Sylvia: Software-Style Symbolic Execution for the Security Verification of Hardware Designs

ABSTRACT

This paper presents *Sylvia*, a software-style symbolic execution engine developed for the security verification of hardware designs. Sylvia works directly on designs implemented in the Verilog Hardware Description Language (HDL) at the Register Transfer Level (RTL) without requiring translation to a software simulation of the design. The benefit of this approach is that Sylvia can leverage key characteristics of hardware designs, namely the modular structure of many designs and the parallel nature of hardware, to optimize search and mitigate the path explosion problem. Sylvia implements three optimizations for improving performance and decreasing the time needed to find security assertion violations. The hardware-oriented optimizations decrease Sylvia's runtime by 98%–99%.

We evaluate Sylvia on multiple open-source SoC and CPU designs, including the PULPissimo SoC used in a recent Hack@DAC competition. Using 70 properties related to CPUs and SoCs taken from the security literature Sylvia successfully finds known security vulnerabilities in the open-source OR1200 RISC CPU and the PULPissimo SoC. On average, it takes Sylvia 75 seconds and 221 seconds to find each assertion violation in the OR1200 and PULPissimo SoC, respectively.

1 Introduction

The verification of hardware designs is a key activity to ensure the correctness [**?**] and security [**?**] of a design early in the hardware lifecycle. Current best practice includes assertion-based verification (ABV) [19], 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. Both practices have been successful, but both have their limitations. Simulation-based testing struggles with providing coverage guarantees, especially for complex assertions []. Formal verification techniques have a reputation for being difficult to use without specific expertise in the field. One challenge in particular with using formal verification is conducting the root-cause analysis after a property violation is found.

A new approach, software-style symbolic execution,[1] was recently introduced as an alternative to both testing and formal verification [43], especially with respect to security validation. Software-style symbolic execution can provide high coverage plus meaningful coverage metrics, can be used without expertise in formal methods, and eases the root-cause analysis task when a property violation is found. However, symbolic execution was developed for use with software, not hardware. While Zhang et al. [43] demonstrates its use and value in hardware verification, there does not yet exist a symbolic execution engine developed explicitly for hardware. This paper presents *Sylvia*, a software-style symbolic execution engine developed for hardware designs implemented in the Verilog Hardware Description Language (HDL) at the Register Transfer Level (RTL).

Symbolic execution is a technique that generalizes software 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 an updated syntax and semantics of the program's language that includes symbols. As execution proceeds the symbols are used in place of literal values. The result of symbolically executing a program is a tree of paths through the program, each one associated with a unique *path condition* that describes the conditions satisfied by branches taken along the path. 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. KLEE [11], Mayhem [13], and angr [29] are all examples of software-style symbolic execution engines, but there are many more.

Software-style symbolic execution can improve hardware security validation efforts, finding security-critical bugs in hardware designs that neither testing nor model checking find. However, Zhang's work relied on a process of first translating the design from a hardware description language to a cycle-accurate representation in C++ and then using tools developed for software to symbolically explore the design. Sylvia takes the Verilog RTL of a hardware design and directly performs the symbolic execution without first translating to a software simulation of the design.

We identify and demonstrate two benefits to building a symbolic execution engine designed specifically for hardware. First, Sylvia leverages the characteristics of hardware to build more efficient search strategies. An engine designed for software misses the opportunity to do so. Second, Sylvia searches for bugs using properties relating the signals, reg-

[1]We use the phrase *software-style symbolic execution* to avoid confusion with the hardware technique, symbolic simulation. Going forward we will use the full phrase or the shorter phrase, *symbolic execution*, interchangeably.

isters, and ports of the design. A workflow that requires translating the design to C++, on the other hand, also requires rewriting the properties to use the variables of the C++ representation, which often do not have a one-to-one mapping with registers in the original Verilog.

This paper presents the following contributions: (1) We design and implement, Sylvia, a hardware-oriented symbolic execution engine that requires no translation to a higher level programming language to perform the analysis. To the best of our knowledge, this is the first symbolic execution engine developed specifically for hardware. (2) We develop algorithms and search strategies for exploring the design space tailored specifically for properties inherent to hardware designs. (3) We evaluate Sylvia's ability to find known security vulnerabilities across multiple open-source CPU and SoC designs. (4) We compare Sylvia's efficacy to existing workflows for hardware security verification.

2 Background

This section will provide the necessary background information for the discussion of a symbolic execution engine developed specifically for hardware designs. We begin with standard software-style symbolic execution [30]. Then we will provide some context about the Verilog hardware description language and hardware designs written at the register transfer level. We will also provide some intuition for what it means to symbolically execute such a hardware design.

2.1 Symbolic Execution

Symbolic execution is a powerful technique for automatically generating test cases in the software community. Program inputs are made symbolic where each symbol represents an equivalence class of input values that drive execution down the same path. This abstraction allows for more systematic and complete test coverage than supplying purely random inputs. A symbolic execution of the program proceeds by updating the standard syntax and semantics of the programming language to include symbols. A symbolic execution engine is the tool that drives this symbolic program execution. Symbolic execution engines are also responsible for keeping track of the "execution state." For most engines, this will consist of two main components:

1. A symbolic store with mappings between program variables and symbolic expressions.

2. The path condition: a boolean formula over symbolic expressions describing the conditions satisfied by branches taken along a path.

When a branching statement is reached, the symbolic execution engine will fork, considering each possible path. This exhaustive search does not scale to large programs as the number of paths to explore grows exponentially with the number of branches in a program. This is known as the path explosion problem, and typically heuristics or merging strategies are used to guide the exploration to maximize coverage or depth.

An execution tree is a way of representing different paths through the program as taken by the symbolic execution engine. Each node in the tree maps to an executed statement, and the transitions between statements are directed arrows, symbolizing a change in execution state. Each node represents one such state, with information such as the path condition and current symbolic variable values.

Common constructs in software programs can present challenges for symbolic execution. For example, unbounded loops make for an infinite number of infinitely long execution paths through the program. Pointers, string manipulation, and floating point operations make for infeasible queries to the constraint solver. And, external libraries represent an opaque box, which execution must travel through, but about which the symbolic execution engine cannot reason. Hardware designs do not have theses challenges, making the designs particularly well suited to the use of software-style symbolic execution.

2.2 Hardware Description Languages

Verilog is a hardware description language and is the industry standard for developing real-world computer systems. A basic unit of design in Verilog is a module. Modules often contain other modules, making the design hierarchical. A module combines multiple sub-modules by making the output signals of one module connect to the input signals of a second module, with connection wires and registers in between. Verilog has several other constructs that allow the hardware to flow differently than a sequential-only software program would. The specific constructs of interest will be discussed more in sections 4.2.5-4.2.8. An important piece to keep in mind is that if a signal is within a certain stateful block of the design, an `always` block, it will only be updated at the rising edge of a clock. Within an `always` block is you have a collection of statements that all get executed in parallel whenever signals in a sensitivity list go high.

2.3 Constraint Solving

SMT is a generalization of boolean satisfiability (SAT) which takes a formula and determines if there is an assignment that makes it true. SMT generalizes this by supporting theories that are more expressive and can capture more functionality. An example of this is supporting array operations or linear arithmetic. SMT solvers have become quite powerful and are able to scale to process expressions relying on combinations of hundreds of variables. However, SMT solving is still an NP-complete problem.

