MABFuzz: Multi-Armed Bandit Algorithms for Fuzzing Processors

Vasudev Gohil*, Rahul Kande*, Chen Chen*, Ahmad-Reza Sadeghi†, and Jeyavijayan Rajendran*
*Texas A&M University, †Technische Universität Darmstadt
*{gohil.vasudev, rahulkande, chenc, jv.rajendran}@tamu.edu,
†{ahmad.sadeghi}@trust.tu-darmstadt.de

***Abstract*—As the complexities of processors keep increasing, the task of effectively verifying their integrity and security becomes ever more daunting. The intricate web of instructions, microarchitectural features, and interdependencies woven into modern processors pose a formidable challenge for even the most diligent verification and security engineers. To tackle this growing concern, recently, researchers have developed fuzzing techniques explicitly tailored for hardware processors. However, a prevailing issue with these hardware fuzzers is their heavy reliance on static strategies to make decisions in their algorithms. To address this problem, we develop a novel dynamic and adaptive decision-making framework, `MABFuzz`, that uses multi-armed bandit (MAB) algorithms to fuzz processors. `MABFuzz` is agnostic to, and hence, applicable to, any existing hardware fuzzer. In the process of designing `MABFuzz`, we encounter challenges related to the compatibility of MAB algorithms with fuzzers and maximizing their efficacy for fuzzing. We overcome these challenges by modifying the fuzzing process and tailoring MAB algorithms to accommodate special requirements for hardware fuzzing.**

We integrate three widely used MAB algorithms in a state-of-the-art hardware fuzzer and evaluate them on three popular RISC-V-based processors. Experimental results demonstrate the ability of `MABFuzz` to cover a broader spectrum of processors' intricate landscapes and doing so with remarkable efficiency. In particular, `MABFuzz` achieves an average speedup of $53.72\times$ in detecting vulnerabilities and an average speedup of $3.11\times$ in achieving coverage compared to a state-of-the-art technique.

***Index Terms*—Multi-Armed Bandits, Hardware Fuzzing, Hardware Vulnerability Detection**

I. Introduction

Modern processors are getting increasingly complex with the goal of offloading software functionalities such as cryptographic evaluations to hardware, all while optimizing power, performance, and area. This complexity, paired with the shrinking verification time, poses an immense challenge to existing verification tools. Insufficient verification will lead to a rise in security-critical hardware vulnerabilities that can be exploited by cross-layer attacks, threatening the reputation of companies, and undermining the safety, security, and resilience of critical infrastructure [1]. For example, the broken hyper-threading vulnerability in Intel Skylake and Kaby Lake leads to data corruption, or data loss [2]. The *Zenbleed* vulnerability in AMD Zen 2, allows attackers to potentially access secret data [3]. Billions of dollars are being invested by both industry and government entities in research and development efforts aimed at fortifying the security posture of these critical components of modern computing infrastructure.

A. Hardware Fuzzing

One such approach that has gained prominence is hardware fuzzing that employs automated testing techniques to probe processors for vulnerabilities. Several hardware fuzzers have already been developed [4]–[10]. These fuzzers have proven to be faster, automated, and scalable compared to traditional verification techniques such as random regression, directed testing, or formal verification [5]–[8]. Even commercial enterprises such as Intel and Google are actively developing new and efficient hardware fuzzers for vulnerability detection [6], [11].

B. Limitations of Existing Hardware Fuzzers

Although hardware fuzzing is promising, existing hardware fuzzers are inefficient because they make many static decisions disregarding the design complexity and the design space explored. Recent research has shown that replacing the static decisions in the mutation engine of fuzzer with a dynamic strategy is likely to cover more points in the design [10]. However, fuzzers make many other static decisions that are not under the scope of [10]. For example, [7] selects the tests from its database in a static first-in-first-out method and does not prioritize selecting the tests with more potential first. To address this limitation of static strategies in fuzzers, we develop an approach to equip any hardware fuzzer with a dynamic decision-making technique, multi-armed bandit (MAB) algorithms.

MAB algorithms are devised for dynamic decision-making under uncertainty and striking a good balance between the exploration of novel decisions and exploitation of known, well-performing decisions [12]. Researchers have used MAB algorithms to develop promising solutions to several security problems such as intrusion detection [13] and securing cyber-physical systems [14]. However, to the best of our knowledge, the potential of MAB algorithms in hardware fuzzing has not been explored. Through this work, we aim to fill this gap by using MAB algorithms to optimize the selection of input tests and speed up vulnerability detection.

