FuSS: Coverage-Directed Hardware Fuzzing with Selective Symbolic Execution

ARUNA JAYASENA, University of Florida, USA
SAI SUPRABHANU NALLAPANENI, University of Florida, USA
PRABHAT MISHRA, University of Florida, USA

Fuzzing is a promising validation method to detect design flaws as well as security vulnerabilities in a wide variety of electronic systems. Traditional fuzzing methods can outperform validation using random test vectors but they can lead to a coverage plateau due to the increasing number of hard-to-activate areas in complex hardware designs. While property checking aids in the exploration of hard-to-active scenarios in recent fuzzing solutions, they face several practical limitations, including inefficient utilization of fuzzing efforts and state space explosion for complex scenarios. This paper introduces a novel approach to hardware fuzzing that synergistically integrates coverage-guided fuzzing with selective symbolic execution. Specifically, when hardware fuzzing reaches a coverage plateau, our framework utilizes selective symbolic execution to explore hard-to-activate areas. Unlike property-checking based fuzzing that tries to generate an input sequence from the start state, selective symbolic execution utilizes the existing fuzzing trajectory to produce a minimal input sequence to efficiently explore hard-to-activate areas. Extensive evaluation using four RISC-V based designs demonstrates that our framework can significantly improve both branch and toggle coverage compared to state-of-the-art fuzzing techniques. Our theoretical analysis and empirical results prove that our framework will always reach a coverage goal faster than existing fuzzing methods.

CCS Concepts: • **Hardware → Semi-formal verification**; *Theorem proving and SAT solving*.

Additional Key Words and Phrases: Hardware Fuzzing, Processor Validation, Symbolic Execution, Semi-Formal Verification, Directed Test Generation

ACM Reference Format:
Aruna Jayasena, Sai Suprabhanu Nallapaneni, and Prabhat Mishra. 2025. FuSS: Coverage-Directed Hardware Fuzzing with Selective Symbolic Execution. *ACM Trans. Embedd. Comput. Syst.* 00, 0, Article 000 (2025), 24 pages. https://doi.org/000000.0000000

1 Introduction

Due to the increasing complexity of modern System-on-Chip (SoC) designs, ensuring functional correctness [20] or security guarantees [17, 19] through traditional verification methods presents a significant challenge. Conventional verification approaches, such as random and constrained-random verification, struggle to achieve adequate coverage of the vast state space in modern SoCs [18]. For instance, even millions of random or constrained-random test vectors (input stimuli) fail to activate a majority of complex execution paths, leaving many potential corner cases untested [6]. The growing integration of heterogeneous components, including CPUs, GPUs, accelerators, memory controllers, and security modules, further exacerbates this challenge by increasing the number of potential interactions and dependencies within the system.

Authors' Contact Information: Aruna Jayasena, University of Florida, Gainesville, Florida, USA; Sai Suprabhanu Nallapaneni, University of Florida, Gainesville, Florida, USA; Prabhat Mishra, University of Florida, Gainesville, Florida, USA.

ACM 1558-3465/2025/0-ART000
https://doi.org/000000.0000000

Fuzzing has emerged as a promising alternative for hardware verification, leveraging prior test cases (corpus) and randomized input mutations to systematically uncover hardware bugs [3, 16, 41] and security vulnerabilities [2, 4, 11, 14, 15, 40, 50]. Unlike traditional verification methods, which rely on specifications, directed testbenches, or constrained-random testing, fuzzing operates without explicit modeling of expected behaviors. Instead, it dynamically explores unexpected and rare execution paths by continuously mutating inputs and observing the resulting hardware states. This approach is particularly effective in discovering vulnerabilities that may not be easily identified through conventional verification techniques, such as timing violations, privilege escalations, and unexpected microarchitectural interactions.

1.1 Coverage Plateau

However, traditional fuzzing techniques often encounter a coverage plateau when they struggle to explore new execution states or paths due to the inherent randomness of input mutations. This is particularly problematic in hardware fuzzing, where stateful behaviors, deeply nested conditions, and long execution traces require more intelligent exploration strategies. Although recent fuzzing solutions utilize assistance of formal methods with property checking to explore hard-to-activate regions and outperform traditional fuzzing methods [5, 9], they also encounter a coverage plateau due to inherent limitations: (i) property checking relies on pre-defined properties that may not cover diverse corner case scenarios, and (ii) property checking can lead to state space explosion for large designs or complex scenarios since it tries to construct an input sequence from the start state of the design. This ultimately causes verification tools to take longer to meet high-assurance requirements or eventually fail to uncover hidden vulnerabilities.

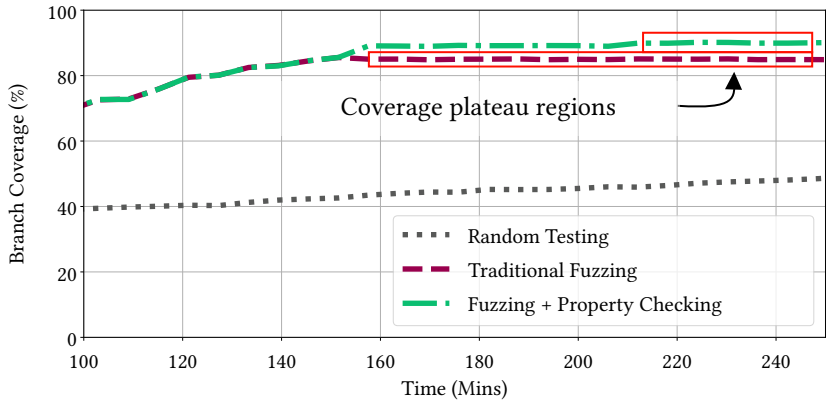


Fig. 1. Illustration of branch coverage on RISC-V CVA6 SoC across random testing, traditional fuzzing, and fuzzing with property checking. Traditional fuzzing and fuzzing with property checking reach the coverage plateau after approximately 150 and 230 minutes, respectively.

Illustrative Example 1: Figure 1 illustrates an instance of a coverage plateau during the RISC-V-based *CVA6* SoC validation, where the branch coverage of the design does not improve beyond 60% for random testing. While traditional fuzzing outperforms random testing, it also encounters a coverage plateau of around 85% since it is unable to active branches with rare conditions. Although recent fuzzing solutions with property checking outperforms traditional fuzzing methods, they also reach a coverage plateau near 88% due to the state space explosion. The coverage plateau problem becomes further aggravated if we consider other coverage metrics. For example, as demonstrated in Figure 11, state-of-the-art fuzzing solutions can only reach up to 70% in toggle coverage. A coverage

plateau occurs as the fuzzer repeatedly generates inputs that fail to reach unexplored areas of the state space, limiting the discovery of critical bugs and corner cases. ■

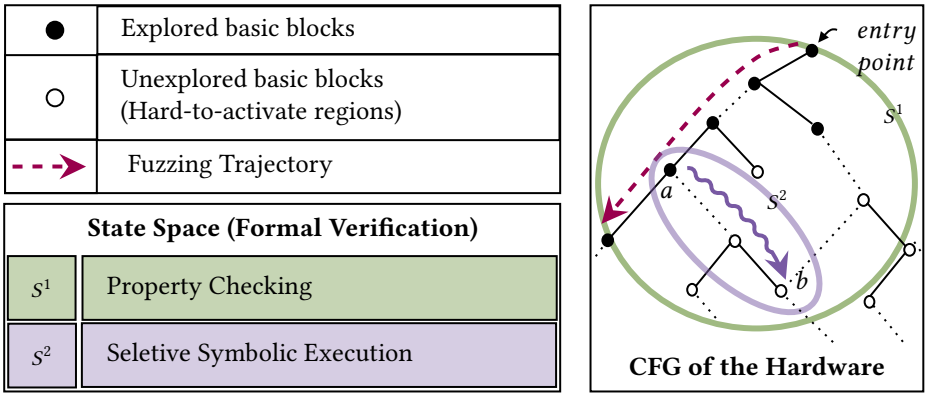


Fig. 2. Comparison between the state space faced by the property checking (S^1) and the selective symbolic execution (S^2) for activating the unexplored basic block b of the hardware design. Here, S^1 includes almost the entire state space of the design from the entry point to the target destination point b. *FuSS* framework only has to deal with a small state space (S^2) starting from a to the target destination point b, due to effective utilization of fuzzer output for symbolic execution.

1.2 Research Contributions

This paper introduces a coverage-guided fuzzing framework that combines the advantages of **Fu**zzing with **S**elective **S**ymbolic execution (FuSS). Major contribution of our framework is improved fuzzing effectiveness through enhanced coverage, particularly in hard-to-reach areas of processor designs. This targeted approach overcomes the coverage plateau problem by bypassing heavily explored regions and directing the fuzzer towards hard-to-reach states. Unlike property-checking based fuzzing that can lead to state space explosion while generating an input sequence from the start state, selective symbolic execution utilizes the existing fuzzing trajectory to produce a minimal input sequence to efficiently explore hard-to-activate areas. Figure 2 shows an illustrative example to highlight the fact that fuzzing with property checking needs to deal with a huge state space S^1 (green oval) while our framework needs to explore a significantly smaller state space S^2 (purple oval) to activate b due to effective utilization of current fuzzing trajectory as well as selective symbolic execution. Specifically, this paper makes the following contributions:

- We introduce an automated coverage plateau detection to invoke the symbolic execution only when it is needed.
- We propose an efficient context mapping that allows us to align the generated program from the fuzzer with the control flow of the hardware implementation.
- We prune the generated program by the fuzzer to prepare the starting state for the symbolic execution.
- We instrument a methodology to utilize the fuzzing trajectory for selective symbolic execution, efficiently activating the hard-to-reach targets while avoiding state space explosion.
- We perform a theoretical analysis of the proposed technique that is backed up by empirical results.
- Extensive evaluation on several RISC-V-based SoC designs demonstrates the effectiveness of improving the design coverage and avoiding the coverage plateau.

Our proposed approach can be directly used on top of any existing fuzzing techniques to improve the coverage. Our framework provides the required interfaces to utilize the fuzzing trajectory to generate input sequences using selective symbolic execution to quickly explore unreachable regions.

