SymbFuzz: Symbolic Execution Guided Hardware Fuzzing

Samit Shahnawaz Miftah
Department of Electrical, Computer, and Systems Engineering
Rensselaer Polytechnic Institute
Troy, New York, USA
miftas@rpi.edu

Amisha Srivastava
Department of Electrical and Computer Engineering
University of Texas at Dallas
Dallas, Texas, USA
amisha.srivastava@utdallas.edu

Hyunmin Kim
Technology Innovation Institute
Abu Dhabi, United Arab Emirates
Hyunmin.Kim@tii.ae

Shiyi Wei
Department of Computer Science
University of Texas at Dallas
Dallas, Texas, USA
swei@utdallas.edu

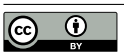
Kanad Basu
Department of Electrical, Computer, and Systems Engineering
Rensselaer Polytechnic Institute
Troy, New York, USA
basuk@rpi.edu

Abstract

Modern hardware incorporates reusable designs to reduce cost and time to market, inadvertently increasing exposure to security vulnerabilities. While formal verification and simulation-based approaches have been traditionally utilized to mitigate these vulnerabilities, formal techniques are hindered by scalability issues, while conventional simulation methods frequently overlook critical edge cases. Fuzzing, as a simulation-based strategy, has demonstrated considerable promise in enhancing the security of both software and hardware; however, it is impeded by challenges such as limited input coverage, difficulties in traversing branching paths, and the complexity of managing circuit parameters, in addition to the limited adaptability of existing hardware fuzzing techniques within industrial workflows. To address these limitations, we propose *SymbFuzz*, an innovative hybrid hardware fuzzing methodology that leverages symbolic execution to achieve superior coverage. *SymbFuzz* is the first hardware fuzzing technique to be implemented on the industry-standard Universal Verification Methodology (UVM), facilitating seamless integration into commercial hardware verification flows. *SymbFuzz* was evaluated on a diverse set of processor RTLs, including OpenTitan (Ibex), CVA6, Rocket-Core, and Mor1kx. These designs span a range of processor architectures and complexities. *SymbFuzz* detected all bugs previously found by existing fuzzers and additionally uncovered 14 new bugs, including a vulnerability in OpenTitan, reported in the CWE 2025 database. It also achieved up to 6.8× faster convergence compared to traditional UVM random testing and over 2×10^4 additional functional coverage points compared to state-of-the-art fuzzers, demonstrating its effectiveness in improving RTL validation across varied processor architectures.

Keywords

Hardware Fuzzing, Hardware Security, Security Vulnerabilities, Processor Architecture.



MICRO '25, Seoul, Republic of Korea

ACM ISBN 979-8-4007-1573-0/25/10
https://doi.org/10.1145/3725843.3756131

ACM Reference Format:
Samit Shahnawaz Miftah, Amisha Srivastava, Hyunmin Kim, Shiyi Wei, and Kanad Basu. 2025. SymbFuzz: Symbolic Execution Guided Hardware Fuzzing. In *58th IEEE/ACM International Symposium on Microarchitecture (MICRO '25), October 18–22, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3725843.3756131

1 Introduction

Increasing design complexity, shrinking feature sizes, and market pressures have intensified hardware vulnerabilities [8, 44, 62]. These vulnerabilities span from functional bugs [34] to critical security flaws [33, 39], posing growing risks to system integrity. The Common Vulnerabilities and Exposures (CVE) and Common Weakness Enumeration (CWE) maintained by MITRE document issues across hardware and software, highlighting the need for rigorous verification [1, 2, 17]. In modern SoCs for critical applications, over 70% of resources are allocated to verification [24]. Security verification, a central element of SoC design, focuses on vulnerabilities that adversaries could exploit to compromise systems. To mitigate such risks, semiconductor firms incorporate a structured security development lifecycle (SDL) into traditional hardware design flows, reinforcing security before chip release [18, 27].

Various techniques and tools in academia and industry have been developed to detect hardware vulnerabilities, including industry-standard tools for formal and simulation-based verification [7, 13, 52, 56, 66], and advanced SoC verification methods such as information flow tracking (IFT) [6, 36, 37, 48, 49, 60, 68], run-time detection [28, 50, 63], concolic testing [42, 69], and hardware fuzzing [14, 15, 29, 31, 35, 38, 45, 61]. Formal verification excels with smaller designs but struggles with larger SoCs due to state space explosion [17, 19]. IFT enhances data tampering and leakage detection yet requires extensive setup [31, 60]. Run-time detection introduces security checks at high re-spin costs [50, 63], while simulation-based methods may overlook security-critical scenarios [42]. Hardware fuzzing has become popular for SoC verification but still faces coverage challenges despite utilizing state-of-the-art optimization methods [14]. Hybrid fuzzing, which merges concolic execution with fuzzing, shows potential in software [16, 40, 47, 54, 67, 69] and may enhance hardware security coverage.

Developing an efficient hardware fuzzer requires addressing key challenges. Accurate **coverage monitoring** is essential for guiding exploration and generating precise constraints. Improving **guidance efficiency** reduces computational overhead, enhancing performance. Furthermore, a robust **reset mechanism** ensures deterministic testing by reliably mapping reset states in complex SoC designs. These challenges underscore the need to balance accuracy, efficiency, and reliability in hardware fuzzing.

To address these challenges, we propose *SymbFuzz*, a hybrid fuzzing technique that improves state coverage with minimal computational overhead by combining Universal Verification Methodology (UVM) [5], widely used in hardware verification, with symbolic execution. *SymbFuzz* employs a white-box approach, offering full visibility into the design under verification (DUV) and using a coverage monitor to track reachable states. The framework identifies the DUV's I/O interface, extracts the reset tree for deterministic testing, and constructs a control flow graph (CFG) to capture branch and register data. The CFG serves as a core component, guiding state exploration by reducing dependence on hard resets, while checkpoints within the CFG enable efficient path alternation and broader node coverage. Seed generation based on the CFG steers exploration toward unvisited states, ensuring thorough and robust testing. **Novelty:** *SymbFuzz* makes the following three key contributions utilizing industry-standard UVM-based verification: (1) Generating checkpoint setups, (2) Enabling the traversal of specific checkpoints after a reset through a saved input pattern sequence, and (3) Developing a symbolic execution-based guidance mechanism to optimize the fuzzing process, as detailed in Section 4.

In this paper, our main contributions are:

- We introduce *SymbFuzz*, a novel hardware verification methodology based on hybrid fuzzing, designed to systematically explore and verify all relevant states associated with specified security properties, ensuring comprehensive coverage.
- *SymbFuzz* is developed using UVM, enabling efficient integration into the hardware security development lifecycle (SDL) and enhancing the robustness of security verification in hardware design.
- We develop a checkpoint mechanism using reset operation, reducing computational cost and enabling partial reset to accelerate path exploration.
- *SymbFuzz* is evaluated on a buggy version of the OpenTitan SoC [46] from *HACK@DAC'24*, where it was able to detect 17 bugs, including one new bug in the original OpenTitan SoC, which was recognized as a novel entry in the CWE's 2025 database. Moreover, *SymbFuzz* outperforms current hardware fuzzing approaches by achieving 2×10^4 additional coverage points.

2 Background

Coverage-guided Fuzzing:

Fuzzing builds upon boundary value analysis (BVA) [21] by systematically testing valid and invalid input ranges to identify vulnerabilities and undefined behaviors [55]. It employs two main techniques: mutation-based (altering existing inputs) and generation-based (creating inputs using protocol specifications). Tools like Google's OSS Fuzz and ClusterFuzz have effectively applied fuzzing across various software systems, uncovering numerous security flaws [20, 23, 26, 31, 41, 51, 57, 59, 65, 70].

Coverage-guided fuzzing enhances test generation by maximizing code path exploration [10, 11, 25, 43]. This approach uses runtime instrumentation to track execution coverage, iteratively optimizing inputs to reach unexplored code regions.

Symbolic Execution: Symbolic execution is a widely used program analysis technique designed to understand the relationship between inputs and program behavior [9, 12, 32]. Rather than using specific, concrete values, symbolic execution employs symbolic values to represent inputs. This allows the interpreter to generate expressions incorporating these symbolic values for program variables. During symbolic execution, the interpreter assigns constraints to conditional branches based on symbolic values, representing the conditions under which different branches will be executed. By solving these constraints, it is possible to determine the branching behavior of the program for various input patterns. In other words, symbolic execution can identify which inputs will cause the program to follow specific execution paths.

Universal Verification Methodology (UVM):

The Universal Verification Methodology (UVM) is a standardized framework widely used in the semiconductor industry to verify hardware designs. Developed by Accellera and based on SystemVerilog [5], UVM offers a reusable and structured verification environment essential for managing the complexity of modern hardware systems. As a widely adopted standard, UVM allows engineers to verify diverse designs, including processors and SoCs, with high accuracy and efficiency. By automating and standardizing test procedures, UVM establishes a robust methodology that simplifies verification and minimizes the risk of undetected bugs.

The UVM framework consists of key components, each crucial to the verification process. These components collaborate to create a robust verification environment supporting test generation, execution, and analysis. Their primary functions are outlined below:

UVM Testbench Environment: The testbench environment forms the essential framework that configures and connects all UVM components. It integrates modules, interfaces, and classes, defines the design under verification (DUV), and initializes tests. Acting as the UVM framework's core, the testbench unifies verification components into a cohesive system.

