End-to-End Automated Exploit Generation for Validating the Security of Processor Designs

Rui Zhang
University of North Carolina at Chapel Hill
rzhang@cs.unc.edu

Calvin Deutschbein
University of North Carolina at Chapel Hill
cd@cs.unc.edu

Peng Huang
Johns Hopkins University
huang@cs.jhu.edu

Cynthia Sturton
University of North Carolina at Chapel Hill
csturton@cs.unc.edu

***Abstract*—This paper presents *Coppelia*, an *end-to-end* tool that, given a processor design and a set of security-critical invariants, automatically generates complete, replayable exploit programs to help designers find, contextualize, and assess the security threat of hardware vulnerabilities. In Coppelia, we develop a hardware-oriented backward symbolic execution engine with a new cycle stitching method and fast validation technique, along with several optimizations for exploit generation. We then add program stubs to complete the exploit. We evaluate Coppelia on three CPUs of different architectures. Coppelia is able to find and generate exploits for 29 of 31 known vulnerabilities in these CPUs, including 11 vulnerabilities that commercial and academic model checking tools can not find. All of the generated exploits are successfully replayable on an FPGA board. Moreover, Coppelia finds 4 *new* vulnerabilities along with exploits in these CPUs. We also use Coppelia to verify whether a security patch indeed fixed a vulnerability, and to refine a set of assertions.**

***Index Terms*—Symbolic Execution, Exploit Generation, Processor Security**

I. Introduction

This paper describes *Coppelia*, a tool to find, and generate software exploits for, bugs in hardware designs. Given a processor design and a set of security properties Coppelia systematically explores the design using symbolic execution, and if security violations are found Coppelia generates complete, *replayable* exploits comprising both a trigger and the payload. These exploits can help hardware designers analyze and contextualize the nature of the security threat.

Exploitable bugs in hardware designs present an enticing target to attackers [1], [2], [3], [4], [5]. Current practice in hardware design verification combines formal static analysis techniques with simulation-based testing, methods which can successfully find assertion violations, but which do little to differentiate bugs in the design from bugs in the assertion itself or bugs in the environment set-up. Nor do they provide any information to the designer about which bugs pose security risks and which likely do not. Determining the root cause for found violations typically involves inspecting waveforms displaying the state of the signals and registers during the clock cycles leading up to the violation. The process is difficult and often tedious and requires cooperation between security experts, hardware designers, and formal methods experts [6], [7]. With Coppelia we present an end-to-end solution that can automatically find bugs and demonstrate the security threat for exploitable bugs in hardware designs.

Within the software security community, symbolic execution is a powerful technique for automatically generating test cases to trigger security vulnerabilities [8]. It has a reputation for relative ease of use, and we believe it can be brought to the same level of utility and usability for hardware designs. The use of software-style symbolic execution for hardware designs has been proposed before [9], but not yet fully developed. We take the next steps toward doing so in this paper. Using open source, RISC CPUs to evaluate our methods, we show that out-of-the-box symbolic execution does not scale to full hardware designs, and we develop a hardware-oriented backward symbolic execution engine that can.

Two characteristics of hardware designs require rethinking the standard symbolic execution typically used in the software domain. The symbolic execution of a hardware design represents an exploration of the design for a single clock cycle, but hardware executes continuously, and security vulnerabilities may only become apparent many clock cycles after the initial state. Symbolic execution can never provide exhaustive coverage for systems with infinite execution trees [10]. As a further complication, the large state space of a modern processor design precludes joining redundant states during exploration, as is sometimes done for software designs [11].

Second, security properties developed for hardware designs capture the semantics of particular signals and their connecting logic. By contrast, security properties developed for software are applicable throughout a code base. For example, invalid- or missing-bounds checks occur throughout a software code base, and a symbolic execution engine that looks for such violations is likely to find more examples just by exploring more broadly and deeply. Compare this to, say, the security-critical property of some RISC architectures that the general purpose register `R0` should always be set to zero [12]. A violation of this property will occur only in that part of the design that touches the `R0` register. Finding such violations is akin to finding a needle in a haystack; if an exhaustive search is not possible, a strategy is needed to focus the search toward the target.

We propose in Coppelia a strategy of backward symbolic execution (Section II-D). Starting at the point of an assert statement, we symbolically execute the design backward searching for a path from an assertion-violating state back to the reset state. To handle symbolic execution across multiple clock cycles, we propose a cycle stitching method (Section II-D6)

that can generate a complete sequence of instructions that triggers a bug starting from the reset state. In comparison, state-of-the-art hardware verification tools such as Cadence IFV generate *intermediate* triggering input (Section IV-C). To make Coppelia efficient as a practical tool, we leverage cone of influence analysis and a search optimization, which we term *fast validation* (Section II-D4). As is typical for symbolic execution engines, Coppelia is sound, but not complete – the assertion violations found are true violations and the exploits returned are replayable on synthesized hardware, but Coppelia may fail to find violations that exist.

To better analyze and assess the security consequences of a found bug, we must move beyond the mere triggering of the bug to the generation of complete exploit programs that demonstrate a possible, concrete attack. Continuing with our example, producing the sequence of instructions that let `R0` be set to some non-zero value is useful in demonstrating that a true bug exists; but generating an exploit that uses the non-zero value to write arbitrary data to an arbitrarily chosen address can clarify the security consequences of the bug for hardware designers. This is important for informing prioritization decisions. It is not always apparent to the person verifying the design how severe a property violation may be. Security properties are reused across generations of an architecture, or even semi-automatically generated [12].

To extend a sequence of inputs beyond a bug trigger to a full exploit, we need an appropriate payload. We observe that hardware exploits differ based on the nature of the violated property, rather than the nature of the bug or its trigger. Therefore, in Coppelia we generate a payload *stub* for each class of property (e.g., memory address related). The stub extends a bug-triggering input into an exploit.

Coppelia is built on top of KLEE [13], a popular symbolic execution engine. We adapt KLEE for the symbolic exploration of hardware designs at the register transfer level (RTL) and implement the backward symbolic execution engine. We evaluate Coppelia on three processor designs: OR1200, PULPino, and Mor1kx, representing two different RISC architectures: OR1k and RISC-V. To measure Coppelia's efficacy against a ground truth, we use known vulnerabilities described in prior work [5], [12]. Coppelia finds and generates complete exploits for 29 of the 31 known vulnerabilities on the evaluated processors, including 11 vulnerabilities that Cadence IFV does not find. All of the generated exploits are replayable on an FPGA board. In addition, using existing security properties, Coppelia finds four **new** bugs in these processors, and generates replayable exploit programs.

In summary, this paper presents the following contributions: (1) We design and implement the first tool, to the best of our knowledge, to generate software exploits, including both a trigger and the payload, of processor designs; (2) We develop the hardware-oriented backward symbolic execution engine to enable a targeted search through the hardware design space for rare assertion violations; (3) We evaluate the efficacy and performance of Coppelia on three processors from two different architectures. Our tool found four new security bugs in two processors, and generated the exploits for these bugs; (4) We demonstrate that security properties developed for one architecture can be usefully applied to new architectures.

II. Design

We first provide an overview of the three phases of Coppelia: preprocessing, building a trigger, and adding the payload. We then describe each phase in detail in the following sections. Figure 1 shows the workflow of Coppelia.

We are targeting vulnerabilities in a processor design that are exploitable, post-deployment, by software. We assume the attacker does not modify the processor design, but is capable of finding vulnerabilities that exist within the design. Post deployment, we assume the attacker is able to send network packets, execute a particular sequence of instructions, or both on the target machine.

A. Overview of Coppelia

Coppelia takes as input an HDL implementation of a hardware design and a set of security-critical assertions.

Preprocessing. To begin, Coppelia translates the RTL hardware design from an HDL implementation to C++. We use the Verilator tool [14] for this step and can translate designs written in Verilog or SystemVerilog, although the basic approach would apply to other HDLs as well. Translating the RTL design to C++ allows us to take advantage of KLEE [13], a mature symbolic execution engine, and use it as the foundation of Coppelia. We discuss this step further in Section II-B. After translation, Coppelia adds the security-critical assertions to the generated testbench and compiles the newly translated design to LLVM bytecode using the Clang compiler [15].

