SEIF: Augmented Symbolic Execution for Information Flow Verification

Kaki Ryan
UNC Chapel Hill
Chapel Hill, NC, USA
kakiryan@cs.unc.edu

Matthew Gregoire
UNC Chapel Hill
Chapel Hill, NC, USA
mattyg@cs.unc.edu

Cynthia Sturton
UNC Chapel Hill
Chapel Hill, NC, USA
csturton@cs.unc.edu

ABSTRACT

We present *SEIF*, an exploratory methodology for information flow verification based on symbolic execution. SEIF begins with a statically built overapproximation of the information flow through a design and uses guided symbolic execution to provide a more precise picture of how information flows from a given set of security critical signals. SEIF can recognize and eliminate non-flows with high precision and for the true flows can find the corresponding paths through the design state with high coverage. We evaluate SEIF on two open-source CPUs, an AES core, and the AKER access control module. SEIF can be used to find counterexamples to information flow properties, and also to explore all flows originating from a source signal of interest. SEIF accounts for 86–90% of statically identified possible flows in three open-source designs. SEIF's search strategies enable exploring the designs for 10-12 clock cycles in 4-6 seconds on average, demonstrating that this new exploratory style of information flow analysis can be practical.

CCS CONCEPTS

• **Security and privacy → Security in hardware**.

KEYWORDS

information flow, symbolic execution, hardware security, path coverage

ACM Reference Format:
Kaki Ryan, Matthew Gregoire, and Cynthia Sturton. 2023. SEIF: Augmented Symbolic Execution for Information Flow Verification. In *Hardware and Architectural Support for Security and Privacy 2023 (HASP '23), October 29, 2023, Toronto, Canada.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3623652.3623666

1 INTRODUCTION

Analyzing how information flows through a hardware design is critical to verifying the security of the design [4, 5, 11, 12, 14, 19, 33–35, 37, 38, 42, 43, 49, 50, 57, 58]. Unwanted flows of information to or from a signal in the design can violate desired security policies in the form of access authorization violations [45, 46], memory leakage vulnerabilities [17, 24], and possible privilege escalation vulnerabilities [55]. Existing research and commercial information flow analysis tools use model checking [1], taint tracking [32], and symbolic analysis [28] to detect these vulnerabilities early in the hardware verification life-cycle [22, 30].

However, these conventional methods of information flow verification require the engineer to provide well-specified security properties to the verification engine, which in turn requires the engineer to have signal-level and often cycle-accurate knowledge of where information should and should not be allowed to flow. Developing these properties is challenging and time-consuming. Furthermore, verifying a single property about, for example, information leakage from a security-critical signal may say nothing about whether information is leaking to another unprivileged area of the design not covered by the property. In response to these challenges, a new style of analysis is emerging that allows the developer to explore information flow through the design, rather than prove or disprove a set of information flow properties [21, 39].

We present *SEIF* (pronounced "safe"), an information flow analysis methodology and tool in this new exploratory style that uses symbolic execution to give the engineer a view of how information flows through a design. SEIF takes as input a hardware design in RTL Verilog and a set of security-critical signals of interest, and produces, for each signal, a set of information flow paths that describe how information flows from the given signal, which signals can be influenced by the given signal, and what sequence of design inputs will drive execution to produce the flow.

Symbolic execution is a path-based symbolic analysis that can provide a precise information-flow analysis. The challenge is that for even modestly sized hardware designs, the number of paths to explore is too large. To mitigate this problem, we use an augmented signal connectivity graph built by parsing the RTL Verilog that acts as a guide to the symbolic execution engine. The graph provides an overapproximation of all possible information flows through the design, and for each possible flow, provides a sequence of landmark points in the hardware design that execution must reach in order to realize the information flow. Additionally, we develop and evaluate four search strategies with the goal of making the search space more tractable.

SEIF is able to provide traditional counterexamples to information flow properties and will return, when applicable, the set of all found flows that violate a property, which can be helpful when patching a bug or doing the root-cause analysis for a violation. Outside of property violations, SEIF will return all found flows from the given source signal, pointing the security engineer to areas of the design that may require attention. The flows that SEIF finds may represent flaws in the RTL that are exploitable post-deployment. In those cases, SEIF can find flaws that occur by benign human error in the specification, design, or implementation phases.

HASP '23, October 29, 2023, Toronto, Canada

ACM ISBN 979-8-4007-1623-2/23/10.
https://doi.org/10.1145/3623652.3623666

In our toolflow, we use the parsing and static analysis portion of the hyperflow graph tool [39] to build the signal connectivity graph, and we use Sylvia, an open-source symbolic execution engine [47], to perform the symbolic analysis.

Our contributions are:

- Define *SEIF*, an augmented symbolic execution methodology for information flow analysis.
- Implement the methodology and search heuristics on top of the Sylvia symbolic execution engine [47].
- Evaluate SEIF on four open-source designs.

2 PRELIMINARIES

It is useful to keep in mind three models: the state diagram of the design showing machine states and transitions between them; the *Hyperflow (HF)* graph, which is the labeled, directed signal-connectivity graph [39]; and the symbolic execution (SE) tree, showing execution paths through the RTL. We first introduce a fragment of Verilog RTL as a toy example to help illustrate the three models.

2.1 Toy Example

The code snippet of Figure 1 shows a flow from an input, `secret`, to an output, `led`. The flow is guarded by an internal, state-holding variable and the secret will only flow to the LED output in the clock cycle after `count = 3`. Note that with non-blocking assignments ("<=") all right-hand side expressions are calculated at the same time and assignments take effect at the next clock cycle. Blocking assignments ("=") take effect immediately.

```
1  always @(posedge clk) begin
2     if (enable) begin
3        prev <= count;
4        count <= count + 1;
5     end
6
7     if (count == 3)
8        guard <= secret;
9     else
10       guard <= 0;
11 end
12
13 assign led = (prev == 3) ? guard : 0;
```

Figure 1: Toy example. `clk`, `enable`, and `secret` are input wires. `count`, `prev`, and `guard` are state-holding regs. Not shown is the initialization, which sets `count`, `prev`, and `guard` to 0. `led` is an output wire. `secret` flows through `guard` to `led` after four clock cycles.

