ARTIFACT EVALUATED
usenix ASSOCIATION
AVAILABLE

GenHuzz: An Efficient Generative Hardware Fuzzer

Lichao Wu
Technical University of Darmstadt

Mohamadreza Rostami
Technical University of Darmstadt

Huimin Li
Technical University of Darmstadt

Jeyavijayan Rajendran
Texas A&M University

Ahmad-Reza Sadeghi
Technical University of Darmstadt

Abstract

Hardware security is crucial for ensuring trustworthy computing systems. However, the growing complexity of hardware designs has introduced new vulnerabilities that are challenging and expensive to address after fabrication. Hardware fuzz testing, particularly whitebox fuzzing, is promising for scalable and adaptable hardware vulnerability detection. Despite its potential, existing hardware fuzzers face significant challenges, including the complexity of input semantics, limited feedback utilization, and the need for extensive test cases.

To address these limitations, we propose GenHuzz, a novel white-box hardware fuzzing framework that reframes fuzzing as an optimization problem by optimizing the fuzzing policy to generate more subtle and effective test cases for vulnerability and bug detection. GenHuzz utilizes a language model-based fuzzer to intelligently generate RISC-V assembly instructions, which are then dynamically optimized through a Hardware-Guided Reinforcement Learning framework incorporating real-time feedback from the hardware. GenHuzz is uniquely capable of understanding and exploiting complex interdependencies between instructions, enabling the discovery of deeper bugs and vulnerabilities. Our evaluation of three RISC-V cores demonstrates that GenHuzz achieves significantly higher hardware coverage with fewer test cases than four state-of-the-art fuzzers. GenHuzz detects all known bugs reported in existing studies with fewer test cases. Furthermore, it uncovers 10 new vulnerabilities, 5 of which are the most severe hardware vulnerabilities ever detected by a hardware fuzzer targeting the same cores, with CVSS v3 severity scores exceeding 7.3 out of 10.

1 Introduction

Hardware is the cornerstone of trust in secure computing systems. Despite enforcing specific hardware countermeasures guided by established security standards [1, 2], the increasing complexity of hardware designs has led to functional bugs and security-critical vulnerabilities that are uncovered, which can potentially be exploited through sophisticated cross-layer attacks via, e.g., specific weaknesses in hardware implementations [3,4] or inherent functional features [5–7]. The Common Weakness Enumeration (CWE) identifies numerous vulnerabilities affecting hardware and software [8]. Addressing these issues after fabrication (post-silicon) can be prohibitively expensive [9]. Therefore, it is crucial to identify and address these vulnerabilities before fabrication (i.e., pre-silicon) to ensure the security of hardware systems and avoid the substantial costs associated with post-silicon patches.

To reduce human intervention in security assessment, researchers have developed methods such as formal verification [10–14], runtime detection (a.k.a. dynamic verification) [15, 16], information flow tracking [17–19], and hardware fuzzing [20, 21]. Among them, hardware fuzzing has emerged as a promising approach due to its scalability and adaptability to various designs. Inspired by the success of software fuzzing [22], researchers have extended its principles to hardware fuzzing, particularity in the *white-box model*, to identify vulnerabilities in hardware. Hardware fuzzing shows considerable promise by automating vulnerability and bug detection processes, particularly in testing across multiple processor cores [20, 21, 23–31].

Hardware fuzzing challenges. Despite the advantages mentioned above, hardware fuzzing faces several challenges. Firstly, the hardware fuzzer frequently overlooks the *complexity of input semantics*. Indeed, a fuzzer should understand the interdependencies between instructions to trigger vulnerabilities and bugs that hide more profoundly in the DUT (Device Under Test). Unfortunately, existing approaches solely focus on basic instruction mutations [20, 24, 32]. This oversight means that vulnerabilities requiring a combination of multiple instructions often remain undetected. Secondly, existing fuzzers have *low feedback utilization* from DUT. Although conventional hardware fuzzing methods have a feedback loop from the DUT to the fuzzer, this feedback typically serves only to discard ineffective test cases rather than to inform the generation of better ones. This limited feedback causes hard-

ware coverage to plateau quickly, leaving some vulnerabilities unexplored if they are associated with unreachable coverage points. Thirdly, the *portability problem* affects the generality of hardware fuzzing. Although existing works explore RISC-V ISA, they typically need to customize the fuzzer for a specific target implementation, limiting their applicability across the broader spectrum of hardware security assessments. Lastly, hardware fuzzing requires a large number of test cases to detect specific vulnerabilities, leading to *efficiency issues*. Considering the substantial overhead involved in the hardware simulation and execution of the test cases and the time constraints typically imposed during security evaluations, the vulnerability detection capability of hardware fuzzing becomes limited. We discuss the shortcomings of existing white-box hardware fuzzing in Section 9.

Our goals and contributions: This work introduces a novel coverage-based white-box hardware fuzzing framework, GenHuzz, which effectively addresses the challenges mentioned above. Unlike existing fuzzers, GenHuzz employs a novel approach by framing hardware fuzzing as an optimization problem. This is done by optimizing the *fuzzing policy* for test case generation using real-time hardware feedback. We have developed a custom language model-based fuzzer to understand and generate RISC-V assembly code. The fuzzing policy is dynamically adjusted through reinforcement learning guided by feedback from the hardware, a method we term Hardware-Guided Reinforcement Learning (HGRL). The integration of HGRL allows GenHuzz to go significantly beyond random instruction generation; it comprehends the semantics of the assembly language and discovers subtle but crucial inter-relation among the several instructions that may inherit a vulnerability, an insight that other fuzzers cannot provide. Consequently, the augmented test case with complex data and control flows significantly advances hardware coverage and vulnerability detection. Our main contributions are:

1. We introduce a novel language model-based white-box fuzzer, GenHuzz, capable of accurately generating RISC-V assembly. Unlike conventional fuzzers that rely on mutations and instruction rules to create new test cases, GenHuzz comprehends the semantics of the instructions, enabling greater vulnerability detection capability to the specifics of DUTs.

2. We propose a novel learning scheme, Hardware-Guided Reinforcement Learning (HGRL), which fine-tunes the fuzzing policy based on real-time hardware coverage feedback. This approach guides the fuzzer in producing high-coverage test cases while avoiding the pitfalls of local optima.

3. We demonstrate the flexibility of GenHuzz across RISC-V processors. We experimentally validate GenHuzz on three different RISC-V cores without the need to adapt GenHuzz. The results outperform state-of-the-art fuzzing frameworks, reaching high hardware coverage with 1% of test cases compared with state-of-the-art.

4. GenHuzz successfully identified 10 previously unreported bugs and vulnerabilities in the tested cores (RocketChip [33], Boom [34], and CVA6 [35]). Among them, GenHuzz uncovered five new security vulnerabilities with high severity scores exceeding 7.3 (out of 10) according to the Common Vulnerability Scoring System Version 3.1 (CVSS V3) [36]. In addition, we also tested older versions of the benchmark cores and successfully triggered all previously reported bugs and vulnerabilities in previous work [20, 24, 25, 27, 28]. The discovered vulnerabilities and bugs are detailed in Section 6.

The remainder of this paper is structured as follows. We provide the necessary background information in Section 2. In Section 3, we describe the design of GenHuzz in detail; the implementation is introduced in Section 4. Section 5 evaluates the hardware coverage of GenHuzz and benchmark with state-of-the-art fuzzers. Section 6 details the vulnerabilities and bugs detected by the GenHuzz. Section 7 assesses the performance of GenHuzz and the influence of key components. Section 8 provides a discussion on the proposed method. Section 9 elaborates on related works. Finally, we summarize this work in Section 10.

2 Preliminaries

2.1 Fuzzing

Fuzzing has gained significant traction due to its low deployment costs and expedited verification process for testing complex designs. Fuzzing primarily involves generating and mutating random test cases, monitoring the Device Under Test (DUT), and analyzing for vulnerabilities or vulnerabilities. Traditional fuzzing begins with generating random test cases or input stimuli. To efficiently cover the DUT's state space, most fuzzers use mutation algorithms to create new test cases, differentiating it from dynamic verification. The input stimuli are then fed into the DUT, where its status is monitored, and any crashes during the fuzzing period are recorded. The monitored output is analyzed for vulnerabilities, and the DUT executes test inputs until a crash is recorded. Software fuzzers analyze these crashes to detect vulnerabilities; hardware fuzzing, on the other hand, verifies the expected outcome from the DUT against assertions or the output from the Golden Reference Model (GRM), which generates the expected responses.
Fuzzing techniques can be broadly classified into three types depending on the available information regarding the DUT: black-box fuzzing, grey-box fuzzing, and white-box fuzzing [21, 37]. For instance, white-box fuzzers have comprehensive knowledge of the DUT and peripheral information

about data flow, control flow, data format, protocols, and high-level architecture. Coverage-based white-box fuzzing aims to achieve maximum code coverage through feedback engines. Various coverage metrics exist in hardware, including finite state machine (FSM), line, condition, and MUX toggle coverage. During fuzzing, input seeds are stored and passed to the mutation engine, which performs mutation operations to generate multiple input seeds. Then, coverage reports are extracted based on the input provided to the DUT and fed back to the mutation engine. The mutation engine discards uninteresting seeds from the input pool and further mutates the interesting input seeds to generate a new set. The design is simulated on inputs with these seeds, and any potential crashes are saved for vulnerability analysis.

2.2 Reinforcement Learning

Reinforcement learning (RL) [38] is a machine learning (ML) technique that trains software to make decisions to achieve the most optimal results. It mimics the trial-and-error learning process that humans use to achieve their goals. Actions that work towards the objective are reinforced, while actions that detract from the objective are ignored. RL differs from other forms of machine learning, such as supervised and unsupervised learning [39], in that the agent cannot access labeled data or explicit rules.

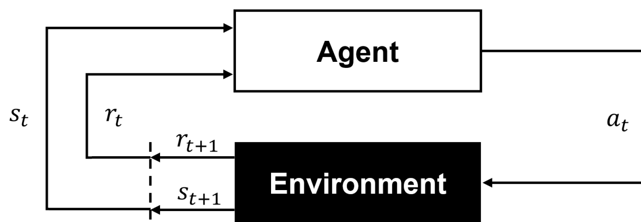


Figure 1: A demonstration of a generic RL environment.

