Toward Hardware Security Property Generation at Scale

Calvin Deutschbein | Willamette University
Andres Meza and Francesco Restuccia | University of California San Diego
Matthew Gregoire | University of North Carolina at Chapel Hill
Ryan Kastner | University of California San Diego
Cynthia Sturton | University of North Carolina at Chapel Hill

Using formal methods requires first developing the properties to be verified, which is a difficult and time-consuming task. This article describes our research on automating the property generation process for information-flow properties that are critical to the security of hardware designs.

One barrier to using formal methods is not having the right properties to prove. Formal methods, ranging from lightweight symbolic execution to midweight model checking to full formal proofs of correctness, all require that a property or set of properties (or theorems in the case of correctness proofs) first be stated in the formal specification language of the tool.

Model checking has long been a part of the design verification stage for hardware designs. More recently, security assurance teams inside hardware companies have begun to incorporate formal methods into the security validation process as well. The state-of-the-art commercial tools use symbolic simulation, a variant of model checking, to verify properties related to how information can flow through the design. This type of verification is crucial to the effort to identify information leakage and access control violation vulnerabilities early in the design stages, but the verification effort is only as strong as the properties verified.

This article presents our research into automating the generation of security properties capturing how information flows through a hardware design.[1] These properties form an information-flow specification of the design as studied and capture high-level notions of best design practices or designed intent in formally falsifiable register transfer level expressions. These expressions, especially unexpected expressions, can then be further studied with formal verification tools, checked against Common Weakness Enumerations (CWEs), and compared to design intent. The methodology we developed joins together two disparate fields of research: information-flow tracking (IFT) and security specification mining, in a novel technique whereby IFT logic makes information-flow properties expressible as trace properties amenable to mining. The resulting tool, Isadora, creates an 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. Isadora

Digital Object Identifier 10.1109/MSEC.2022.3155376
Date of current version: 29 April 2022

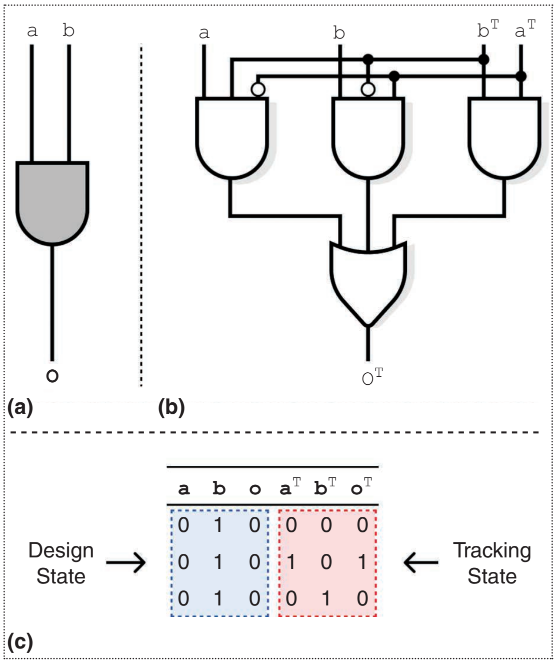
requires no input from the designer beyond the design and testbench.

Hardware Security Specification Mining

Hardware security specification mining automates the development of security properties for use with assertion-based verification techniques. Initial research in this area was limited to producing *trace properties*—properties that can be evaluated over individual traces of execution. Unfortunately, information-flow properties are not expressible as trace properties. Analysis situated in the context of a single trace of execution cannot show whether or not information has flowed from a particular source to a particular sink. Information-flow properties fall into a more complex set of properties called *hyperproperties*, which express behaviors over sets of traces. However, combining security specification mining with hardware IFT can produce information-flow properties. We provide a brief background on both IFT and security specification mining.