UVM Agents: Agents represent the primary entities that interact with the DUV. They are responsible for driving and monitoring signals at the DUV's interfaces. Each agent typically consists of a driver, a monitor, and a sequencer: (1) The **Driver** sends specific stimuli to the DUV by translating high-level commands into low-level transactions. (2) The **Monitor** observes the DUV's outputs, captures results, and forwards them to the scoreboard for analysis. (3) The **Sequencer** provides stimulus to the driver, controlling the flow of transactions sent to the DUV.

3 Related Works and Existing Challenges

Recent advances in hardware security verification have introduced diverse techniques with two shared goals: improving scalability and enhancing coverage. This section focuses on hardware fuzzing, which shows promise in achieving both, as noted in Section 1. Early

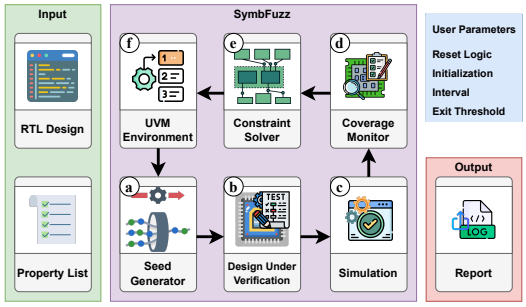


Figure 1: Overview of *SymbFuzz*.

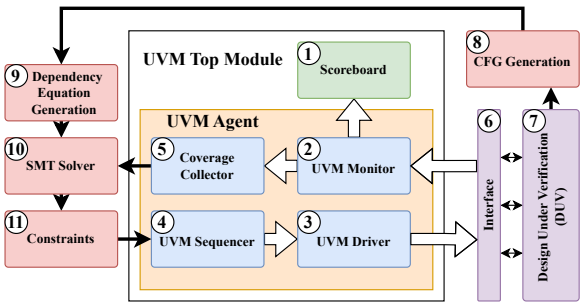


Figure 2: UVM integration into *SymbFuzz*.

work like **RFuzz** introduced mux-coverage-guided fuzzing for FPGAs [35], but faced scalability issues due to high computational overhead [29]. **HyperFuzzing** used grammar-based software modeling [45], yet struggled with FSMs and required manual intervention. **DifuzzRTL** enhanced register-level coverage [29] but lacked insight into FSM transitions and signal toggles [3, 31]. **HWFP** used Verilator for translation, limiting support for complex HDL constructs [61]. **TheHuzz** and **PSOFuzz** optimized input generation via instruction tuning and swarm-based path selection [14, 31], though both incurred feedback and performance limitations. **HypFuzz** integrated formal methods to deepen exploration [15] but risked stagnation. Finally, **GenFuzz** leveraged GPUs for faster execution, though with limited gains in bug detection [38].

State-of-the-art (SOTA) hardware fuzzing techniques face key limitations. Software fuzzers like AFL++ [4], when combined with Verilator [53], fail to model critical hardware behaviors such as clock signals, delays, and four-state logic. As a result, they are inadequate for thorough hardware security verification. These fuzzers also lack bit-level precision and proper handling of register semantics. In contrast, hardware-specific fuzzers often use simplistic coverage metrics (*e.g.,* mux coverage), missing crucial corner cases. Additionally, static feedback mechanisms slow convergence and hinder deep, state-dependent exploration.

Several recent works have used symbolic reasoning to enhance hardware fuzzing. Compared to these, *SymbFuzz* introduces three key advantages: (1) *Detection model: SymbFuzz* inserts confidentiality and privilege assertions directly in RTL. This allows detection of violations that do not affect the visible architectural state. Differential techniques, like *HyPFuzz* [15], may miss such faults. (2) *Design scope: SymbFuzz* drives RTL inputs directly and works across processor types and peripheral IPs without changes. In contrast, fuzzing approaches that generate ISA binaries are limited to processor pipelines and require extra translation. (3) *Guidance efficiency: SymbFuzz* combines symbolic search with a checkpoint-rollback loop. It substitutes concrete register values and constrains solving undefined pin values. This speeds up control state exploration. In our tests, *SymbFuzz* found over 2×10^4 new coverage points and 17 new bugs, including a CWE-classified leak. Prior work showed smaller gains (*e.g.,* only 1.7% branch coverage on the same cores).

Novelty: The novelty of our proposed technique, *SymbFuzz*, can be summarized as follows: (1) We integrate an SMT-based symbolic execution engine with the sequencer–driver structure of UVM. This integration enables us to generate input stimuli based on constraint-solved execution paths that consider the complete RTL state. As a result, we remove the need for software-based approximations or surrogates. (2) We introduce a checkpoint and partial-reset acceleration technique. This approach uses a lightweight snapshot mechanism that saves only the essential transaction history and architectural state. It enables precise re-entry into complex microarchitectural states without requiring a full system reboot, thereby eliminating unnecessary simulation time. (3) We redefine coverage in terms of control-register interaction tuples and formulate their data dependency equations. Solving these equations with an SMT solver analytically guides mutations to RTL regions that are unreachable by random fuzzers. (4) We show that the same test harness can detect bugs in a variety of hardware designs—including the 32-bit in-order Ibex core, the 64-bit out-of-order CVA6 core, and several peripheral IPs—without requiring any ISA-specific modifications. This demonstrates that our approach is portable and effective across different types of architectures.

4 SymbFuzz Architecture

The *SymbFuzz* framework, shown in Figure 1, includes three components: (1) simulation setup (**a** - **c**), (2) coverage measurement (**d**), and (3) seed mutation (**e** - **f**). Inputs are the RTL design of the DUV, its security properties, and user-defined fuzzing parameters. The simulation setup features the *UVM environment* [5], a *seed generator*, and a *simulation platform*. In the UVM environment, the sequence generator creates randomized input vectors and constraints, which the driver sends to the DUV. The monitor checks security properties and generates a report. The *coverage monitor* tracks coverage, directing the *SymbFuzz constraints solver* to optimize input vector generation through.

Algorithm 1 outlines the fuzzing workflow of *SymbFuzz*. It tests the RTL design of the DUV, $\mathcal{D}$, against a property set $\mathcal{P}_{lst}$. The parameters include an interval $\mathcal{I}$ (clock cycles before logging) and a threshold Th (stagnation limit for invoking symbolic execution). A bug report is produced at the end. The process starts by extracting the I/O interface (`IF`) and building the control flow graph (CFG) of $\mathcal{D}$ using *Pyverilog* [58]. Registers are classified, and key states (checkpoints) are marked (lines 2–4).

The CFG begins at reset and covers all execution paths. Each node is a unique hardware state, defined by its register vector. Transitions are edges with unique IDs, based on input conditions and

Algorithm 1: *SymbFuzz* algorithm.

Input: RTL Design, $\mathcal{D}$; Property List, $\mathcal{P}_{lst}$
Parameter: Interval, $\mathcal{I}$; Exit Threshold, Th
Output: Report, $\mathcal{R}$

1 IF ← Extract($\mathcal{D}$) /* IF = Interface */
2 CFG ← Generate($\mathcal{D}$);
3 **Categorize** registers **in** $\mathcal{D}$;
4 ChckPoints $\xleftarrow{mark}$ CFG;
5 $\mathcal{TB} \xleftarrow{Setup}$ UVM environment using IF, $\mathcal{I}$;
6 $\mathcal{D} \xleftarrow{Bind} \mathcal{P}_{lst}$;
7 **while** *All ChkPoints* ***not*** *Covered* **do**
8 $SimFile \xleftarrow{Dump\ VCD} Simulate(\mathcal{D})$;
9 $Coverage \xleftarrow{Record} Read(SimFile)$;
10 **if** *All states covered* **then**
11 **mark** current ChckPoint **as** covered
12 **end**
13 **if** $NoIncrement(Coverage) > Th$ **then**
14 identify(last covered state);
15 find (nearest ChckPoint);
16 **while** *No New State can be reached* **do**
17 find (previous ChckPoint in the CFG);
18 **end**
19 reset back to the ChckPoint;
20 Constraints ← SolveFor(newStateCondition);
21 $\mathcal{TB} \xleftarrow{Apply}$ Constraints;
22 **end**
23 **if** *Bug found* **then**
24 $\mathcal{R} \leftarrow B_{det}$ /* Bug Details, B_{det}, contain clock cycle time and violated property */
25 **end**
26 **end**

context. SymbFuzz logs each transition as a tuple – ⟨edge ID, register value⟩. Checkpoints are nodes with three or more outgoing edges. In our analysis, full coverage signifies each edge is exercised at least once. This ensures exploration of all key behaviors. Coverage is deterministic since node and edge IDs are unique (see Section 4.6). After each simulation, *SymbFuzz* resumes from the last checkpoint. A UVM environment is created using IF, interval $\mathcal{I}$, and property set $\mathcal{P}_{lst}$ (lines 5–6). Fuzzing continues until all checkpoints are covered (line 7). Each round simulates $\mathcal{I}$ cycles, logs the trace (line 8), and updates coverage using control register values (line 9). A checkpoint is marked covered when all descendant paths are exercised (lines 10–12).

