RLFuzz: Accelerating Hardware Fuzzing with Deep Reinforcement Learning

Raphael Götz*, Christoph Sendner*, Nico Ruck*,
Mohamadreza Rostami†, Alexandra Dmitrienko*, and Ahmad-Reza Sadeghi†,
*University of Würzburg, Germany
†Technical University of Darmstadt, Germany

***Abstract*—Once hardware is manufactured, it becomes immutable, and any design flaws or vulnerabilities are permanent. Therefore, comprehensive testing of hardware designs is essential before production. However, the increasing complexity of modern processors makes static analysis and formal verification increasingly challenging. Fuzzing is a highly effective technique for detecting software vulnerabilities but adapting it to hardware presents unique challenges due to the fundamental differences between software and hardware. However, the current state-of-the-art in hardware fuzzing still has significant limitations, particularly in terms of performance, practicality, and the need for human intervention. To address these issues, we introduce RLFuzz, a novel hardware fuzzer that employs reinforcement learning to explore processors autonomously, achieving faster and more comprehensive coverage. Reinforcement learning enables the fuzzer to learn from its interactions with the processor without requiring labeled data, allowing it to more effectively select modifications for test cases and target previously unexplored areas. RLFuzz uses an asynchronous training mechanism that permits concurrent fuzzing and neural network training. To optimize RLFuzz, we performed an extensive evaluation of various deep Q-learning optimization techniques and hyperparameters. We then tested the optimized fuzzer on three complex RISC-V cores—the Rocket core, CVA6, and BOOM—and compared its performance to a current hardware fuzzer, TheHuzz [20]. Our results show that RLFuzz achieved up to 1.77% higher coverage, and on average 2.35 times faster, and demonstrated a maximum speedup of up to 6.93 times. For branch coverage, the average speedup was 7.85 times, with a peak speedup of 92.98 times. Additionally, RLFuzz needed less time to complete 100,000 test cases and did not require any human intervention.**

***Index Terms*—Fuzzing, Reinforcement learning.**

I. Introduction

Hardware components are the foundation of any computing system, and the security of a software system is inherently tied to the security of the underlying hardware. Once the hardware is manufactured, it becomes immutable, meaning that any design flaws or vulnerabilities are permanent and cannot be fixed after production. This immutability poses significant security risks, especially considering that hardware such as CPUs typically have a lifespan of five to ten years, depending on maintenance and usage. Recent discoveries of data-leaking vulnerabilities such as Downfall [34] and Zenbleed [38] - similar to the well-known Meltdown and Spectre side-channel attacks [15] - underscore the critical importance of identifying bugs and flaws during the hardware design phase. These vulnerabilities have affected billions of devices and highlight the growing challenge of securing today's increasingly complex processors.

There are several approaches to detecting hardware vulnerabilities. Static code analysis [28] is an examination of the hardware description code without execution, but without access to the source code, it is not applicable. Formal verification [3] uses mathematical techniques to prove specific hardware behavior. However, the complexity of modern hardware makes these methods impractical [22]. Fuzzing is an automatic and highly effective technique that is well-established in software testing and vulnerability detection. It involves inputting large amounts of both valid and invalid data into the system and observing its responses to identify unwanted behavior [4], [20], [40]–[42], [46]. However, adapting fuzzing techniques from software to hardware is challenging [40]. Unlike software, hardware does not crash, making it difficult to detect bugs using traditional software fuzzing techniques. In addition, applying software fuzzers to hardware requires translating hardware into software models, which is not only computationally expensive but can also introduce inaccuracies. To address these challenges, dedicated hardware fuzzers have been developed, such as DifuzzRTL [17] and TheHuzz [20]. DifuzzRTL uses established hardware simulation tools to simulate hardware directly, without the need for software models, and compares the output to a Golden Reference Model (GRM). The approach uses a coverage metric based on control register values, which results in only a small number of coverage points, and therefore the fuzzer often misses security-critical vulnerabilities [20]. TheHuzz improves on this by using code coverage metrics, such as branch and condition coverage, to guide the fuzzer instead. However, a significant drawback of TheHuzz is its dedicated profiling phase before fuzzing begins, which requires additional proprietary licenses and human intervention.

To further advance hardware fuzzing, we present RLFuzz, a novel hardware fuzzer that uses machine learning, specifically reinforcement learning (RL) [19], [37], to learn the behavior of the processor based on its responses to the inputs and subsequently optimize the test cases. Specifically, RLFuzz uses deep Q-learning (DQL) [32], which allows autonomous exploration of the processor through generated inputs and observed outputs. A dedicated neural network refines the input generation process based on the coverage achieved by previous

inputs, resulting in faster exploration of previously uncovered hardware components.

Integrating RL into a hardware fuzzer required overcoming several challenges:

- The first challenge was aligning reinforcement learning with the fuzzing process by defining finite and manageable state and action spaces. We defined the state as a single instruction in a test case and the action space as the set of available mutation operators for that instruction.
- The second challenge was synchronizing the fuzzing process and the machine learning algorithm to allow them to operate concurrently without creating performance bottlenecks. RLFuzz implements an asynchronous training mechanism that allows the fuzzing and training cycles to run in parallel and efficiently manages data transfer between them.
- The third challenge was to optimize the input representation for the neural network to provide enough relevant information while avoiding excessive features that could negatively impact performance. Therefore, we explored three input representation approaches: numerical values, embeddings, and dimensionality reduction techniques combined with additional features.
- The final challenge concerns the reward. Specifically, it was necessary to design a neural network that could handle feedback for entire test cases at once, and mitigate both large rewards at the beginning and sparse rewards in the later stages of the fuzzing run. RLFuzz incorporates a Seq2Seq [48] architecture to process entire test cases at once, applies reward clipping to counteract high rewards, and uses additional reward signals to strategically target uncovered parts of the processor in later stages of the fuzzing run.

After designing and implementing RLFuzz, we conducted an ablation study of various DQL optimization techniques to optimize its performance. Our main contributions are:

- We introduce RLFuzz, a novel hardware fuzzing approach that leverages machine learning, specifically DQL, to learn processor behavior from test case responses and efficiently select mutations for new inputs.
- We implement an asynchronous training mechanism for RLFuzz, eliminating the negative impact of neural network training on fuzzing speed.
- Compared to TheHuzz, RLFuzz requires fewer proprietary software licenses and no human intervention.
- We conduct an ablation study of various DQL optimizations and hyperparameters to optimize RLFuzz.
- We evaluate RLFuzz using three modern RISC-V processors: Rocket core [2], CVA6 [56] and BOOM [57], comparing its performance against TheHuzz [20]. Our evaluation considers overall coverage, branch coverage, coverage speed, and the total time required for the entire process.
- RLFuzz not only achieves up to 1.77% more coverage but also attains coverage on average $2.35\times$ faster than the current state-of-the-art , with a maximum speedup of $6.93\times$. For branch coverage, the average speedup is $7.85\times$, with a peak of $92.98\times$. Additionally, RLFuzz completed 100,000 test cases faster than TheHuzz with the same number of licenses and required no human intervention.
- During the evaluation, we confirmed five bugs in the CVA6 core. Three of these were also found by TheHuzz [20], while the other two vulnerabilities were exclusively discovered by HyPFuzz [6] using formal methods.

In summary, this work introduces a novel hardware fuzzer, RLFuzz, that uses reinforcement learning to optimize the mutation-selection process to strategically test previously uncovered parts of the processor. We conduct an ablation study of DQL optimization techniques to optimize RLFuzz and then compare it with a state-of-the-art hardware fuzzer, TheHuzz [20], on three RISC-V processors.

