Compass: Navigating the Design Space of Taint Schemes for RTL Security Verification

Yuheng Yang*
yuhengy@mit.edu
Massachusetts Institute of Technology
Cambridge, Massachusett, USA

Qinhan Tan*
qinhant@princeton.edu
Princeton University
Princeton, New Jersey, USA

Thomas Bourgeat
thomas.bourgeat@epfl.ch
École Polytechnique Fédérale de Lausanne
Lausanne, Switzerland

Sharad Malik
sharad@princeton.edu
Princeton University
Princeton, New Jersey, USA

Mengjia Yan
mengjiay@mit.edu
Massachusetts Institute of Technology
Cambridge, Massachusett, USA

Abstract

Hardware information flow tracking (IFT) using taint analysis provides a methodology to check whether a hardware design satisfies certain security properties. Previous work has shown a broad trade-off space between precision and complexity when using different taint analysis schemes. A careful investigation of this space has led to the insight that applying different taint schemes to different components of a hardware design can improve overall efficiency.

We present Compass, a systematic framework to guide users in designing appropriate taint schemes that are as lightweight as possible while still sufficient to accomplish their security verification goals. We first establish a unified terminology to comprehensively capture existing taint schemes. We then apply counterexample-guided abstraction refinement (CEGAR) for taint refinement to iteratively improve the taint scheme. We evaluated Compass on a set of open-source RISCV processors to verify the information flow properties for speculative execution vulnerabilities, and demonstrate that Compass significantly improves both simulation speed and formal-verification scalability of taint analysis.

CCS Concepts: • **Security and privacy → Information flow control**; **Side-channel analysis and countermeasures**; • **Hardware → Model checking**.

Keywords: Hardware Information Flow Tracking; Taint Analysis; Counterexample-guided Abstraction Refinement; Speculative Execution Attacks

*Two co-first authors contributed equally to this research.

ASPLOS '26, Pittsburgh, PA, USA

ACM ISBN 979-8-4007-2359-9/2026/03
https://doi.org/10.1145/3779212.3790144

ACM Reference Format:
Yuheng Yang, Qinhan Tan, Thomas Bourgeat, Sharad Malik, and Mengjia Yan. 2026. Compass: Navigating the Design Space of Taint Schemes for RTL Security Verification. In *Proceedings of the 31st ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '26), March 22–26, 2026, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3779212.3790144

1 Introduction

Recent recurring discoveries of hardware microarchitectural attacks [26, 27, 31, 33, 34] call for efficient and comprehensive security evaluation solutions for hardware designs, including both formal verification and testing. Many of these security problems can be formulated as an *information flow property*, also referred to as *non-interference property*. For example, when verifying a side-channel mitigation mechanism, we aim to check whether a specified secret value located in the victim domain can affect any microarchitectural signals that can be observed by an attacker. If the attacker's observations do not change even with changing values of the secret, we consider the system to have achieved the desired non-interference property.

Hardware information flow tracking (IFT) using taint analysis provides a general methodology to check whether a given hardware design satisfies a specified information flow property. A taint analysis scheme assigns taint bits to circuit elements to indicate whether these elements can be influenced by the secret. It also involves a taint propagation scheme, which takes the source of taint and a sink (e.g., attacker observable signals in hardware), and propagates the taint from the source to determine whether the sink will be tainted.

1.1 Motivation

Different taint-propagation schemes such as GLIFT [46], CellIFT [39], and RTLIFT [1] make different tradeoffs between taint precision and taint-propagation complexity. Lower precision provides lower propagation complexity but may result

in larger overapproximation and thus more false taint propagation of the secret to the sink. One research challenge of using taint analysis for formal verification is balancing this precision-complexity tradeoff to find a lightweight taint scheme that is still precise enough to verify a given property.

Rather than picking one global taint scheme uniformly for all the gates in the design, a more effective approach is to use different taint schemes in different hardware modules. For example, consider an out-of-order processor consisting of various modules, including a branch predictor unit and an ROB. The designer implements a speculative execution mitigation in the processor to prevent the secret from influencing the branch predictor, and thus prevents side channel leakage of the program counter (PC). If we treat the branch predictor unit in the exact same way as other modules, we will assign one taint bit for every bit in the branch predictor and compose the corresponding taint tracking logic. This is linear in the size of the branch predictor and hence will generate significant taint overhead given the size and complexity of modern branch predictors.

When a defense scheme is correctly implemented, there should be no chance for any taint bit inside the branch predictor to become tainted. As such, an optimization opportunity emerges from this insight: we can use a single bit to conservatively track the taint status of the whole branch predictor module. This coarse-grained approach is adequate because, if the defense scheme is incorrectly implemented, that single bit for the branch predictor will be tainted, capture the leakage, and offer useful information for bug localization.

However, we cannot blindly apply this optimization to other structures in the processor, such as the reorder buffer (ROB). Different entries and fields in the ROB can have different taint statuses. Using an overly coarse-grained taint scheme will lead to false alarms. Hu et al. [21] made similar observations on cryptographic accelerators.

In summary, it is promising to mix taint schemes in a design to optimize for verification efficiency. However, the space of possible taint scheme combinations is enormous. For example, given a Rocket processor with around 193K gates, assuming we have even only 3 taint options for each gate, the total configuration space grows exponentially to 3^{193K} options. Confronted by the challenge of finding effective strategies within the vast space of taint schemes, users can feel overwhelmed, unsure of *where to begin* to explore the space of taint schemes and craft an effective taint scheme for each circuit element that balances precision with complexity.

1.2 Scope and Contributions

In this paper, we present a systematic and mechanical framework Compass, to help users navigate the vast space of taint schemes for RTL verification.

(1) Taint Terminology. We start by establishing a unified terminology to comprehensively capture existing taint schemes. There exist various taint schemes proposed by prior work [22], including GLIFT [46], CellIFT [39], and RTLIFT [1]. These taint schemes differ from each other in various aspects, such as operating on different abstraction levels or providing varied logic complexity. Our terminology defines and structures the taint space—a multi-dimensional space of taint schemes—that will be later explored by Compass.

(2) CEGAR for Hardware Taint Refinement. Compass is the first work that explores this taint space by applying the counterexample-guided abstraction refinement (CEGAR) [11] principle. An abstraction over-approximates all possible behaviors of a design and is easier to verify compared to the original design. Similarly, taint analysis over-approximates the information flow of a design.

In a CEGAR loop, the user starts with an abstraction of the implementation of a design and checks whether the abstraction satisfies the specified security property using automatic formal verification tools (typically, a model checker). If the verification tool generates a spurious counterexample (false positive), the user (or an algorithm) inspects the counterexample, comes up with a refinement of the abstraction, and repeats the above operations until the security property is proved or a valid counterexample is found.

(3) Taint Refinement Algorithm. Compass uses CEGAR to refine taint schemes. We start by using a very coarse-grained taint scheme throughout the circuit, and use a counterexample generated by a model checker to locate a place where imprecision is introduced. Locating the imprecision is done automatically using a backward information flow tracing algorithm using the values at the gates and the taint propagation logic set by the counterexample. We then let the user refine the taint scheme at this location by examining several options in a predefined order. It is as if the user is faced with a gigantic map (the circuit/netlist), and our tool works as a compass to step-by-step guide the user to derive an efficient taint scheme that is lightweight and precise enough to accomplish the verification task.

(4) Experimental Validation. We demonstrate the efficacy of Compass on a set of open-source RISCV processors in verifying information-flow properties for speculative execution vulnerabilities. Compass generates lightweight taint schemes, improving both simulation speed and formal-verification scalability.

First, Compass finds lightweight taint schemes with significantly reduced taint logic (gates and register bits) overhead compared to the state-of-the-art taint scheme, CellIFT [39], and reduces simulation overhead from 351% to 205%. Second, our framework can improve model checking scalability across all verification tasks. For example, when verifying Sodor [10], the refined taint scheme helps reduce the time taken to produce an unbounded proof from hours to just 5.2 minutes (including the time spent on taint refinement).

When verifying the Rocket [2] core, Compass finds a taint scheme that improves the cycle bound in bounded model checking from 41 (by using the baseline taint scheme for 7 days) to 159 (by using Compass for 25.3 hours).

Furthermore, we analyze the taint scheme generated by Compass for the target security property, showing how the taint scheme is tailored in a property-specific manner and achieves sufficient precision with less overhead than prior property-agnostic approaches.

Compass is open-sourced at https://github.com/MATCHA-MIT/Compass.

2 Background

2.1 Information Flow Property

Information flow properties describe whether a signal can affect another signal in software programs or hardware systems. They are often used to model security-related properties, such as confidentiality [43], where secrets should not leak to a public attacker-observable location, and integrity, where a secret value cannot be overwritten by an attacker.

Formally, an information flow property can be expressed as a non-interference property [12] on a transition system. Consider a transition system $\mathbb{M} = (I, O, S, T)$ where I and O are vectors of input and output variables within domains $\mathcal{I}$ and $\mathcal{O}$, respectively. S is a vector of state variables in domain $\mathcal{S}$ and a transition function on these states is defined by $T : (\mathcal{S} \times \mathcal{I} \rightarrow \mathcal{S} \times \mathcal{O})$. Let π denote a trace of a transition system and $\pi(x)$ denote the sequence of values of variable x ($\pi(x)[i]$ denotes the value of x at step i). $l(\pi)$ denotes the length of a trace. We define the valid set of traces of executing $\mathbb{M}$ from the initial state S_0 as:

$$\textbf{Trace}(\mathbb{M}) = \Big\{\pi \;\Big|\; \pi(S)[0] = S_0 \;\wedge \big(\forall\, i \in [0, l(\pi)),\; T\big(\pi(S)[i], \pi(I)[i]\big) = \big(\pi(S)[i{+}1], \pi(O)[i]\big)\big)\Big\}$$

A non-interference property specifies a source signal that is part of the inputs ($src \in I$) and a sink signal that is part of the outputs ($sink \in O$).[1] It states that "the sink variables cannot be interfered with by the source variables" as follows.

$$\texttt{NonInterference}(src, sink, \mathbb{M}) := \forall\, \pi_1, \pi_2 \in \textbf{Trace}(\mathbb{M}),\; \big(\forall\, x \in I \setminus \{src\}, \pi_1(x) = \pi_2(x)\big) \implies \pi_1(sink) = \pi_2(sink)$$

Verification of the non-interference property can be expensive as it is a hyperproperty [12], i.e., a property defined over a pair of traces and thus a naive approach will require reasoning over all possible pairs of traces. The standard way to verify non-interference is called *self-composition* [5]. Following the non-interference definition in Formula 2.1, the verification procedure creates two copies of the same transition system, configures the inputs that are not selected as the source to be equal across the two copies, and leaves the source signals in the two copies as two different free variables. It then checks whether the sink signals in the two copies are equal. However, self-composition usually suffers from scalability issues as it doubles the size of the design under verification (DUV). It becomes ever more challenging for model checking, where computation complexity usually scales exponentially with the number of bits of state.