Building a trigger. A vulnerability is defined as a processor state s_n in which a security-critical assertion is violated. Assertions are boolean-valued functions written over (a subset of) the state-holding elements of the processor. They encode desired security properties, and can express safety properties, but not liveness [16] or hyperproperties [17]. The goal is to find a sequence of inputs $i_0, i_1, \ldots, i_k$ that take the system from the initial state s_0 to the violating state s_e. Coppelia builds the sequence backwards, first finding input i_k then i_{k-1} and so on. Each input is found by symbolically exploring the processor.

Adding the payload. To better contextualize and analyze the security threat, Coppelia goes beyond triggering the vulnerability. It adds a program stub to complete the exploit. These program stubs are generated according to the category of the security-critical assertion violated. We describe this step in detail in section II-F.

B. Preprocessing: Transcompiling RTL to C++

In the first phase, we use Verilator [14] to translate the RTL Verilog to logically equivalent C++ code. Verilator is an open source Verilog simulator. It compiles the synthesizable subset of Verilog into cycle-accurate C++ or SystemC code.

Verilator starts with a preprocessing step in which it propagates parameters, determines expression widths, eliminates dead code, unrolls loops, and inlines modules and tasks. It also

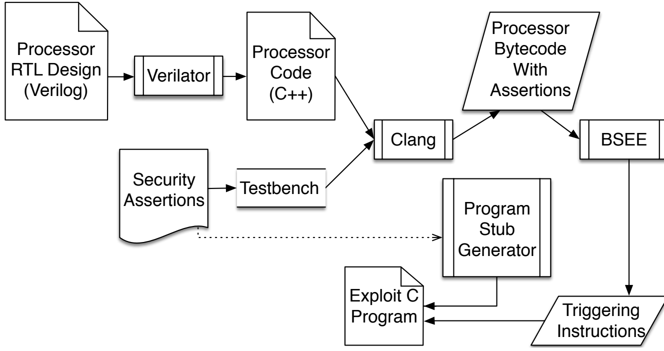

Fig. 1: Workflow of Coppelia. The process labeled BSEE is the backward symbolic execution engine.

eliminates any possible 3-state by replacing don't-care values (X) with random values. Next, Verilator does the translation. The translation of Verilog blocking statements is straightforward as these statements are semantically similar to that of straight-line C++ code. On the other hand, the translation of Verilog non-blocking statements requires additional analysis as there is no semantic equivalent in C++ to the simultaneous execution of multiple statements. Verilator imposes an order on non-blocking assignments and introduces temporary C++ variables so that the resulting straight-line C++ code produces a faithful simulation of the Verilog's behavior at each clock cycle boundary. Finally, Verilator cleans up the code, corrects expression widths, and outputs the result in C++ [18].

In the resulting C++ code, each class corresponds to a module in the Verilog code. The hierarchy of the C++ classes matches the hierarchy of Verilog modules. The interface to the top C++ class is an `eval()` function that calls the functions inside each class necessary to simulate the processor design for a single clock transition. There are two major loops inside the `eval()` function: the initialize loop and the main change loop. The initialize loop executes the initialization statements and propagates the initial values through the design. The main change loop executes circuit logic and propagates value changes to each module (a C++ class). Two calls to the eval() function represent a single clock cycle.

The input signals remain stable during a single execution of the `eval()` function, meaning inputs will only change at clock tick boundaries. This assumption ensures the circuit model converges and improves the efficiency for the code analysis.

C. Background, Notation, and Definitions

Before describing how we build the trigger, we review symbolic execution, introduce notation, and define the problem.

In standard forward symbolic execution input values are replaced with symbols that represent the set of possible values in the domain of the function. The symbolic exploration of a program can be represented by a tree $\mathscr{E}$. Each path through the tree represents a path of execution taken during the symbolic exploration. Each node represents a line of code in the program; the root node represents the entry point and the leaves represent an exit point. Associated with each node is the current program state – the valuation of variables – and a path condition. The path condition (π) for node n defines constraints over the program's input domain such that if the program is run with input values satisfying the constraints, execution would be driven down the path from root to n.

The symbolic exploration of a processor – achieved by symbolically exploring two consecutive calls to the `eval()` function – corresponds to one clock cycle of the design. The root node of the resulting tree represents the state of the processor at a clock-cycle boundary and each leaf of the tree represents a possible next-state of the processor. When referring to a processor state we are referring to a root or leaf node in the symbolic execution tree, not an internal node; the root and leaf nodes represent the processor at cycle boundaries.

We will refer to a tuple (n,i,π) associated with a symbolic exploration tree $\mathscr{E}$. The tuple defines a particular leaf node of interest n, the inputs i that would guide execution from the root node down the path to leaf node n, and the path constraints π associated with leaf node n. We also define a *test case* as a satisfying solution to a path constraint. A test case is one set of concrete input values that will drive the processor down the path associated with the path constraint.

The execution of multiple clock cycles in the processor is represented by multiple symbolic explorations of the design (see Figure 3). Each leaf node of a tree $\mathscr{E}_j$ becomes a root node for a tree $\mathscr{E}_{j+1}$ representing the next exploration of the design, i.e., the next clock cycle of the processor.

We aim to find a sequence of inputs that will take the processor from the reset state to an error state. We define the problem in terms of symbolic exploration trees.

Problem Statement. Given s_e, an error state of the processor in which a security-critical assertion is violated, find a sequence of symbolic exploration trees $\mathscr{E}_0,\mathscr{E}_1,\ldots,\mathscr{E}_k$, and for each tree a particular leaf node $n_0,n_1,\ldots,n_k$ such that

- The root node of the first tree $\mathscr{E}_0$ is the reset state of the processor,
- The leaf node n_k associated with tree $\mathscr{E}_k$ represents the error state of the processor, and
- The leaf node n_j associated with tree $\mathscr{E}_j$ can be *matched* to the root node of tree $\mathscr{E}_{j+1}$.

We say the leaf node of one tree can be *matched* to the root node of a second tree if and only if the nodes are compatible: concrete values are equal and constraints over symbolic values given in one node are mutually satisfiable with constraints over symbolic values given in the second node.

If the above requirements are satisfied then the sequence of path constraints $\pi_0,\pi_1,\ldots,\pi_k$ provided by the sequence of leaf nodes $n_0,n_1,\ldots,n_k$ define the sequence of inputs to the processor that will take the processor from an initial state to the error state.

D. Building the trigger: Backward Symbolic Execution

An error state that is M clock cycles away from the initial state will only be found after $2M$ iterations of the eval() loops. The search space for forward symbolic execution is

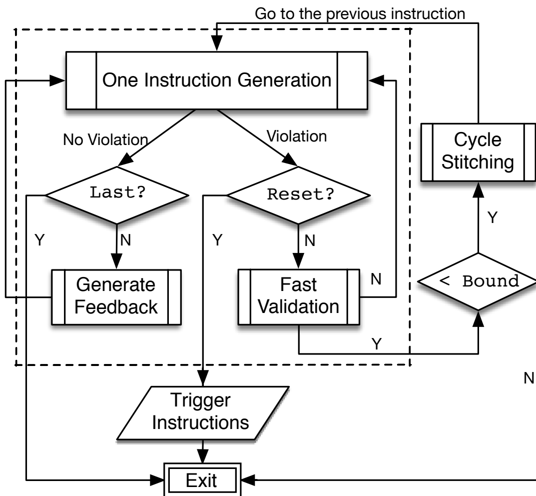

Fig. 2: Workflow of Backward Symbolic Execution

exponential in the number of loop iterations and becomes untenable for even small values of M. (See Section II-D8 for a discussion of the search complexity.)

The key insight of our work is that hardware is well suited to a backward search strategy for symbolic execution. The specificity of security assertions in hardware designs make them amenable to such a targeted search strategy, and the lack of dynamically linked libraries, pointers, and complex computation makes the backward strategy possible.

Rather than start at the processor's initial state and search forward, Coppelia uses backward symbolic execution to start at an error state and search backward. In the first iteration, the backward symbolic execution engine looks for a procesor state s that can reach an assertion failure in one clock cycle, given the right set of inputs. If such an s is found, the problem is ideally reduced: from finding a path of length M from the initial state to the error state to finding a path of length $M-1$ from the initial state to state s. The backward symbolic execution engine continues in this way, stepping back from the error state toward the initial state, one clock cycle at a time.

