Beyond Random Inputs: A Novel ML-Based Hardware Fuzzing

Mohamadreza Rostami[‡*], Marco Chilese[‡*], Shaza Zeitouni[*]
Rahul Kande[†], Jeyavijayan Rajendran[†], Ahmad-Reza Sadeghi[*]
[*]Technical University of Darmstadt, [†]Texas A&M University

***Abstract*—Modern computing systems heavily rely on hardware as the root of trust. However, their increasing complexity has given rise to security-critical vulnerabilities that cross-layer attacks can exploit. Traditional hardware vulnerability detection methods, such as random regression and formal verification, have limitations. Random regression, while scalable, is slow in exploring hardware, and formal verification techniques are often concerned with manual effort and state explosions.**

Hardware fuzzing has emerged as an effective approach to exploring and detecting security vulnerabilities in large-scale designs like modern processors. They outperform traditional methods regarding coverage, scalability, and efficiency. However, state-of-the-art fuzzers struggle to achieve comprehensive coverage of intricate hardware designs within a practical timeframe, often falling short of a 70% coverage threshold. To address this challenge, we propose a novel ML-based hardware fuzzer, *ChatFuzz*. Our approach leverages large language models (LLMs) to understand processor language and generate data/control flow entangled yet random machine code sequences. Reinforcement learning (RL) is integrated to guide the input generation process by rewarding the inputs using code coverage metrics.

Utilizing the open-source RISC-V-based RocketCore and BOOM cores as our testbed, *ChatFuzz* achieves 75% condition coverage in RocketCore in just 52 minutes. This contrasts with state-of-the-art fuzzers, which demand a 30-hour timeframe for comparable condition coverage. Notably, our fuzzer can reach a 79.14% condition coverage rate in RocketCore by conducting approximately 199k test cases. In the case of BOOM, *ChatFuzz* accomplishes a remarkable 97.02% condition coverage in 49 minutes. Our analysis identified all detected bugs by TheHuzz, including two new bugs in the RocketCore and discrepancies from the RISC-V ISA Simulator.

I. Introduction

Traditional hardware verification techniques are crucial for ensuring the reliability and correctness of a hardware design, the design under test (DUT), before fabrication. Among these techniques, random regression and formal verification methods are commonly employed. Despite its capacity to accommodate extensive hardware designs, random regression presents a notable efficiency problem as it tends to slow down when exploring the intricacies of a hardware design. Consequently, it encounters difficulties uncovering vulnerabilities within hard-to-reach critical components [6]. On the other hand, formal verification, which aims to ascertain whether a DUT complies with specified/predefined properties [14], is often regarded as an efficient approach for verifying the correctness of hard-to-reach hardware components. However, formal techniques rely heavily on manual effort from domain experts to define the required properties, which can be error-prone and time-consuming. Furthermore, formal verification frequently results in state explosion, rendering it impractical to verify the entire DUT comprehensively [5]. Hardware fuzzing has emerged as a promising approach for not only broadening the exploration of design space but also for revealing security vulnerabilities within intricate designs, including complex processors [3], [8], [9], [13]. To bolster their effectiveness, hardware fuzzers harness coverage data, such as branch conditions, statements, and multiplexers' control registers or signals, for generating test cases and probing diverse hardware behaviors [8]–[11]. When compared to traditional hardware verification techniques, hardware fuzzers have demonstrated broader coverage, enhanced scalability, and efficiency in identifying real-world vulnerabilities that have been associated with privilege escalation and arbitrary code execution attacks [3], [8], [9]. Nonetheless, state-of-the-art fuzzers struggle to achieve comprehensive coverage of intricate hardware designs within a practical timeframe, often falling short of a 70% coverage threshold in complex hardware such as a RISC-V RocketCore processor [1].

Our Contributions. In this paper, we introduce *ChatFuzz*, the first processor fuzzer that leverages machine learning for input generation and improvement with the help of coverage metrics, addressing a critical challenge in the field of processor fuzzing, namely, generating interdependent data/control flow entangled yet random instructions.