[1] The definition of non-interference is the same with slightly modified notation for the cases when the source and sink variables are internal variables of the transition system, or they include multiple variables.

2.2 Taint Analysis

Taint analysis, also known as information flow tracking (IFT), is a method for verifying information flow properties. In taint analysis, each input, output, and state variable in a system is assigned a "taint variable" that tracks its influence on other parts of the system. Taint propagation policies define how these taint variables should be updated to reflect their information flow status. Hu et al. [22] provide a comprehensive survey on hardware taint propagation policies [22].

Using taint analysis, we can define a property to *over-approximate* the noninterference property. In taint analysis, the sink is considered not to be interfered with by the source if the taint variable associated with it is 0. Unlike self-composition, which requires comparing two traces, taint analysis enables verifying information flow using a single trace. Therefore, taint analysis offers a more efficient way to verify information flow properties.

Over-Approximation. The cost of efficiency is imprecision, as most taint schemes used in practice *over-approximate* the check results obtained from self-composition. Hu et al. [24] have shown that for a taint scheme to precisely verify the non-interference property, the taint logic would need to scale exponentially with the circuit size, which is an impractical requirement for most designs. Therefore, all taint schemes [1, 6, 23, 39, 40, 46] used today conservatively capture information flow, potentially introducing false positives, where a taint may indicate the existence of a flow that actually does not truly exist. On the positive side, all these taint schemes do not have false negatives, meaning it reliably and comprehensively captures all information flows. This property is referred to as the soundness of taint schemes.

Application of Taint Analysis. Taint analysis can be used for two different purposes: pre-silicon security analysis and runtime intrusion detection. For security analysis, taint logic is applied during the design phase and often used by model checking or simulation. The goal is to identify potential vulnerabilities and remove them before chip tape-out. Here, the taint logic is only added temporarily to assist verification or testing and will be removed after validation is complete. Taint analysis can also be synthesized with the design for runtime attack detection, raising alerts when invalid information flows are detected. Runtime detection mechanisms include DIFT [42], Execution Lease [45], and STT [54].

In this paper, we focus on evaluating our taint logic in the verification and testing scenarios, but our taint logic can be used for other purposes.

2.3 Counterexample-Guided Abstraction Refinement (CEGAR)

Abstraction is a technique often used to address the scalability issue of formal verification. An abstract state machine has a smaller state space, and thus is simpler to be verified compared to the original design state machine. However, an abstract machine over-approximates the possible behaviors of the design and thus not only covers all possible execution traces (sequences of state transitions) in the concrete machine but also introduces execution traces not possible in the concrete machine. Consequently, if a property holds on the abstract machine it holds on the concrete machine but the abstraction might lead to spurious counterexamples (false positives). When encountering spurious counterexamples, the abstract machine needs to be refined (introducing more states and removing spurious transitions in the abstract machine) until there is no spurious counterexample, i.e., either giving a real counterexample or proving the property.

Clarke et al. [11] proposed a framework, counterexample-guided abstraction refinement (CEGAR), to automate the above procedure. It can be viewed as a verification loop consisting of 3 steps:

1. *Initialization*: Design an initial abstraction for the design.
2. *Model checking*: Check the property on the abstract machine. If the property is proven or a valid counterexample is generated, exit the loop. If the counterexample is spurious, proceed to step 3 to refine the abstract machine.
3. *Refine the abstraction*: Figure out how the spurious counterexample is introduced and add more states to the abstract machine to remove spurious state transitions. Go back to step 2.

One key challenge in applying CEGAR to a verification task lies in step 3, i.e., how to use the counterexample to figure out an effective abstraction refinement solution. Although manual refinement is possible, it demands extensive domain expertise [36] and becomes especially burdensome for large systems. A naive automated approach is to exclude the specific spurious counterexample generated in step 2. However, this often results in a similar counterexample in the next iteration, as this method fails to address the underlying cause of imprecision. Moreover, it has been shown that finding the most coarse refinement is NP-hard [11].

This paper is the first work to apply CEGAR to refine taint schemes for hardware designs. Further, we demonstrate that when applying CEGAR to taint schemes, a key component of the refinement process can be automated, that is, to locate taint logic that causes the imprecision of the overall taint scheme and thus should be refined (Section 5.3).

3 Systemizing the Taint Space

We present a unified terminology to classify all the existing taint tracking schemes and potential future ones. The goal is not to present new taint schemes, but rather to provide the space to be explored by our Compass tool.

3.1 Three-Dimensional Taint Space

We describe the hardware taint tracking space as a three-dimensional space. The three dimensions are unit level, taint bit granularity, and logic complexity. We show how existing works fit into our space in Section 8 with Table 5.

Unit Level. The unit level dimension specifies at which abstraction level the taint scheme is designed. Gate level means the taint scheme is designed for the smallest logical operation in low-level netlists, such as an `AND`, `OR` gate. In contrast, macrocell (or *cell* for short) stands for the logical macrocell abstraction in high-level Hardware Description Languages (HDLs). A cell is a pre-defined operator and only contains combinational circuits. For example, in many HDLs such as SystemVerilog, one can use the "`+`" operator to instantiate a multi-bit adder cell and the "`*`" operator for a multiplier cell.

To take another step further, one can design taint schemes at the module level, which groups multiple gates and cells together to perform certain functions. A module usually represents an architectural-level abstraction, such as a branch predictor module or a memory module. SystemVerilog, Chisel, and other HDLs provide the keyword "`module`" for programmers to use to specify this unit level. Different from a cell, a module is written by an RTL designer and can include complex sequential circuits.

When designing taint schemes, we first pick the unit level, then choose options from the next two dimensions. Gate-level and cell-level taint schemes can be generated automatically. However, constructing module-level taint schemes usually requires domain knowledge, and in this paper, we consider this can only be done manually.

Taint Bit Granularity. The taint bit granularity (or "taint granularity") dimension specifies how a taint bit associates with each circuit element in the original circuit.

The most fine-grained option is to assign one taint bit for each bit (a wire or a bit in a register) in the original circuit. Alternatively, one can group multiple bits to form a word, such as a 32-bit register entry, and assign a single bit to track the taint status for every bit in this word. Specifically, we consider every bit in the word is tainted when at least one of the bits is tainted, which can potentially introduce false positives. Furthermore, an even more coarse-grained option is to assign one taint bit for a group of registers,[2] like the example of branch predictor in Section 1, where we assign

[2]We intentionally avoid the case of assigning a taint bit for a group of wires as it introduces the risk of creating combinational loops in the taint logic.

one taint bit to track the taint status for all the BTB and BHB entries. Even though registers could be grouped up arbitrarily, our framework only considers grouping registers within a module.

There exists a clear precision and complex tradeoff when choosing taint bit granularity. Coarser-grained choices reduce the number of taint bits required, but come with the cost of reduced precision.

Logic Complexity. The logic complexity dimension specifies how much dynamic information to include in taint computation. Thus, in the paper, we may also refer to this dimension as the "dynamic level" of a taint logic. We classify them into three options: 1) *Naive (no dynamic)*: do not use any dynamic values of the inputs, 2) *partially dynamic*: use the dynamic values of a subset of the inputs, and 3) *fully dynamic*: use dynamic values of all the inputs. To understand the differences, consider the taint logic for a single-bit AND gate: $O = A \cdot B$. The suffix t refers to the taint bit of the corresponding input or output.

- *Naive (No dynamic):* $Ot = At \ + \ Bt$
- *Partially dynamic:* $Ot = At \ + \ (A \cdot Bt)$
- *Fully dynamic:* $Ot = (B \cdot At) \ + \ (A \cdot Bt) \ + \ (At \cdot Bt)$

Essentially, the dynamic options recognize that if the value of some input of the AND gate is 0, then the other input cannot influence the output. Incorporating more dynamic information into the taint computation logic increases its precision but also incurs higher overhead in terms of the number of gates for the taint logic.

3.2 Composing Lower-level Taint Schemes

We typically need to compose the taint-propagation logic for lower-level units to create a taint scheme for higher-level units. Such composition can result in a large design space of taint schemes that Compass targets to explore.

Consider a 2:1 multiplexer (MUX), which uses a single-bit selector S to choose between two bits A and B. A cell-level representation of this MUX is $O = S \ ? \ A : B$. Alternatively, its gate-level representation is $O = (S \cdot A) + ((\neg S) \cdot B)$. Now, we have 2 strategies to design taint logic for it, following the three-dimensional taint space we presented:

- *Customizing taint logic at cell-level:* This strategy enables arbitrary precision-complexity trade-off. However, each design point needs to be manually explored. For example, the most precise taint logic is:

$$Ot = St \cdot \Big((A \neq B) + At + Bt\Big) + (S \ ? \ At : Bt) \tag{1}$$

- *Composing taint logic at gate-level:* A MUX is composed of 1 NOT gate, 1 OR gate, and 2 AND gates. NOT gates, with a single-bit input, only have 1 type of taint scheme, while OR and AND gates have 4 types of taint schemes, given that the dynamic value of either input bit can be used to refine their taint logic. This means a total of $1^1 \times 4^1 \times 4^2$ taint schemes[3] can be automatically composed for the MUX. For example, the most precise taint logic can be simplified to:

$$Ot = (St \cdot A) + (S \cdot At) + (St \cdot At) + (St \cdot B) + (\neg S \cdot Bt) + (St \cdot Bt) \tag{2}$$

For designs more complex than a MUX cell, such as processors, composing taint schemes from lower-level units is the only feasible option. It results in a large, in fact exponentially large, design space to explore.

