Isadora: Automated Information Flow Property Generation for Hardware Designs

Calvin Deutschbein
University of North Carolina at Chapel Hill

Andres Meza
UC San Diego

Francesco Restuccia
Scuola Superiore Santa-Anna Pisa

Ryan Kastner
UC San Diego

Cynthia Sturton
University of North Carolina at Chapel Hill

ABSTRACT

Isadora is a methodology for creating information flow specifications of hardware designs. The methodology combines information flow tracking and specification mining to produce a set of information flow properties that are suitable for use during the security validation process, and which support a better understanding of the security posture of the design. Isadora is fully automated; the user provides only the design under consideration and a testbench and need not supply a threat model nor security specifications. We evaluate Isadora on a RISC-V processor plus two designs related to SoC access control. Isadora generates security properties that align with those suggested by the Common Weakness Enumerations (CWEs), and in the case of the SoC designs, align with the properties written manually by security experts.

ACM Reference Format:
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 5th Workshop on Attacks and Solutions in Hardware Security (ASHES '21), November 19, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3474376.3487286

1 INTRODUCTION

Security validation is an important yet challenging part of the hardware design process. A strong validation provides assurance that the design is secure and trustworthy: it will not be vulnerable to attack once deployed, and it will reliably provide software and firmware with the advertised security features. A security validation engineer is tasked with defining the threat model, specifying the relevant security properties, detecting any violations of those properties, and assessing the consequences to system security.

Existing commercial design tools (e.g., Mentor Questa Secure Check, Cadence JasperGold Security Path Verification, and Tortuga Logic Radix) can verify security properties of a design, but the tools are only as strong as the provided properties. *Defining these hardware security properties is a crucial part of the security validation process that currently involves a significant manual undertaking.* We propose an automated methodology that combines information flow tracking with specification mining to create a human-readable information flow specification of a hardware design. The specification can be used as a set of security properties suitable for use with existing security validation tools, and it can also be studied directly by the designers to support their understanding of how information flows through the design.

Information Flow Tracking (IFT) is a powerful security verification technique that monitors how information moves through a hardware design. Recently, IFT has been demonstrated at the RTL [6, 29] and gate level [7, 30, 41], and has been used to monitor implicit flows through digital side channels [5, 8, 36]. Existing verification engines that incorporate IFT capabilities can be used to confirm whether a given information flow property holds. However, it is up to the designer to specify the full set of desired flow behaviors.

The technique of specification mining offers an automatic alternative to manually writing properties. Specification mining can be applied to software [4] and hardware [24] and has recently been applied to system on a chip (SoC) designs [21, 37]. Security specification mining focuses on developing the security goals of a design and has been developed for processors [17, 18, 45]. However, many important vulnerabilities violate security goals related to how information flows, goals that are not expressible as the trace properties that specification miners discover.

The insight that led to this research is that mining the traces produced by an IFT-instrumented design will generate trace properties that correspond to information flow properties over the original, uninstrumented design. The information flow tracking logic transforms the information-flow properties from the space of hyperproperties [12] – where trace-based mining does not apply – to the space of trace properties, where trace-based mining can apply.

A naive application of trace-based mining to an IFT-instrumented design quickly runs into issues of complexity: the instrumented designs are large and overwhelm the miner. Additionally, the miner will discover properties over tracking signals and original design signals that are meaningless and cannot be transformed back to the space of information flow properties in the original design. To handle these issues we separate the process of identifying source–sink flow pairs in the design from the process of mining for the conditions that govern those flows. The first can be done by leveraging existing information flow tracking tools and the second makes use of existing trace miners. The key to making the approach work is to synchronize the two parts using clock-cycle time.

The methodology we present here can inform an automated analysis of a hardware design by identifying flow relations between all design elements, including flow conditions and multi-source and multi-sink cases. The methodology requires no input from the designer beyond the design and testbench.

To evaluate our methodology, we developed Isadora, a fully automatic security specification miner for information flow properties. Isadora uses information flow tracking (IFT) technology from Tortuga Logic's Radix-S simulation based security verification engine [3] and is implemented on top of Daikon [20], a popular invariant miner.

To our knowledge, Isadora represents the first specification miner capable of extracting information flow security properties from hardware designs. Our results demonstrate:

- Isadora characterizes the flow relations between all elements of a design.
- Isadora identifies important information flow security properties of a design without guidance from the designer.
- Isadora can be used to find undesirable flows of information in the design.
- Isadora is applicable to SoCs and CPUs.

To measure our methodology and the usefulness of Isadora's mined specification, we evaluated Isadora over an access control module, a multi-controller and multi-peripheral system with a known security policy, and a RISC-V design. We evaluated the output of Isadora versus expected information flow policies of the design and found information flow specifications that, if followed, protect designs from known and potential future attack patterns.

2 PROPERTIES

Isadora generates two styles of information flow properties: no-flow properties, in which there is no flow of information between two design elements; and conditional-flow properties, in which there exists some flow of information between two design elements, but only when the design is in a certain state. Isadora can also generate unconditional-flow properties, but these tend to be less interesting for purposes of security validation.

2.1 Tracking Information Flow

IFT can precisely measure all digital information flows in the underlying hardware, including, for example, implicit flows through hardware-specific timing channels. Isadora uses IFT at the register transfer level [6] to track data flow between registers rather than considering individual bits, with 'registers' in this context referring to the Verilog notion of a register. Isadora may additionally be configured to consider Verilog wires, though doing so provided no observable improvements to generated specifications and considerably increased trace generation costs. The Isadora methodology can be applied to individual bits, as the underlying information flow tracking used within Isadora does consider individual bits. However, bit level analysis would result in extraordinarily high trace generation costs for even small designs.

Tracking proceeds as follows: for each signal $\mathtt{s}$ in the design, a new tracking signal $\mathtt{s}^T$ is added along with the logic needed to track how information propagates through a design. Once the tracking signals and tracking logic are added to the design, one or more signals may be set as the *information source* by initializing their associated tracking signals to a nonzero value. All other tracking signals are initialized to zero. As the design executes, and information from a source signal propagates to a second signal, that second signal's tracking signal is updated from zero to nonzero.

2.2 Information Flow Restrictions

Using information flow tracking, we can express the property that information from register $\mathtt{r}_1$ should never flow to register $\mathtt{r}_2$ as a trace property: if $\mathtt{r}_1$ is the only signal whose tracking signal $\mathtt{r}_1^T$ is initialized to nonzero, then for all possible executions of the design, $\mathtt{r}_2$'s tracking signal $\mathtt{r}_2^T$ should remain at zero:

$$(\forall \mathtt{r}_i,\ \mathtt{r}_i^T \neq 0 \leftrightarrow i = 1) \rightarrow \mathbb{G}(\mathtt{r}_2^T = 0)$$

This style of no-flow property can be useful for ensuring unprivileged users cannot influence sensitive state or for ensuring that sensitive information cannot leak through, for example, debug ports. However, it cannot capture conditional properties, for example that register updates are allowed only under certain power states.

