Pathfinder: Constructing Cycle-accurate Taint Graphs for Analyzing Information Flow Traces

Katharina Ceesay-Seitz
ETH Zurich
Switzerland
kceesay@ethz.ch

Flavien Solt
UC Berkeley
USA
flavien.solt@berkeley.edu

Alexander Klukas
ETH Zurich
Switzerland
aklukas@student.ethz.ch

Kaveh Razavi
ETH Zurich
Switzerland
kaveh@ethz.ch

***Abstract*—Hardware Information Flow Tracking (IFT) is gaining traction for detecting security vulnerabilities in hardware designs. Analyzing IFT violation traces can be extremely time-consuming since they often contain hundreds, if not thousands, of signals that need to be manually analyzed to establish the root cause behind the unexpected information flow. To resolve this problem, we introduce taint graphs that provide context as to *where*, *when*, and *why* information flows. To generalize to different IFT verification methods, we first develop a theoretical foundation for unifying taint tracking and self-composition under a common abstraction. Relying on this abstraction, we then build Pathfinder for automatically generating taint graphs from a given Hardware Description Language (HDL) design and a trace of the information flow violation given either by simulators or formal model checkers. We demonstrate the effectiveness of taint graphs in simplifying root cause analysis of information flows through multiple case studies that involve constant-time violations, temporal fencing, hardware Trojans, and Spectre. By extracting only the relevant signals on a path, Pathfinder reduces the number of signals that need to be manually analyzed between 1.6 and 769.9 times in these case studies.**

I. Introduction

With an increasing number of discovered hardware vulnerabilities [1]–[10], non-interference has become a popular security property to verify confidentiality and integrity in hardware designs [5], [11]–[13]. Non-interference is inherently an information flow property. The violation of such a property is caused by an unauthorized information flow, where a secret input is leaked to an observable output (i.e., a confidentiality violation) [5], [11]–[13], or attacker-controlled input maliciously changes the system's state (i.e., an integrity violation) [5], [14]. While existing Information Flow Tracking (IFT) methods can detect such flows, verification engineers currently lack the necessary support to efficiently analyze the root-cause behind these flows. This paper aims to fill this gap.

Information flow tracking. Information flow properties can be verified with various IFT methods, which are built for detecting unwanted information flows that would, for example, breach confidentiality or integrity properties. IFT methods can be broadly categorized into two types: taint tracking and self-composition. Taint tracking augments hardware designs with taint logic and can operate at different levels of abstraction and precision [15]–[18]. Self-composition, also known as miter circuit [6], [11]–[13], [19], relies on juxtaposing two identical instances of a design. Both methods can be used with formal verification [5], [6], [11]–[13], [19]–[21] or simulation [14], [17], [21], [22]. IFT property violations are reported as cluttered violation trace waveforms that intertwine information flows with design signals.

The authors would like to thank the anonymous reviewers for their valuable feedback, and Tobias Kovats for his feedback on Pathfinder and for providing the new CVA6 trace. This work was supported in part by the Swiss State Secretariat for Education, Research and Innovation under contract number MB22.00057 (ERC-StG PROMISE).

Root-cause analysis. The state-of-practice for understanding the root cause of information flows, may it be through taint tracking or self-composition, is to manually analyze the waveforms consisting of thousands of signals over many clock cycles. Verification engineers spend around half (reportedly 47% [23]) of their time debugging failing test cases or formal properties, which slows down hardware development cycles. Determining the signals that lie on an information flow path is similarly a heavy manual burden. Information that is unrelated to the failing property may flow through the design via many paths, obfuscating the actual path. Hence, analyzing IFT traces requires first manually determining which signals are part of the path and then manually ordering them into a temporal sequence, by manually inspecting the source code in parallel. Additionally to extracting the information flow path, verification engineers must analyze design states and understand their intertwinements. These aspects severely complicate root-cause analysis, which is necessary to identify the underlying vulnerability before it can be fixed.

IFT unification. Currently, IFT methods fail at explaining why an information flow exists. Providing context information for a diverse set of IFT methods requires first understanding their similarities. In this paper, we provide a unified formal definition of hardware information flow tracking, which allows us to design a unified violation trace analysis method applicable to all existing IFT methods, for both formal and simulation-based approaches. To build generic support for root-cause analysis of information flows, we first need a common abstraction between taint tracking and self-composition. To this end, we formally prove the correspondence between taint tracking logic and self-composition. These results show that taint tracking and self-composition are equivalent in terms of detection capabilities, apart from the potential imprecision of taint tracking. This unification further allows for designing a representation of information flows that is agnostic to the underlying IFT method.

Taint graphs. We then introduce *Taint Graphs (TGs)*, a novel representation for visualizing information flows by providing contextual spatio-temporal information (i.e., *where* and *when* information propagates). Only inspecting signals on the path is, however, insufficient to understand *why* information propagated unexpectedly and how to resolve the hardware flaws that caused the unwanted flow. To address this, TGs include *flow control* signals, which we define as signals interfering with the information flow path, and which are able to block the information flow entirely, if they have the right values. Thus, these signals hint at the design modifications necessary to block the flow and thus remove the vulnerability.

Pathfinder. We introduce Pathfinder for generating TGs. Pathfinder relies on a *Temporal Information Flow Graph (TIFG)*, which provides spatio-temporal information of all potential information flows in a given Hardware Description Language (HDL) design. Pathfinder then takes an information flow trace and uses the TIFG to generate the

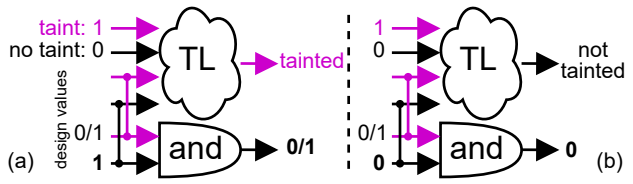

Fig. 1: Taint logic (TL) for an `and` gate propagates taint (from top taint input tracking the top design value) if the untainted input (bottom) has design value 1 (a). No taint propagates if the output of the `and` (0) is *independent* of the tainted input (0/1) (b). [15].

corresponding TG. Pathfinder is able to isolate information flows between chosen information source and sink signals, and it provides means for identifying unknown information sources (or sinks). Due to our IFT unification, Pathfinder can be applied to self-composition traces as well as explicit taint tracking traces. Pathfinder is compatible with state-of-the-art open-source and commercial tools as it relies on standardized tool outputs. We demonstrate the effectiveness of Pathfinder on a range of case studies and show that it significantly reduces the manual effort required to understand the root cause of information flows in hardware.

Contributions. In summary, our contributions are:

- We formally show that taint tracking and self-composition, two different IFT methods, provide similar detection capabilities, enabling the design of a generic information flow representation.
- We introduce taint graphs, a novel representation of information flows that aids verification engineers in understanding the root-cause of non-interference violations, by focusing on *when*, *where* and *why* information was flowing.
- We introduce Pathfinder, a tool that automatically generates taint graphs from a formal counter example or test case violation VCD trace, and the design description in any standard HDL [24].
- We apply Pathfinder to recent non-interference violation reports [5], [19], [21], including Spectre [1], and show that they significantly reduce the manual effort required to understand the underlying vulnerability.