C. Our Contributions

Applying MAB algorithms to hardware fuzzing is not trivial. This is because new coverage achieved by fuzzers decreases with time whereas traditional MAB algorithms are designed to increase returns with time [12], [15], [16]. So, to ensure that our technique is efficient and effective, we need to (i) devise a way to monitor the coverage returns from the fuzzer, and (ii) modify MAB algorithms so that they are compatible with this

979-8-3503-4859-0/DATE24/© 2024 EDAA

diminishing returns property of hardware fuzzers. We designed MABFuzz by addressing these challenges which resulted in faster vulnerability detection speed and covered more hardware. Overall, the main contributions of this work are as follows:

1) To the best of our knowledge, we develop the first technique that uses MAB algorithms, MABFuzz, to select test inputs in hardware fuzzers.
2) We overcome challenges in adapting MAB to hardware fuzzers. In particular, we use monitors to identify saturated inputs and modify MAB algorithms to handle such inputs.
3) We integrate three widely-used MAB algorithms in a fuzzer, demonstrating the agnostic nature of our technique.
4) We evaluate MABFuzz on three widely-used, open-sourced RISC-V processors and achieve an average speedup of $53.72\times$ in detecting vulnerabilities and an average speedup of $3.11\times$ in achieving coverage compared to the state-of-the-art simulation-based fuzzer.

II. Background

A. Hardware Processor Fuzzers

Hardware fuzzing is a regression-based vulnerability detection technique that tests the target hardware, such as a processor, by iteratively generating inputs, termed *tests* (i.e., a sequence of instructions or binary executables) and executing on the target [7], [8], [10]. It generates the first set of these inputs, termed *seeds* by randomly creating instructions. These seeds populate the fuzzer's input database, i.e., *test pool*. Fuzzer selects *tests* from the input pool, simulates the target to collect coverage feedback, and trace log outputs. The coverage feedback comprises information about the activity in hardware such as toggling a register bit or entering a control path in the form of coverage *points*. Fuzzer *mutates* tests (a.k.a *interesting tests*) that cause new activity in the hardware, i.e., cover new points. This mutation performs bit manipulation operations such as flipping a bit on the current test to create new tests and adds them to the input pool. Most processor fuzzers use differential testing to detect vulnerabilities [5], [7], [8], [10]. This technique compares the changes in the architectural state of the hardware processor and a reference model for every executed instruction to detect mismatches and flag potential vulnerabilities. They use an ISA simulator, such as SPIKE [17] for RISC-V ISA [18], as the reference model. This reference model is run with the same input as the hardware processor. Hence, it generates the expected correct architectural state to compare with the state of the hardware processor.

B. Multi-Armed Bandits (MABs)

MABs represent a framework for decision-making under uncertainty. In a typical MAB problem, an agent is faced with a set of choices to take, or *arms* to pull, each of which yields an uncertain reward. The agent's challenge lies in balancing the exploration of these arms to learn their reward probabilities and the exploitation of known information to maximize cumulative rewards. This exploration-exploitation trade-off is pervasive in numerous real-world applications, including online advertising, clinical trials, and recommendation systems.

There has been extensive research on the MAB problem, resulting in a diverse range of algorithms. These algorithms aim to optimize decision-making strategies based on the evolving understanding of arms' performances. Researchers have used MABs and its more general form, reinforcement learning, to develop promising solutions for other problems in security research [19]–[23]. However, to the best of our knowledge, this is the first work to use MAB algorithms for finding vulnerabilities in hardware using fuzzing.

III. MABFuzz: Multi-Armed Bandit Algorithms for Fuzzing Processors

In this section, we first describe why MAB algorithms are a good fit for finding vulnerabilities using hardware fuzzing. Then, we detail our preliminary formulation. Next, we explain some challenges the preliminary formulation faces and our solutions. Finally, we detail our final MABFuzz formulation.

A. Why MAB Algorithms?

As described in Sec. II-A, hardware fuzzing involves several moving pieces, e.g., generating seeds, selecting tests to simulate, prioritizing interesting tests to mutate to obtain high coverage, deciding mutation operators to apply to these tests, and so on. Most existing processor fuzzers either make these decisions randomly or use static strategies [5], [7], [8]. However, recent research has shown this is not ideal. For instance, dynamically selecting mutation operators is likely to cover more design points than static probabilities [10]. A hardware fuzzer should have two key features to obtain higher coverage and find vulnerabilities faster, as explained next.

Dynamic Decision-Making Under Uncertainty. An effective hardware fuzzer must continually adapt its input generation strategies to explore untested design regions (which also change with time) within the hardware's logic and functionalities. It must make dynamic decisions when faced with the ambiguity of which tests are most likely to unveil vulnerabilities.