2.2 State Diagram

Figure 2 shows one possible sequence of state transitions for the toy example. In this sequence, the initial state ($s_0 = \langle \texttt{prev} = 0, \texttt{count} = 0, \texttt{guard} = 0\rangle$ with output $\texttt{led} = 0$) transitions to state $s_1 = \langle \texttt{prev} = 0, \texttt{count} = 1, \texttt{guard} = 0\rangle$ when `enable` is high on the positive clock edge. Other sequences are possible, for example, if `enable` remains low on the first positive clock edge.

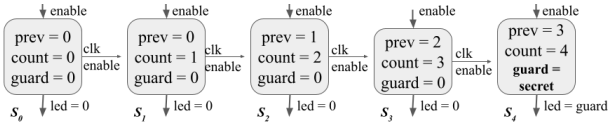

Figure 2: State transitions of the toy example (Figure 1) in which information flows from `secret` to `led`.

2.3 Symbolic Execution

In symbolic execution, concrete input values are replaced with abstract symbols. The design is executed using the symbols in place of concrete values. When a branch point (e.g., `if(enable)`) is reached, both paths are separately explored. For each path, the branching condition that must be true for that path (e.g., `enable == 1`). is maintained in the *path condition*. At the end of a single path of symbolic execution, satisfying assignments to the constraints in the path condition can be used as concrete input values to drive concrete execution down that same path.

Symbolic execution is modeled as a tree of nodes. Each node is associated with a line of code in the RTL and contains a symbolic state and path condition. The tree that results from the complete exploration of a design represents one clock cycle of execution. A path from the root node to any leaf node in that tree corresponds to a single state transition of the design. The number of paths to explore grows quickly. For example, the symbolic execution of the design in Figure 1 for the four clock cycles necessary for `secret` to flow to `led` would yield the tree of nodes shown in Figure 3.

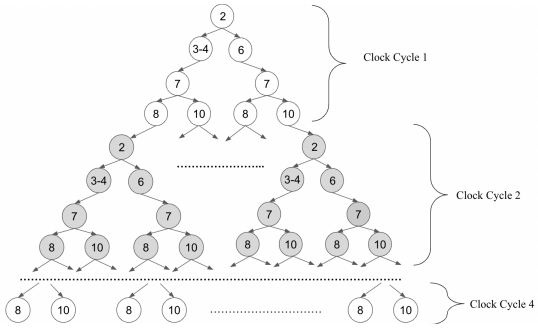

Figure 3: Symbolic execution tree of the design in Figure 1 after four clock cycles.

2.4 Hyperflow Flow (HF) Graph

The Hyperflow (HF) graph [39] is a labeled, directed graph that captures signal connectivity from the RTL Verilog. Nodes represent the variables (`wires` and `regs`) of the design, and edges indicate a possible flow of information from one variable to another. An edge (v_1, v_2) exists when there is an explicit flow in the form of an assignment in the RTL from v_1 to v_2 (e.g., v_2 <= v_1), or when there is an implicit flow because v_1 appears in a condition (e.g., `if`(v_1)), and v_2 appears on the left-hand side of an assignment in either branch. The edge is labeled with the line number of the relevant Verilog statement and lists the surrounding conditions in the code that must be true for the information flow to take place.

For example, in Figure 4, which shows the HF graph for the code in Figure 1, the edge (`secret`, `guard`) is labeled (although not shown for space) with the condition that `count == 3`.

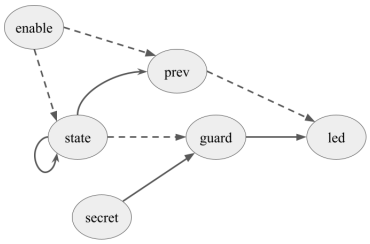

Figure 4: A HF graph for the code in Figure 1. Dashed lines represent implicit flows of information and solid lines represent explicit flows. Labels are omitted for space.

Note that the HF graph has no inherent notion of timing or clock cycles. For example, the fact that it takes at least three clock cycles to reach a state in which `count == 3` is not discernible from the graph. In addition, multi-hop paths through the HF graph may not correspond to viable information flows. Consider the code in Figure 1, but with the last line replaced with the following:

```
assign led = (prev == 2) ? guard : 0;
```

The HF graph is still as shown in Figure 4, although the label for edge (`guard`, `led`) would be different, but the path from `secret` to `guard` to `led` does not correspond to any flow of information through the design. This is because when `prev == 2`, the value in `guard` is 0, not the secret value.

There are two reasons why a path through the HF graph may not correspond to a true information flow. Either (as seen above) the sequence of conditions needed for each edge cannot be satisfied, or a path through the HF graph from x to y may not correspond to a true flow of information (e.g., $y = x \oplus x$).

3 SEIF: SYMBOLIC EXECUTION FOR INFORMATION FLOW

Given a design and input signals of interest, the goal is to find how information flows from the sources through the design during execution. Our approach harnesses the HF graph to guide symbolic execution to find those flows. A high-level view of the SEIF workflow is shown in Figure 5. Once the HF graph is generated, the analysis proceeds in three main phases: pruning globally unrealizable paths, symbolically executing the design to find realizable paths through the design, and analyzing each found path to find true paths of information flow. In the following sections, we describe each phase.

3.1 Pruning Globally Unrealizable Paths from the HF Graph

In the first phase, the goal is to eliminate paths through the HF graph that are easily falsified before moving on to the next, more expensive phase. Consider the example code in Figure 6. The variable `temp` carries the input `secret` only when the input signal `enable` is high. The `secret` information is conditionally passed on to `result`

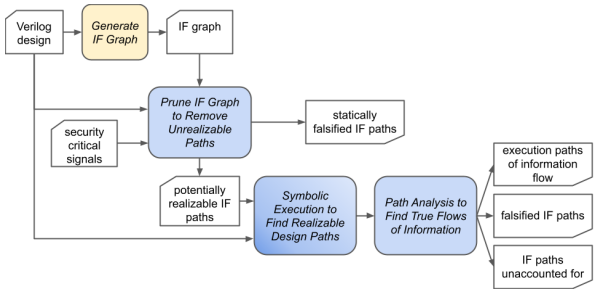

Figure 5: SEIF workflow. We use an existing tool [39] to generate the HF graph. We augment an existing symbolic execution engine [47] to implement our search strategies and heuristic.

and from there to `led2`. The corresponding HF graph is shown in Figure 7. While the HF graph appears to show a flow of information from `secret` to `led2` via `temp`, the constraints for edges (`secret`, `temp`) and (`temp`, `result`) require `enable` to be high and low, respectively. Since both edges must occur in the same clock cycle, this flow cannot be realized.

```
1  wire temp = (enable) ? secret : 0;
2  always @(posedge clk) begin
3     if (enable) begin
4        result <= 0;
5        prev <= count;
6        count <= count + 1;
7     end else
8        result <= temp;
9
10    if (count == 0)
11       guard0 <= secret;
12    else if (clear)
13       guard0 <= 0;
14    else
15       guard0 <= guard0;
16
17    if (count == 3)
18       guard <= guard0;
19    else begin
20       guard <= 0;
21 end
22
23 assign led = (prev == 3) ? guard : 0;
24 assign led2 = result;
```

Figure 6: A design demonstrating globally unrealizable paths and the challenges of stalling.

