Sylvia: Countering the Path Explosion Problem in the Symbolic Execution of Hardware Designs

Kaki Ryan
University of North Carolina
Chapel Hill, NC, USA
kakiryan@cs.unc.edu

Cynthia Sturton
University of North Carolina
Chapel Hill, NC, USA
csturton@cs.unc.edu

***Abstract*—Symbolic execution is a powerful verification tool for hardware designs, in particular for security validation. However, symbolic execution suffers from the path explosion problem in which the number of paths to explore grows exponentially with the number of branches in the design. We introduce a new approach, *piecewise composition*, which leverages the modular structure of hardware to transfer the work of path exploration to SMT solvers. Piecewise composition works by recognizing that independent parts of a design can each be explored once, and the exploration reused. A hardware design with N independent `always` blocks and at most b branch points per block will require exploration of $O(2^bN)$ paths in a single clock cycle with our approach compared to $O(2^{bN})$ paths using traditional symbolic execution.**

We present *Sylvia*, a symbolic execution engine implementing piecewise composition. The engine operates directly over RTL without requiring translation to a netlist or software simulation. We evaluate our tool on multiple open-source SoC and CPU designs, including the OR1200 and PULPissimo RISC-V SoC. The piecewise composition technique reduces the number of paths explored by an order of magnitude and reduces the runtime by 97% compared to our baseline. Using 84 properties from the security literature we find assertion violations in open-source designs that traditional model checking and formal verification tools do not find.

***Index Terms*—symbolic execution, verilog, register transfer level, verification, formal methods, hardware security**

I. Introduction

The verification of hardware designs is a key activity for ensuring the correctness and security of a design early in the hardware lifecycle. Current best practice includes assertion-based verification (ABV) [17], which has simulation-based testing as the underlying means of verification, and formal verification techniques, an umbrella term encompassing many techniques with the goal of proving a given property of a design. One technique that has gained recent attention, especially in security verification applications, is symbolic execution [8], [40], [44], [48]. In addition to finding property violations, symbolic execution has been used to verify information-flow properties [25] or to find hardware trojans [44].

Symbolic execution generalizes testing by replacing input values with symbols, where each symbol represents the set of possible values of the input parameter. A symbolic execution engine drives symbolic execution using the semantics of the program's language, but updated to include symbols. As execution proceeds the symbols are used in place of literal values. When a branch point, or control flow statement, is reached (e.g., an `if` statement), both possible branches are explored separately. The result of symbolically executing a design for one clock cycle is a tree of paths, each one associated with a unique *path condition* that describes the conditions satisfied by branches taken along the path. Symbolic execution is often used to find assertion violations. If any path is found to violate a given assertion, then the associated path condition acts as a precise description of the inputs that will drive (concrete) execution along the same path. Concrete values that satisfy the path condition are a counter-example to the assertion.

Unfortunately, symbolic execution suffers from the path explosion problem: the number of paths grows exponentially with the number of branch points in the design. Prior work has sought to avoid the path explosion problem by combining symbolic execution with model checking [6], concrete execution traces [40], or by limiting the use to small designs [42].

We introduce *piecewise composition*, a technique that leverages the structure of hardware designs and the power of satisfiability modulo theories (SMT) solving to reduce the amount of repeated work. A single clock cycle of symbolic execution produces a full tree of paths, where the root of the tree is the initialized reset state and the leaves are realizable design states in the next clock cycle. The inspiration behind piecewise composition is the recognition that independent parts of the design are being re-explored unnecessarily in each root-to-leaf path. Instead, each independent block of logic can be explored once, without consideration of the other blocks. To reconstruct full root-to-leaf paths through the design, whether for finding assertion failures, describing how information flows through a design, or to generate testcases, the algorithm uses SMT queries to combine the independently explored path fragments.

Perhaps surprisingly, we show that with piecewise composition, a design with N `always` blocks, each with at most b binary branch points, symbolic execution for a single clock cycle requires exploring $O(2^bN)$ paths, instead of the $O(2^{bN})$ paths typical of standard symbolic execution. The number of paths to explore grows exponentially with only the number of branch points in any one independent block, and linearly with the number of blocks.

Symbolic execution is closely related to symbolic simulation [2] [3] [6]. In both, concrete input values are replaced with symbolic values, representing any possible value, and the symbolic values are allowed to propagate through the design. How-

https://doi.org/10.34727/2023/isbn.978-3-85448-060-0_19

ever, in symbolic simulation, the analysis is centered around dataflow. At the end of a simulation run, each signal may hold the value true, false, or a boolean expression characterizing the entire circuit that drives that particular signal. Where there are control points in the circuit, they are expressed as `if-then-else` (`ITE`) statements in the boolean expression. In symbolic execution, the analysis is centered around control flow. At the end of one iteration of symbolic execution, each signal holds a symbolic expression in a subset of first-order logic that characterizes the particular path taken through the register-transfer level (RTL) code. In addition, there is a path condition that represents the conditions under which execution would follow the particular path through the design.

In comparing symbolic simulation and symbolic execution, there is a trade-off being made between the complexity of queries sent to the SMT solver (symbolic simulation) and the number of paths to explore (symbolic execution). With piecewise composition, we examine a new point in the design space, reducing the number of paths to explore to a tractable amount, while still keeping SMT queries simple enough for modern solvers. The result is a symbolic execution engine that can handle large designs and operate directly over the register-transfer level design. Sylvia targets the Verilog hardware description language (HDL), however the approaches and principles presented in this work are applicable to other HDLs.

This paper presents the following contributions:

1) Introduction and definition of *piecewise composition*, a technique that leverages the modular nature of hardware designs to counter the path explosion problem in symbolic execution;
2) Design and implementation of *Sylvia*, a symbolic execution engine for Verilog RTL using piecewise composition;
3) Evaluation of piecewise composition and our implementation on five open-source designs, including an SoC and two CPUs.

II. Preliminaries

A. Example Verilog RTL Fragment

In the following discussion we will use the fragment of Verilog shown in Figure 1. In this example, `inpA` and `inpB` are input signals (along with `clk`), while all other named variables are state-holding `regs`. We use the set V to denote all design variables (`regs`, `wires`, etc.) other than `clk`. In Figure 1, $V = \{\texttt{inpA}, \texttt{inpB}, \texttt{g0}, \texttt{g1}, \texttt{y}, \texttt{z}\}$.

B. Symbolic Execution

In symbolic execution [33], concrete values are replaced with symbolic values. Each symbol represents an arbitrary, but fixed, value of appropriate type. As execution proceeds, the symbols are used in place of concrete values wherever they occur. Variables may take on concrete values ($\{0,1\}^*$), symbolic values (α, β, γ,...), or a symbolic expression (e), which is an expression in a quantifier-free subset of first-order logic that supports bitvector arithmetic, the standard Verilog operators and the theory of equality. The symbolic execution

```
1  always @ (posedge clk) begin
2     if (g0)
3        y <= inpA; //inpA is an input signal
4     else
5        y <= 0;
6  end
7
8  always @ (posedge clk) begin
9     if (g1)
10       z <= inpB; //inpB is an input signal
11    else
12       z <= 0;
13 end
```

Fig. 1: Verilog RTL fragment with two branches.

engine (or just *engine* from now on) implements the semantics of the RTL Verilog; there is no compilation down to a netlist.