1.3 Paper Organization

The remainder of this paper is organized as follows. First, we provide relevant background and survey related efforts in Section 2. Next, we describe our proposed FuSS framework in Section 3. In Section 4, we perform a theoretical analysis comparing the effectiveness of FuSS with traditional fuzzing and fuzzing with property checking. In order to evaluate the effectiveness of the proposed framework, we perform experiments in Section 5. Finally, Section 6 concludes the paper.

```
case (state)
    IDLE: begin
        if (start_write)
            next_state = ADDR;
    end
    ADDR: begin
        awvalid = 1;
        if (awready)
            next_state = WRITE;
    end
    WRITE: begin
        wvalid = 1;
        if (wready)
            next_state = RESPONSE;
    end
    RESPONSE: begin
        bready = 1;
        if (bvalid)
            next_state = DONE;
    end
    DONE: begin
        next_state = IDLE;
    end
endcase
```

(a) Write logic of 'axi4_lite' master interface.

```
case (state)
    IDLE: begin
        if (awvalid)
            next_state = ADDR_READY;
    end
    ADDR_READY: begin
        awready = 1;
        if (awvalid)
            next_state = WRITE_READY;
    end
    WRITE_READY: begin
        wready = 1;
        if (wvalid)
            next_state = RESPOND;
    end
    RESPOND: begin
        bvalid = 1;
        if (bready)
            next_state = DONE;
    end
    DONE: begin
        next_state = IDLE;
    end
endcase
```

(b) Read logic of 'axi4_lite' slave interface.

Fig. 3. Illustrative example of hard-to-activate region with 'axi4_lite' interface in Verilog hardware description language. AXI-4 is a commonly used bus implementation in SoC designs. In order to correctly perform operations between the AXI-4 master device and slave device, two finite state machines should be synchronized between their state transitions.

2 Background and Related Work

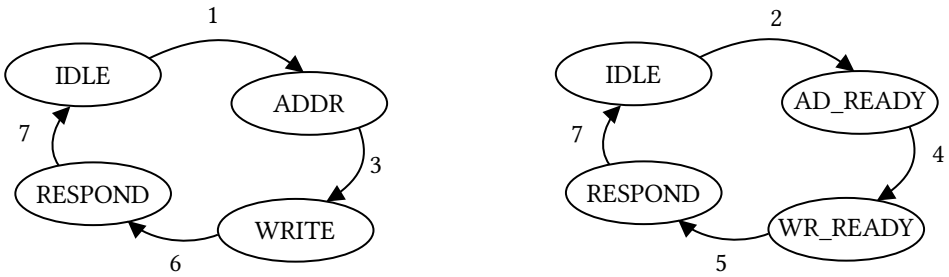
In this section, we review the background and related work in hardware fuzzing. We first examine hard-to-activate areas in SoC implementations, which contribute to coverage plateaus by limiting

the exploration of certain design states. Next, we summarize existing hardware fuzzing approaches, highlighting their methodologies, strengths, and limitations in addressing these challenges.

2.1 Hard-to-Activate Areas

Hardware designs are implemented as finite state machines (FSMs) that enable parallel execution, allowing multiple operations to occur simultaneously. Unlike sequential software execution, hardware designs rely on concurrency to optimize performance, minimize latency, and maximize resource utilization. To ensure proper coordination between different FSMs, various synchronization mechanisms are used, such as control signals, handshaking protocols, and status flags. These mechanisms require specific sequences of operations that enable different regions of the design to activate in a controlled manner. While synchronization ensures functional correctness, it also introduces complex dependencies between signals, states, and timing relationships. The complex interactions between multiple FSMs coupled with rare conditional dependencies can create hard-to-activate regions in the implementation.

Illustrative Example 2: Figure 3 shows a part of a Verilog implementation of an AXI-4 Lite interface that includes two parallel finite state machines (FSMs) responsible for handling a data write operation. Figure 3a shows the implementation for the master device and Figure 3b shows the implementation for the slave device. In a typical SoC implementation, both of these devices are used to communicate between components. These FSMs coordinate through a series of handshakes to ensure a correct and successful write transaction.



(a) FSM and synchronization sequence of transitions of ‘axi4_lite’ master interface.

(b) FSM and synchronization sequence of transitions for write logic of ‘axi4_lite’ slave interface.

Fig. 4. Illustrative example of the Finite State Machine (FSM) for hard-to-activate region of ‘axi4_lite’ interface presented in Figure 3. The transitions between two Finite state machines should be synchronized and follow the transition sequence from 1,2,...,7 in order to perform the transaction properly.

Figure 4 shows the simplified FSMs implemented by Figure 3. In order to perform a successful write operation by the AXI4 master device to the AXI4 slave device, correct sequence of actions needs to be performed. This process begins with the master FSM in the IDLE state, waiting for a write request. Upon initiation, it transitions to the ADDR state (transition 1 in Figure 4a), asserting `awvalid` to indicate that a valid address is being sent. Simultaneously, the slave FSM, also starting in the IDLE state, moves to the `ADDR_READY` state upon detecting `awvalid` and asserts `awready` to acknowledge receipt of the address (transition 2 in Figure 4b). Once both `awvalid` and `awready` are asserted, the master transitions to the `WRITE` state (transition 3 in Figure 4a), where it asserts `wvalid` and sends the data. At this stage, the slave FSM transitions to the `WRITE_READY` state, asserting `wready` to confirm its readiness to receive data (transition 4 in Figure 4b). The transaction proceeds

when both wvalid and wready are high, allowing the slave to capture the data and move to the RESPOND state (transition 5 in Figure 4b). The slave then asserts bvalid to indicate that the write operation has completed and a response is available. The master, upon detecting bvalid, transitions to the RESPONSE state and asserts bready to acknowledge receipt of the response (transition 6 in Figure 4a). Once the handshake is complete, both FSMs transition to the DONE state before returning to the IDLE state, signaling the completion of the write operation (transition 7 in both Figure 4a and Figure 4b). ■

As demonstrated by the above example, the correct ordering of these FSM transitions is crucial for ensuring compliance with the implementation protocol while maintaining efficiency in hardware design. However, testing such implementations with fuzzing is particularly complex due to the challenge of identifying the correct sequence of actions needed to drive the design through all possible states. The dependency between multiple handshakes and state transitions makes it difficult to generate meaningful test cases that achieve high coverage, as incorrect sequences may lead to deadlocks or incomplete transactions, ultimately reaching a coverage plateau.

2.2 Related Work

Fuzzing is widely used for both hardware and software validation to complement traditional validation techniques, such as simulation with random tests and formal methods. There are recent efforts to utilize software fuzzing [12, 13, 21, 23, 25, 26, 29, 32, 35, 36, 44, 48, 53, 54, 56–58] to verify hardware systems by translating hardware designs into software models [50]. With various application specific modifications, Fuzzing has been used in diverse domains, including processor validation [2, 11], memory vulnerability detection [40], SoC security verification [14, 15] and network protocol verification [1, 28, 30, 33, 34, 37, 38, 42, 52, 55]. For example, SoCFuzzer [15] employed a cost function-driven feedback mechanism to identify vulnerabilities autonomously in complex architectures. Authors have dynamically evaluated test case effectiveness based on factors such as code coverage and state transitions, which prioritizes inputs that are more likely to uncover subtle security flaws. Similarly, TaintFuzzerd [14] integrated taint inference with fuzz testing to improve SoC robustness. For this purpose, authors have analyzed the data flow within the SoC. Then they proposed a technique that can identify how the input data propagates through the system, enabling the detection of vulnerabilities that might be missed by traditional testing methods. Gohil et al. utilized multi-armed bandit algorithms to adaptively improve detection efficiency in processors [11]. Similarly, WhisperFuzz used microarchitectural state transitions to identify and localize timing-related issues in processor designs [2]. This technique automatically extracts microarchitectural state transitions from a processor's register-transfer level (RTL) design and instruments the design to monitor these transitions as coverage metrics, enabling precise detection and localization of timing vulnerabilities.

Recent approaches improved hardware fuzzing performance in various directions. DifuzzRTL [16] employed differential fuzz testing to uncover vulnerabilities in CPU designs. Authors introduced a register-coverage metric tailored for RTL designs, effectively guiding input generation to comprehensively explore CPU states and detect vulnerabilities. Another promising approach, TheHuzz [22] focused on generating assembly-level instructions to detect security-critical bugs. This technique enhances hardware fuzzing by generating assembly-level instructions aimed at increasing specific coverage metrics, thereby effectively uncovering hardware bugs that are exploitable from software. HD-Fuzz [24] integrated hybrid memory-mapped input-output (MMIO) modeling to address hardware dependencies and perform firmware fuzzing for improved bug detection. Authors employed a three-phase hybrid MMIO modeling approach with initial modeling, mutation modeling, and

additional searching to efficiently generate access models optimized for identifying firmware bugs with the help of virtual prototypes. ChatFuzz [41] leveraged large language models to efficiently generate complex instruction sequences for processor validation. PSOFuzz [4] utilized particle swarm optimization for dynamic mutation scheduling. ProcessorFuzz [3] guided fuzzing using control and status registers. MABFuzz [11] applied multi-armed bandit algorithms for adaptive fuzzing. Cascade [46] generated intricate RISC-V programs to expose processor vulnerabilities, while SurgeFuzz [49] targeted corner cases in RISC-V CPUs using surge-inducing instruction sequences.

State-of-the-art fuzzing methods utilized formal assistance from property checking to explore hard-to-activate areas. For example, FormalFuzzer [9] employed property checking to reduce the input space for fuzzing, leveraging counterexamples and feedback-based cost functions to guide mutations effectively. Specifically, authors have implemented an emulation-based hybrid technique by incorporating template-based assertion generation and fuzzing testing together. Authors have further narrowed down the search space for the fuzzer with the help of generated properties. HyPFuzz [5] integrated formal verification tools to strengthen fuzzing, utilizing property checking to ensure correctness while improving coverage and detection speed. Authors have utilized several strategies to select the coverage points for the property checking tool perform test generation such that these tests can be used by the fuzzer.

