End-to-End Automated Exploit Generation for Processor Security Validation

Rui Zhang, Calvin Deutschbein, Peng Huang, and Cynthia Sturton, *Member, IEEE*,

Abstract—This paper presents Coppelia, an end-to-end tool that, given a processor design and a set of security-critical assertions, 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.

1 Introduction

In recent years we have seen reports of exploitable vulnerabilities in major, commercially available chips [1], [2]. A study of the AMD errata from 2007 to 2013 finds that 28 of the 301 processor errata are security-critical [3]. The recent Spectre and Meltdown attacks, and their variants [4], [5], further demonstrate the severe consequences of flaws in hardware designs. By using software-only attacks to exploit vulnerabilities in off-the-shelf hardware products, an attacker can gain control of the entire system, even if the system is running only secure software.

The current state of the art for finding errors in processor designs is to use formal static analysis or simulation-based testing. However, neither method is complete. We develop here a third option: software-style symbolic execution for hardware designs. It systematically explores paths in hardware designs to uncover errors.

Uncovering a potential bug is only the first step during a security validation process. Hardware designers must then assess the severity and security implication of each found bug. Our work takes an end-to-end approach by automatically generating software exploits to expose potential vulnerabilities. In particular, for each found bug, the tool generates a sequence of instructions that will trigger the bug plus a program stub that carries an exploit payload. The payload stub is generated based on the violated security properties. Together, the trigger and the payload stub form a complete exploit program to demonstrate a possible, concrete attack.

Generating the exploits not only allows hardware designers to uncover and reproduce vulnerabilities with concrete test cases, but also helps them contextualize, analyze and assess the security implications of a potential vulnerability. Furthermore, by using whether an exploit can be generated as a criterion, hardware designers can validate patches and refine assertions.

In this paper, we present Coppelia, an end-to-end exploit generation tool for use during the security validation of hardware designs. We evaluate Coppelia on three RISC processors of different architectures. Coppelia is able to find and generate exploits for 29 of 31 known vulnerabilities in these processors, and finds 4 new vulnerabilities along with exploits in these processors.

2 Hardware Security Validation

2.1 Validation Approaches

Current practice in hardware security validation at design-time often leverages assertion based verification (ABV) or information flow tracking (IFT).

ABV is the form of testing in which assertions added to the design encode security critical properties. Once assertions are added, simulation-based testing or formal static analysis may then be used to search for violations of the assertions. In simulation-based testing, test cases are used to find whether the assertions will be triggered; but assertion violations that exist along untested paths will not be discovered. In formal static analysis, the design is unrolled some number of cycles and the state space is methodically explored; but the state space grows exponentially which limits how far the design can be unrolled.

IFT uses dataflow tracking to track the flow of untrusted network, file and user inputs through memory. It requires tagging source variables with the appropriate level (e.g., "high" or "low") of information, asserting the correct level is maintained for sink variables, and deciding when to conditionally disable the assert or under what circumstances to allow declassification.

2.2 Security Properties

The security properties for hardware security validation are often written for use with a particular verification method, and each method has an associated specification language in which the properties can be expressed. The security properties and assertions used in hardware security validation may be manually or semi-automatically developed.

The security properties that have been developed to date for ABV make use of existing industry standard libraries for expressing assertions [11] and are written in a fragment of linear temporal logic that includes the globally (G) and next (X) operators. The properties expressible in the temporal logic are trace properties: individual traces of execution either satisfy or violate the given property. However, properties about how information flows through the processor are not immediately expressible as trace properties.

The information flow properties can be handled at the language level, using typed hardware description languages. An alternative approach is gate level information flow tracking in which shadow state added to the hardware design tracks how data flows. Standard trace properties expressed over the shadow state can then evaluate how information is allowed to flow through the original design. This approach has the advantage that existing designs, written in current industry standard hardware description languages, can be validated.

2.3 Symbolic Execution

Software symbolic execution explores a program using symbolic inputs that represent the set of possible values in the domain of the function. The program executes symbolically and when a branch statement is reached, execution forks and explores both subsequent paths. The symbolic exploration of a program can be represented by a tree (Figure 3). Each path through the tree represents a path of execution through the code; each node represents a line of code in the program. The symbolic execution engine will determine whether any assertions fire in the course of exploration. The engine also generates concrete test vectors that will drive execution down a particular path. The test vectors come from solving the accumulated path constraints on each path. Symbolic execution is often used for automatic exploit generation in software by searching the code base for buffer overflows, format string attacks, and memory corruption.