In general, RL consists of five basic blocks: 1) agent: the learner and the decision maker; 2) environment: the physical world in which the agent operates; 3) state (s): the current situation of the agent; 4) reward (r): feedback from the environment; 5) policy (π): the method to map agent's state to actions. A graphical representation can be found in Fig. 1. An agent makes observations from the environment called states. In time step t, the agent receives state s_t from the environment and acts by following a specific policy π or transition probability Pr by taking action a_t. The action is rewarded with r_t by applying a reward function f to the state-action pair (s_t, a_t); the state is updated to s_{t+1}. When the agent reaches a predetermined terminal state, the environment sends a terminate signal to the agent. From there on out, a new sequence of states, actions, and rewards begins.

2.3 Language Model

Language models are pivotal in natural language processing (NLP), functioning as systems that can understand, generate, and manipulate human language. These models are trained on vast amounts of text data to predict the likelihood of a sequence of words, enabling them to perform various tasks such as translation, summarization, and question-answering. Among the most advanced language models are those based on transformer architectures [40], notably the Generative Pre-trained Transformer (GPT) [41], and Bidirectional Encoder Representations from Transformers (BERT) [42]. For instance, GPT models, such as GPT-3 and GPT-4, are designed using a unidirectional approach. The model predicts the next word in a sentence based on the preceding words. This is achieved through a stack of transformer decoder layers comprising multi-head self-attention mechanisms and feed-forward neural networks. The GPT models are pre-trained on a diverse corpus of text and can be fine-tuned for specific tasks with relatively small amounts of task-specific data.

3 GenHuzz

GenHuzz employs a teacher-student framework, where the Design Under Test (DUT) acts as the teacher, guiding the fuzzer to generate more effective test cases that explore a broader state space of the DUT. Aligned with existing works, this paper focuses on white-box fuzzing of the open-source cores with the standard RISC-V ISA, as they publicly provide RTL and detailed hardware coverages, in contrast to closed-source processors from, such as ARM and Intel.

A high-level overview of this method is depicted in Fig. 2, which comprises three main stages: 1) fuzzer initialization, 2) hardware-guided reinforcement learning (HGRL), and 3) vulnerability detection. A detailed explanation of each stage is provided in subsequent sections. In the first stage, we initialize a fuzzer based on a language model (top-left graph) to grasp the fundamental rules of assembly instructions. This allows it to generate highly accurate and adaptable random assembly instructions. The initialized fuzzer actively interacts with the DUT in the second stage through the HGRL framework. As these interactions increase, the fuzzer learns the fuzzing policy by understanding the semantics of the test cases and the entanglement between instructions, generating test cases that potentially cover more hardware states. The feedback from the DUT is driven by a reward system designed to accurately reflect the hardware's behavior and encourage the fuzzer to generate test cases that may uncover previously unexplored vulnerabilities. In the final stage, execution logs from the DUT and the Golden Reference Model (GRM) are compared to detect bugs. GRM serves as the ground truth for correct execution. It executes RISC-V instructions according to the ISA specification, ensuring its output can be used as a

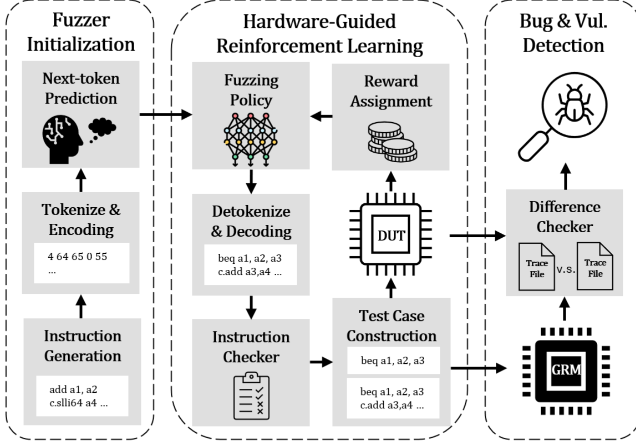


Figure 2: An Overview of the GenHuzz.

reference to compare with DUT outputs [1].

3.1 Fuzzer Initialization

We conceptualize a test case composed of multiple instructions as a coherent sentence communicated to the Device Under Test (DUT). The proposed fuzzer leverages a language model for its exceptional performance in natural language processing tasks; the implementation details are provided in Section 4. Fundamentally, an effective fuzzer should comprehend both *intra-instruction semantics* (i.e., how to generate valid individual instructions) and *inter-instruction semantics* (i.e., how to strategically combine multiple instructions to uncover more complex bugs and vulnerabilities). GenHuzz is initialized with an understanding of intra-instruction semantics. The inter-instruction semantics are learned dynamically through interaction with the hardware, detailed in Section 3.3. The initialization of the fuzzer begins with data generation. We generate a set of N *assembly* instructions, denoted as $I = I_1, I_2, \ldots, I_N$. Assembly language strengthens the semantic relationships between instructions, allowing for more meaningful combinations that can reveal subtle hardware vulnerabilities. The generated instructions are concatenated into a single sequence, $\mathcal{D}$, using a specific separator, `SEP`:

$$\mathcal{D} = I_1 \texttt{ SEP } I_2 \texttt{ SEP } I_3 \texttt{ SEP } \ldots I_N. \tag{1}$$

The `SEP` operator flags each instruction's beginning and end, thus helping the fuzzer better understand the intra-instruction semantics. Next, the combined instructions dataset is tokenized and encoded to prepare it for training the model. Each instruction I_i is tokenized into a sequence of tokens $T_i = (t_{i,1}, t_{i,2}, \ldots, t_{i,k_i})$, where k_i is the number of tokens in instruction I_i. The tokenized dataset is then represented as $\mathcal{T} = T_1\ T_{\texttt{SEP}}\ T_2\ T_{\texttt{SEP}}\ \ldots\ T_{\texttt{SEP}}\ T_N$.

[1] Since the RISC-V GRM we use [43] does not support speculative or out-of-order processing, GenHuzz can only detect static bugs, aligned with literature [20, 24, 25, 27, 28].

The tokens are subsequently encoded into numerical representations suitable for model input. We define a vocabulary V and a mapping function to the word embedding $f : V \to \mathbb{R}^d$, where d is the dimension of the embedding space uniquely representing each encoded token, which are high-dimensional vectors optimized to capture semantic similarities between words. The encoded dataset $\mathcal{E}$ is thus represented as:

$$\mathcal{E} = f(T_1)\ f(T_{\texttt{SEP}})\ f(T_2)\ f(T_{\texttt{SEP}})\ \ldots\ f(T_{\texttt{SEP}})\ f(T_N). \tag{2}$$

The encoded data is then used to initialize the fuzzer. The training pipeline is detailed in Appendix A. The training objective of the fuzzer is to predict the next token in the sequence based on the preceding tokens so that the fuzzer can generate new instructions based on previous instructions. Formally, let $\mathbf{x} = (\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_M)$ be the sequence of embedded tokens, where M is the total number of tokens in the dataset $\mathcal{E}$. The fuzzer is trained to predict the next token $\mathbf{x}_{t+1}$ given the sequence $(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_t)$. The model parameters θ are optimized by minimizing the cross-entropy loss $\mathcal{L}$:

$$\mathcal{L}(\theta) = -\sum_{t=1}^{M-1} \log P_\theta(\mathbf{x}_{t+1} \mid \mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_t), \tag{3}$$

where the probability P_θ is parameterized by the trainable parameter θ of the fuzzer. Finally, we utilize gradient descent to minimize the loss function:

$$\theta^{(k+1)} = \theta^{(k)} - \eta \nabla_\theta \mathcal{L}(\theta^{(k)}), \tag{4}$$

where η is the learning rate and k denotes the iteration step. This process iterates until the number of generated tokens equals a preset maximum token number. We use this value to constrain the number of instructions per test case.

3.2 State Transition and Reward Function

As mentioned in Section 3.1, the fuzzer takes action by predicting the next token based on previous tokens. The reward should be assigned to each action (token) for the reinforcement learning tasks as the action changes the state. However, assigning a reward when an action is an element of an instruction, e.g., an operand, is more challenging than the common reinforcement learning tasks in which each action immediately gets a reward. There are three reasons: 1) only a complete instruction can be rewarded; thus, the reward for an action will be delayed; 2) not all instructions will be executed. For instance, a branch instruction could skip multiple instructions. The unexecuted instructions do not contribute to the hardware coverage, but they are essential for generating the upcoming instructions; 3) even a syntactically correct instruction could cause exceptions when executing on the DUT. We cannot simply assign zero or negative rewards to these instructions, as they also contribute to the fuzzing process.
To assign proper rewards to each action, we first identify possible instruction status when executing on the DUT, thus

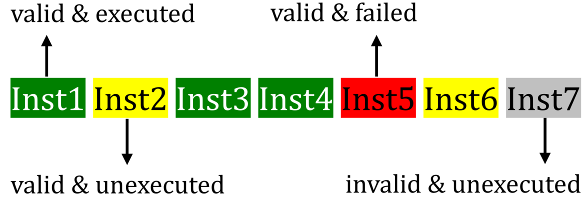


Figure 3: Possible instruction status in a test case.

defining the state transition for a generated test case. Using a test case shown in Fig. 3 as an example, it consists of seven instructions. The DUT throws an exception in the fifth instruction (inst5), highlighted in red, and the following instructions (inst6 and inst7) are unexecuted. In this case, there are four instruction statuses in total[2]: 1) *Valid & Executed*: an instruction is valid and is executed by DUT; 2) *Valid & Unexecuted*: an instruction is valid but unexecuted by DUT due to, for instance, branch instructions or failed instructions before the current instruction; 3) *Valid & Failed*: an instruction is valid and is executed by DUT, but DUT throws an expectation, and the test case execution is stopped; 4) *Invalid & Unexecuted*: an instruction is invalid (violates the ISA specification) and, naturally, not executed by DUT.

Due to the complexity of the instruction status, we separate the state transition into three steps, detailed in Algorithm 1. First, as shown in lines L2 to L5, the instruction token is generated without any restrictions until the *done* condition is fulfilled, realized by a token number threshold. As mentioned, we use this threshold to control the number of instructions that form a test case. Unlike other hardware fuzzers that terminate the test case generation process once an illegal instruction is identified, these instructions are considered part of our test case. Although these instructions are excluded from the hardware execution, they become a learning experience that gives feedback to the fuzzer so that the fuzzer is enforced to generate valid instruction while involved based on the hardware feedback. Next, the post-processing begins with separating the generated token array into individual instructions with `SEP`. Each instruction is checked for its validity in L8. Finally, valid instructions are concatenated and simulated by the golden reference model (GRM), which returns the index of executed instructions.