While the property checking-based fuzzing approaches outperform traditional fuzzing, they also face the coverage plateau problem due to their inherent limitations in exploring hard-to-activate regions, such as (i) ignoring fuzzing results during property checking, (ii) state space explosion during property checking of complex corner cases, and (iii) property checking is limited by its reliance on pre-defined properties. In contrast, our proposed framework effectively utilizes the current fuzzing trajectory to construct a minimal input sequence to activate rare scenarios dynamically using selective symbolic execution.

3 Fuzzing with Selective Symbolic Execution (FuSS)

Figure 5 provides an overview of our hardware fuzzing framework with selective symbolic execution (FuSS). Our framework consists of five major components: (i) design instrumentation, (ii) performing the fuzzing until reaching a coverage plateau, (iii) hardware-program context mapping, (iv) source and destination extraction, and (v) symbolic execution to break the coverage plateau and repeat the fuzzing process. The next five describes these five steps in detail. Finally, we provide an illustrative example to highlight the key concepts of our proposed framework.

3.1 Design Instrumentation

Our objective with fuzzing is to verify processor-based hardware implementations and achieve the highest possible design coverage. The first step is to instrument the hardware implementation so that both the fuzzing framework and the symbolic execution engine can understand the hardware description. Specifically, we implement a testbench that instantiates the hardware description and controls the clock and reset lines. The input instruction sequence (program) is provided as an external argument to the testbench in the form of a byte (8-bit) array. We define internal segments within this array separating each of the internal instructions, as illustrated by Figure 6. This structure allows the fuzzer to mutate individual components separately while making it easier to track individual instructions to be used for facilitating symbolic execution.

Once the instrumentation is completed, we convert the entire hardware description with the testbench into a software model. This allows us to utilize verification tools (both fuzzing and symbolic execution) designed for software to be used on hardware implementations with minimum modifications.

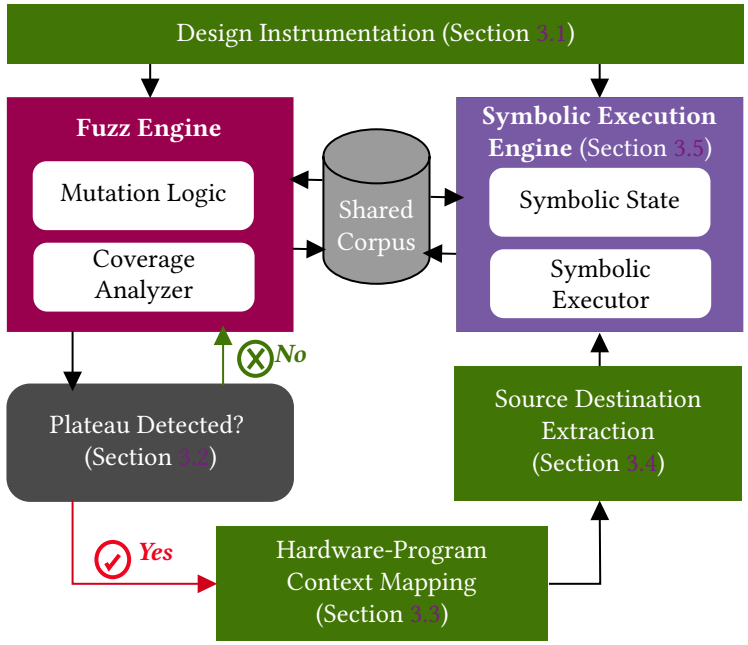


Fig. 5. Overview of the FuSS framework that consists of five major steps: (i) design instrumentation, (ii) performing the fuzzing until reaching a coverage plateau, (iii) hardware-program context mapping (iv) source and destination extraction, and (v) symbolic execution. Both fuzzing and symbolic execution engines share the same test vector corpus.

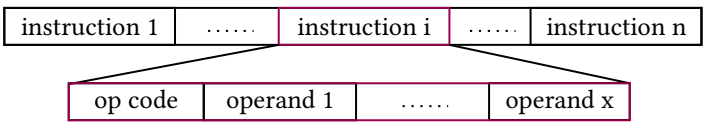


Fig. 6. Illustration of program structure with internal instruction segments and the instruction-related components.

3.2 Coverage Plateau Detection

Once the instrumentation is done and the hardware implementation is converted into a software model, we provide the instrumented model to the fuzzer to start the fuzzing. However, instead of starting from random instructions, we start the fuzzing with pre-compiled benchmark (program) and let the fuzzer improve the coverage until it reaches the coverage plateau. This is the point where we need to invoke the symbolic engine.

Detecting the coverage plateau is not trivial since fuzzing coverage might be stuck in a coverage plateau but the coverage may still fluctuate around it. In order to accurately identify the coverage plateau, we define a threshold function $P(t)$. Let $C(t)$ represent the cumulative code coverage achieved after t fuzzing iterations. We use a sliding window with a window size of W, which represents the number of iterations to consider for evaluating the coverage trend. The choice of W depends on the specific fuzzing context and desired sensitivity. Coverage increment is computed by $\Delta C(t, W) = C(t) - C(t - W)$. We compute the average increment, as shown in Equation 1.

$$\overline{\Delta C(t,W)} = \frac{C(t) - C(t-W)}{W} \tag{1}$$

We define a coverage increment threshold θ that represents the minimal acceptable coverage increase per iteration. This threshold determines when the coverage growth is considered negligible. We implement a plateau detection function, $P(t)$, as shown in Equation 2.

$$P(t) = \begin{cases} 1 & \text{if } \overline{\Delta C(t,W)} < \theta, \\ 0 & \text{otherwise.} \end{cases} \tag{2}$$

When $P(t) = 1$, it indicates that a coverage plateau is detected at the particular iteration, and $P(t) = 0$ indicates that coverage is still improving adequately. Once we detect the plateau, we need to prepare the generated programs for the symbolic execution. Let the i'th assembly instruction generated by the fuzzer as a_i^f. Therefore, the instruction sequence up to the coverage plateau point can be represented by the set A_f as shown in Equation 3.

$$A_f = \{a_1^f, \ldots, a_n^f\} \tag{3}$$

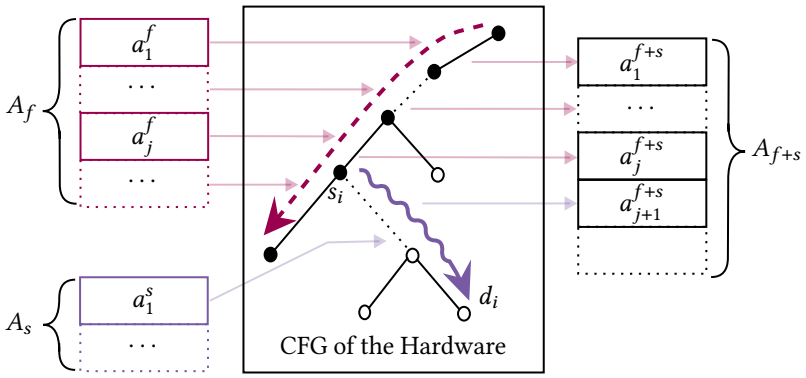


Fig. 7. Illustration of trimming the input sequences generated by the fuzzer (illustrated by ■) to stop at node s_i and integration of the test vector generated from selective symbolic execution (illustrated by ■) to explore the node d_i. Here ⇢ corresponds to the path explored by fuzzer and ⇝ corresponds to path explored by selective symbolic execution.

3.3 Hardware-Program Context Mapping

Once we identify the coverage plateau, we need to invoke the symbolic execution engine. Existing fuzzers with property checking construct properties to activate the unexplored areas in the design. However, they do not effectively utilize the effort taken by the fuzzer to explore the design so far. In contrast, we map the context between the hardware and the generated program. We start the mapping process by constructing the control-flow graph (CFG) of the hardware implementation. The CFG of the hardware implementation is a graph where each basic block is represented as a node (vertex), and edges denote possible control flow transfers between blocks due to branches (such as multiplexers, and finite state machines). A basic block is a straight-line sequence of hardware logic code with no branches except at the entry and exit points. In order to map the generated instructions with each of the vertex available in the CFG, we create function hooks in each basic

block that will send an interrupt when a particular basic block is activated. This essentially helps to annotate the basic blocks in the hardware that were activated with a particular instruction of the generated program code from the fuzzer.

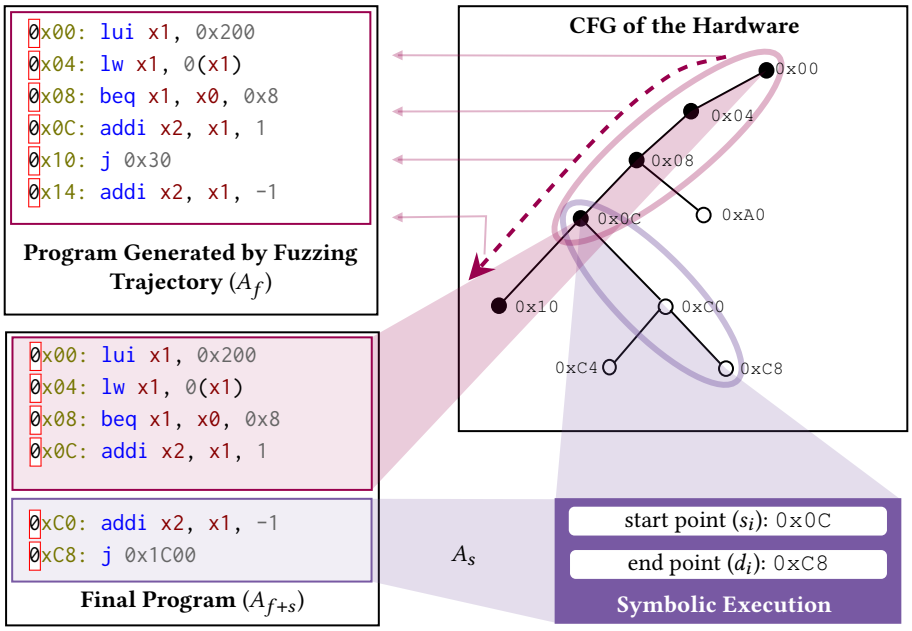


Fig. 8. An example scenario for Figure 7 showing the integration of fuzzing with symbolic execution to reach 0xC8, which is blocked by a rare condition at 0x0C. The upper box (■) in the final program represents the sequence of instructions to reach the entry point of the unexplored region (0x0C). The lower box (■) in the final program represents the instruction sequence to explore from 0x0C to 0xC8. FuSS takes advantage of both current fuzzing trajectory and selective symbolic execution.