We **open sourced** Pathfinder at https://comsec.ethz.ch/pathfinder.

II. Background and Motivation

Hardware designs undergo extensive simulation and formal verification before production. Verification engineers spend a substantial portion of their time debugging formal counter examples (CEXs) or test case violations [23]. Existing verification tools typically provide Value Change Dump (VCD) [25] traces that can represent a simulation trace or a CEX to a formal property. The VCD format is a standard compact textual representation for storing signal value changes per clock cycle that can be visualized as a waveform or processed by further tools. Analyzing *security property violations* with existing functional verification tools requires even more effort than functional property violations, since these tools are not aware of the nature of the properties they verify. Before discussing the effort involved in analyzing such violations, we briefly discuss security verification with Information Flow Tracking (IFT) which has recently seen a surge for security verification [5], [6], [11], [12], [14], [17], [19]–[21], [26].

A. Information Flow Tracking (IFT)

Non-interference is a common security property for verifying confidentiality or integrity [15], [27]. IFT methods express this property in terms of information sources and sinks. Since such an

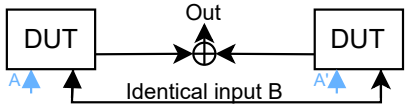

Fig. 2: Self-composition: the outputs of two instances of the DUT are compared (xor) to identify whether substituting A for A' results in differences.

Information Flow (IF) property is a hyperproperty (i.e., a property over sets of traces [28]), it cannot be expressed over a single instance of the design logic. As such, IFT methods must provide means for obtaining the necessary information for proving or disproving the hyperproperty. Two common ways for doing so are broadly known as Taint Tracking (TT) and Self-Composition (SC), which we discuss next. Both TT and SC reduce the hyperproperty to a trace property. Both methods can be used in dynamic (i.e., simulation) [17], [22], [29] or static (i.e., formal) settings [5], [6], [11], [12], [19], [20].

Taint Tracking (TT). As shown in Figure 1, TT instruments an RTL design with a parallel (shadow) taint logic. It adds one taint bit per design signal bit for calculating and tracking symbolic information flows through a design, without affecting the original design logic [15]–[18]. It dynamically tracks at runtime where information, introduced via some sources, propagates during interactions with the design. TT was originally proposed based on gate instrumentation [15], and later extended to RTL and cells for better scalability [17], [18], [21].

Self-Composition (SC). Self-composition, also known as miter [6] or product circuit [11], juxtaposes two identical instances of the Design Under Test (DUT). Information sources are presented with values that differ between the two instances, while all other inputs are driven equally on both sides. IFs manifest as differing values of the two instances of a state or output, as illustrated in Figure 2.

Information can propagate via explicit paths, like in the 'and' gate in Figure 1, or via implicit paths. For example, the condition in a conditional assignment implicitly influences the result of the assigned signal. Similarly, signals that influence the update time of a signal (e.g., an enable signal of a flip-flop) implicitly influence a sink via a timing path [5], [26]. While TT and SC are implemented differently and often used in different contexts, they verify the same underlying property: the existence of information flows. We argue that it should be possible to analyze their violation traces in a unified manner.

Research question 1. How can we unify taint tracking and self-composition under a common abstraction?

We answer this question in Section III.

B. Manual information flow trace analysis

Adding taint logic or self-composing a design inevitably leads to an increased number of signals that verification engineers needs to analyze. Figure 3 shows an excerpt of violation traces when verifying an IF property on the Ibex CPU (more information in Section VI-C). The verified property asserts that no taint shall flow to the signal 'PC', the program counter. The left side depicts an excerpt of a taint tracking trace obtained as CEX to the property, which fails in the 4th clock cycle. Taint propagated from 'ready_t' to 'iaddr_t', via some intermediate taint logic ('_5_t', '_9_t'), to 'PC_t'. The right trace shows a self-composition trace of the same example.

Manual backtracking. Due to the lack of generic techniques for debugging IFT traces, verification engineers currently resort to functional wave form analysis tools for understanding *where*, *when*, and *why* information could flow. These tools are able to show the signals

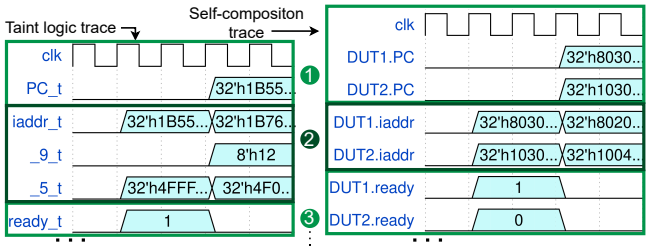

Fig. 3: **Waveform excerpts.** Left: A taint logic trace, showing taint signals (suffix _t), where a bit being one means that the corresponding design signal bit is tainted. We show the **manual** backtracking steps to determine the first three signals on the path out of, often, thousands of signals. ❶ Signal PC_t (taint signal of PC) is tainted by ❷ $iaddr\_t$ (taint signal of instruction address), via some intermediate taint logic signals (_5_t, _9_t) and 'ready"s taint signal ❸ $ready\_t$. Right: Self-composition trace for the same example. Information propagation is represented as propagation of differences between the signals of the two DUT versions. E.g., DUT1.ready differing from DUT2.ready corresponds to $ready\_t$ being 1.

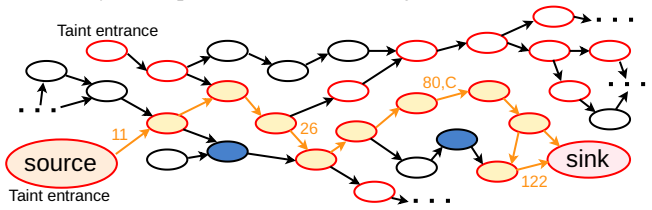

Fig. 4: A manually created graph based on a trace with multiple taint entrances and taint paths (nodes with red border). Pathfinder generates a TG that extracts a temporal path (yellow nodes) between a chosen source and sink and annotates it with (possibly multiple) clock cycles of IF occurrences, and a 'C' for implicit paths. Control flow signals, controlling the existence of the path, are shown in blue.

in the fan-in that contributed to the violation. However, being unaware of the semantics of taint logic, the tools show only the taint signal 'PC_t' as cause of the violation in the example of Figure 3. This information alone is insufficient for understanding the root cause behind a CEX or test case violation. While it is possible to extract further context information from the waveform, this manual process is extremely cumbersome: verification engineers need to manually backtrack the taint logic, which not only doubles the number of signals in the design by inserting the shadow signals but also inserts the shown additional intermediate tracking signals. Analyzing IF traces obtained from SC is also cumbersome as the two copies of each signal need to be inspected. The number of backtracking steps depends on the number of signals that can be influenced by an information source, which can be in the order of several thousands in complex designs and long traces.