II. Background

A. Fuzzing

Fuzzing [25], [29], [43] is a dynamic testing methodology used to discover vulnerabilities and bugs in software or hardware by inputting large volumes of both valid and invalid data and observing how the target system responds to the inputs. In software testing, the primary goal is usually to cause crashes, as they can reveal potential vulnerabilities. However, since hardware lacks a clear definition of a crash, a different approach must be employed to identify issues. State-of-the-art hardware fuzzing approaches [6], [20], [40], [41] involve comparing the output traces of the Design Under Test (DUT) with those of a golden reference model (GRM), which accurately simulates the expected behavior for a given set of inputs. If a mismatch is detected between the traces, it signals a potential bug. Fuzzing is highly effective in uncovering edge cases and unanticipated inputs that developers may have overlooked. A common measure for the success of a fuzzer is the *coverage* of the target [33]. It states how many parts of the target were utilized during the processing of the inputs and is therefore an indicator of how thoroughly the target was tested. Fuzzing has evolved significantly from naive approaches that generate inputs randomly [10] to more sophisticated mutation-based methods that apply specific mutation operations to modify inputs [11]. Modern evolutionary fuzzers [11], [13], [26] further enhance this process by using fitness functions to evaluate and prioritize inputs and mutations that are likely to uncover new vulnerabilities. Hongfuzz [13], AFL [11], and libFuzzer [26] are examples of modern evolutionary software fuzzers. Machine learning techniques are increasingly being integrated into fuzzers to enhance their capabilities [43], [54]. These techniques can help with generating better inputs and optimize the selection of mutation operators. They can also be used to filter out less promising inputs before they are tested on the DUT, and can additionally help in analyzing the output to facilitate the detection of bugs [47].

B. Seq2Seq Models and LSTMs

Sequence to sequence (Seq2Seq) [48] models are a type of neural network architecture designed to transform one data sequence into another. They consist of an encoder, which processes the input sequence and encodes it into a fixed-length context vector, and a decoder, which takes this vector and generates the output sequence. Commonly used in tasks like machine translation, text summarizing, and speech recognition, Seq2Seq models can handle variable-length input and output sequences. Seq2Seq models can be built using long short-term memory (LSTM) [16] networks, an extension of recurrent neural networks (RNN) [31]. RNNs allow cycles in their computational graphs, which means they can implement an internal state, i.e., memory. LSTMs extend this concept by incorporating memory cells with three gates. The forget gate allows the cell to forget the previously stored state. The input state gate allows it to modify the internal state using the current input, and finally, the output gate creates an output of the cell by combining both the new internal state and the input. This structure allows LSTMs to retain and utilize information over long periods effectively.

C. Deep Q-learning

Reinforcement learning (RL) [19], [37] allows algorithms to learn from interactions with the environment without relying on labeled data. In RL, an agent receives feedback in the form of rewards or penalties based on its actions. The agent's goal is to develop a strategy that maximizes long-term rewards. In traditional Q-learning [55], the agent learns an optimal policy by iteratively updating an action-value function calculated from a table that lists the expected rewards for each combination of state and action. However, this approach is not practical when the state and action spaces are very large. Deep Q-Learning (DQL) [32] instead approximates the action-value function using a deep neural network, allowing RL to be used in large discrete and even continuous state spaces. DQL often suffers from instability and divergence [7], [49]. To address these issues and improve its stability and performance, the following optimizations are commonly applied:

Double DQL uses two separate neural networks to decouple action selection from value estimation and counter over-optimistic value estimates

Experience Replay stores past experiences in a replay buffer and randomly samples them during training to break correlations between consecutive updates.

Mini-batches update the network with multiple experiences per iteration, providing a more representative sample than single experiences.

Upsampling reuses individual experiences multiple times during training. This gives the network more opportunities to learn from very significant single experiences.

Reward Management involves modifying rewards to improve training efficiency, such as reward clipping in environments with large rewards and using additional feedback signals in scenarios with sparse rewards.

III. Approach

To address the threat of hardware vulnerabilities, we introduce RLFuzz, a novel hardware fuzzing approach that integrates RL. Specifically, RLFuzz employs a DQL-based mutation engine that autonomously explores the processor through the generated inputs and dynamically learns to select mutations that increase its coverage.

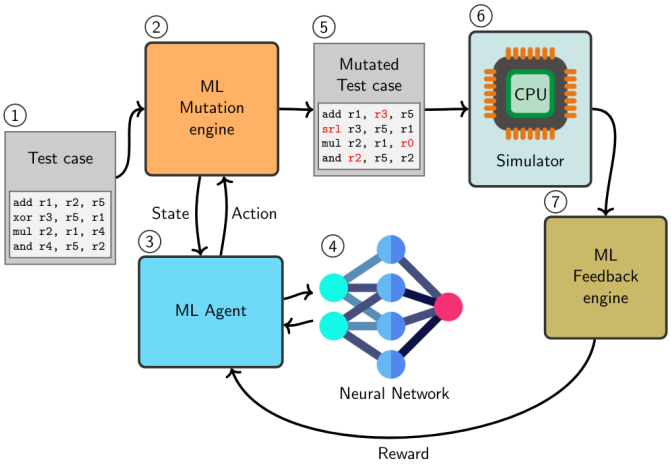

Fig. 1: Overview of RLFuzz

A. RLFuzz's fuzzing loop

Fig. 1 illustrates the general fuzzing approach of RLFuzz. The fuzzing process is iterative, with the RLFuzz's loop consisting of seven steps. In step (1), a base test case is created, either from scratch or reused from a previous iteration. Then, in step (2), information is extracted to serve as a state for the RL process. Step (3) further converts this state into an input vector. In step (4), the vector is used to perform inference on the neural network, which approximates an action-value function to predict actions likely to yield the highest reward for the given state. In the case of RLFuzz, the state is the test case's instructions, the actions are mutation operators, and the reward is the gain in coverage of the DUT that the test case achieves. Step (5) then applies the mutation operators selected by the neural network to the test case. This mutated test case is subsequently used as the input program for the simulated DUT in step (6). Then, in the final step (7), the output is analyzed to obtain the archived coverage values, which are combined and used as the reward for reinforcement learning.

B. Components of RLFuzz

To implement this fuzzing loop, RLFuzz consists of five components, shown in Fig. 1.

ML Mutation Engine. The task of the mutation engine is to modify incoming test cases to create new ones. The mutation engine is responsible for modifying test cases to generate new ones. This process begins by separating the test instructions from the configuration instructions required to run a program on the processor. The extracted test instructions

are sent to the ML agent, which selects a mutation operator for each instruction. The ML mutation engine then applies these selected mutations to the test case. It is important to note that the number of test instructions in a test case remains fixed throughout this process. RLFuzz's RL action space comprises the mutation operators, which were adopted from TheHuzz [20]. An overview of these operators is provided in Tab. I. The Mutations M0–M7 modify only the data bits of instructions, leaving the opcode unchanged, while the remaining mutations can manipulate both data and opcode bits, including random bit overwrites, opcode modifications, instruction deletion, and instruction cloning.