This analysis requires knowing where clock cycle boundaries are. In the HF graph, an edge corresponding to a non-blocking assignment (e.g., `result <= temp`) denotes a clock cycle boundary. When `result` is updated in one clock cycle, the updated value can be read in the next clock cycle.

At the start of this phase, the *information flow* (IF) path is divided into *segments* with one division happening at each non-blocking assignment in the path. If a path has n non-blocking

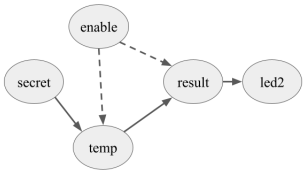

Figure 7: The partial HF graph for the code shown in 6, showing only the paths through `temp`. Although the graph shows a path from `secret` to `led2`, an SMT query finds that the constraints along the path will never be co-satisfiable.

assignments, it has $n + 1$ segments. Each segment can contain a sequence of hops in the HF graph, representing both implicit and explicit flows. For example, consider the IF path $\langle$(`secret`, `temp`), (`temp`, `result`), (`result`, `led2`)$\rangle$ in Figure 7. This path has two segments: $\langle$(`secret`, `temp`), (`temp`, `result`)$\rangle$ and $\langle$(`result`, `led2`)$\rangle$.

For every segment in a given IF path, the conditions involved in that segment are collected and checked for co-satisfiability. If the hops in any one segment have mutually contradictory constraints, that path is discarded. In Figure 7, the segment $\langle$(`secret`, `temp`), (`temp`, `result`)$\rangle$ has contradictory constraints, as the first hop requires that `enable` is high, while the second hop requires it to be low.

This pruning analysis is sound—only unrealizable paths are discarded—as long as the co-satisfiability check considers only state-holding signals and input signals in the satisfiability query, as these signals do not change value in the middle of a clock cycle.

3.2 Symbolic Execution to Find Paths through the Design

In the second phase, the goal is to find true paths through the design for each remaining path in the HF graph. We use symbolic execution to find a sequence of machine states and a corresponding sequence of input signals (for example, as seen in Figure 2) that aligns with the HF graph path.

3.2.1 Symbolic Execution Guided by IF Path Segments. The segment analysis done in the first phase provides information about where the clock cycle boundaries lie, and which lines of code must execute for each hop in a segment. In each clock cycle, the symbolic execution engine is restricted to following only those design paths which include the lines of code that must be executed for the current IF-path segment to be realized. In Figure 1, for example, the symbolic execution engine would consider only paths which take the branch at line 8, when `count == 3`, significantly reducing the search space.

However, there may still be many possible paths through the design to consider, only some of which allow the complete IF path to be realized. Consider the first clock cycle of symbolic execution, shown at the top of Figure 3. The path of interest, annotated by lines of code, is

$$\langle(\texttt{secret}, \texttt{guard})_{\text{line 8}}, (\texttt{guard}, \texttt{led})_{\text{line 13}}\rangle.$$

In Figure 3, it would appear that two of the four possible paths achieve the desired flow in clock cycle 1. But annotations in the HF graph tell us that the sequence of conditions $(\texttt{count == 3})_{s3}$, $(\texttt{prev == 3})_{s4}$ needs to be met. For this to happen, lines 3–4 need to execute in the first four clock cycles and lines 8, 13 need to execute in only the fourth clock cycle. While this is clear to see when examining the state transition diagram (Figure 2), there is nothing in the HF graph, or even the code itself, indicating that it will take four clock cycles to realize this flow.

3.2.2 Pruning Unrealizable Paths at Clock Cycle Boundaries. At each clock cycle, the engine first checks the co-satisfiability of the conditions required in the current IF segment, similar to the check done to prune globally unrealizable paths (Section 3.1). However, now the SMT query includes the current symbolic state along with the conditions required for the IF segment.

In our example from Figure 1, at the start of the initial clock cycle, the symbolic execution engine checks whether the condition required for the first hop in the HF graph (`count == 3`) is co-satisfiable with the initial symbolic state (in which `count == 0`). It is not, and the symbolic execution engine discards any paths that would include line 8, which is required for the first hop in the HF graph.[1] At this point, SEIF recognizes that realizing the first segment of the HF graph at the current state (state $s0$) is infeasible.

3.2.3 Stalling the IF Path to Advance to a New Machine State. The second strategy used by SEIF is to pause the search for realizing a segment of the IF path in order to advance the design to a next-state. In our example, the first segment of the HF graph cannot be realized from the initial reset state. SEIF symbolically executes the design for a single clock cycle, without considering the constraints required by the next IF path segment, to advance the design to a new state. SEIF then checks whether the HF graph segment can be realized from this new state.

There are many possible next-states and SEIF must find one that advances the design toward a state in which the next IF segment can be realized and also does not undo any prior progress along the IF path that has been made. We discuss search strategies for finding valuable next-states in the next section.

To demonstrate how prior progress can be undone, consider the IF path from `secret` to `guard0` to `guard` to `led` in Figure 6. To achieve the second flow segment, $\langle$(`guard0`, `guard`)$\rangle$, SEIF needs to first advance the design to a state $s' = \langle\texttt{count == 3}\rangle$. However, if the `clear` signal is set, a 0 would be written to `guard0` undoing the flow from `secret` to `guard0` in the prior IF path segment.

To prevent undoing prior progress SEIF *stalls*, which means symbolically executing the design for a single clock cycle to advance to a next-state, but without changing the position along the IF path. To do this, SEIF considers the node n in the IF path, in which information currently "resides." In our current example, this would be the node `guard0`. SEIF then uses the HF graph to find all edges incident to node n, which are labeled with lines of code corresponding to flows of information from variables in the design to n, and prevents explorations of any corresponding design paths.

SEIF handles the edge cases of self-loops and direct assignments of constants during symbolic execution by abandoning the execution paths in which they occur.