Three-Step ML-Based Input Generation. We present a three-step training process, including unsupervised learning to understand machine language structures, reinforcement learning with a disassembler for valid instruction generation, and further reinforcement learning using RTL simulation as a reward agent to improve the coverage.

Significant Speed Enhancement. *ChatFuzz* demonstrably expedites enhancing condition coverage, attaining a coverage level of 74.96% within less than one hour. In contrast, the current leading hardware fuzzer, TheHuzz [9], requires a much longer period of roughly 30 hours to achieve the same coverage, i.e., 34.6× faster. In the case of BOOM, *ChatFuzz* accomplishes a remarkable 97.02% condition coverage in 49 minutes. It is worth noting that TheHuzz exhibits greater efficiency compared to random regression techniques and is approximately 3.33× swifter than DifuzzRTL [8].

Findings. During fuzzing, *ChatFuzz* detects approximately 6K mismatches and identifies more than 100 unique mismatches after automated analysis. These findings include all bugs that were detected by TheHuzz [9] and two new bugs, namely the cache coherency management issue (CWE-1202) and the execution tracing (CWE-440). Moreover, *ChatFuzz* exposes deviations in the behavior of the RocketCore compared to the specifications in the RISC-V ISA. This showcases *ChatFuzz*'s efficiency in delving into the processor search space, thoroughly

‡These authors contributed equally to this work.

979-8-3503-4859-0/DATE24/© 2024 EDAA

investigating even the most detailed corner cases specified in the RISC-V ISA specification.

II. Background & Related Work

A. Fuzzing

Fuzzing provisions a large number of inputs to the program under test to uncover faults, bugs, or vulnerabilities that traditional testing methods may miss [7]. The fuzzer may generate random, malformed, or unusual inputs to test how the program handles them. The initial set of test inputs, also known as *seeds*, can be automatically generated or manually crafted by verification engineers. During each fuzzing round, the fuzzer manipulates the best test inputs from the preceding round using mutation operations like bit/byte flipping, swapping, deleting, or cloning to generate new inputs. In recent years, fuzzing has gained significant attention from the hardware security community due to its numerous advantages over existing verification methods. In particular, fuzzing is highly automatable, cost-effective, scalable to real-world applications, and comprehensively covers the tested application. These factors have contributed to its growing popularity and adoption among researchers and practitioners in the field of software as well as hardware security [2], [8]–[10].

1) Processor Fuzzers: Traditional processor fuzzers such as DifuzzRTL [8] and TheHuzz [9] use code coverage and control register coverage as feedback to guide the mutation process. These fuzzers generate seeds through random generation of instructions and mutate the instructions in the current input to generate new inputs. Recent research also led to hybrid hardware fuzzers such as HyPFuzz [3] and PSOFuzz [4] that combine the capabilities of fuzzers with formal tools and optimization algorithms to improve the coverage achieved. However, these hybrid fuzzers also use the seed generation and mutation engines inherited from traditional processor fuzzers such as TheHuzz [9]. While the seed generator and mutation engine in these fuzzers can identify valid instructions from the ISA, they do not have well-defined feedback to determine a meaningful sequence of instructions that will lead to deep design regions.

B. Machine Learning

1) Reinforcement Learning (RL): RL is a branch of machine learning that studies how agents can learn from their actions and environment feedback to achieve a goal. RL differs from other forms of machine learning, such as supervised and unsupervised learning, in that the agent does not have access to labeled data or explicit rules but must discover the optimal behavior through trial and error. The agent's objective is to find a policy, a function that maps each state to an action that maximizes the expected cumulative reward over time. This is achieved using various algorithms, such as policy-based or actor-critic methods. Proximal Policy Optimization (PPO) is a family of model-free RL algorithms. PPO updates policy parameters for higher expected rewards based on policy gradient methods. Unlike traditional policy gradient methods, PPO employs a clipped surrogate objective function to control policy updates and prevent large deviations from the previous policy, ensuring stability and efficiency. PPO algorithms have been successfully applied to various domains, e.g., natural language generation.