Balanced Trade-Off Between Exploration and Exploitation. Another feature required from a good hardware fuzzer is its ability to strike the delicate equilibrium between exploration and exploitation, which is driven by the inherent constraints of limited resources. The fuzzer must judiciously allocate its constrained resources, such as time or number of tests, to explore new areas of the design while capitalizing on what is already known, effectively maximizing coverage. This careful balance ensures not only the discovery of hidden vulnerabilities but also the efficient utilization of scarce resources.

Tasks that require these features are the kind of problems that MAB algorithms are ideal for solving—**sequential decision-making under uncertainty** and striking a **balance between exploration and exploitation** in a search space. Hence, next, we architect a preliminary framework to fuzz processors with MAB algorithms to find vulnerabilities.

B. Preliminary Formulation

There are several avenues for decision-making in hardware fuzzers where MAB algorithms can be applied. One of the most critical points of these avenues is deciding which seeds to pick

```
1  reg MyReg1;
2  reg MyReg2;
3  ...
4  always@(posedge clk) begin
5     if (MyReg1)
6        cov1 // coverage point 1
7     if (!MyReg1)
8        cov2 // coverage point 2
9     if (MyReg2)
10       cov3 // coverage point 3
11    ...
12 end
```

Listing 1: Code snippet for motivational example

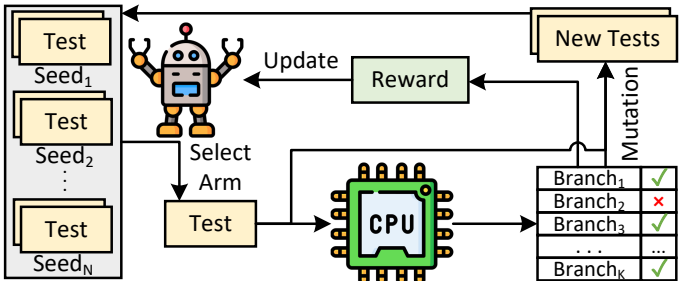

Fig. 1: High-level flow of the preliminary formulation.

from a pool of seeds. Most existing hardware fuzzers select seeds uniformly at random or select seeds greedily based on their historical coverage [7], [8]. This is not ideal, as different seeds can have vastly different impacts on covering new points in hardware. The following example demonstrates this.

Motivational Example. Consider the code shown in Listing 1 with three target coverage points, `cov1`, `cov2`, and `cov3`. Next, suppose we have two seeds, S_1 and S_2, such that S_1 contains instructions that control the values of `MyReg1`, and S_2 contains instructions that control the value of `MyReg2`. Also, suppose that in the past, S_1 covered `cov1` and `cov2`, and S_2 has not been selected so far. If we pick seeds greedily (i.e., exploit seeds that have performed well in the past), we would pick S_1 and not cover `cov3`. On the other hand, if we use a dynamic strategy that balances well between the exploitation of known information (i.e., select seeds that have performed well historically) and exploration for updating information (i.e., try out untested or less tested seeds), we would likely select S_2, resulting in covering a novel region of the design, `cov3`. Note that although this is a hypothetical example, the implications are applicable in the case of real-world processors, too, as evidenced by the V7 vulnerability in our results in Sec. IV-B.

We now map the seed selection problem in hardware fuzzing to an MAB problem by defining the different aspects of MABs in terms of hardware fuzzing.

- **Arms** $\mathcal{A}$ is the set of arms from which the agent can choose. Each individual arm, $a_i \in \mathcal{A}$, corresponds to a different seed in the set of seeds of the fuzzer. Note that at each time step, t, the agent can only select, i.e., pull, one arm.
- **Reward Distributions** $\mathcal{R}$ indicate the likelihood of rewards for the arms in $\mathcal{A}$. For hardware fuzzers, the reward for pulling an arm, i.e., selecting a seed, changes with time. So, we denote the reward for pulling an arm a_i at time t, as $R_t(a_i)$. Since the objective is to maximize coverage, we define this reward as:

$$R_t(a_i) = \alpha \times |cov_t^L(a_i)| + (1-\alpha) \times |cov_t^G(a_i)|, \text{ where}$$
$$cov_t^L(a_i) = \{c_j \mid c_j \text{ is covered by } a_i \text{ at } t \text{ but not before } t\}$$
$$cov_t^G(a_i) = cov_t^L(a_i) \cap \{c_j \mid c_j \text{ is not covered before } t\}$$

Here, $cov_t^L(a_i)$ denotes the set of coverage points that are covered by a_i in the current step, t, but have not been covered by a_i in the past (superscript L denotes local). $cov_t^G(a_i)$ denotes the set of coverage points that are covered by a_i in the current step, t, and have not been covered by any test of any arm in the past (superscript G denotes global). Thus, the reward, $R_t(a_i)$, is a weighted sum of the number of global and local new coverage points. The parameter $\alpha \in [0, 1]$ controls the weight, i.e., the relative importance of finding new coverage points not covered by arm a_i and those not covered by any arm.