Applying symbolic execution to hardware designs for verification and testing has also been studied. STAR [6] 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. PATH-SYMEX is a forward symbolic execution engine that takes in ANSI-C interpretation of the RTL code [7].

3 Coppelia Overview

We design a tool, Coppelia, to provide an end-to-end solution for validating the security of processor designs. 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.

Coppelia takes an Hardware Description Language (HDL) implementation of a hardware design and a set of security-critical assertions as inputs, searches for violations of the assertions and, if it finds any, generates complete exploit programs as output. These security-critical assertions are developed either by manually studying the specification, or by assertion mining techniques; and if violated, there are vulnerabilities in the design. Figure 1 shows the workflow of Coppelia. There are three main steps in Coppelia: preprocessing, finding violations and building the triggers, and adding the payloads. We describe the overview of each step in the following sections.

With the purpose of reproducing bug exploits, Coppelia focuses more on the sequential depth of the exploration with backward symbolic execution scheme and Coppelia can be easily integrated to the current industrial verification flow at the design time by leveraging software verification methods.

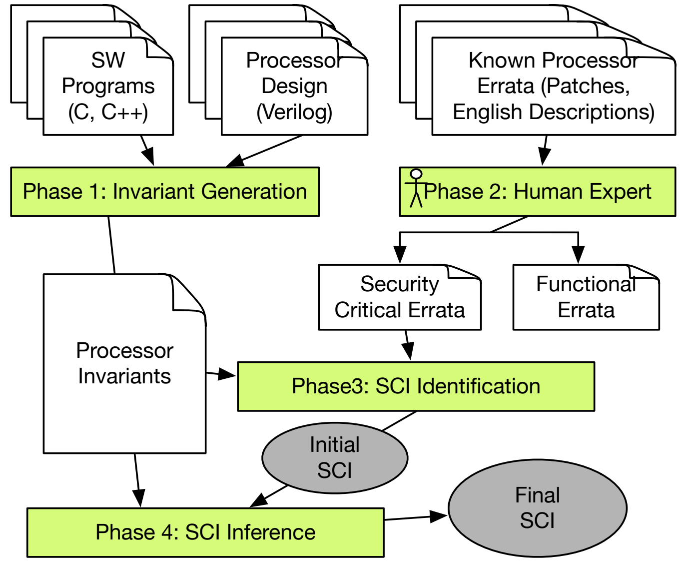

Fig. 1: Workflow of Coppelia. The process labeled BSEE is the backward symbolic execution engine.

```
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.

3.1 A Vulnerability Example

Before we describe an overview of each step of Coppelia, we first give an example of a vulnerability in the OR1200 processor. Listing 1 shows a security-critical bug (b20) from the OR1200 processor Bugzilla database (Bugzilla #51). 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, 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.

An exploit program to such vulnerabilities typically include two parts: a trigger and a payload (Figure 2). We use the security-critical assertions to build the triggering part, i.e. make the compare flag unset in the supervisor register; and we append a program stub as the payload part, i.e. use a branch instruction to redirect the program control flow.

3.2 Preprocessing

To begin, Coppelia translates the RTL hardware design from an HDL implementation to C++. We use the Verilator tool [8] for this step and can translate designs written in Verilog or

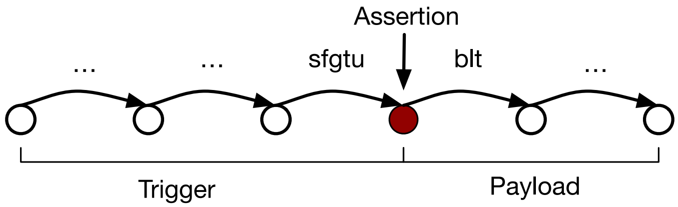

Fig. 2: Illustration of components of an exploit program.

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 [9], a mature symbolic execution engine, and use it as the foundation of Coppelia. 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. We assume that the input signals remain stable during a single execution of one clock cycle and will only change at clock cycle boundaries. Although this assumption can potentially lead to missing corner cases which include input signals change between cycle boundaries, it ensures the circuit model converges and improves the efficiency for the code analysis.

3.3 Building a trigger

The goal of this step is to find a *vulnerability*, a processor state in which a security-critical assertion is violated, and generate a sequence of inputs that take the system from the initial state to the vulnerable state.

Coppelia achieves these two goals uses symbolic execution. There are two challenges with applying symbolic execution to hardware designs. First, the symbolic execution of a hardware design represents an exploration of the design for a single clock cycle. However, hardware executes continously, and security vulnerabilities may only become apparent many clock cycles after the initial state. Second, security properties developed for hardware designs capture the semantics of particular signals and their connecting logic. Finding violations of these properties is akin to finding a needle in a haystack.

We propose in Coppelia a strategy of backward symbolic execution. We start from a random state and symbolically execute the design searching for a path from the random state to the point of an assert statement. We then symbolically execute the design multiple times, cycle-by-cycle, backwardly, searching for a path from an assertion-violating state back to the initial state. Within each iteration, we run the symbolic execution forwardly. Figure 3 shows our backward symbolic execution strategy. 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.

3.4 Adding the payload

To better analyze and assess the security consequences of a found bug, Coppelia move beyond the mere triggering of the bug to the generation of complete exploit programs that demonstrate a possible concrete attack.