2) Large Language Models (LLMs): LLMs are large ML models for processing and generating natural language text. They leverage neural networks (NN), often using the transformer architecture, to learn from sequential data and capture long dependencies. LLMs can contain billions of parameters (weights), dictating how the model handles input and generates output. LLMs are trained using different learning paradigms, from self-supervised to reinforcement learning, which means that they do not require labeled data or explicit rules but learn from the patterns and structures inherent in the text corpus. LLMs can perform various natural language processing (NLP) tasks, such as recognition, summarization, translation, prediction, and generation. LLMs are general-purpose models that can adapt to different domains and applications with minimal fine-tuning or prompt engineering. In a concurrent work, LLMs have been utilized in software fuzzing [12]. The proposed method creates test cases, particularly for fuzzing compilers, by training a large language model on a task that relies on human-defined prompts to generate and modify test cases. In contrast, our approach does not rely on human interaction during training and is additionally steered by coverage metrics.

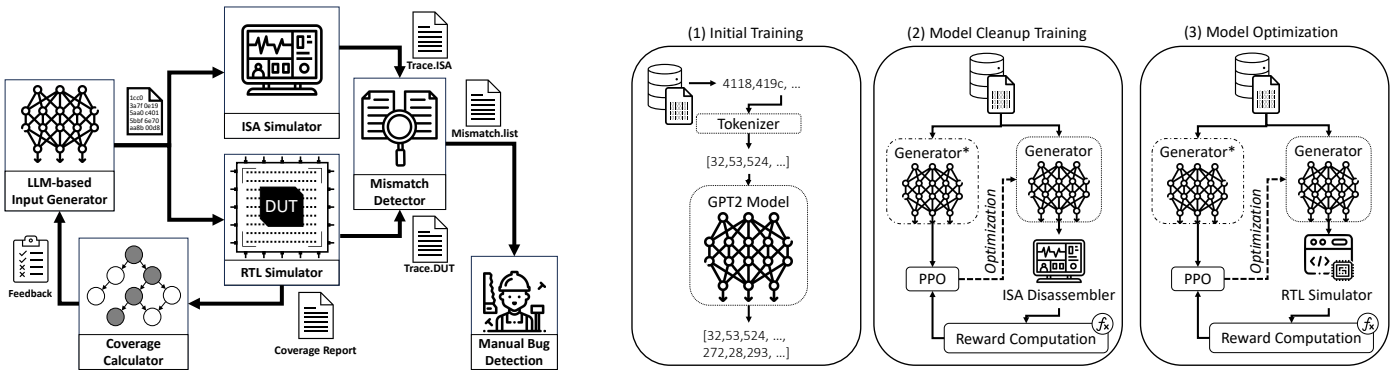
III. *ChatFuzz*

Utilizing recent advancements in LLMs, we propose *ChatFuzz* as an innovative approach for enhancing hardware security. *ChatFuzz* involves training LLMs using machine language (specifically, machine codes) and employing the trained model to generate sequences of pseudo-random yet interconnected instructions for hardware fuzzing. Unlike existing methods, our approach prioritizes creating interdependent data/control flow entangled instruction sequences.

ChatFuzz, illustrated in Figure 1a, comprises several components. The *LLM-based Input Generator* generates instruction sequences for fuzzing the targeted CPU. Details about this component are discussed in subsection III-A and subsection III-B. The *RTL* and *ISA Simulators* execute the given inputs on the targeted CPU and its golden model, respectively, while recording execution traces. For each test input the RTL Simulator reports coverage information, which is utilized by the LLM-based Input Generator to optimize the input generation process. The *Mismatch Detector* compares execution traces to identify mismatches or potential bugs, which are manually inspected for confirmation as elaborated in subsection III-C. In the following, we elucidate the fundamental components of our approach, encompassing *A) the acquisition of a training dataset for instructing the LLM model*, *B) the training process of the LLM model to grasp machine language intricacies*, and *C) the execution of hardware fuzzing and bug detection procedures.*