3.4 Source and Destination Extraction

After mapping the hardware CFG to program inputs generated by the fuzzer, we use the CFG to identify the start and end points for symbolic execution. Next, we perform concrete simulation using the programs generated by the fuzzer, and select the explored and unexplored points in the CFG as the sources and destinations for the selective symbolic execution. Specifically, we select destination nodes from unexplored CFG nodes such that they are reachable within N edges from the explored nodes, where N is a parameter. This strategy ensures that symbolic execution targets new paths efficiently, avoiding redundant exploration and building upon the fuzzer's existing coverage. At the end of this step, we have a list of source (s_i) and destination (d_i) pairs that we have to perform symbolic execution as illustrated by Equation 4.

$$SD = \{(s_1, d_1), \ldots, (s_n, d_n)\} \tag{4}$$

3.5 State Preparation and Symbolic Execution Manager

Once the source and destinations are identified, the next step is to use the symbolic execution to break the coverage plateau. For this purpose, we implement a symbolic execution manager. Since our symbolic execution is not started from the hardware design entry point, the symbolic execution manager requires three inputs, execution start point, execution endpoint, and the start state.

In order to compute the inputs for the symbolic execution manager, we select each pair from the set SD from Equation 4, and then use concrete simulation with the generated assembly instructions (A_f) from the fuzzer. Figure 7 illustrates the process of reaching the s_i on the CFG using the A_f. Once we reach a particular source (s_i), we use the internal state of the hardware design (snapshot of the hardware) at that point of the simulation as the start state while annotating the current execution instruction (a_j^f). Next, we select s_i as the start point while selecting the corresponding destination d_i as the end state. After properly initializing, we can start the symbolic execution manager. Upon successful symbolic execution, the symbolic execution manager will provide a test vector $A_s = \{a_1^s, \ldots, a_k^s\}$, which can be used to construct the final program (A_{f+s}), by concatenating with the initial part of the instructions up to a_j^f from fuzzer-generated program, as illustrated by Equation 5.

$$A_{f+s} = \{a_1^f, \ldots, a_j^f || a_1^s, \ldots, a_k^s\} \tag{5}$$

This program is added to the fuzzing corpus and the process is repeated for all source-destination pairs in the set SD.

Illustrative Example 3: Figure 8 illustrates a concrete example that demonstrates how the generated assembly instructions are mapped to basic blocks, and how the final assembly program is constructed. Each node in the hardware CFG has its own address. In this example, the address range is `0x00-0xC8`. In this case, the program generated by the fuzzer explores the path `0x00`→`0x04`→`0x08`→`0x0C`→`0x10`, while there are four unexplored nodes (`0xA0`, `0xC0`, `0xC4`, and `0xC8`). Here we have three source and destination pairs nodes such that there are at most $N = 2$ edges between them: SD = {(`0x08`,`0xA0`), (`0x0C`,`0xC8`), (`0x0C`,`0xC4`)}. The instance in Figure 8 corresponds to the second pair of source and destination values, where the start state for the symbolic execution is initialized by concrete simulation of the instructions corresponding to the nodes `0x00`,`0x04`,`0x08`, and `0x0C`, in that order. In this example, symbolic execution provided the instructions required to activate the path `0x0C`→`0xC0`→`0xC8`. The final program can be generated by combining the path that was taken from `0x00-0x0C` (fuzzer generated) with the path from `0x0C-0xC8` (generated by symbolic execution). ■

3.6 Overall Framework for FuSS

Algorithm 1 illustrates the five key components of our hardware fuzzing framework with selective symbolic execution (FuSS), as outlined in Figure 5. First, design instrumentation is performed (line 4). Then, fuzzing is carried out (line 6) until a coverage plateau is detected (line 8). Upon plateau detection, hardware-program context mapping is initiated (line 9), followed by the extraction of source and destination state pairs (line 10). Symbolic execution is then used to generate new inputs and overcome the plateau (lines 12–14), after which the fuzzing process resumes. The previous sections described each of these components. In this section, we provide an illustrative example to highlight the key concepts using Figure 9. In this example, we perform fuzzing of a simple processor, `toyProcessor`. The Verilog implementation is shown in Figure 9a, which takes `clk`, `reset`, `data_in`, and `flags` as inputs and produces a 5-bit `state` as output. The corresponding control flow graph (CFG) is presented in Figure 9b, where each node represents a basic block in the hardware design. The example illustrates two separate fuzzing iterations and two selective symbolic execution iterations, denoted by ⟶, ⇢ and ⟶, ⇢, respectively.

First Fuzzing Iteration (⟶): During the first fuzzing iteration, the fuzzer successfully explores basic blocks **BB0**, **BB1**, **BB2**, **BB12**, and **BB13**. However, it fails to solve the branch at **BB2** to reach **BB3**,

Algorithm 1: Fuzzing with Selective Symbolic Execution (FuSS)

Input: Hardware design $\mathcal{P}$, Initial input corpus $\mathcal{I}$, Coverage goal C_g, Plateau window W
Output: Test input set $\mathcal{T}$ achieving coverage $\geq C_g$

$\mathcal{T} \leftarrow \mathcal{I}$ // Testset is initialized with corpus
$C(0) \leftarrow 0$ // Initial coverage is set to zero
$t \leftarrow 1$ // Iteration counter is initialized
$\mathcal{H} \leftarrow$ `instrumentDesign`$(\mathcal{P})$ // Design instrumentation in Section 3.1
while $C(t-1) < C_g$ **do**
  $\mathcal{T}_{new} \leftarrow$ `runFuzzer`$(\mathcal{H}, \mathcal{T}, T)$ // Invoke fuzzer in Section 3.2
  $C(t) \leftarrow$ `measureCoverage`$(\mathcal{H}, \mathcal{T} \cup \mathcal{T}_{new})$ // Coverage measurement
  if $(t \geq W$ ***and*** $\frac{C(t)-C(t-W)}{W} < \theta)$ **then** // Plateau detection
    $\mathcal{CFG} \leftarrow$ `mapContext`$(\mathcal{H}, \mathcal{T}_{new})$ // Context mapping in Section 3.3
    $\mathcal{SD} \leftarrow$ `getSourceDestPairs`$(\mathcal{CFG}, \mathcal{T})$ // Extract src/dest in Section 3.4
    foreach *(src, dst)* $\in (\mathcal{SD})$ **do**
      State $s \leftarrow$ `statePrep`$(\mathcal{H}, \mathcal{T}_{new})$ // State preparation in Section 3.5
      Constraint $\phi \leftarrow$ `runSymbolic`(s, src, dst) // Invoke symbolic execution
      if `solveSMT`$(\phi) \neq \emptyset$ **then**
        $\mathcal{T} \leftarrow \mathcal{T} \cup \{i\}$ // Append test cases
      end
    end
  else
    $\mathcal{T} \leftarrow \mathcal{T} \cup \mathcal{T}_{new}$ // Append test cases to the fuzzing corpus
  end
  $t \leftarrow t + 1$ // Increment the iteration counter
end
return $\mathcal{T}$

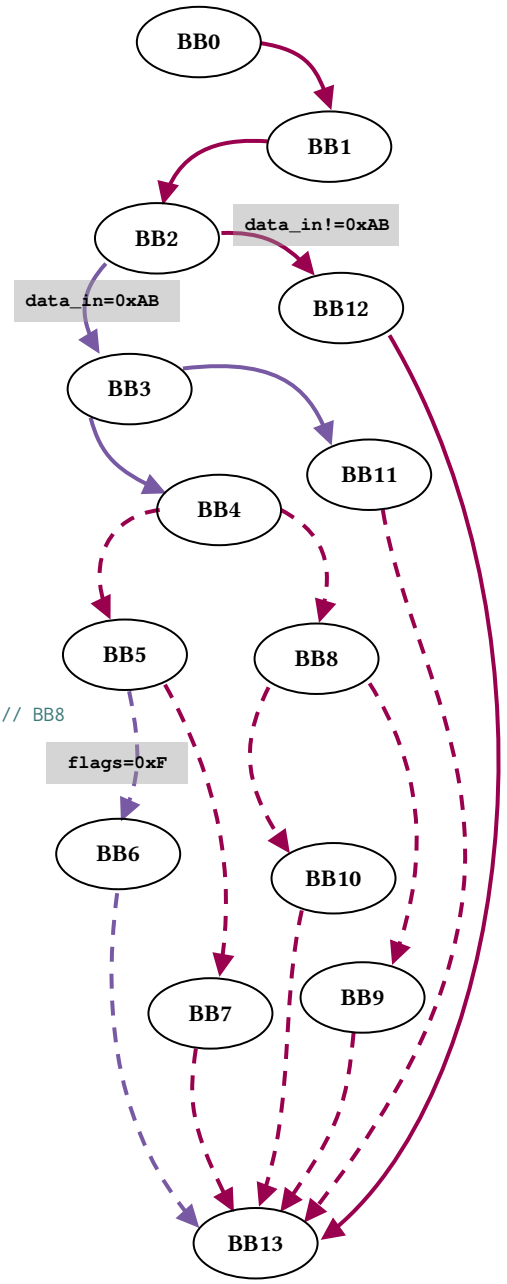
due to the condition `data_in == 32'hAB`. The probability of satisfying this condition through random input is extremely low, only $\frac{1}{2^{32}}$. As a result, the fuzzer hits a coverage plateau, unable to progress further into the design. This triggers the coverage plateau detection mechanism described in Section 3.2, prompting the invocation of selective symbolic execution.

First Selective Symbolic Execution Iteration (⟶): When the selective symbolic execution is invoked, the framework first analyzes the CFG with the inputs that were generated by the fuzzer to look for source and destination pairs as discussed in Section 3.3 and Section 3.4. In this particular example, it identifies the corresponding source and destination pairs as {(**BB2**, **BB4**), (**BB2**, **BB11**)}. With this information, the symbolic execution engine is invoked using the corresponding source and destination points as the start and end points for the symbolic execution. Symbolic engine solves the branch condition at the **BB3** and provides the input value of `0xAB` for `data_in`. This essentially adds new basic blocks of **BB3**, **BB4**, and **BB11** into the explored set of basic blocks.