Path isolation. Figure 4 shows how verification engineers could manually create a graph with nodes being HDL signals and edges being a potential influence between them for a given CEX trace. It highlights the challenges of answering *where*, *when*, and *why* information propagated. If source and sink signals are known, the difficulty lies in extracting the specific path(s) between them, i.e. finding the yellow filled nodes out of all red-bordered ones. If either the sink or source is unknown, the difficulty lies in finding where the information flew from or where it reached, i.e. finding the sources or sinks respectively. In particular, outgoing paths of a source may propagate over large portions of the design without ever reaching the sink. For example, register data (source) propagates through various modules in a CPU, but only reaches the PC (sink) through some of those. While information propagates through the modules in space and time, it influences other dependent signals, where many are irrelevant for the property. There may exist multiple paths, or sources or sinks, further complicating the analysis given that even more signals may now need to be considered. Thus, verification engineers are faced with thousands of signals that need to be manually examined and sorted, and mentally or per hand mapped to the HDL design (i.e., *where*) and particular clock cycles (i.e., *when*).

While backtracking the information flow helps reveal the spatial and temporal path taken by information through modules and signals, the underlying cause of the flow—why the information propagated—often lies beyond the path itself. The untainted signals that interfere with the path, which we define as *flow control signals*, are key to understanding the root cause of the violation. These signals represent specific results of the conditions that are able to block the information flow, hinting directly to the design logic that needs to be modified to fix the vulnerability. Hence, we require a new representation of IFs that facilitates answering these crucial questions by providing a simple to grasp, but sufficiently detailed, view that only presents essential information.

Research question 2. How should we represent information flows to facilitate root-cause analysis?

We answer this question in Section IV and show how this representation facilitates root-cause analysis using a number of case studies covering different methods and scenarios in Section VI.

C. Open-source hardware synthesis

Yosys [24] is an open source hardware synthesis tool that parses Register Transfer Level (RTL) designs into its own RTL Intermediate Language (RTLIL). RTLIL represents a design as a directed bit-level graph of macro-cells that form RTL modules or common state and their logic functions. Design signals are represented as wires between these elements. Yosys can be extended with passes that, similar to compiler passes, can operate on and transform the RTLIL representation. For example, CellIFT [17] is implemented as a Yosys pass that instruments a design with taint logic for IFT.

III. Unifying Information Flow Properties

In this Section, we unify the information flow methods of TT and SC under a common ground.

A. Notations

TT notations. Regarding TT, we adopt the notations introduced by Tiwari et al. [15]: for a given bit a in the original design, a corresponding taint bit a^t in the shadow logic is introduced and is set to 1 if a is tainted.

TT at cell level. Consider a combinational cell C (e.g., an addition, as defined in [17]) that takes an input vector I with a taint vector of same length I^t. Equation 1 [17] defines taint propagation through C, where $\oplus$ represents an exclusive-or, and $C(I)$ and $C^t(I, I^t)$ refer to C's output bits and their corresponding taint bits respectively. j indicates a bit within a vector. Taint propagates from a non-zero taint input I^t to $C^t(I, I^t)$ if there exist two different input vectors I and $\tilde{I}$ that match on zero bits of I^t and yield different cell outputs $C(I)$ and $C(\tilde{I})$. Stateful cells delay taint propagation like their corresponding design cells [15], [17].

$$C^t(I, I^t)_j = 1 \Leftarrow \exists \tilde{I} \mid (I \oplus \tilde{I}) \wedge \overline{I^t} = 0 \text{ and } C(\tilde{I})_j = \overline{C(I)_j} \quad [17] \qquad (1)$$

The unidirectionality of the implication arrow in Equation 1 is because it does not assume precise taint propagation for two reasons. First, in practice, some cells are instrumented imprecisely to reduce the overhead [17], [18]. Second, composing cells together does not preserve the precision of the taint propagation [15].

SC notations. SC verifies the property expressed in Equation 2 for a hardware design D with disjoint input sequence sets A and B and output sequence $o = D(A, B)$. The SC approach creates a circuit that instantiates two copies of the design and compares their outputs, as illustrated in Figure 2.

$$\exists A, A' | D(A, B) \neq D(A', B) \tag{2}$$

B. Correspondence between self-composition and taint tracking

We now prove that SC traces are included in TT traces.

Single-cell correspondence. Replacing D with C in Equation 2 yields Equation 1 if interpreting $C^t(I, I^t)$ as a solution to the existence problem expressed in Equation 2, where A corresponds to the bits in I for which the corresponding taint bit in I^t is set, and B corresponds to the remaining input bits in I. The existence of an A' corresponds to the existence of an $\tilde{I}$ in Equation 1. This expresses the correspondence between TT and SC for a single cell. Additionally, if the cell is instrumented precisely, i.e., if the implication in Equation 1 is an equivalence, then the two approaches are equivalent for this cell.

Scaling to composite designs. To generalize the inclusion of SC flows in TT flows to hardware designs made of any number of cells, we proceed by induction.

Proof. We have already shown that the inclusion holds for a single cell. Let us now assume that the inclusion holds for all designs with n cells, and let us consider a design D_{n+1} with $n+1$ cells, and let us divide its inputs into two disjoint sets A and B and let us consider the existence problem of A, A' such that $D_{n+1}(A, B) \neq D_{n+1}(A', B)$.

Let us take a cell C that takes as input I the concatenation α, β such that $\alpha \subseteq A$ and $\beta \subseteq B$. α or β can be empty. For simplicity and without loss of generality, assume that C has a single output bit. Let us name D_n the design D_{n+1} without C, and let us name y the output of C. Let us name the inputs of D_n, besides y, $\tilde{A} \subseteq A$ and $\tilde{B} \subseteq B$. The single-cell inclusion of SC in TT holds for D_n consisting of one cell. If $D_n(\tilde{A}, \tilde{B}, y) \neq D_n(\tilde{A}', \tilde{B}, y)$, then the inclusion holds for D_{n+1}. If not, then by the single-cell inclusion, $C(\alpha, \beta) \neq C(\alpha', \beta)$, hence $D_{n+1}(A, B) \neq D_{n+1}(A', B)$ implies $D_n(\tilde{A}, \tilde{B}, C(\alpha, \beta)) \neq D_n(\tilde{A}', \tilde{B}, C(\alpha', \beta))$, which holds by induction. □

Conclusion. Detecting information flows with TT comes back to the existence problem expressed in Equation 2. Due to TT imprecisions, a flow detected with TT might not imply the existence of a concrete flow, but a concrete flow detected via SC implies a taint flow. This allows us to uniformly represent both flows as taint, as we will discuss in Section IV-B. Because taint symbolically represents the existence of a flow, a single trace obtained via TT carries more information than a single trace obtained via SC. However, this difference is only relevant when using non-exhaustive testing methodologies. Hence, besides the potential imprecisions in the information propagation due to the instrumentation, and shortcomings in testing methodologies, TT and SC are equivalent in terms of their IF detection capabilities and have the same objective. In practice, false positives are rare [5], [17], [18], [20], [22]. In the rest of this paper, we will adopt the point of view of taint tracking to ease the intuition and to account for potential overtainting, which does not occur with SC.

IV. Information Flow Representations

We define Temporal Information Flow Graphs (TIFGs), and show how Pathfinder uses them to generate Taint Graphs (TGs), the abstraction for highlighting *where*, *when*, and *why* information propagates between sources and sinks.

```
assign key = key_i;
always @(posedge clk)
  if (Tj_Trig)
    SHIFTreg <= key;
```

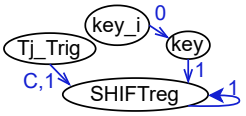

Fig. 5: **TIFG** generation from HDL inspired by the Hardware Trojan case study (Section VI-C).