We cannot, in the general case, be sure that each iteration actually reduces the problem. An intermediate state s may not be reachable from the initial state, or we may find ourselves stitching together a path that has a loop and never converges toward the initial state. We introduce heuristics to help the backward symbolic execution engine identify unreachable states, loops, or paths that are not tending toward the initial state.

1) Backward Symbolic Execution Engine: We describe the workflow of our hardware-oriented backward symbolic execution engine (see Figure 2). In the following sections, we describe each step in detail.

1) **One Instruction Generation:** In the first iteration, the engine initializes input and internal signals to be symbolic values and explores the processor design for one complete clock cycle. In pipelined RISC processors, one clock cycle represents the completion of one instruction. In subsequent iterations, input signals are made symbolic, but internal signals may be partially constrained or concrete. (Sections II-D2 and II-D3.)
2) **Assertion Violation:** When the engine encounters an assertion violation, it produces a path constraint describing the precondition necessary to reach that error state. If the processor's reset state can satisfy the constraint, the backward symbolic execution engine is done. It outputs the trigger instruction(s) and Coppelia moves to the next phase: adding the payload.
3) **Fast Validation:** If the processor's reset state does not satisfy the current path constraint, the engine does a fast validation of the current intermediate state. This step uses heuristics to eliminate intermediate states that are less likely to bring the search closer to the reset state.
4) **Bound Checking:** If the current state passes the fast validation, the engine then checks whether the sequence of instructions generated so far exceeds a bound. The bound is a tunable parameter to the engine.
5) **Stitching Cycles:** If the length of the sequence is within the bound, the engine stitches the current state to the previously found state and continues on to the next iteration of the One Instruction Generation step.
6) **Feedback Generation:** When any of the preceding steps fail, the engine goes back to the prior iteration and, using feedback generated during prior runs, continues exploration in a new direction.

2) One Instruction Generation: In the first iteration, the backward symbolic execution engine starts the search for a security property violation from an unconstrained processor state. It sets both the input and the internal signals to symbolic values, and then explores the processor design until it reaches a state that violates the security property. If exploration completes and no assertion violation is found, Coppelia returns with a result of `no violation found`. Otherwise, the resulting exploration tree, $\mathscr{E}_k$, has a leaf node n_k that represents the error state (s_e) of the processor. Associated with that leaf node is the path condition π_k that describes the sufficient constraints on processor state and input signals such that the processor will move from the constrained state (s_{e-1}) to the error state in a single clock cycle. In addition to the constraints, the engine returns a satisfying solution to the constraints over input signals. These concrete input values will form the last instruction in the trigger sequence.

In the next iteration, the engine again starts the search from an unconstrained processor state. This time the engine is looking for s_{e-1}, a state that satisfies the constraints returned in the prior iteration, but not s_e. If such a state is found, the engine returns a path condition π_{k-1} and a satisfying solution to the constraints over the input signals. These concrete input values will form the penultimate instruction.

Iterations continue in this way, searching backward through trees $\mathscr{E}_k, \mathscr{E}_{k-1}, \ldots, \mathscr{E}_0$ until we reach the initial processor state. In the following sections we discuss the heuristics and optimizations we introduce to help the search converge toward an initial state.

3) Stateful Signals: A naive implementation of hardware oriented symbolic execution might make all variables of type `reg` symbolic because these internal signals can store state. However, the resulting exploration tree is too large. Using this set-up, we ran Coppelia for one clock cycle. After 24 hours it had generated over 1 million test cases – each is a different leaf node in the tree – but had not triggered any assertions.

We identify those signals that can be safely left concrete without affecting completeness of the search. First, `reg` signals are used in one of two ways in a hardware design: as part of sequential logic in which case they store state from a previous clock cycle, or as part of combinational logic in which case their value depends only on input signals in the current clock cycle. Using static analysis, we identify those signals which depend entirely (albeit, possibly indirectly) on input signals and do not make those symbolic in each iteration of exploration. Second, not all `reg` signals are relevant for a particular security property. Only those signals in the property's cone of influence are made symbolic. Section II-E3 describes the dependency analysis that Coppelia performs to identify which signals to make symbolic.

4) Fast Validation: At the end of each successful iteration j, the backward symbolic execution engine checks the following: are the constraints given in path condition π_j satisfied by the initial state? If so, Coppelia has found a successful trigger and moves on to the next phase, appending the payload.

If not, in order to steer the search toward the initial state, we introduce two rules to eliminate those intermediate states that are less likely to quickly lead back to the initial state. These rules form the fast validation step.

Empirically, we found that if the number of variables whose values are different from the initial state is small, we are more likely to be able to back track to an initial state. We set the number of differing variables to be:

$$diff((n_j,i_j,\pi_j),(n_0,i_0,\pi_0)) \leq \lfloor |s|/4 \rfloor + 1 \quad (1)$$

where (n_0,i_0,π_0) is the tuple associated with the initial tree $\mathscr{E}_0$, (n_j,i_j,π_j) is the tuple associated with an intermediate tree $\mathscr{E}_j$, *diff* calculates the number of different values between two tuples, and $|s|$ represents the number of internal symbolic variables. With this rule, at most a quarter of internal state variables may differ from their reset state.

The second rule targets loops that are preventing backward progress toward the initial state. We enforce that each new iteration should produce a tuple (n_j,i_j,π_j) that is not the same as any previously generated tuples:

$$(n_j,i_j,\pi_j) \notin \{(n_l,i_l,\pi_l) \mid j < l \leq k\} \quad (2)$$

If the values of internal signals are the same as ones already generated, we are not making any progress in this run and risk entering an infinite loop. Thus, if the generated (s_j,i_j,π_j)-tuple is a repeat, Coppelia will keep running until a different tuple is found.

5) Bound Checking: As a final heuristic, Coppelia uses bounded checking to counter the fact that the sequence of trees may never converge toward the initial state. We set a bound for the exploit length. If the trace of inputs generated so far exceeds the bound, Coppelia will exit with a message that it did not find an exploit within the bound.

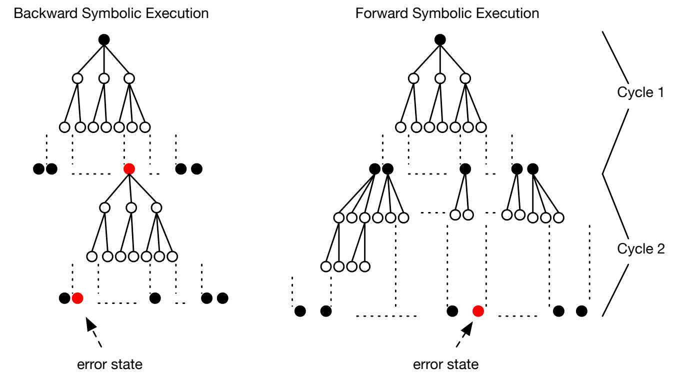

Fig. 3: Comparison of backward and forward symbolic execution for 2 clock cycles.

6) Stitching Cycles: If the length of the sequence is within the bound, we stitch the current clock cycle to the previous clock cycle and continue with the next iteration. The sequence of trees must be stitched together appropriately, making sure a leaf node of one tree correctly aligns with the root node of a tree previously generated.

Ideally, in order for the results of cycle $\mathscr{E}_j$ and cycle $\mathscr{E}_{j-1}$ to align, we need to replace the values of internal signals in node n_{j-1} with the path constraint π_j obtained in node n_j. This ensures completeness – we will not miss a possible test case. However, the complexity of this method is similar to forward symbolic execution (see Section II-D8). The more cycles we symbolically execute, the longer the path constraints will be and the more complicated the queries will be to the SMT solver. In Coppelia, we adopt a light-weight approach. The insight is that while each clock cycle is explored symbolically, the individual cycles can be stitched together using only concrete values. This sacrifices completeness for speed: after each iteration, we find satisfying solutions to a subset of the internal signals and use these conrete values to partially define the state to search for in the next iteration. This will no doubt lead us to miss some possible violating paths. In practice, we can iterate, incrementally replacing concrete values with constrained symbols if no assertion violations are found.