Additionally, composition comes at the cost of precision, even if the most precise taint logic is used for each lower-level units. By carefully examining the two formulas above, Formula 1 is actually more precise than Formula 2 because when $A = B = 1, At = Bt = 0, St = 1$, the cell-level taint logic correctly leaves the output untainted, while the gate-level falsely taints it. Such precision loss comes from composition and has been extensively discussed in previous work [24, 46]. As the source of this imprecision comes from the correlation between signals (e.g., $S \cdot A$ and $(\neg S) \cdot B$ cannot both be 1 in our example), this imprecision is also called *correlation-based imprecision.*

Compass Targets all Imprecision Except Correlation-based. As a final remark of this section, Compass as a tool aims to remove all *local imprecision* in taint schemes (i.e., imprecision arising from low taint bit granularity or low taint logic complexity) and considers correlation-based imprecision beyond its scope. As shown in Section 5.4, when discovering correlation-based imprecision, Compass will output an alert to ask users to manually identify the source of this imprecision and customize more precise taint logic at a higher unit level (e.g., at the module level).

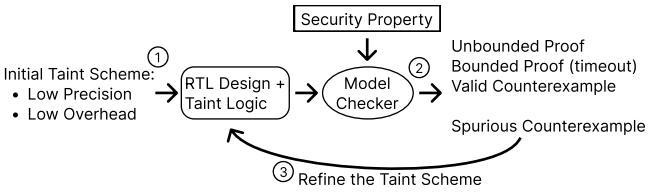

Figure 1. Compass overview.

4 Counterexample-Guided Taint Refinement

Using taint analysis for RTL verification requires making delicate trade-offs between precision and complexity. The goal is to craft an appropriate taint scheme that is as simple as possible and, meanwhile, sufficiently precise to accomplish a verification task. Compass applies counterexample-guided abstraction refinement (CEGAR, Section 2.3) to the problem of taint refinement. Figure 1 describes how we leverage the CEGAR loop to navigate the taint space, consisting of the following 3 steps:

[3]Some of these taint schemes can be logically equivalent.

Step 1: Taint Initialization. We begin the refinement process with a coarse-grained taint scheme as the initial configuration. Specifically, we assign a single taint bit for the entire module, which tracks whether any of the registers in the module hold tainted values, and apply the naive taint logic using no dynamic values. This scheme represents the most coarse-grained tracking capability and introduces the lowest taint propagation overhead. We also refer to it as a *blackboxing* taint scheme.

Step 2: Model Checking and Counterexample Validation. Given an RTL design with taint instrumentation, we check whether it satisfies the security property using a model checker. The model checker can have several outcomes: (1) It can generate an unbounded proof to confirm the design is secure. (2) It might fail to produce an unbounded proof within the compute budget (i.e., timeout), and hence only finish with a bounded proof for a certain number of cycles. This is still valuable since it provides some confidence in the design's security within a fixed number of cycles from the reset state. (3) It generates a counterexample, and we need to examine whether this counterexample is valid or spurious, that is, whether the sink is *correctly tainted* due to an actual information flow from the source, or *falsely tainted* due to imprecision in the taint scheme. The latter indicates a spurious counterexample and requires us to proceed to step 3 to refine the taint scheme.

Testing Falsely Tainted Signals. Given a concrete execution trace of a counterexample, we say that a signal is "falsely tainted" if it is marked as tainted, but the taint source has no influence on the signal's value. More precisely, the falsely-tainted signal's value stays the same for all possible values of the taint source within the given trace.

We can construct a model checking task to test whether a signal is falsely tainted. We first construct two copies of the original design. The public inputs are initialized with the concrete values from the counterexample. For private inputs, we initialize the first copy using the concrete values from the counterexample and initialize the second copy with symbolic values. If the model checker proves that the tainted signal under study in the two copies remains identical for k cycles (k is the length of the counterexample trace), we conclude that the signal is falsely tainted.

Note that this model checking task for determining falsely tainted signals is lightweight for two reasons: (1) Only private inputs in one copy are symbolic. Private inputs in the second copy and public inputs are concrete values from the counterexample. (2) The analysis only requires a bounded check for the number of cycles in the counterexample.

Step 3: Taint Refinement. Abstraction refinement is widely recognized as the most challenging step in the original CEGAR loop, often requiring significant domain expertise and deep familiarity with formal methods. As the main contribution of Compass, we significantly reduce the burden on verifiers when refining taint logic. A taint refinement task involves answering two key questions: (1) where to refine, and (2) how to refine.

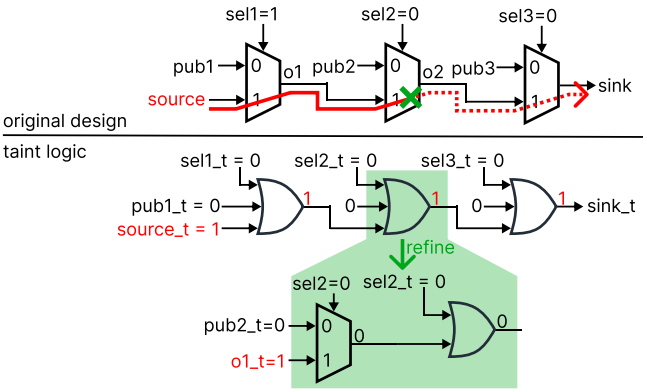

Figure 2. A circuit example (top) and its corresponding taint tracking logic (bottom). Refining the taint logic for the second multiplexer acts as a cut in the taint propagation graph.

The first question is to identify taint logic instances that are responsible for imprecision. Given that realistic designs contain thousands to millions of gates, this step poses significant challenges that urgently need automation. To solve them, we develop a fully automated algorithm to identify the taint refinement location, that is, to determine which of the taint logic instances need to be replaced with a more fine-grained alternative. The second question is to decide which new taint scheme to substitute for the existing imprecise taint instance. At each identified refinement location, we propose an ordered strategy to select taint options.

After the refinement, we go back to step 2 (forming the loop) to perform another round of model checking using the updated taint scheme.

5 Automating Taint Refinement

5.1 Introducing Cuts to Taint Propagation Graphs

We formulate the taint refinement task as a graph cut problem. As discussed in Section 2.2, taint logic works as an over-approximation of the information flow in a circuit. Conceptually, by unrolling the circuit over time, we can represent the information flow captured by the taint tracking logic as a directed acyclic graph (DAG), describing how taint propagates from sources to sinks. In this graph, some edges represent true flows, while others correspond to spurious, false flows due to the over-approximation of the taint logic. Refining taint logic works by introducing cuts to the graph to eliminate selected false-flow edges. In the following discussion, we refer to this graph as a *taint propagation graph*.

Figure 2 illustrates a concrete example. The top half of the figure depicts a circuit consisting of three multiplexers

connecting a source to a sink. The bottom half shows the corresponding taint tracking logic, where we first use a naive taint scheme that computes the output taint by ORing the taint bits of all inputs and then show its refinement below.

The red line shows the taint propagation graph. In the counterexample introducing this taint propagation, the first multiplexer selects the secret source, producing a true information flow from the source to intermediate signal o1 (solid line). In contrast, the second and third multiplexers select public values, resulting in false flows (dotted lines) from o1 to o2, and then from o2 to the sink. These false flows cause the sink to be falsely tainted.

The taint refinement task involves selecting a taint logic and replacing it with a more precise version to block the false flow. In this example, we show a refinement strategy that targets the second multiplexer's taint logic, replacing the coarse-grained OR gate with a more precise but also more complex scheme. Since the multiplexer selects a public input, the refined taint logic computes a taint value of 0 for o2, therefore effectively cutting the taint propagation graph. This refinement is denoted by the green cross symbol.

Challenges. Identifying which taint logic instance to refine presents several challenges. First, along a single taint propagation path from the source to the sink, there may be multiple potential refinement locations. For example, in Figure 2, refining the taint logic of the third multiplexer has the same effect as refining the second one. This raises the question: how should we choose among alternative refinement locations that achieve the same outcome?

Second, when multiple taint propagation paths exist between the source and the sink, it may be necessary to refine several taint logic instances together. In such cases, multiple cuts must be applied in combination to eliminate the spurious taint. This leads to another challenge: how can we identify a sufficient set of refinements to block the counterexample?

Finally, given the inherent tradeoff between precision and the overhead of taint schemes, it is natural to ask for a minimal number of cuts on the taint propagation graph. However, recall that the whole CEGAR loop may require us to eliminate multiple spurious counterexamples, each of which corresponds to a potentially different taint propagation graph. A global solution minimizing the number of refinements across all these graphs poses significant algorithmic challenges.

In this paper, we propose a practical heuristic solution, and leave finding the minimum solution as future work.

5.2 Compass's Taint Refinement Overview

The goal of refinement is to eliminate spurious counterexamples. We aim to perform refinement over one or multiple taint logic instances, each corresponding to a cut in the taint propagation graph, and eliminate the false taint at the sink. To find all these refinements, we propose an iterative approach to identify and apply one refinement per iteration.

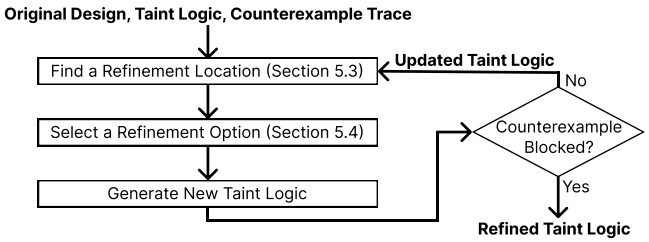

Figure 3. Overview of refining a taint scheme (Step 3 in the CEGAR loop).