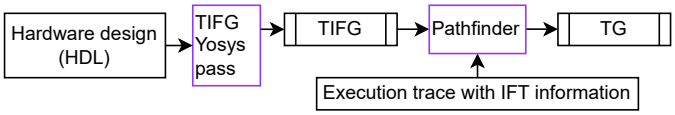

Fig. 6: **Tool flow.** Pathfinder builds Taint Graphs (TGs) from the output of our TIFG Yosys [24] pass and a trace.

A. Temporal Information Flow Graph

Definition. The *Temporal Information Flow Graph* (TIFG) of an RTL design encodes potential information flows [30], their temporality and their control conditions. We define the TIFG as a directed graph where edges can be bidirectional. Vertices (V) represent non-constant signals of input HDL design, while edges (E) represent potential information flows as can be deduced from the RTL description of the design. We define a *local potential flow* function $\pi : V^2 \rightarrow \{\text{true}, \text{false}\}$ such that $\pi(x, y)$ is true if there exists an arbitrary assignment of values in $V \setminus \{x, y\}$ that makes y a non-constant function of x. We call π local because it relies on local, cell-level dependencies of signals , and the required condition on the values in $V \setminus \{x, y\}$ might not be achievable in the design or specific trace. An edge $(x, y) \in E$ is annotated with a clock cycle delay that represents after how many clock cycles the value change in x causes a value change in y. The temporal information is essential for being able to construct a TG, discussed in Section IV-B. Additionally, edges are annotated with a flag 'C' if they represent (local) implicit flows.

Example. Figure 5 shows a TIFG obtained from the HDL snippet on the left, taken from Hu et al. [21]. The signal *key* is a function of *key_i* (in this case a direct assignment). Hence, there is an edge from *key_i* to *key*. Since the assignment represents a wire, the edge is annotated with a delay of 0. The signal *SHIFTreg* is implicitly influenced by *Tj_Trig*, hence the edge is annotated with a 'C'. There is a local potential flow π from *key* to *SHIFTreg*. A concrete flow occurs, when *Tj_Trig* equals 1. Signal *key* influences *SHIFTreg* explicitly. Since *SHIFTreg* is a state element, it keeps its value between clock edges, and information from *Tj_Trig* and *key* propagates after one clock cycle. Therefore, its incoming edges are annotated with a delay of 1.

B. Taint Graphs

A *Taint Graph* (TG) visualizes a concrete information flow as a clock-cycle-accurate propagation of taint through the design. As illustrated in Figure 6, a TG is built from 1) the TIFG extracted from the hardware design, and 2) a waveform representing a specific execution trace obtained from TT or SC. By projecting this specific trace onto the TIFG, the TG only shows IFs actually present in the trace. The TIFG provides structural information, including the annotations discussed in Section IV-A, while the trace provides clock-cycle accurate design (and in case of TT taint) signal values. Figure 7 shows that delay information in the TIFG is essential to inform Pathfinder whether taint from signal a_t in cycle 1 or 2 propagates to signal b_t in cycle 2. If the delay in the TIFG would be 0 in this example, there would be no edge in the TG for this trace.

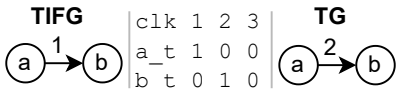

Fig. 7: Left: Design-specific TIFG, with 1 clock cycle delay. Middle: A trace showing corresponding taint signals a_t and b_t over three cycles (clk). Right: Trace-specific TG. Taint propagates at clk 2.

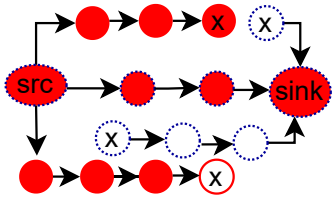

Fig. 8: **TG graphs.** A top graph (red filled circles) is intersected with a bottom graph (blue dotted borders). Their intersection (red filled nodes with blue dotted border) represents the isolated path.

Taint tracking (TT) and self-composition (SC). In TT, information flows are explicitly tracked through taint bits managed by specific instrumentation [15]–[17]. In SC, information flows are derived from the existence of differences between the two copies of design signals. To process the IF traces in a unified manner, we translate self-composition traces into precise taint, which is sound as explained in Section III-B. Taint represents symbolic information, capturing multiple traces at once, whereas a flow obtained via SC is concrete. As π is local by definition (see Section IV-A), π is not more precise than TT, itself not more precise than SC, as demonstrated in Section III. Thus, if there is an edge in an IF trace path obtained via SC, it will also exist in the TG, and thus in the TIFG. The same is true for a trace obtained via TT, independent of the precision of the TT logic. Hence, *the TG representation contains all the information that can be extracted by the underlying IFT method.*

Applications. TGs are useful for a number of applications in information flow analysis. How far information propagates from a specific source into the design can be interesting as an initial security assessment, e.g., to determine how close information gets to a secret asset or output [31], [32]. Finding information sources, e.g., microarchitectural buffers inside CPUs, from which information leaks, often reveals the origin of timing side channels [19], [33]–[35]. Determining the exact path that information took between a source and a sink is indispensable for understanding the root cause of IF property violations [5], [6], [12]. So far, verification engineers needed to perform these analysis manually by examining waveforms. To automate these tasks, Pathfinder introduces three taint graph types shown in Figure 8: the *top graph*, *bottom graph* and *path isolation*. We discuss these graphs next.

Spatio-temporal taint path isolation. First, we determine *where* and *when* information propagated. The top and bottom graph serve to automatically discover taint sinks or sources, respectively. To generate the top graph, shown in red, Pathfinder takes a taint source ('src') and extracts the taint signals from the fan-out taint logic of the source that are part of an actual taint path in a given execution trace. It can be constructed over a given number of clock cycles or until all sinks are found. Similarly, the bottom graph, shown in blue, visualizes all incoming taint paths extracted from the sink's fan-in taint logic. Thus, it is the backward tree that backtracks the information flows that affect the sink. For path isolation, which is what we use for analyzing most of the case studies discussion in Section VI-C, Pathfinder extracts all taint paths that originate from given sources and reach given sinks. It is constructed by intersecting the top and bottom graph, shown with blue dashed borders and red filling.

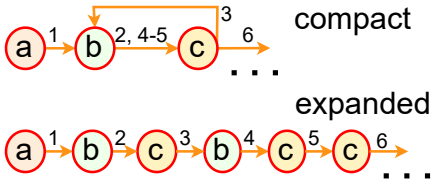

Fig. 9: **Views.** Top: Compact TG has one node per tainted signal and aggregates multiple clock cycles on an edge. Bottom: Expanded TG has one node per tainted signal per clock cycle.