2.3 Information Flow Conditions

Using information flow tracking, we can express the property that information from a register $\mathtt{r}_1$ may flow to another register $\mathtt{r}_2$ under some condition P: if $\mathtt{r}_1$ is the only signal whose tracking signal $\mathtt{r}_1^T$ is initialized to nonzero then for all possible executions of the design, $\mathtt{r}_2$'s tracking signal $\mathtt{r}_2^T$ will only become non-zero if some predicate P holds:

$$(\forall \mathtt{r}_i,\ \mathtt{r}_i^T \neq 0 \leftrightarrow i = 1) \rightarrow \mathbb{G}(\neg P \rightarrow (\mathtt{r}_2^T = 0 \rightarrow \mathbb{X}(\mathtt{r}_2^T = 0)))$$

This style of a conditional flow property can be used to express, for example, that register updates are allowable only under certain power states, or that memory accesses are allowable only when specific access control checks have succeeded.

2.4 Grammar of Properties

In order to produce properties that use only the signals in the original design, without including the tracking signals, we need an operator that expresses some notion of information flow. Both no-flow and conditional flow properties can be expressed using a no-flow operator, for which we use the notation `=/=>`. The grammar of Isadora properties is as follows.

$$\begin{aligned}
\phi &\doteq \mathtt{r}_1 \texttt{=/=>}\ \mathtt{r}_2 \mid e \rightarrow \mathtt{r}_1\ \texttt{=/=>}\ \mathtt{r}_2 \\
e &\doteq b \wedge e \mid b \\
b &\doteq \mathtt{r} \in \{x, y, z\} \mid \mathtt{r}_1 = \mathtt{r}_2 \mid \mathtt{r}_1 \neq \mathtt{r}_2 \mid \mathtt{r} = \mathrm{prev}(\mathtt{r})
\end{aligned}$$

The property $\mathtt{r}_1$ `=/=>` $\mathtt{r}_2$ states that no information flows from $\mathtt{r}_1$ to $\mathtt{r}_2$. The property $e \rightarrow \mathtt{r}_1$ `=/=>` $\mathtt{r}_2$ states that information may flow from $\mathtt{r}_1$ to $\mathtt{r}_2$ only when $\neg e$. The symbol $\mathtt{r}$ is a register in the design, $\mathtt{r} \in \{x, y, z\}$ means that $\mathtt{r}$ may take on any one of the values in a set of cardinality less than or equal to three, and $\mathrm{prev}(\mathtt{r})$ refers to the value of $\mathtt{r}$ in the previous clock cycle.

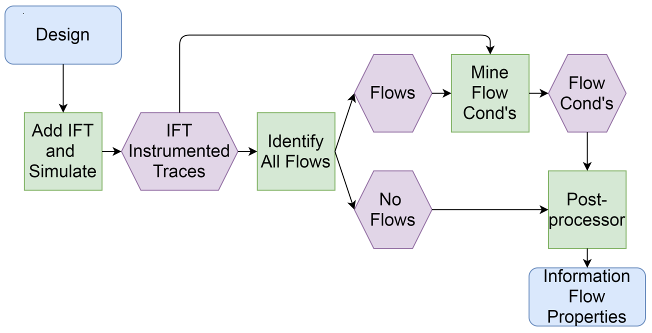

Figure 1: An overview of the Isadora workflow

3 METHODOLOGY

Isadora analyzes a design in four phases: generating traces, identifying flows, mining for flow conditions, and postprocessing. An overview of the workflow is presented in Figure 1.

First, Isadora instruments the design with IFT logic and runs the instrumented design in simulation using the user-provided set of testbenches. The result is a trace set that specifies the value of every design signal and every tracking signal at each clock cycle during simulation.

Next, Isadora studies the trace set to find every flow that occurred during the simulation of the design. This set of flows is complete: if a flow occurred between any two signals, it will be included in this set. At the end of this phase, Isadora also produces the complete set of information flow restrictions: pairs of signals between which no information flow occurs.

Then, Isadora uses an inference engine (Daikon [20]) to infer, for every flow that occurred, the predicates that specify the conditions under which the flow occurred.

The final phase removes redundant and irrelevant predicates from the set and logically combines the predicates with the information flows to produce the conditional flow properties. These, along with the no-flow properties from the second phase, form the information flow specification produced by Isadora.

3.1 Generating Traces with Information Flow Tracking

To generate a trace set, the design is instrumented with IFT logic and then executed in simulation with a testbench or sequence of testbenches providing input values to the design. Let $\tau_{\texttt{src}} = \langle \sigma_0, \sigma_1, \ldots, \sigma_n \rangle$ be the trace of a design instrumented to track how information flows from one signal, `src`, during execution of a testbench. The state σ_i of the design at time i is defined by a list of triples describing the current value of every design signal and corresponding tracking signal in the instrumented design:

$$\sigma_i = [(\mathsf{s}_1, v_1, v_1^t), (\mathsf{s}_2, v_2, v_2^t), \ldots, (\mathsf{s}_m, v_m, v_m^t)]_i.$$

In order to distinguish the source of a tainted sink signal, each input signal must have a separate taint label. However, tracking multiple labels is expensive [30]. Therefore, Isadora takes a compositional approach. For each source signal, IFT instrumentation is configured to track the flow of information from only a single input signal of the design, the `src` signal. This process is applied to each signal in a design. The end result is a set of traces for design D and testbench T: $\mathcal{T}_{\text{DT}} = \{\tau_{\texttt{src}}, \tau_{\texttt{src}'}, \tau_{\texttt{src}''}, \ldots\}$. Each trace in this set describes how information can flow from a single input signal to the rest of the signals in the design. Taken together, this set of traces describes how information flows through the design during execution of the testbench T.

3.2 Identifying All Flows

In the second phase, the set of traces are analyzed to identify:

(1) every pair of signals between which a flow occurs, and
(2) the times within the trace at which each flow occurs.

Each trace $\tau_{\texttt{src}}$ is searched to find every state in which a tracking signal goes from being set to 0 to being set to 1. In other words, every signal–values triple (s, v, v^t) that is of the form $(\mathsf{s}, v, 0)$ in state σ_{i-1} and $(\mathsf{s}, v, 1)$ in state σ_i is found and the time i is noted. This is stored as the tuple $(\texttt{src}, \mathsf{s}, \{i_0, i_1, \ldots\})$, which indicates that information from `src` reached signal s at all times $i \in \{i_0, i_1, \ldots\}$. We call this the *time-of-flow tuple.* There can be multiple times-of-flow within a single trace because the tracking value of signals may be reset to zero by design events such as resets.

Once all traces have been analyzed, the collected time-of-flow tuples $(\texttt{src}, \mathsf{s}, \{i_0, i_1, \ldots\})$ are organized by time. For any given set of times $\{i_0, i_1, \ldots\}$ there may be multiple discovered flows. For all traces $\tau_{\texttt{src}}$ generated by a single testbench, the timing of flows from one source `src` can be compared to the timing of flows from a second source `src′`; the value i will refer to the same point in the testbench. At the end of this phase, the tool produces two outputs. The first is a list of the unique sets of times present within time-of-flow tuples and all the corresponding register pairs for which flow is discovered at precisely the times in the set:

$$\begin{aligned} S_{\text{flows}} = [&\langle \{i_0, i_1, \ldots\} : \{(\texttt{src}_1, \mathsf{s}_1), (\texttt{src}_2, \mathsf{s}_2), \ldots\} \rangle; \\ &\langle \{i_0', i_1', \ldots\} : \{(\texttt{src}_1', \mathsf{s}_1'), (\texttt{src}_2', \mathsf{s}_2'), \ldots\} \rangle; \ldots]. \end{aligned}$$

The same `src` may flow to many sinks $\mathsf{s} \in \{\mathsf{s}_1, \mathsf{s}_2, \ldots\}$ at the same times $i \in \{i_0, i_1, \ldots\}$, and the same sink s may receive information from multiple sources $\texttt{src} \in \{\texttt{src}_1, \texttt{src}_2, \ldots\}$ at the same times $i \in \{i_0, i_1, \ldots\}$.