SMT solvers are crucial to symbolic execution both in generating assignments to symbolic variables and test cases, and also in checking feasibility of paths as the engine progresses. Some of the most widely used solvers are STP [34] (used in KLEE) and Z3 (used in Mayhem and angr) [20]. There is a large body of work being done on different optimizations made to make the solving stages of symbolic execution more efficient. For example, KLEE presents several strategies like implied value concretization and keeping a constraint cache. Designers of symbolic execution engines should be thinking about ways to keep the queries being sent to the solver as simple as possible, because the solving can be a major bottleneck of the analysis if these optimizations are not done right.

2.4 Symbolic Execution of Hardware

The symbolic execution of a hardware design corresponds to the complete exploration of the design for a single clock cycle. The symbolic state kept by a symbolic execution engine for hardware need only be clock-cycle accurate. This means that the internals of the engine provides an accurate representation of the design at each clock-cycle boundary, but in order to get to this accurate representation some operations and calculations resulting in intermediate inconsistent states may have been necessary.

Despite the symbolic execution tree being finite for a single clock tick, hardware executes continuously and latent security vulnerabilities may only become clear several clock cycles after the initial state, so hardware is still unable to escape the path explosion problem. An exploration of a design for multiple clock cycles will produce a symbolic execution tree where after each clock tick, each leaf node becomes a root node for a symbolic execution tree. These root or leaf nodes of symbolic execution trees are processor states, representing the processor or hardware design at a clock-cycle boundary.

Related to the software-style symbolic execution of a hardware design, but fundamentally different in spirit are hardware simulation techniques. Simulation is a traditional method for testing and debugging hardware designs. The user feeds in a set of arguments as inputs and verifies that the output is as expected. Symbolic simulation is an extension of this where the inputs can be made symbolic to represent a range of possible concrete values. STAR [33] provides a bridge between random and directed testing by allowing for concolic simulation: varying degrees of symbolic or concrete simulation. The tool is successful in finding high branch coverage, but limited by space and time when it came to fully unrolling designs. In [37], the authors present a tool that provides fully automated verification for Verilog RTL. Their toolflow involves a translation from Verilog into C and a single-path, forward symbolic execution engine, PATH-SYMEX, that runs on an ANSCI-C translation of the original hardware design.

3 Running Example

cks: I don't know if we need this separate section, but right now we don't have a full explanation of the two queues example anywhere. This can be just placeholder till we figure that out.

The code for the top-level module is shown in Figure 1. The code for each queue module is shown in Figure 2. The code is an adapted implementation of the two queues schematic presented in [15].

4 Design

Sylvia works in two phases as shown in Figure 3: Preprocessing and Symbolic Execution. We start with an overview of Sylvia and then delve into the details, including a description of how Sylvia leverages the common characteristics of

```
module top (
    input clk,
    input [31:0] i_data,
    input i_irdy, o_trdy,
    output [31:0] o_data,
    output o_irdy, i_trdy
);

    wire [31:0] data;
    wire irdy, trdy;
    wire q1_is_empty, q1_is_full;
    wire q2_is_empty, q2_is_full;

    queue q1(.clk(clk),
             .write_data(i_data),
             .write_en(i_irdy),
             .read_en(trdy),
             .read_data(data),
             .is_empty(q1_is_empty),
             .is_full(q1_is_full));

    // Handshake signals
    assign irdy = ~q1_is_empty;
    assign trdy = ~q2_is_full;

    assign i_trdy = ~q1_is_full;
    assign o_irdy = ~q2_is_empty;

    queue q2(.clk(clk),
             .write_data(data),
             .write_en(irdy),
             .read_en(o_trdy),
             .read_data(o_data),
             .is_empty(q2_is_empty),
             .is_full(q2_is_full));

endmodule
```

Figure 1: Top Level Module

hardware designs to optimize search.

Preprocessing. As a first step, the Verilog RTL is parsed to build an abstract syntax tree (AST). Sylvia then performs a single AST traversal to build a flattened representation of the design's decision points, a bitstring that we call the *path-code*. This is a key part of the interface between the AST and execution engine used to guide exploration and inform which branches are to be explored in a given run of symbolic execution. This phase also involves a preliminary dependency analysis that discovers the ordering of the dependent submodules to be symbolically executed.

Symbolic Execution. cks: revisit The core phase of the symbolic execution engine is powered by AST traversals where each individual tree traversal corresponds to a new path through the design. This involves interpreting Verilog AST nodes as instructions to be executed and implementing a faithful symbolic semantics of Verilog in Python. The parallelism and hierarchical structure of hardware designs present unique challenges to development of a software-style symbolic execution engine. In section 4.2.5-4.2.8 we go into detail about each one and provide details on our solution.

As the exploration progresses, Sylvia pushes constraints to an SMT solver, Z3 [20], keeping track of what conditions have brought us down a particular path. At the end of each

```
module queue (
    input clk,
    input [31:0] write_data,
    input write_en, read_en,
    output [31:0] read_data,
    output is_empty, is_full
);


reg [31:0] content = 0;
reg [1:0] in_use = 0;

always @(posedge clk) begin
  if (write_en) begin
    content <= write_data;
    in_use <= 1;
  end

  if (read_en) begin
    in_use <= 0;
  end
end

assign read_data = (read_en) ? content : 0;
assign is_empty = in_use[0];
assign is_full = ~is_empty;

endmodule
```

Figure 2: Queue Module

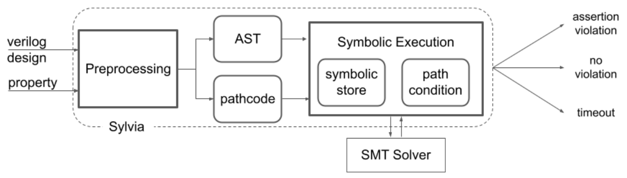

Figure 3: An overview of Sylvia's design.

run of the engine, we can solve for these constraints and use Sylvia to find security assertion violations.

Optimizations. Sylvia leverages the modular nature and the structural and behavioral parallelism inherent to many RTL designs to mitigate the path explosion problem and reduce the complexity of queries to the constraint solver. Optimization passes perform analyses to identify sets of execution states that can be merged and to restrict the search to areas of the design pertinent to the given security assertions.

4.1 Preprocessing

In the first phase, Sylvia builds the data structures necessary for symbolic execution. At the end of preprocessing Sylvia has an abstract syntax tree (AST) representation of the design, an accounting of the conditional dataflow points in the design, and a dependency graph of modules in the design showing the dataflow relationship between modules in which one module reads a register or wire written to by another module.

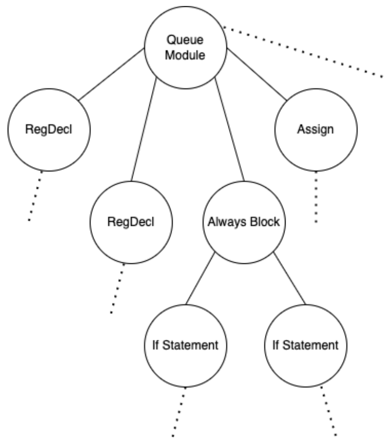

Figure 4: Snippet of Queue Module AST.

4.1.1 The Abstract Syntax Tree (AST)

An abstract syntax of a language keeps the core arithmetic expressions, boolean expressions, and commands that make up the language, while dropping the written syntax, such as the semicolon, that is used for reading and writing code. The AST of a Verilog design represents the complete design as a tree, with the root and internal nodes corresponding to commands and operators in the abstract syntax of Verilog, and leaf nodes corresponding to the signals, registers, ports, and literal values in the design. Once built, the AST serves as the authoritative representation of the design; Sylvia symbolically executes the design by traversing the AST. Figure 4 shows a snippet of the AST corresponding to the queue module in Figure 2. The dotted lines mean that additional child Nodes would be expected to stem from that particular parent Node but are elided for the purpose of brevity.