IFT

A foundational theory within system security is the lattice model of secure information flow, which establishes a means of defining and restricting information flow through a system to meet a set of security



a	b	o	a^T	b^T	o^T
0	1	0	0	0	0
0	1	0	1	0	1
0	1	0	0	1	0

Figure 1. IFT for a two-input AND gate. (a) A two-input AND gate with inputs a, b, and output o. (b) A tracking logic showing a logical model of information-flow propagation. Two additional variables a^T and b^T correspond to whether a and b carry tainted information. The tracking logic computes the taint of the output variable o^T. (c) A partial truth table for the tracking logic highlighting that it is possible to have different tracking states for the same design state.

requirements.[2] The first practical applications of the lattice used a notion of labels to monitor how information flowed through a system. More recently, IFT was developed to analyze hardware designs.

Hardware IFT provides a powerful mechanism that can be used to precisely measure digital information flows[3] and can validate security properties related to confidentiality, integrity, and availability. Hardware IFT techniques can be used within the context of formal verification, simulation-based validation, emulation-based validation, and runtime checking. Commercial hardware IFT tools now exist from companies like Siemens EDA, Cadence, and Tortuga Logic.

The major challenge with using hardware IFT tools is that one must first develop the properties to be validated. Isadora addresses that problem.

IFT generally works as follows. For each signal s in the design, a new tracking signal s^T is added along with the logic needed to track how information propagates through a design. A set of tracking signals is initially marked at *tainted* by setting the signals' s^T value as a nonzero value. This indicates that the signals contain important information relative to the security of the design. All other variables are marked as *nontainted*; that is, their tracking signal is set to a value of zero. The IFT tools analyze how that information moves throughout the system using some combination of formal methods, simulation, emulation, or at runtime checking. Security properties are required to indicate what signals to track and to specify allowable (and nonallowable) flows of information.

Figure 1 shows a simple example tracking information flow through an AND gate. This logic performs precise IFT, meaning that it considers the functional input values when determining the taint value of the output value. This models cases where certain input values stop the flow of information. For example, consider a scenario where $a^T = 0$ (a is not tainted) and $b^T = 1$ (b is tainted); that is, we wish to understand how information flows from signal b. When the design state is $a = 0$ and $b = 1$, the output will not be tainted ($o^T = 0$ despite one of the inputs being tainted). This occurs because the functional value of a dominates the output, and thus, stops the flow of taint from b. Precision plays an important role in hardware IFT.[4]

The key challenge for hardware security validation is understanding which signals to track and where that information should or should not flow during execution. This equates to writing IFT security properties. Currently, this is a challenging, mostly manual process. Automating property generation is a fundamental challenge in hardware security validation.

Security Specification Mining

The formative work in specification mining comes from the software domain, in which execution traces are examined to infer temporal specifications as regular expressions. Hardware miners soon followed suit with tools such as IODINE, which looks for possible instances of known design patterns, such as one-hot encoding or mutual exclusion between signals,[5] and Goldmine, which combines trace mining with static analysis.[6] Sturton and her team[7] developed specification mining for security-critical properties of a hardware design; their approach used machine learning and a starting set of known vulnerabilities to find new properties capable of exposing previously unknown vulnerabilities. In subsequent work, Deutschbein and Sturton[8] used known security properties to develop templates for further mining at scale. In Isadora, we move from generating security-critical trace properties to generating information-flow properties of a design.

Specification mining works by looking for patterns of behavior that can be defined over a single trace of execution, rather than across multiple traces of execution, and is therefore limited to finding only trace properties. To achieve our objective of capturing information flows, our approach is to first instrument a design with IFT logic and then apply security specification mining techniques to the instrumented design. The miner is still producing only trace properties, but now, the properties include information from the tracking signals.

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 the purposes of security validation.