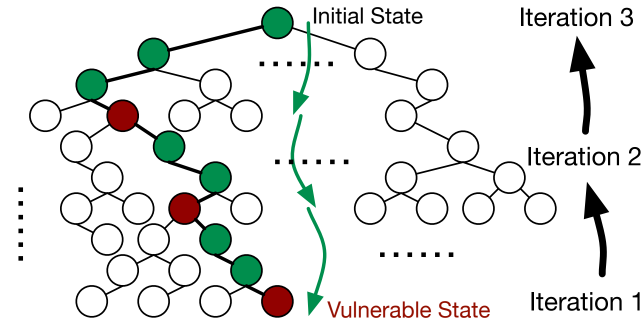

Fig. 3: Backward symbolic execution strategy: We search for a path from the last cycle to the first cycle (black arrows). Within each cycle, we symbolically execute the hardware design forwardly (green arrows).

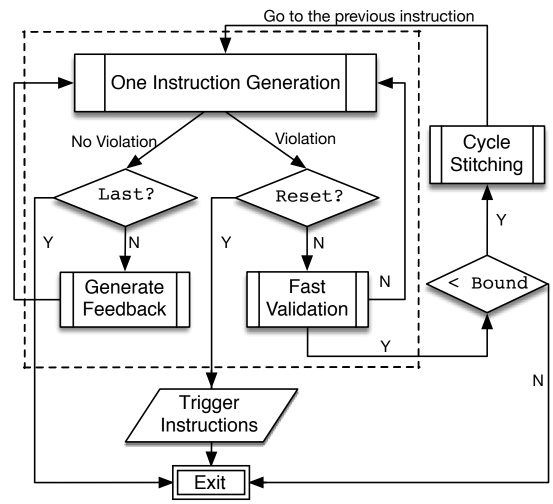

Fig. 4: Workflow of Backward Symbolic Execution

We observe that although the triggers may differ, the same payload is often used across multiple exploits. Thus, we 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 properties into five classes as in SCIFinder [10]: control flow related properties; exception related properties; memory access related properties; properties to ensure execution of the correct and specified instructions; properties about correctly updating results.

4 Backward Symbolic Execution

We describe the workflow of our hardware-oriented backward symbolic execution engine (see Figure 4). In the following sections, we describe each step in detail.

4.1 One Instruction Generation

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 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 has a leaf node that represents the vulnerable state of the processor. Associated with that leaf node is the path condition that describes the sufficient constraints on processor state and input signals such that the processor will move from the constrained state 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 a state that satisfies the constraints returned in the prior iteration. If such a state is found, the engine returns a path condition 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 (searching forward within trees) 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.

4.2 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.

4.3 Fast Validation

At the end of each successful iteration, the backward symbolic execution engine checks the following: are the constraints given in path condition 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.

The first rule is to steer the search toward the reset state. 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 define the empirical distance between two states as a count of stateful registers whose valuations differ in the two states. At each interation, we set a threshold. If the empirical distance is above the threshold, we abort the current iteration. Otherwise, we continue with our backward search.

The second rule targets loops that are preventing backward progress toward the initial state. At each new iteration, the set of processor states may include states found in previous iterations, in which case the search may have entered a loop. Thus, we define a set to keep track of the states found in previous iterations. In subsequent iterations, if the state is in this set, we continue the symbolic execution until we find a new state. Otherwise, we update the set with the current state.

4.4 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.

4.5 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 the current cycle and the previous cycle to align, we need to replace the values of internal signals in node in the previous cycle with the path constraint obtained in node in the current cycle. This ensures completeness – we will not miss a possible test case. However, the complexity of this method is similar to forward symbolic execution. 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.

4.6 Feedback Generation

If the engine finishes exploring all paths and no violations are found and this is not the first iteration (Figure 4), it means a violation was found in previous runs but the engine

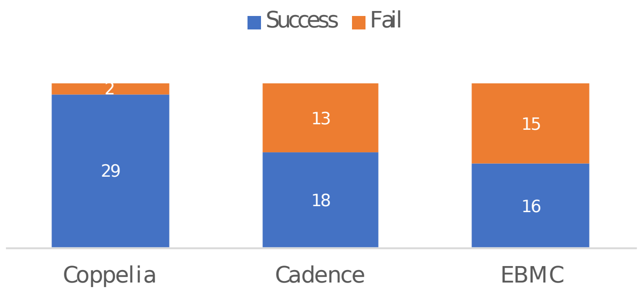

Fig. 5: Generating exploits of collected bugs.

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.

5 Evaluation

We evaluate Coppelia across multiple CPU designs to study its efficacy and its practicality. We collected 31 security-critical bugs of the OR1200 processor from two prior papers, SPECS [3] and SCIFinder [10]. We collected 35 security-critical assertions from SPECS [3], Security Checkers [11], and SCIFinder [10]. 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 62GB of available RAM.

5.1 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.

Figure 5 summarizes the results. Coppelia fails to generate exploits for two cases. For one of them, we did not have an assertion; for the other one, it is a bug outside of the OR1200 core. 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 of the security assertion triggered by the bug. Listing 2 shows the generated exploit for the vulnerability described in Session 3.1. The total CPU time required for generating this exploit is 9m40s.