We model the symbolic execution of a design as a transducer $\text{SE} = (\text{RTL}, \Sigma, \Pi, \sigma_0, \pi_0, \mathscr{E})$:

- RTL is the design, modeled as a partially ordered sequence of Verilog statements.
- $\Sigma \subset 2^{V \times E}$ is the set of symbolic stores. Each symbolic store $\sigma \in \Sigma$ is a function mapping program variables in V to symbolic expressions in E: $\sigma : v \to e$.
- Π is the set of path conditions. A path condition $\pi \in \Pi$ is a boolean formula in the same subset of first of first-order logic mentioned in the preceding paragraph. The path condition is composed of symbolic and concrete literals, which describe the conditions satisfied by branches taken along the current path.
- σ_0 is the initial symbolic store. All input variables other than the clock are initialized with fresh symbols.
- π_0 is the initial path condition, which is always initialized to $\pi_0 = \text{True}$.
- $\mathscr{E} \subseteq \text{RTL} \times \Sigma \times \Pi \times \text{RTL} \times \Sigma \times \Pi$ is the transition relation of the engine. Given the current RTL statement, symbolic store, and path condition the engine updates the symbolic store and path condition and moves to a next statement to execute (at branch points there is more than one to choose from).

To continue with our example, in the RTL fragment shown in Figure 1, the symbolic store σ would maintain the values of variables `inpA`, `inpB`, `g0`, `g1`, `y`, `z`. (In the following discussion, we write out only the part of the symbolic store that is relevant to the discussion.) Let the (partial) initial symbolic store and path condition be:

$$\sigma_0 = \{\texttt{inpA} = \alpha, \texttt{g0} = \gamma\}, \quad \pi_0 = \text{True}$$

When a branching statement is reached (e.g., line 2 in Figure 1), the engine uses the current path condition to decide which path of execution to follow. If the engine has current path condition π and is at a branch statement with the boolean condition b, and if $\pi \to b$, the `then` branch is taken and the

path condition is updated: $\pi = \pi \wedge b$. Otherwise, if $\pi \rightarrow \neg b$, the `else` branch, if present, is taken and the path condition is updated: $\pi = \pi \wedge \neg b$. If neither implication holds, then both paths must be explored in turn. In our example, the engine will explore the two paths from line 1 to line 6, resulting in the following two symbolic stores and path conditions:

1) $\sigma_6 = \{\texttt{inpA} = \alpha, \texttt{g0} = \gamma, \texttt{y} = \alpha\}$, $\pi_6 = \text{True} \wedge \gamma == 1$
2) $\sigma_6 = \{\texttt{inpA} = \alpha, \texttt{g0} = \gamma, \texttt{y} = 0\}$, $\pi_6 = \text{True} \wedge \gamma == 0$

This example is simple, but in practice, path conditions quickly become complex, involving hundreds of terms and complex constraints in the theories necessary to express all Verilog operators.

At each branch point, the number of paths to explore doubles. This is the path explosion problem, and the result is that not all paths can feasibly be explored. Typically heuristics are used to guide the exploration toward paths that will maximize coverage or depth, or path-merging strategies are used to reduce the number of paths at the expense of less precise analysis [6], [18], [35].

C. Symbolic Execution Trees

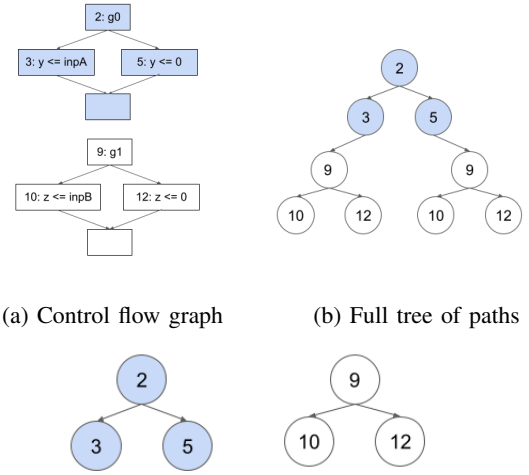
A trace, τ, of symbolic execution is a sequence of symbolic store and path condition pairs, $\tau = \langle(\sigma_0, \pi_0), (\sigma_i, \pi_i), (\sigma_j, \pi_j), \ldots, (\sigma_n, \pi_n)\rangle$, where the subscripts indicate the line of code associated with the symbolic state and path condition, and $0 < i < j < n$.[1] The complete symbolic execution of the RTL produces a tree of traces, T, as seen, for example, in Figure 2b. A path through the tree from the root node to a leaf node is a symbolic execution trace τ.

Each node (σ_i, π_i) in the tree is associated with a line of code in the RTL. More than one node in the tree will be associated with the same line of code. For example, in Figure 2b there are two distinct nodes associated with line 9, representing the two paths that can be taken to arrive at that line. These two nodes will necessarily have unique path conditions, the conjunction of which will be unsatisfiable.

D. Multiple Clock Cycles

The symbolic execution of a hardware design corresponds to a single clock cycle. Every path through the tree from the root node $n_r = (\sigma_0, \pi_0)$ to a leaf node $n_l = (\sigma_n, \pi_n)$ corresponds to a realizable step of the design from one state to the next. If n_r corresponds to a reachable state of the design (e.g., the reset state) that can be reached in k clock cycles, then n_l corresponds to a reachable state of the design that can be reached in k+1 clock cycles. The path condition associated with a leaf node n_l can be viewed as a predicate describing the (concrete) input values that would drive execution down the current path.

[1]The engine considers the subset of Verilog that uses only statically bounded loops and unrolls loops before execution, so that the engine never executes line i after line j, $i < j$.



(a) Control flow graph

(b) Full tree of paths

(c) Trees of paths to be independently explored under piecewise composition

Fig. 2: Piecewise Composition

E. Comparison with Symbolic Simulation

Symbolic simulation is a well-established technique in hardware testing and verification [9], [10], [12], [13], [34], [37]. Conceptually, symbolic simulation and symbolic execution are closely related. In both, symbols are used in place of concrete values for input or state variables. The symbolic values propagate as execution proceeds, and variables in the design take on symbolic expressions as values. Both techniques generalize testing, symbolic execution in the software domain [33], and symbolic simulation in the hardware domain [43].

A key difference in the two techniques, however, is in how branch points are handled. In symbolic execution, execution proceeds separately down each path following the branch point, with the path condition π keeping track of the conditions associated with the current path of execution. In symbolic simulation, however, there is no notion of separate paths and there is no path condition. Instead, branch points are captured as conditional assignments to variables. For example, for the code in Figure 1, as before, the value of y at the end of symbolically simulating the code fragment would be[2]:

$$\begin{aligned} \texttt{y} &:= \texttt{ite}(\gamma, \alpha, 0) \\ &:= (\gamma \wedge \alpha) \vee (\neg\gamma \wedge 0) \\ &:= \gamma \wedge \alpha \end{aligned}$$

The value of each output variable is a symbolic expression capturing the complete dataflow path from the inputs. However, without the separation of paths, the symbolic expression for variables becomes complex; this complexity is the limiting factor for symbolic simulation and is managed by initializing control variables to concrete values, a reasonable constraint for many functional verification tasks (e.g., [31]). For example, g0 would be given a concrete value, rather than the symbolic γ.

[2]`ite` is the `if-then-else` operator.