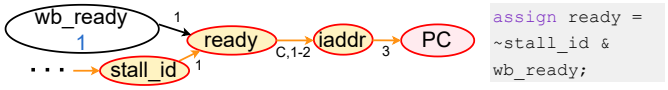

Fig. 10: **A taint graph** corresponding to the waveform in Figure 3. Signal 'stall_id' taints sig. 'ready' in cycle 1, *only if* 'wb_ready' is 1 in that cycle, i.e., if the flow control condition of 'ready' ('wb_ready' == 0) is *false*. Taint propagates via an implicit path (C) to 'iaddr' in cycles 1-2, and via an explicit path to 'PC' in cycle 3.

Timing representations. Information often traverses the same path multiple times, forming loops (e.g., register data in a CPU that is operated on and written back). Taint fades when tainted data is overwritten by non-tainted data, and signals can get tainted again at a later time. Temporal information is important to relate an IF to the design states that enable it. Pathfinder can show the temporal information propagation in two views as shown in Figure 9. First, a compact view shows each signal on the path as one node, and labels edges with all the clock cycles in which taint propagates to this signal. Second, Pathfinder provides an expanded view where each signal has a corresponding node per clock cycle in which it is tainted. While this graph is much larger than the first, it can aid in understanding loops by transforming them into a linear sequence of nodes. Within a clock cycle, taint propagates through combinational logic in a specific sequential order. In a waveform, design signals involved in combinational circuits appear tainted in the same clock cycle, making it impossible to discern a logical order within a cycle. TGs fill this gap by *depicting the sequential order in which information propagates through combinational logic within a clock cycle*, aiding in understanding the IF.

C. Flow control

Now, we establish *why* information propagates. For understanding the reason behind the existence of a taint path, verification engineers need to understand which *design states* caused this unexpected information flow. Considering again Figure 1, the output is only tainted if the *untainted* input is 1. Similarly, a multiplexer or an enable signal in a flip-flop can control the flow of information through their data ports. We refer to these signals as *flow control signals*, and the conditions they must satisfy to block a flow as *flow control conditions*.

Definition. A flow control signal is a signal that is able to block the propagation of taint through a path it interferes with.

We discuss the relationship between taint logic and flow control conditions using cells [17], without loss of generality. For each cell type, a bit-accurate condition must be satisfied for information to flow. From this condition, it is possible to derive a flow control condition per tainted bit that is able to block the information flow propagation

from that bit through the cell. In the toy example shown in Figure 10, based on a case study from [5], information propagates from signal *stall_id* to signal *PC*. The code snippet shows the usage of *stall_id*, where its negation is and'ed with the *wb_ready* signal and assigned to some *ready* signal. Taint, i.e. information, coming from *stall_id* can only propagate to *ready* if signal *wb_ready* is 1. Thus, the negation of this condition corresponds to the flow control condition fc. fc from *stall_id* to *ready* is $\neg wb\_ready$. If this fc is true, taint propagation from *stall_id* to *ready* is blocked, because *ready* is 0 regardless of the value of *stall_id*. Note that taint could also reach *ready* if *wb_ready* is tainted and 0, but in this case it is the taint of *wb_ready* that propagates. Flow control conditions can be defined for all cell types. They are defined equally when using SC, since blocking taint flows implies blocking concrete flows (see Section III-B).

Knowing the values of flow control signals thus provides valuable information for understanding *why* an information flow occurred. Therefore, Pathfinder optionally adds signals from the TIFG that interfere with the taint path, but are themselves not tainted, to the TG and shows their cycle-accurate values. We will discuss an example in Section VI-C (Figure 12 and Figure 13).

V. Implementation

In this Section, we describe the TIFG and TG implementations.

A. TIFG Implementation

We implemented the TIFG as a pass in Yosys. We build the TIFG from RTLIL after executing the Yosys optimization passes, which reduces the size of the graph while preserving its semantics. Our TIFG pass traverses all RTLIL cells and abstracts away most of the logic function of one or multiple cells between design signals by connecting inputs of cells with their outputs. We only preserve the information flow direction and timing behaviors on the taint path, which are relevant for TG generation and the further root cause analysis discussed in Section IV. We extract timing information by examining the clock signal of state-elements in RTLIL. To provide context, we augment the TIFG with local control-flow information by marking edges from multiplexer selects to multiplexer outputs, as well as from potential enable signals of state elements to their output. This annotation does not influence the construction of the TG.

Decluttering the TIFG. Designs augmented with TT logic via Yosys passes can contain many intermediate signals for connecting the logic between shadow signals. Our pass optionally removes these intermediate signals from the TIFG. We traverse the in- and outgoing connections of these signals and connect all original design signals driving them to those they drive.

B. Pathfinder

Pathfinder builds a TG from a TIFG and a VCD waveform trace that contains information flow information (taint bits or self-composition). Pathfinder is implemented in Python 3.12. It uses the vcdvcd library [36] to load the VCD waveform and Pandas dataframes [37] to store the TG. It then loads the TIFG, which is stored as comma-separated values file containing node pairs, together with the temporal and control information. Pathfinder can generate taint paths from any information flow tracking method (DIFT instrumentation and self-composition).

Translating self-composition to taints. To process the information flow trace in a unified manner, we translate self-composition traces into a taint abstraction. For each design signal, we obtain the time value pairs for both copies of the signal via vcdvcd. The VCD stores an initial value for each signal, from which we initialize a new taint signal with 1 if the initial values differ, and with 0 otherwise. Then we iterate over all time-value pairs of the two copies in time sequential order. Whenever one signal changes, we update the taint signal as follows: if after the change the two signals differ, we set the taint signal to 1, otherwise to 0. After that, TG generation continues in the same way, regardless of the underlying IFT method.

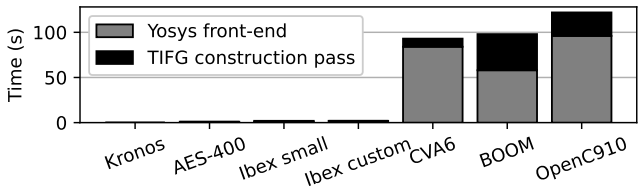

Fig. 11: TIFG construction runtime performance, split between the Yosys front-end and the dedicated TIFG construction pass.

Building the top and bottom graphs. To build the top graph, Pathfinder finds the first clock cycle in which the user-provided taint source is tainted in the VCD. Starting from the taint source, Pathfinder consults the TIFG and the VCD and iteratively connects all signals that are directly connected to the taint source without clock delay in the TIFG and are tainted in that cycle. It repeats the process for all newly added signals until all paths end at a clock boundary. Then it advances one clock cycle and adds all signals that are connected with a one clock cycle delay in the TIFG and are tainted in the next cycle. Then it repeats the process starting with directly connected signals again. Top graph construction ends either when a user-specified taint sink is found, after a user-defined number of clock cycles, or, if none of these are provided, at the last clock cycle available in the waveform trace. The bottom graph is built analogously, but in the reversed manner, starting with a taint sink. To isolate the path between source and sink, Pathfinder intersects the top and bottom graph.

Flow control. We define signals that directly interfere with the path and are not tainted in the cycles of interference as *flow control signals*. For each time step Pathfinder iterates over all tainted nodes v in the TG, consults the TIFG for finding edges with v on the right-hand side and a flow control signal that is untainted in the same or previous clock cycle (depending on the delay information). The flow control signal is annotated with the design value, representing the current value of the flow control condition within the examined trace. Pathfinder can add design control flow signals analogously.