Figure 3 describes the iterative procedure. Each iteration begins by locating a refinement location using an automated backtracing algorithm (Section 5.3). Once a refinement location is identified, we explore taint refinement options in a prioritized order (Section 5.4). This step is not automated in our current infrastructure and requires human intervention due to infrastructure compatibility issues — this can be resolved with further engineering effort.

At the end of each iteration, we simulate the counterexample over the netlist and the updated taint logic to check whether the refinement performed so far is sufficient to block the counterexample (i.e., the sink is correctly untainted). If the spurious taint persists, we proceed to the next iteration with the updated taint logic.

5.3 Automated Backtracing Algorithm

To identify a taint refinement location, we propose an algorithm to perform a backward traversal of the taint propagation graph, starting from the falsely tainted sink and tracing upstream toward the source. During this traversal, the algorithm addresses two key decisions at each gate: (1) whether to refine the taint logic at the current gate, or to continue tracing backwards; and (2) if the gate has multiple inputs, which input should be selected for further tracing back.

Base Algorithm. We present a basic version of the algorithm and then identify and address its two limitations. This base algorithm traces back to all falsely tainted fan-ins and identifies a refinement location closer to the source.

The pseudocode of the algorithm is in Algorithm 1 (ignoring the blue code). It takes as input a circuit netlist and a waveform generated by simulating the counterexample on the circuit. It begins at the sink signal and marks it as the current `falselyTaintedSignal`, and identifies all the fan-in signals of the current output signal (line 4). Lines 5-10 collect fan-in signals to trace back: For each fan-in, we check whether it is falsely tainted. If so, it is added to a set of traceback candidates. Next, lines 11-15 decide whether to trace back further. If the candidate set is non-empty, we randomly select one signal from this set to trace back in the next iteration (line 12). Otherwise, if no candidate exists, meaning any taints on the inputs are not false taints, then

Algorithm 1: Backward tracing algorithm

```
1  Function findRefinementLocation(netlist, waveform):
2      falselyTaintedSignal = netlist.getSink()
3      while True do
4          allFanIns = netlist.getFanIns(falselyTaintedSignal)
5          candidates = []
6          for fanIn ← allFanIns do
7              if fanIn.isFalselyTainted(waveform) &
                fanIn.isObservable(waveform) then
8                  candidates.append(fanIn)
9              end
10         end
11         if candidates.notEmpty() then
12             falselyTaintedSignal = candidates.pickOne()
13         else
14             return falselyTaintedSignal
15         end
16     end
17 end
```

the imprecision arises from the taint logic that computes the current falsely-tainted signal's taint bit (line 14).

Despite its simplicity, this basic version of the algorithm exhibits two limitations that can lead to prohibitive performance overheads. First, frequently querying whether a signal is falsely tainted can be computationally expensive. Second, randomly selecting a tainted input for backward tracing can result in a substantial number of unnecessary refinements.

Fast Test to Identify False Taint. Recall that testing for falsely tainted signals can be performed via constructing a model checking task (Section 4). For efficient refinement, we propose a *fast test* approach to approximate whether a signal is falsely tainted via simulation.

The key intuition is that, if a signal is truly tainted, it will likely be influenced by secrets in a large number of cases. Rather than exhaustively analyzing all possible secret inputs using symbolic values and model checking, we simply test a second concrete secret (by flipping all bits) via simulation to see if it can change the output value. If it is sufficient to change the output, then we are sure the output is truly tainted. Otherwise, we claim it to be falsely tainted.

As a consequence, we may claim more signals as falsely tainted and eventually make more refinements than necessary. Importantly, each of our refinements preserves the soundness of the overall taint scheme, as the refined taint logic still over-approximates information flow. These extra refinements do not affect the soundness of Compass, but might increase the complexity of the final taint scheme. Nevertheless, our empirical evaluation (Section 6) shows that the refined taint logic produced using the fast test still provides a substantial advantage over existing taint schemes.

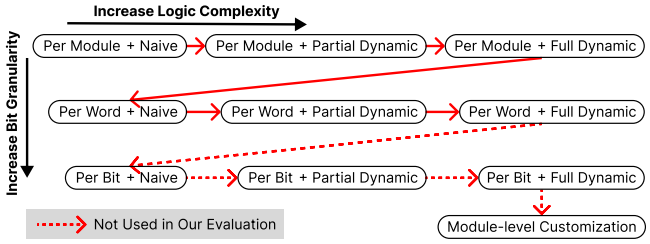

Figure 4. Priority among refinement strategies.

Limiting Tracing Back to Observable Fan-ins. The other limitation of the base algorithm is that it often traces back to signals that are *unobservable* at the sink which results in unnecessary refinements. We use the term "observable" to refer to a set of fan-ins of a given gate that, under the given concrete assignment of the gate's input signals, can influence the output of the gate, i.e., it is possible to change the gate output by changing the value of this set of inputs. Conversely, "unobservable" fan-ins do not influence the gate's output under the current assignments of the other fan-ins. A rigorous definition of observability is in Appendix A.

Consider a case where we currently trace back to a multiplexer (MUX): $O = S\ ?\ A : B$. The counterexample states $S = 1$, meaning the MUX selects A, and thus B is unobservable. When both A and B are falsely tainted, the base algorithm might randomly pick B to trace back. However, as B is not selected by this MUX, even if we manage to untaint B with refinements, it cannot help further untaint the output O. We found such unnecessary refinements very common in our experiments.

To address this limitation, we propose to only trace back to *observable* fan-ins. See the blue code in line 7 in Algorithm 1. In the MUX example above, the updated algorithm will not trace back to B and avoid unnecessary refinements.

In our implementation, we use JasperGold's "why" function [8] to collect observable fan-ins of gates.

5.4 Choosing a Refinement Strategy

After identifying a taint logic instance for refinement, the next step is to determine which new taint scheme should replace it to improve the overall precision of the taint analysis. Guided by the three-dimensional taint design space introduced in Section 3.1, we explore candidate taint schemes following a predefined order to minimize overhead. For each candidate scheme, we manually test whether substituting it for the original taint logic blocks the false taint, i.e., flips the output taint bit at the refinement location from 1 to 0. As both substitution and re-evaluation are performed *locally* at the chosen refinement location, this process is lightweight.

Figure 4 illustrates the order in which the candidate taint schemes are considered for refinement. Each node in the figure represents a taint option, and directed arrows indicate the order in which these options are explored. Neighboring

nodes to the right and below each circle represent viable next steps that increase taint precision. The ordering prioritizes options that incur lower overhead, first by increasing logic complexity, then by refining bit granularity, and finally by applying module-level customization.

We note that if none of the refinement options can yield an effective taint cut, this indicates that the imprecision is likely due to correlation-based effects, and our framework will generate an alert to notify the user. We never encountered this in our experiment (represented as dotted arrows in Figure 4). As discussed in Section 3.2, our tool is designed to address local imprecision. Addressing correlation-based imprecision needs to address a distinct set of challenges [25] and is beyond the scope of this paper.

6 Evaluation

We first present how our refined taint logic helps reduce circuit overhead and achieve faster simulation (Section 6.2), followed by showing its advantages in achieving faster model checking (Section 6.3). Then, we provide detailed statistics for the CEGAR refinement loop (Section 6.4) as well as the final taint scheme (Section 6.5).

6.1 Experiment Setup

We evaluate Compass on several open-source processors to verify information flow properties related to speculative execution vulnerabilities [31, 34]. We choose this class of security properties because they have received significant recent attention from the community. At the same time, there are still significant scalability challenges to verify these properties [30, 44, 50].

Information Flow Properties. The information flow property for secure speculation can be formulated as a software-hardware contract [19]. Such a contract states that if a program satisfies a non-interference assumption when executed on a single-cycle ISA machine, the program should also preserve non-interference when executing on a hardware implementation of the processor (the DUV). In previous work [44], this non-interference assumption is referred to as the *contract constraint check*, and the subsequent non-interference verification is called the *leakage assertion check*.

We use taint analysis to perform both checks. Since taint analysis provides a conservative check for non-interference, applying it to the contract constraint check slightly changes the semantics of the contract. Nonetheless, this is acceptable as long as the contract constraint check (which checks a program's ISA-level behavior) is no more conservative than generally used program analysis techniques such as type systems [4, 9].

There exist several variations of contracts, and we used the sandboxing contract in our evaluation. For the ProSpeCT processor, we use the property specified in its original paper [13]. Formalization of these security properties is provided in Appendix B.

Table 1. Detailed processor configurations.

`Sodor` [10]	In-order processor Configuration: 2-stage pipeline, 1-cycle DCache Code size: 9 modules, 6k lines of code
`Rocket` [2]	In-order processor Configuration: 5-stage pipeline, 2-cycle DCache Code size: 43 modules, 18k lines of code
`BOOM` `BOOM-S` [57]	Out-of-order processor Configuration: 16-entry ROB, 2-cycle DCache Code size: 105 modules, 26k lines of code
`ProSpeCT` `ProSpeCT-S` [13]	Out-of-order processor with speculative defense Configuration: 16-entry ROB Code size: 41 modules, 8k lines of code

Design under Verification. We evaluate Compass on four different processors, shown in Table 1. BOOM-S represents BOOM with speculative execution vulnerabilities patched by delaying load instructions from issuing until reaching the head of ROB. ProSpeCT-S comes from fixing two implementation bugs in ProSpeCT (detailed in Appendix C). We discovered these bugs during our experiments and confirmed them with the authors.

For model checker verification, we configure each processor to use its L1 cache as the main memory, assuming no cache misses occur. As the memory address is an attacker-observable output in the contract property, any information flow to the memory address will be detected as timing side-channel leakage. Both the data and instruction caches are configured with a capacity of 64 bytes, which is one cache line and can hold 16 RISCV instructions. We configure the last 8 bytes of the data cache to store secret values and the rest to store public values. This scaled-down setup is consistent with previous formal verification work [44, 50] to manage the scalability issue of dealing with large memory arrays. When evaluating the simulation performance of the generated taint schemes, we configure the cache size to 2 KB.

For each processor, we add shadow logic to extract commit-stage information for the contract constraint check, following the procedure described in [44].