The instruction status forms the basis of the reward assignment, shown in Eq. (5):

$$R = r_{valid} + \alpha * hardware_coverage + r_{bonus}, \quad (5)$$

where r_{valid} denoted the reward for the valid and invalid instructions, α represents the weights of the hardware coverage. The last two terms on the right of the equation are only assigned when the instruction is valid and executed. As mentioned, r_{valid} is essential for our framework as it ensures the fuzzer generates valid instructions during fine-tuning. The involvement of the *hardware_coverage* is a natural process as we want the fuzzer to generate test cases with higher coverage, thus leading to a higher probability of vulnerability detection. The realization of the *hardware_coverage* calculation is presented in Section 4.1, relies on the access to the RTL code. Finally, we involve a bonus reward r_{bonus}, assigned if the generated test case leads to the highest hardware coverage compared to all previously tested coverage. r_{bonus} further pushes the fuzzer to generate high-quality test cases. Our experiment in Section 7 shows that without r_{valid} and r_{bonus}, the fuzzer works unstable and quickly loses the dynamic instruction generation capability after a few learning iterations.

In this work, a test case that contains N instructions is separated into N sub-test cases, with each sub-test case concatenating a new instruction in sequence. This implementation leads to an accurate R evaluation and ensures that the upcoming instructions do not overwrite the architectural state of a triggered bug. Consequently, we detect bugs with significantly fewer test cases than existing works, presented in Section 6.

[2]One may wonder about the existence of the invalid & executed instructions. This instruction status does not exist in our framework, as the generated assembly will first be compiled into the binary and then sent to the hardware. An invalid instruction will fail the compilation process.

Algorithm 1 State Transition and Identification

Require: $env, fuzzer, GRM$
1: $obs \leftarrow$ `reset`(env)
2: **while** not $done$ **do**
3: $\quad action \leftarrow fuzzer(obs)$
4: $\quad next_obs, done \leftarrow$ `step`$(env, action)$
5: $\quad obs \leftarrow next_obs$
6: $instructions \leftarrow$ `sperate`(obs)
7: **for** $inst$ **in** $instructions$ **do**
8: $\quad valid \leftarrow$ `verify`$(inst)$
9: $\quad$ **if** $valid$ **then**
10: $\quad\quad valid_instructions \leftarrow$ `append`$(inst)$
11: $executed_instruction \leftarrow GRM(valid_instructions)$

3.3 Hardware-Guided Reinforcement Learning

The fuzzer initialized in Section 3.1 can be employed for fuzzing tasks by generating random assembly instructions. However, its lack of interaction with the Device Under Test (DUT) limits its ability to detect specific hardware vulnerabilities requiring multiple instruction combinations. To address this limitation, we bridge the fuzzer-DUT interaction using Proximal Policy Optimization (PPO), a model-free reinforcement learning (RL) algorithm. Applying PPO directly to hardware fuzzing poses significant challenges. Firstly, conventional RL agents typically manage simple states, such as sequences of binary decisions, and are not equipped to handle complex scenarios involving assembly instructions. Furthermore, as discussed in Section 2.2, RL methods, in general, are designed to optimize a policy that enables an agent to complete a task efficiently. This objective does not align with

the goals of hardware fuzzing: an effective fuzzing policy that generates a test case with high hardware coverage or identifies a hardware vulnerability becomes obsolete after execution. Reapplying the same policy would be redundant, as it has already been tested and evaluated.

To overcome these challenges, we introduce the Hardware-Guided Reinforcement Learning (HGRL) framework, illustrated in Fig.4, which dynamically adapts the fuzzer during the fuzzing process. The HGRL framework consists of four fundamental components: a *fuzzer* that generates assembly instructions (described in Section 3.1), a *scorer* that evaluates the actions of the fuzzer by estimating the value of being in a specific state or taking a particular action (detailed in Section 3.2), a *reset module* that balances the exploration and exploitation of the fuzzer, and a *DUT* that provides hardware coverage feedback to refine the fuzzer and scorer iteratively.

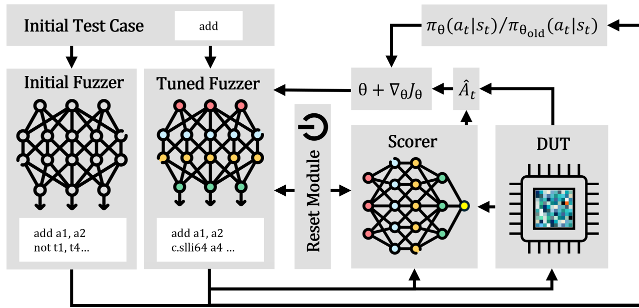


Figure 4: Hardware-Guided Reinforcement Learning.

Formally, let the state at time step t be s_t, the generated instruction elements such as opcode and operand (a.k.a. action) be a_t, and the coverage feedback (a.k.a. reward) provided by the DUT be r_t (see Eq. (5)). The HGRL objective is to maximize the expected cumulative reward over time following the Bellman Equation [44]:

$$J(\theta) = \mathbb{E}_\theta[\sum_{t=0}^{T} \gamma^t r_t],\ 0 \leq \gamma \leq 1, \tag{6}$$

where γ is the discount factor prioritizing immediate rewards over future rewards; T is the time horizon that defines the total number of time steps considered for the cumulative reward. Eq. (5) allows the fuzzer to "look forward": its goal is not only maximizing the reward with the current generated instructions, but the current generated instruction should also help boost the reward for the upcoming instructions. This characteristic is beneficial for archiving coverage that requires a combination of multiple instructions. Based on Eq. (5), the fuzzer will evolve to generate more valid and executed test cases that maximize *hareward_coverage* and try to get the r_{bonus}.

HGRL involves multiple iterations of fuzzer/scorer-DUT interaction. Based on the initial input, the fuzzer first generates assembly instructions to form a test case in each iteration. The DUT executes the test case and provides coverage feedback through rewards r_t. The current fuzzing policy π is evaluated by computing the advantage estimates $\hat{A}_t$:

$$\hat{A}_t = r_t + \gamma V_\phi(s_{t+1}) - V_\phi(s_t), \tag{7}$$

where $V_\phi(s_t)$ represents the value estimate of the state s_t outputted by the scorer parameterized by ϕ, and γ is the discount factor that determines how future rewards are weighted relative to immediate rewards. Eq. 7 estimates the relative advantage of taking a particular action in the given state s_t, compared to the expected baseline $V_\phi(s_t)$. In this work, instead of directly using the DUT as a scorer or building a scorer from scratch, we reuse the initialized fuzzer with one additional output layer for the value prediction. Indeed, a scorer who can evaluate the assembly input should understand its semantics. By reusing the fuzzer structure and weight, we avoid learning a value function from scratch. The scorer is trained to estimate the expected value from a given state s_t. The scorer, represented by the value function $V_\phi(s_t)$, is updated by minimizing the mean squared error between the predicted values and the observed returns, shown Eq. (8). This ensures that the value function accurately reflects the expected return from any given state.

$$L^{\text{scorer}}(\phi) = \mathbb{E}_t[(V_\phi(s_t) - (r_t + \gamma V_\phi(s_{t+1})))^2]. \tag{8}$$

$\hat{A}_t$ is used to update the policy followed by the fuzzer. Concretely, the fuzzer parameters θ are updated by maximizing the clipped surrogate objective $L^{\text{fuzzer}}(\theta)$.

$$\begin{aligned} L^{\text{fuzzer}}(\theta) =& \mathbb{E}_t[\min(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}\hat{A}_t, \\ & \text{clip}(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}, 1-\varepsilon, 1+\varepsilon)\hat{A}_t)], \end{aligned} \tag{9}$$

where π_θ and $\pi_{\theta_{\text{old}}}$ represent the current policy and the previous policy before the fuzzer update, respectively. For simplicity, we use π_θ and π interchangeably throughout this paper. The hyperparameter ε controls the clipping range to ensure the stability of policy updates. The clipping mechanism restricts the deviation of the new policy from the old one by taking the minimum of the clipped and unclipped objective values, thereby maintaining stability during fuzzer updates.

According to Eq. (9), actions with a higher advantage estimate, $\hat{A}_t$, are more likely to be selected in future iterations. Although this approach is standard to guide the policy toward actions that yield higher rewards, it presents significant limitations when applied to hardware fuzzing. First, the fuzzing process tends to converge prematurely, leading to saturation in hardware coverage. This phenomenon arises from the action generation process, where a_t are sampled from a probability distribution generated by the fuzzer's policy $\pi_\theta(a_t|s_t)$ based on the current state s_t. The probability distribution over actions is defined as:

$$\pi_\theta(a_t|s_t) = \text{softmax}(F_\theta(s_t)), \tag{10}$$

where $F_\theta(\cdot)$ outputs a vector of logits representing the likelihood of each action. Indeed, certain opcodes and operands could be sampled more frequently. Consequently, those with lower sampling probabilities, denoted as a'_t, are less likely to receive rewards. This creates a negative feedback loop where these instructions eventually disappear from the action space:

$$\lim_{t \to \infty} P(a'_t) \approx 0. \quad (11)$$

As a result, the fuzzer tends to repeatedly generate specific instructions, limiting the diversity of test cases. This behavior is similar to issues observed in training large language models (LLMs) with LLM-generated content, where skewed sampling leads to performance degradation [45]. Although the contexts differ, the underlying problem of reduced diversity is common, reducing effectiveness in both scenarios.

The second limitation stems from the fuzzing policy π itself. RL inherently seeks to optimize the test case generation process by creating a stable policy π^* that maximizes the expected reward (Eq. 6). However, this stability can lead to suboptimal exploration. Once the policy π^* reaches a point where certain hardware coverage or specific vulnerabilities are triggered, it becomes biased toward actions that have previously resulted in high rewards. This results in the policy repeatedly generating similar test cases, reducing the likelihood of exploring new instruction combinations that might reveal additional vulnerabilities. Finally, π converges to a narrow region of the action space $\mathcal{A}_{\text{optimal}}$, where:

$$\pi^*(a_t \in \mathcal{A}_{\text{optimal}} | s_t) \approx 1. \quad (12)$$

When looking at the entropy of the action distribution $H(\pi_\theta(a_t|s_t)) = -\sum_{a_t} \pi_\theta(a_t|s_t) \log \pi_\theta(a_t|s_t)$, which measures the diversity of chosen actions. π^* decreases entropy, repeatedly favoring specific opcodes and operands.