#	Name	Description
M0	Bitflip 1/1	Flip a single bit
M1	Bitflip 2/1	Flip two adjacent bits
M2	Bitflip 4/1	Flip four adjacent bits
M3	Bitflip 8/8	Flip a single byte
M4	Bitflip 16/8	Flip two adjacent bytes
M5	Arith 8/8	Treat a single byte as an 8-bit integer, +/- value from 0 to 35
M6	Arith 16/8	Treat 2 adjacent bytes as a 16-bit integer, +/- value from 0 to 35
M7	Arith 32/8	Treat 4 adjacent bytes as a 32-bit integer, +/- value from 0 to 35
M8	Random 8	Overwrite a random byte with a random value
M9	Delete	Delete the instruction
M10	Clone	Clone the instruction
M11	Opcode	Overwrite opcode bits

TABLE I: Mutation operators [20]

ML Agent. The ML agent implements the DQL component of RLFuzz, performing both inference and training of the neural network simultaneously. Upon receiving test instructions from the ML mutation engine, the agent converts them into a suitable input vector, which is subsequently fed into the network. The neural network then yields mutation for each instruction. These mutations are sent back to the ML mutation engine which applies them to the test case. Neural network training occurs in parallel with the fuzzing process. The ML agent manages the training data, which are previously executed test cases, and utilizes the rewards, computed by the ML feedback engine, to calculate the loss necessary for updating the network weights. To balance the trade-off between exploiting current knowledge and exploring new options, our ML agent uses the well-established epsilon greedy strategy [30]. With a probability of ϵ, the agent selects random mutations instead of performing the forward pass on the neural network. This approach ensures continuous exploration of new actions. As learning progresses, ϵ is gradually reduced to favor the exploitation of the optimal policy.

Neural Network. In the context of DQL, a neural network learns to approximate an action-value function, which assigns an expected reward to each possible action for a given state. In the case of RLFuzz, a state consists of the instructions of a test case, and actions are mutation operators applied to them. The reward is the coverage the mutated test case will achieve. Therefore, the neural network learns to select mutation operators that maximize coverage. RLFuzz employs a custom LSTM-based seq2seq network that processes entire fixed-length test cases as input and outputs a mutation operator for each instruction. As depicted in Fig. 2, the model consists of a single LSTM cell with a hidden dimension, of which the size is determined by a hyperparameter. The LSTM output is passed through a linear layer, producing a vector corresponding to the available actions, 12 mutation operators, and one no-mutation option, resulting in 13 output channels. The output is normalized using the log softmax function. The network operates as a seq2seq model, with loss computation and backpropagation performed only after processing the complete test case, as the reward signal is based on the overall coverage achieved by the entire test case.

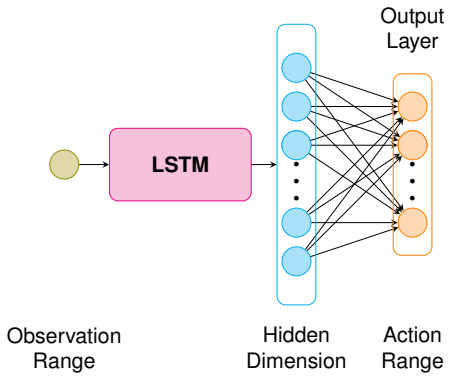

Fig. 2: Neural Network Structure

Hardware Simulator. The test case is executed by the DUT, which is simulated using a well-established hardware simulator, e.g., VCS [50] and Verilator [45]. The output execution traces of the DUT are compared to the expected behavior of a golden reference model (GRM), and discrepancies are logged for manual analysis later.

ML Feedback Engine. The feedback engine evaluates the output from the simulator to determine the coverage achieved by each test case and calculates the coverage gain. We use the same coverage metrics as TheHuzz [20]. For calculating the reward we specifically use the notion of *total coverage* which sums up five subtypes of coverage:

- Branch Coverage: Assesses whether both outcomes of each conditional branch are evaluated.
- Condition Coverage: Tracks if every condition block has been tested for all possible input values
- FSM Coverage: Examines all possible states and transitions of a 3-bit state register in the finite state machine.
- Toggle Coverage: Tracks the toggling of all flip-flops between 1, 0, and floating states.
- Line Coverage: Ensures that every line of RTL code is executed at least once.

Based on the total coverage, the reward for the ML agent is computed according to our reward strategy, which for some versions of RLFuzz includes reward clipping and sparse reward countering. The original instructions, the applied mutations, the resulting mutated instructions, and their corresponding rewards are then stored as training data for the neural network.

C. Design Considerations

Four key factors must be considered in the design of RLFuzz's, which are discussed in the following section.

Consideration I - Finite Space. Deep Q-Learning (DQL) requires finite and manageable state and action spaces [55]. Large state spaces demand more complex input layers, while large action spaces inflate the output layer, as each action requires its own feature. To address this, we define the RL problem in RLFuzz's can as a Markov Decision Process (MDP), represented by the 4-tuple (S, A, P_a, R_a):

- S: The state space, which for RLFuzz represents the CPU states reaches during execution.
- A: The action space, which in our case are the available mutation operators
- P_a: The transition probability of moving to a new state after applying an action
- R_a: The reward for transitioning to a new state, is measured as the additional coverage achieved.

Since RLFuzz uses DQL, the policy function, π, which maps states to actions and guides the model to transition to particular states by selecting specific actions, is modeled by an ML model. **Consideration II - Scheduling.** In RLFuzz, the neural network is simultaneously used for mutation generation and is trained and optimized based on the environment's feedback. Performing these actions sequentially would significantly slow down the fuzzing process and negate the performance benefits of integrating RL. Therefore, RLFuzz employs an efficient data transfer and synchronization mechanism to allow the fuzzing process and training to run concurrently. This mechanism operates as follows: When the DUT has processed a test case, the resulting sample is written to a buffer. The training process uses the samples in the buffer to train the neural network and calculate updated network weights. Before the fuzzing process performs inference to select mutations, it checks if new weights are available. If they are, the fuzzing process updates its working copy of the network. However, if not, the fuzzing process does not wait and immediately performs inference using the currently available version of the network. This ensures that the fuzzing process runs at maximum speed without delays.

Consideration III - Input Representation. The input representation plays a critical role in the performance of neural networks. On one hand, it must contain enough relevant information for the network to make accurate predictions. On the other hand, including too many features, especially irrelevant ones, can negatively affect the training process due to the *curse of dimensionality* [53]. For RLFuzz, the primary input to the neural network consists of the instructions within the test cases. We considered and evaluated three different approaches for input representation: numerical value, embedding, and adding coverage information to the input combined with dimensionality reduction. The first approach represents the instructions directly as float64 values, allowing the neural network to process them without any additional preprocessing or intermediate layers. The second approach uses embeddings, a technique widely used in natural language processing, which maps discrete input tokens to continuous vectors, resulting in a more compact input representation. Finally, we explored the possibility of adding the current coverage information of branching constructs in the processor, i.e., branch coverage, to the input vector. To address the high dimensionality of coverage data, we considered two *feature extraction* strategies: Principal Component Analysis (PCA) with Singular Value Decomposition (SVD) and embedding layers. PCA reduces dimensionality by transforming input data into orthogonal components ranked by variance and discarding less significant components. However, our experiments found embedding layers, a technique widely used in natural language processing, to be more effective. Embedding layers map discrete categorical variables to continuous dense vectors. This mapping is learned during network training. Embedding layers are most effective when the vocabulary size is small, meaning they handle a limited number of distinct input tokens. We, therefore, use TheHuzz's instruction detection engine [20] to reduce the vocabulary size to the number of available instructions for the processor plus one additional token for any unrecognized instructions.