7) Feedback Generation: If the engine finishes exploring all paths and no violations are found and this is not the first iteration (Figure 2), it means a violation was found in previous runs but the engine chose a wrong path, either because of the fast validation, the light-weight stitching, or because it stopped exploring after finding one violation. In this case Coppelia will go back to the previous runs and continue the exploration. Coppelia generates a feedback to the engine including which instruction causes the violation and what test cases have been explored. When rerunning that instruction generation, Coppelia only explores the specific instruction and skips the test cases already explored.

8) Forward and Backward Symbolic Execution: In forward symbolic execution, in the first clock cycle, a tree with N_f leaves will be explored (the N_f black dots in the first layer of the symbolic execution tree on the right in Figure 3). In the second clock cycle, the tree must be explored again, once for each of the N_f leaves. Exploring forward M clock cycles has complexity $O(N_f^M)$.

The complexity for backward symbolic execution is $O(N_b \cdot M)$, where M is the number of cycles to execute. Note that the N_b here is larger than the N_f in forward symbolic execution because the internal signals are set to be symbolic values which increases the paths to explore. On the other hand, in general only $j \leq N_b$ paths are explored because exploration stops once the error state is found. This is illustrated on the left side of Figure 3.

E. Building the Trigger: Optimizations

Each iteration of the symbolic exploration of the processor is expensive. We introduce the following optimizations tailored for hardware designs to improve the speed.

1) Preconditioned Symbolic Execution: As an optimization, Coppelia constrains the opcodes to only instructions in the architecture to force the SMT solver to return legal instructions.

We also add constraints to support bit level representations. In KLEE, the minimum width supported is a byte. However, hardware signal widths are not necessarily byte multiples. Thus, we add constraints to inform the symbolic execution engine of the value range of such signals. For example, for a signal of width n, we constrain the value of the signal to be less than or equal to $(2^n - 1)$.

2) Path Selection Heuristic: We observe that if Coppelia is exploring the right processor instruction, it will find the vulnerability in a short time. However, it often takes a long time before Coppelia starts exploring the right instruction. (In our experience, the engine spends more than three hours to analyze paths under 13 instructions.) To find vulnerabilities more efficiently, Coppelia uses a hybrid search heuristic. Coppelia selects the next symbolic execution state to run by interleaving together breadth-first search and depth-first search to both explore as many processor instructions in as short a time as possible and explore each instruction in as much depth as possible. Each of them are run in a fixed number of times (chosen heuristically). We run depth-first more than breadth-first to allow enough time for the engine to explore the paths for each instruction.

3) Cone of Influence Analysis: In Coppelia, we apply a cone of influence (CoI) analysis to reduce the search space. The analysis is performed at the LLVM level during the static analysis phase, and removes signals from the design whose values do not affect, directly or indirectly, the value of signals in the security-critical assertions.

In developing the CoI analysis we found that performing the analysis at the function level was too conservative and led to little or no pruning. Almost every function was found to affect the function containing the assertion, but not all those functions affected the assertion itself. Therefore, we perform the dependency analysis at the instruction level. On the other hand, pruning at the instruction level was too costly. Program completeness could not be guaranteed and the symbolic execution engine had to check at each instruction whether to execute it or not. Therefore, we perform the pruning at the function level. Any function containing at least one instruction affecting a signal in the assertion is kept; all other functions are pruned. This hybrid approach allows us to prune aggressively while maintaining program completeness and keeping the run-time overhead low.

Algorithm 1: Cone of Influence Analysis

Input : A list of vars in the assertions varsInAssert
Output: A list of nodes in the graph nodeSet

```
1  trackedInstrs ← ∅;
2  nodeSet ← ∅;
3  dg ← BuildDependencyGraph();
4  for v ∈ varsInAssert do
5  |   vLocSet ← GetVarLocation(v);
6  |   for loc ∈ vLocSet do
7  |   |   nodes, instrs ← DependenceAnalysis(dg, loc,
   |   |     trackedInstrs);
8  |   |   nodeSet ← nodeSet ∪ nodes;
9  |   |   trackedInstrs ← trackedInstrs ∪ instrs;
10 |   end
11 end
```

The first step of our CoI analysis (Algorithm 1) is developing an interprocedural dependency graph. Each function forms a node and an edge from node a to node b is added if the inputs to b depend on the outputs of a. The second step is performing dependency analysis for the signals in the security-critical assertions. We extract the target signals in the assertions and get the location of these signals. Starting from these locations, we search backward through functions to track the LLVM instructions these signals depend on.

4) Compiler Optimizations: Verilator provides different levels of compiler optimizations for improving simulation performance [19]. We initially disabled optimizations and used $-O0$ flag because higher optimization levels adversely affect code readability and complicates the application of security-critical assertions because many of the signals and variables asserted over can be optimized out. Although using the $-O0$ flag confers significant readability benefits and eases assertion application, it slows the symbolic execution (Section IV-D). In Coppelia, we use the compiler optimizations to improve performance (Section IV-D) and modify the assertions for the optimized code.

F. Adding the Payload: Program Stubs

The sequence of instructions generated by the symbolic execution engine only triggers the bug. To better understand the security implications, we generate and append a payload to complete the exploit. This is based on our observation that although the triggers may differ, the same payload is often used across multiple exploits. Thus, we can use similar stubs for similar exploit situations.

Coppelia generates these program stubs according to the category of the security-critical properties being violated. We classified the security-critical properties into five classes as in the SCIFinder project [12]: `CF`: control flow related properties, `XR`: exception related properties, `MA`: memory access related properties, `IE`: properties to ensure execution of the correct and specified instructions, and `CR`: properties about correctly updating results.

III. Implementation

Coppelia is primarily implemented in C++ and Python. We build the state exploration part on top of KLEE, and the CoI analysis is written as LLVM passes. When we implement security assertions on the OR1200 processor in Cadence IFV as part of the evaluation, we use SystemVerilog.

A. Testbench Generation

Coppelia provides an automatic process to generate a testbench environment within which to verify the processor design. This environment provides stimulus to input ports, simulates the design, and checks for violations of security assertions.

We first make all inputs symbolic and then assign these symbolic values to input ports. The symbolic values are constrained by preconditions in order to generate legal instructions (Section II-E1). The whole processor design is simulated twice for each clock cycle (Section II-B). The simulation runs for as many clock cycles as there are pipeline stages to allow signals' values to be propagated through the entire pipeline. At the end of each clock cycle, we check whether security-critical assertions are violated.

B. Translating Security Assertions

The initial security assertions that we collected (Section IV-A) are developed specifically for the OR1200 processor, which is a 32-bit implementation of the OR1k architecture with Harvard microarchitecture, 5-stage integer pipeline, virtual memory support, and basic DSP capabilities. As part of our effort to find new bugs in different platforms and architectures, we also manually translate these assertions to the Mor1kx-Espresso processor (OR1k architecture) and the PULPino-RI5CY processor (RISC-V architecture).

The Mor1kx assertions correspond directly to OR1200 assertions because of their shared architecture so we need only adapt the assertions to different variables and pipeline stages. Assertions for the PULPino processor differ at a deeper level. We need to first verify that the examined security properties are still applicable to the new architecture. To do so we check both the RISC-V specification and the PULPino processor specification. We then adapt the assertions to appropriate variables and pipeline stages. The translation took us 1 day for the Mor1kx processor, and 2 days for the PULPino processor.

C. Program Stubs

For each category of the security-critical properties, we implement a few program stubs to complete the exploits. For some bugs, the instruction traces generated by symbolic execution cannot be directly connected to the program stubs. We manually implement the connecting code. Table I shows the number of stubs for each category and the average lines of code. As an example, the `R0` bug belongs to the memory access related category. The symbolic execution engine generates an instruction sequence that stores a non-zero value to `R0`. We then generate a program stub (in C) that exploits the bug by triggering a memory access instruction that expects `R0` to be zero. This demonstrates that an attacker using this bug can exploit it to write data to a memory locations as specified by the attacker.

IV. Evaluation

We evaluate Coppelia across multiple CPU designs to study its efficacy and its practicality. Our evaluation aims to answer the following research questions: 1) Can Coppelia effectively generate high-quality exploits for known CPU security bugs? 2) How does Coppelia perform compared to hardware model checking tools? 3) Is Coppelia practical for use on full-scale CPU designs, and what effect do our optimizations have on performance? 4) Can Coppelia be used to expose, and generate complete exploits for, new CPU security-critical bugs?