No-Flow Properties

Using IFT, 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 $\mathtt{r}_2$'s tracking signal $\mathtt{r}_2^T$ should remain at zero:

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

The $\mathbb{G}$ operator is borrowed from temporal logic and indicates that the consequent is globally true. In other words, $\mathtt{r}_2^T$ will remain equal to zero at all future time steps after the antecedent is satisfied. This style of no-flow property can be useful for ensuring that unprivileged users cannot influence sensitive states or for ensuring that sensitive information cannot leak through, for example, debug ports. However, it cannot capture conditional properties, such as a property that register updates are allowed only under certain power states.

Conditional-Flow Properties

Using IFT, 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 $\mathtt{r}_2$'s tracking signal $\mathtt{r}_2^T$ may become nonzero only if some predicate P holds:

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

The $\mathbb{X}$ operator is borrowed from temporal logic and indicates the next time-step. If $\mathtt{r}_2^T$ equals zero at some clock cycle, then in the next clock cycle, it will still be zero. 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.

No-Flow Operator

To express properties in a format that uses 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 $=/=>$. Using this operator, no-flow properties can be expressed as $\mathtt{r}_1 =/=> \mathtt{r}_2$, where $\mathtt{r}_1$ and $\mathtt{r}_2$ are registers in the design.

Conditional-flow properties can be expressed as $\mathtt{r}_1 =/=> \mathtt{r}_2 \rightarrow e$, which states that information may flow from $\mathtt{r}_1$ to $\mathtt{r}_2$ but requires that a design not be some state e, where e is a Boolean expression in the form of a conjunction of comparison relations between two registers or between a register and a constant.

Isadora

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 key to making the approach work is to synchronize the two parts using clock-cycle time.

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 2.

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 the 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[9]) 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.

Trace Generation

To generate traces, we first instrument our original design with logic to track information flows and then execute the testbench in simulation on this instrumented design. Each trace is a sequence of states that this design assumes. Let $\tau_{\texttt{src}} = \langle \sigma_0, \sigma_1, \ldots, \sigma_n \rangle$ be the trace of our design, instrumented to track how information flows from one signal, src, during testbench execution. Each state σ_i is a list of triples that store each design signal's name, value, and corresponding tracking signal:

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

To distinguish the source of a tainted sink signal, each input signal must have a separate taint label. However, tracking multiple labels is expensive. 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, and these attached labels can be used within Isadora or for other design explorations, such as formal verification. The end result is a set of traces, each of which 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 the execution of the provided testbench.

Finding All Flows

In the second phase, the set of traces is analyzed to identify, first, every pair of signals between which a flow occurs, and second, the times within the trace at which each flow occurs. Each trace $\tau_{\texttt{src}}$ is searched to find every clock cycle in which a taint's tracking signal transitions from zero to one. In other words, every signal-values triple $(\texttt{s}, v, v^t)$ that is of the form $(\texttt{s}, v, \texttt{0})$ in state σ_{i-1} and $(\texttt{s}, v, \texttt{1})$ in state σ_i is found, and the time i is noted. Isadora stores this as the tuple $(\texttt{src}, \texttt{s}, \{i_0, i_1, \ldots\})$, which indicates that an information flow from src to s occurred at each time $i \in \{i_0, i_1, \ldots\}$. 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 (for example, resets).

Once all traces have been analyzed, the collected time-of-flow tuples $(\texttt{src}, \texttt{s}, \{i_0, i_1, \ldots\})$ are reorganized by time. It's possible that, for a given set of times $\{i_0, i_1, \ldots\}$,