Fig. 1 illustrates the high-level flow of this preliminary formulation. First, a set of seeds, with one seed for each arm, is created randomly. Each arm has a test pool consisting of its tests. Then, the agent picks one of the arms according to the initial (random) arm selection probabilities of an MAB algorithm. Then, a test from the selected arm's test pool is simulated to obtain coverage data. Next, using this coverage data, the original test is mutated to generate new tests, which are added to the arm's pool. Additionally, the coverage information is also used to provide the reward to the agent for the chosen arm. The MAB algorithm uses the obtained reward to update the arm selection probabilities. Then, the agent again selects an arm according to the updated arm selection probabilities, and the cycle continues. Through a balance of exploration and exploitation, the agent tries to maximize the design coverage.

Since the application of MAB for hardware fuzzing is unexplored, we designed `MABFuzz` to support any MAB algorithm. In our experiments, we use three different widely-used MAB algorithms, ε-greedy [12], UCB [15], and EXP3 [16], that have achieved good performance historically.

C. Tackling Diminishing Marginal Rewards

The preliminary formulation described above works to an extent, but a crucial property of hardware fuzzing is not taken into account by the MAB algorithms: after a few time steps or iterations, the number of new coverage points obtained by mutating tests decreases drastically with time. To address this, we (i) modify the fuzzing loop to check when an arm is no longer effective and (ii) modify the MAB algorithms to accommodate this peculiar property of hardware fuzzing.

Recognizing and Handling Saturated Arms. To identify when an arm is not capable of covering novel regions of the design, we monitor the coverage of the arm over the most recent window of γ iterations when the arm was picked by the agent. If the coverage does not increase in that γ-window, we mark the arm as depleted and replace it with a new arm.[1] We call this

[1] γ controls the trade-off between possibly covering deeper regions of the design at the cost of time (large γ) versus trying to explore other novel regions in the design while potentially sacrificing deep coverage points (small γ).

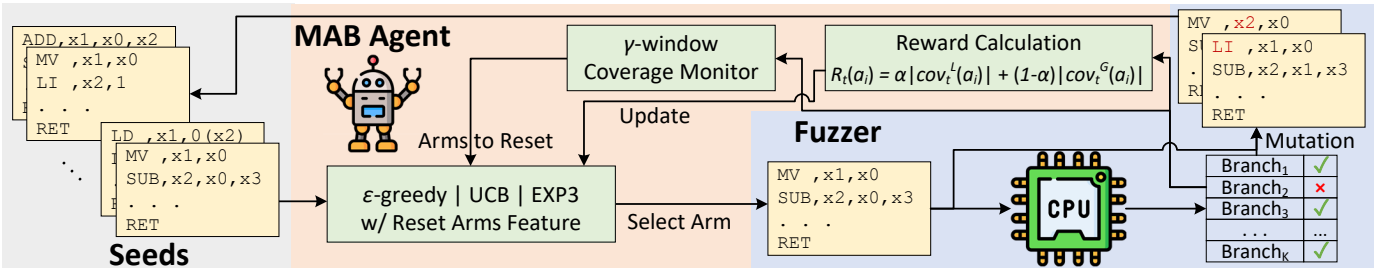

Fig. 2: Final MABFuzz framework.

Algorithm 1: Modified ε-greedy [12] and UCB [15]

Initialization: $Q(a) \leftarrow 0$, $N(a) \leftarrow 0\ \forall a \in \mathcal{A}; t \leftarrow 0$

1 **while** Fuzzing continues **do**
2 $t \leftarrow t + 1$
3 **if** algorithm == ε-greedy **then**
4 $A \leftarrow \begin{cases} \arg\max_a Q(a) & \text{with probability } 1-\varepsilon \\ \text{random } a \in \mathcal{A} & \text{with probability } \varepsilon \end{cases}$
5 **else if** algorithm == UCB **then**
6 $A \leftarrow \arg\max_a \left[Q(a) + \sqrt{\frac{2\times \ln t}{N(a)}}\right]$
7 $R_t(A) \leftarrow PullArm(A)$ // Simulate Tests
8 $N(A) \leftarrow N(A) + 1$
9 $Q(A) \leftarrow Q(A) + \frac{1}{N(A)}[R_t(A) - Q(A)]$
10 **if** A needs to be reset **then**
11 $N(A), Q(A) \leftarrow 0$
12