Compass Setup. The Compass taint refinement loop requires a taint logic generation engine and a model checker. We implement the taint generation process as a FIRRTL compiler pass. FIRRTL [29] is an intermediate representation (IR) language for RTL used when compiling Chisel [3] code to Verilog. Our pass can generate the taint logic at different bit granularity and logic complexity (Section 3.1) based on user annotations in Chisel or FIRRTL.

We perform model checking using the commercial hardware verification tool JasperGold [8]. We use its `Mp`, `AM`, and

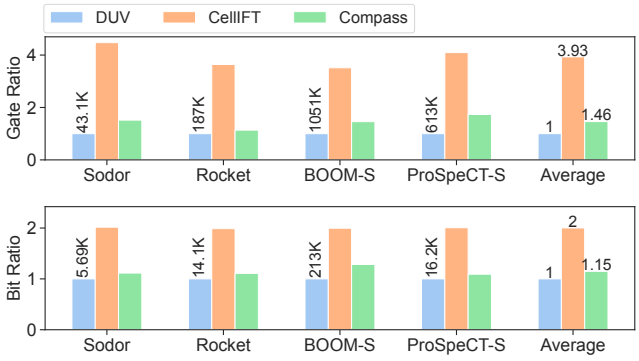

Figure 5. Number of logic gates and register bits in processors instrumented using CellIFT and Compass, normalized to the number in the original, uninstrumented processors.

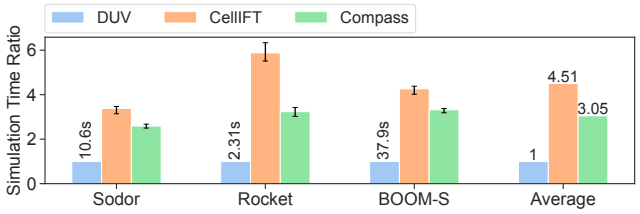

Figure 6. Simulation time of running a set of benchmarks on processors instrumented using CellIFT [39] and Compass. Numbers are first normalized to DUVs and then averaged across all benchmarks. Black vertical lines indicate the range of variation across benchmarks.

I solving engines for unbounded checks and proofs, and use the `Ht` engine for bounded checks and bug finding. The performance results were obtained on a server with an Intel Xeon 5220 processor running at 3.9 GHz. We defer the description of our simulation setup to Section 6.2.

6.2 Reduced Logic Overhead and Simulation Time

We start by comparing our scheme with CellIFT [39] in terms of taint logic overhead on gates and register bits as well as the time to simulate instrumented designs. Figure 5 and 6 present our results, both normalized to the original, uninstrumented design under verification (DUV).

Gates and Bits. CellIFT [39] uses the most precise taint logic (i.e., per-bit granularity and fully dynamic logic) for each macrocell, requiring an average of 293% logic gate overhead compared to the original design, incurring even higher overhead than self-composition (which has 100% overhead as it duplicates the DUV). Using Compass to refine taint schemes manages to reduce the gate overhead to 46% on average.

In terms of register bits introduced by the taint scheme, Compass reduces the overhead from 100% in CellIFT to an average of 15%. Most of the reduction comes from the benefits of using 1 taint bit per module and per-word taint granularity, in contrast to the 1 taint bit per data bit granularity in CellIFT.

Simulation. Figure 6 shows that the reduction of taint logic gates and bits can directly result in faster simulation.[4] The results are collected by simulating a set of RISC-V benchmarks [28, 37, 52] (median, rsort, qsort, matrix_mul, rsa) using Verilator [49]. To support these benchmarks, we increase the L1 data and instruction caches to 2KB,[5] and reduce input data sizes to fit into these caches. The first 4 elements in the input data are tainted initially.

On average, Compass reduces the simulation overhead from 351% to 205%, demonstrating its efficacy in evaluating security using simulation-based testing.

Although Compass generates smaller taint schemes, enabling faster simulation, it does not sacrifice precision in the same way as prior work [6, 23]. For the property on which we run the refinement loop, precision is fully preserved up to the cycle bound that was checked during refinement. For other properties, however, we do not provide any precision guarantees. We will show how the generated taint schemes achieve this *property-specific* feature in Section 6.5.

6.3 Improving Model Checking Coverage

We show how Compass's lighter-weight taint schemes improve model-checking coverage. We compare the verification performance of the refined taint schemes produced by Compass against self-composition (as used in Contract Shadow Logic [44]) and CellIFT [39].

Table 2 presents the model-checking results, demonstrating our improvement for both unbounded and bounded proofs. For each verification task, we report the verification time if an unbounded proof is produced, or the number of cycles successfully checked when the verification times out. A higher number of checked cycles indicates stronger confidence in the security of the design. We give Compass a 24-hour time-out limit and give other approaches 7-day limits, overcompensating for the time spent on Compass's refinement step.

First, for Sodor, Compass reduces the verification time from 1.6 hours when using CellIFT to 9.8 seconds (or to 5.2 minutes, including refinement time), indicating a significant improvement in proof efficiency. Next, for both Rocket and

[4]We did not include simulation results for ProSpeCT due to a framework incompatibility issue – ProSpeCT is developed using SpinalHDL [41]. We can only derive taint refinement annotations after converting it to FIRRTL. However, we cannot increase the memory size at the FIRRTL level to run our simulation benchmarks due to the complex, low-level memory indexing scheme.

[5]Compass maintains its advantage in reducing taint logic overhead when using a 2KB cache configuration, decreasing gate overhead from 255% to 45% and bit overhead from 100% to 15%, relative to CellIFT.

Table 2. Summary of verification time.

	self-composition [44]	CellIFT [39]	Compass t_{veri}	Compass $t_{refine} + t_{veri}$
Sodor	23h	1.6h	9.8s	5.2min
Rocket	7d (19)	7d (41)	24h (159)	25.3h (159)
BOOM-S	7d (22)	7d (26)	24h (28)	55h (28)
ProSpeCT-S	7d (29)	7d (29)	24h (29)	59h (29)

† Green entries indicate proofs being found. Other entries indicate timeouts for finding proofs (7 days for self-composition and CellIFT, 24 hours for Compass), and the number of cycles being checked is shown in the parentheses following the verification time. For Compass, we show both the verification time using the final taint logic it derives (t_{veri}) and the end-to-end time spent on refinement plus verification ($t_{refine} + t_{veri}$).

BOOM-S. Compass identifies refined taint schemes that significantly increase the number of cycles verified with much shorter times. As a highlight, our approach enables checking up to 159 cycles on Rocket, compared to only 41 cycles when using CellIFT. Lastly, although Compass does not improve the bound it checks on ProSpeCT in Table 2, we report a more detailed data point, showing Compass still improves model checking performance: To achieve a 29-cycle proof, using the taint scheme from Compass takes 15 hours, while CellIFT takes 47 hours, and self-composition takes 76 hours.

6.4 Analysis of the Refinement Process

Recall that the refinement procedure uses the model checker to generate counterexamples, each of which may require one or multiple refinements. For each counterexample, we invoke our automated backtracing algorithm to identify a refinement location, and then substitute a more precise taint scheme at that location. We repeat this process until the counterexample no longer causes the sink to be falsely tainted. Table 3 summarizes the statistics of the refinement process for different processors. We report in each column (1) the number of counterexamples ("CEX" for short) eliminated before the model checker times out, (2) the accumulated number of refinements performed over all eliminated counterexamples, and (3) the breakdown of total runtime.

The first observation is that for complex designs like BOOM-S and ProSpeCT-S, the model checking time consumes most of the runtime. This is primarily because the false counterexamples require longer execution traces compared to the ones generated for the other cores, leading to longer model checking times. Second, across all evaluated processors, simulating the counterexamples on the hardware designs also contributes notably to the runtime. This simulation time includes compiling Verilog into simulation binaries and executing the counterexample over a fixed number of cycles. Among these two phases, compilation dominates the overall simulation time, while executing the compiled binary is relatively fast, as each counterexample typically only spans 10–30 cycles.

Table 3. Statistics for the taint refinement procedure.

	# of CEX Eliminated	# of Refinement	Runtime Breakdown t_{MC}	t_{Simu}	t_{BT}	t_{Gen}
Sodor	6	12	0.8m	2.6m	1.3m	0.3m
Rocket	15	74	18m	44m	14m	3.4m
BOOM-S	14	161	21.4h	6.5h	1.8h	0.9h
ProSpeCT-S	13	39	32.2h	1.0h	0.8h	0.7h

† The total runtime is broken down into 4 columns: total model checking time (t_{MC}), simulation time (t_{Simu}, including compiling Verilog into simulation binaries), backward tracing algorithm time (t_{BT}), and taint logic generation time (t_{Gen}). We stopped the refinement when no more counterexamples could be found by model checkers in 24h.

6.5 Analysis of the Final Taint Scheme

Table 4 summarizes the final taint scheme generated for Rocket, with the design hierarchy expanded up to two levels (column 1, 2). Column 3 shows the taint bit granularity, as well as the detailed numbers of register bits in the original design (before taint instrumentation) and taint register bits added. Column 4 shows the total number of cells in the original, uninstrumented design and the fraction of those cells that are associated with partially or fully refined taint logic.[6]

Before trying to understand the result, we want to point out that our CEGAR loop can still generate unnecessary refinements, which will be discussed at the end of this subsection. Until then, we will focus on describing and summarizing necessary refinements.

Choice of Bit Granularity and Logic Complexity. First, we observe that a per-module granularity is selected if the module shares a consistent secret status (all secret or all public) across any input programs, such as I/D-TLB, Page Table Walker (PTW), and MulDiv modules, contributing to substantial savings in taint bits. Otherwise, if secret and public data can be mixed inside the module, the refinement process decides to use per-word granularity. For example, the DCache data array is partially initialized with secrets, and some pipeline registers inside the Core module can also be reached by secrets. Thus, per-word granularity is necessary for them to avoid over-approximating the taint status.

Second, we find that refined taint logic is used when the original logic is selecting among secret and public data sources, and naive taint logic is used when only public data is involved. The last column of Table 4 shows a significant amount of refined taint logic is used in the top modules of Frontend, Core, and DCache. The following representative examples illustrate the choices made by Compass and were obtained after examining the detailed refinement results (Appendix D). In the Core module, the writeback data

[6]These numbers do not exactly match Figure 5 as they are collected inside the taint logic generation compiler pass before various RTL-specific optimizations are applied and Verilog code is generated.