VI. Evaluation

In this Section, we first evaluate Pathfinder's performance in terms of runtime performance (Section VI-A), then in terms of search space reduction (Section VI-B), and finally demonstrate its practicality through case studies (Section VI-C).

Evaluation setting. We obtained the performance results on a machine equipped with two AMD EPYC 7413 processors at 3.6 GHz with 128 GB of DRAM. We construct the TIFGs with 900 lines of new code lines in Yosys 0.45 and construct the TGs with 1500 lines of Python 3.12 and vcdvcd 2.3.6 [36]. We use Verilator 5.029 and Questa Sim 2022.3_1 for RTL simulation and Cadence Jasper Gold 2024.09 Formal Property Verification [38].

A. Runtime performance

The execution of Pathfinder is divided into two steps: TIFG construction (once per RTL design) and TG generation (once per trace). Hence, we evaluate the performance of these two steps separately. Figure 11 shows the runtime for the TIFG construction for the open-source designs listed in Table I, divided into the Yosys front-end and

TABLE I: **TIFG signals and connections** in original design.

Design	Nr. signals	Nr. connections
Kronos [5]	370	965
AES-400 [21]	1'293	2'232
Ibex small [5]	1'694	9'418
Ibex custom [5]	1'728	9'493
CVA6 [17]	11'759	69'065
Boom [17]	106'365	323'510

the dedicated TIFG construction pass. TIFG construction requires up to two minutes per design, dominated by Yosys front-end passes. Table II presents case studies from formal verification and simulation using SC or various TT logics as IFT methods (see Section VI-C for details). TG generation completes in a few minutes per trace.

B. Search space reduction

Pathfinder reduces the search space for verification engineers for understanding *where*, *when*, and *why* information propagates occur, while discarding irrelevant information. We measure the number of signals in the TIFG and in the TG. Without having the TG, in some cases all taint logic signals or both instances in an SC setting (e.g. Section VI-C) would need to be manually inspected. Table I summarizes the number of signals and connections in the TIFGs for several open-source designs (versions as cited) *before* instrumentation or self-composition. We note that the Boom CPU has more than 300 000 connections in the TIFG, i.e., as many potential information flows $\pi(x, y)$ in the design. Table II provides statistics of reproduced security property violation traces of several references [5], [15], [19] that we analyze in Section VI-C. **Tainted signals** shows the total number of signals that are tainted in the concrete trace. This number is already significantly smaller than the number of signals in the TIFG, i.e., the potential information flows that could occur, hence the vast majority of signals are indeed discarded. Further, not all taint that propagates through the design is necessarily caused by the violation. **Signals on path** shows all signals on all taint paths between chosen or discovered sources and sinks, which are the ones relevant for understanding the violation. Previously, verification engineers had to manually extract these from the **Tainted signals**. The reduction factor shows how much fewer signals now need to be manually inspected. The number of relevant signals is lower than the total number of tainted signals by a factor of 1.6× for bottom graphs (CVA6, AutoCC [19]), and 1.84× (Ibex small) to 81.3× (Boom) for path isolations. This is orders of magnitude less than the number of potential dependency relationships, expressed in the TIFG by Table I. For understanding *why* the IF occurred, flow control signals can be optionally added.

C. Case studies

We provide a set of case studies, as mentioned in Table II, that demonstrate the effectiveness of Pathfinder in providing context for the root cause analysis of information flows in hardware designs.

Trojan analysis [21]. Hu et al. [21] conduct a Hardware Trojan analysis based on the GLIFT instrumentation [15]. They propose to inject taint into the key passed to an Advanced Encryption Standard (AES) cryptography accelerator module (named TSC) that contains a Hardware Trojan. A formal property asserts that the key must not reach the Trojan's transmission channel, implemented as a shift register (`SHIFTReg`). We reproduce this study and obtain a CEX due to the Trojan. While the original analysis was entirely manual, we use Pathfinder to obtain the information flow paths through the design.

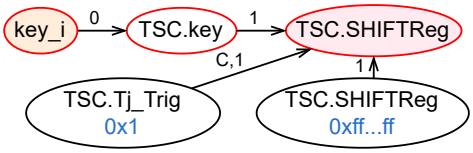

Fig. 12: Pathfinder's full output for a hardware trojan violation trace using GLIFT [21]. Information from the AES key reaches the shift register after 1 clock cycle. The reason is shown by the *flow control* signal 'TSC.Tj_Trig': Information flows if the trigger is active. The previous value of 'TSC.SHIFTReg' also interacts with the flow, but in this example, it can not block it.

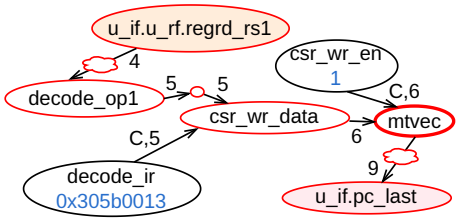

Fig. 13: TG excerpt showing an operand (u_if.u_rf.regrd_rs1) of instruction *addi* unexpectedly flowing into the *mtvec* CSR register in Kronos [5]. Clouds represent multiple signals. Signals on path: 19.

The code snippet in Figure 5 shows a simplified version of the HDL code corresponding to the graph. Figure 12 shows the corresponding TG, with untainted flow control signal `Tj_Trig` that interferes with the taint path. The TG shows that the taint propagates from the key to the shift register after one clock cycle, if `Tj_Trig` = 1, hence highlighting *where*, *when*, and *why* the taint propagates.

Microarchitectural control flow integrity (μCFI) [5]. In [5], Ceesay-Seitz et al. introduce and verify the μCFI property. Such an information flow spreads over several clock cycles across a large portion of the CPU microarchitecture, which is abundant with control signals. Figure 13 shows a reproduced CEX to the μCFI property for the Kronos RISC-V CPU. Here, μCFI attempts to prove the absence of a data flow from an operand of an *addi* instruction to the Program Counter (PC), using the CellDFT instrumentation [5]. The taint graph shows that the *addi*'s operand flows into the machine trap-vector base address register (*mtvec*) control and status (CSR) register, which is an integrity violation. In addition to the spatial and temporal information, the path highlights the condition that lets taint propagate, i.e., the inverse of the flow control condition: The decoded instruction word (`decoded_ir`)'s immediate bits correspond to the *mtvec* register address, which enables the buggy write.

Ibex traces in Table II reproduce CVE-2024-28365 [5] with CellIFT instrumentation. In 'Ibex custom', operand 1 of a remainder instruction is tainted, in 'Ibex small', operand 2 of a division instruction is tainted, and both taint the PC, causing leakage via a timing side channel. For reference, we also constructed these paths manually, which took several hours each.

New trace. We further simulate a new trace on CVA6 [39] (commit 109f9e9e), where we perform a division with tainted operand, which taints the PC (causing a timing side channel), because it is implemented with data-dependent execution latency [5].