Algorithm 2: Modified EXP3 [16]

Parameters: $\eta \in (0, 1]$ // learning rate

Initialization: $W(a) \leftarrow 1\ \forall a \in \mathcal{A}; t \leftarrow 0$

1 **while** Fuzzing continues **do**
2 $t \leftarrow t + 1$
3 $P(a) \leftarrow (1-\eta)\frac{W(a)}{\sum_{j\in\mathcal{A}} W(j)} + \frac{\eta}{|\mathcal{A}|} \quad \forall a \in \mathcal{A}$
4 $A \sim P$ // Randomly sample A according to P
5 $R_t(A) \leftarrow PullArm(A)$ // Simulate Tests
6 $R_t(A) \leftarrow \frac{R_t(A)}{|C|}$ // Normalize reward
7 $x \leftarrow \frac{R_t(A)}{P(A)}$
8 $W(A) \leftarrow W(A) \times e^{(\eta x/|\mathcal{A}|)}$
9 **if** A needs to be reset **then**
10 $W(A) \leftarrow \frac{\sum_{j\in\mathcal{A}\setminus A} W(j)}{|\mathcal{A}|-1}$ // Set weight of A as average weight of other arms
11

resetting an arm. However, resetting an arm leads to another complication: the existing MAB algorithms are not designed to work when an arm is reset, i.e., replaced with another new arm. Next, we address this complication.

Accommodating Reset Arms in MAB Algorithms. The core idea to accommodate a reset arm is to treat the arm as a fresh arm and reset appropriate terms (e.g., the number of times the arm has been picked in the ε-greedy algorithm). Algorithms 1 and 2 detail our modified MAB algorithms for ε-greedy and UCB, and EXP3 algorithms, respectively. The parts highlighted in red are our modifications to accommodate reset arms.[2] For ε-greedy and UCB, if the selected arm, A, needs to be reset, we reset the counter for the number of times the arm has been pulled, $N(A)$, and its value, $Q(A)$ to 0 (line 11 in Algorithm 1). For EXP3, we set the weight of the arm, $W(A)$, as the average weight of the other arms (line 10 in Algorithm 2). Additionally, since EXP3 requires normalization, we divide the reward by the total number of coverage points, $|C|$ (line 6).

D. Final MABFuzz Formulation

Fig. 2 illustrates the final MABFuzz framework. Given a seed pool with each seed corresponding to an arm, the MAB agent selects an arm. This selection is done according to a predetermined modified MAB algorithm with the reset arms feature, such as the modified UCB algorithm (Algorithm 1) or the modified EXP3 algorithm (Algorithm 2). Next, the test corresponding to the selected arm is simulated on the target processor design to obtain the coverage information. After that, the original test is mutated to generate new tests, which are added to the pool, and the coverage information is (i) used to compute the reward for the agent's choice of arm and (ii) monitored to decide if the selected arm needs to be reset. The computed reward is used to update the MAB agent, which again selects an arm, and the cycle continues.

[2]Since the non-highlighted parts of the algorithms are unchanged, we refer an interested reader to [12], [15], [16] for their explanations.

IV. Experimental Evaluation

A. Experimental Setup

Implementation Setup. We implement MABFuzz with *TheHuzz* as the base fuzzer since *TheHuzz* is one of the state-of-the-art simulation-based processor fuzzers [7]. Similar to [7], we implement *TheHuzz* and MABFuzz in Python. We run all experiments on a machine with CentOS Linux distribution on Intel Xeon processors with 64 threads, 512GB RAM, and a 2.6GHz clock. We set the number of arms for MABFuzz as 10. As the ultimate objective is to maximize coverage, we set the parameter α (that controls the relative weights of new local and global coverage points) as 0.25, meaning that we assign $3\times$ importance to a point previously uncovered by any arm compared to a point previously uncovered by the chosen arm but covered by some other arm. Based on experiments, we observed that the value of 3 for the reset threshold, γ, yields good results, so we set it as 3. We set the learning rate, η, in Algorithm 2 as 0.1.