A. Dataset and Experiment Setup

For our evaluation, we collected 31 security-critical bugs (Table II) of the OR1200 processor from two prior papers, SPECS [5] and SCIFinder [12]. We collected 35 security-critical assertions from SPECS [5], Security Checkers [20], and SCIFinder [12]. We translated 30 assertions for the Mor1kx processor, and 26 assertions for the PULPino processor. The experiments are performed on a machine with Intel Xeon E5-2620 V3 12-core CPU (2.40GHz, a dual-socket server) and 62G of available RAM.

B. Generating Exploits for Known Bugs

To evaluate the efficacy of our tool against a ground truth, we test whether it can find and generate exploits for the known bugs we collected. These security-critical bugs are implemented in the OR1200 processor and we test Coppelia on the core of the processor. We run Coppelia by making both input signals and internal signals symbolic and executing backward toward the reset state.

Table II summarizes the results. For bug b16 we did not have an assertion. Bug b25 is a bug outside of the OR1200 core. Thus, we are not able to generate exploits for these two cases. In the remaining 29 cases, Coppelia is able to automatically generate exploits to expose the known bug for all of them. Overall, the generated exploits are concise, frequently only one or two instructions (excluding the size of the stubs). We can also see that for bugs that involve multiple cycles, Coppelia can indeed generate a series of instructions to exercise these deep error states.

For each generated exploit, we verify its ability to expose a vulnerability by running it on an FPGA board (DE0Nano). Each exploit contains a generated stub according to the type

Cat.	Description	Bug No.	No. of Stubs	Avg. LoC
CF	Control flow related	b20, b21, b27	2	15
XR	Exception related	b02, b03, b07, b08, b09, b10, b11, b14 , b15, b18, b19, b23, b29	3	29
MA	Memory access related	b17, b22, b24, b28, b30, b31	2	16
IE	Correct instructions	b06, b12	2	12
CR	Correctly updating results	b01, b04, b05, b13	2	13

Table I: Program stub categories for each bug and implementation details.

No.	Synopsis	Instructions Generated			Replayable		
		Coppelia	Cadence	EBMC	Coppelia	Cadence	EBMC
b01	Privilege escalation by direct access	2	1	1	✓	×	×
b02	Privilege escalation by exception	2	×	×	✓	-	-
b03	Privilege anti-de-escalation	1	1	1	✓	✓	✓
b04	Register target redirection	3	1	1	✓	×	×
b05	Register source redirection	1	1	1	✓	✓	✓
b06	ROP by early kernel exit	50	1	3	✓	×	×
b07	Disable interrupts by SR contamination	1	1	1	✓	✓	✓
b08	EEAR contamination	1	×	×	✓	-	-
b09	EPCR contamination on exception entry	2	×	×	✓	-	-
b10	EPCR contamination on exception exit	2	1	8	✓	✓	✓
b11	Code injection into kernel	2	1	1	✓	✓	✓
b12	Selective function skip	1	1	1	✓	×	×
b13	Register source redirection	1	1	1	✓	✓	✓
b14	Disable interrupts via micro arch	2	1	1	✓	✓	✓
b15	l.sys in delay slot will enter infinite loop	2	×	×	✓	-	-
b16	l.macrc immediately after l.mac stalls the pipeline	-	-	-	-	-	-
b17	l.extw instructions behave incorrectly	4	1	7	✓	×	×
b18	Delay Slot Exception bit is not implemented in SR	1	×	×	✓	-	-
b19	EPCR on range exception is incorrect	1	×	×	✓	-	-
b20	Comparison wrong for unsigned inequality with different MSB	3	1	1	✓	×	×
b21	Incorrect unsigned integer less-than compare	5	×	×	✓	-	-
b22	Logical error in l.rori instruction	5	×	×	✓	-	-
b23	EPCR on illegal instruction exception is incorrect	2	×	×	✓	-	-
b24	GPR0 can be assigned	2	1	6	✓	×	×
b25	Incorrect instruction fetched after an LSU stall	-	-	-	-	-	-
b26	l.mtspr instruction to some SPRs in supervisor mode treated as l.nop	3	×	×	✓	-	-
b27	Call return address failure with large displacement	2	1	1	✓	×	×
b28	Byte and half-word write to SRAM failure when executing from SDRAM	1	1	1	✓	✓	✓
b29	Wrong PC stored during FPU exception trap	2	×	×	✓	-	-
b30	Sign/unsign extend of data alignment in LSU	1	1	-	✓	✓	-
b31	Overwrite of ldxa-data with subsequent st-data	1	1	-	✓	✓	-

Table II: Generating exploits of collected bugs. The first 14 bugs are from SPECS [5] and the last 17 bugs are from SCIFinder [12]. The Instructions Generated column shows the number of instructions generated; the Replayable column shows whether the generated exploits can be replayable on an FPGA board. × means either the triggering information cannot be generated or the generated exploit is not replayable.

```
assign a_lt_b = comp_op[3] ? ((a[width-1] & !b[width-1]) |
  (!a[width-1] & !b[width-1] & result_sum[width-1]) |
  (a[width-1] & b[width-1] & result_sum[width-1])) :
  (a < b);               // Bug Free Version
  result_sum[width-1];  // Buggy Version
```

Listing 1: A security bug from OR1200 processor Bugzilla.

of the security assertion triggered by the bug (see Table I). As shown in Table II, all the exploits are successfully replayed on the FPGA board.

As an example, Listing 1 shows a security-critical bug (b20) from the OR1200 processor Bugzilla database (Bugzilla #51 [21]). The code snippet is from the ALU module in the OR1200 processor. It shows the logic to determine whether operand `a` is less than operand `b`. The buggy implementation works fine in most cases, but it fails for the `l.sfgtu` (set flag greater than equal) instruction. According to the OpenRISC specification [22], the instruction `l.sfgtu rA, rB` compares the contents of general-purpose registers `rA` and `rB` as unsigned integers. If the value of the first register is greater than the value of the second register, the compare flag is set; otherwise the compare flag is cleared. However, with this bug, if the highest-order bit in register `rA` is 1 the compare flag will not be set, even if `rA` is greater than `rB`. An attacker can exploit this bug to control which branch to execute. The security bug violates the security-critical assertion: the comparison flag should be set correctly. Listing 2 shows the generated exploit. (The full payload is abbreviated for space reasons.) The total CPU time required for generating this exploit is 9m40s. The exploit is replayable on an FPGA board.

C. Comparison with Model Checking

A current standard for hardware verification is model checking. In this section, we compare Coppelia against the commercial hardware model checking tool, Cadence's Incisive Formal Verifier (IFV), and against a research tool, EBMC [23]. We use each tool to look for the known bugs from Section IV-B

```
void foo() {
  printf("Attack success!\n"); // Payload
}
int main() {
  gotoUserMode();              // Payload
  asm volatile (               // Trigger
    l.movhi r16 0x8000;
    l.nop;
    l.sfgtu r16 r0;);
  jumpToFoo();                 // Payload
}
```

Listing 2: The exploit program generated by Coppelia.

and compare the results with Coppelia. We add the same constraints (Section II-E1) in both Cadence IFV and EBMC. The results are shown in Table II.

We make several observations:

(1) Cadence successfully finds and generates triggers for 18 bugs and EBMC for 16 bugs.

(2) Cadence fails to find or generate triggers for 11 bugs and EBMC fails for 13 bugs. All of them are found by Coppelia.

Among these bugs, 8 of them (b02, b08, b09, b15, b18, b19, b23, b29) are related to exception handling for managing privilege levels in the processor. Although we could not determine the exact reason why Cadence and EBMC fail to find these bugs, we note that the relevant properties for these bugs all include the condition (`wb_insn == syscall`). However, both Cadence and EBMC can find bug b14, which also relies on that same condition.

Bugs b21, b22, b26 are related to accessing register files. The OR1200 processor uses two dual-port RAMs for implementing register files. These two RAMs are written and read at the same time so that the processor can read two registers within a single clock cycle. However, we find that (`operand_b == 0`) is always true when running both model checking tools. This means data reading from `ram_b` is always 0, which is incorrect. We suspect that Cadence and EBMC build an incorrect model for the two RAMs.