The second output from this phase is a list of source-sink pairs between which information never flows:

$$S_{\text{no-flow}} = \{(\texttt{src}, \mathsf{s}), (\texttt{src}', \mathsf{s}'), \ldots\}.$$

The pairs in this set comprise the noninterference properties of the design, and can be specified using the no-flow operator, for example `src =/=> s`

3.3 Mining for Flow Conditions

In the third phase, Isadora finds the conditions under which a particular flow will occur. For example, if every time `src` flows to s, the register r has the value x, Isadora infers the conditional information flow property:

$$\neg(\mathsf{r} = x) \rightarrow \texttt{src =/=> s}$$

Isadora uses the technique of dynamic invariant detection [20] on traces to infer design behavior using pre-defined patterns. In order to isolate the conditions for information flow between two registers, Isadora uses S_{flows} to find all the trace times i at which information flows from `src` to `s` during execution of the testbench. The corresponding trace(es) are then decomposed to produce a set of trace slices that are two clock cycles in length, one for each time i. Consider time-of-flow tuple $(\texttt{src}, \texttt{s}, [i, j, k, \ldots])$, which as a notational convenience here uses distinct letters to denote time points rather than subscripts for clarity in the following expression. Given this tuple, Isadora will produce the trace slices $\langle \sigma_{i-1}, \sigma_i \rangle, \langle \sigma_{j-1}, \sigma_j \rangle, \langle \sigma_{k-1}, \sigma_k \rangle$. These trace slices include only the signals of the original design, all tracking logic and shadow signals are pruned. Using trace slices, or trace windows of length two, allows dynamic invariant detection to generation predicates specifying design state both immediately prior to and concurrent with the occurence of some flow. Predicates match one of the four patterns for expressions given in the grammar of Isadora properties in Section 2.

3.4 Postprocessing

Finally, Isadora performs additional analysis to find invariants that may hold over the entire trace set by running the miner on the unsliced trace. Isadora eliminates any predicate that is also found to be a trace-set invariant. One such trivial example is the invariant $\texttt{clk} = \{0, 1\}$.

The final output properties from postprocessing are the conditional flow properties. To ease readability, Isadora can express the conditional flow properties as multi-source to multi-sink flows, where all flows within the same property occur at the same time and under the same conditions. This produces comparatively few properties, which in practice were approximately as many as the number of unique source signals, and avoids redundant information. The conditional flow properties and the no-flow properties discovered in phase 2 (Section 3.2) make up the set of information flow properties produced by Isadora. Two examples of postprocessed properties are shown in Appendix A.

4 IMPLEMENTATION

Isadora uses the Tortuga Radix-S simulation-based security verification technology [3] to generate IFT logic for a hardware design, the Questa Advanced Simulator [2] to simulate the instrumented design and generate traces, and the Daikon [20] invariant miner to find flow conditions. A Python script manages the complete workflow and implements flow analysis and postprocessing.

Traces are generated for all signals within a design. An automated utility identifies every signal within a design and configures Tortuga Radix-S to build the IFT logic separately for each of these registers. We run Tortuga in exploration mode, which omits cone of influence analysis, and track flows to all design state using the `$all_outputs` variable. The resulting instrumented designs are simulated in QuestaSim over a testbench (see Evaluation, Sec. 5) to produce a trace of execution.

Phase two is implemented as a Python tool that reads in the traces generated by QuestaSim and produces the set of no-flow properties and the time-of-flow tuples. This phase combines the bit-level taint tracking by Radix-S into signal-level tracking. Each n-bit signal in the original design is then tracked by a 1-bit shadow signal, which will be set to 1 at the first point in the trace that any of the component n shadow bits where set.

The mining phase is built on top of the Daikon invariant generation tool [20], which was developed for use with software programs. Daikon generates invariants over state variables for each point in a program. We built a Daikon front-end in Python (411 LoC, including comments) that converts the trace data to be Daikon readable, treating the state of the design at each clock cycle as a point in a program. The front-end also removes any unused or redundant signals and outputs relevant two-clock-cycle slices as described in Sec. 3.3.

5 EVALUATION

We assess the following questions to evaluate Isadora:

(1) Can Isadora independently mine security properties manually developed by hardware designers?
(2) Can Isadora automatically generate properties describing Common Weakness Enumerations (CWEs) [1] over a design?
(3) Does Isadora scale well for larger designs, such as CPUs or SoCs?

5.1 Designs

We assessed Isadora on two designs, the Access Control Wrapper (ACW) proposed within the AKER framework [39] and the PicoRV32 RISC-V CPU. An ACW wraps an AXI controller and enforces on it a *local access control policy*, which is setup and maintained by a trusted entity (e.g., a Hardware Root of Trust or a trusted processor). The ACW checks the validity of read and write requests issued by the wrapped AXI controller and rejects those that violate the configuration of the local access control policy.

We used the AKER framework in two configurations: first implementing a single-controller AKER-based access control system; second, implementing a system with two traffic generators, each wrapped by an ACW, connected to three AXI peripherals though an AXI interconnect. This setup simulates the use of the ACWs in an SoC environment. In both cases, the input signals of the ACWs are dictated by the testbench, which initializes them with the access control policies and acts as the trusted entity. We refer to these two designs as the *"Single ACW"* and *"Multi ACW"* cases. They are shown in Figures 2 and 3, respectively.

PicoRV32 is a CPU core that implements the RISC-V RV32IMC Instruction Set, an open standard instruction set architecture based on established reduced instruction set computer principles.

The secure operation of the ACW and AKER-based access control systems has been verified through a property-based security validation process by the designers. We study the AKER framework to evaluate how Isadora's properties compare to a manually developed security specification. We use the PicoRV32 to evaluate how well Isadora automatically generates properties describing CWEs and to evaluate how well Isadora scales on a CPU design.

5.2 Time Cost

Isadora ran on a system with an Intel Core i5-6600k (3.5GHz) processor with 8 GB of RAM. Traces were generated on a Intel Xeon CPU E5-2640 v3 @ 2.60GHz server. Trace generation dominated time costs, and scaled slightly worse than linear with number of unique signals in a design. Trace generation was suitable for parallelization though parallelization was not considered in the evaluation.

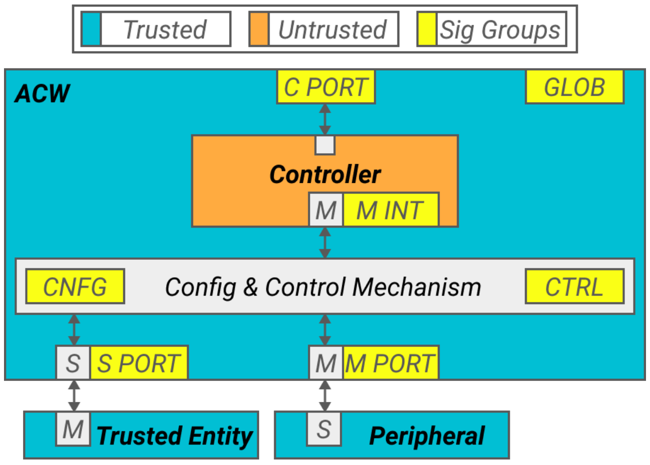

Figure 2: Block diagram of the Single ACW design, with labeled signal groups.