Consideration IV - Rewards. For RLFuzz, there were several challenges related to the reward used for the neural network. Rewards are a core aspect of RL and have a significant impact on the performance of the network. The first issue is that DQL necessitates the provision of a reward for each action. However, given that RLFuzz uses feedback from the simulator to compute the reward, it is not feasible to assign a reward to individual mutations; there is only one single reward for the entire test case. This problem has been addressed by adopting a seq2seq approach, which performs the backward pass for an entire test case using a loss calculated from the corresponding reward. Second, calculating an appropriate reward based on the coverage of the DUT is challenging because large portions of the processor are inherently covered, irrespective of the input instructions. This is due to basic instructions that are necessary for configuring the processor after power-up, as well as for initializing registers and components, such as the floating-point unit in the processor. As a result, the network initially receives large rewards regardless of the chosen mutations. To address the challenges of reward distribution, the optimized version of RLFuzz employs both reward clipping and sparse reward counting simultaneously. To implement reward clipping, we define two thresholds, thus creating four reward levels: coverage gains above the upper threshold receive the maximum reward, gains below the upper threshold are split into medium and low rewards, and no reward is given if there is no coverage gain. Sparse reward counting complements this by encouraging the exploration of uncovered branches. It tracks branch combinations and awards an additional reward that is combined with the regular clipped reward. The maximum additional reward is given for new discoveries and then reduced for previously visited combinations based on their frequency. These two mechanisms work together to balance high rewards in the early stages with

sparse rewards in the later stages.

IV. Evaluation

This section presents a thorough analysis of RLFuzz's performance. In Sect. IV-A, we present the results of an extensive study on various DQL optimizations to fine-tune and optimize RLFuzz. Next, Sect. IV-B compares the optimized version of RLFuzz against the state-of-the-art hardware fuzzers, TheHuzz [20] and MABFuzz [12]. Finally, Sect. IV-C details the analysis of five bugs confirmed by RLFuzz.

Testbed. For our experiments, we used two Ubuntu 20.04.6 LTS servers with different configurations. Server 1 is equipped with an Intel Xeon Gold 6342 CPU, 251 GiB of RAM, and a 3.71 TiB PERC H755 Front disk. Server 2 has an AMD EPYC 7313 CPU, 95 GiB of RAM, and a 128 GiB disk. Both servers run identical versions of Chipyard (1.7.1), Spike (1.1.0), and Synopsys VCS (2021.09). In addition, Server 2 has additional tools such as fd (7.4.0) and conda (23.7.2) that are not installed on server 1.

Processors. We evaluate the performance of RLFuzz on three RISC-V cores: the in-order Rocket [2] and CVA6 [56] cores, and the out-of-order BOOM [57] core. The Rocket core, developed at UC Berkeley and implemented in Chisel, is a five-stage in-order processor that supports the 64-bit RISC-V instruction set and is optimized for efficient instruction-level parallelism. The Rocket chip has been successfully manufactured and Linux was able to boot on silicon prototypes. CVA6, also known as Ariane, is a six-stage in-order CPU implementing the 64-bit RISC-V instruction set. It is designed for power efficiency and full OS support and is capable of running Unix-like operating systems. The third processor, the Berkeley Out-of-Order Machine (BOOM), differs from the others in its out-of-order execution capabilities. Although it conceptually spans ten stages, it effectively implements them in seven. We generate the RTL for these cores using Chipyard [52]. To evaluate the performance of the cores on the test cases, we use the open-source ISA simulator Spike [39]. While these RISC-V cores were chosen for their open-source availability, the techniques used in RLFuzz are not limited to the RISC-V architecture.

Methodology. The evaluation comprised two phases. First, an ablation study was conducted to optimize RLFuzz's configuration, focusing on hyperparameters and DQL enhancements. Each experiment in this phase simulated 50,000 test cases on the Rocket core. Second, the optimized RLFuzz was compared against the state-of-the-art hardware fuzzer TheHuzz [20] across all three processors, using 50,000 test cases per experiment. Each experiment was repeated three times for reliability. Additionally, we included results from the state-of-the-art hardware fuzzer MABFuzz [12] for the Rocket and BOOM cores and provided a runtime comparison between TheHuzz and RLFuzz over 100,000 test cases.

ML Hyperparameters. In Sect. IV-A, we explore optimal hyperparameters and DQL optimizations for RLFuzz. However, certain hyperparameters remained consistent across all experiments. Specifically, we used the AdamW optimizer [27], the Smooth L1 loss function, a learning rate of 0.0001, a discount factor of 0.99, and the epsilon-greedy strategy to balance exploration and exploitation.

Metrics. We adopt the coverage metrics from Kande *et al.* [20] used for TheHuzz to ensure comparability between our approach and theirs. As mentioned in Sect. III-B, our primary metric, *total coverage*, is a composite measure that integrates five specific coverage types: branch coverage, condition coverage, FSM coverage, toggle coverage, and line coverage. In addition to these coverage metrics, we also consider the time required to complete a fuzzing run, which is critical in scenarios such as chip design or when using expensive hardware. We also evaluate the number of test cases required to achieve a given coverage level. By comparing the test case requirements of our approach with those of the reference method, TheHuzz, we can calculate the coverage speedup achieved by our approach.

Reference Fuzzers. RLFuzz is based on TheHuzz [20] and most components except the mutation engine shared between the two fuzzers. Therefore, to directly evaluate the impact of the RL-based mutation engine, we conduct a comparison with TheHuzz. However, to demonstrate that RLFuzz remains competitive with cutting-edge hardware fuzzers, we also compare it to MABFuzz [12], which uses a multi-armed bandit algorithm to optimize seed selection. Note that RLFuzz and MABFuzz focus on different aspects of the fuzzing process and the approaches are complementary and could theoretically be combined. Since MABFuzz's results were obtained directly from its developers, its runtime performance could not be evaluated, and its results on the CVA6 core were not directly comparable.

A. Hyperparameter Ablation Study

This section presents the results of experiments with various DQL optimizations and hyperparameter configurations, identifying the most effective configurations for optimizing RLFuzz.

Double DQL. The first optimization was double DQL, which employs an additional neural network to mitigate over-optimistic value estimates. As shown in the first graph of Fig. 3, this technique consistently achieved higher coverage after processing 5,000 test cases compared to the version without it, and therefore it was included in the final approach.

Experience Replay. Another optimization tested was experience replay, designed to reduce the correlation between consecutive training data. As depicted in the second graph of Fig. 3, when used alone, experience replay resulted in slightly faster early coverage but converged to the same final coverage as the version without it. However, in experiments where experience replay was combined with other optimizations—specifically mini-batches, double DQL, and optimized reward management, the coverage was significantly enhanced, justifying its integration into the final approach.

Mini-Batches. For mini-batches, we tested batch sizes of 10, 32, 64, and 128, and compared them to a version that did not use mini-batching. In the first half of the experiment, depicted in the third graph of Fig. 3, all versions performed

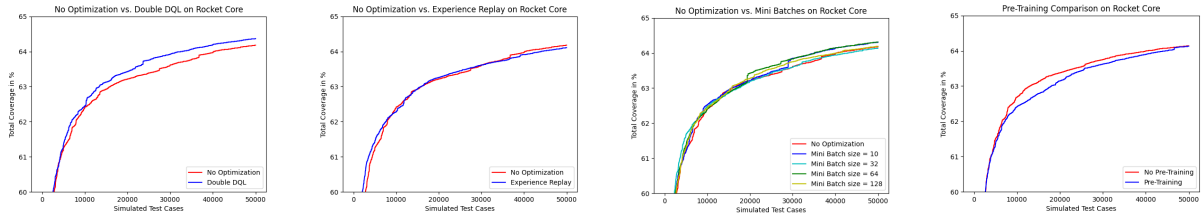

Fig. 3: Results of the ablation study concerning double DQL, experience replay, mini-batches, and pretraining