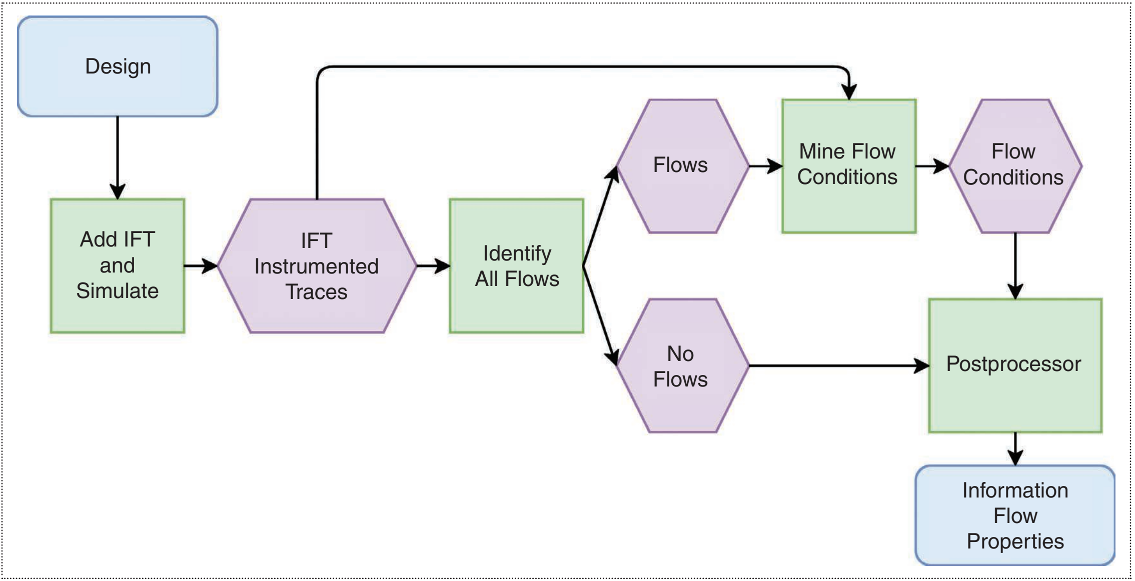


Figure 2. An overview of the Isadora workflow.

there exist multiple discovered information flows. For each such set of times output by the previous step, Isadora creates a set $\{(\mathtt{src}_1, \mathtt{s}_1), (\mathtt{src}_2, \mathtt{s}_2), \ldots\}$ of all flows of information that occurred at exactly those times. This mapping from sets of times to corresponding flows between registers is one of the tool's two outputs from this stage.

The other output is a list of source-sink pairs $\{(\mathtt{src}, \mathtt{s}), (\mathtt{src}', \mathtt{s}'), \ldots\}$ between which information never flows. The pairs in this set comprise the noninterference properties of the design and can be specified using the no-flow operator. For example, $\mathtt{src} =/=> \mathtt{s}$.

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:

$$\neg(\mathtt{r} = x) \rightarrow \mathtt{src} =/=> \mathtt{s}.$$

Isadora uses the technique of dynamic invariant detection[9] on traces to infer design behavior using predefined patterns. To isolate the conditions for information flow between two registers, Isadora analyzes the flow pair output by the previous step to find all trace times i at which information flows from `src` to `s` during the execution of the testbench. We then analyze the corresponding trace to produce a set of trace slices of the form $\langle \sigma_{i-1}, \sigma_i \rangle$, one for each time i at which a flow occurred. These trace slices include only the signals of the original design; all tracking logic and shadow signals are pruned. Using these trace slices of length two allows dynamic invariant detection to generate predicates specifying the design state both immediately prior to and concurrent with the occurrence of some flow as either could potentially contain security-relevant information. These predicates are used as the Boolean expressions e in the conditional no-flow operator as described previously.

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. For example, the trivial invariant $\mathtt{clk} \in \{0,1\}$ always holds and would be removed.

The final output properties from postprocessing are the conditional-flow properties. For readability, Isadora can express the conditional-flow properties as multisource-to-multisink flows, collapsing together all source-to-sink flows that occur under identical conditions. This produces comparatively few properties, which in practice were approximately as many as the number of unique source signals, and avoids redundant information. These conditional-flow properties as well as the no-flow properties discovered earlier are the final set of information-flow properties produced by Isadora.