EBMC fails to find and generate triggers for bugs b30 and b31 because it fails to parse assertions with deep hierarchies.

(3) As a tool designed for assertion verification rather than exploit generation, Cadence IFV only generates intermediate results when a property is invalidated. By contrast, the complete trigger is generated in Coppelia. For example, for bug b24 (the `R0` bug described in the introduction) Cadence generates the single instruction `l.addi r0, r1, 0`. This instruction will only trigger the bug if `r1` already holds a nonzero value, which is not the case for the reset state (`r1` is set to 0 at reset). In the traces Cadence generates, a number of signals are not in the reset state. It is nontrivial for designers to set the processor to a particular state in order to trigger the assertion. Table II shows that 12 exploits are not directly replayable from the reset state. For EBMC, we have similar results. Although EBMC returns multiple instructions, they are not always directly replayable from the reset state.

(4) We currently remove the memory from the processor and only run these tools on the processor core. When adding the memory back, it took Cadence several hours to build the model. It is necessary to rerun formal builds every time the verilog is changed so this would be a significant impediment to rapid development of bug fixes. Coppelia does not require long model building time but it fails to handle the memory because the queries to the solver are too long. We have not done optimizations for memory models but research on optimizing symbolic execution for arrays is ongoing [24] and could be incorporated into future versions of Coppelia.

D. *Effects of Optimizations*

To evaluate the effectiveness of our optimizations (Section II-E), we first randomly select six bugs which require only one instruction to trigger (examining longer bugs without optimizations took on the order of several days). For each bug, we make input signals symbolic and run Coppelia for one clock cycle, starting from the reset state. We show how each optimization influences the performance of symbolic execution. In the Original KLEE setup, we use KLEE's default settings, i.e., random search heuristic, 2000M maximum memory consumption, and counter example cache enabled. In the Hybrid Search setup, we enable the hybrid search heuristic. Specifically, we start with BFS and alternate the BFS and DFS. The BFS is set to run 10,000 times and the DFS is set to run 500,000 times. In the Compiler Optimizations configuration, we enable Verilator's optimizations when generating C++ code while keeping KLEE's settings the same as the previous column. In the CoI Analysis setup, we enable the CoI analysis in addition to all the settings in the previous column.

Table III summarizes the results, from which we make the following observations: (1) Adding all the optimizations yields an average overall speedup of two-to-three orders of magnitude compared to the original KLEE. On average, each optimization can enhance the performance by about one order of magnitude. (2) On average, the Hybrid Search heuristic improves the performance the most. (3) Applying all optimizations does not necessarily yield the best performance. For example, for bug b09 and b13, applying only the hybrid search heuristic can reduce the searching time to only 3 seconds, but adding other optimizations increases the search time.

Table IV shows the result of the Cone of Influence Analysis. Running the CoI Analysis can prune out a number of functions for symbolic execution. The effects of the CoI Analysis mainly depend on the security-critical assertions added. For the six bugs we picked, the first five have three variables in the assertions and the last one has four variables in the assertion. On average, the CoI Analysis prunes out 8% of the LLVM instructions and 30% of the functions. Table V shows that using `O3` level in the Compiler Optimizations can reduce 39% of the C++ code generated. Figure 4 compares the performance among different search heuristics. The upper figure shows the number of instructions covered in the generated test cases as time changes. The BFS covers the most instructions in a given amount of time. The lower figure shows the number of test cases per instruction generated as time changes. The DFS generates the most test cases per instruction in the given

No.	Original	Hybrid Search		Compiler Optimizations		CoI Analysis		Overall Speedup
	Time	Time	Speedup	Time	Speedup	Time	Speedup	
b05	3h50m5s	3m41s	62.47x	0m14s	15.54x	2m11s	0.11x	104.58x
b09	>24h	0m3s	>28800x	15m59s	0.004x	4m37s	3.46x	>311.91x
b10	19h30m49s	35m55s	32.60x	15m54s	1.16x	2m11s	7.32x	536.25x
b13	>24h	0m3s	>28800x	0m15s	0.22x	2m12s	0.11x	>654.55x
b24	19h31m33s	35m40s	32.85x	16m20s	2.18x	2m33s	6.42x	406.27x
b27	>24h	>6h	-	17m38s	>27.22x	11m29s	1.54x	>125.40
Avg.	>19h	>1.2h	>11545x	11m3s	>7.72x	4m12s	3.16x	>356.49x

Table III: Effects of optimizations. This table is aggregative, e.g. Compiler Optimizations means that Coppelia is running with both Hybrid Search and Compiler Optimizations on. Time columns show the CPU time. Speedup columns show the relative improvements in CPU time compared to previous columns.

No.	Func	Func Left	LLVM Instr	Instr Left
b05	47	34 (72.3%)	12501	11505 (92.0%)
b09	47	33 (70.2%)	12458	11427 (91.7%)
b10	47	33 (70.2%)	12475	11444 (91.7%)
b13	47	34 (72.3%)	12504	11508 (92.0%)
b24	47	34 (72.3%)	12474	11478 (92.0%)
b27	47	34 (72.3%)	12485	11489 (92.0%)

Table IV: Details of the Cone of Influence Pruning.

Optimization Level	Total LoC in C++
O0	14118
O3	8587 (61%)

Table V: Details of the Compiler Optimizations.

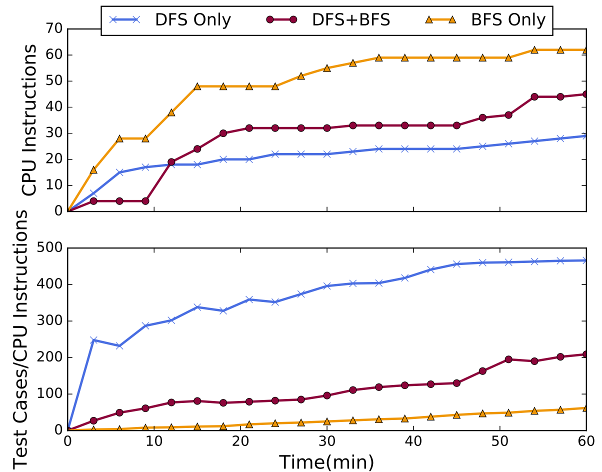

Fig. 4: Comparison of different search heuristics.

amount of time. Our hybrid search heuristic combines the advantages of both the BFS and the DFS heuristics.

E. Performance

For the 29 bugs Coppelia successfully generates exploits, 18 (62%) out of 29 of the exploits are generated within 15 minutes, demonstrating that Coppelia can be a practical quality control tool for hardware vendors. However, 2 (7%) out of 29 took a longer time (over 2 hours) to generate even for bugs involving only a single instruction. We find two reasons for the longer time: 1) Coppelia takes longer to reach the target instruction either because making internal signals symbolic increases the symbolic execution states to explore or because the instruction is near the end of the queue of all instructions to explore. 2) The bug is deep in the pipeline (in the 4th or 5th stage) and increasing the pipeline stages can dramatically increase the number of symbolic execution states. If we only run Coppelia for the target instruction (instead of all the instructions in the ISA), the time for generating the exploits can be reduced to only a few minutes.

F. Finding New Bugs

In this section, we examine Coppelia's efficacy in finding unknown bugs on new platforms and architectures. We run Coppelia on two new processors: Mor1kx-Espresso and PULPino-RI5CY. The Mor1kx is the most recent implementation of the OR1k architecture. We evaluate our tool on the Espresso core which is a 32-bit implementation with 2-stage integer pipeline and delay slot. The PULPino is an open-source single-core 32-bit low-power processor based on the RISC-V architecture. We evaluate our tool on the RI5CY core, which is an in-order, RV32-ICM implementation with 4-stage integer pipeline and DSP extensions. Table VI shows the new security bugs and their exploits we found in Mor1kx-Espresso processor and PULPino-RI5CY processor.

Bug b32 is the same as the motivating example `R0` bug. This bug was not fixed in the OR1200 processor and we still found it in the new generation of OpenRISC processor. This shows that security-critical bugs can persist to the next generation of processor designs.