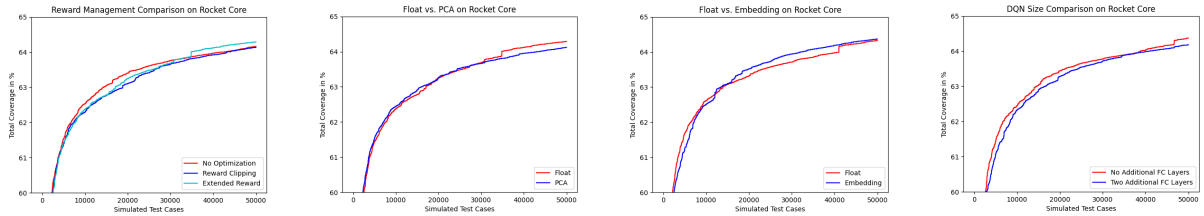

Fig. 4: Results of the ablation study concerning reward management, PCA, embedding and additional input layers

very similarly, but in the second half, the versions with batch sizes of 10 and 64 clearly outperformed the others, achieving the highest coverage. Consequently, we selected a mini-batch size of 10 for the final version.

Pre-training. The goal of pre-training is to use knowledge from previously collected data to allow faster exploration of new processors. However, this did not manifest itself in our experiments. We evaluated a pre-trained version of the network, trained on 150,000 previously simulated test cases, and the results, presented in Fig. 3, showed that the pre-trained model consistently underperformed compared to the non-pre-trained version throughout most of the experiment. As a result, pre-training was not used in the final implementation.

Reward Management. The fourth graph in Fig. 4 compares the performance of three reward management strategies: a baseline version without any reward optimization, reward clipping alone, and an extended reward system that combines reward clipping and sparse reward countering. These techniques are described in detail in Sect. III. In the early stages of the experiment, the baseline version without reward optimization achieved the highest coverage. However, by the end of the experiment, the extended reward system outperformed both the baseline and the reward clipping-only approaches, leading us to adopt the extended reward system in the final version of RLFuzz.

PCA. As described in *Consideration III* in Sect. III-C we considered a version that adds the current branch coverage to the input vector and then applies PCA to reduce its dimensionality. However, as shown in the first graph of Fig. 4, the PCA-based approach significantly underperformed compared to the version without PCA, leading us to exclude the additional branch coverage information and PCA from the final methodology.

Embedding. We also explored feature reduction using a 32-dimensional embedding. More details about this version are provided in Sect. III. The second graph in Fig. 4, compares this version to one without feature reduction. After approximately 12,000 simulated test cases the version using the embedding showed significantly higher coverage than the non-embedding version and it was therefore incorporated into our final approach.

Additional Linear Input Layers. The basic DQN architecture includes an LSTM and a linear output layer. We experimented with an alternative model that added two linear layers in front of the LSTM. However, as shown in the third graph of Fig. 4, this version underperformed compared to the simpler baseline for most of the experiment. The increased complexity of the additional layers did not provide any benefit and instead worsened computational efficiency. Therefore, these extra layers were not included in the final model.

Transformer Encoder. We also experimented with replacing the LSTM with a transformer encoder with twelve layers and twelve attention heads. However, as shown in the fourth graph of Fig. 5, after approximately 30,000 simulated test cases, the LSTM-based model outperformed the transformer encoder in terms of coverage. In addition, the complexity of the transformer encoder significantly increased the computational resource requirements and training time without providing significant benefits. As a result, the LSTM was used for the final version of RLFuzz.

Upsampling. We also experimented with upsampling, a technique that reuses training data multiple times. We compared different levels of upsampling, ranging from no upsampling to using the same data for up to five consecutive training steps. Throughout the experiment, the performance differences between the versions were minimal. However, in the end, the version with no upsampling achieved the highest coverage. Therefore, upsampling was not included in our final approach.

Number of Instructions. We evaluated the effect of varying the number of instructions per test case. We compared the

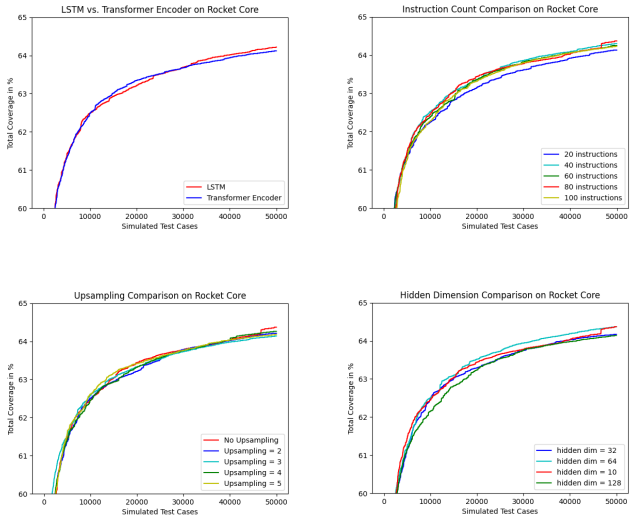

Fig. 5: Results of the ablation study concerning the transformer encoder, number of instructions, upsampling, and hidden dimension size

configuration of TheHuzz [20] 20 instructions per test case, to versions that used 40, 60, 80, and 100 instructions per test case. As shown in the left graph of Fig. 5, after 10,000 simulated test cases, the 20-instructions version showed significantly lower coverage than all others. In the end, the 80 instructions per test-case version slightly outperformed the others, so it became the choice for the final approach.

Hidden Dimension Size. We investigated the effect of different hidden layer sizes in the neural network, testing versions with 10, 32, 64, and 128 dimensions. The right graph of Fig. 5 shows that the 128-dimension version performed the worst, with a significantly slower coverage gain in the first half of the experiment. Both the 10-dimension and 64-dimension versions achieved similar final coverage. However, due to the higher computational efficiency of a smaller network, we chose a hidden dimension size of 10 for the final model.

TABLE II: Optimized configuration of RLFuzz

	Configuration
DQN	LSTM + FC layer
Hidden Dimension	FC layer: 10
Input	Embedding
Double DQL	True
Experience Replay	True
Pre-Training	False
Mini Batch Size	10
Hidden Dimensions	10
Number of Instructions	80
Reward	Clipping + Countering

B. Comparison to the State-of-the-Art

The following section compares the optimized version of RLFuzz, listed in Tab. II, against the state-of-the-art hardware fuzzers TheHuzz [20] and MABFuzz [12].

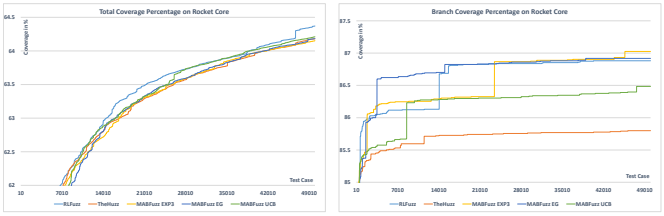

Fig. 6: Coverage percentage of RLFuzz, MABFuzz, and TheHuzz on the Rocket core. The left graph depicts total coverage and the right graph shows branch coverage

On the Rocket core, RLFuzz consistently outperformed TheHuzz and MABFuzz in total coverage, as shown on the left of fig. 6. By the end of the experiment, RLFuzz achieved 64.37% total coverage, exceeding TheHuzz's 64.18% and MABFuzz UCB's 64.20%. For branch coverage, shown on the right of fig. 6, RLFuzz significantly outperformed TheHuzz and MABFuzz UCB but fell short of MABFuzz EXP3 by 0.03%. RLFuzz achieved an average total coverage speedup of 1.18× over TheHuzz, with a peak of 1.5×. For branch coverage, it achieved an average speedup of 16.25× with a peak of 92.98× over TheHuzz.