If coverage does not improve for several intervals, a stagnation counter increases. Once it passes Th, symbolic execution begins. It finds the latest covered state (line 14), locates the closest checkpoint (line 15), and backtracks the CFG to find unexplored paths (lines 16–18). The system resets and uses a constraint solver to reach those paths (lines 19–21). If no stagnation is seen, unguided fuzzing continues. Detected bugs and property violations are logged with their timestamps and added to the final bug report $\mathcal{R}$ (lines 23–25).

4.1 UVM Integration into SymbFuzz

We illustrate the architecture of *SymbFuzz* and its integration of UVM using Figure 2. Blocks (**1**) to (**7**) depict a standard UVM setup, while *SymbFuzz* extends this framework through blocks (**8**) to (**11**).

SymbFuzz automates the verification framework by first configuring the simulation platform with a UVM template. Next, it extracts the input-output (I/O) interface of the DUV using PyVerilog [58], allowing it to connect with the UVM Driver and UVM Monitor. This establishes the UVM-based verification environment. In order to proceed with the fuzzing mechanism, *SymbFuzz* generates the CFG of the DUV using PyVerilog (**8**). From this CFG, it derives dependency equations that express the control registers as functions of the input interface (**9**). These equations are then solved by an SMT solver (z3-solver), producing constraints that enable UVM sequencers to generate input sequences tailored to alter control register values [22]. This adjustment directs the DUV through unexplored paths in the CFG, especially when coverage improvement reaches a plateau (**10**). The generated constraints are subsequently integrated into the UVM sequencer, initiating the next batch of simulations.

SymbFuzz is designed to handle specific tasks using a combination of assumptions and modular components, each responsible for particular subtasks, as detailed in the following sections. In Listing 1, we present an RTL design for an Arithmetic Logic Unit (ALU) that accepts two data inputs, *A* and *B*, along with an operation code, *op*. The most significant bit of *op* specifies whether the ALU operates in 8-bit or 16-bit mode. This example will be used to elaborate on each task and component in the subsequent sections.

```
1  module ALU (input nrst,input[15:0] A,
2  input[15:0] B,input[3:0] op,output[15:0] Out);
3  typedef enum logic [2:0] {INIT = 0,ADD = 1,
4      SUB = 2,AND = 3,OR = 4,XOR =5} state;
5  logic OPmode;
6  always_comb begin : resetLogic
7      if (!nrst)state=0;
8      else begin
9      state=op[2:0];
10     OPmode=op[3];
11 end
12 always_comb begin : FSM
13     if (OPmode) begin
14         Out[15:8]= 0;
15         case (state)
16             INIT: Out[7:0]=0;
17 ...
18             default: Out = 0;
19         endcase
20     else
21         case (state)
22             INIT: Out=0;
23 ...
24             default: Out = 0;
25         endcase
26     end
27 endmodule
```

Listing 1: A toy example for a DUV (in SystemVerilog).

Figure 3 illustrates the corresponding CFG for Listing 1. In this diagram, every state of the design is represented as a node, while each transition between states is denoted by a uniquely numbered

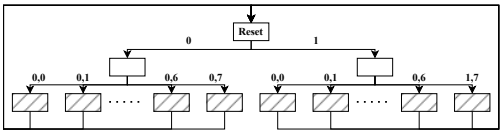


Figure 3: Diagram of CFG from Listing 1.

edge. The numbers represent the parent nodes from which the edge originates. The hatched nodes in the bottom row are leaf nodes: they terminate the execution path and lead only to the reset state, with no further successors.

4.2 Input Definition

During simulation, the RTL top module connects to a testbench or input pins. Test inputs are packed into bit vectors, split into bytes each cycle, and reapplied after resetting the DUV to a known state. *SymbFuzz* uses PyVerilog to extract the DUV's I/O ports and link them to the UVM Driver [58]. It detects fields such as address, data, and control, and randomizes them to produce valid test sequences. These inputs pass through the UVM sequencer, which drives the DUV with functional and fuzzed stimuli (see a in Figure 1). Listing 1 shows *SymbFuzz* extracting ports (*nrst*, *A*, *B*, *op*, *Out*), generating random bitstrings, feeding them into the DUV.

4.3 Deterministic Test Execution

In order to make sure that the DUV is reset at each test case, it is necessary to declare the *clock* and *reset* signal for each IP in the DUV. For example, in Listing 1, the *reset* signal is '*nrst*'. Using this, *SymbFuzz* creates a reset distribution tree. This enables *SymbFuzz* to reset part of the overall design and reduce runtime for fuzzing. Furthermore, while executing fuzz testing, we assume a white-box model. All internal variables/registers are assumed to be observable, and interaction is only possible through the input ports.

4.4 Register Initialization

To optimize wafer space, circuit designs often omit register reset circuitry, leaving DUV registers undefined at reset. Two-state simulators randomly initialize these registers to '0' or '1', while four-state simulators add an 'X' state for uncertainty. Our UVM-based approach uses four-state simulation with predefined *clock* and *reset* signals to generate reset trees and identify control registers.

4.4.1 ***Identification of Control Registers.*** To assess coverage, we first identify control registers that influence the DUV's path in the CFG. In this structure, all non-leaf nodes are governed by control registers, while leaf nodes represent DUV states. Coverage related to a specific state or checkpoint thus maps to the corresponding leaf nodes under that control register. Using this hierarchy, *SymbFuzz* locates CFG branching points and lists the associated control registers. These registers are key to coverage analysis. In the RTL design shown in Listing 1, the registers *OPmode* and *state* serve this role by determining control flow based on their values.

4.4.2 ***Calculation of Dependency.*** To enable the fuzzer to transition the DUV to the next state effectively, two key steps are required: (1) identifying the target state and (2) generating input patterns that guide the DUV into the desired state. In hardware design, state transitions depend on state registers, which are influenced by sequences of input patterns, combinations of inputs, or both. Achieving this requires an understanding of how state registers depend on input ports, which is analyzed using dependency graphs and equations derived through the CFG.

After identifying the control registers (Section 4.4.1), *SymbFuzz* constructs a dependency graph that expresses each register as a function of input ports. This helps generate input sequences to change the register values. To build dependency equations, *SymbFuzz* classifies assignments into two types: *(i) static assignments*, which update a register unconditionally (*e.g.,* `assign r = a & b`), and *(ii) dynamic assignments*, which depend on control paths and design states. Listing 2 shows an example.

```
1 assign a = in1 & in2;
2 assign b = in3;
3 if (in4) begin
4     c = a + b;
5     if (c > const1 && ^in3) out = const1;
6 end
```

Listing 2: Example of dependency-equation construction.

The expression to drive `out` to `const1` (line 5) is expanded in Eqn. 1 to include only input pins.

$$\begin{aligned} & c > const1 \;\&\&\; {}^{\wedge}in3 \\ &= (a+b) \;\&\&\; {}^{\wedge}in3 \\ &= ((in1 \;\&\; in2) + in3) \;\&\&\; {}^{\wedge}in3 \end{aligned} \tag{1}$$

Once Boolean equations are derived, the SMT solver is used to solve them. For example, a register Reg_c depending on inputs $\langle in_1, in_2, \ldots, in_n \rangle$ is modeled as Boolean function $\mathcal{F}(in_1, in_2, \ldots, in_n)$. These **dependency equations** guide the mutation engine to generate input seeds that explore new states (Section 4.8).

$$state = op[2:0] \wedge nrst \tag{2}$$

In Listing 1, the control register `state` is linked to Eqn. 2. Since it is sequential, *SymbFuzz* includes path constraints in the equations. These guide the solver to produce values for each control path. The steps illustrated in Section 4.4.1 and Section 4.4.2 are performed before fuzzing is started. These steps help in measuring the coverage d and generating constraints for seed mutation e in Figure 1.

4.5 Checkpoint Setup

In hardware design simulation, performing a full reset can lead to inefficiencies during verification. Re-simulating an already-explored control flow branch with the same execution path results in redundant computations without providing additional verification value. To optimize this process and eliminate unnecessary repetitions, *SymbFuzz* avoids complete resets, preventing re-evaluation of identical functional paths.

To revisit RTL states without full resets, *SymbFuzz* uses checkpointing during CFG-based fuzzing (Step a in Figure 1). A node's branching factor is the count of its immediate successor states, *i.e.,* its outgoing edges. Control structures like `if-else` or `case` are not treated as fixed two-way branches; instead, actual successors are counted. Nodes with three or more successors are marked as high-branching. Our pilot experiments suggest that a higher threshold

(Th) reduces checkpoints but increases branch re-exploration. A lower Th adds more checkpoints but needs more memory.

SymbFuzz tracks CFG paths by recording input sequences. When a new node is encountered, the input sequence is updated and marked as the path for the current checkpoint. If a leaf node or non-transitioning state is reached, *SymbFuzz* replays the sequence to return to the checkpoint. Once all nodes from a checkpoint are visited, *SymbFuzz* moves to the previous checkpoint. This process is repeated until all nodes are covered.

For instance, starting at reset node N_{rst}, an input sequence Seq_1 transitions the DUV to node N_1, recording the path "N_{rst} to N_1." A subsequent sequence Seq_2 transitions from N_1 to N_2, marking the path. If N_2 is a checkpoint, *SymbFuzz* efficiently revisits it from N_{rst} using Seq_1 and Seq_2.