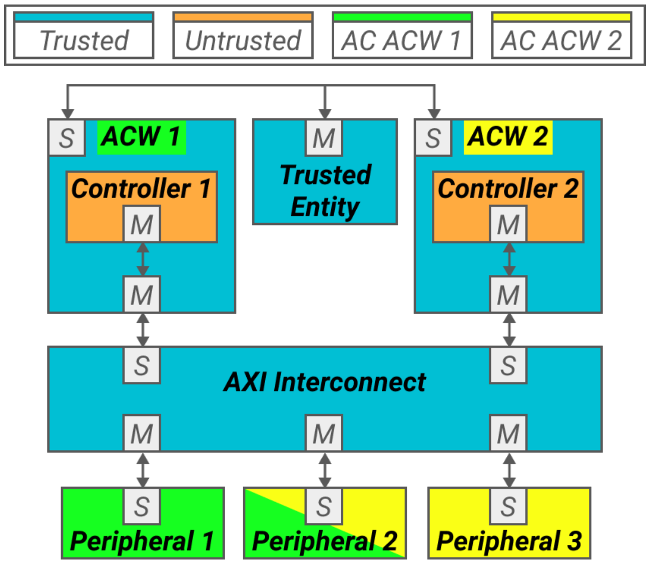

Figure 3: Block diagram of the Multi ACW design

The design sizes are given in Table 1. For the Single ACW, trace generation took 9h33m. For the Multi ACW, trace generation exceeded 24 hours so we consider a reduced trace, which tracks sources for one of the ACWs, though all signals are included as sinks or in conditions. The reduced trace was generated in 6h48m. For PicoRV32, trace generation took 8h35m.

5.2.1 Theoretical Gains to Parallelization. When parallelizing all trace generation and all case mining, Isadora could theoretically evaluate the Single ACW case fully in less than five minutes. Parallelizing the first phase requires a Radix-S and QuestaSim instance for each source register, and each trace is generated in approximately 100 seconds. Further, the trace generation time is dominated by write-to-disk, and performance engineering techniques could likely reduce it significantly, such as by changing trace encoding or piping directly to later phases. Parallelizing the second phase requires only a Python instance for each source register, and takes between 1 and 2 seconds per trace. Parallelizing the third phase requires a Daikon instance for each flow case, usually roughly the same number as unique sources, and takes between 10 and 30 seconds per flow case. The final phase, postprocessing, is also suitable for parallelization. Maximally parallelized, this gives a design-to-specification time of under four minutes for the single ACW and for similarly sized designs, including PicoRV32.

5.3 Designer Specified Security Properties

For the Single ACW we compared Isadora's output against security assertions developed by the AKER [39] designers using the Common Weakness Enumerations (CWE) database [1] as a guide. These assertions, the CWEs described, and the results of Isadora on the Single ACW are shown in Table 2. For each assertion Isadora mined either a property containing the assertion or found both a violation and the violating conditions for each assertion. We reported the observed violations to the designers who determined that the design remained secure but a conditional flow had been incorrectly specified as always illegal. Isadora also found the conditions for legality.

Only 9 Isadora properties, out of 303 total Isadora properties generated, were required to cover the designer-provided assertions, including conditions specifying violations. The Isadora output properties may contain many source or sink signals that flow concurrently and their corresponding conditions, whereas the designers' assertions each considered a single source and sink. For example, on the ACW nine distinct read channel registers always flow to a corresponding read channel output wire at the same time, so Isadora outputs a single property for this design state. This state included the reset signal and a configuration signal both set to non-zero values, which were captured as flow conditions, demonstrating correct design implementation. This single Isadora property captured 18 low level assertions related to multiple CWEs.

5.3.1 Case Study: Unintended Proxy. In the Multi ACW case, we studied CWE 411: Unintended Proxy or Intermediary ('Confused Deputy'). The system contained two controllers (C), with two access control modules (ACW), a trusted entity that configured each ACW (T), and three peripherals (P). The ACWs each implemented an access control (AC) policy shown in Figure 3 and given as:

$$\mathrm{AC}_1 \text{ of } \mathrm{ACW}_1: \quad R = \{P_1, P_2\},\ W = \{P_1\}$$
$$\mathrm{AC}_2 \text{ of } \mathrm{ACW}_2: \quad R = \{P_3\},\ W = \{P_2, P_3\}$$

Isadora discovered legal flows from the ACW_2 write data to P_3 read and write data, and P_2 read data. Isadora also finds an illegal flow to P_1 write data. The ACW_2-to-P_1 illegal flow has a flow condition specifying a prior flow from the relevant signals within ACW_2 to ACW_1. While not constituting a precise path constraint, this captures an access control violation and suggests the confused deputy scenario because the flow profile from ACW_2 is consistent with this path.

5.4 Automatic Property Generation

For the two designs with full trace sets, the Single ACW and PicoRV32, Isadora generates a specification describing all information

Design	Unique Signals	Unique Sources	LoC	Trace Cycles	Trace GBs	Daikon Traces	Isadora Properties	Miner Time In Minutes
Single ACW	229	229	1940	598	.7	252	303	29:51
Multi ACW	984	85	4447	848	4.3	378	160	8:31
PicoRV32	181	181	3140	1099	.6	955	153	15:09

Table 1: Various size measures of studied designs

Source	Sink	Invariant	Provided Assert's	Result	Isadora Properties	CWEs
M PORT	M INT	GLOB	19	✓	2, 40, 43, 53, 54, 204, 214	1258, 1266, 1270, 1271, 1272, 1280
M INT	M PORT		19	✓		
M PORT	M INT	C PORT	19	✓		1258, 1270, 1272, 1280
M INT	M PORT		19	✓		
S PORT	CNFG	-	4	✗	2, 6	1269, 1272, 1280

Table 2: Isadora performance versus manual specification, on the Single ACW

flows and their conditions with hundreds of properties. To assess whether these properties are security properties, for each design we randomly selected 10 of the 303 or 153 total properties (using Python random.randint) and assessed their relevance to security.

We use CWEs as a metric to evaluate the security relevance of Isadora output properties. To do so, for each design, we first determine which CWEs apply to the design. For both the ACW and PicoRV32, we used the Radix Coverage for Hardware Common Weakness Enumeration (CWE) Guide [3] to provide a list of CWEs that specifically apply to hardware. We considered each documented CWE for both designs. CWEs, while design agnostic, may refer to design features not present in the Single ACW or PicoRV32 or may not refer to information flows. High level descriptions in multiple CWEs may correspond to the same low level behavior for a design and we consider these CWEs together.

Information flow hardware CWEs describe source signals, sink signals, and possibly conditions. CWEs provide high level descriptions, but Isadora targets an RTL definition. To apply these high level descriptions to RTL, we first group signals for a design by inspecting Verilog files and, if available, designer notes. With the groups established, we label every property by which group-to-group flows they contain. We also determine which source–sink flows could be described in CWEs, which often correspond to, or even match exactly, a signal group. We use these groups to find CWE-relevant, low-level signals as sources, sinks, and conditions in an Isadora property. We also use these groups to characterize the relative frequency of conditional flows between different groups, which we present as heatmaps in the following subsections.