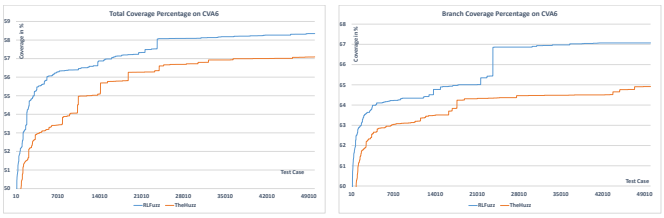

Fig. 7: Coverage percentage of RLFuzz and TheHuzz on CVA6. The left graph depicts total coverage and the right graph shows branch coverage

Unfortunately, the MABFuzz results available to us for CVA6 were obtained under a setup significantly different from the one used for TheHuzz and RLFuzz, with a different number of total achievable coverage points, making a comparison infeasible. As shown on the left of Fig. 7, RLFuzz quickly surpassed TheHuzz in total coverage from the start and maintained a significant lead throughout the experiment. For branch coverage, depicted on the right of Fig. 7, RLFuzz consistently outperformed TheHuzz. Specifically, RLFuzz achieved 58.35% total coverage compared to TheHuzz's 57.09%, reflecting an overall coverage speedup of 2.98x to 6.93x and a branch coverage speedup of 1.74x to 8x.

The BOOM core, which supports out-of-order execution, exhibited significant performance differences between RLFuzz, MABFuzz, and TheHuzz. As shown on the left of

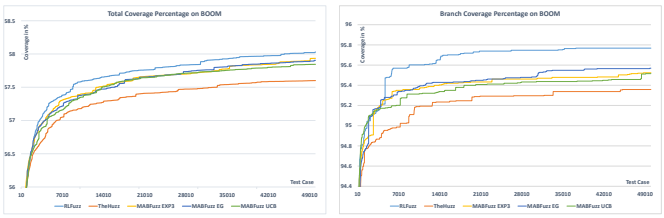

Fig. 8: Coverage percentage RLFuzz, MABFuzz, and TheHuzz on the BOOM core. The left graph depicts total coverage and the right graph shows branch coverage

fig. 8, RLFuzz achieved a steeper total coverage slope in the early stages and maintained consistently higher coverage throughout the experiment, ultimately reaching 58.02% coverage compared to TheHuzz's 57.60% and MABFuzz's 57.93% (for EXP3). For branch coverage, shown on the right of fig. 8, the early advantage of RLFuzz was even more pronounced. Compared to TheHuzz, RLFuzz achieved a total coverage speedup of 2.9x on average, peaking at 4.53x, and a branch coverage speedup averaging 5.58x with a maximum of 17.05x.

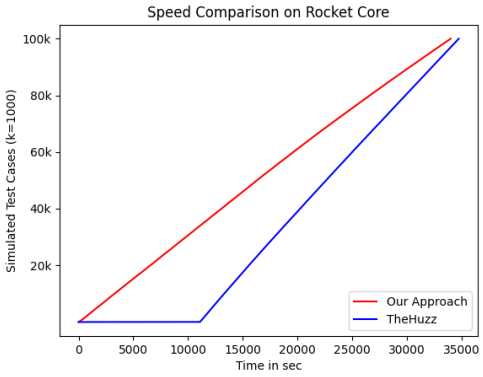

Fig. 9: Speed comparison between RLFuzz and Thehuzz on 100,000 test cases

Finally, Fig. 9 showcases the results of runtime comparison between RLFuzz and TheHuzz on the Rocket core. Unlike TheHuzz [20], RLFuzz does not require a profiling phase, or any similar pertaining or data preparation phases, and therefore it was able to complete the 100,000 test cases 12 minutes faster, despite using more instructions per test case. It is important to note that TheHuzz also requires human interaction at the end of its profiling run, which is not included in Fig. 9. While RLFuzz takes more time per test case due to the higher number of instructions, the lack of a profiling phase and the ability to achieve more and faster coverage make this drawback negligible. Up to 100,000 test cases, RLFuzz is the overall speedier approach. In conclusion, the experiments on three RISC-V cores have consistently shown that RLFuzz not only matches TheHuzz in terms of coverage and efficiency but often significantly outperforms it. The DQL techniques used by RLFuzz offer a significant advantage in hardware fuzzing.

C. Bugs

We compared the output of the CVA6 core with the GRM to identify potential bugs. In total, we uncovered five bugs, three of which (B1, B2, B3) had already been identified by TheHuzz [20].

Bug B1: The first bug involves exception handling in the CVA6 processor, where an illegal instruction exception is sometimes raised instead of a load access fault or instruction access fault exception. Kande et al. [20] traced this issue to the instruction queue of the core. This bug can cause undefined behavior or memory corruption in operating systems that depend on correct exception handling and is therefore classified as a memory and storage issue (CWE-1202).

Bug B2: The second bug concerns the FENCE.I instruction, which in some cases threw an illegal instruction exception in the CVA6 core, even though the GRM executed it without any problems. This bug occurs because the CVA6 core incorrectly expects certain fields to be zero, which should be ignored according to the RISC-V specification. Since FENCE.I ensures cache coherency, this bug could result in cache coherency violations and subsequent data inconsistencies. The issue is therefore classified as a specification violation (CWE-440).

Bug B3: The third bug involves the handling of unknown instructions. While the GRM correctly raises an illegal instruction exception immediately, the CVA6 core sometimes delays raising this exception and continues executing the unknown instructions. This behavior indicates that the CVA6 core may inadvertently execute certain unknown instructions, and is therefore classified as an undocumented feature vulnerability (CWE-1242).

Bug B4: The fourth bug identified by RLFuzz is located in the read logic of the CSR module, specifically related to access to the hardware performance counters (HPCs). The CVA6 core incorrectly allows reads from non-existent HPCs, causing the processor to return undefined values. This issue, previously identified by Chen *et al.* [6], can cause undesired behavior during synthesis and potentially compromise the functionality of the HPCs (CWE-1281). In addition, this vulnerability allows invalid memory access (CWE-125) and is therefore documented as CVE-2022-33021. Unlike the previous bugs, TheHuzz [20] did not discover this issue.

Bug B5: The final bug discovered by RLFuzz affects the CVA6 processor's memory controller. The CVA6 core incorrectly executes load and save instructions with addresses outside the valid range, while the GRM correctly throws load and save access error exceptions. This vulnerability, previously reported by Chen *et al.* [6], is significant because operating systems rely on such exceptions to prevent unauthorized access to isolated memory areas. Without these safeguards, an attacker could exploit this flaw to read (CWE-125) or write (CWE-1252) beyond the intended memory boundaries. This bug was not discovered by TheHuzz [20].

V. Related Work

This section provides an overview of the related literature. First, Sect. V-A presents work that utilized ML techniques

in the context of software fuzzing. Then Sect. V-B discusses several dedicated hardware fuzzers.