[1] Discarding these paths can be done prior to exploration of any paths in the current clock cycle, as the engine has information from the design's statically built control flow graph about which lines of code are included in which path.

3.2.4 Search Strategies. The search space for the symbolic execution is large. An IF path with n segments requires at least n clock cycles through the design. When stalling is needed, the number of clock cycles required is unbounded (but finite). Additionally, a single IF hop can correspond to many paths through the symbolic execution tree. We developed four search strategies to tackle the large search space.

Baseline 1: Continue / Stall Only. In this strategy, SEIF will either symbolically execute until a design path is found in which the segment conditions are satisfied (termed a *continue*), or will stall for some bounded number of cycles. For an IF path, SEIF exhaustively tries all possible *continue*, *stall* combinations, prioritizing shorter paths with fewer stalls.

Baseline 2: Backtracking Only. In this strategy SEIF begins by symbolically executing until a design path is found for the first segment. If the flow is found, SEIF moves to the next segment in the IF path. If at any segment SEIF runs out of design paths to try for that clock cycle, SEIF *backtracks* to an earlier segment to find a different design path that satisfies the same segment conditions.

Stalling with Backtracking. This strategy is a hybrid of baselines 1 and 2. For any given *continue*, *stall* pattern, after successfully executing consecutive *continues*, and reaching a *stall*, SEIF stalls for a bounded number of clock cycles and attempts to find execution paths where SEIF can make forward progress in the next segments. If all symbolic execution paths are explored, or SEIF reaches a pre-determined bound, it backtracks.

Stalling with Heuristic. This strategy builds on top of stalling with backtracking. Our heuristic relies on the *UNSAT core*, the subset of constraints in a SAT query for which no satisfying assignment exists. If SEIF stalls, it searches for a new machine state that will satisfy the conditions of the next IF path segment. In this case, SEIF pushes the symbolic state and the constraints from the next segment to the SMT solver, which returns the UNSAT core. For each path explored while stalling, SEIF checks if the UNSAT core became smaller. If it did, SEIF continues searching for a new machine state along the path. If it grows, SEIF prioritizes the next candidate stall path.

3.3 Semantic Analysis to Identify True Information Flows

For each path found in the prior phase, SEIF analyzes the semantics of the information flow to discard paths that represent viable design paths, but not true flows of information. SEIF discards explicit textual flows which do not represent information flows (e.g., `y <= x ⊕ x`), but cannot detect implicit textual flows that are not true information flows (e.g., `if (x ⊕ x)`).

In the case of reconvergent fan-out SEIF may or may not find the flow. This is because if an input signal splits off into different areas of the design (i.e. modules, `always` blocks), different bits of a signal may be written under different conditions. Based on the path conditions in the different areas of the design and contents of the writes, SEIF will only correctly account for reconvergent flows if it is able to exhaustively explore all paths in that clock cycle.

4 IMPLEMENTATION

We implemented SEIF using the open-source Sylvia symbolic execution engine [47] and using the static-analysis portion of the hyperflow graph tool [39]. SEIF is implemented in python3: Sylvia is a python-based engine and the HF graphs generated by the hyperflow graph tool can be loaded into a python environment. Sylvia implements the Verilog semantics according to the IEEE 1364-2005 standard. SEIF uses the Z3 SMT solver, via the python Z3 API, for SMT queries in all phases. The SEIF tool is publicly available at github.com/kakiryan/SEIF.

5 EVALUATION

We evaluate SEIF over four open-source designs to study its efficacy in producing information flows of a hardware design. The evaluation starts with two case studies (Sec. 5.2) demonstrating how SEIF can provide a richer information flow verification environment, providing supporting information alongside traditional property counterexamples. We then empirically address the following performance questions (Sec. 5.4-5.5): 1) How well does SEIF perform in fully explicating the complete set of information flow paths? 2) How effective are the search strategies at guiding SEIF to realizable paths through the design? We conclude with an investigation into the types of false positive flows eliminated by SEIF from the statically built HF graph.

5.1 Dataset and Experimental Setup

The Verilog designs used in the evaluation are the OR1200 [3], a 5-stage open-source RISC processor, openMSP430 [2], a synthesizable 16-bit microcontroller core often used in embedded systems; the AKER Access Control Wrapper (ACW) [45], and an AES implementation from TrustHub [25, 26]. The experiments are performed on a machine with an Intel Xeon E5-2620 V3 12-core CPU (2.40GHz, a dual-socket server) and 62G of available RAM.

5.2 Case Studies: Providing Feedback Beyond Traditional Counterexamples

The SEIF workflow starts with a Verilog RTL design and a set of source signals of interest. The signals can either be ones the engineer is interested in exploring, or can be taken from an information flow property. In the former case, SEIF will return a set of true information-flow paths through the design, originating at the given source, and if desired, a sequence of inputs to the design that will drive execution down that path. In the latter case, SEIF can be configured to add constraints to the solver specifying the preconditions of the property. If the property can be violated (within the clock bounds set by the engineer), SEIF will again return a set of true information-flow paths, along with the associated sequence of inputs, each of which is a counter-example to the property. We verify these counterexamples are replayable in simulation. Otherwise, SEIF will return a result of no violations found. Whether used for information flow exploration or for finding property violations, SEIF returns a set of information flow paths and summary information about the paths, including how many distinct information-flow paths are found, how many destination signals the source flows into, and for property verification tasks, how many violating paths are found.

The first case study examines all flows from the program counter in the MSP430 core. The program counter may be assumed to flow everywhere in the core, however, anyone with access to information from the program counter may be able to learn confidential information about the program executing. For this reason it is important that information from the program counter does not leak to the debug access port during normal operation. A property to that effect, slightly modified from the TrustHub Security Property/Rule Database [25, 26] is as follows. We use the notation =/=> to indicate no flow and $\rightarrow$ to indicate conditional flows.

```
(dbg_0.dbg_en_s == 1'b0) →
frontend_0.pc =/=> dbg_uart_txd
```

To verify this property, and gain additional insights into how information from the program counter flows through the design, we ran SEIF on the MSP430 with the source signal set to the program counter, and path condition constrained to match that of the property. The results are shown in Table 1. SEIF finds the known property counter-example given in the TrustHub database, and also finds 45 additional paths that violate the property. SEIF also reports that 41 sink signals are influenced by the program counter. On average, SEIF searches through just over 8 clock cycles and finds a counter-example path in less than a second.