To address these issues, we introduce a *reset module* that dynamically balances exploration and exploitation by monitoring entropy and hardware coverage. We track the maximum hardware coverage $C_{\max}$ achieved over time. If coverage stops to improve, it suggests that the framework is trapped in a local optimum and failing to explore new, potentially rewarding test cases. Then, the reset module is activated to restore the parameters θ and ϕ of both the fuzzer and the scorer to their initial states:

$$\theta \leftarrow \theta_{\text{init}};\ \phi \leftarrow \phi_{\text{init}}. \quad (13)$$

The reset module forces the fuzzer to explore the action space anew rather than relying on previously successful actions. Simultaneously, resetting the scorer allows it to focus on newly discovered vulnerabilities rather than existing ones. This dynamic balance between exploration and exploitation ensures that the fuzzer is not trapped in local optima, leading to more comprehensive hardware coverage and the discovery of previously undetected bugs and vulnerabilities.

3.4 Bug and Vulnerability Detection

We employ differential testing, a commonly used approach in the hardware fuzzing framework, to define a crash triggered by generated test cases. The detailed implementation is presented in Section 4.1. This technique involves running the same test case on both the RTL (Register-Transfer Level) model of the DUT and the Instruction Set Architecture (ISA) golden reference model, followed by a comparison of their execution traces [20, 21, 25, 27]. However, discrepancies between these two models frequently lead to numerous mismatches [21], many of which are duplicates. Consequently, after several hours of fuzzing, many of these mismatches are identified as *false positives* or *duplicate mismatches*, reducing the fuzzing process's effectiveness. To address these two challenges, we have developed two heuristic algorithms, which we will describe in the following two sections. The differential testing process is *fully automated* within our framework. Specifically, it is capable of 1) executing the same test cases on the GRM, 2) collecting and parsing the execution traces for both the core system and the GRM, 3) identifying mismatches between the execution traces of the core system and the GRM, and 4) applying developed heuristics to the detected mismatches to minimize repetitive and unrelated cases. While manual verification is necessary to determine if a vulnerability triggers a detected mismatch, our framework offers a significant overhead reduction compared to advanced techniques [20, 24, 25] that do not incorporate these heuristics.

3.4.1 False Positives

Our analysis of the most recent hardware fuzzing literature identified two primary causes that can lead to *False Positives* when comparing the ISA model and RTL traces: 1) *Differences in the device tree*: the device tree describes the hardware components and their configurations in a system [46]. Differences in the device tree between the RTL and ISA reference models can lead to discrepancies in execution traces. These discrepancies occur because the RTL model might interpret or access hardware resources differently from the ISA model due to mismatched hardware configurations, leading to mismatches that are falsely interpreted as bugs or vulnerabilities. 2) *Differences in boot ROM code*: the boot ROM code initializes the hardware and loads the test case, operating system, or firmware. Variations in the boot ROM code between the RTL and ISA reference models can result in different system states or initialization sequences. These differences can produce mismatched traces during the comparison process, which are falsely flagged as a bug or vulnerability. For instance, if the RTL model's boot ROM code changes the privilege mode to user mode before resuming the execution of the test case, but the ISA reference model does not change the privilege mode and remains in supervisor mode, the resulting execution traces will differ. This difference in privilege mode can lead to discrepancies in handling instructions and address

access, causing mismatches incorrectly identified as bugs or vulnerabilities by the differential testing process. To address these two primary causes, during the fuzzing, we fixed the device tree and boot ROM code for both RTL and ISA reference models to ensure consistency in the device tree and boot ROM between the RTL models [33–35] and Spike [43].

3.4.2 Duplicate Mismatches

Different fuzzers triggered the same bug or vulnerability multiple times, leading to a mismatch report each time. We provided a signature extraction algorithm to avoid duplicate mismatch reports in our mismatch detector mechanism. This algorithm generates a unique signature for each mismatch based on the opcode, register values, and exception cause. We did not include the register's name to keep the signature register independent since some bugs or vulnerabilities have the same root cause, but they trigger with register sequences. Using these signatures, our tool reports only one instance of each unique mismatch, reducing the manual work developers require. We generate a unique signature for each mismatch containing the instructions, source and destination register values, and the exception cause value. These details are extracted by analyzing execution traces. Memory addresses and values are skipped, as this information is redundantly captured within the registers during execution.

4 Implementation

GenHuzz leverages a GPT-based language model as both a fuzzer and scorer. The original GPT-2 model, with 1.5 billion parameters and a 50 257-word vocabulary trained on eight million web pages, uses Byte-Pair Encoding (BPE) tokenization. However, BPE splits opcodes into subwords, increasing the risk of invalid instructions; the already small RISC-V vocabulary, around 500 unique opcodes and operands, minimizes the need for vocabulary size compression with BPE. Consequently, we adopt a word-level encoding strategy that assigns each opcode, operand, and immediate value a unique token. This approach preserves the validity of opcodes and operands, requires less aggressive vocabulary reduction, and simplifies training by shortening the tokenized length of each instruction. Since assembly instructions are far simpler than human languages like English, we further reduce the complexity by employing a model roughly ten times smaller than GPT-2 with only approximately 12 million parameters. Our customized model incorporates hardware feedback while significantly reducing computational overhead.
We prepare five million randomly generated instructions as the training dataset, containing the basic RISC-V 32 and 64-bit ISA and their extensions, including integer multiplication and division instructions, single-precision floating-point instructions, atomic instructions, compressed instructions, and machine-level instructions. We train the model with 50 000 epochs with a learning rate of 1e-5, which takes around one hour with a single NVIDIA A6000 GPU. Once the fuzzer is trained, the same model is copied to build a scorer with one additional layer as the output layer. Using a shared structure for fuzzer and scorer could lead to a worse performance due to the interference between objectives [47]. Since the scorer still needs the training effort to estimate the action value reliably, we use a larger learning rate for the scorer (5e-5) than the actor (1e-5) to balance the training effort.

4.1 Fuzzing Process

The HGRL framework bridges the gap between the fuzzer/scorer and DUT. The goal of the fuzzer is to generate actions that form a better test case; the scorer evaluates the generated action, including its intermediate reward and (discounted) future rewards. The HGRL framework starts with the state transition and reward assignment, detailed in Section 3.2. Concretely, in Eq. (5), r_{valid} is assigned with -0.7 and 0.1 for the invalid and valid instruction tokens, respectively. The *hardware_coverage* metric is computed by parsing and analyzing various hardware behavior coverage metrics from Register-Transfer Level (RTL) code. These metrics include finite state machine (FSM) coverage, condition coverage, and line coverage. The coverage data is generated during the simulation using Synopsys VCS [48], allowing us to assess how thoroughly the hardware design has been exercised. FSM coverage measures how well the states and transitions of finite state machines within the design are tested. Condition coverage assesses the evaluation of all possible logical conditions, while line coverage determines the extent to which the lines of code in the RTL have been executed during simulation. Together, these metrics provide a comprehensive view of the effectiveness of the test cases in validating the hardware design. When an instruction is valid and executed, the instruction token gets an additional reward from the hardware coverage and a reward bonus R_{bonus}. The corresponding weight actor α equals 0.2; R_{bonus} is assigned with 0.4. Intuitively, we want the last two factors of the reward function, namely $\alpha * hardware_coverage$ and R_{bonus}, to have a balanced contribution to the reward to control the exploration and exploitation of the model. Furthermore, we normalize the reward for the straightforward reasoning: when the fuzzer performs rather badly, it receives much worse than good rewards. Normalization makes the gradient steeper for (puts more weight on) the good rewards and shallower for (puts less weight on) the bad rewards. Besides, making their distribution consistent helps stabilize training.
The GRM and DUT execute the generated test case; register values have been randomized to increase the possibilities of covering edge cases. The output is twofold: the hardware coverage is feedback to the fuzzer and scorer for the training. The execution traces are fed to a mismatch detector for vulnerability detection. To implement the vulnerability de-

tection component of our framework, we utilized Synopsys VCS and the Spike simulator [43] to perform simulations for both RTL and the Spike, which serves as the reference model for RISC-V ISA simulations. Synopsys VCS generates hardware simulation traces that detail the values of modified registers and memory transactions for each executed instruction. In contrast, the Spike simulator provides reference traces, which indicate the expected values of registers and memory locations after each instruction is executed when running a RISC-V binary file (e.g., an ELF file) according to the RISC-V ISA. Therefore, any discrepancies between the values of registers or memory transactions, such as differences in register values, memory addresses, or memory data, could indicate a bug or vulnerability.

We developed an automated mismatch detection tool designed to parse execution trace files from various RISC-V cores, including RocketChip [33], Boom [34], and CVA6 [35], with Spike [43] serving as the RISC-V ISA reference. By analyzing these execution traces, the tool extracts state changes for each test case at the granularity of individual instructions. Specifically, for each instruction in a fuzzing test case, the resulting CPU state changes are identified and recorded. To ensure reusability and ease of integration across different cores, our tool employs a unified data structure. This structure organizes each test case as an ordered list of instruction trace objects, where each object encapsulates the state changes of architecturally visible CPU components resulting from the execution of the corresponding instruction. This unified approach simplifies the process of comparing execution traces across cores. The workflow of the tool begins by parsing the execution traces and storing them in the unified data structure. Subsequently, it compares the state changes caused by each instruction in the core's execution trace against the corresponding trace from Spike, which serves as the GRM. Any deviations from the ISA [49, 50], as defined by Spike [43], are identified and reported as mismatches.

5 Coverage Evaluation

As a coverage-based white-box fuzzing method, GenHuzz aims to maximize the exploration of hardware states, thereby increasing the likelihood of detecting bugs and vulnerabilities. In this section, we evaluate GenHuzz compared to four state-of-the-art fuzzers: Cascade [27], DifuzzRTL [28], TheHuzz [20], and ChatFuzz [25]. Given that Cascade is the most recent fuzzers, we perform a comprehensive coverage analysis using condition, line, and FSM metrics across three RISC-V cores: RocketChip [33], Boom [34], and CVA6 [35]. For the other fuzzers, due to space constraints, our benchmarking focuses on RocketChip with condition coverage metrics.