A. Machine Learning Guided Fuzzing.

Numerous works have integrated ML techniques into fuzzing. A survey by Wang *et al.* [54] identifies five general approaches where ML can be applied in the fuzzing process: seed file generation, test case generation, test case filtering, mutation operator selection, and exploitability analysis. For seed file generation, Cheng *et al.* [8] presented a method that uses a machine learning framework to optimize seed inputs for an AFL-based fuzzer, specialized for programs that take PDF files as input. In the realm of test case generation, She *et al.* [44] introduced NEUZZ, a technique that defines fuzzing as an optimization problem with the goal of maximizing the number of vulnerabilities found within a given time frame. NEUZZ employs a program smoothing technique based on neural networks to predict the control flow of the target program for a given input. The following work belongs to the third category identified by Wang *et al.* [54]: test case filtering. Drozd *et al.* [9] proposed FuzzerGym, a libFuzzer-based approach that uses RL to optimize the selection of mutation operators. They connected program monitors of the LLVM Sanitizer to a DNN to learn mutation selection strategies directly from the input data, aiming to maximize the number of unique lines covered. Karamcheti *et al.* [21] also used RL for mutation operator selection, modeling it as a multi-armed bandit problem. Each arm corresponds to a single mutation operator, and they used Thompson sampling [1] to learn an efficient mutation operator distribution. Boettinger *et al.* [5] experimented with RL to improve the fuzzing process, formalizing the problem using Markov decision processes and optimizing rewards based on the runtime properties of the tested program. Jauernig *et al.* [18] proposed an evolution-based mutation scheduling approach. This method approximates the ideal probability distribution of mutation operators by optimizing the individual probabilities of selecting specific operators based on feedback from the fuzzer. While these works have effectively used ML, and some even RL, in their methodologies, none have specifically addressed vulnerability detection in hardware, which presents a unique set of challenges.

B. Fuzzing-Based Hardware Testing.

Fuzzing in the hardware domain is typically used in two primary contexts: hardware design verification and vulnerability detection. The former focuses on optimizing hardware verification processes, while the latter aims at uncovering security flaws. However, ML techniques are underexplored in either category. For hardware design verification, Herdt *et al.* [14] use state-of-the-art fuzzing techniques to generate test cases for a RISC-V simulator. Trippel *et al.* [51] use existing software fuzzers to identify hardware inefficiencies by translating the hardware design into a software model. Laeufer *et al.* [23] present RFUZZ, a coverage-directed fuzzing approach for register transfer level (RTL) circuits. Regarding vulnerability detection, Kande *et al.* [20] present TheHuzz, which interacts directly with simulated hardware designs by generating and injecting inputs and comparing the resulting trace information with a GRM. This fuzzer served as the basis for RLFuzz, but unlike RLFuzz, it does not use ML techniques. Chen *et al.* [6] introduced HyPFuzz, an extension of TheHuzz that incorporates formal verification to increase coverage and speed. Muduli *et al.* [36] transformed hardware designs into software models for fuzzing, introducing a new logic for specifying system-level security properties. Hur *et al.* [17] present DifuzzRTL, an RTL fuzzer designed to detect vulnerabilities in CPUs using a unique coverage metric based on control registers and state transitions. Solt *et al.* [46] introduced Cascade, a CPU fuzzer that uses asymmetric ISA presimulation, a technique that entangles control flow with data flow, to generate test cases that result in non-termination if a bug is triggered in the target CPU, thereby improving effective bug detection. Le *et al.* [24] focuses on detecting Trojans in high-level synthesis designs rather than RTLs. RLFuzz fundamentally differs from the aforementioned methods, as none of them use ML techniques, specifically RL, in their approaches. However, recently some RL-based hardware fuzzing approaches have been considered in related literature. MABFuzz, presented by Gohil *et al. et al.* [12] employs multi-armed bandit algorithms to optimize the seed selection of a fuzzer. However, MABFuzz is orthogonal to RLFuzz, as it utilizes a simpler RL mechanism and focuses on a different part of the fuzzing pipeline. Specifically, RLFuzz optimizes the mutation process itself, while MABFuzz targets the efficiency of the inputs to that mutation process. Mondol *et al.* [35] proposed RL-TPG, a framework that combines RL with traditional verification methods to generate intelligent test patterns aimed at detecting hardware vulnerabilities in RTL designs of hardware intellectual properties (IP). Unlike RLFuzz, RL-TPG employs model-free reinforcement learning methods, specifically using two variants: Advantage actor-critic and proximal policy optimization. Furthermore, RL-TPG does not target CPUs, focusing on hardware IPs, whereas RLFuzz is specifically designed for CPU fuzzing with a DQL-based mutation engine.

VI. Conclusion

To overcome limitations of existing hardware fuzzers, we introduce RLFuzz, a novel hardware fuzzer leveraging reinforcement learning (RL) for autonomous, strategic mutation selection, enabling faster and broader processor coverage. RLFuzz directly learns from interactions with the processor to efficiently target untested areas without human intervention. We fine-tuned RLFuzz using various DQL optimization techniques and evaluated it on three RISC-V processors: Rocket Core, CVA6, and BOOM. RLFuzz surpassed TheHuzz [20], achieving up to 1.77% higher total coverage and reaching coverage $2.35\times$ faster on average (peak $6.93\times$). For branch coverage, it achieved an average speedup of $7.85\times$, peaking at $92.98\times$, processing 100,000 test cases faster than TheHuzz.

ACKNOWLEDGEMENT

The research was funded by the European Union's Horizon Europe research and innovation program under grant agreement No 101070537, project CROSSCON (Cross-platform Open Security Stack for Connected Devices). We would like to thank Synopsys for providing us with a VCS license, which supported the completion of the experiments for this research. Additionally, we would also like to thank Gohil *et al.* [12] for sharing the evaluation results of MABFuzz with us, which allowed us to compare our method with their approach.

REFERENCES

[1] Shipra Agrawal and Navin Goyal. Analysis of thompson sampling for the multi-armed bandit problem. In *Conference on learning theory*, pages 39–1. JMLR Workshop and Conference Proceedings, 2012.

[2] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 4:6–2, 2016.

[3] Per Bjesse. What is formal verification? *ACM SIGDA Newsletter*, 35(24):1–es, 2005.

[4] Pallavi Borkar, Chen Chen, Mohamadreza Rostami, Nikhilesh Singh, Rahul Kande, Ahmad-Reza Sadeghi, Chester Rebeiro, and Jeyavijayan Rajendran. Whisperfuzz: White-box fuzzing for detecting and locating timing vulnerabilities in processors, 2024.

[5] Konstantin Böttinger, Patrice Godefroid, and Rishabh Singh. Deep reinforcement fuzzing. In *2018 IEEE Security and Privacy Workshops (SPW)*, pages 116–122. IEEE, 2018.

[6] Chen Chen, Rahul Kande, Nathan Nguyen, Flemming Andersen, Aakash Tyagi, Ahmad-Reza Sadeghi, and Jeyavijayan Rajendran. {HyPFuzz}:{Formal-Assisted} processor fuzzing. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1361–1378, 2023.

[7] Yifei Chen, Lambert Schomaker, and Marco A Wiering. An investigation into the effect of the learning rate on overestimation bias of connectionist q-learning. In *ICAART (2)*, pages 107–118, 2021.

[8] Liang Cheng, Yang Zhang, Yi Zhang, Chen Wu, Zhangtan Li, Yu Fu, and Haisheng Li. Optimizing seed inputs in fuzzing with machine learning. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 244–245. IEEE, 2019.

[9] William Drozd and Michael D Wagner. Fuzzergym: A competitive framework for fuzzing and learning. *arXiv preprint arXiv:1807.07490*, 2018.

[10] Joe W Duran and Simeon C Ntafos. An evaluation of random testing. *IEEE transactions on Software Engineering*, (4):438–444, 1984.

[11] Andrea Fioraldi, Dominik Christian Maier, Dongjia Zhang, and Davide Balzarotti. Libafl: A framework to build modular and reusable fuzzers. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 1051–1065, 2022.