A. Machine Language Dataset

A major challenge in training LLMs is the need for an extensive training dataset. While collecting data for natural languages like English is relatively easy, it becomes much more complicated for machine languages. To explore this issue, we will investigate two key questions: How do we collect a machine language



(a) Overview of *ChatFuzz* (b) *ChatFuzz*'s training steps

Fig. 1: *ChatFuzz*'s final model results from three consequent training steps: (1) Unsupervised training based on the GPT2 model to learn the inner structure of the machine language; (2) Utilizing a disassembler as a scoring agent during PPO-based RL training, the initial model is refined by cleaning up the learned language and removing bad combinations of instructions; (3) Improving the coverage with a PPO-based RL process where the refined generator is trained through a reward function based on coverage information attained through RTL simulation.

dataset? And how do we represent the machine language data set for LLMs?

1) Training Data Collection: We have two options for collecting a machine language dataset. i) Dynamic data collection. Recording instructions as a program runs is convenient; however, it faces challenges that disrupt data inter-dependency, such as context switches and kernel-related instructions. Rarely executed code sections may also be missing due to conditional constraints, affecting data completeness and interdependence. These issues are more pronounced when collecting data from complex programs, e.g., the Linux kernel, where repetitive instructions and infrequent execution of critical sections add complexity to data collection and interdependence. ii) Static data collection directly gathers training data from fixed sources like GUI-compiled code, avoiding dynamic program execution complexities such as context switches and kernel-related instructions. This approach keeps the collected data isolated from OS concurrent tasks' interference, preserving intrinsic data relationships as coded in the source. Static data collection also comprehensively captures all code segments, including rarely executed blocks, without relying on their activation during program runtime.

In this work, we opted for static data collection as it effectively overcomes the challenges posed by instruction inter-dependency and code block rarity encountered in dynamic data collection.

2) Training Data Representation: This step is challenging due to several factors, including the presence of metadata (such as headers and linking information) within machine codes resulting from program compilation. Metadata can introduce complexity and ambiguity, potentially hindering the LLM model's ability to learn the language effectively and maintain the meaningfulness and interdependence of the training dataset. To address this challenge, as illustrated in subsection III-B, we disassemble the binaries generated from program compilation and automatically identify the start and end locations of functions within the disassembled files. We then include the machine code of each function as an individual entry in the training dataset designed for our LLM model. This approach ensures that each function, being associated with a distinct responsibility and meaning, contributes to the creation of a training set characterized by a high degree of inter-dependency among the instructions and their sequencing.

B. Training of LLM Model

Figure 1b depicts the ML subsystem. Our approach involves a structured three-step pipeline, each phase dedicated to training the language model, advancing steadily towards our ultimate objective.

1) Initial Training: In this step, the model is initialized and trained using the collected dataset. This step aims to learn the language utilized by the CPU. For this purpose, we train a tokenizer on the full ISA. The tokenizer then prepares the inputs for the model as shown in Figure 1b(1).

2) Model Language Cleanup: Once the initial training is completed, the model can commit numerous errors in the text generation (e.g., wrong/illegal combinations of instructions). Therefore, a refinement phase is crucial. Hence, at this stage of the pipeline, Figure 1b(2), our goal is to clean up the generations of the trained model, enforcing the correct instruction associations to minimize the number of ineffective generations. For this purpose, we designed an RL process that leverages, as a reward agent, the ISA disassembler (cf., subsection III-A). We avoid using, as commonly done, a probabilistic scorer, such as a neural network, for the rewarding task to prevent uncertainty and reduce errors. Employing a deterministic reward agent, we can provide the model with more precise guidance during the training, leading to better optimization policies and more precise model updates. This step helps avoid unnecessary CPU simulation of bad/malformed data and thus improves the overall performance of our fuzzer.

3) Model Optimization: Finally, we aim to improve the training of our LLM to achieve our goal, which is generating sequences of pseudo-random yet interconnected instructions that lead to better CPU coverage. To do so, we employed another RL-based training step, utilizing a deterministic reward agent similar to

the previous step. In this case, the reward function embeds the scores provided as fuzzing loop feedback comprising hardware coverage information collected during the simulation of the generated data on the targeted CPU as shown in Figure 1b(3). We performed the previous steps for RISC-V ISA. However, it is worth noting that the approach described above is generalizable to any CPU architecture.