4.6 Coverage Measurement

To improve coverage assessment, we use control registers to guide traversal through the control flow graph (CFG). Each node represents a hardware state defined by current *control register* values, while edges denote transitions between states.

Node Count: Let the design expose N control registers and let n_j be the number of legal encodings of register j. The total population of distinct nodes can therefore be expressed as,

$$\#\text{ of nodes } = \prod_{j=1}^{N} n_j. \tag{3}$$

For the ALU in Listing 1, *state* is three bits wide and *OPmode* is one bit wide, giving Eqn. 4.

$$\#\text{ of nodes } = 8 \times 2 = 16. \tag{4}$$

An individual node is written as Eqn. 5 ($i_1 \in [0, 7]$ and $i_2 \in [0, 1]$).

$$C_{(i_1,i_2)} = state_{i_1} \cdot OPmode_{i_2}, \tag{5}$$

Edge Coverage: Nodes may fan out to multiple successors. A node with three or more outgoing edges is designated a *checkpoint*. At such nodes, SymbFuzz also tracks *edge coverage*: a tuple $\langle \text{edge ID}, C_{(i_1,i_2)} \rangle$ is recorded for every transition, and coverage is complete when each outgoing edge has been exercised at least once.

Run-time bookkeeping: After every simulation, *SymbFuzz* scans the dump file, maps every covered node and edge onto the CFG (Fig. 1 d), and decides whether progress has stalled. If new nodes or edges are discovered, fuzzing proceeds autonomously; otherwise, after a timeout, the seed-generation engine (Section 4.8) applies SMT-driven UVM stimuli to steer the process toward unexplored nodes or untraversed edges.

Coverage Target: In *SymbFuzz*, the coverage target is defined as the full Cartesian product of all possible outcomes of the conditions within a control statement. For instance, if a control decision depends on two conditions, such as $r1 == 0$ and $r2 == 1$, then the total number of outcome combinations–or fanouts–is $2 \times 2 = 4$. In the case of a case statement, such as the one in Listing 1 driven by the register $state$, all eight possible cases must be covered, resulting in eight distinct branches.

Some control predicates do not divide the design space into a small, well-defined set of outcomes. For example, if we consider the condition $r1 == 0$ when $r1$ is a 32-bit register. Treating it as two cases–$r1 = 0$ and $r1 \neq 0$–is not sufficient, since there might be some non-zero values of $r1$ that can still circumvent the intended logic and activate the "zero" branch. To account for these corner cases, it is necessary to fuzz for $r1 \neq 0$ values.

4.7 Path Progression Strategy

SymbFuzz utilizes UVM constraints and random bit-string generation [5] to create and mutate seeds for fuzzing the DUV. When progress stagnates, it consults the coverage log to identify unexplored nodes. Based on this log, *SymbFuzz* determines the next target node and identifies the control registers requiring modification, as indicated by `e` in Figure 1.

To select the next node, *SymbFuzz* examines the fuzzer's current position in the CFG. If no further progress is made, it suggests the input sequence is insufficient to modify control register values effectively. In this case, *SymbFuzz* evaluates additional nodes or checks for terminal points. If the search depth limit is not reached, an SMT solver is employed to analyze dependency equations, prioritizing constraints that enable exploration of the most new nodes. For instance, if $Constr_A$ unlocks three new nodes and $Constr_B$ unlocks four, *SymbFuzz* prioritizes $Constr_B$ for the UVM Sequencer.

If all sub-nodes of a checkpoint are explored, *SymbFuzz* moves to its parent in the CFG, setting it as the new "current checkpoint." The solver then generates constraints to traverse new nodes from this position. For example, in Listing 1, if *SymbFuzz* halts at line 13, it activates the constraint solver to find the optimal '*OPmode*' for path selection. *SymbFuzz* evaluates path counts from the CFG node and picks the '*OPmode*' maximizing paths, defaulting to the smallest Hamming distance in case of ties.

4.8 Seed Generation and Mutation

SymbFuzz uses UVM's random bit string generation to create initial fuzzing seeds. Operating within a defined time interval, *SymbFuzz* then evaluates coverage. The fuzzer employs the '*UVM Sequencer*' environment to generate and mutate seeds with constraints, as illustrated in Figure 1 by `f`.

SymbFuzz generates and mutates seeds for fuzzing the DUV by using UVM's constraint mechanism [5]. When the fuzzer needs assistance to explore new states, *SymbFuzz* analyzes the coverage log to identify where the fuzzer has stalled. Based on this analysis, *SymbFuzz* identifies the next target state along with the specific control registers required to reach it. Using dependency equations outlined in Section 4.4.2, *SymbFuzz*'s mutation engine employs an SMT solver to solve these equations. This process enforces constraints in UVM's random input generation, allowing *SymbFuzz* to perform directed fuzzing and guide the fuzzer to unexplored states.

```
1 constraint OPcodeCTRL {op [3] == 1};
```

Listing 3: An example of constraints (in SystemVerilog).

The UVM testbench's initial path selection imposes a constraint that sets *OPmode* to 1, targeting the DUV with *SymbFuzz* fuzzing in 8-bit operation mode (Listing 3).

4.9 Bug Identification

SymbFuzz employs property-based verification to detect security vulnerabilities using a predefined set of security properties observed via a UVM monitor. These properties are derived from the CVE and

CWE databases to reflect practical security issues. Since *SymbFuzz* targets security rather than functional correctness, a golden reference model is not sufficient. Many security bugs do not manifest as output mismatches but as violations of internal invariants. Therefore, *SymbFuzz* focuses on detecting property violations instead. For instance, in AES designs, a typical flaw involves leakage of key shares through the bus. A property (Listing 20) ensures that bus outputs do not match key shares, preserving confidentiality.

Fuzzing specializes in effectively exploring shallow paths where vulnerabilities often surface. During the exploration, the simulator checks for property violations. When a violation occurs, the simulator logs the property name, simulation timestamp, waveform, and input vectors starting from the *reset* activation. This data is compiled into *SymbFuzz*'s report, $\mathcal{R}$.

5 Evaluation and Results

Evaluation Setup:

We evaluated *SymbFuzz* on four RISC-V processors: Ibex (from OpenTitan), CVA6, Rocket-Chip, and Mor1kxx. Ibex, used in the OpenTitan SoC and featured in the *HACK@DAC'24* competition [30, 46], offers a rich set of real-world security bugs, making it an ideal benchmark for assessing *SymbFuzz* 's detection capabilities. The additional cores—CVA6 (an out-of-order RV64GC core), Rocket-Chip (a widely used in-order RV64GC core), and Mor1kxx (a lightweight OpenRISC-based design)—were selected to provide architectural diversity and varying levels of complexity. This mix enables a comprehensive evaluation of *SymbFuzz*'s scalability, cross-architecture applicability, and performance compared to existing fuzzing approaches. Across all benchmarks, *SymbFuzz* consistently detected bugs and demonstrated improved coverage.

To ensure a fair performance comparison between *SymbFuzz* and the fuzzers RFuzz, DifuzzRTL, and HWFP, each fuzzer was run four times on OpenTitan. The average coverage and resource utilization from these runs were computed, offering an unbiased foundation for the performance analysis discussed in Section 5.2.

Evaluation Platform Details: Our server uses an AMD EPYC 7513 CPU (32 cores, 64 threads, 2.6 GHz, 128 MB cache). It includes 16 DDR4-3200 RDIMMs (64 GB each) in dual-rank mode at 3200 MT/s. Simulations were performed using Xilinx Vivado [64].

Parameter Setup: The system saves simulation files every three intervals, with each interval lasting 300 clock cycles. This setting balances performance and resource usage, enabling *SymbFuzz* to capture essential data for coverage analysis. All registers start in an unknown ('X' or *don't care*) state. Verification begins by asserting the *reset* signal to bring the DUV to a known initial state. The parameter choices were guided by pilot runs on smaller designs.

Evaluation Metrics: To assess *SymbFuzz*'s efficacy, we evaluate three metrics: (1) bug detection, (2) coverage enhancement, and (3) resource utilization. We compare *SymbFuzz* against existing fuzzing techniques, defining coverage as unique combinations of control register values and input patterns.

We benchmark *SymbFuzz* against Rfuzz, DifuzzRTL, and HWFP using their publicly available implementations to evaluate performance improvements over existing hardware fuzzing frameworks. Furthermore, we compare *SymbFuzz*'s bug detection capabilities with TheHuzz, PSOFuzz, and HypFuzz on three other benchmarks–cva6, rocket-chip, and mor1kxx processors.

5.1 Detected Bugs

This section analyzes the bugs detected by *SymbFuzz*. We identified 17 bugs and mapped them to their corresponding CWE classes, excluding those flagged by linters. As linters perform static checks without execution, they fall outside our evaluation scope. We focus on 14 bugs missed by at least one existing fuzzing framework. We do not include the bugs, which could be detected by all tools, in order to emphasize the prowess of *SymbFuzz*. Notably, one bug (Bug #01) existed in the original version of OpenTitan and remained in its SoC version. This bug has been officially added to the 2025 CWE database. Table 1 lists the bugs by IP, submodule, location (column 3), line count (column 4), CWE ID (column 5), and required input vectors (column 6). Section 5.2 compares *SymbFuzz*'s performance with Rfuzz, DifuzzRTL, and HWFP.