4.1.2 Conditional Dataflow

In the synthesizable subset of Verilog accepted by Sylvia, the language constructs that represent decision points in how data flows through the design are `if...else`, including nested `else ifs`, ternary if-then-else (`?...:`), `case`, and `for`. cks: maybe we need a small table that is: exact syntax of each construct; number of branches. The `if` statements always yield two branches (`else if` statements are treated as nested `ifs`). The number of branches in a `case` statement is determined by the number of alternatives, i.e., the number of possible cases. The number of branches in `for` loop is determined statically by the initial state and condition of the loop.

Sylvia does a preliminary traversal of the AST to assess the path complexity of the design. An accounting of the decision points in the design is taken and an upper bound on the number of possible paths is calculated. The *pathcode* is also built. The pathcode is a bitstring with one bit allocated for boolean condition to be evaluated across as part of a decision point in the design. An `if` is a decision point with two branches, where one condition is evaluated, and gets one bit in the pathcode. A `case` statement is a decision point with the number of branches determined by the number of alternatives. A `case` statement with N alternatives would be allocated N bits – one bit for each possible condition

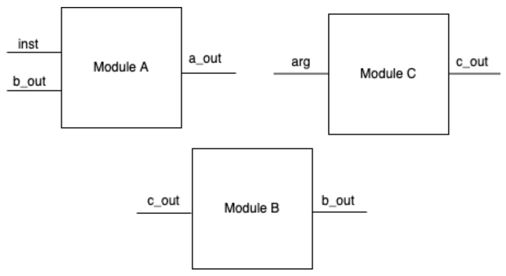

Figure 5: Illustration of dependencies between modules

evaluated during the exploration of the `case` statement.

Symbolic execution of a hardware design treats the data flow decision points in the design – those language constructs that can be synthesized into MUXes – as control flow execution points. In each run/iteration, Sylvia follows a single path of data flow. For example, an `if...else` statement would yield two paths to follow, one in which the condition is assumed to be true and one in which it is assumed to be false. kar: I moved this from the background, maybe can be weaved in better, but I think its helpful

4.1.3 *Preliminary Dependency Analysis*

The hierarchical and modular nature of hardware means that submodules are operating in parallel and the output signals of some modules become the input to other modules. Before symbolic execution can begin, Sylvia computes the dependency graph of modules. Each module of the design is represented as a node in the graph, and directed edges between nodes represent a read–write dependence. From this graph, an order of execution of modules in the design can be computed. Figure 5 shows a simple example, in which module A has an input that depends on module B's output, and B has an input that depends on C's output. Sylvia will explore the modules in the necessary order to resolve the dependency: C followed by B, and then B followed by A. On the other hand, independent modules for which there is no path from one to the other in the dependency graph, present an opportunity for optimization. We discuss the optimization in Section 4.3.1.

4.2 Symbolic Execution

kar: I think we should maybe have a different title for this? Or a different title for the section in the background, since they are both called Symbolic Execution The execution state of Sylvia is defined by the AST, pathcode, symbolic store, and path condition. Symbolic execution proceeds by traversing the AST, using the pathcode to guide the traversal. As execution progresses, Sylvia interprets each language construct in the AST using an extended semantics of Verilog that includes a notion of symbols to update the symbolic store and path condition.

If a security assertion is violated by any traversal, a satisfying solution to the associated path condition will produce concrete values that can be used to drive concrete execution (either in simulation or on a synthesized design) to the violating state.

4.2.1 *The Symbolic Store*

In traditional software-style symbolic execution, engines developed for software programs will maintain a symbolic store, or a mapping between program variables and symbolic expressions. In Sylvia, we maintain mappings between the signals, registers, and ports of the hardware design and their respective symbolic expressions. At the beginning of each clock cycle, all input ports in the top level module will be given fresh symbols in the symbolic store. As execution proceeds and different areas of the design are explored, the values held in the symbolic store will be updated according to the semantics of each Verilog statement that gets symbolically executed, ensuring clock-cycle accuracy at the end of every cycle.

4.2.2 *Defining a Symbolic Verilog Semantics*

We define an extended semantics to Verilog that introduces the notion of symbolic values. In this paper we will use α, β, and γ to represent symbols, although in practice each symbol is a unique alphanumeric string. Symbols are used to replace literal input values to a design. In the queue module, for example, the `write_data` input net can be initialized with the symbolic value α.

Symbolic execution proceeds by traversing the AST. At each node of the AST, Sylvia uses the extended semantics of Verilog to interpret the Verilog construct in the presence of symbols. As a replacement to literal values, symbols may appear on the right-hand side of assignment expressions or in the condition of a decision point. Nets (`wires`) and variables (`regs`) may hold symbolic values at any point of execution.

Declarations. The declaration of `regs` (state-holding variables), and input `wires` can be made symbolic. In this case the interpreted variable is given a fresh symbolic value in the symbolic store.

If and Case. If the bit in the *pathcode* corresponding to the particular `if` or `case` condition is set to 1, that means the condition is being evaluated as `True`. This conditional expression, which is either an identifier or some combination of operators over identifiers and literals, is parsed and interpreted such that all identifiers are replaced by their corresponding symbolic expressions held in the symbolic store. Once the condition is made symbolic, we attempt to simplify it, add it to the path condition, and begin exploration of the statement in the then block. If the bit in the *pathcode* were set to 0, we would do the same, but with the negation of the condition.

Blocking Assignment. Visiting blocking assignment statements results in updates to the symbolic store. When some `lhs` gets assigned some `rhs`, the entry in the symbolic store for the `lhs` is immediately overwritten to hold the symbolic expression residing in the symbolic store for the `rhs`.

Non-blocking Assignment. Non-blocking assignments similarly result in updates to the symbolic store. However, several non-blocking assignments can take place in parallel within the same clock cycle. kar: I'm not totally sure what to write for these little paragraphs without accidentally just explaining the challenge/solution

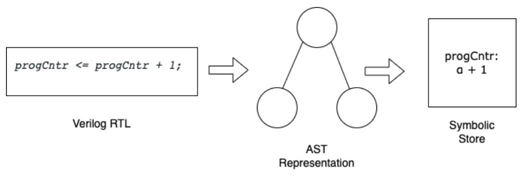

Figure 6: AST Traversals/Updating Symbolic Store

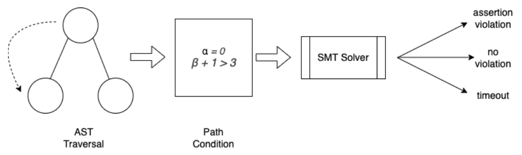

Figure 7: SMT Queries

4.2.3 *Traversing the AST*

Symbolic execution proceeds by traversing the AST. In each traversal Sylvia visits a subset of the nodes in the AST, corresponding to one path through the design. Sylvia uses the pathcode built during preprocessing to keep track of which nodes in the AST to visit during a particular traversal. Recall that each bit in the pathcode corresponds to one branch point in the design. If the bit is set to 1, Sylvia will follow the branch, otherwise it will not.

Every traversal begins from the top level module and proceeds downward. As execution proceeds Sylvia updates the symbolic store according to the extended semantics of Verilog. Figure 6 illustrates this flow where we have a line of Verilog RTL being built into its corresponding AST representation, and symbolically executed, resulting in an appropriate update to the symbolic store.

