A Recursive Strategy for Symbolic Execution to Find Exploits in Hardware Designs

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

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

Abstract

This paper presents hardware-oriented symbolic execution that uses a recursive algorithm to find, and generate exploits for, vulnerabilities in hardware designs. We first define the problem and then develop and formalize our strategy. Our approach allows for a targeted search through a possibly infinite set of execution traces to find needle-in-a-haystack error states. We demonstrate the approach on the open-source OR1200 RISC processor. Using the presented method, we find, and generate exploits for, a control-flow bug, an instruction integrity bug and an exception related bug.

CCS Concepts • **Security and privacy → Logic and verification**;

Keywords Recursive Strategy, Symbolic Execution, Hardware Security, Exploit Generation

ACM Reference Format:
Rui Zhang and Cynthia Sturton. 2018. A Recursive Strategy for Symbolic Execution to Find Exploits in Hardware Designs. In *Proceedings of 2018 ACM SIGPLAN International Workshop on Formal Methods and Security (FMS'18).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3219763.3219764

1 Introduction

This paper presents the use of a recursive strategy for symbolic execution to find exploitable vulnerabilities in hardware designs. Hardware vulnerabilities present an enticing target to attackers [5–8, 21]. As demonstrated by the Spectre and Meltdown attacks, bugs in hardware designs can have wide-reaching consequences and can be difficult to patch post-deployment [3, 25, 29]. By making symbolic execution feasible for large scale hardware designs, we provide a new, effective, and practical method for the hardware designer to use in the ever-present race to detect vulnerabilities before tape-out.

Current practice in hardware design verification combines formal static analysis techniques with simulation-based testing. While these methods can successfully find functional assertion violations, neither are well-suited to the identification and analysis of security-critical vulnerabilities.

The efficacy of simulation-based testing depends on the coverage of the testbenches used, and is unlikely to uncover a vulnerability that is exploitable by only a handful of possible input sequences. On the other hand, formal static analysis, typically in the form of model checking, requires the user to determine the root cause for any found violations. This analysis is difficult – involving consideration of environment constraints, invalid input sequences, and specification errors – and often requires cooperation between security experts, hardware designers, and formal methods experts [22, 32].

Symbolic execution, widely used in the software security community, is a powerful technique for automatically generating test cases to trigger security vulnerabilities [16]. It has a reputation for relative ease of use and high code coverage. Yet, standard symbolic execution as used in the software domain is not directly applicable to full-scale hardware designs; its use has been limited to individual modules [33]. The large state space of a modern processor combined with the continuous execution of hardware makes for a search space that is infeasible to tackle. In this paper we develop the use of symbolic execution for full-scale hardware designs.

We focus on processor designs. Our goal is to use symbolic execution to generate a sequence of input values – instructions and control signals – that will take the processor from the reset state to a vulnerable state. We tackle the problem recursively, generating that last input values in the sequence first, thereby (ideally) reducing the problem from finding a sequence of length m to finding a sequence of length $m-1$. In each iteration we use forward symbolic execution to find the set of input values that will move the processor one clock-cycle forward along the desired path. Because there is no guarantee the recursive search strategy will converge to the reset state, we develop heuristics that steer the search toward the reset state and break potential loops.

FMS'18, June 18, 2018, Philadelphia, PA, USA
© 2018 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 978-1-4503-5833-0/18/06…$15.00
https://doi.org/10.1145/3219763.3219764

Using standard symbolic exploration tree definitions, we formalize the strategy of hardware-oriented symbolic execution with recursive reasoning. We show that this strategy is more efficient than standard forward symbolic execution, and produces concrete, replayable instruction sequences that exploit hardware vulnerabilities.

We have built the recursive reasoning method based on KLEE [14], a popular symbolic execution engine, and demonstrate its effectiveness on the OR1200 RISC processor. Our use of recursive reasoning makes symbolic execution feasible for full-scale hardware designs, bringing the benefits of fast, easy-to-use symbolic execution to the hardware design space.

In summary, this paper presents the following contributions:

1. We formally define our strategy of recursive reasoning with hardware-oriented symbolic execution to discover vulnerabilities and generate exploits in full-scale hardware designs.
2. We present the heuristics that make the recursive search strategy practical and effective.
3. We demonstrate the method on three security-critical bugs in the OR1200 RISC processor, successfully finding and generating exploits for all three bugs, including one that the commercial model checking tool, Cadence, could not find.

2 Problem Formulation

We define our threat model, introduce notation, and define the problem we tackle.

2.1 Threat Model

We consider a processor design with vulnerable flaws that are exploitable, post-deployment, by software. By exploiting the vulnerabilities, attackers may be able to escalate their privilege level, leak or modify confidential data in memory, or redirect the processor to execute code of their choosing.

We assume the attacker acts post-deployment, and therefore does not modify the processor design. The attacker is, however, capable of finding vulnerabilities that exist within the design. We assume the attacker is able to send network packets, execute a particular sequence of instructions, or both on the target machine.