In contrast, symbolic execution separates paths and uses the path condition to store the constraints along the current path. As a result, control signals can by made symbolic without adding complexity to the symbolic expressions for each variable. Symbolic control signals allow for verifying a series of control-flow dependent properties without modifying the verification environment. For example, in our evaluation, the same environment set-up was used to verify all properties of a given design: the design was started in its initialized state, all input variables were made symbolic, and the desired property was checked.

While the limiting metric for symbolic simulation is the complexity of each variable's symbolic expression, the limiting metric for symbolic execution is the number of paths to explore. In this paper we present a technique to reduce the number of explorations needed.

F. Comparison with Bounded Model Checking

In bounded model checking the initial state of the system and the transition relation of the system are formally defined in a logic system, typically a subset of first-order logic [14], [15]. The reachable states of the system are computed up to a bound and checked against a desired property. Techniques such as IC3 [26] can allow for unbounded proofs of a property.

Prior work has reported that symbolic execution is at times able to find security property violations that model checking does not (see Section VI). In addition, in recent years, the hardware security community has turned its attention to analyzing how information flows through a design [4], [27], [29]. Doing so requires reasoning about hyperproperties [16], which requires self-composition in model checking [24], adding to the complexity of the verification task.

Symbolic execution, on the other hand, is suited to information-flow analysis, as the symbolic state σ and path condition π provide precise tracking of information flow from reset to the current execution point. A number of recent papers have explored the use of symbolic execution to analyze information flow through a hardware design [6], [25], [41].

G. Symbolic Simulation, Model Checking, Symbolic Execution

We do not advocate replacing either symbolic simulation or model checking with symbolic execution. Rather, it has become clear in recent years that symbolic execution is a valuable tool to add to the formal verification toolbox, especially when it comes to security verification tasks [6], [21], [25], [40]. We present an algorithmic technique to improve the performance of symbolic execution for hardware designs.

III. Piecewise Composition

In conventional symbolic execution, each line of code is potentially visited multiple times, once for each path explored. Our approach is to aggressively decompose the design into independent blocks, symbolically explore each block once, then use an SMT solver to compose path conditions and symbolic stores from each block. This strategy is made possible by the inherent modular nature of hardware designs, and lets us leverage the relative speed of modern SMT solvers compared to the cost of symbolically executing lines of code.

While the number of paths in the full symbolic execution tree is exponential in the number of branches in the design, the engine explores a number of paths exponential in only the number of branches in any single independent block and polynomial in the number of blocks.

A. Motivating Example

Figure 1 shows a snippet with two `always` blocks and branch points at lines 2 and 9. The corresponding control flow graph with an arbitrary ordering of the `always` blocks is given in Figure 2a, and the tree of paths through the design is given in Figure 2b. With conventional symbolic execution, each of the four root-to-leaf paths in Figure 2b is symbolically executed. This is the strategy taken by current approaches (e.g., [8], [25], [48]) that translate a hardware design into a C++ representation and then use the KLEE symbolic execution engine [11]. The two subtrees rooted at a node labeled 9 represent repeated work. For each subtree, the symbolic execution engine is exploring the same paths through the block starting at line 9.

The branching condition and assignments in lines 2–5 are independent of the branching condition and assignments in lines 9–12. Regardless of which path is taken at the first branch (line 2), the symbolic execution starting at the second branch point (line 9) will produce the same sub-tree. The feasibility of the second condition will be the same, and updates to the symbolic state will be the same. For example, let the initial symbolic store and path condition be:

$$\sigma_0 = \{\texttt{inpA} = \alpha, \texttt{g0} = \gamma_0, \texttt{inpB} = \beta, \texttt{g1} = \gamma_1\}$$
$$\pi_0 = \texttt{True}.$$

After symbolically executing the path in which both branches are taken (nodes $\langle 2,3,9,10\rangle$ in the symbolic execution tree), the symbolic store and path condition would be:

$$\sigma_{2,3,9,10} = \{\texttt{inpA} = \alpha, \texttt{g0} = \gamma_0, \texttt{inpB} = \beta, \texttt{g1} = \gamma_1, \texttt{y} := \alpha, \texttt{z} := \beta\}$$
$$\pi_{2,3,9,10} = \gamma_0 \wedge \gamma_1.$$

Whereas, for the path in which the first branch is not taken, but the second one is ($\langle 2,5,9,10\rangle$), the symbolic store and path condition would be:

$$\sigma_{2,5,9,10} = \{\texttt{inpA} = \alpha, \texttt{g0} = \gamma_0, \texttt{inpB} = \beta, \texttt{g1} = \gamma_1, \texttt{y} := 0, \texttt{z} := \beta\}$$
$$\pi_{2,5,9,10} = \neg\gamma_0 \wedge \gamma_1.$$

In both paths, the updates to `z` are the same, despite the different updates to `y`.

B. Piecewise Composition

With piecewise composition, the engine explores independent blocks of the RTL separately, producing independent trees of path fragments. In the above example, piecewise composition results in the two trees shown in Figure 2c.

The engine now explores the second `if-else` block only once. Continuing with our example, piecewise composition will separately explore the two `always` blocks, producing the following four path fragments with associated path conditions and (partial) symbolic stores:

$$\begin{aligned}
\langle 2,3\rangle: &\quad \sigma_{2,3} = \{y := \alpha\}, \quad \pi_{2,3} = \gamma_0 \\
\langle 2,5\rangle: &\quad \sigma_{2,5} = \{y := 0\}, \quad \pi_{2,5} = \neg\gamma_0 \\
\langle 9,10\rangle: &\quad \sigma_{9,10} = \{z := \beta\}, \quad \pi_{9,10} = \gamma_1 \\
\langle 9,12\rangle: &\quad \sigma_{9,12} = \{z := 0\}, \quad \pi_{9,12} = \neg\gamma_1
\end{aligned}$$

To find full paths through the design and to successfully find assertion violations, all realizable combinations of path fragments are composed with the help of an SMT solver. For example, to realize path $\langle 2,5,9,10\rangle$, the engine queries the SMT solver to find whether the two path fragments, $\langle 2,5\rangle$ and $\langle 9,10\rangle$ can be joined: $\texttt{isSAT}(\mathrm{y} = 0 \wedge \neg\gamma_0 \wedge \mathrm{y} = \beta \wedge \gamma_1)$. In this simple example, all four combinations of path fragments are possible, but in general that will not always be the case.

Piecewise composition will ultimately be constructing the same path conditions as conventional symbolic execution. The difference is in repeated work during the path exploration. Looking again at Figure 2c, The conventional approach will execute the following paths and their corresponding lines of code: $\langle 2,3,9,10\rangle$, $\langle 2,3,9,12\rangle$, $\langle 2,5,9,10\rangle$, $\langle 2,5,9,12\rangle$. Lines 2-3, 2-5, 9-10 and 9-12 are all explored twice. Piecewise composition will explore the following path fragments and corresponding lines of code, each only once: $\langle 2,3\rangle$, $\langle 2,5\rangle$, $\langle 9,10\rangle$, $\langle 9,12\rangle$. With this small example, piecewise composition is able to cut the path exploration workload in half. As the size of the design grows, the number of paths to explore with piecewise composition will be exponential only in terms of the number of branch points in a given `always` block and linear in the number of `always` blocks. We examine this more closely in the complexity analysis in Section III-D.