4.2.4 *Updating the Path Condition*

Each traversal begins with the path condition set to `True`. At each branch point in the AST traversal, Sylvia updates the path condition by pushing the constraints of the decision point to an instance of an satisfiability modulo theories (SMT) solver (e.g., [20]). Before proceeding down a branch, Sylvia checks whether the path condition is satisfiable. If it is, Sylvia continues. If it is not, the particular path is not feasible, Sylvia halts the traversal, and proceeds to the next path. At the end of a traversal, the satisfying solution to the path condition will provide concrete values to all input nets and variables that were initialized to be symbolic. If the path explored was one where an assertion violation was encountered, these concrete values are guaranteed to drive execution to the assertion-violating state. Figure 7 illustrates this flow where we have an exploration of the design resulting in an update to the path condition, which can be sent to the SMT solver to produce assertion violations.

4.2.5 *Continuous Assignments*

Continuous assignments are used to model combinational logic in Verilog and are denoted using the keyword `assign`. For example, lines 24-26 in Figure 2 are all continuous assignments. In a continuous assignment, anytime any of the signals on the right-hand side of the assignment change value, the right-hand side is re-evaluated, and the left-hand side wire is updated. This can be a problem for Sylvia: if the variables on the right-hand side of a continuous assignment change after Sylvia has already visited the assignment, Sylvia would fail to note the new value for the left-hand side. We destruct the continuous-assignment problem into four cases, and consider each one in turn, using snippets of Verilog to explain the problem and our solution.

1. **Case 1**. `assign` read_data = input_signal;

   When an input_signal, for example, is coming from a sensor, it can change at any time before, during, or after a clock cycle. Sylvia makes the assumption that all input signals to the top-level module remain fixed for the duration of a clock cycle. These top-level input signals get a fresh symbolic value at the start of each clock cycle, and keep that symbolic value for the duration of the clock cycle.

2. **Case 2**. `assign` read_data = content[31 : 0];

   The right-hand side, content, is a register and its value changes only at clock cycle boundaries. The symbolic expression associated with `assign` has to be re-evaluated at the end of the clock cycle to catch any changes to content within that clock cycle. Sylvia re-evaluates left-hand side of any continuous assignment one time at the end of the clock cycle. The new value will reflect any changes in the values of the right-hand side expression.

3. **Case 2a**. `assign` read_data2 = some_content[31 : 0];
   `assign` read_data = read_data2 & content;

   A variant of Case 2 requires additional consideration. In this Case, at the end of the clock cycle, Sylvia has to make sure to re-evaluate write_data2 before re-evaluating write_data. Sylvia does the dependency analysis between continually assigned signals to find the proper ordering.

4. **Case 3**. `assign` read_data = read_data & content;

   There is a circular dependency between the left- and right-hand side of the assignment. Circular dependencies involving only combinational logic are caught during a step of preprocessing and are out of scope. Running Sylvia with the "-G" flag enabled will output a dataflow graph and inform users of combinational loops present in their designs.

Having considered the four cases, we can describe the algorithm for handling continuous assignments within a module. First, check for any combinational loops during preprocessing. If present, Sylvia cannot proceed. Otherwise, during symbolic execution of a module, Sylvia first computes the ordered list of `always` blocks in the module and the ordered list of continuously assigned signals in the module, each ordered by dependency. Then Sylvia symbolically evaluates each `assign`, in order. Then, Sylvia symbolically executes each `always` block. As the block is executed Sylvia keeps track of a dirty bit for each signal, which gets set to 1 when the signal is updated. Once a single `always` block has been symbolically executed, Sylvia re-evaluates every `assign` statement, in order, for which the right-hand side involves a dirty signal. During this re-evaluation, Sylvia continues to track when signals become dirty. Finally, Sylvia moves on to symbolically execute the next `always` block.

4.2.6 Conditional Expressions

Conditional expressions in Verilog are expressed syntactically using the traditional ternary operator, where you have 3 expressions (however, they can be nested), with the first being a boolean expression. We have an example of a non-nested conditional expression on line 24 of Figure 1. Sylvia offers two modes of execution for handling these constructions, as there are tradeoffs associated with both options. The choice is ultimately left up to the user.

One option is for conditional expressions to be handled as control flow structures, meaning each decision point or boolean condition within a conditional expression will be allocated a bit in the *pathcode*. This also means that the conditions appearing in the predicate will appear in the path condition. For some designs that might include deeply nested conditional expressions, and may incur an exponential increase in the complexity of the search space.

The alternative is that the update is only seen in the symbolic store, not in the path condition. This means the signal whose right-hand side was a conditional expression would get a new symbolic expression in the symbolic store that is of the form:

```
If(cond, symbolic_store[ true_expr]
   symbolic_store[ false_expr] )
```

.

This would then need to be properly parsed and loaded into the Z3 solver as an ITE query. These symbolic expressions can get messy and complex very quickly.

4.2.7 Non-blocking Assignments

A third challenge is parallelism is present within and across always blocks. So in Figure 2, all of the statements on lines 14-20 will be will be executed in parallel every time the clock signal goes high. First, we need to ensure clock-cycle accuracy of the state updates, or that statements actually look like they executed in parallel when an observer inspects the symbolic store.

Second, it would be a naive approach to consider a symbolic execution tree that considered all possible orderings of these possible statements, even though all possible orderings of these statements are completely valid configurations of the design. We use a strategy akin to partial ordering reduction in model checking to reduce our total search space. Partial ordering reduction in model checking harnesses the commutative properties of concurrently executed transitions in systems that result in the same resultant states when executed in different orders [24]. In our case, these statements will result in the same symbolic state explored in different orders, so can just choose one to reduce the search space.

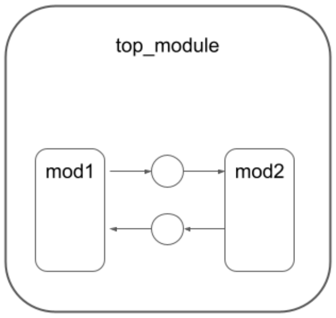

Figure 8: A simplified schematic with inter-module continuous assignments.

To summarize, the key takeaway is once we've picked an ordering, parallel statements explored individually by the engine need to appear as if they were symbolically executed in parallel. The symbolic execution engine is visiting each statement one by one and updating the store one by one, but at the end of each clock cycle, the internal state must be clock-cycle accurate. Sylvia maintains this by storing the previous value of registers at the beginning of the new clock cycle. When an always block is being explored, any reads need to come from the previous values that we just stored prior to beginning execution. This ensures that no matter what order we decide to explore the statements in, the resultant state at the end of the block is consistent at the next clock-cycle boundary.

4.2.8 Hierarchical Modules

So far, we have considered the issues of continuous assignments and non-blocking assignments within a single module. However, hardware designs are modular and hierarchical in nature. Any individual module may instantiate an arbitrary number of submodules, which may depend on each other in unknown ways. Consider the schematic of a design shown in Figure 8, in which a top module instantiates two submodules, mod1 and mod2. The output of mod1 becomes the input to mod2, after first passing through some combinational logic, shown in the Figure as a circle. For example, the output may be partially masked before it is fed into mod2. Similarly, the output of mod2 becomes the input to mod1. The signals connecting the two submodules are defined in the top-level module. If, during the symbolic execution of mod1, the output signal is updated, that needs to be reflected in the top-level module.