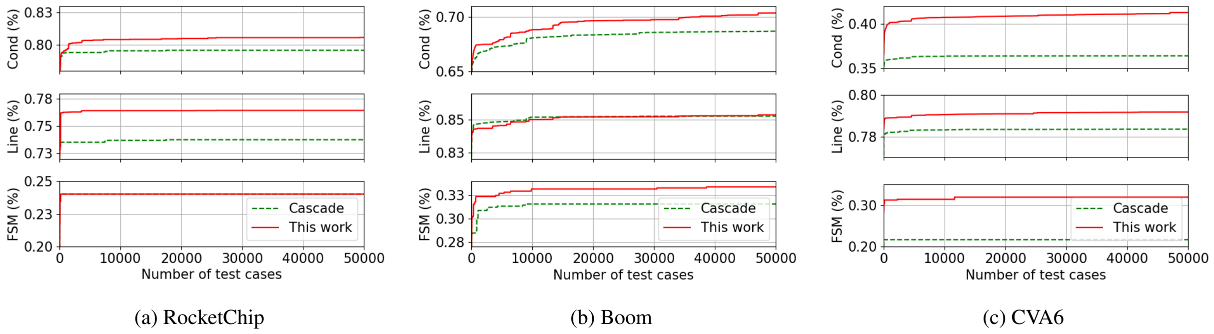
As depicted in Fig. 5, except for the FSM metric on RocketChip (the bottom figure on Fig. 5a) where GenHuzz and Cascade perform identically, GenHuzz consistently outperforms Cascade across all cores and evaluation metrics. Notably, the coverage metric of GenHuzz continues to increase with an increasing number of test cases (e.g., Fig. 5b), whereas the coverage for Cascade plateaus after a relatively small number of test cases. Indeed, it highlights the critical role of the HGRL scheme that integrates reinforcement learning and a language model, enabling intelligent instruction combinations to target specific coverage points. Moreover, when the fuzzing policy converges to a local optimum, GenHuzz's reset module activates to encourage exploration of previously untested hardware states, further extending its effectiveness.

Subsequently, we benchmark the performance of all considered fuzzers on RocketChip using condition coverage metrics, with Cascade included for completeness. As illustrated in Fig. 6, GenHuzz significantly outperforms all other fuzzers. The coverage for DifuzzRTL, TheHuzz and ChatFuzz quickly saturates, indicating limited exploration capacity. In contrast, GenHuzz maintains its superior coverage even with a substantial increase in test cases (up to 100K). Remarkably, GenHuzz achieves comparable coverage to these fuzzers using only 1% of the test cases, demonstrating its ability to efficiently explore diverse hardware states. This efficiency directly enhances bug and vulnerability detection, highlighting GenHuzz as a powerful tool for hardware security evaluation.

6 Detected Vulnerabilities and Bugs

GenHuzz identifies ten new vulnerabilities and bugs in the tested cores that have not been previously reported. Additionally, to assess the effectiveness of GenHuzz, we tested it on older versions of the benchmark cores to determine if it could detect bugs that were already reported in previous studies [20, 24, 25, 27, 28]. GenHuzz successfully triggered all of the previously reported bugs and vulnerabilities with only 50 000 test cases, indicating its strong capability in vulnerability and bug detections.

Table 1 shows all the newly discovered vulnerabilities and the most significant known vulnerabilities detected by GenHuzz. Five of the 10 newly discovered vulnerabilities have a severity score greater than 7.3 out of 10, according to the CVSS V3 [36]. This indicates that these vulnerabilities pose serious security risks to the benchmark cores. Furthermore, triggering these vulnerabilities requires carefully crafting and combining multiple instructions, often more than *two* and sometimes up to *six* instructions. It is highly unlikely (or practically impossible) for previous fuzzers, which rely on random instruction generation [20, 24, 27, 28], to generate such complex instruction sequences. This is because two challenging requirements must be met: 1) the generated instructions must have both data and control flow entanglement, e.g., subsequent instructions must use data loaded by one instruction, and 2) the instructions must be semantically intertwined, e.g., configuring Physical Memory Protection (PMP) settings and PMP address CSR registers within the same test case. However, by learning the inter-instruction semantics of assembly language



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

Figure 5: Coverage Benchmark between GenHuzz and Cascade

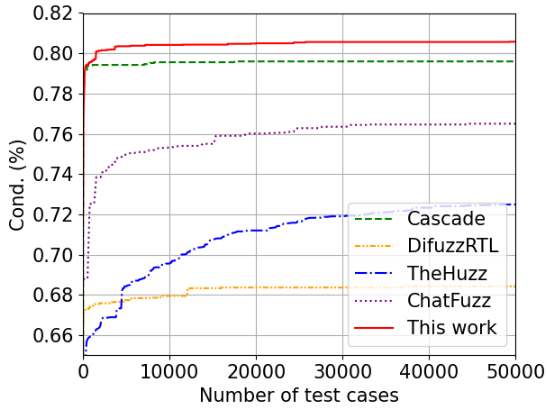


Figure 6: Coverage benchmark on the RocketChip.

through hardware feedback, GenHuzz generates test cases with integrated data and control flow, utilizing informative feedback to the model.

The following section provides detailed explanations and discusses potential exploitation scenarios for three of the most complex (which require a minimum of *four* instructions to trigger the vulnerability successfully) and critical vulnerabilities detected by GenHuzz. Due to the page limit, the descriptions of other detected novel bugs and vulnerabilities are provided in Appendix B.

PMP Configuration Leading to Bypass of Memory Protection (V_1). In the RISC-V architecture, PMP is designed to provide fine-grained memory protection for both user and privileged modes. PMP allows for configuring access permissions and address-matching modes for various memory regions to enforce security policies and prevent unauthorized access. In CVA6, the granularity of the PMP is determined by setting the address range to four bytes, as evidenced by isting 1, which is implemented based on the standard defined by RISC-V ISA specification [50].

Here, register `a3` is set to `0x003FFFFFFFFFFFF`, indicating that the least significant bit is set to `0`, thus confirming a

Listing 1: Code snippets for reading minimun PMP granularity on CVA6.

```
1 csrw pmpcfg0, x0
2 addi a3,x0,-1
3 csrw pmpaddr0, a3
4 csrr a3, pmpaddr0 // a3=0x003FFFFFFFFFFFFF
```

four-byte granularity for PMP [50]. However, if the PMP address matching mode is set to `0x2` (indicating Naturally Aligned four-byte regions `NA4`), the core automatically resets the PMP configuration register to `0x0`, effectively deactivating the memory protection mechanism within a security-critical section, thereby enabling the attacker to perform unauthorized read, write, and execute operations on the memory.

GenHuzz generates a test case that attempts to configure PMP using the unsupported `NA4` mode. The test case first configures the PMP address and then sets the PMP configuration register. However, the vulnerability detection module identified that the PMP configuration was not maintained as intended because setting the `NA4` mode in the CVA6 core results in the core resetting the PMP configuration to `0x0`, leaving the memory region unprotected. This discrepancy does not occur in the Spike RISC-V ISA simulator, where the same sequence of instructions creates a protected region and throws a `Load Access Fault` exception in case of access attempts. The sequence of instructions demonstrating this vulnerability is shown in Listing 2.

The implementation of the PMP in CVA6 demonstrates two critical flaws. First, there is no support for the `NA4` mode, and any attempt to configure it results in the complete disabling of memory protection. Second, this behavior introduces a significant security risk as it allows the bypassing of PMP-based protection entirely. Consequently, code running on the CVA6 core (whether in user, kernel, or firmware) may operate without the intended memory protections, leading to potential information leakage and privilege escalation vulnerabilities, i.e., `CWE-1281` and `CWE-1220`, in multi-threaded systems. The behavior of the CVA6 core under these conditions needs hard-

Listing 2: Proof of concept code snippets for bypassing PMP protection on CVA6.

```
1 auipc a3, 0x0       // a3 = [PC]
2 addi a3, a3, 0x1c   // a3 = [protected address]
      ↪  = [PC + 0x1c]
3 addi t1, x0, 0x2    // t1 = 0x2
4 srl t1, a3, t1      // t1 = a3 >>  0x2
5 csrw pmpaddr0, t1   // pmpaddr0 = t1
6 addi t0, x0, 0x94   // t0 = 0x94
7 csrw pmpcfg0, t0    // pmpcfg0 = t0
8 auipc a3, 0x0       // a3 = [protected address]
      ↪  = [PC + 0x1c]
9 lbu a4, 0(a3)
```

ware revision to enforce correct PMP behavior.

Improper Handling of Load-Reserved/Store-Conditional Instructions (V_8). In RISC-V architecture, the Load-Reserved (LR.W, LR.D) and Store-Conditional (SC.W, SC.D) instructions are designed to facilitate atomic read-modify-write operations in a multi-threaded or multi-processor environment. These instructions help implement synchronization mechanisms, such as locks or semaphores, to ensure that critical code sections are executed atomically, preventing race conditions and ensuring data integrity. An implementation can allocate an arbitrary subset of the memory space during each Load-Reserved operation, and multiple LR reservations can coexist concurrently for a single hardware thread (hart). The success of a Store-Conditional operation is contingent upon the absence of external accesses by other harts to the designated address between the SC.D instruction and the most recent LR.D instruction that reserved the same address within the hart. Notably, this LR.D instruction might have referenced a different address, but it reserved the memory subset that includes the address designated by the SC [49, 50].

Listing 3: Proof of concept code snippets to trigger and break LR.D and SC.W conditonal store on Boom and Rocket.

```
1 auipc   t0, 0xff // t0 = PC
2 c.lui   t1, 0x2 // t1 = 0x2000
3 lr.d    t3, (t0) // Reserve on [t0]
4 sd      t1, 512(t0) //
5 sc.d    t3, t1, (t0) // Store Conditional on [
      ↪ t0]
```

However, GenHuzz successfully generated a test case that intelligently reserves an address using the LR instruction. Following this, GenHuzz, guided by the inferred semantics of the RISC-V ISA specification [49, 50], generates a store instruction targeting a random address within the program's address space. It then issues a conditional store instruction (SC) to the same reserved address. This carefully constructed sequence of instructions demonstrates GenHuzz's ability to understand and apply several key considerations: 1) the semantics of RISC-V assembly language, ensuring that LR and SC instructions are ordered correctly and related within a program, 2) data and control flow entanglement, where both LR and SC share the same address operand, and 3) the use of randomness combined with strategic decision-making in instruction generation, such as inserting a standard store instruction. Upon executing this test case, our vulnerability detection module identified that the Store-Conditional (SC) operation failed because the reservation was broken by the effect of a normal store operation to a random address. The ISA implementations in both Boom and Rocket seem to register reservations for the entire physical memory range. This approach leads to several issues; most notably, it creates significant overhead since any store operation to an arbitrary address can disrupt the reservation. As a result, achieving a single successful Load-Reserved and Store-Conditional operation may require numerous attempts, particularly in multi-threaded environments. This inefficiency can lead to potential vulnerabilities such as information leakage and denial of service, specifically CWE-1281 and CWE-1421. Listing 3 provides an example sequence of instructions to replicate this behavior.