Evaluating the Security Properties

We evaluated Isadora on two designs: the PicoRV32 CPU core, an open source, open-standards processor, and the access control wrapper (ACW) of AKER, a system-on-chip (SoC) verification framework developed by three of the authors in separate research.[10] The PicoRV32 CPU core is a 32-bit, single-core, unpipelined CPU implementing the RISC-V RV32IMC instruction set, which is an open standard instruction-set architecture based on established reduced-instruction-set computer (RISC) principles.

The ACW wraps an AXI controller and enforces on it a local access control policy, which is set up and maintained by a trusted entity (for example, 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 studied the ACW in two configurations: first, implementing a single-controller AKER-based access control system, and 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.

Table 1 provides details about the size of the designs, the number of properties generated for each design, and the time it took to generate the properties. This time was dominated by the trace generation phase. For the multi-ACW, trace generation exceeded 24 h, so we consider a reduced trace, which tracks sources for one of the ACWs, though all signals are included as sinks or in conditions.

Assessing the Relevancy of the Properties

For the two designs with full trace sets, the single ACW and PicoRV32, Isadora generates a specification describing all information 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 used the CWE database[11] as a metric to evaluate the security relevance of Isadora output properties. To do so, for each design, we first determined which CWEs apply to the design. For both the ACW and PicoRV32, we used the "Radix Coverage for Hardware Common Weakness Enumeration (CWE) Guide"[12] 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

Table 1. Various size measures of the studied designs.

Design	Unique signals	Unique sources	LoC	Trace cycles	Trace GBs	Daikon traces	Isadora properties	Miner time (minutes)
Single ACW	229	229	1,940	598	.7	252	303	29:51
Multi-ACW	984	85	4,447	848	4.3	378	160	8:31
PicoRV32	181	181	3,140	1,099	.6	955	153	15:09

correspond to the same low-level behavior for a design, and we consider these CWEs together.

ACW conditional information flow. Over the ACW, we assessed 14 CWEs, which we mapped to five plain-language descriptions of the design features, as shown in Table 2. For the ACW, all 10 sampled properties encode CWE-defined behavior to prevent common weaknesses, as shown in Table 3. In this table, the columns labeled by a CWE number and a + refer to all the CWEs given in a row of Table 2. Eight out of the 10 properties provide separation between read and write channels, which constitutes the main functionality of the ACW module. CWEs 1,267, 1,269, and 1,282 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 that they are included in Isadora's set of no-flow properties.

PicoRV32. Over PicoRV32, we assessed 18 CWEs, which we mapped to seven plain-language descriptions of the design features, as shown in Table 4. For PicoRV32, we found that eight of 10 sampled properties encode CWE-defined behavior to prevent common weaknesses. These results are shown in Table 5. The columns labeled by a CWE number and a + refer to all the CWEs given in a row of Table 4. The remaining two Isadora properties were single-source or single-sink properties representing a logical combination inside the decoder and captured only functional correctness.

Table 2. The 14 CWEs considered for ACW.

CWE(s)	Description
1,220	Read/write channel separation
1,221–1,259–1,271	Correct initialization, reset, and defaults
1,258–1,266–1,270–1,272	Access controls use operating modes
1,274–1,283	Anomaly registers log transactions
1,280	Control checks precede access
1,267–1,269–1,282	Configuration/user port separation

Visualizing Information Flows

The properties generated by Isadora can be visualized on a heatmap, which can help designers understand where

Table 3. Sampled Isadora properties on the single ACW.

Number	Description	1,220	1,221+	1,258+	1,274+	1,280
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 and reset	✓		✓		
252	Read validity respects channel separation and reset	✓		✓		

information is flowing through the design. This visualization can also provide useful feedback to Isadora.

The heatmap shown in Figure 3 groups signals in the single ACW design according to their function and shows how many conditional information flows are found between and within the groups. The seven groups are the global ports (GLOB), AXI secondary interface ports (S PORT), connections to non-AXI ports of the controller (C PORT), AXI main interface ports of the ACW (M PORT), configuration signals (CNFG), AXI main interface ports of the controller (M INT), and control logic signals (CTRL).