A conventional symbolic execution engine will query the SMT solver at branch 9 twice, once for path $\langle 2,3,9\rangle$ and once for path $\langle 2,5,9\rangle$. These queries are checking for feasibility of the branching condition at line 9 in the RTL. Piecewise composition will only query the solver for the branch on line 9 once. Piecewise composition will then require queries for all four complete paths through the design: $\langle 2,3,9,10\rangle$, $\langle 2,3,9,12\rangle$, $\langle 2,5,9,10\rangle$, $\langle 2,5,9,12\rangle$. These queries are ensuring that the accumulated path conditions are satisfiable and the execution path is realizable. In this small example, we do not reduce our SMT solving workload, but as the design becomes more complex piecewise composition yields a slight reduction in queries performed as we reduce the amount of redundant branch points explored. We evaluate the impact of piecewise composition both on lines of code explored and SMT queries in Section VI.

C. Comparison with Backtracking and Caching

Piecewise composition shares some similarities with backtracking and caching, two techniques often used in software symbolic execution engines (e.g., KLEE [11], Angr [32]). But, there are key differences. Backtracking reduces repeated work by maintaining state at each point in a path and allowing two paths with a shared prefix to reuse the saved state. For example, If path $\langle 2,3,9,10\rangle$ has been explored, then when the engine explores path $\langle 2,3,9,12\rangle$, backtracking allows the engine to reuse the saved state at point 9 and continue exploration from there. Backtracking prevents re-exploring path $\langle 2,3\rangle$ for each of $\langle 9,10\rangle$ and $\langle 9,12\rangle$; piecewise composition also prevents this re-exploration. However, with backtracking, paths $\langle 9,10\rangle$ and $\langle 9,12\rangle$ will be re-explored to create the paths starting with prefix $\langle 2,5\rangle$; this re-exploration is prevented by piecewise composition.

Caching queries reduces the time spent in the SMT solver by reusing the results from prior queries. Caching queries is a technique orthogonal to piecewise composition. Using the two techniques together could further reduce runtime.

D. Complexity Analysis

We separately compute the lines of code visited by the engine during symbolic execution and the number of queries to the solver for both the baseline implementation and the implementation that uses piecewise composition to develop a theoretical understanding of the benefits of piecewise composition.

We perform the analysis with the following Verilog design parameters and assumptions:

- b: The maximum number of branch points in any one `always` block. All branch points are assumed to have at most two branches. `Case` statements can be rewritten to use only 2-branch conditionals.
- N: The number of sequential-logic `always` blocks.
- c: The maximum number of lines of code after any branch point until either the next branch point or an exit point. In other words, the maximum number of lines of code in any *basic block*.
- Assumption 1: We assume that all loops are unrolled.
- Assumption 2: We approximate combinational logic with a fixed constant for both the baseline and piecewise composition approaches. We do this because the piecewise composition technique is only applicable to sequential `always` blocks. The underlying implementation strategy used by Sylvia to ensure clock-cycle accuracy with combinational logic statements results in each block of combinational logic being executed twice per clock cycle in the worst case (see Section IV-A for more details).

When we use the baseline approach, the symbolic execution of a design is represented by a single binary tree (as in Figure 2b). To simplify the analysis, we assume a perfect binary tree in which every interior node has two children and all leaf nodes are at the same level. This assumption only holds in practice if every branch point in the code is reachable along every path, which in general is not the case. Our analysis, therefore, provides a loose upper bound on complexity; a tighter bound may be possible.

When we use piecewise composition, the symbolic execution of a design is represented by N binary trees, one for each sequential-logic `always` block (as in Figure 2c).

1) Baseline: Lines of Code Symbolically Executed

Every node in the tree is visited once per path it belongs to.

$$\underbrace{bN(2^{bN})}_{(a)} + \underbrace{cbN(2^{bN})}_{(b)} \tag{1}$$

(a) Visit and execute each branch point (bN) once per path it is part of (2^{bN})

(b) Visit and execute the c lines of code in the basic block of either the right or left branch for each branch point (bN), once per path (2^{bN}).

The time complexity for executing the design symbolically using the baseline implementation is $O(cbN2^{bN})$. As the design size grows, the number of lines of code explored is growing exponentially with bN, the total number of branch points in the entire design.

2) Baseline: SMT Queries

Every branching node in the tree generates one query per visit, and is visited once per path it belongs to.

$$\underbrace{bN(2^{bN})}_{(a)} \tag{2}$$

(a) Visit each branch point (bN) once per path (2^{bN}) it is part of, and each visit generates one query.

The time complexity for querying the SMT solver under the baseline implementation is $O(bN2^{bN})$.

3) Piecewise Composition: Lines of Code Symbolically Executed

Every node in every tree is visited once per path it belongs to.

$$\underbrace{bN(2^{b})}_{(a)} + \underbrace{cbN(2^{b})}_{(b)} \tag{3}$$

(a) For each tree (N), visit and execute each branch point (b) once per path it is part of (2^b)

(b) For each tree (N), visit and execute the c lines of code in the basic block of either the right or left branch for each branch point (b), once per path (2^b).

The time complexity for executing the design symbolically using piecewise composition is $O(cbN2^{b})$. Compared with the baseline implementation, piecewise composition drops the exponential factor N and explores each unique path fragment once.

4) Piecewise Composition: SMT Queries

In addition to the queries for each branching node visited, one query is generated for every combination of paths, one from each tree, in order to recreate the full root-to-leaf paths.

$$\underbrace{bN(2^{b})}_{(a)} + \underbrace{(2^{b})^{N}}_{(b)} \tag{4}$$

(a) For each tree (N), visit each branch point (b) once per path it is part of, and each visit generates one query.

(b) Generating each path through the full design requires one query. Each of N trees has 2^b paths, and all combinations need to be combined.

The time complexity for querying the SMT solver using piecewise composition is $O(2^{bN})$. In the limit, there is a slight advantage in the number of SMT queries compared to the baseline implementation, and in practice we do see less time spent in the solver (see Figure 3).

IV. A Symbolic Execution Engine with Piecewise Composition

We introduce Sylvia, a symbolic execution engine implementing piecewise composition. Importantly, Sylvia operates directly over the Verilog RTL without translating to C or compiling down to the netlist. This allows for greater human-readability of any found assertion violations. Sylvia is cycle accurate. We assume no combinational latches, no asynchronous resets, and `always` blocks are conditioned on input clocks. These assumptions are in keeping with prior work in this area [6].

The core data structures Sylvia builds and uses are the Verilog AST and control-flow graphs (CFG). Sylvia constructs one CFG per `always` block. The symbolic execution trees described in the preceding sections are useful as a conceptual model of symbolic execution, but in practice the engine executes over the basic blocks of statements that are collected in each CFG. A single execution path in Sylvia is encoded as a combination of individual paths through the set of CFGs.

The engine achieves piecewise composition by decomposing a design into partitions: one partition to contain all combinational logic in the design, one partition for all register declarations, and a set of N partitions, one per `always` block, to handle the sequential logic in the design. Each partition is symbolically explored once per clock cycle, with the exception of the combinational logic partition, discussed next in Section IV-A.

Each of the N sequential `always` block partitions are explored independently of the other `always` blocks, and the exploration produces a set of path fragments. The complete exploration of the full design produces N sets, one per `always` block. The set of full root-to-leaf symbolic execution paths through the design is formed by taking the cross-product of the N sets of path fragments. The SMT solver is used to ensure only those combinations that are sound – that correspond to true paths through the design – are kept.