Our objective is to generate a trace of instructions that can exploit a vulnerability in the processor design so that designers can be alerted to the vulnerability, assess the threat posed by the vulnerability, and patch the vulnerability before tape-out. We limit our scope to the register transfer level processor designs; we do not analyze designs at the gate- or transistor-level.

2.2 Definitions and Notation

We first introduce the notation needed to define a processor and exploit program, and then formally state the problem we tackle.

Definition 1. We model the processor as a tuple $\mathcal{M} = (S, s_0, I, \delta, O, \omega)$, where

- S is the finite set of processor states,
- $s_0 \in S$ is the initial state of the processor,
- $I = \{0,1\}^n$ is the finite set of input strings to the processor,
- $\delta : S \times I \rightarrow S$ is the transition function of the processor,
- $O = \{0,1\}^m$ is the finite set of output strings of the processor, and
- $\omega : S \rightarrow O$ is the output function that maps processor states to output strings.

A *processor state* $s \in S$ is a concrete valuation of all state-holding, internal elements of the processor. These include micro-architectural registers, control signals, and memory.

The *initial state* s_0 of the processor is the reset state. Many registers have a valuation of 0 in the reset state.

All *input strings* $i \in I$ are of fixed-length n over the binary alphabet $\{0,1\}$. The string i is a concatenation of all input values to the processor: the next processor instruction, any data fetched from memory, and acknowledge, error, debug, interrupt, and some control signals.

The *transition function* δ describes the evolution of the processor in a single clock cycle. It is a left-total function: for every $s \in S$ and every $i \in I$, $\delta(s,i)$ is defined. The function is determined by the combinational logic of the processor design.

All *output strings* $o \in O$ are of fixed-length m over the binary alphabet $\{0,1\}$.

The *output function* ω is the identity function over a subset of processor state elements.

We will define an *exploit program* as a sequence of inputs that take the processor from the initial state to an error state. We must first introduce additional notation and define assertions and error states.

The processor state can be viewed as a vector of the state-holding, internal elements of the processor: $s = \langle r_0, r_1, \ldots, r_{|s|} \rangle$. (We denote the number of such elements as $|s|$, although it is invariable and does not change with each state.)

An *assertion* encodes a desired property of the processor. Let $A(s)$ be an assertion over a processor state $s \in S$:

$$A(s) \doteq \phi(\langle r_0, r_1, \ldots, r_{|s|} \rangle), \qquad (1)$$

where ϕ is a boolean-valued function over (a subset of) the state-holding elements of the processor. The property ϕ is a function in a quantifier-free fragment of first order logic [10]. An example ϕ is $\phi(\texttt{GPR0}) \doteq \texttt{ITE(GPR0 == 0,T,F)}$; asserting this function encodes the property that the general purpose register GPR0 should always have value 0.

For any assertion A, there may be a set of error states $E_A \subseteq S$, which violate the assertion:

$$E_A = \{s \in S | \neg A(s)\} \tag{2}$$

We can now define an exploit program $P_{A,\mathcal{M}}$ for a processor $\mathcal{M}$ and assertion A.

Definition 2. The exploit program $P_{A,\mathcal{M}} = (i_0, i_1, \ldots, i_m)$ is a sequence of input strings that, starting from the initial state, invokes a sequence of states, $s_0, s_1, \ldots, s_{m-1}, s_m, s_e$, where

- s_0 is the initial state,
- $\forall j\ \ 0 \leq j \leq m. \quad i_j \in I$,
- $\forall j\ \ 0 \leq j \leq m. \quad s_j \in S$,
- $s_e \in E_A$ is an error state that violates the assertion, and
- $s_1 = \delta(s_0, i_0), \quad s_2 = \delta(s_1, i_1), \ldots, \quad s_e = \delta(s_m, i_m)$.

We now formally state the problem we tackle in this paper, which is two-fold.

Problem Statement Given a processor $\mathcal{M}$ and assertion A,

1. Determine whether there exist any error states that violate the assertion ($E_A \neq \{\}$), and
2. If any such states exist, generate an exploit program $P_{A,\mathcal{M}}$ for each violating state.

3 Symbolic Execution of Hardware Designs

In this section we define symbolic execution for the domain of hardware designs. We start with a brief review of standard symbolic execution as applied to software [23] and then provide an abbreviated primer on hardware designs at the register transfer level [37]. We end with a description of the symbolic execution of hardware designs.

3.1 Symbolic Execution

In software symbolic execution, the input values are replaced with symbolic representations of the set of possible values in the domain of the function. As execution continues the symbols are used in place of concrete values. In addition to the symbolic values, a path condition π is associated with the current path of execution. When execution begins $\pi :=$ `True`. When a conditional branch is reached the condition p is evaluated. If $\pi \longrightarrow p$, the `then` branch is taken. If $\pi \longrightarrow \neg p$, the `else` branch is taken. If neither implication holds then both branches are possible. Execution forks and each path is explored in turn. The path condition is updated: $\pi := \pi \wedge p$ for the `then` branch and $\pi := \pi \wedge \neg p$ for the `else` branch of execution.