Improper Handling of PMP Execute-Only Memory Regions(V_9). In the RISC-V architecture, the Physical Memory Protection (PMP) mechanism provides fine-grained control over memory access permissions, allowing regions to be configured with Read (R), Write (W), and Execute (X) permissions. According to the RISC-V privileged specification [50], an execute-only region should have permissions set as R=0, W=0, X=1, allowing code execution within this region while preventing any load or store operations. The combination R=0 and W=1 is specifically reserved and should not occur. However, in the BOOM core implementation, setting a region to be execute-only (R=0, W=0, X=1) incorrectly triggers an Instruction Access Fault when attempting to execute an instruction within that region. This behavior deviates from the RISC-V ISA specification [50], which expects instructions in an execute-only region to execute without fault. The issue can be reproduced with the code snippet in Listing 4.

Listing 4: Proof of concept code snippets to trigger improper Handling of PMP execute-only memory regions on BOOM.

```
1  auipc t0,0x0     // t0 = PC
2  addi t0,t0,0x43 // t0 = PC + 0x43
3  addi t1,x0,0x2   // t1 = 0x2
4  srl t1,a4,t1
5  addi t2,x0,0x94 // Set R=0, W=0, X=1
6  csrw pmpaddr0, t1
7  csrw pmpcfg0, t2
8  ... // v-- Mem offset = 0x43 --v
9  auipc a4, 0x0 // @Exception Instruction Access
      ↪  Fault
10 lbu a4,0(a4)
```

In the BOOM core, executing the Listing 4 results in an Instruction Access Fault. This inconsistency suggests a flaw in the PMP implementation, as other RISC-V cores like Rocket [33], CVA6 [35], and the Spike [43] simulator correctly allow execution within execute-only regions without

raising an exception. This incorrect behavior in the BOOM core could lead to unexpected behavior, i.e., CWE-1281, potentially affecting system stability and the correct execution of software that relies on execute-only regions for code protection and security.

7 Performance Evaluation

In this section, we investigate GenHuzz 's performance by evaluating three key aspects: the stability of the fuzzing process, the necessity of the reset module, and the impact of different reward assignments. Aligned with Fig. 6, we show experimental results on RocketChip with condition coverage metric in Fig. 7.

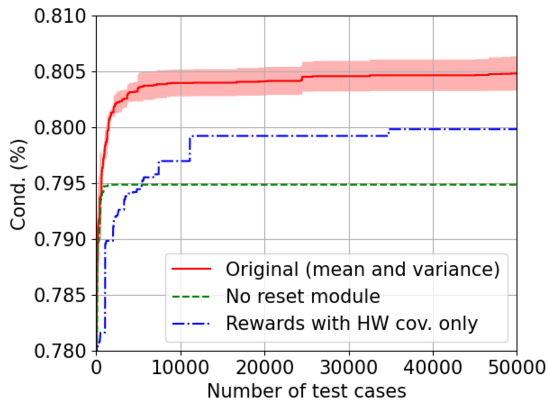


Figure 7: Performance evaluation of GenHuzz.

Stability of the Fuzzer. We repeat the fuzzing process ten times under different random seeds to validate that GenHuzz 's results remain stable despite inherent randomness. Each run begins with a unique initial condition, and we compare the resulting coverage trends and final coverage values. As shown in Fig. 7 (the mean and the standard deviation are in red and pink, respectively), GenHuzz consistently converges to high coverage levels regardless of the starting seed. Moreover, the lower bound consistently outperforms all other fuzzers (see Fig. 6), demonstrating its robust performance. Indeed, three key mechanisms foster this stability: 1) the HGRL framework constrains policy updates, preventing drastic shifts and maintaining smoother training dynamics; 2) the hardware coverage metric directly guides the instruction generator's fine-tuning, ensuring a focus on producing more effective test cases; and 3) the reset module actively monitors progress, intervening to prevent the model from settling into local optima. These design choices ensure that GenHuzz 's RL-language model-driven approach is inherently stable.

The Need for the Reset Module. GenHuzz incorporates a reset module to break out of local optima. This module is activated dynamically when the fuzzing process begins to stagnate. We conduct an ablation study to assess its effectiveness by disabling the reset module and comparing the results against the original implementation. As shown in the green line in Fig. 7, coverage growth plateaus prematurely without the reset module. In contrast, with the reset module engaged, GenHuzz demonstrates sustained coverage improvement over time. This finding confirms that the reset module is crucial in encouraging broader state exploration and preventing the fuzzing process from getting stuck in suboptimal policy.

Impact of Reward Assignment. The reward structure in GenHuzz is designed to guide the fuzzer toward reaching significant hardware states while ensuring the validity and diversity of generated test cases. To evaluate the impact of different reward components on performance, we conducted experiments using GenHuzz with only the hardware coverage reward, excluding the additional terms r_{valid} and r_{bonus} from Eq. 5. As shown in Fig. 7, this simplified reward structure still allows GenHuzz to improve coverage. However, the rate of improvement and the final coverage achieved are both lower compared to the comprehensive reward setup. The absence of r_{valid} often results in the fuzzer generating syntactically incorrect instructions, rendering the test cases invalid. While the reset module can mitigate this issue by reinitializing the fuzzer, the lack of r_{bonus} leads to another challenge: the generation of repetitive instructions. This phenomenon, known as reward hacking [51], occurs when the fuzzing policy maximizes the reward without fulfilling the intended objective. Incorporating r_{bonus} addresses this issue by encouraging the fuzzer to explore new states, resulting in better overall performance. These findings highlight that additional incentives for generating valid and diverse test cases, beyond merely maximizing coverage, are crucial for achieving richer state exploration and superior outcomes.

8 Discussion

GenHuzz is a novel hardware fuzzing framework that combines reinforcement learning (RL) and language models (LM) to address the challenge of generating test cases that can trigger deeper vulnerabilities. While RL has been explored in software fuzzing, hardware fuzzing presents unique challenges, such as tighter coupling between instructions and states, deterministic timing, and a more complex state space.

Impact of RL and LM Integration. Unlike conventional RL-based fuzzers, which rely primarily on adaptive search policies without deeper semantic insights, GenHuzz integrates RL with LM to generate contextually meaningful instruction sequences. By enriching the search process with a model that understands the structure and semantics of the code, GenHuzz can systematically identify coverage gaps and detect nuanced vulnerabilities. This semantic-driven strategy, commonly called semantic-aware fuzzing [52], allows GenHuzz to uncover vulnerabilities often missed by random or mutation-based fuzzing methods.

ID	Core	Bug Description	New	CVSS	CWE
$\mathbf{V_1}$	CVA6	Bypassing PMP Memory Protection in `NA4` Mode	✔	9.3	1281
$\mathbf{V_2}$	CVA6	Incorrect Exception Handling for Misaligned Load-Reserved	✔	7.3	1298
$\mathbf{V_3}$	CVA6	Incorrect `tval` Register Value for Breakpoint Exceptions	✔	6.8	1281
$\mathbf{V_4}$	CVA6	Incorrect `NaN` Boxing in FP `fsqrt.s` Instruction	✔	6.8	1281
$\mathbf{V_5}$	CVA6	Incorrect `NaN` Boxing in Single-Precision FP Division Instructions	✔	6.8	1281
$\mathbf{V_6}$	CVA6	Missing exceptions for some illegal instructions	✗	4.0	1281
$\mathbf{V_7}$	CVA6	Incorrect Setting of `NX` Flag for FP Instructions	✗	5.1	1281
$\mathbf{V_8}$	BOOM	Any store-related instruction breaks the `LR` and `SC`	✔	8.5	1281
$\mathbf{V_9}$	BOOM	Improper Handling of Execute-Only Memory Regions in PMP	✔	8.5	1281
$\mathbf{V_{10}}$	BOOM	No exception for invalid `rm` in single-precision operation	✔	4.4	1281
$\mathbf{V_{11}}$	BOOM	Improper Zeroisation of `minstret` CSR	✗	6.5	1239
$\mathbf{V_{12}}$	RocketChip	Any store-related instruction breaks the `LR` and `SC`	✔	8.5	1281
$\mathbf{V_{13}}$	RocketChip	No exception for invalid `rm` in single-precision operation	✔	4.4	1281
$\mathbf{V_{14}}$	RocketChip	Incorrect `minstret` value for after `EBREAK`	✗	3.3	1281

Table 1: Detected Vulnerabilities and Bugs (The Maximum CVSS Score is 10). CWE (Common Weakness Enumeration) is a standardized list of software and hardware vulnerability categories. Since detected bugs and vulnerabilities are inside the hardware, the CWE column only contains hardware CWEs. The corresponding Common Vulnerabilities and Exposures (CVE) numbers are detailed in the Open Science section.

Computation Efficiency. The computational efficiency of a fuzzer directly influences its overall performance. In GenHuzz, the main overhead arises from 1) test case generation (Section 3.2), 2) HGRL processing (Section 3.3), and 3) instrumentation. Since GenHuzz uses the same DUT as prior work, instrumentation time remains the same across fuzzers. The HGRL module imposes negligible overhead, completing tuning in under a second. Therefore, we assess GenHuzz 's efficiency primarily via test case generation time. Additionally, computational resources significantly affect fuzzing efficiency. Although GenHuzz executes largely on a GPU, we measure GenHuzz's CPU and GPU performance against state-of-the-art fuzzers for a fair comparison.

On an AMD EPYC 9684X CPU, GenHuzz demonstrates an execution time of 4.46 seconds per test case while achieving a significantly reduced execution time of 1.58 seconds on an A6000 GPU. Although the CPU overhead is marginally higher than that of state-of-the-art fuzzers such as CASCADE (2.06 seconds) and TheHuzz (2.47 seconds), this discrepancy arises from the fact that machine learning models are inherently optimized for efficient execution on GPUs. Consequently, the CPU overhead observed in GenHuzz is slightly larger. The additional overhead arises from the real-time tuning mechanism of GenHuzz (see Fig. 4), which restricts high parallelization during the generation of test cases. When fully utilizing parallelization in input generation, GenHuzz achieves a throughput of more than 10 000 test cases per second. Additionally, even in a non-parallelized configuration, GenHuzz demonstrates superior efficiency by effectively exploring hardware states, achieving comparable coverage to state-of-the-art fuzzers while using only 1% of the test cases. Furthermore, it shows effectiveness in uncovering complex vulnerabilities that involve multiple instructions. Integrating LLM with HGRL optimizes the process by reducing the test case number to achieve comprehensive coverage, thus minimizing the computational cost of hardware fuzzing.