Metric	Result
Total IF paths from source:	19060
Total sinks reachable from source:	41
Total IF paths violating security property:	58
Avg. time to produce a counterexample (s):	0.678
Avg. no of clock cycles explored:	8.13
Total realizable paths violating security property:	46

Table 1: Security Property Verification: Program Counter in MSP430

The second case study examines all flows from the key input in the AES module. In order to prevent key leakage through information side channels, it is important that information from the AES key does not leak to any unprivileged internal data registers [62]. If at least one bit of the key material flows into the same intermediate register that at least one bit of the plaintext flows into, that intermediate register is vulnerable to side-channel attacks. The properties to check this requirement for a specific intermediate register, `Drg[0]`, take the following form, where `Krg` is the secret key. [25, 26] :

$$\texttt{(clk\_count == 3)} \rightarrow Krg[i] \texttt{=/=> Drg[0]}$$

The results are shown in Table 2. SEIF finds the known property counter-example given in the TrustHub database, and also finds 24 additional paths that violate the property. SEIF reports that 39 sink signals are influenced by the AES key material. This information can be used by the engineer to do a quick spot-check that no surprising signals are showing up in this list. On average, SEIF searches through just over 4 clock cycles and finds a counter-example path in about half a second.

Metric	Result
Total IF paths from source:	61639
Total sinks reachable from source:	39
Total IF paths from source violating security property:	57
Avg. time to produce a counterexample (s):	0.505
Avg. no of clock cycles explored:	4.102
Total realizable paths violating security property:	25

Table 2: Security Property Verification: Secret Key in AES Implementation

5.3 Methodology for Empirical Evaluation

In the following experiments we look at the OR1200, MPS430, and ACW designs and analyze security-critical signals collected from known security properties. For the OR1200 and ACW, we identified 20 source signals each from properties developed in the security literature [13, 31, 52, 60, 61] [21, 46]. For the MSP430, we selected 10 signals analogous to those in the OR1200 security properties.

For each source signal there can be tens of thousands of IF paths. For the efficacy and performance evaluations in this and the next two sections, we analyze a subset of the total paths. For each source signal of interest, we randomly selected 300 paths from the HF graph for analysis.

We empirically determined the number of clock cycles to stall for each search strategy that involves stalling: baseline 1, bounded stalling with backtracking, and the UNSAT core heuristic (baseline 2 does not involve stalling). We used the OR1200 and 5 security-critical signals from our dataset for this analysis. Once we did not see any gains from stalling for an additional clock cycle, the bound was set. We experimentally found the bounds to be 5 cycles, 5 cycles, and 4 cycles for the respective three stalling-based heuristics.

5.4 Accounting for Paths in the HF Graph

We examine SEIF's ability to explicate paths in the HF graph, either by finding paths through the design that correspond to the HF path, or by eliminating the IF path as infeasible. Figure 8 summarizes the results. For 86% to 90% IF paths on average, SEIF either finds the corresponding path through the design or can falsify it – statically or during symbolic execution. The majority of accounted-for IF paths, 58% to 77% on average in the three designs, are true paths in the design, indicating that the static analysis done to build the HF graph is a decent first approximation of information flow through the design. We show the percentage of the IF paths for which SEIF returns a design path that starts at the reset state vs. a design path that starts at some intermediate state. Paths that start at the reset state are better for the engineer as they can be immediately replayed from the known reset state.

5.5 Evaluation of Search Strategies

In the following we evaluate the four search strategies. Figure 9 reports the percentage of IF paths found by each. As expected, the heuristic guided search outperforms the others in all three designs, improving over the baselines by 26% on average and over bounded stalling with backtracking by 11% on average. Baseline 2, which does not include stalling, is the least successful at finding

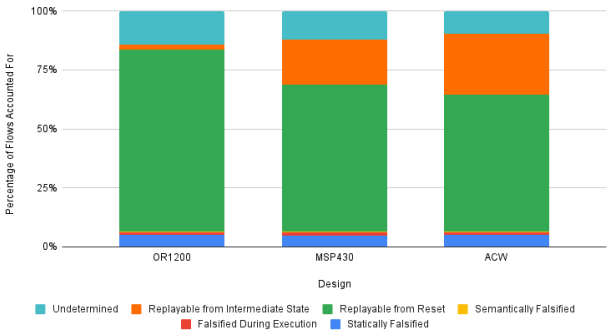

Figure 8: Accounting for IF Paths

corresponding paths in the design. This highlights the value of SEIF: many IF paths give an incomplete picture of a path through the design and include points where the design must advance to a new state before the IF path can continue. Without SEIF, it is up to the engineers to figure out how and whether to advance the design state.

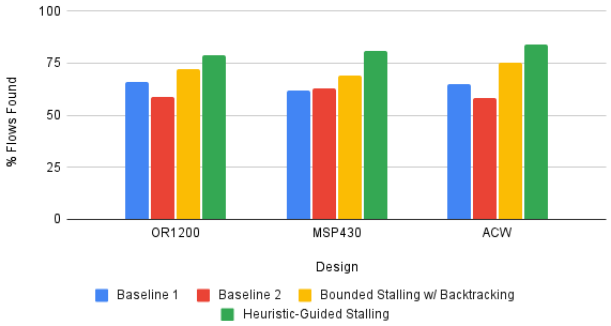

Figure 9: Finding Design Paths Corresponding to IF Paths

Figures 10 and 11 report on the performance of the four search strategies. The heuristic-guided search outperforms the other strategies, completing the search for each IF path in 3-6 seconds. Figure 12 shows that the amount of backtracking that is required is lowered when we incorporate bounded stalling. Adding the heuristic improves the efficacy of stalling and therefore decreases backtracking even further.