The verification properties are rooted in the vulnerability descriptions provided by Common Vulnerabilities and Exposures (CVE) [1] and Common Weakness Enumeration (CWE) [2]. These properties capture broader patterns of vulnerabilities and are inherently generalized in nature. Register names and design-specific details are adapted to align the properties with the DUV, ensuring applicability without compromising their underlying generality.

```
always_ff @(posedge clk_i or negedge rst_ni) begin
  if (!rst_ni) begin
    q <= RESVAL; end
  end else if (wr_en) begin
    q <= wr_data; end end
...
always_comb begin
wr_err = (reg_we & ((addr_hit[0] & (|(SCMI_PERMIT[0]
      & !reg_be))) | ...
...
```

Listing 4: Write request in the OpenTitan Mailbox.

Bug 1: A bug has been found in the Mailbox implementation of the OpenTitan SoC. Here, the protocol fails to notify the host when several write attempts to reserved addresses have been made. Although the data written to these addresses is correctly discarded, the host does not receive any feedback indicating the attempt, as shown in Listing 4. This flaw allows an attacker to repeatedly write to reserved addresses without any error or warning signals. Note that this IP was generated using OpenTitan's *reggen* tool and was not part of the *HACK@DAC'24* competition.

```
property writeReqCheck;
  addr_hit[0] == 1 && !(|SCMI_PERMIT[0]);
endproperty
```

Listing 5: Security Property for write request.

SymbFuzz detects this bug using the property in Listing 5. Despite being triggered around 370 times during execution, the module failed to raise any error or warning signals. The bug has been acknowledged as a new entry in CWE 2025 database.

```
...
always_comb begin : p_fsm
 unique case (fsm_state_q)
 ...
 endcase
...
```

Table 1: Details for the Detected Bugs in the Benchmark SoC.

Bug No.	Bug Description	Sub modules involved		CWE Number	# of input vectors
		Sub-Module	Total LoC		
01.	No feedback for data error in the Mailbox.	*scmi_reg_top*	735	*New Entry*	6.47×10^6
02.	Undefined default state.	*lc_ctrl_fsm*	735	CWE-1199	1.64×10^7
03.	Enables the production function before testing in unlocked states is completed.	*lc_ctrl_signal _decoder*	384	CWE-1245	6.84×10^6
04.	Key shares are leaked into the bus using key share offset.	*aes_reg_top*	1880	CWE-1342	6.97×10^6
05.	Not clearing pseudo-random data registers.	*aes_core* and *aes_cypher_core*	1867	CWE-459	8.24×10^5
06.	AES masking operation with pseudo-random number is always off.	aes_prng _masking	234	CWE-1300	7.43×10^5
07.	Blanking operation in OTBN is disabled.	otbn_mac _bignum	240	CWE-325	8.32×10^6
08.	ROM control skips checking state.	*rom_ctrl_fsm*	312	CWE-1269	6.82×10^6
09.	Incomplete clear process in Power manager.	*pwr_mgr_fsm*	542	CWE-1304	4.82×10^6
10.	Not checking ROM integrity check flag.	*pwr_mgr_fsm*	542	CWE-1304	4.82×10^6
11.	The system cannot turn off the parity check.	*uart_rx*	137	CWE-1257	6.82×10^6
12.	Reseed Interval cannot be checked via the checker logic.	*csrng_reg_top*	2042	CWE-1257	1.82×10^7
13.	System Reset Controller has the wrong value for the error flag.	sysrst_ctrl _reg_top	7218	CWE-1320	1.56×10^7
14.	Data flush upon receipt of the *enable* signal.	otp_ctrl_dai	860	CWE-1266	8.14×10^6

```
7 end
```

Listing 6: Undefined default state.

Bug 2: *SymbFuzz* detected a missing default state in the 16-state FSM of the Life Cycle Controller (LC_CTRL), which relies on the 16-bit register *fsm_state_q*, as shown in Listing 6.

To detect this bug, the property used is shown in Listing 7. Here, the property defines that the value of register *fsm_state_q* is required to be one of the defined values.

```
property read_key;
   !$isunknown(fsm_state_q);
endproperty
```

Listing 7: Security Property for FSM.

Bug 3: The LC_CTRL IP also contained this bug. Due to this bug. production state functions execute in the unlocked state, as shown in Listing 8 (lines 4-6), where LC_CTRL initiates production functions before completing all tests in the final unlocked state.

```
1 ...
2 LcStTestUnlocked0,
3 ...
4 LcStProd: begin
5   ...
6 end
7 LcStTestUnlocked7: begin
8 ...
```

Listing 8: Production functions before unlocked states.

This bug was identified by *SymbFuzz* using the property in Listing 9, which requires that the Non-volatile Memory (NVM) debug state must be disabled unless the LC_CTRL is in the Return Material Authorization (RMA) state.

```
property read_key;
   lc_state_i != LcStRma |-> !lc_nvm_debug_en;
endproperty
```

Listing 9: Security Property for function checking.

Bug 4: The AES module leaks key values via bus output (Listing 10) when key_share offset flag is set to 'HIGH.' Instead of returning 0, the module exposes *key shares*, creating a vulnerability.

```
always_comb begin
 reg_rdata_next = '0;
 unique case (1'b1)
 ...
 addr_hit[1]: begin
  reg_rdata_next[31:0]=reg2hw.key_share0[0].q;
 end
 ...
 endcase
...
end
```

Listing 10: Key leak Bug in AES.

```
property read_key;
 tl_o.d_data != $isinside(reg2hw.keyshare.q);
endproperty
```

Listing 11: Security Property for read keyshare value.

To detect this bug, we have used the property shown in Listing 11. This property defines that the output data to the bus can not be equal to one of the *key shares*.

```
always_comb begin: data_in_reg_clear
  for (int i = 0; i < NumRegsData; i++) begin
    hw2reg.data_in[i].de = data_in_we;
  end end
```

Listing 12: Wiping data leaks secret information.

Bug 5: *SymbFuzz* identified a bug in both the AES core and the cipher core within the AES module. As illustrated in Listing 12, this bug causes the AES module to erroneously replace residual data in the registers with input data instead of pseudo-random values. This flaw introduces a security vulnerability, enabling the extraction of input data through the use of the wipe command.

```
property Data_clear;
  for (int i = 0; i < NumRegsData; i++) begin
    hw2reg.data_in[i].de != $past(hw2reg.data_in[i].
     de);
  end endproperty
```

Listing 13: Security property for wipe data check.

The bug was effectively identified using the property in Listing 13, which asserts that the data register must not retain its previous value after a 'clear' command.

```
assign data_o =
(SecAllowForcingMasks && force_masks_i) ? '0 :
phase_q ?                               '0 : '0;
```

Listing 14: Always-off masking with PRNG.

Bug 6: This bug also appears in the AES module, where masking with pseudo-random numbers is always disabled by unconditionally setting it to 0, as shown in Listing 15.

```
assign data_o =
(SecAllowForcingMasks && force_masks_i) ? '0 :
phase_q ?                               '0 : '0;
```

Listing 15: Always-off masking with PRNG.

The bug was identified using the property in Listing 16, which states that when *phase_q* is HIGH, *data_o* must equal *perm[0]* concatenated with *perm[NumChunks-1:1]*.

```
property Data_clear;
  phase_q |-> data_o == {perm[0], perm[NumChunks
    -1:1]}; endproperty
```

Listing 16: Security property for masking condition check.

Bug 7: A bug was discovered in the OpenTitan Big Number (OTBN) accelerator that keeps the blanking operands continuously active, rendering them transparent. Consequently, the blanking operation is effectively disabled, causing the OTBN to emit a detectable power trace during its operation, as illustrated in Listing 17.

```
prim_blanker #(.Width(WLEN)) u_operand_b_blanker (
  .in_i (operation_i.operand_b),
  .en_i (1'b1),
  .out_o(operand_b_blanked)
);
```

Listing 17: The blanking operation in the OTBN is disabled.

This bug was detected using the property shown in Listing 18. This property ensures that the blanking operation must be active in cases where the OTBN processor is not engaged in MAC operations or utilizing the ALU.

```
property blanking;
  !mac_predec_bignum |-> !u_otbn_mac_bignum.
     u_operand_b_blanker.out_o;
endproperty
```

Listing 18: Security property for blanking operation.

Bug 8: The ROM control IP's FSM contains a bug that skips the *Check* state, directly transitioning from task completion to the *Done* state, bypassing critical verification (as can seen in Listing 19).

```
KmacAhead: begin
 if (counter_done) state_d = Done; end
```

Listing 19: Checking state skipping Bug in ROM control.

The security property in Listing 20 identifies this bug. It requires the past value of to be checking when *state_d* equals Done.

```
property read_key;
 @(state_d)
 state_d==done|->$past(state_d)==checking;
endproperty
```

Listing 20: Security property for FSM operation check.

Bug 9: A bug in the *Power Manager* IP of the SoC prevents proper data clearing for always-on IP cores. Instead of waiting for *reset_reqs_i[ResetMainPwrIdx]*, *clr_slow_req_o* is set to HIGH, prematurely halting the clearing process, as shown in Listing 21.

```
FastPwrStateResetWait: begin
 rst_lc_req_d = {PowerDomains{1'b1}};
 clr_slow_req_o = 1'b1;
```

Listing 21: Incomplete "clear" Bug in Power Manager.

```
property read_key;
@(state_q)
state_q==FastPwrStateResetWait |->
clr_slow_req_o==reset_reqs_i[ResetMainPwrIdx];
endproperty
```

Listing 22: Security Property for Power Mgr. reset wait time.