Table 4. The final taint scheme for Rocket.

	Modules	Bit Granularity (taintBit/origBit)	RefinedCell /OrigCell
Frontend	(Pipeline)	word (42/266)	137/393
	I-TLB (1)	module (1/2015)	75/635
	ICache	word (62/1282)	0/200
	BTB	word (724/2103)	4/2863
	Fetch Queue	word (80/630)	0/207
Core	(RF+Pipeline)	word (199/3514)	485/1175
	IBuf (1)	word (24/179)	23/406
	Breakpoint	word (0/0)	0/62
	CSR	word (260/2889)	358/1234
	ALU	word (0/0)	0/95
	MulDiv	module (1/346)	32/99
FPU (35)		mixed (150/4281)	135/3497
DCache	(Tag+Pipeline)	word (172/1481)	886/1854
	Tag Arbiter	word (0/0)	56/63
	Data Array	word (130/1029)	394/531
	Data Arbiter	word (0/0)	27/30
	D-TLB (1)	module (1/2015)	76/1154
	PMAChecker (1)	module (1/2015)	76/1154
PTW (1)		module (1/922)	0/1099

† We report refinement statistics for the five top-level modules in Rocket, and expand selected modules to a depth of two levels. For each expanded module, the first row reports statistics for logic directly contained in the module, followed by rows for its immediate submodules. If a submodule contains multiple deeper, unexpanded modules, their statistics are aggregated into a single row, with the number of child modules shown in parentheses.

is selected among memory data, ALU result, and CSR data using a MUX cell with signals such as `dmem_resp_valid`, `io.dmem.resp.bits.tag`, and `wb_ctrl.csr`. The dynamic values of these signals are used to refine the taint logic of this MUX cell. This ensures that, in execution traces where public value flows through, the taint bit will not propagate. Similar selector signals also exist inside DCache, such as valid signals (`s1_valid_not_nacked`, `io.cpu.req.valid`), pipeline commands (`s1_flush_valid`, `s2_req.cmd`), and load-store hazard (`pstore1_valid_likely`, `pstore1_addr`), which are all used to refine taint logic. On the other hand, decode logic, such as logic that computes whether an instruction is memory or ALU type and whether a memory command should be load or store, uses naive taint logic, as it only deals with public signals.

To summarize, the final taint scheme is tailored to block information flow at the *boundary between secret and public values*. This boundary is fundamentally specific to the property under study. For locations that secret values can never reach under any program, as well as locations where secret values are allowed to flow legitimately, the scheme captures information flow coarsely using per-module granularity with naive taint logic. In contrast, for specific cells where information flow depends on dynamic, run-time values from program execution, the scheme captures information flow precisely using per-word granularity and refined taint logic.

Reused Structures. We observe that when similar structures are reused in a system, they may or may not end up with similar taint schemes. For example, I/D-TLB, DCache Data Array, and ICache all contain register arrays. The arrays storing DCache data use a more precise taint scheme than others. As discussed above, this is because the DCache data array is at the boundary between secret and public values. Another example is that pipeline registers with associated update/stall logic widely exist in the Frontend, the Core, and the DCache. However, the taint status of these registers can be updated with very different taint logic depending on how they interact with secret values.

Unnecessary Refinements. Lastly, we provide examples of unnecessary refinement and explain why they occur in our CEGAR loop. In units such as CSR and MulDiv, it is unexpected to see lots of refined taint logic, as secrets should never flow to them. Such refinements help block counterexamples early in the CEGAR loop by cutting information flow closer to the sink. However, they become unnecessary once the CEGAR loop introduces refinements closer to the source.

Specifically, in counterexamples early in the CEGAR loop, input data to these units are falsely tainted, and refinements are made to avoid these falsely tainted data to flow through the modules. For example, inside CSR, `io.rw.cmd` is used to decide whether CSR will be read or write, and a false counterexample can flow input taint into CSR even if it is a read command. In this case, using the dynamic value of `io.rw.cmd` to refine the CSR write logic can help block the counterexample. Later, when the input taint of this module is fully blocked, this refinement becomes unnecessary. Similar unnecessary refinements appear in all pipelined modules, such as MulDiv (`io.req.valid`, `state`) and the frontend pipeline (`s2_valid`, `s2_btb_taken`, `s2_btb_resp_valid`). Future work can improve Compass by pruning unnecessary refinements.

7 Discussion

Soundness and Precision. Ideal taint schemes should be sound and precise to avoid false negatives and false positives when analyzing security properties.

A taint scheme is sound if it does not miss any information flow leakage. All taint schemes explored by Compass are sound because the taint logic of each hardware unit in our library is manually designed to be sound, and the soundness is preserved [46] when composing the taint logic of smaller units to create taint schemes for larger units.

As shown in Section 6.5, the precision of taint schemes generated by Compass is optimized for the security property under study. For this property, it is precise and will not generate false positives (i.e., false counterexamples) up to the number of cycles checked by the model checker (Table 2), but it may produce false positives beyond that bound. Fundamentally, Compass explores different trade-offs between precision and overhead by being *property-specific*, while existing works [1, 6, 23, 39, 40, 46] are *property-agnostic*.

Scalability of Model Checking. As described in Section 6.4, Compass can be limited by the scalability of the model checker in generating false counterexamples. Specifically, Compass targets to simplify taint logic while leaving the original design unmodified, delegating the complexity of the full design to model checkers. As future work, it is possible to incorporate orthogonal works to optimize this analysis on original designs, for example, through functionality abstraction [7, 32, 38] and invariants learning [11, 15, 35].

8 Related Work

Both CEGAR and taint analysis are widely used in formal verification for software and hardware systems. We now discuss the works that are closely relevant to us.

CEGAR for Taint Refinement. To the best of our knowledge, Lazy self-composition [53] is the only existing work that explores using CEGAR for taint refinement. It is designed for generic transition systems, focusing on software programs. When encountering a spurious counterexample, Lazy self-composition will use the model checker's internal information about the counterexample to refine the taint logic "for relevant parts of the program." Only a single refinement scheme is proposed: self-composition for the selected section of the program. In contrast, our approach (1) addresses CEGAR taint refinement for hardware and (2) provides an algorithm to identify the part(s) of the design that need taint refinement for a given false counter-example and (3) selects the simplest taint-refinement scheme that can eliminate the false counter-example.

The scalability of Lazy self-composition is unclear. They have only evaluated the approach on small C programs with up to 100 lines of code and fewer than 500 variables. Given that their implementation is not open-sourced, we are unable to test their approach on the processors in our case studies.

Existing Taint Schemes. Table 5 shows how existing hardware taint schemes fit into the three-dimensional taint space we have defined. GLIFT [46] uses taint logic that uses 1 taint bit for every data bit and fully dynamic values. Imprecise Security [23] and Arbitrary Precision [6] allow trade-offs between precision and complexity by adjusting the dynamic level. RTLIFT [1] introduces taint logic for operators in the Verilog syntax tree and supports two choices of dynamic level: fully dynamic and no dynamic. Similarly, CellIFT [39] is a macrocell-level taint scheme that allows choosing between fully dynamic and no dynamic for those cells. HybriDIFT [40] customizes taint logic for memory modules.

Table 5. Analyze existing taint schemes using our three-dimensional taint space.

	Unit Level			Bit Granularity			Logic Complexity		
	gate	cell	module	bit	word	reg group	full dyn	partial dyn	naive
GLIFT [46]	✓			✓			✓		
[23], [6]	✓			✓			✓	✓	✓
RTLIFT [1]		✓		✓			✓		✓
CellIFT [39]		✓		✓			✓		✓
HybriDIFT [40]			✓	Customized			Customized		
Compass	✓	✓	✓	✓	✓	✓	✓	✓	✓

Taint Analysis for Hardware Security Verificaiton. Hu et al. [21] leverage static analysis to determine relevant logic gates for the given security property and only generate taint logic for these gates. However, they still use the precise taint logic for all such gates.

Multiple works use taint analysis to discover timing side-channel vulnerabilities. However, they consistently use a global uniform taint scheme for the whole design, and do not explore the taint space as we do. For example, IODINE [47] and Xenon [48] check whether a hardware design leaks secrets via timing. Their taint schemes do not use any dynamic values except for the select signals of multiplexers. SecVerilog [56] and SpecVerilog [55] develop security-typed hardware description languages and include taint analysis in their type-checking rules. Given they perform taint analysis during compilation, their taint logic only uses static values. RTL2MμPath [20] uses CellIFT [39] to identify instructions that exhibit multiple microarchitecture execution paths. Isadora [14] uses RTLIFT [1] to extract information flow properties from hardware execution traces.

Formal Verification for Secure Speculation. Several existing works represent the state-of-the-art in verifying speculation contracts on RTL designs, including UPEC [16, 17, 30], LEAVE [51] and Contract Shadow Logic [44]. All these three works use self-composition to check contract properties. In our evaluation, we have shown that Compass has better performance in formal verification than self-composition.

9 Conclusion

We propose Compass, a framework to help users navigate the vast design space of taint tracking schemes and craft taint schemes that are light-weight and sufficiently precise for their verification tasks. The key insight of Compass is to apply counterexample-guided abstraction refinement to iteratively refine taint schemes. At the core, we introduce

a backtracing algorithm to automatically identify the taint logic that causes imprecision. We evaluate Compass in generating taint schemes to verify speculation contract properties for four open-source processors. The evaluation results show that Compass finds taint schemes with significantly reduced logic overhead and taint bits overhead compared to CellIFT. Moreover, these schemes also achieve better security evaluation performance in both simulation and model checking scenarios.

Acknowledgments

The authors thank Yu-Wei Fan (Princeton) the Matcha Group (MIT) for helpful feedback and discussions. This work was supported in part by NSF under grants CCF 2422052, CCF 2422053, and CNS 2046359; by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

References

[1] Armaiti Ardeshiricham, Wei Hu, Joshua Marxen, and Ryan Kastner. 2017. Register transfer level information flow tracking for provably secure hardware design. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE.