The symbolic exploration of a program can be represented as a tree. Each node represents a line of code in the program and has a path condition associated with it. Each path through the tree represents a path of execution taken during the symbolic exploration.

```
reg [1:0] count;
always @(posedge clk)
  if (reset)
    count <= 0;
  else if (count == 0)
    count <= {count[0], 1'b1};
  else
    count <= {count[0], 1'b0};
end
```

Listing 1. An example of sequential block in Verilog.

3.2 Register Transfer Level Hardware Designs

The hardware designs we are targeting are specified at the register transfer level. A register transfer level design specifies the flow of data and connecting logic between state-holding elements of the design. The designs we are interested in are implemented in a hardware description language. To make our conversation concrete, we use one such hardware description language, Verilog, in our examples [34]. The ideas generalize for other hardware description languages.

A basic unit of design in Verilog is a module. One module may implement, for example, an arbiter for bus communication or the pipeline logic of a CPU. Modules can contain other modules, making the design hierarchical. A module combines multiple sub-modules by making the output signals of one module connect to the input signals of a second module, with possibly some connecting logic in between.

The variables of a module are input `wire`s, intermediate `reg`s and `wire`s, and output `reg`s and `wire`s. The body of the module consists of assignments to the intermediate and output `reg`s and `wire`s. A variable of type `wire` can not hold state and must be continually driven. Variables of type `reg`, on the other hand, can hold state. Assignments to `reg`s are grouped in `always` blocks, and these may be sequential (stateful) or combinational (stateless) assignments. The value of a `reg` assigned in a combinational block changes as soon as any of the inputs to the block change value. The value of a `reg` assigned in a sequential block changes only at time-delta boundaries (e.g., only on the rising edge of a clock signal) and otherwise maintains its current value (its state) between deltas (see Listing 1). [1]

3.3 Hardware-Oriented Symbolic Execution

We can now describe the symbolic execution of hardware designs.

Definition 3. Let $\mathcal{E}$ be a directed rooted tree representing the symbolic exploration of the processor design. Each node of the tree $n = (\bar{s}, \pi)$ is a tuple representing a point in the symbolic exploration of the design, where

[1]We can now be more specific about what we mean by a processor state s in our model of a processor. A state $s \in S$ represents a concrete valuation of all `reg` signals that are assigned in sequential `always` blocks.

- $\bar{s}$ is the current symbolic state of the processor, and
- π is the path constraint associated with the current point of execution.

The *symbolic state* $\bar{s}$ is a (partially) symbolic valuation of the variables of the processor design. At any point of execution each variable may have a symbolic or concrete value.

The *path constraint* π is a boolean formula over the internal and input variables of the design.

The *root node* $n_r = (\bar{s_r}, \pi_r)$ represents a (partially) symbolic state of the processor at a clock cycle boundary and always has path constraint $\pi_r := \texttt{True}$.

A *leaf node* $n_l = (\bar{s_l}, \pi_l)$ represents a symbolic next-state—the state at the next clock cycle boundary—of the processor.

A particular path in $\mathcal{E}$ from n_r to n_l corresponds to a particular evolution of the processor in one clock cycle and is prescribed by the constraints π_l: if these constraints over the internal and input variables are satisfied, the processor will always follow the same path from n_r to n_l. In our analysis we are concerned with the state of the processor at clock-cycle boundaries; these are the states represented by the root or leaf nodes of a symbolic exploration tree.

Given a tree $\mathcal{E}$ we can reason about the set of processor states and next-states represented by the root and leaf nodes of the tree. We do this by applying the constraints in π to the symbolically defined values in $\bar{s}$ to produce a set of concrete states. We use the notation $\bar{s} \circ \pi$ to represent such a set.

Definition 4. Let $\mathcal{E}$ represent the symbolic exploration of one clock cycle of a processor modeled by $\mathcal{M}$. Let $n_r = (\bar{s}_r, \pi_r)$ be the root node of tree $\mathcal{E}$ and let $n_l = (\bar{s}_l, \pi_l)$ be a leaf node of the same tree. Then $\bar{s}_r \circ \pi_l$ represents the set of concrete states, and $\bar{s}_l \circ \pi_l$ represents the set of concrete next-states, that are at the end-points of the path from n_r to n_l.

For example, if count is made symbolic ($\texttt{count} := \alpha$), as shown in Listing 1 and Figure 1, then by the time execution reaches the leaf node, it might have value $\texttt{count} = ((\alpha\&1) \ll 1)\&3$, as shown in the right leaf node of Figure 1. If the constraint given by π_l is $\alpha \neq 0$, then $\bar{s_r} \circ \pi_l$ produces the set of concrete states $\{\texttt{count} := 1, \texttt{count} := 2, \texttt{count} := 3\}$, and $\bar{s_l} \circ \pi_l$ produces the set of concrete next-states $\{\texttt{count} := 0, \texttt{count} := 2\}$.