[12] Vasudev Gohil, Rahul Kande, Chen Chen, Ahmad-Reza Sadeghi, and Jeyavijayan Rajendran. Mabfuzz: Multi-armed bandit algorithms for fuzzing processors. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2024.

[13] Google. Hongfuzz. https://github.com/google/honggfuzz, 2016. Accessed: 16-June-2022.

[14] Vladimir Herdt, Daniel Große, and Rolf Drechsler. Closing the RISC-V compliance gap: Looking from the negative testing side. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020.

[15] Mark D Hill, Jon Masters, Parthasarathy Ranganathan, Paul Turner, and John L Hennessy. On the spectre and meltdown processor security vulnerabilities. *IEEE Micro*, 39(2):9–19, 2019.

[16] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[17] Jaewon Hur, Suhwan Song, Dongup Kwon, Eunjin Baek, Jangwoo Kim, and Byoungyoung Lee. Difuzzrtl: Differential fuzz testing to find cpu bugs. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1286–1303. IEEE, 2021.

[18] Patrick Jauernig, Domagoj Jakobovic, Stjepan Picek, Emmanuel Stapf, and Ahmad-Reza Sadeghi. DARWIN: Survival of the Fittest Fuzzing Mutators. *arXiv preprint arXiv:2210.11783*, 2022.

[19] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[20] Rahul Kande, Addison Crump, Garrett Persyn, Patrick Jauernig, Ahmad-Reza Sadeghi, Aakash Tyagi, and Jeyavijayan Rajendran. {TheHuzz}: Instruction fuzzing of processors using {Golden-Reference} models for finding {Software-Exploitable} vulnerabilities. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3219–3236, 2022.

[21] Siddharth Karamcheti, Gideon Mann, and David Rosenberg. Adaptive grey-box fuzz-testing with thompson sampling. In *Proceedings of the 11th ACM workshop on artificial intelligence and security*, pages 37–47, 2018.

[22] Christoph Kern and Mark R Greenstreet. Formal verification in hardware design: a survey. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 4(2):123–193, 1999.

[23] Kevin Laeufer, Jack Koenig, Donggyu Kim, Jonathan Bachrach, and Koushik Sen. RFUZZ: Coverage-directed fuzz testing of RTL on FPGAs. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2018.

[24] Hoang M Le, Daniel Große, Niklas Bruns, and Rolf Drechsler. Detection of hardware trojans in SystemC HLS designs via coverage-guided fuzzing. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 602–605. IEEE, 2019.

[25] Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. Fuzzing: State of the art. *IEEE Transactions on Reliability*, 67(3):1199–1218, 2018.

[26] LLVM. libfuzzer - a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html, 2017. Accessed: 16-June-2022.

[27] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization, 2019.

[28] Panagiotis Louridas. Static code analysis. *Ieee Software*, 23(4):58–61, 2006.

[29] Valentin JM Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering*, 47(11):2312–2331, 2019.

[30] Aakash Maroti. Rbed: Reward based epsilon decay. *arXiv preprint arXiv:1910.13701*, 2019.

[31] Larry Medsker and Lakhmi C Jain. *Recurrent neural networks: design and applications*. CRC press, 1999.

[32] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[33] Audris Mockus, Nachiappan Nagappan, and Trung T Dinh-Trong. Test coverage and post-verification defects: A multiple case study. In *2009 3rd international symposium on empirical software engineering and measurement*, pages 291–301. IEEE, 2009.

[34] Daniel Moghimi. Downfall: Exploiting speculative data gathering. In *32th USENIX Security Symposium (USENIX Security 2023)*, 2023.

[35] Nurun Nahar Mondol, Arash Vafei, Kimia Zamiri Azar, Farimah Farahmandi, and Mark Tehranipoor. Rl-tpg: automated pre-silicon security verification through reinforcement learning-based test pattern generation. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2024.

[36] Sujit Kumar Muduli, Gourav Takhar, and Pramod Subramanyan. Hyperfuzzing for soc security validation. In *Proceedings of the 39th International Conference on Computer-Aided Design*, pages 1–9, 2020.

[37] Thanh Thi Nguyen and Vijay Janapa Reddi. Deep reinforcement learning for cyber security. *IEEE Transactions on Neural Networks and Learning Systems*, 34(8):3779–3795, 2021.

[38] Tavis Ormandy. Zenbleed. https://lock.cmpxchg8b.com/zenbleed.html, 2023. Accessed: 23-Oct-2023.

[39] Riscv-Software-Src. Riscv-isa-sim: Spike, a risc-v isa simulator. https://github.com/riscv-software-src/riscv-isa-sim.

[40] Mohamadreza Rostami, Chen Chen, Rahul Kande, Huimin Li, Jeyavijayan Rajendran, and Ahmad-Reza Sadeghi. Fuzzerfly effect: Hardware fuzzing for memory safety. *IEEE Security and Privacy*, 22(4):76–86, 2024.

[41] Mohamadreza Rostami, Marco Chilese, Shaza Zeitouni, Rahul Kande, Jeyavijayan Rajendran, and Ahmad-Reza Sadeghi. Beyond random

inputs: A novel ml-based hardware fuzzing. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2024.

[42] Mohamadreza Rostami, Shaza Zeitouni, Rahul Kande, Chen Chen, Pouya Mahmoody, Jeyavijayan Rajendran, and Ahmad-Reza 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*, DAC '22, pages 192–197, 2024.

[43] Gary J Saavedra, Kathryn N Rodhouse, Daniel M Dunlavy, and Philip W Kegelmeyer. A review of machine learning applications in fuzzing. *arXiv preprint arXiv:1906.11133*, 2019.

[44] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. Neuzz: Efficient fuzzing with neural program smoothing. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 803–817. IEEE, 2019.

[45] Wilson Snyder, Paul Wasson, Duane Galbi, and et al. Verilator. https://github.com/verilator/verilator.

[46] Flavien Solt, Katharina Ceesay-Seitz, and Kaveh Razavi. Cascade: Cpu fuzzing via intricate program generation. In *Proc. 33rd USENIX Secur. Symp*, pages 1–18, 2024.

[47] Timo Sorsa and Heikki N Koivo. Application of artificial neural networks in process fault diagnosis. *Automatica*, 29(4):843–849, 1993.

[48] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. *Advances in neural information processing systems*, 27, 2014.

[49] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[50] Synopsys. Vcs functional verification solution: Synopsys verification.

[51] Timothy Trippel, Kang G Shin, Alex Chernyakhovsky, Garret Kelly, Dominic Rizzo, and Matthew Hicks. Fuzzing hardware like software. *arXiv preprint arXiv:2102.02308*, 2021.

[52] Ucb-Bar. Chipyard: An agile risc-v soc design framework with in-order cores, out-of-order cores, accelerators, and more. https://github.com/ucb-bar/chipyard.

[53] Michel Verleysen and Damien François. The curse of dimensionality in data mining and time series prediction. In *International work-conference on artificial neural networks*, pages 758–770. Springer, 2005.

[54] Yan Wang, Peng Jia, Luping Liu, Cheng Huang, and Zhonglin Liu. A systematic review of fuzzing based on machine learning techniques. *PloS one*, 15(8):e0237749, 2020.

[55] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8:279–292, 1992.

[56] Florian Zaruba and Luca 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*, 27(11):2629–2640, 2019.

[57] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. Sonicboom: The 3rd generation berkeley out-of-order machine. In *Fourth Workshop on Computer Architecture Research with RISC-V*, volume 5, pages 1–7, 2020.