5.4.1 *ACW Conditional Information Flow.*

Over the ACW we assess fourteen CWEs which we map to five plain-language descriptions of the design features, as shown in Table 3.

CWE(s)	Description
1220	Read/write channel separation
1221-1259-1271	Correct initialization, reset, defaults
1258-1266-1270-1272	Access controls use operating modes
1274-1283	Anomaly registers log transactions
1280	Control checks precede access
1267-1269-1282	Configuration/user port separation

Table 3: The 14 CWEs considered for ACW

For the ACW signal groups, all registers were helpfully placed into groups by the designer and labeled within the design. The design contained seven distinct labeled groups:

- 'GLOB' - Global ports
- 'S PORT' - AXI secondary (S) interface ports of the ACW
- 'C PORT' - Connections to non-AXI ports of the controller
- 'M PORT' - AXI main (M) interface ports of the ACW
- 'CNFG' - Configuration signals
- 'M INT' - AXI M interface ports of the controller
- 'CTRL' - Control logic signals

GLOB signals are clock, reset, and interrupt lines. S PORT represents the signals that the trusted entity T uses to configure the ACW. C PORT represents the signals which are used to configure the controller C to generate traffic for testing. M PORT carries traffic between the peripheral P and the ACW's control mechanism. CNFG represents the design elements which manage and store the configuration of the ACW. M INT carries the traffic between the ACW's control mechanism and the controller. If it is legal according to the ACW's configuration, the control mechanism will send M INT traffic to M PORT and vice versa. CTRL represents the design elements of the aforementioned control mechanism.

First consider the heatmap view of the Single ACW in Figure 4. In this view, all of the designer-provided assertions fall into just 3 of the 49 categories; these are outlined in red. Further, all of the violations were found with S PORT to CNFG flows, while all satisfied assertions were flows between M INT and M PORT. Another interesting result visible in the heatmap is the infrequent flows into S PORT, which is used by the trusted entity to program the ACW. Most of the design features should not be able to reprogram

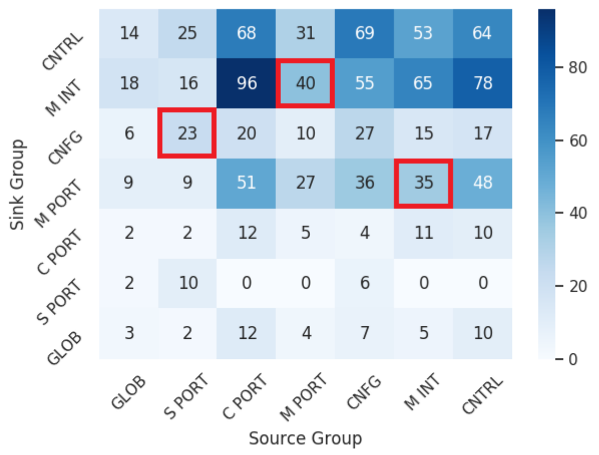

Figure 4: Group-to-group conditional flow heatmap for the Single ACW.

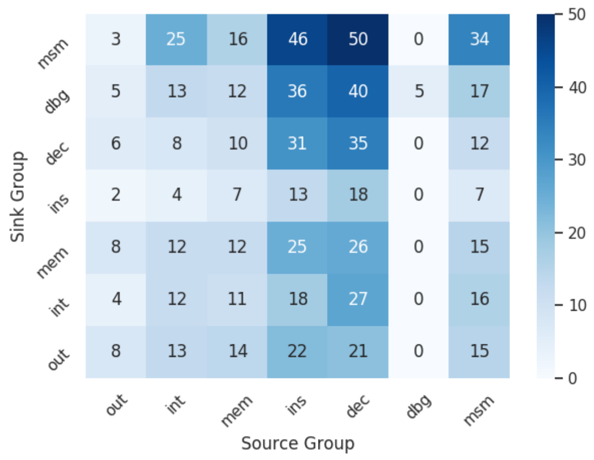

Figure 5: Group-to-group conditional flow heatmap for PicoRV32.

the access control policy, so finding no flows along these cases provides a visual representation of secure design implementation with respect to these features.

For the ACW, all ten sampled properties encode CWE-defined behavior to prevent common weaknesses, as shown in Table 4. In this table, the columns labeled by a CWE number and a '+' refer to all the CWEs given in a row of Table 3. Eight out of the ten properties provide separation between read and write channels, which constitutes the main functionality of the ACW module. CWEs 1267, 1269, and 1282 are not found within the conditional flow properties produced by Isadora as these are no-flow properties, so they are not present within the samples drawn from the numbered, conditional flow properties, but we were able to verify they are included in Isadora's set of no-flow properties.

5.4.2 PicoRV32 Conditional Information Flow. Over PicoRV32 we assess eighteen CWEs which we map to seven plain language descriptions of the design features, as shown in Table 5.

PicoRV32 had no designer-specified signal groups so we used comments in the code, register names, and code inspection to group all signals. We use lower case names to denote these groups were not defined by the designer.

- 'out' - Output registers
- 'int' - Internal registers
- 'mem' - Memory interface
- 'ins' - Instruction registers
- 'dec' - Decoder
- 'dbg' - Debug signals and state
- 'msm' - Main state machine

The memory interface and the main state machine were indicated by comments in the code. The instruction registers, the decoder, and debug all appeared under one disproportionately large section described as the instruction decoder. Debug was grouped by name after manual analysis found registers in this region prefixed with 'dbg_', 'q_', or 'cached_' to interact with and only with one another. Instruction registers prefixed 'instr_' all operate similarly to each other and differently than the remaining decoder signals, which were placed in the main decoder group. Internal signals were the remaining unlabeled signals that appeared early within the design, such as program and cycle counters and interrupt signals, and the output registers were all signals declared as output registers.

First consider the heatmap view of PicoRV32 in Figure 5. An interesting result visible in the heatmap is the flow isolation from debug signals to the rest of the design. Many exploits, both known and anticipated, target debug information leakage, and the heatmap shows this entire class of weakness is absent from the design.

For PicoRV32 we find eight of ten sampled properties encode CWE defined behavior to prevent common weaknesses. We present these results in Table 6. The columns labeled by a CWE number and a '+' refer to all the CWEs given in a row of Table 5. The remaining two Isadora properties were single source or single sink properties representing a logical combination inside the decoder, and captured only functional correctness.

6 DISCUSSION

In this section, we discuss the threats to validity for properties produced using Isadora, including false positives and false negatives.

False positives may be introduced by insufficient trace coverage, by limitations of information flow tracking, or by incorrectly classifying functional properties as security properties. Sampling output properties found a 10% false positivity rate with respect to misclassification. This rate is discussed in greater detail in Section 6.2.