A. Combinational Logic

The engine will check for any combinational latches, and if any appear, will exit with an error. Otherwise, Sylvia first symbolically executes each statement in the combinational

logic partition and then begins to execute the control flow paths through each `always` block. As each `always` block is executed the engine keeps track of a dirty bit for each signal, which gets set to 1 when the signal is updated within that particular clock cycle. The intuition here is that if one of the combinational logic dependencies becomes dirty during the symbolic execution of the `always` block, we need to re-evaluate the corresponding combinational `assign`. Once a path has been completed, every `assign` statement in the combinational logic partition for which the right-hand side involves a dirty signal is re-evaluated. In the worst case, this means each statement in the combinational logic partition may be symbolically executed twice. During this re-evaluation, the engine continues to track when signals become dirty and propagate updates as needed to ensure clock-cycle accuracy.

B. Sequential Logic

Each sequential `always` block is explored independently. This approach is sound if the `always` blocks are truly independent – the path condition and symbolic state of the various paths through one block are the same regardless of the paths taken through other blocks. In the following we discuss the issue of independence in more detail. Consider two sequential `always` blocks, B_0 and B_1, both triggered on the same edge of the input clock signal.

1) Independence

In the simplest case, none of the variables that appear in B_0 appear in B_1. The two blocks are independent and, within a single clock cycle, the execution of one block has no bearing on the execution of the second block. The two blocks can be explored separately and their paths can be composed in any order. This case is rare, however, as an input reset signal typically appears in all or most blocks.

2) Read-read dependence

In the next case, the same variable may appear in a branch condition or right-hand side of an assignment in both B_0 and B_1. The two blocks can still be explored separately and their paths can be composed in any order. However, some combinations of paths may not be feasible, as variables that appear in branch conditions in both blocks, say b_0 and b_1, respectively, will preclude the combination of paths from B_0 in which b_0 holds with paths from B_1 in which b_1 holds when $b_0 \wedge b_1$ is unsatisfiable.

3) Read-write dependence

In the next case, a variable may appear in a branch condition or on the right-hand side of an assignment in B_0 and on the left-hand side of an assignment in B_1. When non-blocking assignments are used, updates to variables in B_1 take effect in the next clock-cycle, whereas reads and conditional branches in B_0 use values set in the previous clock cycle. The symbolic execution engine keeps the appropriate value and there is no conflict. The two blocks can be explored separately and their paths can be composed in any order. Sylvia does not support the use of blocking assignments within sequential `always` blocks.

4) Write-write dependence

In the final case, variables appear on the left-hand side of an assignment in both `always` blocks. This violates best practice in Verilog design. The symbolic execution engine will check for any instances of write-write dependence, and if any appear will exit with an error.

C. Further Optimizations

1) Repeat Submodules

When the modules are duplicate instantiations of the same module there is room for reduction in the total search space. The idea is similar in spirit to piecewise composition; the engine explores each submodule once for each path. Then instead of re-exploring again for each repeat instantiation, the engine merges in the symbolic store and path condition for the given root-to-leaf path using SMT queries.

2) Cone of Influence Analysis

This optimization prunes the exploration space at the block level. The symbolic execution engine will read in the expressions supplied in the assertions, perform a dependency analysis over the signals in the assertions and then complete an AST traversal to determine which blocks read from or write to the signals of interest or their dependencies. After this initial pass, the engine will only explore blocks that involve the signals of interest or their dependencies.

V. Implementation

Sylvia[3] is built in python 3.8 and implements the Verilog semantics according to the IEEE 1364-2005 standard. We use the pyVerilog library to build the Verilog AST, networkX to manage graph search and traversal, and the Z3 python API for SMT solving. The engine reads in a design, including the assertions written according to the SystemVerilog 1800-2017 standard, and outputs replayable counterexamples.

VI. Evaluation

We evaluate Sylvia over five open-source designs to study its viability as a platform for the verification of hardware designs. Our evaluation considers the following questions: 1) How well does piecewise composition counter the path explosion problem? 2) What effect do piecewise composition and the optimizations described in Section IV-C have on performance? 3) Does our engine produce assertion violations with replayable counter-examples for vulnerable designs?

A. Dataset and Experimental Setup

We collected five designs and 84 security critical assertions. The first three designs and associated assertions came from the Security Property/Rule Database available on TrustHub [22], [23]. These are an enhanced version of the Serial Peripheral Interface available on Motorola's MC68HC11 family of CPUs; openMSP430, a synthesizable 16-bit microcontroller core compatible with Texas Instruments' MSP430 microcontroller family; and a CrypTech True Random Number Generator

[3]Sylvia is fully open-source and can be accessed at https://github.com/kakiryan/Sylvia

(TRNG). For each of these designs, the database included 9, 2, and 2 security properties, respectively.

The fourth design is the buggy PULPissimo SoC used in a recent Hack@DAC competition [1]. Using the English description of the properties, as well as the walkthrough of the test-case generation in the RTL-ConTest paper [40] we developed 26 assertions for use with our tool.

The fifth design is the OR1200 processor core. We collected 30 security-critical bugs from two prior papers, SPECS [28] and SCIFinder [49] and 70 security assertions from SPECS [28], Security Checkers [7], SCIFinder [49], and Transys [50].

The experiments are performed on a machine with an Intel Xeon E5-2620 V3 12-core CPU (2.40GHz, a dual-socket server) and 62G of available RAM.

B. Mitigation of Path Explosion

For each design we compare the average number of lines of code and branch points visited to find an assertion violation both with and without piecewise composition. Table I has the results. The number of paths in a design will not change, but the amount of work to realize a path does. Piecewise composition reduces the number of lines of code visited (i.e., reduces redundant visits to the same line of code) by 92%–99% and branch points visited by 64%–99%.

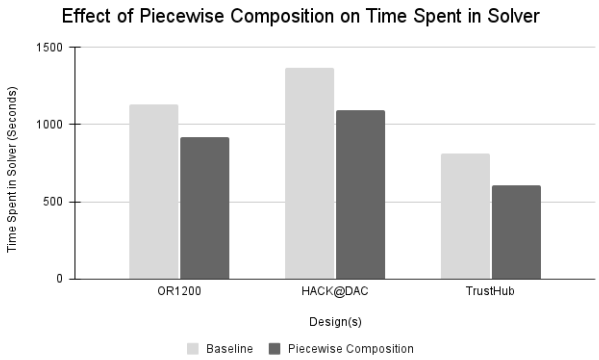
To gain a more complete picture, we symbolically explore all paths through the small MC68HC11 SPI. This design has 15 `always` blocks and 1459 possible paths. These results are reported in Table II. Without piecewise composition more than 90k lines of code and more than 7k branch points are explored. With piecewise composition, the engine needs to explore roughly only 7% of those 90k lines of code and only 4% of those 7k branch points.

The benefits of piecewise composition come from the presence of composable `always` blocks. We report on the number and dependency type (Section IV-B) of these structures in our benchmarks in Table III. As expected, all `always` blocks have at most read-write dependencies, allowing them to be composed.