C. Hardware Fuzzing and Bug Detection

After training the LLM model (cf., subsection III-B), we initiate the fuzzing loop. As delineated in Figure 1a, the LLM model generates a batch of test inputs, where each entry represents a list of instructions. These entries are then executed on the golden model and the targeted CPU using the ISA and RTL Simulators, respectively. The resulting two execution traces of each entry are analyzed by the Mismatch Detector to identify traces' discrepancies, which are documented for subsequent manual inspection as part of the bug detection process.
Additionally, the RTL Simulator reports hardware coverage metrics to the Coverage Calculator, which computes standalone, overall, and incremental coverage values for each entry as described in section IV. These values are then used to score each entry generated by the LLM model, leading to a precise evaluation of the entries and guiding the LLM model to generate further inputs that have the potential to enhance the coverage.

IV. Implementation

In this section, we will provide details on the implementation of *ChatFuzz* components. We deployed Synopsys VCS and the Spike simulator for RTL and RISC-V ISA, i.e., the golden model, simulations. Additionally, we developed custom components for Mismatch Detection and Coverage Calculation.

A. Mismatch Detection

This component uses differential testing to flag potential vulnerabilities in the targeted CPU. It compares the architectural state changes between the targeted CPU and its golden model when both run the same input and compiles a report with uniquely identified discrepancies. Thus, effectively reducing the manual workload for verification engineers. This is particularly advantageous when multiple instances of the same bug generate numerous mismatches. Further, verification engineers can add filters to the Mismatch Detector in the form of architectural state values that will allow filtering out most of the false positive mismatches and accelerate vulnerability detection.

B. Coverage Calculation

This component is responsible for receiving the coverage reports from the RTL simulator, i.e., Synopsys VCS in our implementation. Subsequently, the coverage reports undergo parsing, facilitating the calculation of three key values: standalone coverage, incremental coverage, and total coverage, for each coverage metric. *Stand-alone* coverage indicates the number of coverage points attained by the input under consideration. *Incremental* coverage gauges the quantity of newly achieved coverage points by the current input compared to the total coverage points recorded in the previous batch. Meanwhile, *total* coverage encapsulates the cumulative tally of coverage points attained thus far, incorporating the contributions of all inputs generated by the LLM model. These values are deployed in the calculation of scores assigned to each test input generated by the LLM-based input generator, thereby facilitating a comprehensive evaluation of the generated inputs, i.e., test inputs, with respect to their coverage effectiveness.

C. LLM-based Input Generation

The ML part of *ChatFuzz* was fully implemented in Python with the use of the frameworks Pytorch (www.pytorch.org) and Huggingface (www.huggingface.co). The use of Huggingface is considered the standard for NLP-related tasks. Specifically, we leveraged its implementations of the tokenizer, the large language model (more precisely, of GPT2 family), and the PPO algorithm for the RL pipeline. All the experiments were conducted on a high-performance server. In the following, we describe the main steps designed for achieving our goal, principally depicted in Figure 1b.
1) Initial Training: The initial step in designing an NLP pipeline is defining the dictionary and its corresponding tokenizer. The tokenizer translates words (i.e., instructions) into tokens by encoding input text into an array of dictionary word indices, always serving as an intermediary step between the dataset and the language model. Decoding, on the other hand, translates an array of tokens back into text (i.e., sequence of instructions).
Next, we trained the selected model to understand the inner workings of the machine language, including grammar and instructions relationships. During the training, the model receives an input fragment of valid test vectors from our collected dataset, resembling $\sim$ 500K test vectors extracted by compiling the Linux Kernel, and learns how to complete it.
2) Model Language Cleanup: After the initial training, the model is able to utilize the CPU's language. However, having the full ISA available as a dictionary, the model will easily commit errors, generating illegal associations of instructions that a disassembler can easily detect. To overcome this limitation, which would significantly impact the quality of the end generations, we decided to perform training through a PPO-based RL, where the scoring agent is the RISC-V disassembler. The reward function is designed in such a way that correct generations are incentivized, and generations with illegal instructions are as penalized as many invalid instructions are present in the generated test vector:

$$f(\text{GenText}_i) = N_i - 5 * \text{Invalid}_i \quad (1)$$

where N_i is the number of instructions generated at time i for GenText_i, and Invalid_i is the number of invalid instructions present in GenText_i.
For the training, we utilized a dataset of 51.2K samples extracted from the larger main dataset. For each sample, we randomly selected the initial 2 to 5 instructions as input for the LLM. The model then completes the test vectors using its learned logic.
The training consists of 30 epochs. We monitored the PPO algorithm's loss, the Kullback-Leibler divergence between op-

timization policies, and the mean rewards assigned at each step to assess the training progress.

3) Model Optimization: Once the model went through two steps of training and the number of errors in the generations was sensibly reduced, we proceeded with the final training, where we wanted to carefully drive the model towards the exploration of the targeted CPU (i.e., increasing the reference coverage) through a PPO-based RL process. In this case, the reward function, based on the values reported by the Coverage Calculator, takes into account the overall knowledge of architecture until the i-th step, the incremental coverage (i.e., whether there was an improvement), and stand-alone coverage (i.e., coverage of the i-th sample). In practice, the reward function guides the search direction toward generations that increase the coverage by giving a bonus and penalizing (i.e., assigning a negative reward) those that do not produce any improvement. This reward function, ultimately, pushes the model to explore more in the direction of interesting generations. Moreover, analogously to the previous step, this training takes place with the same strategy. We utilize the same sampled dataset of 51.2K samples as input. In this case, the training is designed to last at most 15 epochs, during which the values reported by the coverage calculator are used for the reward computation.

V. Evaluation

We used ten instances of Synopsys VCS as a simulator and measured the effectiveness of our solution using the condition coverage metric provided by Synopsys VCS. It is imperative that this feedback captures new hardware behavior and functionalities during fuzzing. Condition coverage aligns with this goal, correlating the satisfaction of hardware design conditions with realizing new functional behaviors. An exemplary instance is fulfilling conditions leading to privilege-level transitions, such as shifting from the user to the supervisory level. We have chosen the widely utilized RISC-V RocketCore and Boom processors, renowned as preeminent open-source processors within the RISC-V ecosystem. In evaluating RISC-V processors, we employed the Chipyard simulation environment, which facilitates the assessment of diverse processors and ensures a uniform testing arena. Each experiment was executed over 24 hours and repeated three times to underscore the robustness and consistency of our findings.

A. Design Coverage

Our analysis revealed that both *ChatFuzz* and TheHuzz incur similar runtime overhead. Nevertheless, when considering an equivalent number of generated tests (1.8K) with same number of instructions, *ChatFuzz* achieved a condition coverage of 74.96%, while TheHuzz reached 67.4%. Remarkably, TheHuzz required around 30 hours to reach a 75% coverage rate, i.e., *ChatFuzz* achieved the same amount of coverage 34.6× faster. Ultimately, *ChatFuzz* achieved a condition coverage rate of 79.14% by generating 199k test cases, while TheHuzz [9] attained a condition coverage rate of 76.7% for the same number of test cases. Furthermore, *ChatFuzz* accomplishes a remarkable 97.02% condition coverage in 49 minutes while running experiments on the Boom processor. Figure 2 provides

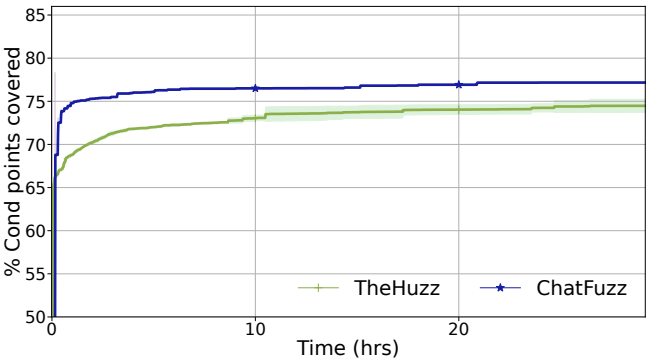


Fig. 2: Coverage analysis of TheHuzz [9] and *ChatFuzz* over time for RocketCore.

visual representation of the condition coverage for *ChatFuzz* and TheHuzz during 24 hours of RocketCore fuzzing.