With regard to false negatives, they fall into two cases: known and unknown. Isadora captured all known assertions for the Single ACW (Section 5.3). In our evaluation of the Single ACW and PicoRV32 properties (Section 5.4), the sampled properties partially addressed all CWEs manually determined to be relevant to the studied designs, but no CWE was completely covered within the sampled properties. A manual inspection of the complete set of properties produced by Isadora is needed to rule out the possibility

#	Description	1220	1221+	1258+	1274+	1280
3	Control check for first read request after reset	✓		✓		✓
10	Secure power-on		✓			
37	Anomalies and memory control set after reset	✓		✓	✓	✓
96	*T* via S PORT configures ACW	✓			✓	✓
106	Interrupts respect channel separation	✓				
154	Base address not visible to *P* during reset			✓		
163	Write transaction legality flows to *P*	✓				
227	Write channel anomaly register updates	✓			✓	
239	Write validity respects channel separation, reset	✓		✓		
252	Read validity respects channel separation, reset	✓		✓		

Table 4: Sampled Isadora properties on Single ACW

CWE(s)	Description
276-1221-1271	Correct initialization, reset, defaults
440-1234-1280-1299	Memory accesses pass validity checks
1190	Memory isolated before reset
1191-1243-1244...	Debug signals do not interfer with
-1258-1295-1313	...any other signals
1245	Hardware state machine correctness
1252-1254-1264	Data and control separation

Table 5: The 18 CWEs considered for PicoRV32

of false negatives with respect to CWEs. Unknown false negatives can arise from limitations in trace coverage or in logical specificity.

6.1 Trace Reliance

As with any specification mining technique, Isadora relies on traces. The second stage of Isadora relies on generating instrumented traces with sufficient case coverage to drive information flow through all channels present in the design. The third stage of Isadora relies on traces to infer flow predicates. Over buggy hardware, these predicates may form a specification describing buggy behavior. Traces may not cover all cases that can be reached by a design or even occur during normal design operation.

Traces may not precisely describe some design features. For example, when considering property number 154 on the Single ACW, one of the sampled properties, Isadora found predicates that ARLEN_wire and AWLEN_wire are both set to be exactly 8 for any flow to occur. This property is shown in full in Appendix A.1.

The AxLEN_wire registers set transaction burst size for reads and writes. For transactions in write channels, the ARLEN_wire value should be irrelevant, and this clause within the broader property constitutes a likely false positive.

The AWLEN_wire is a different case. In a properly configured write channel supporting transcations, this register would necessarily be non-zero, and for wrapping bursts must be a power of two, but manual inspection of the code provides no indication the value must be precisely 8. During development we manipulated this and other values for which similar reasoning applied, but ultimately it was difficult to tightly define possible values for which the design could operate but were distinct from the default test bench for this and other signals.

While Isadora is testbench reliant, and may be useful in testbench generation, testbench generation is an active area of research, and is more fully explored in related works such as Meng et al. [33], which studies concolic testing for RTL.

6.2 Functional Properties

When using CWE-relevance as the metric, Isadora does include functional properties in its output, as shown in Table 6. Sampling output properties found a 10% false positivity rate with respect to misclassification for the sampled properties from both designs, with 0 of 10 properties found to be false positives over the Single ACW version of AKER, and 2 of 10 properties found to be false positives over the PicoRV32 RISC-V CPU.

We attribute finding functional properties solely on RISC-V primarily to differences in design and testbench. The ACW studied was the target of validation efforts related to information flow, and the testbench we used was developed as part of those efforts. Further, as an access control module, by nature much of its functionality was relevant to secure access control.

With RISC-V, a minimal test bench was used that was intended only to run the design in an environment without access to the full RISC-V toolchain (such as our simulation environment for instrumented trace generation), and much of the design was devoted to behavior for which CWEs did not apply, such as logical updates during instruction decoding. One example of an Isadora property classified as functional is shown in Appendix A.2.

6.3 Measuring Interference

Isadora assumes the correctness of the information flow tracking used in trace generation. Information flow tracking is an active area of research, and is more fully explored in related works such as Ardeshiricham et al. [6], which studies IFT for RTL.

6.4 Specification Logic

Isadora does not define temporal properties beyond a single delay slot incorporated in the trace slices of length two. However, manual examination of output properties suggests information flow patterns during initialization, which is the first 4 cycles for AKER and

#	Description	276+	440+	1190	1191+	1245	1252+
1	No decoder leakage via debug				✓		
16	Instructions update state machine		✓			✓	
30	Decoder updates state machine		✓				
47	No state machine leakage via debug				✓		
52	SLT updates state machine					✓	
66	Handling of jump and load			✓	✓		✓
79	Loads update state machine					✓	
113	Decoder internal update						
130	Write validity respects reset					✓	
144	Decoder internal update						

Table 6: Sampled Isadora properties on PicoRV32

first 80 for RISC-V, are highly dissimilar to later flows. During initialization, Isadora discovers flow conditions referencing registers with unknown states. Isadora also finds concurrent flows between elements for which no concurrent flows occur after reset. Because conditions are inferred from comingled trace slices from during and after initialization, the output properties may be insufficiently precise to capture secure behavior related to this boundary.

7 RELATED WORK

7.1 Properties of Hardware Designs

Automatic extraction of security critical assertions from hardware designs enables assertion based verification without first manually defining properties [28]. The Iodine tool looks for possible instances of known design patterns, such as one-hot encoding or mutual exclusion between signals, and creates assertions that encode the discovered patterns [25]. More recent papers use data mining of simulation traces to extract more detailed assertions [11, 26] or temporal properties [31]. Recent work has focused on mining temporal properties from execution traces [13–15, 31]. A combination of static and dynamic analysis extracts word-level properties [32].

The first security properties developed for hardware designs were manually crafted [9, 10, 27]. SCIFinder semi-automatically generates security-critical properties for a RISC processor design [45] and Astarte generates security-critical properties for x86 [19]. Recent hackathons have revealed the types of properties needed to find exploitable bugs in the design of a RISC-based system-on-chip [16].

7.2 Mining Specifications for Software

The seminal work in specification mining comes from the software domain [4] in which execution traces are examined to infer temporal specifications in the form of regular expressions. Subsequent work used both static and dynamic traces to filter out less useful candidate specifications [42]. More recent work has tackled the challenges posed by having imperfect execution traces [44], and by the complexity of the search space [22, 23, 38]. Daikon, which produces invariants rather than temporal properties, learns properties that express desired semantics of a program [20].

In the software domain a number of papers have developed security specific specification mining tools. These tools use human specified rules [40], observe instances of deviant behavior [20, 34, 35], or identify instances of known bugs [43].

8 CONCLUSION

We presented and implemented a methodology for creating information flow specifications of hardware designs. By combining information flow tracking and specification mining, we are able to produce information flow properties of a design without prior knowledge of security agreements or specifications. We show our implementation, Isadora, characterizes the flow relations between all elements of a design and identifies important information flow security properties of an SoC and a CPU according to Common Weakness Enumerations.

9 ACKNOWLEDGMENTS

We thank our reviewers for their insightful comments and suggestions. This material is based upon work supported by the National Science Foundation under Grant No. CNS-1816637, by the Semiconductor Research Corporation, and by Intel. Any opinions, findings, conclusions, and recommendations expressed in this paper are solely those of the authors.

REFERENCES

[1] [n.d.]. The Common Weakness Enumeration Official Webpage. https://cwe.mitre.org/

[2] [n.d.]. Questa Advanced Simulator. https://eda.sw.siemens.com/en-US/ic/questa/simulation/advanced-simulator/

[3] [n.d.]. Radix Coverage for Hardware Common Weakness Enumeration (CWE) Guide. https://tortugalogic.com/wp-content/uploads/2020/03/RadixCWEGuide_20210126.pdf.

[4] Glenn Ammons, Rastislav Bodík, and James R. Larus. 2002. Mining Specifications. In *29th Symposium on Principles of Programming Languages (POPL)* (Portland, Oregon). ACM, 4–16. https://doi.org/10.1145/503272.503275 http://doi.acm.org/10.1145/503272.503275.