Bug b33 allows incorrect escalation of privilege. According to the RISC-V specification, when the `EBREAK` instruction is executed, the privilege mode's epc register should be set to the address of the `EBREAK` instruction itself [25]. However, in RI5CY processor, we found that when the `EBREAK` instruction is executed, the `epc` register is not correctly updated. This is security critical because when the processor returns to user mode, it will jump to an incorrect address.

Bug b34 allows incorrect de-escalation of privilege. In the RISC-V specification, to return after handling a trap, the `SRET` instruction sets the `pc` to the value stored in the `epc` register [25]. However, `pc` is not set correctly when `SRET` is executed in the RI5CY implementation. This can be exploited by redirecting the program counter to an address of the attacker's choosing.

Bug b35 incorrectly updates the target `pc` of the jump instruction. The RISC-V specification states that the target

No.	Processor	Security Property	Instructions	Replayable(ZedBoard)
b32	Mor1kx-Espresso	Calculation of memory address / data is correct	2	✓
b33	PULPino-RI5CY	Privilege escalates correctly	1	✓
b34	PULPino-RI5CY	Privilege deescalates correctly	1	✓
b35	PULPino-RI5CY	Jumps update the target address correctly	1	✓

Table VI: New security-critical bugs and exploits found in Mor1kx-Espresso and PULPino-RI5CY Processor.

Items	No. of Assertions
Total Assertions	35
Pass Check	29
Fail Check (Bugs not fixed)	2
Fail Check (Wrong assertions)	4

Table VII: Security Patch Verification.

address of the indirect jump instruction is calculated by adding the 12-bit signed I-immediate to the register `rs1`, then setting the least-significant bit of the result to zero [25]. However, in the processor, the LSB is never set to 0; the implementation does not meet the specification. This may be leveraged by the attacker to silently redirect the `pc`.

G. Verify Patches and Refine Assertions

While running Coppelia on known bugs (Section IV-B), we also check the assertions by running Coppelia both on the buggy processor and on the patched processor expecting an exploit and no exploit respectively. While this is true for most cases, we sometimes observe that even after a bug is removed, Coppelia can still generate an exploit. This happens because either the processor is still buggy or because the assertions that we use based on prior work are not true assertions (the assertion does not consider some uncommon situation introduced by a correct patch because the assertions are collected from a dynamic simulation [12]). As shown in Table VII, for the 35 assertions we collected, 29 of them pass this check. For the 6 assertions that fail, 2 fail because the processor is still buggy (these 2 assertions pass the check after the bugs are fixed) and 4 are not true assertions.

This phenomenon implies that in addition to using Coppelia to generate exploits, we can also use Coppelia to verify whether a security patch indeed fixed a vulnerability, and to iteratively refine an initial set of assertions.

V. Discussion

We encountered an instance where Verilator failed to remove some dead code. The backward symbolic execution engine can, in this situation, fail to return a possible result. Because in the first iteration execution begins from a fully symbolic state, an assignment to variables that is only possible from within the unreachable dead code may be returned as a possible test case.

As with any symbolic execution engine operating over a system with unbounded execution paths, Coppelia is not complete and may fail to find vulnerabilities. There are two sources of incompleteness: (1) vulnerabilities that exist in the RTL may be optimized away during the translation to C++; (2) some paths may remain unexplored.

More complex processors contain performance-enhancing features such as speculative execution, caches, and out-of-order execution. We expect our backward symbolic execution strategy to be suited to such optimizations, and indeed may have some advantages over model checking or simulation based testing, but new heuristics and optimizations will be needed. For example, out-of-order processors use caches for register renaming. During the backward search, values in these caches must be chosen carefully in order to be meaningful and lead back to a reset state.

VI. Related Work

Automatic Exploit Generation. A closely related line of work is automatic exploit generation for software [26], [27], [28], [29], [8], [30]. Typically, vulnerabilities (e.g., buffer overflows) are first found through static or dynamic analysis, and then program input satisfying identified constraints are found. We tackle similar problems but differ in that we target the hardware domain, which requires a stateful analysis across multiple clock cycles to generate a series of input for hardware, instead of a single input as in software.

Information Flow Security for Hardware. Efficiently tracking information flow in hardware has been studied [31], [32], [33], [34], [35], but this approach often requires modifying or extending the hardware architecture. Cherupalli et al. proposed a gate-level symbolic simulation tool for information flow for particular IoT applications [36]. Some works develop or extend HDL for enforcing information flow security [37], [38], [39], [40]. Although these works can prove that a hardware design meets the security policies, they cannot verify those designs not already implemented in these languages.

Assertion Based Verification for Security. Assertion based verification uses simulation-based testing [41] or formal static analysis [42], [43], [44] to search for violations of assertions added to the design. Historically, functional properties were used [45], but recently security properties have been considered. These security properties may be manually [46], [20], [47], [5] or semi-automatically developed [12]. In this work we make use of these security properties from the literature.

Hardware Symbolic Simulation. Software symbolic execution [48], [49], [8], [13], [11] explores program paths with symbolic inputs [50]. Applying this technique to hardware designs for verification and testing has also been studied [51], [9]. STAR [51] is a functional input vector generation tool combining symbolic and concrete simulation for RTL designs over multiple time frames. It provides high range statements and branch coverage, but is limited by the sequential depth (around 6 cycles) [51]. PATH-SYMEX is a forward symbolic

execution engine that takes in ANSI-C interpretation of the RTL code [9]. Its application is limited to small RTL designs.

VII. Conclusion

We have presented Coppelia, an end-to-end tool for analyzing and contextualizing the security threat of hardware vulnerabilities. Given a processor design and a set of security properties, Coppelia generates C programs with inline assembly that exploit bugs within the design. Coppelia is able to generate exploits for 29 known bugs on the OR1200 processor, and discovered and generated exploit programs for 4 unknown bugs across two different processors and architectures.

Acknowledgments

We would like to thank the anonymous reviewers for their helpful feedback. This material is based upon work supported by the National Science Foundation under Grants No. CNS-1651276 and CNS-1816637, and by a Google Faculty Research Award. Any opinions, findings, conclusions, and recommendations expressed in this paper are solely those of the authors.

References

[1] "Intel Skylake/Kaby Lake processors: broken hyper-threading," https://lists.debian.org/debian-devel/2017/06/msg00308.html, June 2017.

[2] "Xen security advisory CVE-2015-5307,CVE-2015-8104 / XSA-156," http://xenbits.xen.org/xsa/advisory-156.html, Nov 2015.

[3] "Intel Core i7-600, i5-500, i5-400 and i3-300 Mobile Processor Series," *Specification Update*, 2014. [Online]. Available: http://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/core-mobile-spec-update.pdf

[4] "Revision Guide for AMD Family 16h Models 00h-0Fh Processors," *Product Revision*, 2013. [Online]. Available: http://support.amd.com/TechDocs/51810_16h_00h-0Fh_Rev_Guide.pdf

[5] 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: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2694344.2694366

[6] K. Karnane and C. Goss, "Automating root-cause analysis to reduce time to find bugs by up to 50%," Cadence Design Systems, Tech. Rep., 2015. [Online]. Available: www.cadence.com/rl/Resources/whitepapers/indago_debug_platform_wp.pdf

[7] D. Maksimovic, "Novel Directions in Debug Automation for Sequential Digital Designs in a Modern Verification Environment," Master's thesis, University of Toronto, Canada, 2015.

[8] 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. Washington, DC, USA: IEEE Computer Society, 2012. [Online]. Available: http://dx.doi.org/10.1109/SP.2012.31

[9] R. Mukherjee, D. Kroening, and T. Melham, "Hardware Verification using Software Analyzers," in *Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 2015.

[10] S. T. King, J. Tucek, A. Cozzie, C. Grier, W. Jiang, and Y. Zhou, "Designing and implementing malicious hardware," in *Proceedings of the First USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, April 2008.

[11] D. Davidson, B. Moench, T. Ristenpart, and S. Jha, "FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution," in *Proceedings of the 22nd USENIX Security Symposium*. Washington, D.C.: USENIX, 2013. [Online]. Available: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/davidson

[12] 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: ACM, 2017. [Online]. Available: http://doi.acm.org/10.1145/3037697.3037734

[13] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008. [Online]. Available: http://klee.github.io/

[14] "Verilator," https://www.veripool.org/wiki/verilator.