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 that 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 that 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.

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

In developing the ACW, the authors manually crafted 80 information-flow properties critical to the security of the module. The three squares outlined in red on the heatmap are where the equivalent Isadora-generated properties fall. The visualization lets us see that handwritten properties may tend to cluster and fail to consider possible information flows outside that cluster.

For each handwritten property, Isadora either generated an equivalent property or found both a violation and the violating conditions for the property. In the cases where Isadora found a violation of a handwritten property, it was because the handwritten property was too conservative, forbidding a flow that should have been conditionally allowed. Isadora also found the conditions for legality.

The heatmap of PicoRV32 is shown in Figure 4. The seven groups of signals are the output registers (OUT), the internal registers (INT), the memory interface (MEM), the instruction registers (INS), the decoder (DEC), the debug signals and state (DBG), and the main state machine (MSM).

The memory interface and the main state machine were indicated by comments in the code. The instruction

Table 4. The 18 CWEs considered for PicoRV32.

CWE(s)	Description
276–1,221–1,271	Correct initialization, reset, and defaults
440–1,234–1,280–1,299	Memory accesses pass validity checks
1,190	Memory isolated before reset
1,191–1,243–1,244...–1,258–1,295–1,313	Debug signals do not interfere with ... any other signals
1,245	Hardware state machine correctness
1,252–1,254–1,264	Data and control separation

Table 5. Sampled Isadora properties on PicoRV32.

Number	Description	276+	440+	1,190	1,191+	1,245	1,252+
1	No decoder leakage via debug				✓		
16	Instructions update state machine		✓			✓	
30	Decoder updates state machine		✓				
47	No state machine leakage via debug				✓		
52	Machine state updates when setting registers					✓	
66	Handling of jump and load			✓	✓		✓
79	Loads update state machine					✓	
113	Decoder internal update						
130	Write validity respects reset					✓	
144	Decoder internal update						

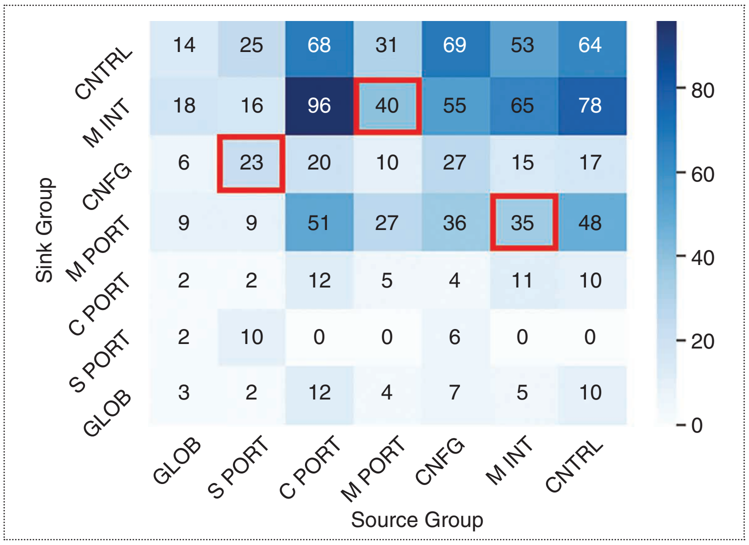


Figure 3. A group-to-group conditional-flow heatmap for the single ACW.