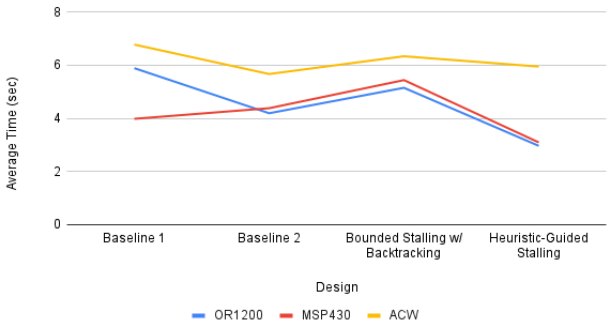

Figure 10: Time to Find Design Paths

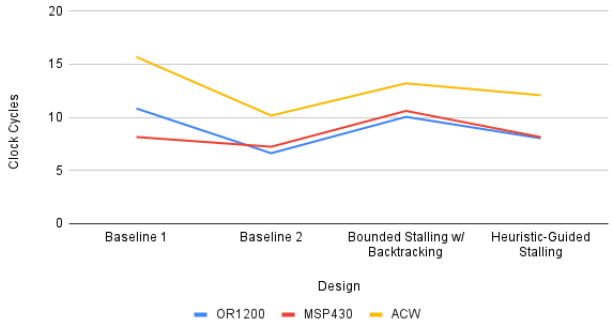

Figure 11: Clock Cycles to Find Design Paths

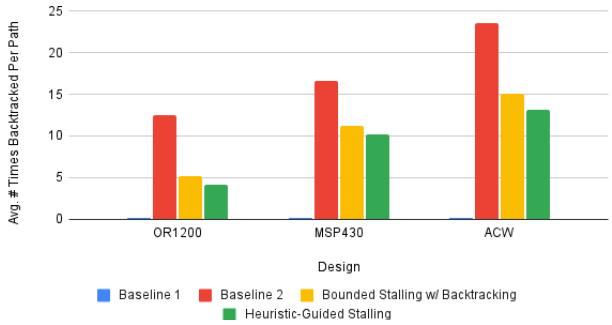

Figure 12: Frequency of Backtracking

5.6 Falsified Paths in the HF Graph

We examine how IF paths that do not correspond to information-flow paths through the design are falsified in Figure 13. This experiment uses the same 300 randomly chosen paths for the 20 security-critical signals in the OR1200. The largest percentage of eliminated paths are found statically before symbolic execution begins. This is good news, as that is the cheapest and quickest phase of the analysis. There is a non-trivial portion, 5% to 7%, that are eliminated because they do not represent true flows of information through the design. The pruning of these IF paths guides the user towards paths that are more likely to realize the path of interest and away from paths that are unrealizable. SEIF's use of symbolic execution allows for this precise analysis, which taint tracking may not be able to provide.

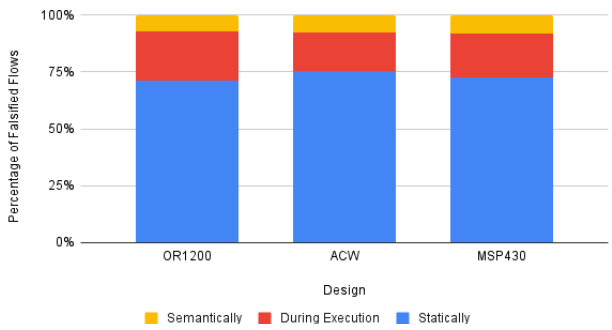

Figure 13: Breakdown of how flows are falsified by SEIF

6 RELATED WORK

Symbolic Execution of HW Designs for Information Flow Analysis EISec uses netlist-level symbolic execution to verify information flow safety and quantify confusion and diffusion in cryptographic modules [28]. Our work allows analysis at the RT-level and enables verification of a wider class of information-flow properties. Other tools use symbolic simulation (e.g., [51]) to verify particular binaries running on the hardware [7, 17].

Symbolic Execution of SW for Information Flow Analysis The software community was perhaps the first to leverage symbolic execution to verify information flow. The approach has been used in combination with taint tracking [16], to find and mitigate side channels [10, 15, 53, 54], and to identify programs that are vulnerable to transient execution attacks [29].

Symbolic Execution of SW or HW to Find Exploitable Flaws There is a long history of using symbolic execution in software to find exploitable security flaws (e.g, [8, 9, 44]). In hardware, symbolic execution has been used to find violations of and exploits for security-critical assertions [59] and to find and trigger trojans in the Verilog RTL [48]. As with SEIF, the main challenge is guiding search to find the salient paths. Static analysis in SW was leveraged to produce a "code property graph" to find insecure code [56]

Information Flow Tracking in HW The state of the art for information flow analysis in hardware is information flow tracking (IFT), which instruments a design with tracking logic [50]. Many tools operate at the netlist level, although some operate at the RTL level [5]. IFT has also been used in analog designs [11], and tools exist to synthesize designs that incorporate tracking logic [42, 43]. IFT can be used to check hyperproperties and has been used to verify the safety and security of many different systems [4, 14, 18, 33, 34, 36, 40, 45, 46, 49]. IFT has also been used to automatically generate information flow properties for use with formal verification engines [20, 21]. We used these properties in our evaluation; although our technique is not property-based we used the security-critical signals they identified for analysis. IFT has also been used to develop security metrics for the OpenTitan Hardware Root of Trust [41]. SEIF also shows that there is interesting information to be gleaned during the verification process in addition to property violations (see Tables 1, 2).

Formal Analysis for Information Flow Proof-checking approaches have been used for detecting security vulnerabilities in hardware designs [24, 35]. These approaches are often less automated, more time intensive, and tackle smaller designs, for stronger results that are both sound and complete. VeriCoq translated Verilog to Coq for proof-carrying designs [12]. Another approach is to use self-composition, or program products, to verify information-flow properties [23]. Security extensions in the hardware description language can enforce information flow policies at the language level [6, 19, 27, 37, 38, 58].

7 CONCLUSION

SEIF uses symbolic execution guided by a HF graph to find information flows in hardware designs. SEIF can be used to find counterexamples to information flow properties, and also to explore all flows originating from a source signal of interest. In our experiments, SEIF accounts for 86–90% of statically identified possible flows in three open-source designs. SEIF's search strategies enable exploring the designs for 10-12 clock cycles in 4-6 seconds on average, demonstrating that this new exploratory style of information flow analysis can be practical.

ACKNOWLEDGMENTS

We would like to thank Andres Meza and Ryan Kastner for their invaluable support and assistance with this project. Without them this work would not have been possible. This material is based upon work supported by the National Science Foundation under Grant No. CNS-2247754, and by a Meta Security Research Award.