The queue example in Figures 1 and 2 provides a concrete example. In Figure 1, the `top` module instantiates two queue modules, q1 and q2. The signals of interest to us here are `data` and `read_data`. The first, `data` is used by q1 as the output signal `read_data`. In the body of q1 (Figure reflst:queuemodule), let's assume `read_en` is true and `read_data` gets updated on line 24 through a continuous assignment. Now from the top level module's view, the `data` wire should have the updated value. And this updated value should be reflected in `data` before Sylvia starts symbolic exploration of q2, where it is used as the input signal `write_data`.

The algorithm for handling inter-module continuous assignments and is similar to that for intra-module continuous assignments, but with inter-module dependencies taken into consideration. At the end of each exploration of a submodule, Sylvia takes a pass to recompute and synchronize all parameter and argument pairs where the parameter is an internal wire or register from the parent and the argument is an output of the submodule.

4.2.9 *Multiple Clock Cycles*

As mentioned in the background section, the symbolic execution of a hardware design means that symbolic exploration of design for one clock tick produces a tree where the root node represents the design in the reset set or initial state, and each each leaf node corresponds to a "next-state" for execution to begin with on the rising edge of the next clock cycle. To explore a design for two clock cycles would mean exploring the design again starting from each leaf node of the tree we constructed during clock cycle 0. Allowing for multi-clock cycle path means an explosion in complexity of the search space, however, it is important for applicability to real systems. For example, pipelined CPUs may have security vulnerabilities residing in the last stage of the pipeline, so Sylvia would need to run for at least the number of pipeline stages in order for the signals to propagate to through the whole pipeline and find the assertion violation. For a design like an AES module, we want to run exploration for as many cycles as there are "rounds."

At the end of each traversal, there is a state merging step, where all of inputs and wires get fresh symbols for the next cycle while stateful elements carry over their expressions from the previous cycle. As we explore the design, we keep a cache of the states we've seen stored by the specific module and *pathcode* pairing. This caching of "seen states" will be useful for the optimizations described in section 3.7.

4.3 Optimizations

The final phase of Sylvia includes several optimizations for producing assertion violations. Note that these are strategies that are made possible by our development of a hardware-oriented symbolic execution engine. Working within other software-oriented tools, like KLEE, leaves us with little control over how to explore the design and no space to experiment with hardware specific search techniques. Hardware designs contain large amounts of concurrency not present in software programs which are largely sequential in execution. For example, we may have parallel always blocks within modules that are executing at the same time or different modules that all execute in lock-step at the positive edge of a clock. This parallelism present in hardware allows for targeted searches and optimizations during Sylvia's exploration that we use to help mitigate the path explosion problem and increase performance.

In this section we will discuss three different optimizations Sylvia implements for improving the performance and decreasing the time needed to find security assertion violations.

4.3.1 *Redundant Submodules*

. As mentioned in the preprocessing section, when hierarchical modules are dependent on each other, we are faced with the challenge of figuring out the correct ordering for exploration. However, when the modules are independent or even duplicate instantiations of the same module there is room for reduction in the total search space.

Take a simple processor, for example, that might have memory, ALU and decoder submodules. There might be 20 paths through the memory, 10 paths through the ALU and 2 paths through the decoder, but all the paths start from a single root in the top-level module. Some of the sub-paths may be completely disjoint in terms of the signals they involve and some may not. This means that some of the time we will need to perform a serialization and impose an order in order to first resolve required dependencies, as discussed previously in section 3.1.4, but sometimes this may be unnecessary and submodules could be symbolically executed independently of each other. This is an opportunity for optimization in the case when there are redundant submodule instantiations. The main idea of the optimization is that we explore the redundant submodule once for each path. Then instead of re-exploring again for each duplicate module, we can statically merge in the symbolic store and path condition for the given *pathcode*.

4.3.2 *Cone of Influence Analysis*

. This optimization performs pruning of the state space at the block level. The general idea is that Sylvia will read in the expressions supplied in the security assertions, performs some dependency analysis over the signals in the assertions and then complete an AST traversal to determine which blocks contain information about the signals of interest or their dependencies. In particular, a mapping is constructed between always blocks and security critical signals, so that each always block may have a subset of security critical signals associated with it. Sylvia always reads the assertions in at the top level module, so we will know exactly what signals are of interest to us as we begin exploration of the design. After this initial pass, we only explore the blocks that have a non-empty set of signals associated with them in the final mapping.

4.3.3 *Parallel Always Blocks*

. Two always blocks are parallel if their sensitivity lists overlap, meaning that given some signal in the list goes high, both blocks of statements are executed in parallel. The key insight of this optimization is that if we can determine independence between two always blocks, then we shouldn't need to explore all orderings of blocks relative to one another. Instead, we can explore each always block independently, statically merging together the states and path conditions for a given *pathcode*. This optimization ends up being relatively straightforward because we make an assumption of no race conditions. Sylvia applies the optimization assuming the design has no race conditions, but will output a warning if it detects that there is race condition across two always blocks. A race condition is the result of a single signal appearing on the left-hand side of more than one non-blocking assignment within more than one parallel always block in a given clock tick. While you can imagine a good hardware engineer would avoid doing this, it is indeed legal Verilog to do so. When this does happen, per the official Verilog standard, the last nonblocking assignment wins. The assignment within the always block that appears later in the code will be the one that takes effect [5].

5 Implementation

The symbolic execution engine is fully implemented in Python and uses other open-source Python research tools. We use PyVerilog [40] to build the AST and use the Python Z3 API to find satisfying assignments for the path condition or assertion violations. The tool only supports hardware designs written in the synthesizeable subset of Verilog, not SystemVerilog. SystemVerilog includes elements that are not synthesizeable and are not yet supported by the AST building step. We also make the assumption of a synchronous reset.

5.1 Interface

The symbolic execution engine takes a hardware design written in Verilog expressed at the RTL level as a command line argument. Large designs that involve many files are passed to Sylvia as a directory. Sylvia then uses the PyVerilog toolkit to flatten the design and build the abstract syntax tree. Security assertions are embedded into these hardware designs in order for counterexamples to be produced by the tool. Sylvia performs all AST traversals in a depth-first order. Exploration of the AST continues accordingly for however many clock cycles were specified by the user at run time. If the "clock-cycle" argument is set to 3, for example, that means each path that Sylvia will explore will be 3 complete clock cycles of the design.

5.2 Translation

As mentioned previously, Sylvia interpret Verilog AST nodes as instructions to be executed.The AST is made of a tree of individual nodes where each node is either a module, declaration, statement or expression. Sylvia was parses information held in PyVerilog structures into symbolic expressions to be held in the symbolic store and implements a small library of functions to simplify and evaluate symbolic Verilog expressions. In addition to processing AST data to be loaded into our symbolic store, Sylvia parses and loads information from the AST into Z3 objects to represent the path condition.

5.3 Translating Security Assertions

For one experiment in which we compare Sylvia's efficacy to existing hardware verification workflows, we implement the security assertions in the subset of SystemVerilog that is supported by EBMC. This was done by manual rewriting.

At the time of this paper's publication, PyVerilog only provides support for Verilog constructs, not SystemVerilog. The biggest difference between Verilog and SystemVerilog are the additional capabilities included in SystemVerilog for the testing and verification of hardware designs. This includes a Hardware Verification Language used to specify properties and make assertions about designs. For example, special functions like "$past" which provides the value of a signal from a previous clock cycle. This language also provides operators not present in the base Verilog language such as the "|->" (non-overlapping implication) operator, which means if the expression on the LHS is true, then the RHS is true in the same cycle or the "##**n**" (delay) operator, which can be used to say that some property should only be true after **n** clock cycles.