Temporal isolation violation [19]. Orenes-Vera et al. [19] present AutoCC, which uses self-composition to track information flows. AutoCC's Algorithm 1 iteratively discovers microarchitectural buffers that leak across a temporal isolation barrier instruction, fence.t, that is intended to allow secure time multiplexing of the CPU by clearing the microarchitectural state [19]. AutoCC constrains all buffers to

TABLE II: Variety of traces we analyze using Pathfinder from different **IFT** methods and **Analysis types**. We report **Graph types**, number of **Tainted signals** in a given trace and the ones that lie on the path of interest (**Signals on path**), as well as the reduction factor (**Reduct. factor**) of signals that need to be manually inspected. We also report the TG generation time (**Gen. time**) in seconds.

Design	IFT	Analysis type	Graph type	Tainted signals	Signals on path	Reduct. factor	Clock cycles	Gen. time [s]
AES-400 [15]	TT	Simulation	Path	15	3	5.0	1	6.03
Kronos [5]	TT	Formal	Path	48	19	2.5	9	0.20
Ibex custom [5]	TT	Formal	Path	330	76	4.3	18	0.59
Ibex small [5]	TT	Formal	Path	164	89	1.8	12	0.78
CVA6 (new trace)	TT	Simulation	Path	82,920	1187	769.9	144	604.0
CVA6 [19]	SC	Formal	Bottom	900	562	1.6	76	4.47
BOOM [17]	TT	Simulation	Path	17,316	213	81.3	988	100.93

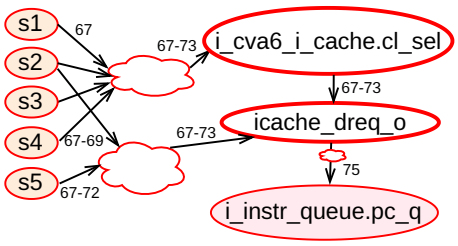

Fig. 14: TG excerpt showing the flush operation that failed to clear the instruction cache, shown by taint in '`icache_dreq_o`' in clock cycle 73, which leads to an IF to the PC ('`i_instr_queue.pc_q`') [19]. Clouds represent multiple signals. s1-s5 represent taint sources. Signals on path: 562.

have arbitrary values that differ between the two design copies before the fence.t is executed. After the fence.t operation completed, it constrains all inputs to equal their counterpart and asserts that the PCs of both copies must be equal. If they differ, information must have propagated across the barrier. Verification engineers had to manually examine the CEX waveform and backtrack the differences between the two design versions to determine the path and information source(s) that lead to the violation. Pathfinder's bottom graph simplifies this analysis significantly by automatically discovering all information sources that actually lead to an IF to a chosen sink. Figure 14 sketches a trace reproduced from AutoCC's artifact's case study 'CEX1' on the CVA6 RISC-V CPU. Besides providing spatial and temporal information about the IF, Pathfinder shows that all outgoing paths of the sources propagate through signal '`icache_dreq_o`', before reaching the PC signal '`i_instr_queue.pc_q`'. Signal '`icache_dreq_o`' exactly corresponds to the signal that had to be additionally cleared in the proposed fix.

Spectre on Boom [17]. Solt et al. [17] present CellIFT, which enables Spectre [1] vulnerability detection via taint tracking, yet understanding the specific leakage path and mechanism had been out of scope, making the implementation of a potential hardware mitigation difficult [40]. We used Pathfinder to produce a TG from CellIFT's simulation trace of the Boom CPU. We choose the memory interface as taint source, which is never accessed architecturally, and the register file as taint sink. Speculative execution caused a microarchitectural access of the memory, leaking through the Miss Status Holding Registers (MSHRs), which is visible in the TG. The number of relevant signals on the TG were only 213 out of 17,316 that had to be manually inspected in the original work.

VII. Related Work

Graphs based on HDL designs are built in many varieties for different purposes, such as coverage analysis [31], property checking [20], [41] or fuzzing [4]. Like the TIFG, variable dependency graphs [42] model relationships between signals, but edges are annotated with the number of dependencies, not the temporality. Information flow graphs model similar dependencies without edge annotations or temporality [4]. Hyperflow graphs [31] are graphs showing potential flows, annotated with information flow quantification metrics. For example, they highlight how often during a particular simulation trace an edge between two signals has been taken and propose coverage metrics such as information flow proximity. Time and Distance Metrics (TDM) were proposed for detecting vulnerabilities by estimating how far information structurally propagates into a design [32]. None of these graphs capture structural signal delays (e.g., of flip-flops), which are essential for clock-cycle accuracy in TG construction. While there exist industrial tools for security path verification [38], [43], [44], their debugging support, if at all existent, cannot be reused across different tools. Furthermore, since their underlying IFT methods are proprietary and closed source, it is not possible to implement IFT variations (e.g., CellDFT [5]), analyze their differences or compare their runtimes, making open source alternatives more attractive [6], [17]–[19]. CellDFT [5] considers paths through bit reductions and comparison cells as belonging to the control path. As this implicitly makes assumptions about the use of these logic elements, we take a more conservative approach and mark only the control paths discussed in Section V-A. Our flow control conditions differ from path conditions [45], defined for software information flow analysis, by including data path influences additionally to control, because, like in the example in Figure 1, signals on the data path can block the IF.

VIII. Conclusion

Information Flow Tracking (IFT) has become popular for hardware security analysis. Violation traces detected with IFT, however, often span hundreds if not thousands of signals. Analyzing these violations manually is time consuming and error-prone. Pathfinder is a new solution that automates the process of analyzing these IFT violations. To achieve this, Pathfinder first generates a Temporal Information Flow Graph (TIFG) from an HDL design which represents potential information flows with timing and control flow annotations. Pathfinder further takes a violation trace waveform, generated via formal verification or simulation, using either of the IFT methods, and converts it into a unified abstraction. Pathfinder then projects the unified violation trace onto the TIFG to generate a Taint Graph which contains the information needed to determine *when*, *where* and *why* information propagated through the design. We show the benefits of Pathfinder by applying it to a number of diverse scenarios spanning different IFT methods. By extracting only the relevant signals on a path, Pathfinder reduces the number of signals that need to be manually analyzed between 1.6 and 769.9 times depending on the scenario.

References

[1] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019.

[2] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *USENIX Security*, 2018.

[3] B. Chen, Y. Wang, P. Shome, C. W. Fletcher, D. Kohlbrenner, R. Paccagnella, and D. Genkin, "Gofetch: Breaking constant-time cryptographic implementations using data memory-dependent prefetchers," in *USENIX Security*, 2024.

[4] M. Rostami, S. Zeitouni, R. Kande, C. Chen, P. Mahmoody, J. Rajendran, and A. Sadeghi, "Lost and found in speculation: Hybrid speculative vulnerability detection," *DAC 2024*, 2024.

[5] K. Ceesay-Seitz, F. Solt, and K. Razavi, "μcfi: Formal verification of microarchitectural control-flow integrity," in *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security*, 2024. [Online]. Available: https://comsec-files.ethz.ch/papers/mucfi_ccs24.pdf

[6] M. R. Fadiheh, D. Stoffel, C. Barrett, S. Mitra, and W. Kunz, "Processor hardware security vulnerabilities and their detection by unique program execution checking," in *DATE*, 2019.

[7] D. Trujillo, J. Wikner, and K. Razavi, "Inception: Exposing new attack surfaces with training in transient execution," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 7303–7320.