[5] Armaiti Ardeshiricham, Wei Hu, and Ryan Kastner. 2017. Clepsydra: Modeling timing flows in hardware designs. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 147–154. https://doi.org/10.1109/ICCAD.2017.8203772

[6] Armaiti Ardeshiricham, Wei Hu, Joshua Marxen, and Ryan Kastner. 2017. Register transfer level information flow tracking for provably secure hardware design. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*. 1691–1696. https://doi.org/10.23919/DATE.2017.7927266

[7] Andrew Becker, Wei Hu, Yu Tai, Philip Brisk, Ryan Kastner, and Paolo Ienne. 2017. Arbitrary precision and complexity tradeoffs for gate-level information flow tracking. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1145/3061639.3062203

[8] Mohammad-Mahdi Bidmeshki and Yiorgos Makris. 2015. Toward automatic proof generation for information flow policies in third-party hardware IP. In *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 163–168.

[9] M. Bilzor, T. Huffmire, C. Irvine, and T. Levin. 2011. Security Checkers: Detecting processor malicious inclusions at runtime. In *International Symposium on Hardware-Oriented Security and Trust (HOST)*. IEEE, 34–39. https://doi.org/10.1109/HST.2011.5954992

[10] Michael Brown. 2017. *Cross-validation Processor Specifications*. Master's Thesis. University of North Carolina at Chapel Hill.

[11] Po-Hsien Chang and Li C Wang. 2010. Automatic assertion extraction via sequential data mining of simulation traces. In *15th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 607–612.

[12] 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.

[13] A. Danese, T. Ghasempouri, and G. Pravadelli. 2015. Automatic extraction of assertions from execution traces of behavioural models. In *Design, Automation Test in Europe Conference Exhibition (DATE)*. 67–72. https://doi.org/10.7873/DATE.2015.0110

[14] A. Danese, G. Pravadelli, and I. Zandonà. 2016. Automatic generation of power state machines through dynamic mining of temporal assertions. In *Design, Automation Test in Europe Conference Exhibition (DATE)*. 606–611.

[15] A. Danese, N. D. Riva, and G. Pravadelli. 2017. A-TEAM: Automatic template-based assertion miner. In *54th Design Automation Conference (DAC)*. ACM/EDAC/IEEE, 1–6. https://doi.org/10.1145/3061639.3062206

[16] Ghada Dessouky, David Gens, Patrick Haney, Garrett Persyn, Arun Kanuparthi, Hareesh Khattri, Jason M Fung, Ahmad-Reza Sadeghi, and Jeyavijayan Rajendran. 2019. Hardfails: Insights into Software-Exploitable Hardware Bugs. In *28th USENIX Security Symposium*. USENIX Association, 213–230. https://www.usenix.org/conference/usenixsecurity19/presentation/dessouky

[17] Calvin Deutschbein and Cynthia Sturton. 2018. Mining Security Critical Linear Temporal Logic Specifications for Processors. In *International Workshop on Microprocessor and SoC Test, Security, and Verification (MTV)*. IEEE. https://ieeexplore.ieee.org/document/8746060

[18] C. Deutschbein and C. Sturton. 2020. Evaluating Security Specification Mining for a CISC Architecture. In *2020 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 164–175. https://doi.org/10.1109/HOST45689.2020.9300291

[19] Calvin Deutschbein and Cynthia Sturton. 2020. Evaluating Security Specification Mining for a CISC Architecture. In *Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE.

[20] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon System for Dynamic Detection of Likely Invariants. *Science of Computer Programming* 69, 1-3 (Dec. 2007), 35–45. https://doi.org/10.1016/j.scico.2007.01.015 http://dx.doi.org/10.1016/j.scico.2007.01.015.

[21] 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. https://doi.org/10.1109/ITC44170.2019.9000170

[22] Mark Gabel and Zhendong Su. 2008. Javert: Fully Automatic Mining of General Temporal Properties from Dynamic Traces. In *16th International Symposium on Foundations of Software Engineering (FSE)* (Atlanta, Georgia). ACM, 339–349. https://doi.org/10.1145/1453101.1453150 http://doi.acm.org/10.1145/1453101.1453150.

[23] Mark Gabel and Zhendong Su. 2008. Symbolic Mining of Temporal Specifications. In *30th International Conference on Software Engineering (ICSE)* (Leipzig, Germany). ACM, 51–60. https://doi.org/10.1145/1368088.1368096 http://doi.acm.org/10.1145/1368088.1368096.

[24] Sudheendra Hangal, Naveen Chandra, Sridhar Narayanan, and Sandeep Chakravorty. 2005. IODINE: a tool to automatically infer dynamic invariants for hardware designs. In *42nd annual Design Automation Conference*. ACM, 775–778. http://xenon.stanford.edu/~hangal/iodine.html

[25] Sudheendra Hangal, Sridhar Narayanan, Naveen Chandra, and Sandeep Chakravorty. 2005. IODINE: A tool to automatically infer dynamic invariants for hardware designs. In *42nd Design Automation Conference (DAC)*. IEEE.

[26] Stav Hertz, David Sheridan, and Shobha Vasudevan. 2013. Mining hardware assertions with guidance from static analysis. *Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32, 6 (2013), 952–965.

[27] 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 *Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Istanbul, Turkey). ACM, 517–529. https://doi.org/10.1145/2694344.2694366 http://doi.acm.org/10.1145/2694344.2694366.

[28] Wei Hu, Alric Althoff, Armaiti Ardeshiricham, and Ryan Kastner. 2016. Towards property driven hardware security. In *2016 17th International Workshop on Microprocessor and SOC Test and Verification (MTV)*. IEEE, 51–56.

[29] Wei Hu, Armaiti Ardeshiricham, Mustafa S Gobulukoglu, Xinmu Wang, and Ryan Kastner. 2018. Property Specific Information Flow Analysis for Hardware Security Verification *(ICCAD '18)*. ACM. https://doi.org/10.1145/3240765.3240839

[30] Wei Hu, Dejun Mu, Jason Oberg, Baolei Mao, Mohit Tiwari, Timothy Sherwood, and Ryan Kastner. 2014. Gate-Level Information Flow Tracking for Security Lattices. *ACM Trans. Des. Autom. Electron. Syst.* 20, 1, Article 2 (Nov. 2014), 25 pages. https://doi.org/10.1145/2676548 https://doi.org/10.1145/2676548.

[31] Wenchao Li, Alessandro Forin, and Sanjit A. Seshia. 2010. Scalable Specification Mining for Verification and Diagnosis. In *47th Design Automation Conference (DAC)* (Anaheim, California). ACM, 755–760. http://doi.acm.org/10.1145/1837274.1837466.

[32] L. Liu, C. Lin, and S. Vasudevan. 2012. Word level feature discovery to enhance quality of assertion mining. In *International Conference on Computer-Aided Design (ICCAD)*. IEEE/ACM, 210–217.

[33] Xingyu Meng, Shamik Kundu, Arun K. Kanuparthi, and Kanad Basu. 2021. RTL-ConTest: Concolic Testing on RTL for Detecting Security Vulnerabilities. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2021), 1–1. https://doi.org/10.1109/TCAD.2021.3066560

[34] Changwoo Min, Sanidhya Kashyap, Byoungyoung Lee, Chengyu Song, and Taesoo Kim. 2015. Cross-checking Semantic Correctness: The Case of Finding File System Bugs. In *25th Symposium on Operating Systems Principles (SOSP)* (Monterey, California). ACM, 361–377. https://doi.org/10.1145/2815400.2815422 http://doi.acm.org/10.1145/2815400.2815422.