To identify this bug, we used the property in Listing 22, which asserts that in the *FastPwrStateResetWait* state, the clear request must equal *reset_reqs_i[ResetMainPwrIdx]*.

```
FastPwrStateRomCheckGood: begin
  state_d = FastPwrStateActive; end
```

Listing 23: Not checking ROM integrity check flag.

Bug 10: This bug was also detected in the *Power Manager* IP of the SoC. Due to this bug, the power manager does not check the integrity of the ROM data before transitioning to the next step, as seen in Listing 23. Here, the power manager is supposed to check the integrity flag "*rom_intg_chk_good*".

```
property intg_check;
  !(state_q == FastPwrStateRomCheckGood) || !
     mubi4_test_true_strict(rom_intg_chk_good) |->
     state_d != FastPwrStateActive; endproperty
```

Listing 24: Security property for checking state transition.

Table 2: Comparison of bug detection by the fuzzers.

Bug No.	*SymbFuzz*	RFuzz	DifuzzRTL	HWFP
01.	✓	✗	✗	✗
02.	✓	✗	✓	✓
03.	✓	✗	✓	✓
04.	✓	✓	✗	✗
05.	✓	✗	✗	✗
06.	✓	✗	✗	✗
07.	✓	✗	✓	✓
08.	✓	✗	✓	✓
09.	✓	✗	✗	✗
10.	✓	✗	✓	✓
11.	✓	✗	✓	✗
12.	✓	✓	✗	✗
13.	✓	✗	✓	✗
14.	✓	✗	✓	✓

This bug was detected by the property shown in Listing 24. This property mandates that if the *Power Manager* is in fast power state checking mode, it must check the integrity of the ROM before activating the fast power state.

Bug 11: *SymbFuzz* detected a bug in the *UART* module where parity is checked even when disabled by the host, causing false error flags. In Listing 25, the error flag is raised if the received data is valid and the XOR of its bits with the parity odd bit is HIGH.

```
assign rx_parity_err = rx_valid_q & (^{sreg_q[9:1],
    parity_odd});
```

Listing 25: UART always parity check on.

Using the property elaborated in Listing 26, *SymbFuzz* detected this bug in the benchmark. The property asserts that when the *rx_parity_err* is raised HIGH, the *parity_enable* should also be HIGH.

```
property read_key;
 @(rx_parity_err)
 rx_parity_err |-> parity_enable;
endproperty
```

Listing 26: Security property for UART error Flags.

```
always_comb begin
  reg_we_check = '0;
  ...
  reg_we_check[7] = 1'b0;
  ...
  reg_we_check[16] = 1'b0;
end
```

Listing 27: 'Reseed interval enable' check Bug.

Bug 12: *SymbFuzz* identified a bug in the SoC's *Cryptographically Secure Random Number Generator* (CSRNG) module. The checkers fail to access the eighth bit of *reg_check*, designated as the "reseed interval enable" flag, preventing proper verification of the reseeding function (Listing 27).

To detect this bug, the security property illustrated in Listing 28 was used. This property dictates that the value of *reg_we_check* should contain a reseed interval enable flag, which can be later checked using the checker logic.

```
property read_key;
 @(reg_we_check)
 reg_we_check[7] == reseed_interval_we;
endproperty
```

Listing 28: Security property for 'reseed interval enable'.

Bug 13: This bug was identified in the "*System Reset Controller*" as shown in Listing 29. It prevents the write error flag from being raised, which is essential for enabling register writing. The parameter should be set to $4'b0001$ to yield a HIGH value when ORed. However, it is incorrectly set to $4'b0000$ (Line 3 in Listing 29), resulting in the error flag not being raised (Line 5 in Listing 29).

```
parameter logic [3:0] SYSRST_CTRL_PERMIT [43] = '{
  ...
  4'b 0000, // index[ 4] SYSRST_CTRL_REGWEN
  ...}
wr_err = (reg_we &
           (...
             (addr_hit[ 4] & (|(SYSRST_CTRL_PERMIT[
    4] & !reg_be))) |
```

Listing 29: Defining a wrong value for parameter.

```
property error_flag_check;
  reg_we || addr_hit[4] || (|(SYSRST_CTRL_PERMIT[ 4]
      & !reg_be) |-> wr_err; endproperty
```

Listing 30: Security property for error flag checking.

This bug was detected using the property in Listing 30, which asserts the error flag when any condition is TRUE.

Bug 14: *SymbFuzz* detected this bug in the "*One Time Programmable*" (OTP) memory controller IP that causes it to wipe data upon receiving an enable signal, as shown in Listing 31.

```
if (data_en) begin
  data_q <= '0;
```

Listing 31: Data cleared when eneable signal is received.

The bug was identified by rigorously applying the verification property outlined in Listing 32, which systematically checks and ensures the accuracy of the output data.

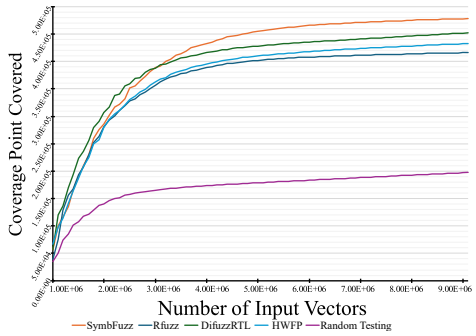
```
property DataCorrectChck;
 data_en && (data_sel == ScrmblData) |-> (data_q ==
     scrmbl_data_i)
endproperty
```

Listing 32: Security property for checking data correctness

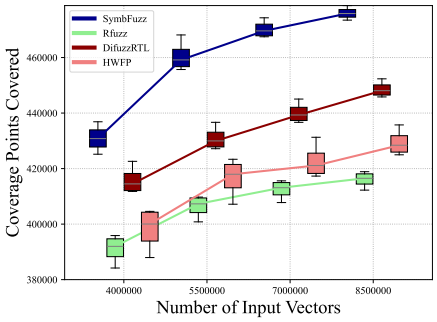
5.2 Performance Analysis

This section compares *SymbFuzz*'s bug detection capabilities with prior hardware fuzzers–RFuzz [35], DifuzzRTL [29], and HWFP [61] –using publicly available source codes. Table 2 lists the bugs from Section 5.1, indicating which fuzzers detected each case. We only discuss bugs missed by one or more fuzzers, including *SymbFuzz*.

As shown in Table 2, *SymbFuzz* detects all the bugs, whereas RFuzz fails to identify 12, DifuzzRTL misses 6, and HWFP overlooks 8. Furthermore, *SymbFuzz* discovered a previously undetected bug in the non-buggy version of the OpenTitan IP, a flaw that had been missed in earlier studies (as elaborated in Bug #1 in Section 5.1).



(a) Coverage comparison with existing fuzzing techniques.



(b) Variance of coverage while fuzzing the benchmark.

Figure 4: Coverage comparison and variance profile.

Table 1 highlights the advantages of assertion-based verification over GRM-based verification for security-critical applications. In Bug #4, although the DUV output aligns with the GRM output when reading a register, this alignment inadvertently exposes key shares, thereby bypassing error detection mechanisms that rely solely on GRM-based verification. This issue represents a security vulnerability, as the bus output should not reveal sensitive information, such as the key or key shares. *SymbFuzz* plays a crucial role here by effectively detecting these discrepancies. Notably, only *SymbFuzz* and RFuzz are capable of identifying hardware-specific vulnerabilities, as demonstrated in Bug #4.

Furthermore, Bug #06, which disables a critical masking operation, is undetected by Verilator and AFL-based fuzzers like (Rfuzz, DifuzzRTL, and HWFP). RFuzz also fails to detect Bug #08 due to its coverage approach, which relies on multiplexers and lacks the coverage precision necessary for detection. Moreover, none of the fuzzers identify Bug #11, where parity checks are mostly enabled.

SymbFuzz exhibits a balanced resource profile, using approximately 4% more memory than DifuzzRTL and 7% more than RFuzz, yet 7% less than HWFP. For CPU efficiency, *SymbFuzz* performs comparably to RFuzz but is 33% and 54% more efficient than DifuzzRTL and HWFP, respectively, on identical hardware. While RFuzz is the most resource-efficient, its lower detection performance, as shown in Table 2, reveals a trade-off between efficiency and detection accuracy. Thus, *SymbFuzz* provides a balanced approach, achieving both effective bug detection and efficient resource usage across memory and CPU metrics.

Observation: *SymbFuzz* outperforms existing tools by detecting security vulnerabilities due to not relying on GRM-based differential testing, which is primarily suited for functional verification.

5.3 Coverage Analysis

In this study, we utilized control registers to assess coverage metrics comprehensively. Each branch within the control path is regarded as a distinct coverage point for *SymbFuzz*. When multiple control path combinations occur simultaneously, each unique combination of selected paths is also considered a coverage point. To ensure a fair comparison with *SymbFuzz*, we used the same coverage points as prior works (*i.e.,* Rfuzz [35], DifuzzRTL [29], HWFP [61]).