Second Fuzzing Iteration (⇢): After the previous selective symbolic iteration is finished, the corresponding inputs that were generated is then integrated into the fuzzing corpus, and the fuzzer can start exploring the design again for the second iteration. In this iteration, the fuzzer is able to add

```
1  module toyProcessor (
2    input clk,
3    input reset,
4    input [31:0] data_in,
5    input [3:0] flags,
6    output reg [4:0] state );
7  localparam S0 = 0, S1 = 1, S2 = 2,
8     S3 = 3, S4 = 4, S5 = 5, S6 = 6, S7 = 7;
9  always @(posedge clk or posedge reset) begin
10   if (reset) state <= S0; // BB0
11     else begin
12       case(state)
13         S0: state <= S1; // BB1
14         S1:if (data_in==32'hAB) begin // BB2
15             if (flags[0]==1'b1) begin // BB3
16               if (flags[1]==1'b0) begin // BB4
17                 if (flags==4'b1111) begin // BB5
18                   state <= S2; // BB6
19                 end else begin
20                   state <= S3; //BB7
21                 end
22               end else if (flags[3]==1'b1) begin // BB8
23                 state <= S4; // BB9
24               end else begin
25                 state <= S5; // BB10
26               end
27             end else begin
28               state <= S6; // BB11
29             end
30           end else begin
31             state <= S7; // BB12
32         end
33         S2, S3, S4,
34         S5, S6, S7: state <= S7; // BB13
35         default: state <= S7;
36       endcase
37     end
38 end
39 endmodule
```

(a) Simple processor design in Verilog.



(b) CFG of the simple processor design.

Fig. 9. An illustrative example demonstrating the end-to-end functionality of the proposed *FuSS* framework. In the first iteration, the fuzzing engine successfully explores basic blocks **BB0**, **BB1**, **BB2**, **BB12**, and **BB13**, as indicated by $\longrightarrow$. However, it fails to solve the branch condition at **BB2**, resulting in a coverage plateau. To address this, the framework invokes selective symbolic execution. With $N = 2$, the source-destination pairs are set as (**BB2**, **BB4**), (**BB2**, **BB11**), and the explored path is shown by $\longrightarrow$. This analysis generates an input value of `0xAB` for `data_in`. The fuzzer is then resumed for a second iteration ($\dashrightarrow$), continuing exploration until the next plateau. This process repeats until the desired coverage goal is achieved.

BB4, **BB5**, **BB7**, **BB8**, **BB9**, and **BB10**. However, the fuzzer fails to solve the branch condition at **BB5**. This eventully triggers the coverage plateau detection mechanism for the second time.

Second Selective Symbolic Execution Iteration (`-->`): After the second fuzzing iteration triggers the coverage plateau, the framework again starts to compute the corresponding source and destination pairs for the current states of the design. This time, corresponding source and destination pairs become {(**BB5**, **BB13**)}. Based on the available source and destination pair, the symbolic execution engine will resolve the branch at **BB5** to set the input value of `4'b1111` for `flags` variable. With this input, the design is completely explored with the final basic block **BB6**.

Due to the simplicity of the example, only two fuzzing iterations and two selective symbolic execution iterations were sufficient to explore the entire design. In practice, real-world hardware designs typically require many more iterations to achieve the desired coverage level.

4 Theoretical Analysis of FuSS

In this section, our objective is to show that FuSS can reach a specific coverage goal in fewer fuzzing iterations compared to existing fuzzing techniques. For this purpose, we establish a runtime bound for the FuSS. Due to the feedback loop, each fuzzing iteration may depend on previous ones, potentially making activation probabilities dependent. We first define several parameters. Next, we analyze the theoretical bounds using Doob's Martingale analysis [10, 31].

4.1 Definitions

For the analysis, we define the following parameters that characterize the success probabilities of different fuzzing strategies and the total number of fuzzing iterations.

- *Success* event: Activation of a hard-to-activate corner case after reaching a coverage plateau.
- Success probabilities: $P_i : i \in \{f, f+p, f+s\}$
  - P_f: success probability with traditional fuzzing.
  - P_{f+p}: success probability with fuzzing with property checking.
  - P_{f+s}: success probability with fuzzing with selective symbolic execution (FuSS).
- Total number of Fuzzing iterations: T

4.2 Theoretical Bounds

In fuzzing with a feedback loop, the activation probability of a hard-to-activate node depends on the previous iterations. To model these dependencies, we employ the edge exposure variant of Doob's martingale, which allows us to analyze the concentration of the total number of successful discoveries around its expected value while considering the dependencies introduced at each iteration.

- Let X be the total number of successful discoveries (reaching new paths) after T iterations.
- Let $Z = (Z_1, Z_2, \ldots, Z_T)$ be the sequence of random variables representing the random choices made during the fuzzing iterations.
- Let $\mathcal{F}t$ be the sigma-algebra generated by $Z_1, \ldots, Z_t$, representing the information up to the t-th iteration.
- Let the Doob martingale sequence $(M_t)_{t=0}^{T} = \mathbb{E}[X|\mathcal{F}_t]$.

$$\mathbb{E}[M_{t+1}|\mathcal{F}_t] = \mathbb{E}[\mathbb{E}[X|\mathcal{F}_{t+1}]|\mathcal{F}_t] = \mathbb{E}[X|\mathcal{F}_t] = M_t \tag{6}$$

By construction, M_t is a martingale due to Equation 6. In our context, since discovering a new state or bug in any iteration can change the expected total number of successes by at most 1, we set $|M_t - M_{t-1}| \leq 1$. Using the Azuma-Hoeffding inequality [7] for martingales with the above parameters, we obtain Equation 7.

$$Pr[X \leq (1-\delta)\mathbb{E}[X]] \leq e^{\frac{-\delta^2 \mathbb{E}[X]^2}{2T}} \tag{7}$$

Simplifying further with $\mathbb{E}[X] = TP_i$, where P_i is the average activation probability per iteration, and solving for T, we get Equation 8.

$$T \geq \frac{2ln(\frac{1}{\epsilon})}{\delta^2 P_i^2} \tag{8}$$

Since $P_{f+s} > P_{f+p} > P_f$ due to the additional guidance provided by symbolic execution (We show that this relationship holds with empirical results in Section 5), the required number of iterations T decreases accordingly. This demonstrates the efficiency of the FuSS framework in reaching critical states or bugs more quickly than the other methods. The theoretical analysis provides a foundational framework that justifies the integration of symbolic execution into fuzzing, predicts when it is most beneficial, and contextualizes our empirical results to strengthen confidence in the broader applicability and effectiveness of our approach.

5 Experiments

In this section, we demonstrate the effectiveness of our proposed framework. First, we discuss the experimental setup. Next, we empirically find the success probabilities for the theoretical analysis. Then, we demonstrate the effectiveness of our proposed framework in solving the coverage plateau problem for branch coverage as well as toggle coverage. Finally, we analyze the coverage results to determine which areas of the tested implementations were more effectively covered by the FuSS framework compared to traditional fuzzing and fuzzing with property checking.

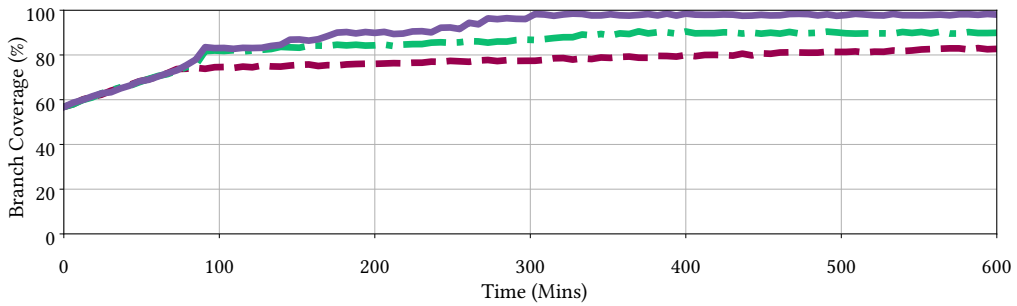
5.1 Experimental Setup

We developed automation scripts for FuSS framework using *GNU Make*, *bash*, and *Python* to streamline the workflow. We use *Verilator* [45] to convert hardware models into cycle-accurate software models for simulation. Verilator serves as the backend in our framework due to its precise modeling of synthesizable Verilog designs. However, our framework is compatible with any compiled simulator that supports cycle-accurate execution.

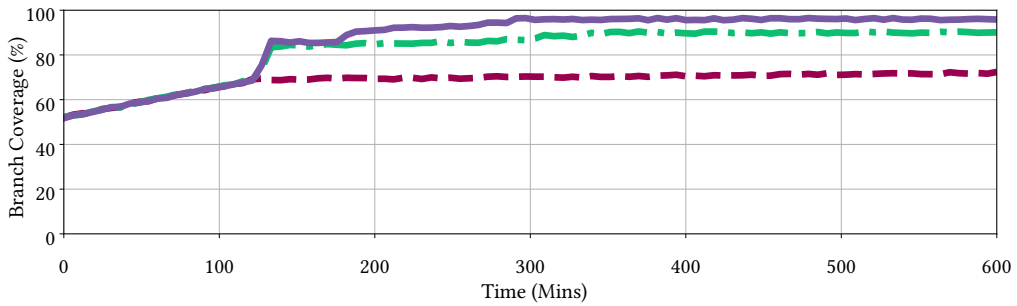
For tasks such as control flow graph (CFG) generation and address decoding, we use the `angr` [51] binary analysis framework, which translates the hardware implementation into the *VEX* [43] intermediate representation for symbolic execution. Constraint solving during symbolic execution and CFG construction is handled by the *Z3* [47] SMT solver, accessed through `angr`'s *claripy* abstraction. Since we do not modify the symbolic execution or SMT solving components, we inherit their well-established soundness [8]. Notably, our framework's correctness does not rely on the SMT encoding itself; coverage is evaluated on the actual hardware simulation. Symbolic execution is used solely to generate input vectors that help overcome fuzzing plateaus. If it fails to do so, it merely results in no additional coverage, without impacting the validity of the overall analysis.