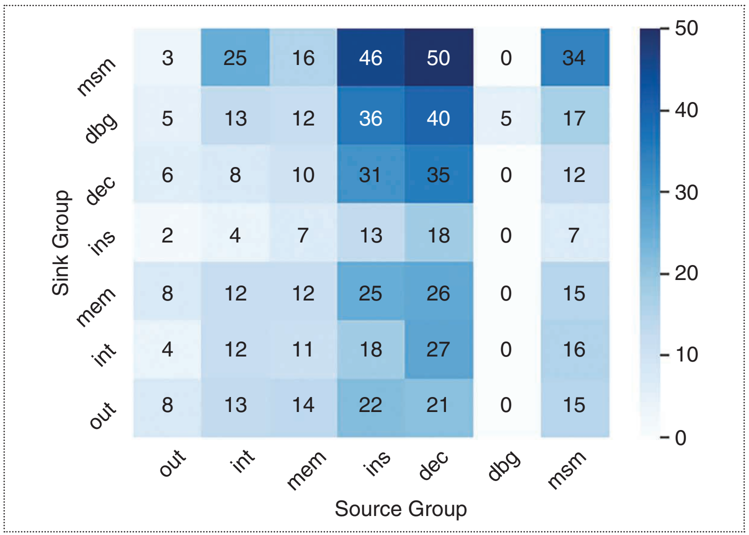


Figure 4. A group-to-group conditional-flow heatmap for the PicoRV32.

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 from 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.

An interesting result visible in this heatmap is the flow isolation from debug signals to the rest of the design. Many exploits, both known and anticipated, target debug information leakage. The lack of flow from the debug signals is a promising indicator, but it might also signal to the designers that more trace data targeting the debug signals should be generated.

The specification and control of information flow through a design is a foundational concept in system security. The automatic generation of information-flow specifications will enable the use of powerful formal verification tools to validate the security of our hardware designs.

In our approach, we see that less formal techniques, like the testbench-and-simulation-based trace generation required for the miner, can be used to support the use of more formal techniques, like model checking-based hardware verification tools. The miner can reliably find valuable security information-flow properties that can then be verified by formal methods. Our experiments so far suggest that for designs of any reasonable complexity, specification mining will outperform humans, finding security-critical properties that a human would not have. This also aligns with the findings of our prior work on security-critical trace properties.

We believe that using formal methods at scale will require the development of the appropriate properties at scale. This research is one step toward achieving that goal. ■

References

1. C. Deutschbein, A. Meza, F. Restuccia, R. Kastner, and C. Sturton, “Isadora: Automated information flow property generation for hardware designs,” in *Proc. Workshop Attacks Solutions Hardware Security (ASHES)*, ACM, 2021, pp. 5–15, doi: 10.1145/3474376.3487286.
2. D. E. Denning, “A lattice model of secure information flow,” *Commun. ACM*, vol. 19, no. 5, pp. 236–243, 1976, doi: 10.1145/360051.360056.
3. W. Hu, A. Ardeshiricham, and R. Kastner, “Hardware information flow tracking,” *ACM Comput. Surv. (CSUR)*, vol. 54, no. 4, pp. 1–39, 2021, doi: 10.1145/3447867.