[2] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. 2016. *The Rocket Chip Generator*. Technical Report UCB/EECS-2016-17. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[3] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: constructing hardware in a Scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference (DAC)*. https://doi.org/10.1145/2228360.2228584

[4] Gilles Barthe, Sunjay Cauligi, Benjamin Grégoire, Adrien Koutsos, Kevin Liao, Tiago Oliveira, Swarn Priya, Tamara Rezk, and Peter Schwabe. 2021. High-Assurance Cryptography in the Spectre Era. In *2021 IEEE Symposium on Security and Privacy*. https://doi.org/10.1109/SP40001.2021.00046

[5] Gilles Barthe, Pedro R D'argenio, and Tamara Rezk. 2011. Secure information flow by self-composition. *Mathematical Structures in Computer Science* (2011).

[6] Andrew Becker, Wei Hu, Yu Tai, Philip Brisk, Ryan Kastner, and Paolo Ienne. 2017. Arbitrary Precision and Complexity Tradeoffs for Gate-Level Information Flow Tracking. In *Proceedings of the 54th Annual Design Automation Conference (DAC)*. https://doi.org/10.1145/3061639.3062203

[7] Jerry R. Burch and David L. Dill. 1994. Automatic verification of Pipelined Microprocessor Control. In *Proceedings of the 6th International Conference on Computer Aided Verification*. Springer-Verlag.

[8] Cadence. [Accessed 23-06-2024]. Jasper Security Path Verification App. https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform/security-path-verification-app.html.

[9] Luwei Cai, Fu Song, and Taolue Chen. 2024. Towards Efficient Verification of Constant-Time Cryptographic Implementations. *Proceedings of the ACM on Software Engineering* (2024).

[10] Christopher Celio and Et al. 2024. riscv-sodor Github Repository. https://github.com/ucb-bar/riscv-sodor. [Accessed 17-09-2024].

[11] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2000. Counterexample-guided abstraction refinement. In *Computer Aided Verification: 12th International Conference*.

[12] Michael R Clarkson and Fred B Schneider. 2010. Hyperproperties. *Journal of Computer Security* (2010).

[13] Lesly-Ann Daniel, Marton Bognar, Job Noorman, Sébastien Bardin, Tamara Rezk, and Frank Piessens. 2023. ProSpeCT: Provably Secure Speculation for the Constant-Time Policy. In *32nd USENIX Security Symposium*.

[14] Calvin Deutschbein, Andres Meza, Francesco Restuccia, Ryan Kastner, and Cynthia Sturton. 2021. Isadora: Automated information flow property generation for hardware designs. In *Proceedings of the 5th Workshop on Attacks and Solutions in Hardware Security*.

[15] Sushant Dinesh, Yongye Zhu, and Christopher W. Fletcher. 2025. H-Houdini: Scalable Invariant Learning. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery. https://doi.org/10.1145/3669940.3707263

[16] Mohammad Rahmani Fadiheh, Johannes Müller, Raik Brinkmann, Subhasish Mitra, Dominik Stoffel, and Wolfgang Kunz. 2020. A formal approach for detecting vulnerabilities to transient execution attacks in out-of-order processors. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[17] Mohammad Rahmani Fadiheh, Alex Wezel, Johannes Müller, Jörg Bormann, Sayak Ray, Jason M Fung, Subhasish Mitra, Dominik Stoffel, and Wolfgang Kunz. 2022. An exhaustive approach to detecting transient execution side channels in RTL designs of processors. *IEEE Trans. Comput.* (2022).

[18] Abraham Gonzalez and Et al. 2025. Chipyard Github Repository on tag 1.11.0. https://github.com/ucb-bar/chipyard/releases/tag/1.11.0. [Accessed 16-12-2025].

[19] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. 2021. Hardware-software contracts for secure speculation. In *2021 IEEE Symposium on Security and Privacy (SP)*.

[20] Yao Hsiao, Nikos Nikoleris, Artem Khyzha, Dominic P Mulligan, Gustavo Petri, Christopher W Fletcher, and Caroline Trippel. 2024. RTL2MμPATH: Multi-μPATH Synthesis with Applications to Hardware Security Verification. In *2024 57th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[21] Wei Hu, Armaiti Ardeshiricham, Mustafa S Gobulukoglu, Xinmu Wang, and Ryan Kastner. 2018. Property specific information flow analysis for hardware security verification. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*.

[22] Wei Hu, Armaiti Ardeshiricham, and Ryan Kastner. 2021. Hardware information flow tracking. *ACM Computing Surveys (CSUR)* (2021).

[23] Wei Hu, Andrew Becker, Armita Ardeshiricham, Yu Tai, Paolo Ienne, Dejun Mu, and Ryan Kastner. 2016. Imprecise security: Quality and complexity tradeoffs for hardware information flow tracking. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. https://doi.org/10.1145/2966986.2967046

[24] Wei Hu, Jason Oberg, Ali Irturk, Mohit Tiwari, Timothy Sherwood, Dejun Mu, and Ryan Kastner. 2011. Theoretical fundamentals of gate level information flow tracking. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2011).

[25] Wei Hu, Jason Oberg, Ali Irturk, Mohit Tiwari, Timothy Sherwood, Dejun Mu, and Ryan Kastner. 2012. On the Complexity of Generating Gate Level Information Flow Tracking Logic. *IEEE Transactions on Information Forensics and Security* (2012). https://doi.org/10.1109/TIFS.2012.2189105

[26] Intel. 2018. Intel Side Channel Vulnerability L1TF. https://www.intel.com/content/www/us/en/architecture-and-technology/l1tf.html. [Accessed 23-06-2024].

[27] Intel. 2023. Indirect Branch Predictor Barrier. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/indirect-branch-predictor-barrier.html. [Accessed 14-11-2024].

[28] RISC-V International. 2025. Benchmarks for RISCV Processor. https://github.com/riscv-software-src/riscv-tests/tree/master/benchmarks. [Accessed 16-08-2025].

[29] A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson, and J. Bachrach. 2017. Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. https://doi.org/10.1109/ICCAD.2017.8203780

[30] Tobias Jauch, Alex Wezel, Mohammad R Fadiheh, Philipp Schmitz, Sayak Ray, Jason M Fung, Christopher W Fletcher, Dominik Stoffel, and Wolfgang Kunz. 2023. Secure-by-Construction Design Methodology for CPUs: Implementing Secure Speculation on the RTL. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE.

[31] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *2019 IEEE Symposium on Security and Privacy*. https://doi.org/10.1109/SP.2019.00002

[32] Shuvendu Lahiri and Randal Bryant. 2003. Deductive Verification of Advanced Out-of-Order Microprocessors. In *Computer Aided Verification*. https://doi.org/10.1007/978-3-540-45069-6_33

[33] Luyi Li, Hosein Yavarzadeh, and Dean Tullsen. 2024. Indirector:High-Precision Branch Target Injection Attacks Exploiting the Indirect Branch Predictor. In *33rd USENIX Security Symposium*.

[34] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium*. USENIX Association.

[35] Andrew Luka and Yakir Vizel. 2025. Property Directed Reachability with Extended Resolution. Springer-Verlag. https://doi.org/10.1007/978-3-031-98668-0_13

[36] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. 2016. Ivy: safety verification by interactive generalization. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/2908080.2908118

[37] Amruth Pillai. 2025. RSA Algorithm in C. https://gist.github.com/AmruthPillai/42f4fef15bd2591aeddccae03b31ab25. [Accessed 16-08-2025].

[38] Alastair Reid, Rick Chen, Anastasios Deligiannis, David Gilday, David Hoyes, Will Keen, Ashan Pathirane, Owen Shepherd, Peter Vrabel, and Ali Zaidi. 2016. End-to-end verification of processors with ISA-Formal. In *International Conference on Computer Aided Verification*. Springer. https://doi.org/10.1007/978-3-319-41540-6_3

[39] Flavien Solt, Ben Gras, and Kaveh Razavi. 2022. CellIFT: Leveraging Cells for Scalable and Precise Dynamic Information Flow Tracking in RTL. In *31st USENIX Security Symposium*.

[40] Flavien Solt and Kaveh Razavi. 2024. HybriDIFT: Scalable Memory-Aware Dynamic Information Flow Tracking for Hardware. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*.

[41] SpinalHDL. 2025. Spinal Hardware Description Language. https://spinalhdl.github.io/SpinalDoc-RTD/master/index.html. [Accessed 16-08-2025].

[42] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. 2004. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. https://doi.org/10.1145/1024393.1024404

[43] Qinhan Tan, Yonathan Fisseha, Shibo Chen, Lauren Biernacki, Jean-Baptiste Jeannin, Sharad Malik, and Todd Austin. 2023. Security Verification of Low-Trust Architectures. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[44] Qinhan Tan, Yuheng Yang, Thomas Bourgeat, Sharad Malik, and Mengjia Yan. 2025. RTL Verification for Secure Speculation Using Contract Shadow Logic. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. https://doi.org/10.1145/3669940.3707243

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

[46] Mohit Tiwari, Hassan M.G. Wassel, Bita Mazloom, Shashidhar Mysore, Frederic T. Chong, and Timothy Sherwood. 2009. Complete information flow tracking from the gates up. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. https://doi.org/10.1145/1508244.1508258

[47] Klaus v. Gleissenthall, Rami Gökhan Kıcı, Deian Stefan, and Ranjit Jhala. 2019. IODINE: Verifying Constant-Time Execution of Hardware. In *28th USENIX Security Symposium*.

[48] Klaus v. Gleissenthall, Rami Gökhan Kıcı, Deian Stefan, and Ranjit Jhala. 2021. Solver-aided constant-time hardware verification. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 429–444.

[49] Veripool. [Accessed 08-08-2028]. Verilator, the fastest Verilog/SystemVerilog simulator. https://veripool.org/verilator/.

[50] Zilong Wang, Gideon Mohr, Klaus von Gleissenthall, Jan Reineke, and Marco Guarnieri. 2023. Specification and Verification of Side-channel Security for Open-source Processors via Leakage Contracts. In *Proceedings of the 30th ACM Conference on Computer and Communications Security (CCS)*.