B. Findings

In the initial stage of our mismatch detection process *ChatFuzz* effectively identified 5,866 instances of disparities within the execution traces originating from the RISC-V ISA simulator and the RocketCore. Subsequently, these identified mismatches underwent a secondary filtration process, separating more than 100 unique mismatches. This filtration process was executed in an automated fashion. Following this, we embarked on a detailed manual analysis of these unique mismatches, the summaries of which are presented below.

1) Bug1: According to the RISC-V specification [15], when there are modifications made to the instruction memory, it is imperative for the software to manage cache coherency through the utilization of the `FENCE.I` instruction. Neglecting this cache coherency management can lead to unforeseeable consequences, wherein processors may rely on outdated data and execute instructions incorrectly. During testing with a generated input program by our fuzzing tool that modified the instruction memory but did not incorporate the `FENCE.I` instruction, an inconsistency was identified in the trace logs of the RocketCore processor and Spike. This disparity could have been prevented if the RISC-V specification or the RocketCore processor could detect violations of cache coherency at the hardware level. This bug has the potential to introduce cache coherency problems in software executed on the RocketCore processor, which might go unnoticed if the `FENCE.I` instruction is misused, ultimately resulting in a memory and storage vulnerability identified as CWE-1202.

2) Bug2: RISC-V specification consists of arithmetic instructions such as multiply and divide [15] that compute a value using the operand registers and update the result in the destination register. The RocketCore processor and ISA simulator behave accordingly when executing the multiply and divide instructions. However, the tracer module in RocketCore is not outputting the write to the destination register in RocketCore's trace output, resulting in Bug2. This bug may not have security consequences as it is present in the debug components of RocketCore. However, bugs like this can mask other security vulnerabilities that can otherwise be detected with the correct trace output information (CWE-440).

3) Other Findings: In conjunction with its capacity for vulnerability detection, our tool has brought to light compelling disparities between the target processor and Spike. While these disparities do not signify security vulnerabilities, they highlight the tool's capabilities in comprehensively examining the target processor. These discrepancies represent exceptional cases within the RISC-V specification and highlight the effectiveness of our approach in exploring the DUT search space. This is achieved by generating interdependent and data/control flow entangled instructions, as opposed to the conventional use of random instructions employed by state-of-the-art hardware fuzzers. We will elucidate the three most significant ones below.

Finding1. In line with the RISC-V specification [15], when an instruction triggers multiple synchronous exceptions, the higher-priority exception is logged in the `mcause` register. The priority hierarchy established in the RISC-V privilege specification places the `Load/store/AMO address misaligned` exception above the `Load/store/AMO access fault` exception. In our fuzz testing using *ChatFuzz*, two test cases emerged. In the first, both *Load access fault* and *Load address misaligned* exceptions were simultaneously raised. In contrast, the second test case triggered both *Store access fault* and *Store address misaligned* exceptions concurrently. Notably, Spike responded with the `Load/Store address misaligned` exception, while RocketCore issued the `Load/Store address fault` exception.

Finding2. In another example, *ChatFuzz* generated a pair of atomic instructions, such as `AMOOR.D`, in which it employed `R0` as a temporary location for loading data from memory, designated as `rd`. Interestingly, our tool observed that this atomic instruction appeared to function as expected, with `R0` receiving data—a behavior seemingly at odds with the RISC-V specification [15]. Upon further investigation, we realized that this behavior represents a corner case within the RISC-V specification [15]. It is conceivable that developers, in pursuit of optimization, could implement the `AMOOR.D` operation within the memory controller. Consequently, if a user specifies `R0` as the destination register (`rd`) for this instruction, the memory controller may perform the atomic operation as intended.