4 Generating Program Exploits

We describe our strategy for generating exploit programs using symbolic execution. We first reframe the problem statement in terms of symbolic execution trees and then describe our algorithm for generating the necessary sequence of trees recursively. We then provide details and introduce heuristics for making the search feasible.

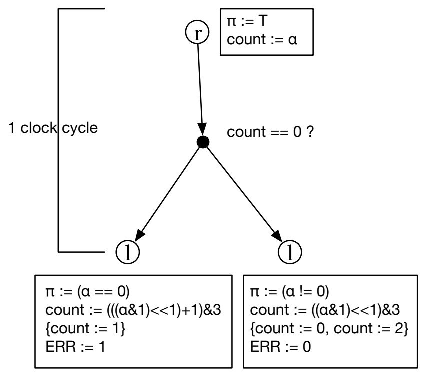

Figure 1. Symbolic exploration tree of Listing 1.

4.1 Using Symbolic Execution

Recall that we wish to generate a sequence of inputs that will take the processor from the reset state to an error state.

Strategy Given E_A, a set of error states of the processor modeled by $\mathcal{M}$, find a sequence of symbolic exploration trees $\mathcal{E}_0, \mathcal{E}_1, \ldots, \mathcal{E}_m$, and for each tree an *identified leaf node* $n_{l,0}, n_{l,1}, \ldots, n_{l,m}$, such that

1. $s_0 \in \bar{s}_{r,0} \circ \pi_{l,0}$,
2. $E_A \cap \bar{s}_{l,m} \circ \pi_{l,m} \neq \{\}$,
3. $\bar{s}_{l,j} \circ \pi_{l,j} \subseteq \bar{s}_{r,j+1} \circ \pi_{l,j+1}$.

The first requirement states that the concrete initial state s_0 of the processor (the reset state) is in the set of concrete states defined by applying $\pi_{l,0}$ to $\bar{s}_{r,0}$ of the first tree in the sequence, $\mathcal{E}_0$.

The second requirement states that the set of concrete states $\bar{s}_{l,m} \circ \pi_{l,m}$ of the identified leaf node of the last tree in the sequence ($n_{l,m}$) includes desired error states.

The third requirement states that for $0 \leq j \leq m-1$, the identified leaf node of tree $\mathcal{E}_j$ can be a starting point (root node) for the path identified in tree $\mathcal{E}_{j+1}$.

If the above requirements are met then the sequence of path constraints $\pi_{l,0}, \pi_{l,1}, \ldots, \pi_{l,m}$ provided by the sequence of identified leaf nodes $n_{l,0}, n_{l,1}, \ldots, n_{l,m}$ define the sequence of inputs to the processor that will take the processor from the initial state to an error state.

4.2 Recursive Algorithm

We recursively generate the desired sequence of trees and identified leaf nodes. Using this strategy, the problem of producing a particular sequence of trees and identified leaf nodes is reduced to producing, in each iteration, a single tree and identified leaf node.

In the base case the algorithm looks for a symbolic execution tree $\mathcal{E}_m$ with identified leaf node $n_{l,m}$ in which the symbolic state $\bar{s}_{l,m}$, when constrained by $\pi_{l,m}$, represents a set of states that includes an error state $s_e \in E_A$. This ensures by construction that the second requirement of our strategy is met.

Algorithm 1: Recursively Generate the Input Sequence

```
Input  : Initial States $S_I$, Error States $S_E$
Output : Input Vector $\vec{i}$
1  Function FindInputs(s, $\vec{i}$)
2      if $\exists i \in I, s_0 \in S_I$ s.t. $s = \delta(s_0, i)$ then
3          return $i : \vec{i}$ ;
4      else
5          if $\exists i \in I, s' \in S$ s.t. $s = \delta(s', i)$ then
6              FindInputs($s', i : \vec{i}$) ;
7          else
8              return ERROR ;
9          end
10     end
11 Function main($S_I, S_E$)
12     if $\exists s_e \in S_E$ s.t. $s_e \in S_I$ then
13         return $(s_e, \epsilon)$ ;
14     else
15         FindInputs($s_e, \epsilon$) ;
16     end
```

At each iteration the algorithm checks whether the reset state is in the set $\bar{s}_r \circ \pi_l$ of the current tree. If so, the first requirement is met and the algorithm has completed the search successfully. If not, a new "error" state is defined that will enforce the third requirement and the algorithm is called recursively.

Algorithm 1 shows the pseudo-code of the recursive algorithm. The algorithm takes initial states and error states of the processor as inputs, and outputs a vector of generated inputs to the processor. It starts from an error state s_e. If s_e is in the set of initial states S_I, the algorithm stops; otherwise, it uses symbolic execution and searches, starting from a fully symbolic state, for a path to the error state s_e in a single clock cycle.