In order to reproduce the results related to fuzzing with property checking, we have used *EBMC* [27] hardware model checker. As the Fuzzer engine we have used a modified version of *DifuzzRTL* [16]. We have used four SoC hardware implementations (PicoSoC, UervSoC, VeeRwolf,

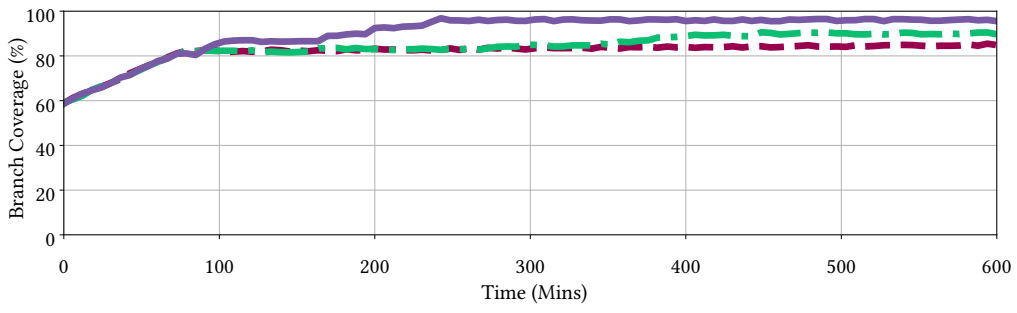
(a) PicoSoC



(b) UeRVSoC



(c) VeeRwolf



(d) CVA6

Fig. 10. Branch coverage results of *FuSS* compared with traditional fuzzing and fuzzing with property checking.

and CVA6 SoC) to evaluate the effectiveness of our proposed approach. All these four SoC configurations include processors that implement RISC-V instruction set architecture and are implemented in *Verilog*. For all the experiments, we have set the following parameters: coverage plateau detection window size as $W = 10$, minimum acceptable coverage increment as $\theta = 0.05$, and maximum number of edges for the destination as $N = 2$. All experiments were conducted on a machine equipped with an x86_64 Intel i7-9700 CPU @ 3.0 GHz, 32 GiB of memory, and an NVIDIA RTX 3090 GPU, running GNU/Linux 22.04.

5.2 Empirical Analysis of the Theoretical Bounds

In the analysis of randomized algorithms, we have to rely on empirical experiments for estimating event probabilities [31]. For example, if we want to analyze the outcome of using an arbitrary coin (without knowing whether it is fair or biased) to determine the chances of winning a game, we first need to empirically determine the probabilities of getting head or tail. Similarly, in this section, we create an experiment to back up our theoretical analysis in Section 4 with empirical results. Specifically, this experiment aims to show $P_{f+s} > P_{f+p} > P_f$. In order to show this, we have selected the *PicoRV* SoC benchmark. We allowed the fuzzing engine to cover the design until it reached the coverage plateau. Once the design reached the coverage plateau, we randomly sampled 1,000 unexplored (hard-to-activate) nodes and applied the three techniques: (i) traditional (continue) fuzzing, (ii) combine fuzzing with property checking, and (iii) our proposed fuzzing with selective symbolic execution. Then we calculated the success probability by computing the ratio of activated nodes to the total sampled.

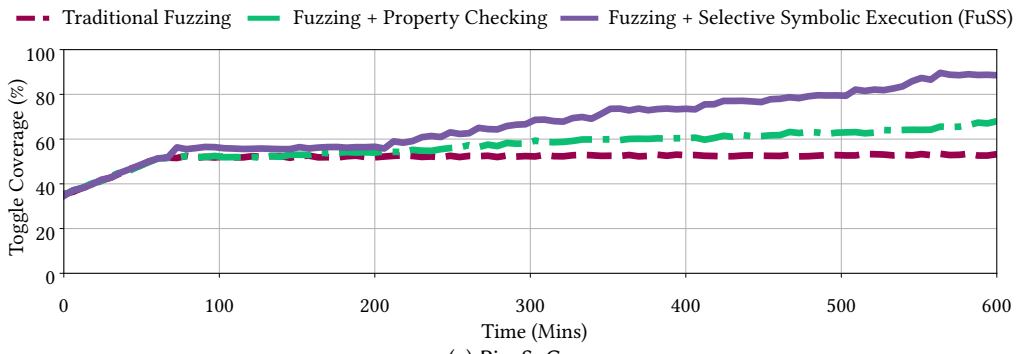
Table 1. Empirical results demonstrating the validity of the relationship ($P_{f+s} > P_{f+p} > P_f$) outlined in Section 4 for different fuzzing iterations.

Fuzzing Duration (hours)		**1**	**2**	**5**	**10**	**15**
Success Probability (P_i)	P_f	0.01	0.09	0.12	0.12	0.12
	P_{f+p}	0.14	0.19	0.24	0.31	0.43
	P_{f+s}	0.42	0.51	0.67	0.73	0.81

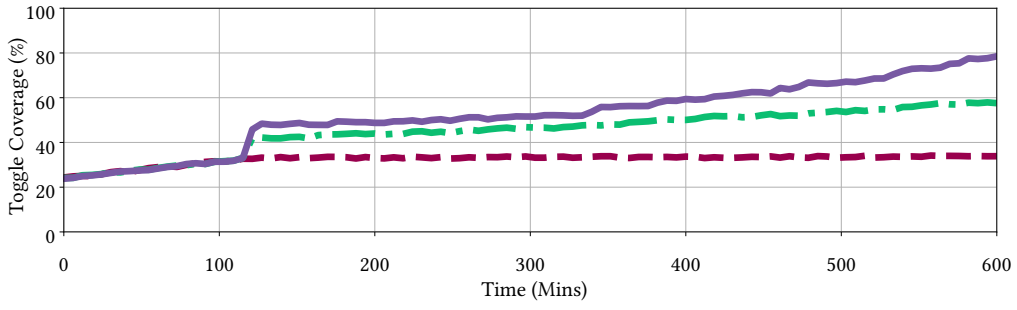
Table 1 presents results for the three activation probabilities after different fuzzing iterations. It confirms the relationship of $P_{f+s} > P_{f+p} > P_f$ in terms of successfully activating unexplored coverage points. Specifically, at each point of time, the success probability of FuSS framework (P_{f+s}) is significantly better compared to the other two fuzzing methods.

5.3 Solving the Coverage Plateau for Branch Coverage

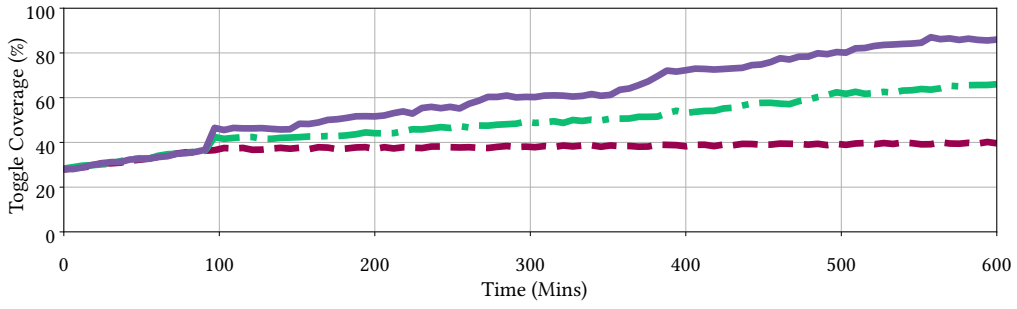
Branch coverage, in the context of hardware verification, measures how well the control flow of a design is exercised by test stimuli, ensuring that all possible decision points—such as conditional statements, multiplexers, and finite state machine transitions—are activated. In complex SoC implementations, certain branches may only be taken under rare conditions, requiring targeted stimulus generation to drive the design into specific states. We have created this experiment to illustrate the branch coverage effectiveness of the proposed FuSS framework compared to traditional fuzzing and fuzzing with property checking. Figure 10 illustrates the result for the four selected RISC-V SoC benchmarks. It can be observed that during the branch coverage, traditional fuzzing reaches the coverage plateau around 80%. While fuzzing with property checking outperforms traditional fuzzing, it also struggles due to the state space explosion. As expected, our framework (FuSS) can achieve close to 100% branch coverage in less than 10 hours due to the seamless integration of selective symbolic execution with prior fuzzing efforts.



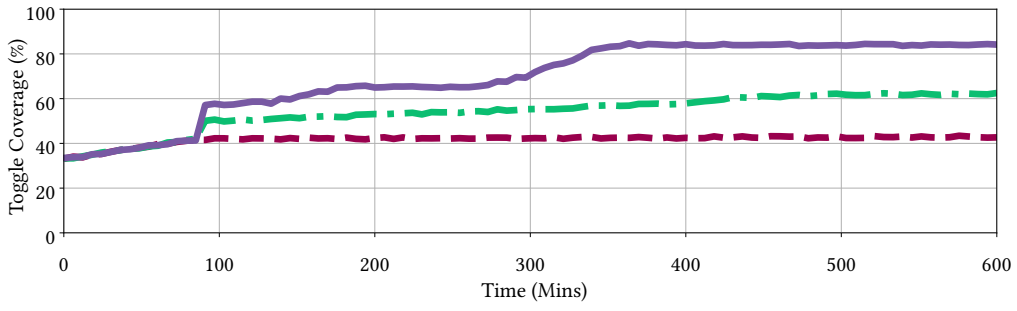
(a) PicoSoC



(b) UeRVSoC



(c) VeeRwolf



(d) CVA6

Fig. 11. Toggle coverage results of *FuSS* compared with traditional fuzzing and fuzzing with property checking.

```
case (mode)
    mode_spi: begin
        buffer = {buffer, io0};
        bitcount = bitcount + 1;
        if (bitcount == 8) begin
            bitcount = 0;
            bytecount = bytecount + 1;
            spi_action;
        end
    end
    mode_dspi_rd, mode_dspi_wr: begin
        buffer = {buffer, io1, io0};
        bitcount = bitcount + 2;
        if (bitcount == 8) begin
            bitcount = 0;
            bytecount = bytecount + 1;
            spi_action;
        end
    end
    ....
```

(a) SPI interface used in the PicoSoC.

```
if (outport_awvalid_o) // Command phase
begin
    // Command not accepted yet
    if (!outport_awready_i)
    begin
        write_hold_r      = 1'b1;
        write_dataphase_r = 1'b0;
    end
    // Address + Data, single, all accepted
    ...
else if (write_hold_r) // Data phase
begin
    // Last data accepted
    if (outport_wvalid_o && outport ...)
    begin
        write_hold_r      = 1'b0;
        write_dataphase_r = 1'b0;
    end
    // Stay in data phase, data outstanding
    ...
```

(b) AXI4_Lite bus arbitration logic in UeRVSoC.