4. W. Hu *et al.*, "Theoretical fundamentals of gate level information flow tracking," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 30, no. 8, pp. 1128–1140, 2011, doi: 10.1109/TCAD.2011.2120970.
5. S. Hangal, S. Narayanan, N. Chandra, and S. Chakravorty, "IODINE: A tool to automatically infer dynamic invariants for hardware designs," in *Proc. 42nd Design Autom. Conf. (DAC)*, 2005, pp. 775–778, doi: 10.1109/DAC.2005.193920.
6. S. Vasudevan, D. Sheridan, S. Patel, D. Tcheng, B. Tuohy, and D. Johnson, "GoldMine: Automatic assertion generation using data mining and static analysis," in *Proc. Conf. Exhib. Design, Autom. Test Europe (DATE)*, 2010, pp. 626–629, doi: 10.1109/DATE.2010.5457129.
7. R. Zhang, N. Stanley, C. Griggs, A. Chi, and C. Sturton, "Identifying security critical properties for the dynamic verification of a processor," in *Proc. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, ACM, 2017, pp. 541–554, doi: 10.1145/3037697.3037734.
8. C. Deutschbein and C. Sturton, "Evaluating security specification mining for a CISC architecture," in *Proc. 2020 IEEE Int. Symp. Hardware Oriented Security Trust (HOST)*, pp. 164–175, doi: 10.1109/HOST45689.2020.9300291.
9. M. D. Ernst *et al.*, "The Daikon system for dynamic detection of likely invariants," *Sci. Comput. Program.*, vol. 69, nos. 1–3, pp. 35–45, Dec. 2007, doi: 10.1016/j.scico.2007.01.015.
10. F. Restuccia, A. Meza, and R. Kastner, "AKER: A design and verification framework for safe and secure soc access control," in *Proc. IEEE/ACM Int. Conf. Comput. Aided Design (ICCAD)*, 2021, pp. 1–9, doi: 10.1109/ICCAD51958.2021.9643538.
11. "2021 CWE most important hardware weaknesses," MITRE, McLean, VA, USA, 2021. https://cwe.mitre.org/
12. "Radix coverage for hardware common weakness enumeration (CWE) guide." Tortuga Logic.com. https://tortugalogic.com/wp-content/uploads/2020/03/RadixCWEGuide_20210126.pdf (Accessed: Dec. 11, 2020).

Calvin Deutschbein is an assistant professor of computer science at Willamette University, Salem, Oregon, 97301, USA, where they study techniques to mine the secure behavior of hardware designs. Their research interests focus on specification mining of hardware for security. Deutschbein received a Ph.D. in computer science from the University of North Carolina at Chapel Hill in 2021 under the advising of Prof. Cynthia Sturton. They are a Member of IEEE. Contact them at ckdeutschbein@willamette.edu.

Andres Meza is a researcher in the Kastner Research Group, University of California San Diego (UCSD), La Jolla, California, 92039, USA. His research interests include hardware security, optimization of machine learning models for hardware deployment, and computer vision. Meza received a B.S. in both computer science and cognitive science with a machine learning and neural computation specialization from UCSD. Contact him at anmeza@ucsd.edu.

Francesco Restuccia is a postdoctoral researcher in the Kastner Research Group, University of California San Diego, La Jolla, California, 92039, USA. Their research interests include predictability, safety, security for hardware acceleration on heterogeneous platforms, cyberphysical systems, and time predictable hardware acceleration of deep neural network models on field-programmable gate array system-on-chip platforms. Restuccia received a Ph.D. in computer engineering (cum laude) from the Scuola Superiore Sant'Anna Pisa, Italy, in 2021. Contact them at frestuccia@ucsd.edu.

Matthew Gregoire is a graduate researcher in the Hardware Security University of North Carolina (UNC) Lab, University of North Carolina at Chapel Hill, Chapel Hill, North Carolina, 27599, USA. His research interests include symbolic execution of hardware for security. Gregoire received a B.S. in both computer science and mathematics from UNC. Contact him at mattyg@cs.unc.edu.

Ryan Kastner is a professor of computer science and engineering at the University of California San Diego, La Jolla, California, 92039, USA, the principal investigator of the Kastner Research Group, and the codirector of both Engineers for Exploration and the Wireless Embedded Systems Master of Advanced Study. His research interests include hardware acceleration, hardware security, and remote sensing. Kastner received a Ph.D. in computer science from the University of California, Los Angeles. He is a Member of IEEE. Contact him at kastner@ucsd.edu.

Cynthia Sturton is an associate professor of computer science and Peter Thacher Grauer Scholar at the University of North Carolina at Chapel Hill, and the principal investigator of the Hardware Security @ UNC lab, University of North Carolina at Chapel Hill, Chapel Hill, North Carolina, 27599, USA. Her research interests include formal methods for hardware security, symbolic execution, and specification mining. Sturton received a Ph.D. in computer science from the University of California, Berkeley. She is a Member of IEEE. Contact her at csturton@cs.unc.edu.