In the `FindInputs` function, if the algorithm finds an input i that can transit the processor from an initial state s_0 to the current error state s, the search is finished and the algorithm outputs the vector $\vec{i}$. Otherwise, if the algorithm finds an input i that can transit the processor from a state $\bar{s}$ to the current error state s, it sets $\bar{s}$ as the new error state, and recursively finds previous inputs. If no inputs can be found, the algorithm returns ERROR.

4.3 Managing Complexity

In order to meet the third requirement of our strategy, we would ideally find all possible leaf nodes in tree $\mathcal{E}_{j-1}$ that are consistent with the path identified in tree $\mathcal{E}_j$. However, the complexity of this method is similar to forward symbolic execution (as discussed below). 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. We adopt a light-weight approach, sacrificing completeness for speed: after each iteration, we find concrete valuations to a subset of the stateful variables 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.

In forward symbolic execution, in the first clock cycle, a tree with N_f leaves will be explored. In the second clock cycle, the whole tree must be explored again, once for each of the N_f leaves. Exploring forward M clock cycles has complexity $O(N_f^M)$. Finding the desired sequence of trees using forward symbolic execution suffers from an exponentially growing set of states to explore and long queries to the solver.

The complexity for the recursive strategy with 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 variables are made symbolic (as opposed to only input variables) which increases the number of paths to explore. On the other hand, in general only $j < N_b$ paths are explored because exploration stops once the desired identified leaf node is found. Compare this to forward symbolic execution in which all paths must be explored through multiple clock cycles until one path is found that reaches the final error state. Although the recursive strategy imposes some additional overhead, it is overall more efficient because it can prune many of the repeated paths compared to forward symbolic execution.

4.4 Convergence Toward Reset

There is no guarantee that the search will converge to the initial state, s_0. In order to steer the search toward s_0, we provide two heuristics to identify and eliminate paths that are less likely to lead back to s_0.

Distance from Reset We found that a particular state is more likely to be reachable from reset within a few clock cycles if the *empirical distance* of that state from the reset state is small.

We define the *empirical distance* between two states as a count of stateful `regs` whose valuations differ in the two states. Specifically, we can represent a processor state s as a vector of bitstrings $s = \langle r_0, r_1, \ldots, r_{|s|} \rangle$ where each r_i is a stateful `reg` of the design. Empirical distance between two states s_1 and s_2 is calculated by:

$$d(s_1, s_2) \doteq \sum_{i=0}^{|s|} (\texttt{ITE}(r_i^1 \neq r_i^2, 1, 0)), \tag{3}$$

where r^1 is a stateful `reg` of s_1, r^2 is a stateful `reg` of s_2, and $|s|$ is the number of stateful `regs` in any state.

```
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 2. A control flow related security-critical bug from OR1200 processor Bugzilla.