Fig. 12. Hardware interface logic from PicoSoC and UeRVSoC that demonstrated better toggle coverage with our proposed FuSS framework, compared to the fuzzing with property checking.

5.4 Solving the Coverage Plateau for Toggle Coverage

Toggle coverage is inherently more challenging to achieve than branch coverage because it not only requires exercising different execution paths but also demands triggering low-level state transitions within hardware components. Unlike branch coverage, which primarily focuses on evaluating conditional statements and control flow paths, toggle coverage captures changes at the individual bit level in registers, latches, and flip-flops. This complexity arises from the inherent concurrency in hardware, where multiple signals operate in parallel, often requiring precise timing and specific sequences of inputs to transition between states. Additionally, the interaction between pipeline stages, clock domain crossings, and asynchronous events further complicates achieving high toggle coverage. We have created this experiment to illustrate the effectiveness of the proposed FuSS framework compared to traditional fuzzing and fuzzing with property checking. Figure 11 illustrates the result for the four selected RISC-V SoC benchmarks. It can be observed that the during the toggle coverage, traditional fuzzing reaches the coverage plateau around 40%. While fuzzing with property checking performs better than traditional fuzzing, it also fails to reach beyond 70% due to the state space explosion. In contrast, our proposed framework (FuSS) can reach up to 90% toggle coverage in less than 10 hours due to effective utilization of prior fuzzing history with selective symbolic execution.

5.5 Analysis of Handling Hard-to-Activate Regions

Based on the results from the toggle coverage experiments, we conducted a deeper analysis to identify areas that remained unexplored by traditional fuzzing and fuzzing with property checking

but were effectively covered by the FuSS framework. To perform this analysis, we utilized Verilog simulation value change dump (VCD) files to track signal activity and examine which parts of the design were exercised by the generated input test patterns.

Hardware Interfaces: A common area with lower toggle coverage with traditional fuzzing and fuzzing with property checking across all four SoC configurations (PicoSoC, UervSoC, VeeRwolf, and CVA6 SoC) was the hardware interfaces. In particular, different variants of AXI4 implementations and external peripherals such as UART, SPI, and I2C introduced bottlenecks that limited signal activation beyond those points. For example, Figure 12a shows the SPI interface used by the PicoSoC implementation. Similarly, Figure 12b shows the AXI4 interface utilized by the wishbone bus architecture of the UeRVSoC. In both of these examples, our proposed framework was able to outperform the toggle coverage achieved by both traditional fuzzing and fuzzing with property checking. As we discussed in Section 2.1, these interfaces require specific initialization sequences, precise timing constraints, protocol handshakes, and state-based dependencies, which selective symbolic execution was able to successfully figure out to generate input patterns.

```
riscv::TOR: begin
  base = '0;
  mask = '0;
  size = '0;
  // check that the requested address is
  // in between the two configuration addresses
  if (addr_i >= ({2'b0, conf_addr_prev_i}
    << 2) && addr_i < ({2'b0, conf_addr_i} ...
    match_o = 1'b1;
  end else match_o = 1'b0;
  // synthesis translate_off
  if (match_o == 0) begin
    assert (addr_i >= ({2'b0, conf_addr_i} ...
  ...
riscv::NAPOT: begin
  // use the extracted trailing ones
  size = {{(32 - $clog2(CVA6Cfg.PLEN)) ...
  mask = '1 << size;
```

(a) Code Snippet from Physical Memory Protection (PMP) unit implementation in CVA6 SoC.

```
WRITE_STATE_IDLE: begin
  s_axi_awready_next = 1'b1;
  if (s_axi_awready && s_axi_awvalid) begin
    write_id_next = s_axi_awid;
    ...
WRITE_STATE_BURST: begin
  s_axi_wready_next = 1'b1;
  if (s_axi_wready && s_axi_wvalid) begin
    mem_wr_en = 1'b1;
    if (write_burst_reg != 2'b00) begin
      write_addr_next = write_addr_reg ...;
    end
    write_count_next = write_count_reg - 1;
    if (write_count_reg > 0) begin
      write_state_next = WRITE_STATE_BURST;
    end else begin
      s_axi_wready_next = 1'b0;
      if (s_axi_bready || !s_axi_bvalid)
```

(b) Memory implementation of WeeRwolf SoC that is connected to the SoC via AXI4 interface.

Fig. 13. Memory and buffer related components of the SoC implementation of CVA6 and WeeRwolf, which demonstrated better toggle coverage with FuSS compared to fuzzing with property checking.

Intermediate Buffers and Memory: Another area with consistently lower toggle coverage with traditional fuzzing and fuzzing with property checking across all four SoC configurations was the intermediate buffers and memory subsystems. Specifically, structures such as FIFO buffers, caches, and internal register files, and their associated logic consistently had lower toggle coverage. For example, Figure 13a shows the Physical Memory Protection (PMP [39]) implementation in the CVA6 SoC implementation. Similarly, Figure 13b presents a part of the memory implementation that connects with the rest of the SoC via AXI4 interface. In both of these cases, our proposed

FuSS framework was able to solve the constraints in order to achieve better toggle coverage in this particular area compared to traditional fuzzing and fuzzing with property checking.

Based on this analysis, it is clear that complex logic conditions available in modern SoC implementations are hard to reach with traditional fuzzing. Although fuzzing with property checking can improve the coverage, the lack of controllability over the search space still causes SMT solver timeouts when dealing with complex corner cases. In contrast, our proposed approach effectively utilizes the inputs generated from the fuzzing effort and dynamically selects the conditions to solve, significantly improving the toggle coverage, outperforming both traditional fuzzing and fuzzing with property checking while significantly reducing the test generation time.

6 Conclusion

Fuzzing is widely used for hardware validation since it can outperform traditional validation methods with random test vectors. Although state-of-the-art fuzzing methods utilize property checking to provide better coverage than traditional fuzzing, they face the coverage plateau problem since they inherit fundamental limitations of property checking, including state space explosion. In this paper, we presented a coverage-directed fuzzing framework with selective symbolic execution (FuSS). Unlike property checking based fuzzing that tries to generate an input sequence from the start state, we utilize the current fuzzing trajectory to identify the closest internal state for selective symbolic execution to reach unexplored regions. In order to achieve seamless integration between the fuzzing engine and the symbolic execution engine, we developed an automated methodology to perform coverage plateau detection, hardware-program context mapping, and symbolic state preparation. We performed both theoretical analysis and empirical evaluation to demonstrate the validity of the improvements provided by FuSS compared to traditional fuzzing as well as fuzzing with property checking. Extensive experimental evaluation using four RISC-V based SoC configurations demonstrated that FuSS can overcome the coverage plateau problem by drastically reducing the number of fuzzing iterations required to achieve a coverage goal compared to state-of-the-art fuzzing solutions. Our framework with selective symbolic execution can be deployed on top of any existing fuzzing engines to overcome their inherent coverage plateau problem.

Acknowledgments

This work was partially supported by the Semiconductor Research Corporation (SRC) grant 2025-HW-3307.

References

[1] Anastasios Andronidis and Cristian Cadar. 2022. Snapfuzz: high-throughput fuzzing of network applications. In *Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis*. 340–351.

[2] Pallavi Borkar, Chen Chen, Mohamadreza Rostami, Nikhilesh Singh, Rahul Kande, Ahmad-Reza Sadeghi, Chester Rebeiro, and Jeyavijayan Rajendran. 2024. Whisperfuzz: White-box fuzzing for detecting and locating timing vulnerabilities in processors. *arXiv preprint arXiv:2402.03704* (2024).

[3] Sadullah Canakci, Chathura Rajapaksha, Leila Delshadtehrani, Anoop Nataraja, Michael Bedford Taylor, Manuel Egele, and Ajay Joshi. 2023. Processorfuzz: Processor fuzzing with control and status registers guidance. In *2023 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 1–12.

[4] Chen Chen, Vasudev Gohil, Rahul Kande, Ahmad-Reza Sadeghi, and Jeyavijayan Rajendran. 2023. PSOFuzz: Fuzzing processors with particle swarm optimization. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 1–9.

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

[6] Mingsong Chen, Xiaoke Qin, Heon-Mo Koo, and Prabhat Mishra. 2012. *System-level validation: high-level modeling and directed test generation techniques*. Springer.