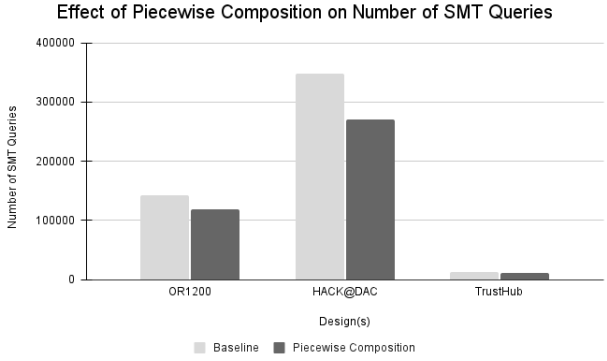
C. Effects of Optimizations

Figure 3 shows the impact of piecewise composition on the average number of SMT queries and average time spent in the SMT solver for each design. One concern might be that the win in minimizing redundant explorations comes at the expense of exploding SMT solver work. However, the opposite occurs. Reducing the number of paths explored also reduces the number of queries to the solver overall; remember that during exploration, the solver is queried at each branch point. With piecewise composition turned on there was an 18% decrease, on average, in the number of SMT queries and a 21% decrease in the amount of time spent solving.

In Table IV we measure runtime for four cases: *Baseline*, with no optimizations enabled; *Piecewise*, with piecewise composition enabled, *Repeat*, with repeated modules explored only once; and *COI*, with cone-of-influence analysis completed



(a) Solver Time



(b) SMT Queries

Fig. 3: Effects of Optimizations on Solver Time and Number of SMT Queries

before exploration. Each case is cumulative, for example, in the *Repeat* case, the *Piecewise* optimization is enabled as well. For each design we take the average when looking for each assertion. For all but the smallest design, *Baseline* could not reliably complete exploration within 30 minutes at which point we stopped searching. Table IV shows the results. Overall, the optimizations decrease the engine's runtime by 95-99%.

D. Finding Assertion Violations

To evaluate the engine's ability to find assertion violations, we run a set of experiments in which we have ground-truth knowledge of the (minimum) number of violations in each design. In these experiments, symbolic execution begins in the reset state with all input signals made symbolic, and execution continues until an assertion violation is found. All counterexamples generated by our engine were successfully replayed in simulation starting from the reset state using Vivado. Table V summarizes the results.

The engine finds 25 of the 31 bugs in the Hack@DAC SoC. The organizers of Hack@DAC report finding 6 and 15 bugs using the commercial tools Cadence SPV and Cadence FPV, respectively [1]. The engine finds 29 of the 30 bugs in the OR1200. The bug missed does not a have a property in our dataset that covered it.

Design	Baseline		Piecewise Composition		Percent Decrease	
	LoC explored	branch points explored	LoC explored	branch points explored	LoC explored	branch points explored
OR1200	54018	7803	881	45	98%	99%
Hack@DAC	493032	15093	3525	276	99%	98%
MC68HC11 SPI	2093	158	174	57	92%	64%
openMSP430	15293	377	489	68	97%	82%
CrypTech TRNG	8930	421	336	91	96%	78%

TABLE I: Average Impact of Piecewise Composition on Path Explosion

Configuration	LoC explored	branch points explored	paths completed
Baseline	90706	7380	1459
Piecewise Composition	6783	323	1459

TABLE II: Full Exploration of MC68HC11 SPI Design

The engine is consistently able to find vulnerabilities that both commercial and open-source model checking tools are unable to find. Table VI summarizes how Cadence and SymbiYosys [2], a symbolic model checking engine built on top of Yosys, fared in finding the same known vulnerabilities.[4]

At the time of writing we don't know why the model checking tools were unable to find all the assertion violations and further investigation is warranted. We do not believe that it is the result of a theoretical limitation of bounded model checking or the strength of the algorithms used by the underlying proof engines. More likely, our hypothesis is that there are some abstractions introduced to manage complexity that cause the property violations to be missed. The properties we are searching for are specifying system level behavior and in some cases have been automatically generated. The complexity inherent to these types of security properties compared to typical functional correctness properties may make it more difficult for a traditional formal verification approach like model checking to find the violations.

We set the bound for the model checking tools to be 5 clock cycles over the minimum bounds needed to find the violation, and let the tools run to completion. Our results align with results reported by the authors of performing comparable experiments [40] [48] and [19] and match those of the authors of the TrustHub designs and properties that we were using for evaluation.

We demonstrate how our approach allows search to scale more efficiently over multiple clock cycles compared to SymbiYosys in Figure 4. We take the MSP430 design and embed properties into the design that require an increasing number of cycles to produce a counterexample. SymbiYosys begins by outperforming our tool in terms of speed – it takes Symbiyosys under half a second to complete 4 cycles while we take around 3 seconds. As the complexity of the search space grows, piecewise composition scales more manageably.

[4]The numbers in the Cadence column are pulled from the literature [1], [23], [40], [48]. Our license does not allow for head-to-head comparisons.

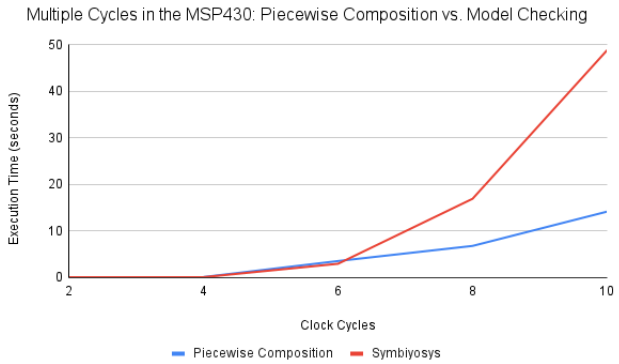


Fig. 4: Scaling Search Over Multiple Cycles

E. Comparison to Current State of the Art

Coppelia is a tool that performs symbolic execution over the C++ model of a Verilog hardware design. In comparison to Coppelia [48], we see significant performance gains. On average, Coppelia takes 4 minutes and 12 seconds to find the same known security vulnerabilities in the OR1200 that our tool is able to find in 25.22 seconds. The authors report that most (62%) of exploits in their experiments are generated within 15 minutes. However, several (7%) are found within 2–4 hours. Symbolic execution with piecewise composition, by contrast, finds the same 7% of exploits in under two minutes.

The authors of RTLConTest [40], a concolic execution engine, report that it takes around an hour and 40 minutes to complete on the PULPissimo SoC, which is a modified version of the HACK@DAC 2018 design. Once they perform the concolic execution and generate the tests, it takes 10 seconds on average to produce a counterexample. They find 14 out of the 31 bugs while our engine is able to find 25. Our tool performs the complete end-to-end symbolic execution workflow to generate counterexamples in 81.83 seconds, on average.