REFERENCES

[1] [n. d.]. *Cadence Verification.* https://www.cadence.com/en_US/home/tools/system-design-and-verification.html
[2] [n. d.]. openMSP430. https://opencores.org/projects/openmsp430
[3] [n. d.]. OpenRISC 1200 Implementation. https://github.com/openrisc/or1200
[4] Armaiti Ardeshiricham, Wei Hu, and Ryan Kastner. 2017. Clepsydra: Modeling timing flows in hardware designs. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 147–154.
[5] Armaiti Ardeshiricham, Wei Hu, Joshua Marxen, and Ryan Kastner. 2017. Register transfer level information flow tracking for provably secure hardware design. In *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*. European Design and Automation Association, 1695–1700. http://dl.acm.org/citation.cfm?id=3130379.3130775
[6] Armaiti Ardeshiricham, Yoshiki Takashima, Sicun Gao, and Ryan Kastner. 2019. VeriSketch: Synthesizing Secure Hardware Designs with Timing-Sensitive Information Flow Properties. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 1623–1638.
[7] Anish Athalye, M. Frans Kaashoek, and Nickolai Zeldovich. 2022. Verifying Hardware Security Modules with Information-Preserving Refinement. In *OSDI*. USENIX Association.
[8] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J Schwartz, Maverick Woo, and David Brumley. 2014. Automatic exploit generation. *Commun. ACM* 57, 2 (2014), 74–84.
[9] Thanassis Avgerinos, Brent Lim Tze Hao, and David Brumley. 2011. Automatic exploit generation. In *Network and Distributed System Security Symposium (NDSS)*. Internet Society.
[10] Qinkun Bao, Zihao Wang, Xiaoting Li, James R Larus, and Dinghao Wu. 2021. Abacus: Precise side-channel analysis. In *International Conference on Software Engineering (ICSE)*. IEEE/ACM, 797–809.
[11] Mohammad-Mahdi Bidmeshki, Angelos Antonopoulos, and Yiorgos Makris. 2017. Information flow tracking in analog/mixed-signal designs through proof-carrying hardware IP. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 1703–1708.
[12] Mohammad-Mahdi Bidmeshki and Yiorgos Makris. 2015. VeriCoq: A Verilog-to-Coq converter for proof-carrying hardware automation. In *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 29–32.
[13] M. Bilzor, T. Huffmire, C. Irvine, and T. Levin. 2011. Security Checkers: Detecting processor malicious inclusions at runtime. In *HOST*.
[14] Travis H Boraten and Avinash K Kodi. 2018. Securing NoCs against timing attacks with non-interference based adaptive routing. In *2018 Twelfth IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*. IEEE, 1–8.
[15] Robert Brotzman, Shen Liu, Danfeng Zhang, Gang Tan, and Mahmut Kandemir. 2019. Casym: Cache aware symbolic execution for side channel detection and mitigation. In *Symposium on Security and Privacy (SP)*. IEEE.
[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, USA, 380–394.
[17] Hari Cherupalli, Henry Duwe, Weidong Ye, Rakesh Kumar, and John Sartori. 2017. Software-based gate-level information flow security for IoT systems. In *50th IEEE/ACM International Symposium on Microarchitecture*.
[18] Michael R. Clarkson and Fred B. Schneider. 2010. Hyperproperties. *J. Comput. Secur.* 18, 6 (Sept. 2010), 1157–1210. http://dl.acm.org/citation.cfm?id=1891823.1891830
[19] Shuwen Deng, Doğuhan Gümüşoğlu, Wenjie Xiong, Y Serhan Gener, Onur Demir, and Jakub Szefer. 2017. SecChisel: language and tool for practical and scalable security verification of security-aware hardware architectures. *Cryptology ePrint Archive* (2017).
[20] Calvin Deutschbein, Andres Meza, Francesco Restuccia, Ryan Kastner, and Cynthia Sturton. 2021. Isadora: Automated Information Flow Property Generation for Hardware Designs. In *Proceedings of the Workshop on Attacks and Solutions in Hardware Security (ASHES)*. ACM.

[21] Calvin Deutschbein, Andres Meza, Francesco Restuccia, Ryan Kastner, and Cynthia Sturton. 2022. Isadora: Automated Information Flow Property Generation for Hardware Security Verification. *Journal of Cryptographic Engineering (JCEN)* (2022).

[22] Vernetta Dorsey and Camille Morhardt. 2020. *Intel Security Development Lifecycle.* Technical Report. Intel.

[23] Marco Eilers, Severin Meier, and Peter Müller. 2021. Product Programs in the Wild: Retrofitting Program Verifiers to Check Information Flow Security. In *Computer Aided Verification (CAV)*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer International Publishing.

[24] M. Fadiheh, A. Wezel, J. Muller, J. Bormann, S. Ray, J. M. Fung, S. Mitra, D. Stoffel, and W. Kunz. 2023. An Exhaustive Approach to Detecting Transient Execution Side Channels in RTL Designs of Processors. *IEEE Trans. Comput.* 72, 01 (jan 2023), 222–235.

[25] Nusrat Farzana, Farimah Farahmandi, and Mark Mohammad Tehranipoor. 2021. SoC Security Properties and Rules. *IACR Cryptol. ePrint Arch.* 2021 (2021), 1014.

[26] Nusrat Farzana, Fahim Rahman, Mark Tehranipoor, and Farimah Farahmandi. 2019. SoC Security Verification using Property Checking. In *2019 IEEE International Test Conference (ITC)*. 1–10.

[27] 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* (Xi'an, China) *(ASPLOS '17)*. ACM, New York, NY, USA, 555–568.

[28] Farhaan Fowze, Muhtadi Choudhury, and Domenic Forte. 2022. EISec: Exhaustive Information Flow Security of Hardware Intellectual Property Utilizing Symbolic Execution. In *Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*. IEEE Xplore.

[29] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez. 2020. Spectector: Principled Detection of Speculative Information Flows. In *2020 IEEE Symposium on Security and Privacy (SP)*. 1–19.

[30] Sunny L. He, Natalie H. Roe, Evan C. L. Wood, Noel Nachtigal, and Jovana Helms. 2015. *Model of the Product Development Lifecycle.* Technical Report. Sandia National Laboratories.

[31] 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 *ASPLOS* (Istanbul, Turkey) *(ASPLOS '15)*. ACM, New York, NY, USA, 517–529.

[32] Wei Hu, Armaiti Ardeshiricham, Mustafa S Gobulukoglu, Xinmu Wang, and Ryan Kastner. 2018. Property Specific Information Flow Analysis for Hardware Security Verification. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–8.

[33] Wei Hu, Baolei Mao, Jason Oberg, and Ryan Kastner. 2016. Detecting hardware trojans with gate-level information-flow tracking. *Computer* 49, 8 (2016), 44–52.

