, C. Rebeiro, and J. Rajendran, "Whisperfuzz: White-box fuzzing for detecting and locating timing vulnerabilities in processors," 2024. [Online]. Available: https://arxiv.org/abs/2402.03704

[13] M. Rostami, S. Zeitouni, R. Kande, C. Chen, P. Mahmoody, J. Rajendran, and A.-R. Sadeghi, "Lost and found in speculation: Hybrid speculative vulnerability detection," in *61st ACM/IEEE Design Automation Conference (DAC '24), June 23–27, 2024, San Francisco, CA, USA*, ser. DAC '22, 2024, pp. 192–197.

[14] S. Hochreiter, "Long short-term memory," *Neural Computation MIT-Press*, 1997.

[15] A. et al., "The Rocket Chip Generator," no. UCB/EECS-2016-17, 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[16] J. Zhao, B. Korpan, A. Gonzalez, and K. Asanovic, "SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine," *4th Workshop on Computer Architecture Research with RISC-V*, 2020.

[17] F. Zaruba and L. Benini, "The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fdsoi technology," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 11, pp. 2629–2640, 2019.

[18] MITRE, "Hardware design cwes," https://cwe.mitre.org/data/definitions/1194.html, 2019.

[19] M. Sundermeyer, R. Schlüter, and H. Ney, "Lstm neural networks for language modeling." in *Interspeech*, vol. 2012, 2012, pp. 194–197.

[20] S. Siami-Namini, N. Tavakoli, and A. S. Namin, "The performance of lstm and bilstm in forecasting time series," in *2019 IEEE International conference on big data (Big Data)*. IEEE, 2019, pp. 3285–3292.

[21] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[22] J. Kober, J. A. Bagnell, and J. Peters, "Reinforcement learning in robotics: A survey," *The International Journal of Robotics Research*, vol. 32, no. 11, pp. 1238–1274, 2013.

[23] C. Berner, G. Brockman, B. Chan, V. Cheung, P. Dębiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse *et al.*, "Dota 2 with large scale deep reinforcement learning," *arXiv preprint arXiv:1912.06680*, 2019.

[24] B. R. Kiran, I. Sobh, V. Talpaert, P. Mannion, A. A. Al Sallab, S. Yogamani, and P. Pérez, "Deep reinforcement learning for autonomous driving: A survey," *IEEE Transactions on Intelligent Transportation Systems*, vol. 23, no. 6, pp. 4909–4926, 2021.

[25] K. Laeufer, J. Koenig, D. Kim, J. Bachrach, and K. Sen, "Rfuzz: Coverage-directed fuzz testing of rtl on fpgas," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2018, pp. 1–8.

[26] Google, "Americal fuzzy loop," https://github.com/google/AFL, 2019.

[27] T. Li, H. Zou, D. Luo, and W. Qu, "Symbolic simulation enhanced coverage-directed fuzz testing of rtl design," in *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2021, pp. 1–5.

[28] T. Trippel, K. G. Shin, A. Chernyakhovsky, G. Kelly, D. Rizzo, and M. Hicks, "Fuzzing hardware like software," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3237–3254.

[29] Synopsys, "Vcs, the industry's highest performance simulation solution," https://www.synopsys.com/verification/simulation/vcs.html.

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

[31] J. Xu, Y. Liu, S. He, H. Lin, Y. Zhou, and C. Wang, "{MorFuzz}: Fuzzing processor via runtime instruction morphing enhanced synchronizable co-simulation," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 1307–1324.

[32] S. Canakci, C. Rajapaksha, L. Delshadtehrani, A. Nataraja, M. B. Taylor, M. Egele, and A. Joshi, "Processorfuzz: Processor fuzzing with control and status registers guidance," in *2023 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2023, pp. 1–12.

[33] M. Rostami, M. Chilese, S. Zeitouni, R. Kande, J. Rajendran, and A.-R. Sadeghi, "Beyond random inputs: A novel ml-based hardware fuzzing," in *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2024, pp. 1–6.

[34] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[35] I. Shumailov, Z. Shumaylov, Y. Zhao, Y. Gal, N. Papernot, and R. Anderson, "The curse of recursion: Training on generated data makes models forget," *arXiv preprint arXiv:2305.17493*, 2023.

[36] M. Andrychowicz, A. Raichuk, P. Stańczyk, M. Orsini, S. Girgin, R. Marinier, L. Hussenot, M. Geist, O. Pietquin, M. Michalski *et al.*, "What matters for on-policy deep actor-critic methods? a large-scale study," in *International conference on learning representations*, 2021.

[37] M. Rostami, C. Chen, R. Kande, H. Li, J. Rajendran, and A.-R. Sadeghi, "Fuzzerfly effect: Hardware fuzzing for memory safety," *IEEE Security and Privacy*, vol. 22, no. 4, pp. 76–86, 2024.

[38] RISC-V, "Spike risc-v isa simulator," https://github.com/riscv-software-src/riscv-isa-sim.