Adaptability. One key strength of GenHuzz is its adaptability to almost any RISC-V implementation, enabled by RISC-V's open nature and the abundance of modifiable RTL (Register Transfer Level) designs. For GenHuzz (or any white-box fuzzing framework) to be transferable to ARM and x86 architectures, three key requirements must be met. First, there must be open-source RTL designs for the target ARM and x86 cores. These designs should be sufficiently detailed and functional to enable accurate simulation and vulnerability detection. Next, GenHuzz requires coverage feedback from the DUT to guide test case refinement. For RISC-V, open-source cores provide direct access to such feedback. For ARM and x86, standardized methods for extracting coverage feedback (either via open hardware or specialized development kits) must be established. Finally, an accessible and modifiable toolchain, including simulators and debuggers for ARM and x86, is crucial for pre-training and GenHuzz's pipeline.

Unfortunately, no fully open-source, standardized ARM or x86 cores exist, as both ISAs are commercially licensed with closed RTL. Lacking modifiable RTL prevents detailed coverage extraction, which is the foundation of GenHuzz. Besides, proprietary simulators typically do not offer the level of transparency required for white-box fuzzing. The absence of open-source RTL designs and accessible toolchains for ARM and x86 not only limits the adaptability of GenHuzz but also highlights a broader issue in hardware security research. Proprietary architectures hinder efforts to develop and evaluate novel hardware vulnerability detection techniques.

This limitation underscores the need for increased openness or at least controlled accessibility within the ARM and x86 development ecosystems. We strongly advocate for greater availability of open-source or accessible designs for ARM and x86 architectures. Such access would enable researchers to apply advanced techniques like white-box fuzzing, uncover previously unknown hardware vulnerabilities, and develop robust defenses. Moreover, fostering an open research environment around these architectures would accelerate innovation in offensive and defensive hardware security, benefiting academia and industry.

9 Related Work

Generic Fuzzers. Laeufer et al. propose RFuzz [53], the first FPGA emulation-based generic hardware fuzzing technique, which relies on multiplexer control signals to generate inputs based on AFL-based mutation functions. However, this method is contained by its coverage metrics detecting vulnerabilities and is computationally expensive. Li et al. proposed a new metric, Full Multiplexer Toggle Coverage, for better hardware coverage performance [54]. Both approaches are closely tied to the Chisel HDL, limiting their broader applicability, while monitoring multiplexers in complex designs introduces significant performance overhead [31]. In contrast, Trippel et al. [23] proposed fuzzing hardware-like software by fuzzing the hardware simulation binary rather than porting software fuzzers directly on the hardware designs. Verilator [55] translates the hardware to an equivalent software model. This approach allows hardware fuzzing to utilize existing coverage metrics commonly used by software fuzzers, such as basic block and edge coverage [31]. Still, it faces scalability problems when, e.g., fuzzing an entire processor.
Processor Fuzzers. TheHuzz [20] simulates the RTL design of the processor with the binary format of the instruction using Synopsys VCS [48] that traces the code coverage through various metrics, including branch, condition, toggle, FSM, and functional coverage. However, this method suffers from low computation efficiency and hardware coverage: the code coverage is only 2.86% higher than that of random methods. DifuzzRTL [28] generates instructions and collects control register coverage to guide the fuzzing process. Following this work, MorFuzz [29] achieves a final coverage that is 4.4 times higher than DifuzzRTL by generating fuzzing seeds based on syntax and semantics and using run-time information feedback to mutate instructions. ProcessorFuzz [31] is a concurrent work that generates instructions and collects coverage of control and status registers. However, these works only focus on the coverage of registers generating the select signals of MUXes, leading to missing bugs and vulnerabilities. To increase design coverage and detect more vulnerabilities, HyPFuzz [24] was proposed to guide the fuzzer using formal verification tools that reach hard-to-reach design spaces. Alternatively, SoC Fuzzer [32] directs the fuzzing based on the security properties (generic cost function) that detect vulnerabilities in the DUT. Finally, Cascade [27] aims to improve instruction execution efficiency by building long programs and eliminating control flow influences. Unfortunately, overly long and complex test cases lead to high time consumption. The overlook of input semantics and low utilization of the hardware feedback constrains the test case's variety and potentially reduces the bug detection capability. ChatFuzz [25] employs reinforcement learning to guide fuzz test case generation but fine-tunes a pre-trained human language model, resulting in suboptimal performance. In contrast, GenHuzz is designed specifically for hardware fuzzing, leveraging a customized GPT model integrated with a tailored HGRL pipeline. By incorporating comprehensive hardware coverage into the fuzzing policy refinement, GenHuzz efficiently uncovers subtle vulnerabilities while maintaining high test case efficiency.

10 Conclusions

We introduce a novel white-box hardware fuzzer, GenHuzz, designed to actively interact with the Device Under Test (DUT) to refine its fuzzing policy, thereby significantly enhancing vulnerability and bug detection capabilities with a limited number of test cases. By leveraging a customized language model-based fuzzer with a hardware-guided reinforcement learning (HGRL) framework, GenHuzz achieves broader hardware state coverage compared to existing fuzzers. The fuzzing results on three RISC-V cores show that GenHuzz identifies all previously reported bugs and uncovers ten new vulnerabilities, including five of high severity. Detecting these vulnerabilities requires complex combinations of multiple instructions, which are nearly impossible to construct using existing fuzzing techniques. In contrast, GenHuzz successfully generates these instruction sequences by understanding inter-instruction semantics.

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.

Ethics and Disclosures

GenHuzz is a hardware fuzzing tool developed to advance the functional and security verification of hardware, particularly processors. Its intended users include security researchers, hardware manufacturers and designers, and hardware security companies. By using the GenHuzz framework, users can discover new bugs and vulnerabilities in hardware designs under test. We thoroughly evaluated GenHuzz on three widely recognized benchmarks, discovering ten new bugs and vulnerabilities. In line with the Menlo Report principles [56], particularly the principle of "Respect for Persons", we have ensured that the security vulnerabilities identified by GenHuzz were promptly communicated to the responsible teams for CVA6 [35], BOOM [34], and RocketChip [33] in August 2024. This timely disclosure was essential to mitigate risks and protect individuals from potential harm that could arise if adversaries were to uncover these vulnerabilities independently. The responsible parties have acknowledged the issues and are actively working on implementing fixes to address the concerns raised in this paper.

To further adhere to the principle of "Justice", which requires fairness in distributing benefits and burdens, we have carefully considered the potential risks associated with the misuse of GenHuzz. To prevent harm and ensure that the framework is used to advance research and enhance hardware security, access to the source code for GenHuzz will be restricted. It will be made available only upon request and with confirmation that it will be used exclusively for responsible research by academic users and hardware manufacturers. This approach aligns with the Menlo Report [56] emphasis on promoting social value while protecting individual rights and privacy, ensuring that GenHuzz contributes positively to the hardware security community without introducing unnecessary risks.

Open Science

In alignment with USENIX Security's open science policy, we release the GenHuzz framework and all vulnerability-related test cases in `https://zenodo.org/records/14727632`, enabling the community to validate and extend our work.

In our responsible disclosure, we received confirmation from the responsible teams and requested Common Vulnerabilities and Exposures (CVE)[3] numbers for these vulnerabilities. Upon submission, CVE numbers were assigned for V2 (CVE-2024-44791), V4 (CVE-2024-44790), V5 (CVE-2024-47789), V8 (CVE-2024-46029), and V12 (CVE-2023-46028). The CVE numbers for the remaining vulnerabilities are pending and will be updated on the above link once they are issued.

[3]CVE is a system for identifying and naming publicly known vulnerabilities. Each CVE entry is assigned to one vulnerability and has a unique ID and basic details about a vulnerability.

References

[1] EMVCo, "EMV specifications (2001)," https://www.emvco.com/.

[2] I. 15408, "Common criteria v3.1 (2017)," https://www.commoncriteriaportal.org/cc/index.cfm?.

[3] O. Mutlu and J. S. Kim, "Rowhammer: A retrospective," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 8, pp. 1555–1571, 2019.

[4] U. Rührmair, J. Sölter, F. Sehnke, X. Xu, A. Mahmoud, V. Stoyanova, G. Dror, J. Schmidhuber, W. Burleson, and S. Devadas, "Puf modeling attacks on simulated and silicon data," *IEEE transactions on information forensics and security*, vol. 8, no. 11, pp. 1876–1891, 2013.

[5] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, "Spectre attacks: Exploiting speculative execution," *Communications of the ACM*, vol. 63, no. 7, pp. 93–101, 2020.

[6] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," *arXiv preprint arXiv:1801.01207*, 2018.

[7] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar *et al.*, "Fallout: Leaking data on meltdown-resistant cpus," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 769–784.

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

[9] Intel, "Machine check error avoidance on page size change/cve2018-12207," https://www.intel.com/content/www/us/en/developer/articles/troubleshooting/software-security-guidance/advisory-guidance/machine-check-error-avoidance-page-size-change.html, 2019.

[10] 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.

[11] 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.

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

[13] 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.

[14] 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.

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

[16] 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.

[17] X. Li, M. Tiwari, J. K. Oberg, V. Kashyap, F. T. Chong, T. Sherwood, and B. Hardekopf, "Caisson: a hardware description language for secure information flow," *ACM Sigplan Notices*, vol. 46, no. 6, pp. 109–120, 2011.

[18] X. Li, V. Kashyap, J. K. Oberg, M. Tiwari, V. R. Rajarathinam, R. Kastner, T. Sherwood, B. Hardekopf, and F. T. Chong, "Sapper: A language for hardware-level security policy enforcement," in *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, 2014, pp. 97–112.

[19] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, "A hardware design language for timing-sensitive information-flow security," *Acm Sigplan Notices*, vol. 50, no. 4, pp. 503–516, 2015.

[20] 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.

[21] 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.

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

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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

[31] 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.

[32] 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.

[33] 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

[34] 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.

[35] 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.

[36] NIST, "Common vulnerability scoring system," https://nvd.nist.gov/vuln-metrics/cvss/v3-calculator, 2019.

[37] R. Saravanan and S. M. Pudukotai Dinakarrao, "The fuzz odyssey: A survey on hardware fuzzing frameworks for hardware design verification," in *Proceedings of the Great Lakes Symposium on VLSI 2024*, 2024, pp. 192–197.

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

[39] B. Mahesh, "Machine learning algorithms-a review," *International Journal of Science and Research (IJSR).[Internet]*, vol. 9, no. 1, pp. 381–386, 2020.

[40] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[41] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, "Improving language understanding with unsupervised learning," 2018.

[42] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

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

[44] E. Barron and H. Ishii, "The bellman equation for minimizing the maximum cost." *NONLINEAR ANAL. THEORY METHODS APPLIC.*, vol. 13, no. 9, pp. 1067–1090, 1989.

[45] 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.

[46] NXP, "Introduction to device trees," https://www.nxp.com/docs/en/application-note/AN5125.pdf, 2015.

[47] 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.