[8] J. Wikner and K. Razavi, "Breaking the Barrier: Post-Barrier Spectre Attacks," in *S&P*, 2025.

[9] O. Oleksenko, C. Fetzer, B. Köpf, and M. Silberstein, "Revizor: Testing black-box cpus against speculation contracts," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 226–239.

[10] F. Solt, K. Ceesay-Seitz, and K. Razavi, "Cascade: CPU Fuzzing via Intricate Program Generation," in *USENIX Security 2024*, 2024.

[11] S. Dinesh, M. Parthasarathy, and C. W. Fletcher, "Conjunct: Learning inductive invariants to prove unbounded instruction safety against microarchitectural timing attacks," in *IEEE Symposium on Security and Privacy (SP)*, 2024.

[12] Z. Wang, G. Mohr, K. von Gleissenthall, J. Reineke, and M. Guarnieri, "Specification and verification of side-channel security for open-source processors via leakage contracts," in *ACM SIGSAC CCS 2023*, 2023.

[13] Q. Tan, Y. Yang, T. Bourgeat, S. Malik, and M. Yan, "RTL verification for secure speculation using contract shadow logic," in *ASPLOS*, 2025.

[14] F. Restuccia, A. Meza, and R. Kastner, "Aker: A design and verification framework for safe and secure soc access control," in *ICCAD 2021*, 2021.

[15] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood, "Complete information flow tracking from the gates up," in *ASPLOS*, 2009.

[16] A. Ardeshiricham, W. Hu, J. Marxen, and R. Kastner, "Register transfer level information flow tracking for provably secure hardware design," in *DATE*, 2017.

[17] F. Solt, B. Gras, and K. Razavi, "Cellift: Leveraging cells for scalable and precise dynamic information flow tracking in rtl," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2549–2566.

[18] F. Solt and K. Razavi, "Hybridift: Scalable memory-aware dynamic information flow tracking for hardware," *ICCAD*, 2024.

[19] M. Orenes-Vera, H. Yun, N. Wistoff, G. Heiser, L. Benini, D. Wentzlaff, and M. Martonosi, "Autocc: Automatic discovery of covert channels in time-shared hardware," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023, pp. 871–885.

[20] Y. Hsiao, N. Nikoleris, A. Khyzha, D. Mulligan, G. Petri, C. W. Fletcher, and C. Trippel, "RTL2MuPATH: Multi-upath synthesis with applications to hardware security verification," in *MICRO*, 2024.

[21] W. Hu, A. Ardeshiricham, M. S. Gobulukoglu, X. Wang, and R. Kastner, "Property specific information flow analysis for hardware security verification," in *ACM ICCAD 2018*, 2018.

[22] J. Oberg, W. Hu, A. Irturk, M. Tiwari, T. Sherwood, and R. Kastner, "Information flow isolation in i2c and usb," in *Proceedings of the 48th Design Automation Conference*, 2011, pp. 254–259.

[23] Siemens AG. The 2022 Wilson Research Group Functional Verification Study. Accessed: 2024-Nov-18. [Online]. Available: https://blogs.sw.siemens.com/verificationhorizons/2022/12/12/part-8-the-2022-wilson-research-group-functional-verification-study/

[24] C. Wolf, J. Glaser, and J. Kepler, "Yosys-a free Verilog synthesis suite," in *Austrochip*, 2013.

[25] IEEE, "Ieee standard for systemverilog–unified hardware design, specification, and verification language," *IEEE Std 1800-2023 (Revision of IEEE Std 1800-2017)*, 2024.

[26] J. Oberg, S. Meiklejohn, T. Sherwood, and R. Kastner, "Leveraging gate-level properties to identify hardware timing channels," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 33, no. 9, pp. 1288–1301, 2014.

[27] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez, "Spectector: Principled detection of speculative information flows," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1–19.

[28] N. Coenen, R. Dachselt, B. Finkbeiner, H. Frenkel, C. Hahn, T. Horak, N. Metzger, and J. Siber, "Explaining hyperproperty violations," in *CAV 2022*, 2022.

[29] P. Borkar, C. Chen, M. Rostami, N. Singh, R. Kande, A.-R. Sadeghi, C. Rebeiro, and J. Rajendran, "WhisperFuzz: White-box fuzzing for detecting and locating timing vulnerabilities in processors," in *USENIX Security*, 2024.

[30] W. Hu, J. Oberg, A. Irturk, M. Tiwari, T. Sherwood, D. Mu, and R. Kastner, "Theoretical fundamentals of gate level information flow tracking," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, no. 8, pp. 1128–1140, 2011.

[31] A. Meza and R. Kastner, "Information flow coverage metrics for hardware security verification," *https://arxiv.org/abs/2304.08263*, 2023.

[32] A. Ayalasomayajula, H. Li, H. Al Shaikh, S. K. Saha, and F. Farahmandi, "Tdm: Time and distance metric for quantifying information leakage vulnerabilities in socs," in *2024 IEEE 42nd International Conference on Computer Design (ICCD)*, 2024, pp. 130–133.

[33] M. Ghaniyoun, K. Barber, Y. Zhang, and R. Teodorescu, "Introspectre: A pre-silicon framework for discovery and analysis of transient execution vulnerabilities," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 874–887.

[34] A. de Faveri Tron, R. Isemann, H. Ragab, C. Giuffrida, K. von Gleissenthall, and H. Bos, "Phantom trails: Practical pre-silicon discovery of transient data leaks," in *USENIX Security*, 2025.

[35] T. Kovats, F. Solt, K. Ceesay-Seitz, and K. Razavi, "Milesan: Detecting exploitable microarchitectural leakage via differential hardware-software taint tracking," in *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, 2025.

[36] vcdvcd - Python Verilog value change dump (VCD) parser library. Accessed: 2025-08-08. [Online]. Available: https://github.com/cirosantilli/vcdvcd

[37] Pandas. Accessed: 2025-08-08. [Online]. Available: https://pandas.pydata.org/

[38] Jasper RTL Apps. Accessed: 2025-08-08. [Online]. Available: https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-verification-platform.html

[39] CVA6 RISC-V CPU. Accessed: 2025-08-08. [Online]. Available: https://github.com/openhwgroup/cva6

[40] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. W. Fletcher, "Speculative taint tracking (stt) a comprehensive protection for speculatively accessed data," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 954–968.

[41] V. S. Vineesh, B. Kumar, and J. Adhaduk, "Identification of effective guidance hints for better design debugging by formal methods," in *VLSI Design and Test*, 2019.

[42] D. Pal, S. Offenberger, and S. Vasudevan, "Assertion ranking using rtl source code analysis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2020.

[43] Questa Verify Secure. Accessed: 2025-08-08. [Online]. Available: https://eda.sw.siemens.com/en-US/ic/questa/formal-verification/secure-check/

[44] Cycuity Radix Technology. Accessed: 2025-08-08. [Online]. Available: https://cycuity.com/solutions/

[45] M. Taghdiri, G. Snelting, and C. Sinz, "Information flow analysis via path condition refinement," in *Formal Aspects of Security and Trust*, 2011.