Finding3. Another notable scenario relates to the behavior of the RocketCore processor, particularly in its treatment of the `R0` register, compared to the Spike ISA simulator. According to the RISC-V ISA specifications, the `R0` register is expected to maintain a constant value of zero, implying immunity to write operations. However, our analysis unveiled that in the execution traces generated by the RocketCore, there are occurrences of attempted writes to the `R0` register within specific sequences of instructions. It is important to note that this discrepancy is solely observed in the output traces and does not affect the functionality of RocketCore.

VI. Conclusion

We introduced *ChatFuzz*, a novel hardware fuzzer that utilizes large language models to learn machine language and generate complex, interdependent, data/control flow entangled and pseudo-random test cases. Our approach significantly improves condition coverage, reaching 74.96% in less than an hour, compared to the 30 hours required by leading hardware fuzzers, i.e., *ChatFuzz* achieved the same amount of coverage 34.6× faster. Also, in the case of Boom, *ChatFuzz* accomplishes a remarkable 97.02% condition coverage in 49 minutes. *ChatFuzz* has successfully identified more than 100 unique mismatches, revealed two novel bugs, and exposed deviations in RocketCore behavior compared to the golden model, even in intricate corner cases specified in the RISC-V ISA specification. These results highlight *ChatFuzz*'s effectiveness in exploring processor vulnerabilities, offering a faster and more comprehensive approach to hardware security and testing.

Acknowledgement

Our research work was partially funded by the Intel's Scalable Assurance Program, Deutsche Forschungsgemeinschaft (DFG) – SFB 1119 – 236615297, the European Union under Horizon Europe Programme – Grant Agreement 101070537 – CrossCon, the European Research Council under the ERC Programme - Grant 101055025 - HYDRANOS, and the US Office of Naval Research (ONR Award #N00014-18-1-2058). 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, the European Research Council, or the US Government.

References

[1] Asanović et al. The Rocket Chip Generator. (UCB/EECS-2016-17), 2016.
[2] Canakci et al. DirectFuzz: Automated test generation for RTL designs using directed graybox fuzzing. In *Design Automation Conference (DAC)*. IEEE Computer Society, 2021.
[3] Chen et al. HyPFuzz: Formal-assisted processor fuzzing. *arXiv preprint arXiv:2304.02485*, 2023.
[4] Chen et al. Psofuzz: Fuzzing processors with particle swarm optimization. *arXiv preprint arXiv:2307.14480*, 2023.
[5] Clarke et al. Progress on the State Explosion Problem in Model Checking. *Informatics*, 2001.
[6] Dessouky et al. HardFails: Insights into Software-Exploitable Hardware Bugs. In *USENIX Security Symposium*. USENIX Association, 2019.
[7] Fioraldi et al. Afl++: Combining incremental steps of fuzzing research. In *USENIX Workshop on Offensive Technologies (WOOT)*. USENIX Association, 2020.
[8] Hur et al. DifuzzRTL: Differential fuzz testing to find cpu bugs. In *IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, 2021.
[9] Kande et al. TheHuzz: Instruction fuzzing of processors using Golden-Reference models for finding Software-Exploitable vulnerabilities. In *USENIX Security Symposium*. USENIX Association, 2022.
[10] Laeufer et al. Rfuzz: coverage-directed fuzz testing of rtl on fpgas. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE Computer Society, 2018.
[11] Trippel et al. Fuzzing hardware like software. In *USENIX Security Symposium*. USENIX Association, 2022.
[12] Xia et al. Universal fuzzing via large language models. *arXiv preprint arXiv:2308.04748*, 2023.
[13] Xu et al. MorFuzz: Fuzzing processor via runtime instruction morphing enhanced synchronizable co-simulation. In *USENIX Security Symposium*. USENIX Association, 2023.
[14] Yuji Kukimoto. Introduction to Formal Verification, 2011.
[15] RISC-V. The risc-v instruction set manual volume i: Unprivileged isa, 2019.