```
l.movhi r16 0x8000
l.nop
l.sfgtu r16 r0
```

Listing 3. The exploit program generated by the recursive searching symbolic execution engine.

We set a threshold $0 < k < |s|$ and at each iteration j, we choose a concrete valuation s from $\bar{s}_{r,j} \circ \pi_{l,j}$ and calculate the empirical distance $d(s, s_0)$ between the chosen concrete state and the reset state. If the distance is above the threshold, we abort the current iteration, back track to the previous iteration, and choose a different node n_l. Otherwise, we continue with our recursive search.

Loops At each new iteration j, the set of processor states possible for step j may include states found in previous iterations, in which case the search may have entered a loop. Let $S_j = \bar{s}_{r,j} \circ \pi_{l,j}$ be the set of possible concrete states found in iteration j and let S_{j+1-m} be the set of possible concrete states found in iterations $j + 1$ through m. There are four possibilities:

- $S_j = S_{j+1-m}$
- $S_j \subset S_{j+1-m}$
- $S_j \supset S_{j+1-m}$
- $S_j \cap S_{j+1-m} = \{\}$

In any of the first three cases, there is the possibility that we are searching in a loop, and will never reach the reset state. To guard against this, while keeping the complexity of our strategy feasible, we introduce the following heuristic. In the first iteration, we define the set $S_m = \{s_m\}$, where s_m is the chosen concrete valuation for the last tree $\mathcal{E}_m$ in our sequence of trees. In each subsequent iteration j, we first choose the concrete valuation s_j and then check whether $s_j \in S_m$. If it is, we continue the symbolic execution until we find an s_k, such that $s_k \notin S_m$, and we update our set: $S_m := S_m \cup \{s_k\}$. Otherwise we update our set: $S_m := S_m \cup \{s_j\}$ and continue.

5 Case Study: OR1200 processor

In this section, we demonstrate the use of our recursive strategy with hardware-oriented symbolic execution to generate replayable exploits for three security vulnerabilities in the OR1200 processor. The OR1200 processor is an open-source 32-bit implementation of the OpenRISC 1000 architecture with Harvard microarchitecture, 5-stage integer pipeline, virtual memory support, and basic DSP capabilities. It is popular in research prototypes and it has been used in commercial products.

We implement our recursive algorithm using KLEE [14], a mature symbolic execution engine, as the base. We first use Verilator [4], a simulation tool, to translate the processor source code, which is written in Verilog or SystemVerilog, to C++. We then add the security-critical assertions collected from prior work [21, 40] to the design and compile the newly translated design to LLVM bytecode using the Clang compiler [1]. Finally, we run the design on our modified version of KLEE to generate exploits. The experiments are performed on a machine with Intel Xeon E5-2620 V3 12-core CPU (2.40GHz) and 62G of RAM.

5.1 A Control Flow Bug

Listing 2 shows a security-critical bug from OR1200 processor Bugzilla (Bugzilla #51 [2]). 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`. For most cases, the implementation of `a_lt_b` works correctly, but the bug introduces an error for the `l.sfgtu` instruction. According to the OpenRISC specification [26], for the `l.sfgtu` instruction (`l.sfgtu rA rB`), the contents of general-purpose register `rA` and `rB` are compared as unsigned integers. If the contents of the first register are greater than the contents of the second register, the compare flag is set; otherwise the compare flag is cleared. However, with this bug, if the highest bit in register rA is 1, even if rA is greater than rB, the compare flag will not be set. In this way, the attacker can control which branch to execute.

The security-critical bug violates the security-critical assertion that the comparison flag should be set correctly. We add the security-critical bug back to the OR1200 processor and the security-critical assertions to the design. We then run our symbolic execution engine with recursive search strategy. Listing 3 shows the generated exploit that triggers the security bug.

The symbolic execution engine first sets all the inputs and stateful signals to symbolic values. It runs and searches for an assertion violation on the whole processor design. In the first iteration, the generated instruction is: `l.sfgtu r16 r0`. The stateful `regs` that do not have their reset valuation are: `GPR16`, and `pcreg_select`. The engine then sets these two values as a new assertion for the second iteration. The second instruction it generates is: `l.nop`. The stateful `reg` that does not have its reset valuation is: `GPR16`. The engine then sets this value as a new assertion for the third iteration. The third instruction generated is: `l.movhi r16 0x8000`. All stateful `regs` have their reset valuation now. The engine

```
always @(posedge clk) begin
  if (!id_flushpipe & !id_freeze)
    sp_return_counter <= (sp_return_counter == 6'd50) ?
    sp_return_counter : sp_return_counter + 6'd1;
end
......
assign if_insn = (sp_return_counter == 6'd50) ? {6'h11,
10'h0, 5'h9, 11'h0} : no_more_dslot | rfe | if_bypass ?
{`OR1200_OR32_NOP, 26'h041_0000} : saved ? insn_saved :
icpu_ack_i ? icpu_dat_i : {`OR1200_OR32_NOP, 26'h061_0000};
```

Listing 4. A security-critical bug about instruction integrity.

stops and outputs the generated instuctions reversely (Listing 3). The total CPU time required for generating this exploit is 9m40s. The exploit is replayable on an FPGA board.

5.2 An Instruction Integrity Bug

Listing 4 shows a security-critical bug from the SPECS project [21]. This bug contaminates the instruction integrity by modifying the instruction to be executed, and exits the current program execution. The code snippets are from the Instruction Fetch (or1200_if) module and the Control (or1200_ctrl) module in OR1200 processor. The `always` block sets a counter which counts the number of instructions that are executed but not flushed away. If the number of instructions executed reaches 50, the `assign` statement will modify the instruction that got fetched from memory. In this example, the code silently modifies the fectched instruction to `l.jr r9`, which means jumping to the address stored in the link register (`r9`). In this way, the attacker can return and exit the current program execution regardless of the original instruction fetched from memory.

This bug violates the security property that instructions should not be changed in the pipeline. Adding back the security-critical bug and the corresponding security-critical assertion, our recursive search with symbolic execution successfully generates an exploit program of 50 instructions. The total CPU time required for generating this exploit is 2m29s. The exploit is replayable on an FPGA board.

5.3 An Exception Related Bug

Listing 5 shows another security-critical bug from the SPECS project [21]. This bug is related to exception handling. The EPCR register in OR1200 processor stores the return address for the exception handler, which is the program counter of the current execution. This bugs modifies the EPCR to an address of the attacker's choice. With this bug, when returning from the exception handler, the processor will jump to a malicious address. As it is returning from the exception handler, the processor is still in the high privilege. Thus, the attacker can execute a piece of chosen code with high privilege. This