Figure 4a compares the coverage obtained by *SymbFuzz* compared to existing approaches, demonstrating its superiority. With 9.1×10^6 input vectors, *SymbFuzz* achieved approximately 6% more coverage than DifuzzRTL [29], about 15% more than Rfuzz [35], and around 10% more than HWFP [61]. These results highlight *SymbFuzz*'s effectiveness in outperforming existing fuzzers in terms of coverage, enhancing its potential to uncover latent bugs.

Moreover, all evaluated fuzzing approaches, including *SymbFuzz*, significantly outperform UVM random testing in terms of coverage gain. Specifically, *SymbFuzz* achieves a 6.8× speed-up in reaching equivalent functional coverage. As shown in Figure 4a, UVM random testing saturates at around 2.7×10^6 input vectors, a point reached by *SymbFuzz* and other fuzzers with only 1.2×10^6 inputs. This performance gap stems from a core limitation of UVM random testing: it generates inputs without any guidance, relying on the randomness to discover new states, resulting in a slow and inefficient exploration. Furthermore, even after 72 hours, UVM testing failed to achieve full coverage, saturating between 88% and 94% across ten runs. In contrast, *SymbFuzz* uses symbolic execution to actively target unexplored states, enabling faster, more directed coverage gains with reduced computational cost. These results underscore *SymbFuzz*'s practical effectiveness and scalability in modern hardware security verification workflows.

Figure 4b illustrates the variance in coverage for *SymbFuzz* across a range of input vectors, specifically from 4×10^6 to 8.5×10^6. This range was chosen since, at the very start of the fuzzing process, the variance is extremely high due to the randomness of initial coverage exploration, which obscures the effective variance trends. Over time, as fuzzing progresses and approaches stagnation, the variance significantly decreases, reflecting a convergence in the paths explored by each fuzzer. Consequently, this selected range offers a more accurate view of how randomness influences variance without the skew from initial, high-variance fluctuations. Within this range, *SymbFuzz* consistently achieves higher coverage gains than Rfuzz, DifuzzRTL, and HWFP. Moreover, since no predefined seed was used in any of the runs, each fuzzer independently explored diverse paths, further highlighting the impact of randomness.

Observation: *SymbFuzz* achieved up to 15% higher coverage than DifuzzRTL, Rfuzz, and HWFP, highlighting its potential for better exploration and bug detection in hardware security verification.

Table 3: Benchmark Details.

Benchmark	Code Size (LoC)	CFG Size		Dependency Eqn. Generated	Latency (minutes)	Constraints Generated
		Nodes	Edges			
OpenTitan	1.5million	1424	4863	300∼350	≈ 250	≈ 600
CVA6	201086	576	1728	100∼120	≈ 60	≈ 200
Rocket-Chip	38500 (Scala)	617	1832	100∼120	≈ 60	≈ 200
Mor1kx	201832	589	1688	100∼120	≈ 60	≈ 200

5.4 Benchmarking with Additional Research

In this section, we show that *SymbFuzz* effectively identifies bugs that were previously detected by other fuzzing methods, including TheHuzz [31], PSOFuzz [14], and HypFuzz [15]. Since the source codes for these tools are not publicly available, we were unable to evaluate them directly on the bugs listed in Table 1. To demonstrate *SymbFuzz*'s bug detection capability, we identified bugs found by these tools in three processors—*CVA6*, *Rocket Core*, and *mor1kx*.

We adopt the enumeration from HypFuzz [15] to categorize previously reported bugs for clarity and consistency. HypFuzz and *SymbFuzz* detect Bugs V1–V3, while all fuzzers identify Bugs V4–V11. Bug V1 poses a critical reliability risk, as no exception is raised on invalid memory access, potentially causing undefined behavior. Bug V2 stems from the incorrect decoding of multiplication instructions, leading to execution errors. Bug V3 involves access to unallocated CSRs, returning undefined values instead of errors, thereby compromising system state integrity.

Observation: *SymbFuzz* was able to detect all bugs that were reported in other existing SoC fuzzing approaches (TheHuzz, PSOFuzz, HypFuzz).

5.5 Discussion

5.5.1 ***Design Rationale.*** Key design choices were made to ensure efficient, scalable, and security-focused verification:
(1) Security Beyond Golden Models: Functional correctness does not ensure security. As seen in Table 1 (Bug #4), secrets can leak even when outputs match the golden reference. SystemVerilog assertions capture such violations at runtime (e.g., Bug #1, #4, #6), revealing hidden flaws missed by golden-model checks.
(2) Hybrid Fuzzing for Coverage: Fuzzing is fast but often shallow. Symbolic execution is deeper but slower. Our hybrid approach uses fuzzing for general exploration and applies symbolic reasoning only at hard-to-reach branches, improving both speed and depth.
(3) Selective Symbolic Analysis in SymbFuzz: Concolic testing struggles with path explosion. *SymbFuzz* limits symbolic reasoning to critical control registers and uses checkpoints to manage state. This enables efficient deep exploration with low solver overhead.

5.5.2 ***Scalability.*** *SymbFuzz* scales efficiently by combining SMT-guided state pruning with checkpoint-based replay. This avoids full flip-flop tracking and instead grows with control-register tuples. On cores like Ibex, Rocket, and CVA6 (≈110k flip-flops), *SymbFuzz* explored 1.1–2.0×10^4 edge–state pairs, doubled functional coverage, and converged 6–7× faster than random UVM fuzzing. Checkpoint replays finish in microseconds, avoiding full reboots. Table 3 shows code size, CFG size, equations, latency, and constraints. The results confirm that *SymbFuzz* scales across diverse runs.

5.5.3 ***Extending Applicability to Detect Manufacturing Faults.*** *SymbFuzz* was developed for detecting security flaws, but it can also identify faults from manufacturing defects. This requires using a golden reference model instead of assertions and comparing outputs during fuzzing. Its core features–checkpointing and dependency-guided CFG traversal–remain effective, showing its adaptability for both security and reliability verification. In the future, we intend to analyze the performance of *SymbFuzz* on manufacturing defects on myriad processor architectures.

5.5.4 ***False Positives and False Negatives.*** Within the tested designs and time limits, *SymbFuzz* achieved coverage comparable to or better than its peers. However, RTL designs have large corner-case spaces, so defects requiring specialized analysis may still exist, though none appeared in our current experiments. In the future, we intend to augment *SymbFuzz* by analyzing emerging processor architectures, including Deep Learning hardware.

5.5.5 ***Architectural Influence on Bug Profiles.*** Our evaluation in Section 5 shows that all bugs in the in-order Ibex and Rocket benchmarks involve either instruction sequencing or CSR handling. These cores execute instructions strictly in order and lack speculative features such as reorder buffers or branch-recovery logic. Thus, our explored architectural bugs are limited to stall-or-flush handshakes and fast CSR accesses and do not involve any speculation-related vulnerabilities.

6 Conclusion

In this paper, we present *SymbFuzz*, a novel hybrid hardware fuzzing framework developed with UVM (as mentioned in Section 4), designed to integrate seamlessly into commercial hardware design flows. *SymbFuzz* leverages coverage-guided fuzzing with symbolic execution to efficiently explore control flow graph nodes, enhancing bug detection and test coverage. When evaluated on a commercial-grade SoC, *SymbFuzz* identified 14 defects, six of which were previously undetectable by conventional hardware fuzzing methods. Notably, *SymbFuzz* uncovered a previously unknown vulnerability in an IP from the OpenTitan SoC, which has since been added to the CWE 2025 database. Furthermore, *SymbFuzz* increased test coverage by 2×10^4 coverage points without requiring any changes to the hardware execution process. *SymbFuzz* was successful in detecting vulnerabilities identified by prior hardware fuzzing approaches. These results demonstrate that *SymbFuzz* outperforms existing approaches in terms of bug detection and coverage, making it a promising candidate for integration into the SDL. *SymbFuzz* demonstrates its potential as a valuable framework for commercial SoC design flow, streamlining the security verification processes and enhancing SoC robustness against vulnerabilities.

References

[1] [n. d.]. CVE - Common Vulnerabilities and Exposures. https://cve.mitre.org/ Accessed: 06/28/2024.

[2] [n. d.]. cwe - common weakness enumeration. https://cwe.mitre.org/ Accessed: 06/28/2024.

[3] [n. d.]. LowRISC Ariane. https://github.com/lowRISC/ariane Accessed: 05/19/2024.

[4] 2014. American Fuzzy Lop (afl) Fuzzer. https://lcamtuf.coredump.cx/afl/technical_details.txt

[5] SIEMENS Verification Academy. 2024. Universal Verification Methodology. Online. https://verificationacademy.com/topics/uvm-universal-verification-methodology/

[6] Armaiti Ardeshiricham, Wei Hu, and Ryan Kastner. 2017. Clepsydra: Modeling timing flows in hardware designs. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 147–154.

[7] Averant Solidify [n. d.]. Averant Solidify. https://averant.com:8443/products-solidify.html Accessed: 04/03/2024.

[8] Wael Badawy and Graham A Julien. 2012. *System-on-chip for Real-time Applications*. Vol. 711. Springer Science & Business Media.

[9] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. 2018. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)* 51, 3 (2018), 1–39.

[10] Maximilian Beckmann and Jan Steffan. 2023. Coverage-Guided Fuzzing of Embedded Systems Leveraging Hardware Tracing. In *Computer Security. ESORICS 2022 International Workshops*. Springer International Publishing, Cham, 362–378.