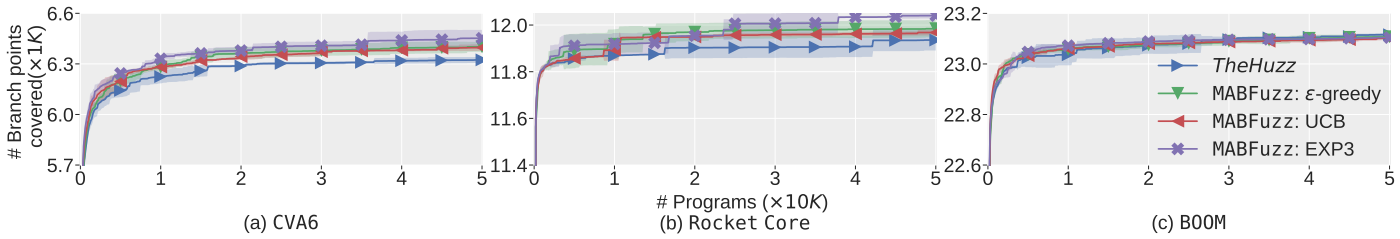

Fig. 3: Branch coverage achieved by MABFuzz: ε-greedy, MABFuzz: UCB, and MABFuzz: EXP3 compared to *TheHuzz* [7].

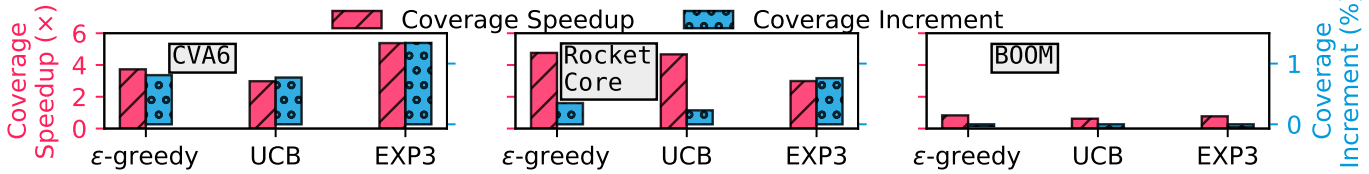

Fig. 4: Coverage speedup and increment compared to the base fuzzer, *TheHuzz* [7], for MABFuzz: ε-greedy, UCB, and EXP3.

TABLE I: Vulnerability detection speedup compared to *TheHuzz* [7]. Vulnerabilities V1-V6 are for CVA6 [24] and V7 is for Rocket Core [25].

Vulnerability	CWE #	*TheHuzz* [7]	ε-greedy	UCB	EXP3
		# Tests	Speedup	Speedup	Speedup
V1: FENCE.I instruction decoded incorrectly	440	600	1.56×	**13.04**×	4.5×
V2: Some *illegal* instructions can be executed	1242	1480	0.73×	2.26×	**8.33**×
V3: Exception type incorrectly propagated in instruction queue	1202	239	**59.75**×	7.71×	11.95×
V4: Undetected cache coherency violation	1202	1200	**2.43**×	1.48×	1.22×
V5: Exception not thrown when *invalid* addresses accessed	1252	2.5	0.35×	0.13×	**0.63**×
V6: Accessing unimplemented CSRs returns X-values	1281	141	2.33×	2.11×	**2.36**×
V7: EBREAK does not increase instruction count	1201	927	**308.89**×	185.34×	73.16×

Evaluation Setup. Following prior works, we evaluate MABFuzz on three open-source processors, CVA6 [24], Rocket Core [25], and BOOM [26], from RISC-V ISA, since most commercial processors are close-sourced [7]. These processors have various features such as out-of-order execution and a custom single instruction-multiple data floating point unit in CVA6, in-order execution in Rocket Core, and BOOM is a superscalar processor. Additionally, although these processors are open-sourced, they can fully boot the Linux operating system, demonstrating their practicality. We simulate the processors using Synopsys VCS and use Chipyard as the system-on-chip simulation environment. Similar to our base fuzzer, we ran experiments with $50,000$ tests for each benchmark for all fuzzers and repeated each experiment at least three times to reduce randomness in results [7] (standard deviation is shown through shading in Fig. 3). We use the branch coverage as the metric for comparison because it is highly correlated with vulnerability detection [27].