[35] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. 2009. Automatically Patching Errors in Deployed Software. In *22nd Symposium on Operating Systems Principles (SOSP)* (Big Sky, Montana, USA). ACM, 87–102. https://doi.org/10.1145/1629575.1629585 http://doi.acm.org/10.1145/1629575.1629585.

[36] Christian Pilato, Kaijie Wu, Siddharth Garg, Ramesh Karri, and Francesco Regazzoni. 2019. TaintHLS: High-Level Synthesis for Dynamic Information Flow Tracking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 5 (2019), 798–808. https://doi.org/10.1109/TCAD.2018.2834421

[37] Mayank Rawat, Sujit Kumar Muduli, and Pramod Subramanyan. 2020. Mining Hyperproperties from Behavioral Traces. In *2020 IFIP/IEEE 28th International Conference on Very Large Scale Integration (VLSI-SOC)*. 88–93. https://doi.org/10.1109/VLSI-SOC46417.2020.9344106

[38] G. Reger, H. Barringer, and D. Rydeheard. 2013. A pattern-based approach to parametric specification mining. In *28th International Conference on Automated Software Engineering (ASE)*. IEEE/ACM, 658–663. https://doi.org/10.1109/ASE.2013.6693129

[39] Francesco Restuccia, Andres Meza, and Ryan Kastner. 2021. AKER: A Design and Verification Framework for Safe and Secure SoC Access Control. *CoRR* abs/2106.13263 (2021). arXiv:2106.13263 https://arxiv.org/abs/2106.13263

[40] Lin Tan, Xiaolan Zhang, Xiao Ma, Weiwei Xiong, and Yuanyuan Zhou. 2008. AutoISES: Automatically Inferring Security Specifications and Detecting Violations. In *17th USENIX Security Symposium* (San Jose, CA). USENIX Association, 379–394. http://dl.acm.org/citation.cfm?id=1496711.1496737.

[41] Wei Hu, A. Becker, A. Ardeshiricham, Yu Tai, P. Ienne, D. Mu, and R. Kastner. 2016. Imprecise security: Quality and complexity tradeoffs for hardware information flow tracking. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–8. https://doi.org/10.1145/2966986.2967046

[42] Westley Weimer and George C. Necula. 2005. Mining Temporal Specifications for Error Detection. In *11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (Edinburgh, UK). Springer-Verlag, 461–476. https://doi.org/10.1007/978-3-540-31980-1_30 http://dx.doi.org/10.1007/978-3-540-31980-1_30.

[43] Fabian Yamaguchi, Felix Lindner, and Konrad Rieck. 2011. Vulnerability Extrapolation: Assisted Discovery of Vulnerabilities Using Machine Learning. In *5th USENIX Conference on Offensive Technologies (WOOT)* (San Francisco, CA). USENIX Association, 13–13. http://dl.acm.org/citation.cfm?id=2028052.2028065.

[44] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. 2006. Perracotta: Mining Temporal API Rules from Imperfect Traces. In *28th International Conference on Software Engineering (ICSE)* (Shanghai, China). ACM, 282–291. https://doi.org/10.1145/1134285.1134325 http://doi.acm.org/10.1145/1134285.1134325.

[45] Rui Zhang, Natalie Stanley, Chris Griggs, Andrew Chi, and Cynthia Sturton. 2017. Identifying Security Critical Properties for the Dynamic Verification of a Processor. In *Architectural Support for Prog. Lang. and Operating Sys. (ASPLOS)*. ACM.

A SAMPLE PROPERTIES

In this section we show examples of Isadora output.

A.1 Case 154: ACW Security Property

```
case 154: 2_121_250_379_543
_src_ in {w_base_addr_wire, M_AXI_AWREADY_wire,
AW_CH_DIS,  w_max_outs_wire, AW_ILLEGAL_REQ,
w_num_trans_wire, AW_STATE,  AW_CH_EN}
=/=>
_snk_ in {M_AXI_WDATA}
unless
0 != _inv_ in {ADDR_LSB, ARESETN, M_AXI_ARBURST_wire,
M_AXI_ARCACHE_wire, M_AXI_ARLEN_wire, M_AXI_ARREADY,
M_AXI_ARSIZE_wire, M_AXI_AWBURST_wire,
M_AXI_AWCACHE_wire, M_AXI_AWLEN_wire, M_AXI_AWREADY,
M_AXI_AWSIZE_wire, M_AXI_BREADY, M_AXI_BREADY_wire,
M_AXI_WREADY, M_AXI_WREADY_wire, M_AXI_WSTRB_wire,
OPT_MEM_ADDR_BITS, S_AXI_CTRL_BREADY,
S_AXI_CTRL_RREADY, data_val_wire, r_burst_len_wire,
r_displ_wire, r_max_outs_wire, r_num_trans_wire,
r_phase_wire, w_burst_len_wire, w_displ_wire,
w_max_outs_wire, w_num_trans_wire, w_phase_wire}
```

Figure 6: An example of an Isadora property, Case 154, over the Single ACW

To consider the output properties of Isadora, Figures 6 shows an example of Isadora output, Case 154 of the 303 output properties over the ACW module. This a case that was sampled during evaluation. Here the condition predicates shown are register equality testing versus zero. Other predicates are captured within the workflow but not propagated to individual properties formatted for output.

A visible difference between an Isadora output property and the property grammar of Section 2 is that at output stage Isadora properties may specify multiple source registers, may consider multiple sink registers though do not do so in this case, and may contain multiple invariants as conditions.

Case 154 includes an example of a flow condition between internal and peripheral visible signals in addition to specifying other aspects of design behavior. This is similar to the example of write readiness from Section 2, but in Case 154, the flow is from the internal signal to the peripheral, though the power state predicate is identical. Of note, as in the case of write readiness, this flow occurs exclusively within the write channel, as denoted by the "W" present in ready wire and the data register.

$$\texttt{AWREADY\_int} =/=> \texttt{WDATA} \text{ unless } (\texttt{ARESETN} \neq 0)$$

A.1.1 Security Relevance. Under the working definition of security properties for Isadora, where internal signals and peripheral signals should not flow to one another unless ACW is not undergoing a reset, this single source, single sink, single invariant description of behavior composed from an Isadora output property establishes Case 154 as a security property under the working definition. Case 154 describes signals marked as sensitive by designers, both labeled as such within the design using comments and present within security properties they specified, and differs from a designer provided property only in the specific pairing of registers.

A.2 Case 144: ACW Functional Property

One example of an Isadora property classified as functional, with truncated flow conditions, is presented in Figure 7, and captures a logical update to an internal decoder signal. This additional shows an example of a property over multiple sinks, a single source, and for which there are predicates capturing both equality and inequality to zero.

```
case 144: 128
_src_ in {instr_lw}
=/=>
_snk_ in {is_slti_blt_slt, is_sltiu_bltu_sltu}
unless
0 == _r_ in {alu_eq, alu_shl, alu_shr,  ... }
0 != _r_ in {alu_add_sub, alu_lts, alu_ltu, ... }
```

Figure 7: An example of an Isadora property, Case 144, over RISC-V.