[11] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 1032–1043.

[12] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 209–224.

[13] Cadence Verification Tools [n. d.]. Cadence Verification Tools. https://www.cadence.com/en_US/home/tools/system-design-and-verification.html Accessed: 04/03/2024.

[14] 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.

[15] 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.

[16] Peng Chen and Hao Chen. 2018. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 711–725.

[17] Wen Chen, Sandip Ray, Jayanta Bhadra, Magdy Abadir, and Li-C Wang. 2017. Challenges and trends in modern SoC design verification. *IEEE Design & Test* 34, 5 (2017), 7–22.

[18] CISCO Trustworthy Solutions [n. d.]. Overview: Cisco Public 1, Cisco Secure Development Lifecycle Securing Cisco Technology. https://www.cisco.com/c/dam/en_us/about/doing_business/trust-center/docs/cisco-secure-development-lifecycle.pdf Accessed: 04/03/2024.

[19] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2001. Progress on the state explosion problem in model checking. *Informatics: 10 Years Back, 10 Years Ahead* (2001), 176–194.

[20] ClusterFuzz [n. d.]. https://google.github.io/clusterfuzz/. Accessed: 04/03/2024.

[21] R.D. Craig and S.P. Jaskiel. 2002. *Systematic Software Testing*. Artech House. https://books.google.com/books?id=2_gbZYZcZXgC

[22] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[23] LaShanda Dukes, Xiaohong Yuan, and Francis Akowuah. 2013. A case study on web application security testing with tools and manual testing. In *2013 Proceedings of IEEE Southeastcon*. IEEE, 1–6.

[24] Farimah Farahmandi, Yuanwen Huang, and Prabhat Mishra. 2019. *System-on-Chip Security*. Springer International Publishing. https://doi.org/10.1007/978-3-030-30596-3_1

[25] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ : Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association. https://www.usenix.org/conference/woot20/presentation/fioraldi

[26] Samuel Groß. 2018. Fuzzil: Coverage guided fuzzing for javascript engines. *Department of Informatics, Karlsruhe Institute of Technology* (2018).

[27] Sunny L He, Natalie H Roe, Evan Wood, Noel M Nachtigal, and Jovana Helms. 2015. *Model of the Product Development Lifecycle*. Technical Report. Sandia National Lab.(SNL-NM), Albuquerque, NM (United States).

[28] Matthew Hicks, Cynthia Sturton, Samuel T King, and Jonathan M Smith. 2015. Specs: A lightweight runtime mechanism for protecting software from security-critical processor bugs. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. 517–529.

[29] 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.

[30] Ibex RISC-V [n. d.]. Ibex RISC-V Core. https://github.com/lowRISC/ibex?tab=readme-ov-file Accessed: 08/15/2024.

[31] 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)*. USENIX Association, Boston, MA, 3219–3236. https://www.usenix.org/conference/usenixsecurity22/presentation/kande

[32] James C King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (1976), 385–394.

[33] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. 2020. Spectre attacks: Exploiting speculative execution. *Commun. ACM* 63, 7 (2020), 93–101.

[34] Dusko Koncaliev. [n. d.]. Bugs in the Intel Microprocessors. https://www.cs.earlham.edu/~dusko/cs63/fdiv.html

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

[36] Xun Li, Vineeth Kashyap, Jason K Oberg, Mohit Tiwari, Vasanth Ram Rajarathinam, Ryan Kastner, Timothy Sherwood, Ben Hardekopf, and Frederic T Chong. 2014. Sapper: A language for hardware-level security policy enforcement. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*. 97–112.

[37] Xun Li, Mohit Tiwari, Jason K Oberg, Vineeth Kashyap, Frederic T Chong, Timothy Sherwood, and Ben Hardekopf. 2011. Caisson: a hardware description language for secure information flow. *ACM Sigplan Notices* 46, 6 (2011), 109–120.

[38] Dian-Lun Lin, Yanqing Zhang, Haoxing Ren, Brucek Khailany, Shih-Hsin Wang, and Tsung-Wei Huang. 2023. Genfuzz: Gpu-accelerated hardware fuzzing using genetic algorithm with multiple inputs. In *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[39] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, et al. 2020. Meltdown: Reading kernel memory from user space. *Commun. ACM* 63, 6 (2020), 46–56.

[40] Rupak Majumdar and Koushik Sen. 2007. Hybrid concolic testing. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 416–426.

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

[42] Xingyu Meng, Shamik Kundu, Arun K Kanuparthi, and Kanad Basu. 2021. Rtl-contest: Concolic testing on rtl for detecting security vulnerabilities. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 3 (2021), 466–477.

[43] Zalewski Michal. [n. d.]. https://lcamtuf.coredump.cx/afl/. Accessed: 04/03/2024.

[44] A Molina and Oswaldo Cadenas. 2007. Functional verification: Approaches and challenges. *Latin American applied research* 37, 1 (2007), 65–69.

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

[46] OpenTitan [n. d.]. OpenTitan | Documentation. https://opentitan.org/book/doc/introduction.html Accessed: 05/19/2024.

[47] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware evolutionary fuzzing.. In *NDSS*, Vol. 17. 1–14.

[48] Kaki Ryan, Matthew Gregoire, and Cynthia Sturton. 2023. Augmented Symbolic Execution for Information Flow in Hardware Designs. *arXiv preprint arXiv:2307.11884* (2023).

[49] Kaki Ryan, Matthew Gregoire, and Cynthia Sturton. 2023. SEIF: Augmented Symbolic Execution for Information Flow in Hardware Designs. In *Proceedings of the 12th International Workshop on Hardware and Architectural Support for Security and Privacy*. 1–9.

[50] Smruti R Sarangi, Abhishek Tiwari, and Josep Torrellas. 2006. Phoenix: Detecting and recovering from permanent processor design bugs with programmable hardware. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. IEEE, 26–37.

[51] Kostya Serebryany. 2017. OSS-Fuzz - Google's continuous fuzzing service for open source software. In *Proceedings of the 26th USENIX Security Symposium*. USENIX Association, Vancouver, BC.

[52] Siemens Modelsim [n. d.]. Siemens Modelsim. https://eda.sw.siemens.com/en-US/ic/modelsim/ Accessed: 04/03/2024.
[53] Wilson Snyder. 2004. Verilator and systemperl. In *North American SystemC Users' Group, Design Automation Conference*.
[54] 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.
[55] Michael Sutton, Adam Greene, and Pedram Amini. 2007. *Fuzzing: brute force vulnerability discovery*. Pearson Education.
[56] Synopsys Verification Family [n. d.]. Synopsys Verification Tools. https://www.synopsys.com/verification.html Accessed: 04/03/2024.
[57] Syzkaller [n. d.]. https://github.com/google/syzkaller. Accessed: 04/03/2024.
[58] Shinya Takamaeda-Yamazaki. 2015. Pyverilog: A Python-Based Hardware Design Processing Toolkit for Verilog HDL. In *Applied Reconfigurable Computing (Lecture Notes in Computer Science, Vol. 9040)*. Springer International Publishing, 451–460. https://doi.org/10.1007/978-3-319-16214-0_42
[59] Ari Takanen, Jared D Demott, Charles Miller, and Atte Kettunen. 2018. *Fuzzing for software security testing and quality assurance*. Artech House.
[60] Mohit Tiwari, Jason K Oberg, Xun Li, Jonathan Valamehr, Timothy Levin, Ben Hardekopf, Ryan Kastner, Frederic T Chong, and Timothy Sherwood. 2011. Crafting a usable microkernel, processor, and I/O system with strict and provable information flow security. *ACM SIGARCH Computer Architecture News* 39, 3 (2011), 189–200.
[61] 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)*. USENIX Association, Boston, MA, 3237–3254. https://www.usenix.org/conference/usenixsecurity22/presentation/trippel
[62] Srivatsa Vasudevan. 2006. An Introduction to IC Verification. *Effective Functional Verification: Principles and Processes* (2006), 3–12.
[63] Ilya Wagner and Valeria Bertacco. 2007. Engineering trust with semantic guardians. In *2007 Design, Automation & Test in Europe Conference & Exhibition*. IEEE, 1–6.
[64] Xilinx Vivado User Guide [n. d.]. Xilinx Vivado. https://www.xilinx.com/support/documents/sw_manuals/xilinx2022_2/ug904-vivado-implementation.pdf Accessed: 05/19/2024.
[65] Wen Xu, Soyeon Park, and Taesoo Kim. 2020. Freedom: Engineering a state-of-the-art dom fuzzer. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 971–986.
[66] YosysHQ. 2024. SymbiYosys: A Front-End Driver for Yosys-Based Formal Verification. https://github.com/YosysHQ/sby.
[67] 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.
[68] Danfeng Zhang, Yao Wang, G Edward Suh, and Andrew C Myers. 2015. A hardware design language for timing-sensitive information-flow security. *Acm Sigplan Notices* 50, 4 (2015), 503–516.
[69] Lei Zhao, Yue Duan, and Jifeng XUAN. 2019. Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing. Network and Distributed System Security Symposium (NDSS).
[70] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. 2022. Fuzzing: A Survey for Roadmap. *ACM Comput. Surv.* 54, 11s, Article 230 (sep 2022), 36 pages. https://doi.org/10.1145/3512345