```
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.

5.2 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 [12]. We use each tool to look for the known bugs from Section 5.1 and compare the results with Coppelia. We add the same constraints in both Cadence IFV and EBMC. The results are shown in Figure 1.

We found that 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 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.

The remaining 3 bugs 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 2 additional bugs because it fails to parse assertions with deep hierarchies.

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, there is one bug that allows users to assign non-zero values to the general purpose register R0. Cadence generates the single instruction `l.addi r0, r1, 0`. This instruction will only trigger the bug if `r1` already holds a non-zero 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. We found 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.

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

No.	Processor	Security Property
b1	Espresso	Calculation of memory address is incorrect
b2	RI5CY	Privilege escalates incorrectly
b3	RI5CY	Privilege deescalates incorrectly
b4	RI5CY	Jumps update the target address incorrectly

Table 1: New security-critical bugs and exploits found in Mor1kx-Espresso and PULPino-RI5CY Processor.

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 and could be incorporated in the future.

5.3 Performance

For the 29 bugs for which 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%) of the exploits took over 2 hours to be generated. 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 run Coppelia for the target instruction (instead of all the instructions in the ISA), the time for generating the exploits can be reduced to a few minutes.

5.4 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 1 shows the new security bugs and their exploits we found in Mor1kx-Espresso processor and PULPino-RI5CY processor.

6 Future Work

Future work will target scalability and expressiveness. Scaling to larger and more complex processor designs will require new optimization approaches. Moving beyond assertions to hyperproperties, for example, would allow Coppelia to find property violations related to information flow.

7 Conclusion

We have presented Coppelia, an end-to-end tool for analyzing and contextualizing the security threats of hardware. 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 thank the reviewers for their insightful comments and suggestions. 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.

References

[1] "Amd processor microcode security update," https://lists.debian.org/debian-security/2016/03/msg00084.html, Mar 2016.
[2] "Intel Skylake/Kaby Lake processors: broken hyper-threading," https://lists.debian.org/debian-devel/2017/06/msg00308.html, June 2017.
[3] 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*, 2015.
[4] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.
[5] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.
[6] 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, pp. 32–37.
[7] 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.
[8] "Verilator," https://www.veripool.org/wiki/verilator.
[9] 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/
[10] 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*, 2017.
[11] 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, pp. 34–39.
[12] D. Kroening and M. Purandare, "EBMC: The enhanced bounded model checker." [Online]. Available: http://www.cprover.org/ebmc/

Rui Zhang earned her Ph.D. from UNC Chapel Hill. Her research focuses on developing tools and systems for security validation of hardware designs.

Calvin Deutschbein is a doctoral student at UNC Chapel Hill (M.S. Computer Science 2017) studying specification mining, hardware security, and assertion based verification.

Peng Huang is an assistant professor at Johns Hopkins University. His research focuses on the reliability and fault tolerance of computer systems. He earned his PhD from UC San Diego.

Cynthia Sturton is an associate professor at UNC Chapel Hill. Her research interests lie at the intersection of hardware design, security, and formal methods. She earned her M.S. and Ph.D. in computer science at UC Berkeley. She is a member of IEEE and the ACM.