[7] Fan Chung and Linyuan Lu. 2006. Concentration inequalities and martingale inequalities: a survey. *Internet mathematics* 3, 1 (2006), 79–127.
[8] Edmund M Clarke. 1997. Model checking. In *Foundations of Software Technology and Theoretical Computer Science: 17th Conference Kharagpur, India, December 18–20, 1997 Proceedings 17*. Springer, 54–56.
[9] Nusrat Farzana Dipu, Muhammad Monir Hossain, Kimia Zamiri Azar, Farimah Farahmandi, and Mark Tehranipoor. 2024. FormalFuzzer: Formal Verification Assisted Fuzz Testing for SoC Vulnerability Detection. In *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 355–361.
[10] Joseph L Doob. 1971. What is a martingale? *The American Mathematical Monthly* 78, 5 (1971), 451–463.
[11] Vasudev Gohil, Rahul Kande, Chen Chen, Ahmad-Reza Sadeghi, and Jeyavijayan Rajendran. 2024. Mabfuzz: Multi-armed bandit algorithms for fuzzing processors. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1–6.
[12] Emre Güler, Cornelius Aschermann, Ali Abbasi, and Thorsten Holz. 2019. {AntiFuzz}: impeding fuzzing audits of binary executables. In *28th USENIX Security Symposium (USENIX Security 19)*. 1931–1947.
[13] Emre Güler, Sergej Schumilo, Moritz Schloegel, Nils Bars, Philipp Görz, Xinyi Xu, Cemal Kaygusuz, and Thorsten Holz. 2024. Atropos: Effective fuzzing of web applications for server-side vulnerabilities. In *USENIX Security Symposium*.
[14] Muhammad Monir Hossain, Nusrat Farzana Dipu, Kimia Zamiri Azar, Fahim Rahman, Farimah Farahmandi, and Mark Tehranipoor. 2023. TaintFuzzer: SoC security verification using taint inference-enabled fuzzing. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 1–9.
[15] Muhammad Monir Hossain, Arash Vafaei, Kimia Zamiri Azar, Fahim Rahman, Farimah Farahmandi, and Mark Tehranipoor. 2023. Socfuzzer: Soc vulnerability detection using cost function enabled fuzz testing. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1–6.
[16] Jaewon Hur, Suhwan Song, Dongup Kwon, Eunjin Baek, Jangwoo Kim, and Byoungyoung Lee. 2021. Difuzzrtl: Differential fuzz testing to find cpu bugs. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1286–1303.
[17] Aruna Jayasena, Richard Bachmann, and Prabhat Mishra. 2025. CISELEAKS: Information Leakage Assessment of Cryptographic Instruction Set Extension Prototypes. *IEEE Transactions on Information Forensics and Security* (2025).
[18] Aruna Jayasena and Prabhat Mishra. 2023. Directed Test Generation for Hardware Validation: A Survey. *Comput. Surveys* (2023).
[19] Aruna Jayasena and Prabhat Mishra. 2023. Scalable detection of hardware trojans using ATPG-based activation of rare events. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42, 12 (2023), 4450–4462.
[20] Aruna Jayasena and Prabhat Mishra. 2024. HIVE: Scalable hardware-firmware co-verification using scenario-based decomposition and automated hint extraction. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2024).
[21] Dae R Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. 2019. Razzer: Finding kernel race bugs through fuzzing. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 754–768.
[22] Rahul Kande, Addison Crump, Garrett Persyn, Patrick Jauernig, Ahmad-Reza Sadeghi, Aakash Tyagi, and Jeyavijayan Rajendran. 2022. {TheHuzz}: Instruction fuzzing of processors using {Golden-Reference} models for finding {Software-Exploitable} vulnerabilities. In *31st USENIX Security Symposium (USENIX Security 22)*. 3219–3236.
[23] Hyungseok Kim, Soomin Kim, Jungwoo Lee, and Sang Kil Cha. 2024. AsFuzzer: Differential Testing of Assemblers with Error-Driven Grammar Inference. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1099–1111.
[24] Juhwan Kim, Jihyeon Yu, Youngwoo Lee, Dan Dongseong Kim, and Joobeom Yun. 2024. HD-FUZZ: Hardware dependency-aware firmware fuzzing via hybrid MMIO modeling. *Journal of Network and Computer Applications* 224 (2024), 103835.
[25] Kyungtae Kim, Dae R Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. 2020. HFL: Hybrid Fuzzing on the Linux Kernel.. In *NDSS*.
[26] Sunwoo Kim, Young Min Kim, Jaewon Hur, Suhwan Song, Gwangmu Lee, and Byoungyoung Lee. 2022. {FuzzOrigin}: Detecting {UXSS} vulnerabilities in browsers through origin fuzzing. In *31st usenix security symposium (usenix security 22)*. 1008–1023.
[27] Daniel Kroening et al. [n. d.]. EBMC: The Enhanced Bounded Model Checker. https://www.cprover.org/ebmc/. Accessed: 2024-10-16.
[28] Dongge Liu, Van-Thuan Pham, Gidon Ernst, Toby Murray, and Benjamin IP Rubinstein. 2022. State selection algorithms and their impact on the performance of stateful network protocol fuzzing. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 720–730.
[29] Yuwei Liu, Yanhao Wang, Xiangkun Jia, Zheng Zhang, and Purui Su. 2024. AFGen: Whole-Function Fuzzing for Applications and Libraries. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1901–1919.
[30] Zhengxiong Luo, Junze Yu, Feilong Zuo, Jianzhong Liu, Yu Jiang, Ting Chen, Abhik Roychoudhury, and Jiaguang Sun. 2023. Bleem: Packet sequence oriented fuzzing for protocol implementations. In *32nd USENIX Security Symposium*

(USENIX Security 23). 4481–4498.
[31] Michael Mitzenmacher and Eli Upfal. 2017. *Probability and computing: Randomization and probabilistic techniques in algorithms and data analysis*. Cambridge university press.
[32] Stefan Nagy, Anh Nguyen-Tuong, Jason D Hiser, Jack W Davidson, and Matthew Hicks. 2021. Same coverage, less bloat: Accelerating binary-only fuzzing with coverage-preserving coverage-guided tracing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 351–365.
[33] Roberto Natella. 2022. Stateafl: Greybox fuzzing for stateful network servers. *Empirical Software Engineering* 27, 7 (2022), 191.
[34] Roberto Natella and Van-Thuan Pham. 2021. Profuzzbench: A benchmark for stateful protocol fuzzing. In *Proceedings of the 30th ACM SIGSOFT international symposium on software testing and analysis*. 662–665.
[35] Lianglu Pan, Shaanan Cohney, Toby Murray, and Van-Thuan Pham. 2024. EDEFuzz: A Web API Fuzzer for Excessive Data Exposures. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–12.
[36] Hui Peng, Zhihao Yao, Ardalan Amiri Sani, Dave Jing Tian, and Mathias Payer. 2023. {GLeeFuzz}: Fuzzing {WebGL} Through Error Message Guided Mutation. In *32nd USENIX Security Symposium (USENIX Security 23)*. 1883–1899.
[37] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2020. Aflnet: A greybox fuzzer for network protocols. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 460–465.
[38] Shisong Qin, Fan Hu, Zheyu Ma, Bodong Zhao, Tingting Yin, and Chao Zhang. 2023. Nsfuzz: Towards efficient and state-aware network service fuzzing. *ACM Transactions on Software Engineering and Methodology* 32, 6 (2023), 1–26.
[39] RISC-V Community News. 2024. Adding Physical Memory Protection to the VeeR EL2 RISC-V Core. *RISC-V International Blog* (March 2024). https://riscv.org/blog/2024/03/adding-physical-memory-protection-to-the-veer-el2-risc-v-core-2/
[40] Mohamadreza Rostami, Chen Chen, Rahul Kande, Huimin Li, Jeyavijayan Rajendran, and Ahmad-Reza Sadeghi. 2024. Fuzzerfly Effect: Hardware Fuzzing for Memory Safety. *IEEE Security & Privacy* (2024).
[41] Mohamadreza Rostami, Marco Chilese, Shaza Zeitouni, Rahul Kande, Jeyavijayan Rajendran, and Ahmad-Reza Sadeghi. 2024. Beyond random inputs: A novel ml-based hardware fuzzing. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1–6.
[42] Sergej Schumilo, Cornelius Aschermann, Andrea Jemmett, Ali Abbasi, and Thorsten Holz. 2022. Nyx-net: network fuzzing with incremental snapshots. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 166–180.
[43] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2015. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. (2015).
[44] Sven Smolka, Jens-Rene Giesen, Pascal Winkler, Oussama Draissi, Lucas Davi, Ghassan Karame, and Klaus Pohl. 2023. Fuzz on the beach: Fuzzing solana smart contracts. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 1197–1211.
[45] Wilson Snyder. 2010. Verilator: Fast, free, but for me? *DVClub Presentation* (2010), 11.
[46] Flavien Solt, Katharina Ceesay-Seitz, and Kaveh Razavi. 2024. Cascade: CPU fuzzing via intricate program generation. In *Proc. 33rd USENIX Secur. Symp.* 1–18.
[47] Z3 solver. 2023. A theorem prover from Microsoft Research. https://github.com/Z3Prover/z3.
[48] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting fuzzing through selective symbolic execution.. In *NDSS*, Vol. 16. 1–16.
[49] Yuichi Sugiyama, Reoma Matsuo, and Ryota Shioya. 2023. SurgeFuzz: Surge-Aware Directed Fuzzing for CPU Designs. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE, 1–9.
[50] Timothy Trippel, Kang G Shin, Alex Chernyakhovsky, Garret Kelly, Dominic Rizzo, and Matthew Hicks. 2022. Fuzzing hardware like software. In *31st USENIX Security Symposium (USENIX Security 22)*. 3237–3254.
[51] Fish Wang and Yan Shoshitaishvili. 2017. Angr-the next generation of binary analysis. In *2017 IEEE Cybersecurity Development (SecDev)*. IEEE, 8–9.
[52] Feifan Wu, Zhengxiong Luo, Yanyang Zhao, Qingpeng Du, Junze Yu, Ruikang Peng, Heyuan Shi, and Yu Jiang. 2024. Logos: Log guided fuzzing for protocol implementations. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1720–1732.
[53] Yilun Wu, Tong Zhang, Changhee Jung, and Dongyoon Lee. 2023. DEVFUZZ: automatic device model-guided device driver fuzzing. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 3246–3261.
[54] Hao Xiong, Qinming Dai, Rui Chang, Mingran Qiu, Renxiang Wang, Wenbo Shen, and Yajin Zhou. 2024. Atlas: Automating Cross-Language Fuzzing on Android Closed-Source Libraries. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 350–362.
[55] Zhenhua Yu, Haolu Wang, Dan Wang, Zhiwu Li, and Houbing Song. 2022. CGFuzzer: A fuzzing approach based on coverage-guided generative adversarial networks for industrial IoT protocols. *IEEE Internet of Things Journal* 9, 21 (2022), 21607–21619.

[56] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*. 745–761.

[57] Zhuo Zhang, Wei You, Guanhong Tao, Yousra Aafer, Xuwei Liu, and Xiangyu Zhang. 2021. Stochfuzz: Sound and cost-effective fuzzing of stripped binaries by incremental and stochastic rewriting. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 659–676.

[58] Bodong Zhao, Zheming Li, Shisong Qin, Zheyu Ma, Ming Yuan, Wenyu Zhu, Zhihong Tian, and Chao Zhang. 2022. {StateFuzz}: System {Call-Based}{State-Aware} Linux Driver Fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*. 3273–3289.