[34] Yier Jin and Yiorgos Makris. 2012. Proof carrying-based information flow tracking for data secrecy protection and hardware trust. In *2012 IEEE 30th VLSI Test Symposium (VTS)*. IEEE, 252–257.

[35] Shuyu Kong, Yuanqi Shen, and Hai Zhou. 2017. Using security invariant to verify confidentiality in hardware design. In *Proceedings of the on Great Lakes Symposium on VLSI 2017*. 487–490.

[36] Elisavet Kozyri, Stephen Chong, and Andrew C. Myers. 2022. Expressing Information Flow Properties. *Foundations and Trends® in Privacy and Security* 3, 1 (2022), 1–102. http://dx.doi.org/10.1561/3300000008

[37] 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* (Salt Lake City, Utah, USA) *(ASPLOS '14)*. ACM, New York, NY, USA, 97–112.

[38] 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* (San Jose, California, USA) *(PLDI '11)*. ACM, New York, NY, USA, 109–120.

[39] Andres Meza and Ryan Kastner. 2023. Information Flow Coverage Metrics for Hardware Security Verification. arXiv 2304.08263.

[40] Andres Meza, Francesco Restuccia, Ryan Kastner, and Jason Oberg. 2022. Safety verification of third-party hardware modules via information flow tracking. In *1st Real-Time Intelligent Edge Computing Workshop (RAGE)*. https://kastner.ucsd.edu/wp-content/uploads/2022/08/admin/rage22-safety.pdf

[41] Andres Meza, Francesco Restuccia, Jason Oberg, Dominic Rizzo, and Ryan Kastner. 2023. Security Verification of the OpenTitan Hardware Root of Trust. *IEEE Security & Privacy* 21, 3 (2023).

[42] Pascal Pieper, Vladimir Herdt, Daniel Große, and Rolf Drechsler. 2020. Dynamic information flow tracking for embedded binaries using SystemC-based virtual prototypes. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[43] Christian Pilato, Kaijie Wu, Siddharth Garg, Ramesh Karri, and Francesco Regazzoni. 2018. Tainthls: High-level synthesis for dynamic information flow tracking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 5 (2018), 798–808.

[44] Matthew J. Renzelmann, Asim Kadav, and Michael M. Swift. 2012. SymDrive: Testing Drivers without Devices. In *10th USENIX Symposium on Operating Systems Design and Implementation*. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/renzelmann

[45] Francesco Restuccia, Andres Meza, and Ryan Kastner. 2021. AKER: A design and verification framework for safe and secure soc access control. In *IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. https://par.nsf.gov/servlets/purl/10298115

[46] Francesco Restuccia, Andres Meza, Ryan Kastner, and Jason Oberg. 2022. A Framework for Design, Verification, and Management of SoC Access Control Systems. *IEEE Trans. Comput.* (2022). https://kastner.ucsd.edu/wp-content/uploads/2022/10/admin/tcomputer-aker22.pdf

[47] Kaki Ryan and Cynthia Sturton. 2023. Countering the Path Explosion Problem in the Symbolic Execution of Hardware Designs. In *Proceedings of the 23rd Conference on Formal Methods in Computer-Aided Design (FMCAD)*.

[48] Lixiang Shen, Dejun Mu, Guo Cao, Maoyuan Qin, Jeremy Blackstone, and Ryan Kastner. 2018. Symbolic execution based test-patterns generation algorithm for hardware Trojan detection. *Comput. Secur.* 78 (2018), 267–280.

[49] Mohit Tiwari, Xun Li, Hassan MG Wassel, Frederic T Chong, and Timothy Sherwood. 2009. Execution leases: A hardware-supported mechanism for enforcing strong non-interference. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 493–504.

[50] Mohit Tiwari, Hassan MG Wassel, Bita Mazloom, Shashidhar Mysore, Frederic T Chong, and Timothy Sherwood. 2009. Complete information flow tracking from the gates up. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*. 109–120.

[51] Emina Torlak and Rastislav Bodik. 2014. A Lightweight Symbolic Virtual Machine for Solver-Aided Host Languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 530–541.

[52] Timothy Trippel, Kang G. Shin, Kevin B. Bush, and Matthew Hicks. 2020. ICAS: an Extensible Framework for Estimating the Susceptibility of IC Layouts to Additive Trojans. In *2020 IEEE Symposium on Security and Privacy (SP)*. 1742–1759.

[53] Shuai Wang, Yuyan Bao, Xiao Liu, Pei Wang, Danfeng Zhang, and Dinghao Wu. 2019. Identifying Cache-Based Side Channels through Secret-Augmented Abstract Interpretation. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 657–674. https://www.usenix.org/conference/usenixsecurity19/presentation/wang-shuai

[54] Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. 2017. CacheD: Identifying cache-based timing channels in production software. In *USENIX Security Symposium*. 235–252.

[55] Jiaming Wu, Farhaan Fowze, and Domenic Forte. 2022. EXERT: EXhaustive Integrity Analysis for Information Flow Security. In *Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*. IEEE. https://dforte.ece.ufl.edu/wp-content/uploads/sites/65/2022/09/EXERT_AsianHost.pdf

[56] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *2014 IEEE Symposium on Security and Privacy*. 590–604.

[57] Drew Zagieboylo, G Edward Suh, and Andrew C Myers. 2019. Using information flow to design an ISA that controls timing channels. In *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*. IEEE, 272–27215.

[58] 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* (Istanbul, Turkey) *(ASPLOS '15)*. ACM, New York, NY, USA, 503–516.

[59] Rui Zhang, Calvin Deutschbein, Peng Huang, and Cynthia Sturton. 2018. End-to-End Automated Exploit Generation for Validating the Security of Processor Designs. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*. IEEE/ACM.

[60] Rui Zhang, Natalie Stanley, Christopher Griggs, Andrew Chi, and Cynthia Sturton. 2017. Identifying Security Critical Properties for the Dynamic Verification of a Processor. In *ASPLOS* (Xi'an, China) *(ASPLOS '17)*. ACM, New York, NY, USA, 541–554.

[61] Rui Zhang and Cynthia Sturton. 2020. Transys: Leveraging Common Security Properties Across Hardware Designs. In *Proceedings of the Symposium on Security and Privacy (S&P)*. IEEE.

[62] Tao Zhang, Jungmin Park, Mark Tehranipoor, and Farimah Farahmandi. 2021. PSC-TG: RTL Power Side-Channel Leakage Assessment with Test Pattern Generation. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. 709–714.