To support the SystemVerilog Assertion language in Sylvia, we used a combination of custom macros and manual translation to port the semantics of the SystemVerilog security assertions into a format parseable by the PyVerilog front-end. We defined an `assert` directive that takes the condition for the assertion as an argument and is a wrapper around a single if-statement. If the assertion condition fails, we execute a system call and log the assertion failure.

In some cases we had to modify the hardware design itself. For example, to support the "$past" operator we had to add signals to the design to hold the previous values of whatever signals of interest were being accessed within a particular property. If a property included an implication or other unsupported operator, we manually rewrote it so that it would be expressing the same logic using Verilog syntax.

6 Evaluation

We evaluate Sylvia over five open-source designs, including CPU and SoC designs, to study its viability as a platform for security verification. Our evaluation considers the following questions: 1) Can Sylvia produce assertion violations for buggy and vulnerable designs? 2) How does Sylvia compare to other hardware verification workflows? 3) What gains does Sylvia get from the optimizations described in Section 4.3?

6.1 Dataset and Experimental Setup

We collected five designs and 84 security critical assertions from several sources. We downloaded the Security Property/Rule Database available on TrustHub [23] [22]. The database includes 30+ designs and over 100 security-critical properties, however, Sylvia only supports designs written in Verilog at the register transfer level. This left us with three Verilog designs: 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 (TRNG). For each of these designs, the database included 9, 2, and 2 security properties, respectively.

We also evaluated Sylvia with the buggy SoC used at the 2018 Hack@DAC hardware security competition [3]. Some of these bugs were inserted manually by the organizers of the competition and others were native to the design [21]. Using the English description of the properties, as well as the walkthrough of the test case generation in the RTL-ConTest paper [35] we were able to reverse engineer assertions for 26 of these bugs for use with our tool.

We also collected 31 security-critical bugs of the OR1200 processor from two prior papers, SPECS [26] and SCIFinder [44] and 70 security assertions developed for the OR1200 processor collected from SPECS [26], Security Checkers [8], SCIFinder [44], and Transys [45].

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. All counterexamples generated by Sylvia were successfully replayed in simulation.

6.2 Finding Assertion Violations

To evaluate Sylvia'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. From our dataset we had ground truth for two of the designs: the Hack@DAC SoC and the OR1200. For the OR1200, we restricted execution to the processor core, and for the Hack@DAC SoC we restrict execution to the particular IPs pertinent to the security assertions. Each design had 31 known bugs, and a summary of the types of bugs can be found in Table 1.

In these experiments, Sylvia begins in the reset state with all input signals made symbolic, and execution continues until an assertion violating state is found. Table 2 summarizes the results. A full table describing each bug and the time to find it is available in Sylvia's public-facing repo [1].

Sylvia finds 27 of the 31 bugs in the OR1200. Of the four missed, one did not have a property that covered it and one was outside the processor core. The third bug was related to the correct instruction being executed and would allow an attacker to carry out a ROP exploit by early kernel exit. The fourth bug was related to control flow and will incorrectly set the link register, which is used when the processor returns from function calls.

Sylvia finds 21 of the 31 bugs in the Hack@DAC SoC. Half of the bugs Sylvia misses were in the cryptography units or related to . For example, one bug Sylvia misses is for a non-functioning cryptography module and another is for an insecure hash function. Sylvia also misses bugs in the Hack@DAC SoC that are memory access related.

Bug Type	# Bugs	# Props	# Bugs Found
Control Flow Related	8	13	6
Exception Related	21	38	18
Memory Access Related	28	13	10
Correct Execution	4	3	1
Correct Results	14	29	13

Table 1: Bug Summary

Design	# Bugs	# Bugs Found	Avg Time (sec)
Hack@DAC SoC	31	21	221
OR1200	31	27	75

Table 2: Known Bugs

6.3 Effects of Optimizations

To evaluate the effectiveness of the optimizations, we use Sylvia to explore the TrustHub designs with and without optimizations enabled. (The larger Hack@DAC and OR1200 designs were too large to explore without the optimizations.) We compare four cases: *Baseline*, with no optimizations enabled; *Parallel*, with only a single ordering of `always` blocks explored, *Redund*, with redundant modules explored only once; and *COI*, with cone-of-influence analysis completed before exploration. Each case is cumulative, for example, in the Redund case the Parallel optimization is enabled as well. Table 3 shows the results. For each case, we provide the absolute runtime in seconds and the decrease in runtime compared to the prior case. For example, for the first design Sylvia runs 15 times faster with the Parallel optimization compared to the baseline. Overall, the optimizations decrease Sylvia's runtime by 98%–99%.

Table 4 shows the breakdown of time spent making queries to the SMT solvers and the impact of the optimizations on this. Again, optimizations are cumulative. The third optimization, in which the state space is pruned based on the signals in the security assertions, provides the biggest improvement in time spent querying the solver. This is not surprising because we waste less time exploring irrelevant blocks that might contain an arbitrary number of decision points. Before we take each branch, we would need to query the solver to ensure it's feasible and after the COI pruning we will only be making queries to the solver when branching in blocks of interest to the security critical signals or when we are solving for the counterexample.

6.4 Comparison to Existing Workflows

Current workflows for hardware verification typically include some form of model checking technology. In this section, we evaluate Sylvia's efficacy against a research model checking tool, EBMC [31] and a commercial hardware model checking tool, Cadence [2]. We compare their results when run with designs and security properties taken from the TrustHub database. The results reported in the Cadence column are derived from waveforms provided as part of the TrustHub database. The checkmark means a counterexample was found. The **X** means no counterexample was found. The ? means we have no information on the results.

For the three designs we collected from TrustHub, we did not have access to ground truth information but compared how each of the three tools performed for each property. The results are summarized in Table 5 . Note that the last design is a provably secure true random number generator [39].

Compared to *Coppelia*, another tool that leverages symbolic execution for security validation, we see performance advantages. We did not run *Coppelia* as part of our experiments, they report that most (62 %)) of exploits can be generated within 15 minutes, and several bugs can be found within 2-4 minutes. However, this is when they have a specific Verilator [4] compiler flag turned on that is necessary for good simulation performance, but performs unknown optimizations that may remove security critical signals from the C++ representation. kar: I don't know if this is the right place

Design & Property	Baseline (sec)	Parallel (sec)	Parallel (% dec)	Redund (sec)	Redund (% dec)	COI (sec)	COI (% dec)	Overall (% dec)
MC68HC11 SPI-1	126.86	8.46	93.33 %	1.86	77.90 %	0.09	94.73 %	99.92 %
MC68HC11 SPI-2	150.03	15.21	89.86 %	2.21	85.44 %	0.12	94.52 %	99.91 %
openMSP430-1	450.42	220.18	51.11 %	50.14	77.23 %	3.62	92.78 %	99.19 %
openMSP430-2	371.15	150.39	59.48 %	38.87	74.15 %	4.19	89.22 %	98.87 %
CrypTech TRNG-1	42631.80	21636.16	50.25 %	5335.97	75.37 %	679.13	87.27 %	98.40 %
CrypTech TRNG-2	42689.13	21890.16	48.74 %	6289.40	71.27 %	650.13	89.66 %	98.47 %