```
`ifdef OR1200_EXCEPT_SYSCALL
  14'b00_0000_01??_????: begin
    except_type <=  `OR1200_EXCEPT_SYSCALL;
    epcr <=  ex_dslot ?     // Bug Free Version
    epcr <=  1'b1 ? 32'hdeadbeef : ex_dslot ? // Buggy
    Version
       wb_pc : delayed1_ex_dslot ?
       id_pc : delayed2_ex_dslot ?
       id_pc : id_pc;
```

Listing 5. A security bug related to exception handling.

bug violates the security property that privilege should escalate correctly.

With the security-critical bug and the assertion in the processor design, our recursive algorithm with symbolic execution successfully generates an exploit program of 1 instruction. The exploit is replayable on an FPGA board. Note that this bug cannot be found by the commercial model checking tool, Cadence IFV.

6 Related Work

6.1 Weakest Precondition

Weakest precondition has been used for program verification and vulnerability discovery [9, 11–13]. The weakest precondition for a program with respect to a vulnerability condition is a boolean formula over the initial state which is true for all inputs which cause the program to terminate in a final state satisfying the vulnerability condition [13]. Our calculation of preconditions at each iteration of the recursive search is different from weakest precondition in that we generate a per-path precondition rather than a single boolean formula representing a precondition over the whole program.

6.2 Hardware Symbolic Simulation

Software symbolic execution [14–17, 20] explores program paths with symbolic inputs [24]. Applying this technique to hardware designs for verification and testing has also been studied [30, 33]. STAR [30] 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 [30]. PATH-SYMEX is a forward symbolic execution engine that takes in ANSI-C interpretation of the RTL code [33]. Its application is limited to small RTL designs.

6.3 Backward Symbolic Execution in Software

Directed symbolic execution [31] and execution synthesis [38] use guided symbolic execution to increase the probability of executing paths of interest. In software, backward symbolic execution has been studied to solve the goal-reachability problem [31, 36]. Otter [31] developed the call-chain-backward symbolic execution which begins at a target line and proceeds backward to the start state. Application to real-world software raises many challenges such as complicated arithmetic (such as floating point), external method calls, and data dependent loops [19, 35, 36].

6.4 Typed Hardware Description Languages

A body of work has emerged on developing new or extending current hardware description languages for enforcing security policies [18, 27, 28, 39]. Although these works can prove that a hardware design meets the information flow security policies, they cannot verify those designs that are not already implemented with these languages.

7 Conclusion

We have presented the formalization of the recursive strategy with hardware-oriented symbolic execution to find, and generate exploits for, vulnerabilities in hardware designs. Our approach allows for a targeted recursive search through a possibly infinite set of infinitely long possible execution traces to find error states. We demonstrate the approach and generate three exploit programs for security-critical bugs in OR1200 processor, one of them cannot be found by the commercial model checking tool, Cadence IFV.

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 Grant No. CNS-1651276. Any opinions, findings, conclusions, and recommendations expressed in this paper are solely those of the authors.

References

[1] [n. d.]. Clang: a C language family frontend for LLVM. https://clang.llvm.org/
[2] [n. d.]. Comparison wrong for unsigned inequality with different MSB. http://bugzilla.opencores.org/show_bug.cgi?id=51
[3] [n. d.]. Reading privileged memory with a side-channel. https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html
[4] [n. d.]. Verilator. https://www.veripool.org/wiki/verilator.
[5] 2013. Revision Guide for AMD Family 16h Models 00h-0Fh Processors. *Product Revision* (2013). http://support.amd.com/TechDocs/51810_16h_00h-0Fh_Rev_Guide.pdf
[6] 2014. Intel Core i7-600, i5-500, i5-400 and i3-300 Mobile Processor Series. *Specification Update* (2014). http://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/core-mobile-spec-update.pdf
[7] 2015. Xen Security Advisory CVE-2015-5307,CVE-2015-8104 / XSA-156. http://xenbits.xen.org/xsa/advisory-156.html.
[8] 2017. Intel Skylake/Kaby Lake processors: broken hyper-threading. https://lists.debian.org/debian-devel/2017/06/msg00308.html.
[9] Mike Barnett and K. Rustan M. Leino. 2005. Weakest-precondition of Unstructured Programs. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '05)*. ACM, New York, NY, USA, 82–87. https://doi.org/10.1145/1108792.1108813
[10] Clark W. Barrett, David L. Dill, and Aaron Stump. 2002. Checking Satisfiability of First-Order Formulas by Incremental Translation to SAT. In *International Conference on Computer Aided Verification*. Springer-Verlag. http://theory.stanford.edu/~barrett/pubs/BDS02-CAV02.pdf
[11] D. Brumley, P. Poosankam, D. Song, and J. Zheng. 2008. Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*. 143–157. https://doi.org/10.1109/SP.2008.17
[12] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. 2008. Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy (SP '08)*. IEEE Computer Society, Washington, DC, USA, 143–157. https://doi.org/10.1109/SP.2008.17