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

[49] RISC-V, "The risc-v instruction set manual volume i: Unprivileged isa," https://github.com/riscv/riscv-isa-manual/releases/tag/riscv-isa-release-8b9dc50-2024-08-30, 2024.

[50] ——, "The risc-v instruction set manual volume ii: Privileged isa," https://github.com/riscv/riscv-isa-manual/releases/tag/riscv-isa-release-8b9dc50-2024-08-30, 2024.

[51] D. Amodei, C. Olah, J. Steinhardt, P. Christiano, J. Schulman, and D. Mané, "Concrete problems in ai safety," *arXiv preprint arXiv:1606.06565*, 2016.

[52] R. Padhye, C. Lemieux, K. Sen, M. Papadakis, and Y. Le Traon, "Semantic fuzzing with zest," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 329–340.

[53] 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.

[54] 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.

[55] Verilator, "Welcome to verilator," https://www.veripool.org/verilator/, 2019.

[56] "The menlo report: Ethical principles guiding information and communication technology research," https://www.dhs.gov/sites/default/files/publications/CSD-MenloPrinciplesCORE-20120803_1.pdf, 2012.

Appendix

A Fuzzer Training Pipeline

The training pipeline of the fuzzer is summarized as follows: First, word embeddings are generated. Positional and segment embeddings are then added to these word embeddings, with positional embeddings indicating each token's position in the sequence and segment embeddings distinguishing between different input segments. This sequence of embeddings is fed through attention layers that use a multi-head masked self-attention mechanism to produce rich contextual embeddings. These embeddings capture the context of each token by incorporating information from other tokens in the sequence. The contextual embedding of the final token is passed to the classification head, which generates unnormalized probabilities for each token in the tokenizer's dictionary. A single token is sampled from these probabilities and appended to the end of the sequence.

B Bugs and Vulnerabilities

Incorrect Exception Handling for Misaligned Load-Reserved Instructions (V_2). In the RISC-V architecture, Load-Reserved (`LR.W`, `LR.D`) and Store-Conditional (`SC.W`, `SC.D`) instructions are intended to perform atomic read-modify-write operations. These operations require the target addresses to properly align according to the instruction size to ensure atomicity and data integrity. For example, the `LR.D` instruction expects an eight-byte aligned address for a double-word load. If a misaligned address is provided, the RISC-V specification dictates that a `Load Address Misaligned` exception should be raised to indicate the improper alignment of the memory access.

However, in the CVA6 [35] core, executing a `Load-Reserved` instruction such as `LR.D` with a misaligned address incorrectly triggers a `Store Address Misaligned` exception instead of the expected `Load Address Misaligned` exception. This deviation from the expected behavior can be reproduced with the code snippets in Listing 5.

Listing 5: Proof of concept code snippets to trigger misaligned on CVA6

```
1 // Load an address in t0 which is not 8-byte
    ↪ aligned, e.g., 0x0000000080001054
2 auipc t0,0x0 // t0 = PC = 0x0000000080001054
3 lr.d s1, (t0)
4 // Exception @ Cause: Store Address Misaligned
```

When the code is executed on the CVA6 core, an exception with the cause `Store Address Misaligned` is thrown, which is incorrect according to the RISC-V specification [49, 50]. On the other hand, when the same code is executed on the Spike RISC-V ISA simulator, a `Load Address Misaligned` exception is correctly raised, aligning with the expected behavior.

The incorrect exception handling introduces a critical issue, potentially causing confusion in software error handling routines and leading to improper software responses to alignment errors.

Incorrect `tval` Register Value for Breakpoint and Fault Exceptions (V_3). In the RISC-V architecture, the `tval`, i.e., trap value, a register is used to hold additional information about exceptions that occur during instruction fetches, loads, or stores. Based on RISC-V ISA specification [49], When a *breakpoint*, address-misaligned, access-fault or page-fault exception occurs, `tval` should contain the faulting virtual address if the exception is related to an instruction fetch, load, or store and if a nonzero value is written to `tval`. This mechanism helps debug by providing the exact virtual address where the exception occurred.

However, in the latest version of the CVA6 core, when such exceptions occur, the instruction value (i.e., the opcode of the instruction that caused the exception) is incorrectly placed in the `tval` register instead of the faulting virtual address. This behavior can be reproduced with the code snippets in Listing 6.

Listing 6: Proof of concept code snippets to trigger `tval` CVA6

```
1 c.ebreak
2 // tval= 0x9002 which is the opcode of c.
    ↪ ebreak
```

When the PoC code is executed on the CVA6 core, the `tval` register is set to `0x9002`, which corresponds to the opcode of the `c.ebreak` instruction. This is incorrect because the RISC-V specification expects the `tval` register to contain the program counter `PC` value at the time of the exception, not the opcode of the instruction that caused the exception. In contrast, executing the same code on the Spike RISC-V ISA simulator correctly places the `pc` value in the `tval` register, demonstrating adherence to the RISC-V specification. This incorrect behavior in the CVA6 core represents a significant deviation from the RISC-V specification and can lead to difficulties in debugging and error handling.

Incorrect `NaN` Boxing in Single-Precision Floating-Point `SQRT` Instruction (V_4). In the RISC-V architecture, single-precision floating-point operations should adhere to the `NaN`-boxing rules as specified in the RISC-V ISA manual [49, 50]. `NaN`-boxing is a mechanism to extend narrower floating-point values into wider registers while maintaining compatibility, e.g., in Machine Learning algorithms. According to the RISC-V ISA specification [49, 50], when an operation like `SQRT.S` (square root, single precision) is executed with a value that is not properly `NaN`-boxed, i.e., where the most significant 32 bits are not all set to 1, the result should be a 32-bit canonical `NaN`, i.e., `0x7fc00000` extended appropriately for the wider

floating-point register.

Listing 7: Proof of concept code snippets for incorrect NaN boxing in single-precision floating-point SQRT instruction on CVA6 [35]

```
1 lui t0, 0x3d0f4
2 // t0  :000000003d0f4000
3 fcvt.d.w ft5, t0
4 // ft5 :41ce87a000000000
5 fsqrt.s fa5, ft5
6 // fa5 :ffffffff00000000
7 csrr gp, fflags
```

However, in CVA6 cores, executing a single-precision floating-point square root operation on an invalid NaN-boxed value incorrectly results in a boxed single-precision floating-point value of `0xffffffff00000000`. This is incorrect according to the RISC-V specification [49, 50], which requires the result to be a 32-bit canonical `NaN`. The incorrect behavior can be reproduced with the code in listing 7. When the code in listing 7 executed on the CVA6 core, the `fa5` register contains the value `0xffffffff00000000`, which is not the correct canonical `NaN`. In contrast, executing the same code on the Spike RISC-V ISA simulator results in `ft5=0xffffffff7fc00000`, correctly adhering to the NaN-boxing rules specified in the RISC-V ISA manual.

This incorrect handling of `NaN`-boxing in the CVA6 core can lead to significant software issues that rely on accurate floating-point arithmetic, particularly in machine learning training, cryptography computation, or any domain requiring precise floating-point operations. The incorrect result returned from floating-point instructions could lead to undetected flaws, potentially impacting the stability of software systems and consequently leading to security issues, e.g., decrease the quality of machine learning inference.

Incorrect `NaN` Boxing and Exception Handling in Single-Precision Floating-Point Division (V_5). As mentioned in appendix B, in the RISC-V architecture, single-precision floating-point operations are expected to follow specific rules for handling `NaN`-boxed values. The GenHuzz found that in the CVA6 core, performing any single-precision floating-point division with an invalid `NaN`-boxed value incorrectly results in a non-canonical value `0xffffffffff800000` and raises the "divide by zero" exception flag `fflags` set to `0x8`. The expected behavior can be reproduced with the code in listing 8.

On the CVA6 core, executing the PoC code results in `ft3` being set to `0xffffffffff800000` and `fflags` containing `0x8`, indicating a *divide by zero* exception. However, based on RISC-V ISA specification [49, 50], if we ran the same code, the `ft3` register should contain the canonical `NaN` value, i.e., `0xffffffff7fc00000`, and `fflags` should be `0x0`, which indicating no exceptions were raised. As discussed in appendix B, this incorrect behavior in the CVA6 core could result in significant problems for operations that depend on

Listing 8: Proof of concept code snippets to trigger incorrect NaN boxing and exception Handling in single-precision floating-Point division on CVA6 [35]

```
1 lui t5, 0xbd131
2 fmv.w.x ft3, t5 // ft3 = 0xffffffffbd131000
3 lui s4, 0xee72f
4 fmv.w.x fa7, s4 // fa7 = 0xffffffffee72f000
5 fcvt.d.l fa7, s4 // fa7 = 0xc1b18d1000000000
6 fdiv.s ft3, ft3, fa7 // ft3 = 0
    ↪ xffffffffff800000
7 csrr a4, fflags // a4 = 0x0000000000000008
```

accurate floating-point calculations. This is particularly concerning for applications in machine learning, where precise floating-point arithmetic is essential for tasks such as floating-point precision conversion, a process that is widely used. Errors in these calculations can lead to inaccuracies in model training, predictions, and overall system performance, potentially compromising the reliability and effectiveness of machine learning algorithms.

No exception for invalid `rm` in single-precision operation(V_{10}).

In the RISC-V architecture, the rounding mode (`rm`) field in single-precision floating-point instructions specifies the rounding behavior for floating-point operations. This field is part of the floating-point control and status register (`fcsr`) and guides how the processor should manage rounding when the result of a floating-point calculation cannot be precisely represented. The rounding mode can be directly encoded within the instruction (`rm` field) or dynamically set using the floating-point rounding mode register (`frm`). If an invalid rounding mode value (`101`–`111` based on RISC-V ISA Specification [49,50]) is specified, the architecture requires that any floating-point operation using this invalid mode must trigger an illegal instruction exception, as outlined in the RISC-V specification [49, 50].

Using GenHuzz, we generated a test case that intentionally included an invalid rounding mode in a single-precision floating-point operation. Upon running this test case, it was observed that the `BOOM` core did not raise a `trap_illegal_instruction` exception, as would be expected when executing instructions with an invalid `rm` field. This deviation indicates non-compliance with the RISC-V standard, which treats such operations as illegal. This finding suggests a potential flaw in the core's handling of rounding modes, which could have further implications for software that relies on correct floating-point rounding behavior.

While this issue had been identified manually in earlier versions of the `BOOM` core, our automated testing—conducted without any previous knowledge of this bug—revealed that the problem persists in the current version. This emphasizes the importance of thorough automated testing methodologies to uncover issues that may have been overlooked in manual inspections.