Table 3: Effects of Optimizations

Design & Property	Baseline		Parallel		Redund		COI	
	solver (sec)	% total	solver (sec)	% total	solver (sec)	% total	solver (sec)	% total
MC68HC11 SPI-1	96.47	76.25 %	6.36	75.25 %	1.29	69.17 %	0.05	57.78 %
MC68HC11 SPI-2	125.23	83.47 %	10.13	66.61 %	1.99	85.76 %	0.09	76.97 %
openMSP430-1	275.13	61.08 %	172.13	78.18 %	34.14	68.08 %	2.62	72.56 %
openMSP430-2	282.22	76.04 %	122.19	81.25 %	25.19	64.80 %	3.29	78.53 %
CrypTech TRNG-1	27535.60	64.59 %	15679.76	72.47 %	4068.91	76.25 %	435.80	64.17 %
CrypTech TRNG-2	27678.89	64.83 %	15900.76	72.64 %	3999.91	63.59 %	512.37	78.81 %

Table 4: Time spent in the SMT Solver and the Effect of Optimizations

Design & Property	**Sylvia**	**EBMC**	**Cadence**
MC68HC11 SPI - 1	✓	✓	✓
MC68HC11 SPI - 2	✓	**X**	✓
MC68HC11 SPI - 3	✓	**X**	✓
MC68HC11 SPI - 4	✓	**X**	✓
MC68HC11 SPI - 5	✓	**X**	✓
MC68HC11 SPI - 6	✓	✓	✓
MC68HC11 SPI - 7	✓	✓	✓
MC68HC11 SPI - 8	✓	✓	?
MC68HC11 SPI - 9	✓	✓	✓
openMSP430 - 1	✓	✓	✓
openMSP430 - 2	✓	**X**	✓
CrypTech TRNG - 1	**X**	**X**	**X**
CrypTech TRNG - 2	**X**	**X**	**X**

Table 5: TrustHub and Model Checking Tools

for this discussion. I also should revisit

7 Related Work

Model Checking Current practice in hardware validation typically involves a combination of simulation-based testing and static analysis techniques like model checking. Model checking can formally verify that a program satisfies a given set of specifications where these specifications are given to the model checker, typically in the form of a temporal logic formula [18]. The general verification procedure is an exhaustive search of the design space. In practice, bounded model checking is used in which the verification explores the design for up to k time steps [17]. Symbolic Trajectory Evaluation is a lattice-based model checking technique developed and used by Intel [9], and more recent work has brought this closer to the word-level [14]. The goal of STE is to formally verify properties of a sequential system over bounded-length trajectories using a modified form of 3-valued symbolic simulation [16].

Dynamic Symbolic Execution Dynamic symbolic execution is the term for the broader class of symbolic execution techniques that combines concrete and symbolic execution with the goal of trying to mitigate some of the challenges that come along with symbolic execution in software like the path explosion problem, constraint solving, and memory modeling. In particular, two distinct classes of dynamic symbolic execution techniques have emerged: Concolic Testing and Execution-Generated Testing (EGT) [12]. CREST [10], SAGE [25], and CUTE [38] are examples of concolic testing engines. KLEE [11], MAYHEM [13], and angr [29] are EGT engines and exemplify what it means to be a software-style symbolic execution engine, like Sylvia.

Recently it has been demonstrated that these methods can be successful in hardware verification workflows. RTLConTest [35] and [41] are examples of concolic testing engines developed for the security verification of hardware designs. Coppelia [43] is a hardware-oriented backward symbolic execution engine built on top of KLEE for RTL designs translated to C++.

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* [32]. The HyperFuzzer expands upon the *mux control coverage* metric to develop coverage measures specifically driven by certain adversarial behavior [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 [28]. HeteroFuzz is a fuzzing tool that targets heterogeneous applications running on FPGA accelerators and is able to extract multi-dimensional guidance feedback from accelerator relevant metrics [42].

Information Flow Tracking In a hardware context, information flow refers to the transfer of information between different signals. Information flow tracking is a verification method that studies how information flows through a hardware design to make statements about security [7] [27]. A hardware design can be instrumented with tracking logic to capture timing [6] or data flow information. This method can provide strong security guarantees, for example, demonstrating leakage of secret key data to undesired output signals.

8 Conclusion

We have presented Sylvia, a software-style symbolic execution engine for security validation that requires no translation to a higher level language. Sylvia finds and generates counterexamples for 48 known bugs across the OR1200 CPU and PULPissimo SoC. Sylvia also takes advantage of the parallelism and modularity inherent to hardware to identify sets of execution states that can be merged and restrict the search to areas of the design pertinent to the given security assertions, and this results in a 98%–99% decrease in runtime.

9 Appendix

REFERENCES

[1] Anonymized for peer review.

[2] Cadence verification. [Online]. Available: https://www.cadence.com/en_US/home/tools/system-design-and-verification.html

[3] "Hack@DAC 2018 SoC," https://github.com/seth-lab-tamu/hackdac-2018-soc, accessed: 2022-02-15.

[4] "Verilator." [Online]. Available: https://www.veripool.org/wiki/verilator

[5] "IEEE Standard for Verilog hardware description language," *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, pp. 1–590, 2006.

[6] A. Ardeshiricham, W. Hu, and R. Kastner, "Clepsydra: Modeling timing flows in hardware designs," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2017, pp. 147–154.

[7] A. Ardeshiricham, W. Hu, J. Marxen, and R. Kastner, "Register transfer level information flow tracking for provably secure hardware design," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, 2017, pp. 1691–1696.

[8] M. Bilzor, T. Huffmire, C. Irvine, and T. Levin, "Security checkers: Detecting processor malicious inclusions at runtime," in *2011 IEEE International Symposium on Hardware-Oriented Security and Trust*, 2011, pp. 34–39.

[9] R. E. Bryant, D. L. Beatty, and C.-J. H. Seger, "Formal hardware verification by symbolic ternary trajectory evaluation," in *Proceedings of the 28th ACM/IEEE Design Automation Conference*, ser. DAC '91. New York, NY, USA: Association for Computing Machinery, 1991, p. 397–402. [Online]. Available: https://doi.org/10.1145/127601.127701

[10] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," in *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, 2008, pp. 443–446.

[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] C. Cadar and K. Sen, "Symbolic execution for software testing: Three decades later," *Commun. ACM*, vol. 56, no. 2, p. 82–90, feb 2013. [Online]. Available: https://doi.org/10.1145/2408776.2408795

[13] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, ser. SP '12. USA: IEEE Computer Society, 2012, p. 380–394. [Online]. Available: https://doi.org/10.1109/SP.2012.31

[14] 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. [Online]. Available: https://doi.org/10.1007/s10703-017-0268-9

[15] S. Chatterjee, M. Kishinevsky, and U. Y. Ogras, "xmas: Quick formal modeling of communication fabrics to enable verification," *IEEE Design Test of Computers*, vol. 29, no. 3, pp. 80–88, 2012.

[16] 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.

[17] E. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded model checking using satisfiability solving," vol. 19, no. 1, p. 7–34, 2001. [Online]. Available: https://doi.org/10.1023/A:1011276507260

[18] 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.