B. Vulnerability Detection

MABFuzz uses the differential testing technique that comes with the base fuzzer, *TheHuzz*, to detect vulnerabilities with the RISC-V ISA simulator, SPIKE [17] as the reference model. We evaluate the ability of MABFuzz to detect vulnerabilities by comparing its detection speed with that of *TheHuzz* on seven different vulnerabilities (Table I). MABFuzz is slower than *TheHuzz* in detecting vulnerability V5 because the vulnerability itself is easy to detect (*TheHuzz* detects it in only 25 tests). However, on average, MABFuzz detects vulnerabilities $53.72\times$, $30.29\times$, and $14.59\times$ faster than *TheHuzz* with ε-greedy, UCB, and EXP3 MAB algorithms, respectively. MABFuzz achieved this highest speedup when detecting vulnerability, V7. V7 is a bug in the decode module, which consists of many `if-else` conditions requiring more exploration than exploitation to detect it. Hence, it can be seen that *TheHuzz*, which emphasizes exploitation of the same seeds, takes a long time to detect this bug, whereas MABFuzz detects it up to two orders of magnitude faster. Also, the speedup achieved by the MAB algorithms varies with the vulnerabilities, with no single algorithm achieving the highest detection speed for all the vulnerabilities. This shows the importance of the compatibility of MABFuzz to support different MAB algorithms, which can be chosen for different use cases.

C. Coverage Analysis

Fig. 3 plots the number of branch coverage points achieved by the algorithms used in MABFuzz on all benchmarks and compares it with *TheHuzz*. As shown, MABFuzz outperforms *TheHuzz*. Furthermore, Fig. 4 shows the coverage speedup and percentage increment in coverage achieved by MABFuzz compared to *TheHuzz* when run for 50,000 tests. MABFuzz is at least $2.98\times$ faster than *TheHuzz* on CVA6 and Rocket Core. This is because MABFuzz minimizes spending time on low-performing seeds by dynamically choosing seeds based on their performance and resetting them when they stop covering new points. This allows MABFuzz to fuzz with more meaningful tests, resulting in at least 0.23% more covered points than *TheHuzz*. Moreover, MABFuzz's ability to explore design space is more evident as the difficulty of covering the points increases. For instance, on CVA6, *TheHuzz* achieves the lowest coverage percentage, and MABFuzz achieves its highest speedup, $5.38\times$. On the other hand, *TheHuzz* achieves $> 95\%$ coverage on the BOOM processor, leaving less room for improvement for MABFuzz. This results in MABFuzz not achieving higher coverage than *TheHuzz*. On average over all processors, MABFuzz achieves $3.11\times$, $2.76\times$, and $3.05\times$ speedup with ε-greedy, UCB, and EXP3 MAB algorithms, respectively.

Overall, all the algorithms of MABFuzz detected vulnerabilities faster than *TheHuzz* and achieved more and faster coverage thanks to the efficient balance of exploration and exploitation of the design space by MABFuzz.

V. Discussion

Other Avenues for MAB Algorithms in Hardware Fuzzers. In this work, we focused on using MAB algorithms to select seeds to fuzz. Future research can investigate the application of MAB algorithms to other avenues such as mutation operator selection or parameters such as the number of instructions in a test since different numbers of instructions have different impacts on coverage at different times.

Theoretical Analysis. MAB algorithms have rich theoretical analyses, which is absent in our empirical evaluation of the modified MAB algorithms. In the future, we plan to address this by conducting a theoretical analysis of the modified MAB algorithms to understand them thoroughly and possibly devise better MAB algorithms for hardware fuzzing.

VI. Conclusion

Prior works on detecting vulnerabilities in processors using fuzzing have shown reasonable performance, but they rely heavily on static strategies in their algorithms, limiting their efficacy. To address this limitation, we develop a new technique, MABFuzz, to make decisions dynamically and balance the exploration of the new test programs and exploitation of the well-performing test programs using MAB algorithms. However, to develop MABFuzz, we face challenges related to the compatibility of MAB algorithms and fuzzing. We overcome these challenges by modifying both, the fuzzing flow and MAB algorithms. As a result, the final MABFuzz formulation is effective, efficient, and agnostic to any hardware fuzzer. Experimental results demonstrate that MABFuzz achieves an average speedup of $53.72\times$ in detecting vulnerabilities and an average speedup of $3.11\times$ in covering the design space. Although this work focuses on applying MAB algorithms to selecting test programs to fuzz, several other avenues of hardware fuzzers have the potential to improve (see Sec. V) when optimized with MAB algorithms. These avenues should be explored in the future to improve the verification and security of processors in the face of increasing complexity.

VII. Acknowledgement

Our research work was partially funded by the US Office of Naval Research (ONR Award #N00014-18-1-2058), Intel's Scalable Assurance Program, the European Research Council (ERC project HYDRANOS 101055025), Deutsche Forschungsgemeinschaft (DFG) SFB 1119 CROSSING/236615297, and European Union's Horizon Europe research and innovation program (No. 101070537). 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 the US Government, Intel, the European Union, or the European Research Council.

References

[1] G. Dessouky *et al.*, "HardFails: Insights into Software-Exploitable Hardware Bugs," *USENIX Security Symposium*, pp. 213–230, 2019.