VII. Related Work

Symbolic Simulation Of the papers presented in Section II-E, we note the early work implementing symbolic simulation at the RT level [34] that introduces a path-merging approach for handling and mitigating the complex queries characteristic of symbolic simulation. A more recent project is the Rosette/Racket solver-aided programming platform [45],

Design	LoC	Always Blocks	Branch Points	% Independent	% Read-Read	% Read-Write	% Composable
OR1200	30611	405	976	6.81%	40.98%	52.21%	100%
Hack@DAC	96444	650	4452	12.89%	42.33%	44.78%	100%
MC68HC11 SPI	527	14	43	21.45%	34.89%	44.66%	100%
openMSP430	9154	144	316	15.34%	27.27%	57.39%	100%
CrypTech TRNG	5926	54	309	8.21%	32.31%	59.48%	100%

TABLE III: Logical Structure of Benchmarks

Design	Baseline	Piecewise		Redund		COI		Overall
	runtime (sec)	runtime (sec)	% dec	runtime (sec)	% dec	runtime (sec)	% dec	% dec
OR1200	timeout (1800)	52.47	97.08%	37.56	21.31%	25.22	12.56%	98.60%
Hack@DAC	timeout (1800)	174.24	90.32%	121.94	28.34%	81.83	16.62%	95.45%
MC68HC11	962	17.53	98.18%	14.30	19.93%	0.07	99.19%	99.99%
openMSP430	timeout (1800)	37.65	97.91%	23.14	38.55%	0.73	96.83%	99.96%
CrypTech TRNG	timeout (1800)	14.92	99.17%	12.08	19.15%	0.09	99.19%	99.99%

TABLE IV: Average Effect of Optimizations on Runtime

Design	# Bugs	# Bugs Found	Avg Time (sec)	Max Clock Cycles Taken
Hack@DAC	31	25	81.83	4
OR1200	30	29	25.22	5
MC68HC11	9	9	0.07	3
openMSP430	2	2	0.73	2

TABLE V: Finding Known Bugs: Runtime Performance

Design	# Bugs	Our Engine	Cadence	SymbiYosys
OR1200	30	29	18	18
Hack@DAC	31	25	21	16
MC68HC11	9	9	8	5
openMSP430	2	2	2	1

TABLE VI: Finding Known Bugs: Comparison to Model Checking

whose use is demonstrated for verification in Notary [5]. Both works, like all symbolic simulation, merge symbolic states after each branch point, and require constraining the control-flow with concrete inputs to manage expression complexity.

Model Checking As discussed in Section II-F, model checking is a mature tool widely used in industry and research. SymbiYosys [2] is a formal verification engine for Verilog that operates at the netlist level. We compare to this tool in our evaluation (Section VI). Symbolic Quick Error Detection [20], [38] is a technique involving self-consistency checks that has been used to find bugs in open-source RISC-V processors and uses the CoSA model checker [39].

Symbolic and Concolic Execution Symbolic execution and the related technique of concolic execution are emerging from the academic research community as useful techniques for the security verification of hardware designs [8], [25], [40], [47], [48]. However, many of these papers rely on first translating Verilog to C++ and using KLEE [11], a tool written for, and fine-tuned for, the symbolic execution of software programs.

Fuzzing Fuzzing has also been shown to be a useful technique for finding security vulnerabilities in SoCs and CPU designs. RFUZZ is a coverage-directed fuzz tester for circuits that presents a hardware-specific coverage metric called *mux control coverage* [36]. DifuzzRTL is an RTL fuzzing tool used to find unknown security bugs that measures coverage based on control registers rather than multiplexors' control signals to improve efficiency and scalability [30]. A recently developed Hardware Fuzzing Pipeline translates the RTL to a software model to improve scalability in bug finding via fuzzing [46].

VIII. Conclusion

We have presented piecewise composition, a technique for countering the path explosion problem in symbolic execution. We implemented Sylvia, a symbolic execution engine using the technique and evaluated the engine on five open-source designs. The engine reduces redundant work by 98%–99% compared to conventional symbolic execution, improves overall performance and successfully finds assertion violations.

IX. Acknowledgments

We would like to thank Sayak Ray and the anonymous reviewers for their insightful comments and suggestions. This material is based upon work supported by the National Science Foundation under Grant No. CNS-1816637 and Grant No. CNS-2247754, and by a Meta Security Research Award. Any opinions, findings, conclusions, and recommendations expressed in this paper are solely those of the authors.

References

[1] "Hack@DAC 2018 SoC," https://github.com/seth-lab-tamu/hackdac-2018-soc, accessed: 2022-02-15.
[2] "SymbiYosys," https://github.com/YosysHQ/sby, accessed: 2022-11-21.
[3] "Voss II," https://github.com/TeamVoss/VossII, accessed: 2022-11-21.

[4] A. Ardeshiricham, W. Hu, J. Marxen, and R. Kastner, "Register Transfer Level information Flow Tracking for Provably Secure Hardware Design," in *DATE*, 2017, pp. 1691–1696.

[5] A. Athalye, A. Belay, M. F. Kaashoek, R. Morris, and N. Zeldovich, "Notary: A Device for Secure Transaction Approval," in *27th Symposium on Operating Systems Principles (SOSP)*. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3341301.3359661

[6] A. Athalye, M. F. Kaashoek, and N. Zeldovich, "Verifying Hardware Security Modules with Information-Preserving Refinement," in *OSDI*. USENIX Association, 2022.

[7] M. Bilzor, T. Huffmire, C. Irvine, and T. Levin, "Security Checkers: Detecting processor malicious inclusions at runtime," in *HOST*, 2011.

[8] N. Bruns, V. Herdt, and R. Drechsler, "Processor Verification using Symbolic Execution: A RISC-V Case-Study," 2023. [Online]. Available: https://agra.informatik.uni-bremen.de/doc/konf/2023_DATE_NB.pdf

[9] R. E. Bryant, "Symbolic Simulation—Techniques and Applications," in *Proceedings of the 27th ACM/IEEE Design Automation Conference*, ser. DAC '90. New York, NY, USA: Association for Computing Machinery, 1991, p. 517–521. [Online]. Available: https://doi.org/10.1145/123186.128296

[10] R. E. Bryant, D. L. Beatty, and C.-J. H. Seger, "Formal Hardware Verification by Symbolic Ternary Trajectory Evaluation," in *ACM/IEEE DAC*, 1991.

[11] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[12] S. Chakraborty, Z. Khasidashvili, C.-J. H. Seger, R. Gajavelly, T. Haldankar, D. Chhatani, and R. Mistry, "Symbolic Trajectory Evaluation for Word-Level Verification: Theory and Implementation," *Form. Methods Syst. Des.*, vol. 50, no. 2–3, p. 317–352, Jun. 2017.

[13] K. Claessen and J.-W. Roorda, "An Introduction to Symbolic Trajectory Evaluation," in *Formal Methods for Hardware Verification*, M. Bernardo and A. Cimatti, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 56–77.

[14] E. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded Model Checking Using Satisfiability Solving," vol. 19, no. 1, p. 7–34, 2001.

[15] E. M. Clarke, "Model Checking," in *Foundations of Software Technology and Theoretical Computer Science*, S. Ramesh and G. Sivakumar, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 54–56.

[16] M. R. Clarkson and F. B. Schneider, "Hyperproperties," *J. Comput. Secur.*, vol. 18, no. 6, pp. 1157–1210, Sep. 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1891823.1891830

[17] C. N. Coelho and H. D. Foster, *Assertion-Based Verification*. Boston, MA: Springer US, 2004, pp. 167–204.

[18] D. Davidson, B. Moench, T. Ristenpart, and S. Jha, "FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution," in *USENIX Security Symposium*, 2013.

[19] G. Dessouky, D. Gens, P. Haney, G. Persyn, A. Kanuparthi, H. Khattri, J. M. Fung, A.-R. Sadeghi, and J. Rajendran, "HardFails: Insights into Software-Exploitable Hardware Bugs," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 213–230. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/dessouky

[20] M. R. Fadiheh, J. Urdahl, S. S. Nuthakki, S. Mitra, C. Barrett, D. Stoffel, and W. Kunz, "Symbolic Quick Error Detection Using Symbolic Initial State for Pre-Silicon Verification," in *Design, Automation & Test in Europe (DATE)*, 2018, pp. 55–60.