[19] C. N. Coelho and H. D. Foster, *Assertion-Based Verification*. Boston, MA: Springer US, 2004, pp. 167–204. [Online]. Available: https://doi.org/10.1007/1-4020-2530-0_5

[20] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[21] 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

[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] C. Flanagan and P. Godefroid, "Dynamic partial-order reduction for model checking software," *SIGPLAN Not.*, vol. 40, no. 1, p. 110–121, jan 2005. [Online]. Available: https://doi.org/10.1145/1047659.1040315

[25] P. Godefroid, M. Y. Levin, and D. Molnar, "Sage: Whitebox fuzzing for security testing: Sage has had a remarkable impact at microsoft." *Queue*, vol. 10, no. 1, p. 20–27, jan 2012. [Online]. Available: https://doi.org/10.1145/2090147.2094081

[26] 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 *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 517–529. [Online]. Available: https://doi.org/10.1145/2694344.2694366

[27] W. Hu, A. Ardeshiricham, M. S. Gobulukoglu, X. Wang, and R. Kastner, "Property specific information flow analysis for hardware security verification," in *Proceedings of the International Conference on Computer-Aided Design*, ser. ICCAD '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3240765.3240839

[28] J. Hur, S. Song, D. Kwon, E. Baek, J. Kim, and B. Lee, "Difuzzrtl: Differential fuzz testing to find CPU bugs," in *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 2021, pp. 1286–1303. [Online]. Available: https://doi.org/10.1109/SP40001.2021.00103

[29] 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.

[30] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, p. 385–394, Jul. 1976. [Online]. Available: https://doi.org/10.1145/360248.360252

[31] D. Kroening and M. Purandare. EBMC: The enhanced bounded model

Bug No.	Description of Bug	Bug Found
b01	Privilege escalation by direct access	✓
b02	Privilege escalation by exception	✓
b03	Privilege anti-de-escalation	✓
b04	Register source redirection	✓
b05	Register source redirection	✓
b06	ROP by early kernel exit	**X**
b07	Disable interrupts by SR contamination	✓
b08	EEAR contamination	✓
b09	EPCR contamination on exception entry	✓
b10	EPCR contamination on exception exit	✓
b11	Code injection into kernel	✓
b12	Selective function skip	✓
b13	Register source redirection	✓
b14	Disable interrupts via micro arch	✓
b15	l.sys in delay slot will enter infinite loop	✓
b16	l.macrc immediately after l.mac stalls the pipeline	**X**
b17	l.extw instructions behave incorrectly	✓
b18	Delay Slot Exception bit is not implemented in SR	✓
b19	EPCR on range exception is incorrect	✓
b20	Comparison wrong for unsigned inequality with different MSB	✓
b21	Incorrect unsigned integer less-than compare	✓
b22	Logical error in l.rori instruction	✓
b23	EPCR on illegal instruction exception is incorrect	✓
b24	GPR0 can be assigned	✓
b25	Incorrect instruction fetched after an LSU stall	**X**
b26	l.mtspr instruction to some SPRs in supervisor mode treated as l.nop	✓
b27	Call return address failure with large displacement	**X**
b28	Byte and half-word write to SRAM failure when executing from SDRAM	✓
b29	Wrong PC stored during FPU exception trap	✓
b30	Sign/unsign extend of data alignment in LSU	✓
b31	Overwrite of ldxa-data with subsequent st-data	✓
b32	Address range overlap between peripherals SPI Master and SoC	✓
b33	Addresses for L2 memory is out of the specified range	✓
b34	Processor assigns privilege level of execution incorrectly from CSR	✓
b35	Register that controls GPIO lock can be written to with software	✓
b36	Reset clears the GPIO lock control register	✓
b37	Incorrect address range for APB allows memory aliasing	✓
b38	AXI address decoder ignores errors	✓
b39	Address range overlap between GPIO, SPI, and SoC control peripherals	✓
b40	Incorrect password checking logic in debug unit.	✓
b41	Advanced debug unit only checks 31 of the 32 bits of the password	✓
b42	Able to access debug register when in halt mode	✓
b43	Password check for the debug unit does not reset after successful check	✓
b44	Faulty decoder state machine logic in RISC-V core results in a hang	✓
b45	Incomplete case statement in ALU can cause unpredictable behavior	✓
b46	Faulty logic in the RTC causing inaccurate time calculation for security-critical flows	✓
b47	Reset for the advanced debug unit not operational	✓
b48	Memory-mapped register file allows code injection	**X**
b49	Non-functioning cryptography module causes DOS	—
b50	Insecure hash function in the cryptography module	—
b51	Cryptographic key for AES stored in unprotected memory	—
b52	Temperature sensor is muxed with the cryptography modules	—
b53	ROM size is too small preventing execution of security code	**X**
b54	Disabled the ability to activate the security-enhanced core	—
b55	GPIO enable always high	✓
b56	Unprivileged user-space code can write to the privileged CSR	**X**
b57	Advanced debug unit password is hard-coded and set on reset	✓
b58	Secure mode is not required to write to interrupt registers	✓
b59	JTAG interface is not password protected	✓
b60	Output of MAC is not erased on reset	✓
b61	Supervisor mode signal of a core is floating preventing the use of SMAP	**X**
b62	GPIO is able to read/write to instruction and data cache	**X**

Table 6: Bug Descriptions. The bugs above the line were present in the buggy OR1200 processor. The bugs below the line were present in teh buggy HACK@DAC SoC. The X means Sylvia was not able to find the bug and the — means the property was not able to be recreated

checker. [Online]. Available: http://www.cprover.org/ebmc/

[32] K. Laeufer, J. Koenig, D. Kim, J. Bachrach, and K. Sen, "Rfuzz: Coverage-directed fuzz testing of rtl on fpgas," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2018, pp. 1–8.

[33] L. Liu and S. Vasudevan, "Star: Generating input vectors for design validation by static analysis of rtl," in *2009 IEEE International High Level Design Validation and Test Workshop*, 2009, pp. 32–37.

[34] B. Mayfield and T. Baird, "STP: A simple theorem prover for IBM-PC compatible computers," in *Proceedings of the 1990 ACM SIGSMALL/PC Symposium on Small Systems*, ser. SIGSMALL '90. New York, NY, USA: Association for Computing Machinery, 1990, p. 98–105. [Online]. Available: https://doi.org/10.1145/99412.99439

[35] 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.

[36] S. K. Muduli, G. Takhar, and P. Subramanyan, "Hyperfuzzing for SoC security validation," in *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2020, pp. 1–9.

[37] 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.

[38] K. Sen, D. Marinov, and G. Agha, "Cute: A concolic unit testing engine for c," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-13. New York, NY, USA: Association for Computing Machinery, 2005, p. 263–272. [Online]. Available: https://doi.org/10.1145/1081706.1081750

[39] B. Sunar, W. J. Martin, and D. R. Stinson, "A provably secure true random number generator with built-in tolerance to active attacks," *IEEE Transactions on Computers*, vol. 56, no. 1, pp. 109–119, 2007.

[40] S. Takamaeda-Yamazaki, "Pyverilog: A python-based hardware design processing toolkit for verilog hdl," in *Applied Reconfigurable Computing*, ser. Lecture Notes in Computer Science, vol. 9040. Springer International Publishing, Apr 2015, pp. 451–460. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-16214-0_42

[41] 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. [Online]. Available: https://doi.org/10.1145/3441297

[42] Q. Zhang, J. Wang, and M. Kim, *HeteroFuzz: Fuzz Testing to Detect Platform Dependent Divergence for Heterogeneous Applications*. New York, NY, USA: Association for Computing Machinery, 2021, p. 242–254. [Online]. Available: https://doi.org/10.1145/3468264.3468610

[43] 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.

[44] R. Zhang, N. Stanley, C. Griggs, A. Chi, and C. Sturton, "Identifying security critical properties for the dynamic verification of a processor," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 541–554. [Online]. Available: https://doi.org/10.1145/3037697.3037734

[45] 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.