[2] H. de Moraes Holschuh, "Intel Skylake/Kaby Lake processors: broken hyper-threading," https://lists.debian.org/debian-devel/2017/06/msg00308.html, 2017, accessed: 2023-08-14.

[3] T. Ormandy, "Zenbleed," 2023, Last accessed on 09/18/2023. [Online]. Available: https://lock.cmpxchg8b.com/zenbleed.html

[4] S. K. Muduli *et al.*, "HyperFuzzing for SoC Security Validation," *ACM/IEEE International Conference on Computer-Aided Design*, pp. 1–9, 2020.

[5] J. Hur *et al.*, "DIFUZZRTL: Differential Fuzz Testing to Find CPU Bugs," *IEEE Symposium on Security and Privacy*, pp. 1286–1303, 2021.

[6] T. Trippel *et al.*, "Fuzzing Hardware Like Software," *USENIX Security Symposium*, 2022.

[7] R. Kande *et al.*, "TheHuzz: Instruction Fuzzing of Processors Using Golden-Reference Models for Finding Software-Exploitable Vulnerabilities," *USENIX Security Symposium*, pp. 3219–3236, 2022.

[8] C. Chen *et al.*, "HyPFuzz: Formal-Assisted Processor Fuzzing," *arXiv preprint arXiv:2304.02485*, 2023.

[9] H. Ragab *et al.*, "BugsBunny: Hopping to RTL Targets with a Directed Hardware-Design Fuzzer," *SILM*, Jun. 2022. [Online]. Available: https://download.vusec.net/papers/bugsbunny_silm22.pdf

[10] C. Chen *et al.*, "PSOFuzz: Fuzzing Processors with Particle Swarm Optimization," *arXiv preprint arXiv:2307.14480*, 2023.

[11] IntelLabs, "Pre-Silicon Hardware Fuzzing Toolkit," https://github.com/IntelLabs/PreSiFuzz, 2023, last accessed on 05/21/2023.

[12] R. S. Sutton *et al.*, *Reinforcement learning: An introduction*. MIT press, 2018.

[13] Z. U. A. Tariq *et al.*, "Network intrusion detection for smart infrastructure using multi-armed bandit based reinforcement learning in adversarial environment," in *2022 International Conference on Cyber Warfare and Security (ICCWS)*. IEEE, 2022, pp. 75–82.

[14] A. Ferdowsi *et al.*, "Cyber-physical security and safety of autonomous connected vehicles: Optimal control meets multi-armed bandit learning," *IEEE Transactions on Communications*, vol. 67, pp. 7228–7244, 2019.

[15] P. Auer *et al.*, "Finite-time analysis of the multiarmed bandit problem," *Machine learning*, vol. 47, pp. 235–256, 2002.

[16] ——, "The nonstochastic multiarmed bandit problem," *SIAM journal on computing*, vol. 32, no. 1, pp. 48–77, 2002.

[17] RISC-V, "SPIKE Source Code," https://github.com/riscv-software-src/riscv-isa-sim, 2023, last accessed on 09/18/2023.

[18] ——, "RISC-V Webpage," https://riscv.org/, 2021, Last accessed on 05/21/2023.

[19] V. Gohil *et al.*, "DETERRENT: Detecting Trojans using Reinforcement Learning," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2022, pp. 697–702.

[20] H. Chen *et al.*, "AdaTest: Reinforcement learning and adaptive sampling for on-chip hardware Trojan detection," *ACM TECS*, vol. 22, no. 2, pp. 1–23, 2023.

[21] V. Gohil *et al.*, "ATTRITION: Attacking Static Hardware Trojan Detection Techniques Using Reinforcement Learning," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 1275–1289.

[22] H. Guo *et al.*, "ExploreFault: Identifying Exploitable Fault Models in Block Ciphers with Reinforcement Learning," in *Proceedings of ACM/IEEE Design Automation Conference*, 2023.

[23] Y. Koike *et al.*, "Slopt: Bandit optimization framework for mutation-based fuzzing," in *Proceedings of the 38th Annual Computer Security Applications Conference*, 2022, pp. 519–533.

[24] F. Zaruba *et al.*, "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 Systems*, vol. 27, no. 11, pp. 2629–2640, 2019.

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

[26] J. Zhao *et al.*, "SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine," *Fourth Workshop on Computer Architecture Research with RISC-V*, May 2020.

[27] A. Mockus *et al.*, "Test Coverage and Post-Verification Defects: A Multiple Case Study," pp. 291–301, 2009.