[15] "Clang: a C language family frontend for LLVM." [Online]. Available: https://clang.llvm.org/

[16] B. Alpern and F. B. Schneider, "Recognizing safety and liveness," *Distributed computing*, vol. 2, 1987.

[17] M. R. Clarkson and F. B. Schneider, "Hyperproperties," *J. Comput. Secur.*, vol. 18, Sep. 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1891823.1891830

[18] W. Snyder, "Verilator," https://www.veripool.org/papers/verilator_philips_internals.pdf, 2005.

[19] J. Bennett, "High Performance SoC Modeling with Verilator," http://www.embecosm.com/appnotes/ean6/embecosm-or1k-verilator-tutorial-ean6-issue-1.html, 2009.

[20] M. Bilzor, T. Huffmire, C. Irvine, and T. Levin, "Security Checkers: Detecting processor malicious inclusions at runtime," in *Hardware-Oriented Security and Trust (HOST), 2011 IEEE International Symposium on*, June 2011.

[21] "Comparison wrong for unsigned inequality with different MSB." [Online]. Available: http://bugzilla.opencores.org/show_bug.cgi?id=51

[22] D. Lampret, "OpenRISC 1000 Architecture Manual," https://github.com/openrisc/doc/blob/master/openrisc-arch-1.1-rev0.pdf?raw=true, 2014.

[23] D. Kroening and M. Purandare, "EBMC: The enhanced bounded model checker." [Online]. Available: http://www.cprover.org/ebmc/

[24] D. M. Perry, A. Mattavelli, X. Zhang, and C. Cadar., "Accelerating Array Constraints in Symbolic Execution," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2010.

[25] K. A. Andrew Waterman, "The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 1.10," https://riscv.org/specifications/privileged-isa/, 2017.

[26] D. Brumley, P. Poosankam, D. Song, and J. Zheng, "Automatic patch-based exploit generation is possible: Techniques and implications," in *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, ser. SP '08. Washington, DC, USA: IEEE Computer Society, 2008. [Online]. Available: https://doi.org/10.1109/SP.2008.17

[27] S. Heelan, "Automatic Generation of Control Flow Hijacking Exploits for Software Vulnerabilities," 2009. [Online]. Available: http://www.cprover.org/dissertations/thesis-Heelan.pdf

[28] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley, "AEG: Automatic Exploit Generation," in *Network and Distributed System Security Symposium*, Feb. 2011.

[29] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley, "Automatic Exploit Generation," *Commun. ACM*, vol. 57, Feb. 2014. [Online]. Available: http://doi.acm.org/10.1145/2560217.2560219

[30] T. Bao, R. Wang, Y. Shoshitaishvili, and D. Brumley, "Your Exploit is Mine: Automatic Shellcode Transplant for Remote Exploits," in *2017 IEEE Symposium on Security and Privacy (SP)*, May 2017.

[31] M. Dalton, H. Kannan, and C. Kozyrakis, "Raksha: A Flexible Information Flow Architecture for Software Security," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ser. ISCA '07. New York, NY, USA: ACM, 2007. [Online]. Available: http://doi.acm.org/10.1145/1250662.1250722

[32] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic, "Flex-iTaint: A programmable accelerator for dynamic taint propagation," in *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, Feb 2008.

[33] H. Chen, X. Wu, L. Yuan, B. Zang, P.-c. Yew, and F. T. Chong, "From Speculation to Security: Practical and Efficient Information Flow Tracking Using Speculative Hardware," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ser. ISCA '08. Washington, DC, USA: IEEE Computer Society, 2008. [Online]. Available: http://dx.doi.org/10.1109/ISCA.2008.18

[34] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood, "Complete Information Flow Tracking from the Gates Up," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIV. New York, NY, USA: ACM, 2009. [Online]. Available: http://doi.acm.org/10.1145/1508244.1508258

[35] A. Ardeshiricham, W. Hu, J. Marxen, and R. Kastner, "Register Transfer Level Information Flow Tracking for Provably Secure Hardware Design," in *Proceedings of the Conference on Design, Automation & Test in Europe*, ser. DATE '17. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2017. [Online]. Available: http://dl.acm.org/citation.cfm?id=3130379.3130775

[36] H. Cherupalli, H. Duwe, W. Ye, R. Kumar, and J. Sartori, "Software-based Gate-level Information Flow Security for IoT Systems," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: ACM, 2017. [Online]. Available: http://doi.acm.org/10.1145/3123939.3123955

[37] X. Li, M. Tiwari, J. K. Oberg, V. Kashyap, F. T. Chong, T. Sherwood, and B. Hardekopf, "Caisson: A Hardware Description Language for Secure Information Flow," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/1993498.1993512

[38] X. Li, V. Kashyap, J. K. Oberg, M. Tiwari, V. R. Rajarathinam, R. Kastner, T. Sherwood, B. Hardekopf, and F. T. Chong, "Sapper: A Language for Hardware-level Security Policy Enforcement," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. New York, NY, USA: ACM, 2014. [Online]. Available: http://doi.acm.org/10.1145/2541940.2541947

[39] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, "A Hardware Design Language for Timing-Sensitive Information-Flow Security," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2694344.2694372

[40] A. Ferraiuolo, R. Xu, D. Zhang, A. C. Myers, and G. E. Suh, "Verification of a Practical Hardware Security Architecture Through Static Information Flow Analysis," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. New York, NY, USA: ACM, 2017. [Online]. Available: http://doi.acm.org/10.1145/3037697.3037739

[41] L.-T. Wang, Y.-W. Chang, and K.-T. Cheng, *Electronic Design Automation: Synthesis, Verification, and Test*. Morgan Kaufmann, 2009.

[42] R. Brayton and A. Mishchenko, "ABC: An Academic Industrial-Strength Verfication Tool," in *Comuter Aided Verification (CAV)*. Lecture Notes in Computer Science, 2010.

[43] D. Brand, "Verification of Large Synthesized Designs," in *Proceedings of the IEEE/ACM International Conference on Computer Aided Design (ICCAD-93)*. IEEE, 1993.

[44] D.Lin, E.Singh, C.Barrett, and S.Mitra, "A structured approach to post-silicon validation and debug using symbolic dquick error detection," in *Proceedings of the IEEE International Test Conference*, 2015.

[45] H. Foster, *Applied Assertion-Based Verification: An Industry Perspective*, ser. Foundations and Trends(r) in Electronic Design Automation. Now Publishers, 2009. [Online]. Available: https://books.google.com/books?id=hL6d2t6Oh4EC

[46] M. Abramovici and P. Bradley, "Integrated Circuit Security: New Threats and Solutions," in *Proceedings of the 5th Annual Workshop on Cyber Security and Information Intelligence Research: Cyber Security and Information Intelligence Challenges and Strategies*, ser. CSIIRW '09. New York, NY, USA: ACM, 2009. [Online]. Available: http://doi.acm.org/10.1145/1558607.1558671

[47] M. Bilzor, T. Huffmire, C. Irvine, and T. Levin, "Evaluating security requirements in a general-purpose processor by combining assertion checkers with code coverage," in *Hardware-Oriented Security and Trust (HOST), 2012 IEEE International Symposium on*. IEEE, 2012.

[48] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: Automatically Generating Inputs of Death," in *Proceedings of the 13th ACM Conference on Computer and Communications Security*, ser. CCS, 2006. [Online]. Available: http://doi.acm.org/10.1145/1180405.1180445

[49] P. Godefroid, M. Y. Levin, and D. Molnar, "SAGE: Whitebox Fuzzing for Security Testing," *Queue*, vol. 10, Jan. 2012. [Online]. Available: http://doi.acm.org/10.1145/2090147.2094081

[50] J. C. King, "Symbolic Execution and Program Testing," *Communications of the ACM*, vol. 19, Jul. 1976. [Online]. Available: http://doi.acm.org/10.1145/360248.360252

[51] L. Liu and S. Vasudevan, "STAR: Generating input vectors for design validation by static analysis of RTL," in *IEEE International Workshop on High Level Design Validation and Test Workshop*. IEEE, 2009.