[13] David Brumley, Hao Wang, Somesh Jha, and Dawn Song. 2007. Creating Vulnerability Signatures Using Weakest Preconditions. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium (CSF '07)*. IEEE Computer Society, Washington, DC, USA, 311–325. https://doi.org/10.1109/CSF.2007.17
[14] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. http://klee.github.io/
[15] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. 2006. EXE: Automatically Generating Inputs of Death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)*. 322–335. http://doi.acm.org/10.1145/1180405.1180445
[16] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. 2012. Unleashing Mayhem on Binary Code. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (SP '12)*. IEEE Computer Society, Washington, DC, USA, 380–394. https://doi.org/10.1109/SP.2012.31
[17] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. 2013. FIE on Firmware: Finding Vulnerabilities in Embedded Systems Using Symbolic Execution. In *Proceedings of the 22nd USENIX Security Symposium*. USENIX, Washington, D.C., 463–478. https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/davidson
[18] Andrew Ferraiuolo, Rui Xu, Danfeng Zhang, Andrew C. Myers, and G. Edward Suh. 2017. 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 (ASPLOS '17)*. ACM, New York, NY, USA, 555–568. https://doi.org/10.1145/3037697.3037739
[19] Arnaud Gotlieb Florence Charreteur. 2010. Constraint-Based Test Input Generation for Java Bytecode. In *Proceedings of the 2010 IEEE 21st International Symposium on Software Reliability Engineering*. ACM, 131–140.
[20] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2012. SAGE: Whitebox Fuzzing for Security Testing. *Queue* 10, 1, Article 20 (Jan. 2012), 8 pages. https://doi.org/10.1145/2090147.2094081
[21] Matthew Hicks, Cynthia Sturton, Samuel T. King, and Jonathan M. Smith. 2015. SPECS: A Lightweight Runtime Mechanism for Protecting Software from Security-Critical Processor Bugs. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. ACM, New York, NY, USA, 517–529. https://doi.org/10.1145/2694344.2694366

[22] K. Karnane and C. Goss. 2015. *Automating root-cause analysis to reduce time to find bugs by up to 50%.* Technical Report. Cadence Design Systems. www.cadence.com/rl/Resources/whitepapers/indago_debug_platform_wp.pdf

[23] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (July 1976), 385–394. https://doi.org/10.1145/360248.360252

[24] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (July 1976), 385–394. http://doi.acm.org/10.1145/360248.360252

[25] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. Spectre Attacks: Exploiting Speculative Execution. *ArXiv e-prints* (Jan. 2018). arXiv:1801.01203

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

[27] Xun Li, Vineeth Kashyap, Jason K. Oberg, Mohit Tiwari, Vasanth Ram Rajarathinam, Ryan Kastner, Timothy Sherwood, Ben Hardekopf, and Frederic T. Chong. 2014. Sapper: A Language for Hardware-level Security Policy Enforcement. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. ACM, New York, NY, USA, 97–112. https://doi.org/10.1145/2541940.2541947

[28] Xun Li, Mohit Tiwari, Jason K. Oberg, Vineeth Kashyap, Frederic T. Chong, Timothy Sherwood, and Ben Hardekopf. 2011. Caisson: A Hardware Description Language for Secure Information Flow. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 109–120. https://doi.org/10.1145/1993498.1993512

[29] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. *ArXiv e-prints* (Jan. 2018). arXiv:1801.01207

[30] Lingyi Liu and Shabha Vasudevan. 2009. 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, 32–37.

[31] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks. 2011. Directed Symbolic Execution. In *Proceedings of the 18th International Static Analysis Symposium*, Eran Yahav (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 95–111. https://doi.org/10.1007/978-3-642-23702-7_11

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

[33] Rajdeep Mukherjee, Daniel Kroening, and Tom Melham. 2015. Hardware Verification using Software Analyzers. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE.

[34] Samir Palnitkar. 2003. *Verilog®Hdl: A Guide to Digital Design and Synthesis, Second Edition* (second ed.). Prentice Hall Press, Upper Saddle River, NJ, USA.

[35] Gul Agha Peter Dinges. 2004. Targeted test input generation using symbolic-concrete backward execution. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 31–36.

[36] S. J. Fink S. Chandra and M. Sridharan. 2009. Snugglebug: a powerful approach to weakest preconditions. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*. ACM, 363–374.

[37] Frank Vahid. 2010. *Digital Design with RTL Design, Verilog and VHDL* (2nd ed.). Wiley Publishing.

[38] Cristian Zamfir and George Candea. 2010. Execution Synthesis: A Technique for Automated Software Debugging. In *Proceedings of the 5th European Conference on Computer Systems (EuroSys)*. 321–334. http://doi.acm.org/10.1145/1755913.1755946

[39] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. 2015. 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 (ASPLOS '15)*. ACM, New York, NY, USA, 503–516. https://doi.org/10.1145/2694344.2694372

[40] Rui Zhang, Natalie Stanley, Christopher Griggs, Andrew Chi, and Cynthia Sturton. 2017. 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 (ASPLOS '17)*. ACM, New York, NY, USA, 541–554. https://doi.org/10.1145/3037697.3037734