[21] M. R. Fadiheh, A. Wezel, J. Muller, J. Bormann, S. Ray, J. M. Fung, S. Mitra, D. Stoffel, and W. Kunz, "An Exhaustive Approach to Detecting Transient Execution Side Channels in RTL Designs of Processors," *IEEE Transactions on Computers*, vol. 72, no. 1, pp. 222–235, jan 2023. [Online]. Available: https://doi.org/10.1109%2Ftc.2022.3152666

[22] N. Farzana, F. Farahmandi, and M. M. Tehranipoor, "SoC Security Properties and Rules," *IACR Cryptol. ePrint Arch.*, vol. 2021, p. 1014, 2021.

[23] N. Farzana, F. Rahman, M. Tehranipoor, and F. Farahmandi, "SoC Security Verification using Property Checking," in *2019 IEEE International Test Conference (ITC)*, 2019, pp. 1–10.

[24] B. Finkbeiner, M. N. Rabe, and C. Sánchez, "Algorithms for Model Checking HyperLTL and HyperCTL," in *Computer Aided Verification*, D. Kroening and C. S. Păsăreanu, Eds. Cham: Springer International Publishing, 2015, pp. 30–48.

[25] F. Fowze, M. Choudhury, and D. Forte, "EISec: Exhaustive Information Flow Security of Hardware Intellectual Property Utilizing Symbolic Execution," in *Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*. IEEE Xplore, 2022.

[26] A. Goel and K. Sakallah, "Model Checking of Verilog RTL Using IC3 with Syntax-Guided Abstraction," in *NASA Formal Methods*, J. M. Badger and K. Y. Rozier, Eds. Cham: Springer International Publishing, 2019, pp. 166–185.

[27] M. Goli and R. Drechsler, "VIP-VP: Early Validation of SoCs Information Flow Policies using SystemC-based Virtual Prototypes," in *2021 Forum on specification & Design Languages (FDL)*, 2021, pp. 1–8.

[28] M. Hicks, C. Sturton, S. T. King, and J. M. Smith, "SPECS: A Lightweight Runtime Mechanism for Protecting Software from Security-Critical Processor Bugs," in *ASPLOS*, ser. ASPLOS '15. New York, NY, USA: ACM, 2015, p. 517–529.

[29] W. Hu, A. Ardeshiricham, M. S. Gobulukoglu, X. Wang, and R. Kastner, "Property Specific Information Flow Analysis for Hardware Security Verification," in *ICCAD*, ser. ICCAD '18. New York, NY, USA: Association for Computing Machinery, 2018.

[30] J. Hur, S. Song, D. Kwon, E. Baek, J. Kim, and B. Lee, "DifuzzRTL: Differential Fuzz Testing to Find CPU Bugs," in *42nd IEEE S&P*. IEEE, 2021, pp. 1286–1303.

[31] R. Kaivola and N. B. Kama, "Timed Causal Fanin Analysis for Symbolic Circuit Simulation," in *Formal Methods in Computer-Aided Design (FMCAD)*, 2022, pp. 99–107. [Online]. Available: https://repositum.tuwien.at/handle/20.500.12708/81329

[32] S. Kim, M. Faerevaag, M. Jung, S. Jung, D. Oh, J. Lee, and S. K. Cha, "Testing Intermediate Representations for Binary Analysis," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2017. IEEE Press, 2017, p. 353–364.

[33] J. C. King, "Symbolic Execution and Program Testing," *Commun. ACM*, vol. 19, no. 7, p. 385–394, Jul. 1976.

[34] A. Kölbl, J. Kukula, and R. Damiano, "Symbolic RTL Simulation," in *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, 2001, pp. 47–52.

[35] S. Krishnamoorthy, M. S. Hsiao, and L. Lingappan, "Tackling the Path Explosion Problem in Symbolic Execution-Driven Test Generation for Programs," in *2010 19th IEEE Asian Test Symposium*, 2010, pp. 59–64.

[36] K. Laeufer, J. Koenig, D. Kim, J. Bachrach, and K. Sen, "RFUZZ: Coverage-Directed Fuzz Testing of RTL on FPGAs," in *ICAAD*, 2018, pp. 1–8.

[37] W. K. Lam, *Hardware Design Verification: Simulation and Formal Method-Based Approaches (Prentice Hall Modern Semiconductor Design Series)*. USA: Prentice Hall PTR, 2005.

[38] F. Lonsing, K. Ganesan, M. Mann, S. S. Nuthakki, E. Singh, M. Srouji, Y. Yang, S. Mitra, and C. Barrett, "Unlocking the Power of Formal Hardware Verification with CoSA and Symbolic QED: Invited Paper," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2019, pp. 1–8.

[39] C. Mattarei, M. Mann, C. Barrett, R. G. Daly, D. Huff, and P. Hanrahan, "CoSA: Integrated Verification for Agile Hardware Design," in *2018 Formal Methods in Computer Aided Design (FMCAD)*, 2018, pp. 1–5.

[40] X. Meng, S. Kundu, A. K. Kanuparthi, and K. Basu, "RTL-ConTest: Concolic Testing on RTL for Detecting Security Vulnerabilities," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 3, pp. 466–477, 2022.

[41] N. Moroze, A. Athalye, M. F. Kaashoek, and N. Zeldovich, "rtlv: push-button verification of software on hardware," in *Workshop on Computer Architecture Research with RISC-V (CARRV)*, 2021. [Online]. Available: https://vm-web.pdos.csail.mit.edu/papers/rtlv:carrv21.pdf

[42] R. Mukherjee, D. Kroening, and T. Melham, "Hardware Verification Using Software Analyzers," in *2015 IEEE Computer Society Annual Symposium on VLSI*, 2015, pp. 7–12.

[43] C.-J. H. Seger and R. E. Bryant, "Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories," *Formal Methods in System Design*, vol. 6, no. 2, p. 147–189, Mar 1995. [Online]. Available: https://doi.org/10.1007/BF01383966

[44] L. Shen, D. Mu, G. Cao, M. Qin, J. Blackstone, and R. Kastner, "Symbolic Execution Based Test-patterns Generation Algorithm for Hardware Trojan Detection," *Comput. Secur.*, vol. 78, pp. 267–280, 2018.

[45] E. Torlak and R. Bodik, "A lightweight symbolic virtual machine for solver-aided host languages," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and*

Implementation, ser. PLDI '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 530–541. [Online]. Available: https://doi.org/10.1145/2594291.2594340

[46] T. Trippel, K. G. Shin, A. Chernyakhovsky, G. Kelly, D. Rizzo, and M. Hicks, "Fuzzing Hardware Like Software," in *USENIX '22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 3237–3254.

[47] H. Witharana, Y. Lyu, and P. Mishra, "Directed Test Generation for Activation of Security Assertions in RTL Models," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 26, no. 4, jan 2021.

[48] R. Zhang, C. Deutschbein, P. Huang, and C. Sturton, "End-to-End Automated Exploit Generation for Validating the Security of Processor Designs," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*. IEEE/ACM, 2018.

[49] R. Zhang, N. Stanley, C. Griggs, A. Chi, and C. Sturton, "Identifying Security Critical Properties for the Dynamic Verification of a Processor," in *ASPLOS*, ser. ASPLOS '17. New York, NY, USA: ACM, 2017, p. 541–554.

[50] R. Zhang and C. Sturton, "Transys: Leveraging Common Security Properties Across Hardware Designs," in *Proceedings of the Symposium on Security and Privacy (S&P)*. IEEE, 2020.