[51] Zilong Wang, Gideon Mohr, Klaus von Gleissenthall, Jan Reineke, and Marco Guarnieri. 2023. Specification and verification of side-channel security for open-source processors via leakage contracts. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 2128–2142.

[52] XiangShan. 2025. The Abstract Machine (AM). https://github.com/OpenXiangShan/nexus-am/tree/master/tests/cputest. [Accessed 16-08-2025].

[53] Weikun Yang, Yakir Vizel, Pramod Subramanyan, Aarti Gupta, and Sharad Malik. 2018. Lazy self-composition for security verification. In *Computer Aided Verification: 30th International Conference (CAV)*.

[54] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W. Fletcher. 2019. Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM. https://doi.org/10.1145/3352460.3358274

[55] Drew Zagieboylo, Charles Sherk, Andrew C. Myers, and G. Edward Suh. 2023. SpecVerilog: Adapting Information Flow Control for Secure Speculation. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. https://doi.org/10.1145/3576915.3623074

[56] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. 2015. A Hardware Design Language for Timing-Sensitive Information-Flow Security. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. https://doi.org/10.1145/2694344.2694372

[57] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. 2020. SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine. In

Fourth Workshop on Computer Architecture Research with RISC-V.

[58] Quan Zhou, Sixuan Dang, and Danfeng Zhang. 2024. CtChecker: A Precise, Sound and Efficient Static Analysis for Constant-Time Programming. In *38th European Conference on Object-Oriented Programming (ECOOP 2024).*

A Observable Fan-in Signals

We use the concept of observability to describe, given a gate and a waveform, which inputs can affect an output. This concept helps our backtracing algorithm traverse only part of the upstream netlist of this gate that can influence its output. The basic idea of formalizing observability can be similar to the non-interference property in Section 2.1. However, the complexity arises when multiple inputs need to be changed together to influence the output, as shown in the definitions below.

Consider a combinational logic $\mathbb{F} : \mathcal{I} \rightarrow \mathcal{O}$. We use I to denote its input variable vector (within domain $\mathcal{I}$) and o to denote its single output variable (within domain $\mathcal{O}$). We define v as a valuation of the function, i.e., given an input or output variable x, $v(x)$ provides its value. Then, given a set of inputs $A \subseteq I$, we define whether this set is observable under a concrete valuation as:

$$\texttt{observable}(A, v, \mathbb{F}) := \exists\, v',$$
$$\Big(\forall x \in I \setminus A, v'(x) = v(x)\Big) \wedge \Big(v'(o) \neq v(o)\Big)$$

This formula states that a set of inputs is observable if and only if it is possible to flip the output by changing only the values of the inputs within this set.

However, with the above definition, a trivial observable set is the set of all inputs. In fact, we are more interested in those observable sets that are minimal:

$$\textbf{MinObserableSets}(v, \mathbb{F}) = \Big\{A \;\Big|\; \texttt{observable}(A, v, \mathbb{F}) \wedge \neg\Big(\exists\, A', A' \subsetneq A \wedge \texttt{observable}(A', v, \mathbb{F})\Big)\Big\}$$

Finally, despite the concept of minimal observable sets precisely describing all different ways that inputs can affect an output, it cannot be easily incorporated into our backtracing algorithm. For example, what if an input appears in multiple sets? Should we give higher priority to tracing back on this input? For simplicity, we use a naive solution that traces back on inputs as long as they appear in any one of these minimal observable sets, with equal priority. The following definition formalizes the notion of observable fan-ins:

$$\textbf{ObserableFanIns}(v, \mathbb{F}) = \bigcup \textbf{MinObserableSets}(v, \mathbb{F})$$

A fan-in is considered *observable* if it belongs to this set. This definition is used in Section 5.3 and Algorithm 1.

B Security Properties

In this section, we give the formal definition of the security properties we verify in Section 6. We adopt the notation and definitions from [44].

Original Contract Property. Software-hardware contracts [19] aim at differentiating the information leakage at the software (architectural) and hardware (microarchitectural) levels. An architectural observation O_{ISA} includes the information that is observable on the software level, e.g., writeback data of committed instructions in the case of the sandboxing contract we verify. It can be viewed as part of the execution trace of an ISA (1-cycle) machine. A microarchitectural observation $O_{\mu Arch}$ includes the cycle-by-cycle information of microarchitectural signals in the processor that are observable through side channels, such as commit signal, memory address, etc.

For a processor, let P denote a program, M_{pub} denote the public region in memory, and M_{sec} denote the secret region. The security property that a processor must adhere to is:

$$\forall\, P, M_{pub}, M_{sec}, M'_{sec},$$
$$\textbf{if} \quad O_{ISA}(P, M_{pub}, M_{sec}) = O_{ISA}(P, M_{pub}, M'_{sec})$$
$$\textbf{then} \quad O_{\mu Arch}(P, M_{pub}, M_{sec}) = O_{\mu Arch}(P, M_{pub}, M'_{sec})$$

Note that the assumption (**if**) and assertion (**then**) are both information flow properties. The contract property states that for arbitrary P, M_{pub} and any pairs M_{sec}, M'_{sec}, if the secret is not leaked through architectural observation on an ISA machine, then it should also not be leaked through microarchitectural observation in the processor.

Contract Property with Taint. The original contract property can be rephrased using taint logic by replacing the equivalence checks between two traces in the formula with taint checks. Let M^t_{pub}, M^t_{sec} be the taint status of M_{pub}, M_{sec}, let O^t_{ISA} be the taint trace for the ISA machine that captures whether any of the observable signals is tainted at every cycle, and let $O^t_{\mu Arch}$ be the similar taint trace for the processor. The rephrased contract property is:

$$\forall\, P, M_{pub}, M_{sec},$$
$$\text{initialize } M^t_{pub} \text{ to be 0, initialize } M^t_{sec} \text{ to be 1,}$$
$$\textbf{if } O^t_{ISA} = [0, 0, \ldots 0] \quad \textbf{then } O^t_{\mu Arch} = [0, 0, \ldots 0]$$

This property requires that if O_{ISA} is not tainted, then $O_{\mu Arch}$ should also not be tainted. There are 2 differences from the original contract property:

- The assumption becomes stronger because of the conservativeness of taint logic. This means it is possible that valid attack programs are filtered out by the assumption check. However, this gap can be minimized by using the most precise version of taint (i.e., CellIFT [39] in our case) for the ISA machine. Besides, it is a common practice to use taint analysis or more conservative type systems in verifying software information flow properties such as constant-time programming [4, 9, 58].
- The assertion also becomes stronger, which may lead to spurious counterexamples. We use Compass to refine the taint logic until there are no spurious counterexamples.

ProSpeCT Property. ProSpeCT [13] defines a slightly different security property from the software-hardware contract. Its memory is statically partitioned into public and secret regions, and loaded data is classified as secret if and only if its address lies in the secret region. The ProSpeCT property can be defined as follows using taint logic:

$$\forall\, P, M_{pub}, M_{sec},$$
$$\text{hardwire } M^t_{pub} \text{ to be 0, hardwire } M^t_{sec} \text{ to be 1,}$$
$$\textbf{if } O^t_{ISA} = [0, 0, ...0] \quad \textbf{then} O^t_{\mu Arch} = [0, 0, ...0]$$

C New Bugs Discovered on ProSpeCT

When evaluating Compass on ProSpeCT, we found two bugs that break the security mechanism.

The first bug is a simple typo. When the taint status of `rs2` register of an instruction should be used as part of logic to determine whether the it should be fired, it uses the taint status of `rs1` by mistake.

The second bug arises from incorrect tracking of transient instructions in a complex scenario involving two nested branches. The inner branch is mispredicted, the outer branch is correctly predicted, and the inner branch is resolved earlier. Then, when the outer branch is resolved, all in-flight instructions are marked as non-transient, including those following the mispredicted inner branch. As a result, no mitigation is applied to these instructions, allowing transient instructions following the inner branch to bypass the security checks.

After confirming these two bugs with the ProSpeCT authors and fixing them, we obtain ProSpeCT-S, which is the secure version of ProSpeCT that we use in our evaluation (Section 6).

D Refined Signals in Rocket

Following Table 4, we now provide signals in each module whose dynamic values are used to refine taint logic. This signal list corresponds to tag 1.11.0 of the chipyard repository [18].

```
// Frontend
Top: io.cpu.req.valid,
  io.ptw.customCSRs.csrs(0).value,
  icache.io.s2_kill, icache.io.resp.valid,
  btb.io.resp.valid, fq.io.mask, s1_valid,
  s2_valid, s2_xcpt, s2_speculative,
  s2_can_speculatively_refill,
  s2_kill_speculative_tlb_refill,
  s2_tlb_resp.cacheable, s2_tlb_resp.miss,
  s2_btb_resp_valid, s2_btb_taken,
  predictBranch, predictJump, predictReturn
I-TLB: pmp.io.pmp(1).cfg.a
BTB: reset_waddr

// Core
Top: ibuf.io.inst(0).bits.replay, take_pc_wb,
  take_pc_mem_wb, id_ctrl.mem, id_ctrl.rocc,
  id_ctrl.amo, id_ctrl.fence_i, id_raddr1,
  id_amo_rl, id_reg_fence, rf_wen, rf_waddr,
  ex_ctrl.mem, ex_ctrl.rocc, ex_ctrl.sel_alu1,
  ex_ctrl.sel_alu2, ex_ctrl.rxs2, ex_reg_rs_lsb,
  ex_reg_rs_bypass, ex_pc_valid, csr.io.rw_stall,
  csr.io.interrupt, csr.io.eret, mem_ctrl.fp,
  mem_ctrl.wxd, mem_pc_valid, mem_wrong_npc,
  dmem_resp_valid, io.dmem.resp.bits.tag,
  mem_reg_valid, mem_reg_xcpt, mem_reg_flush_pipe,
  wb_ctrl.mem, wb_ctrl.rocc, wb_ctrl.csr, wb_xcpt,
  replay_wb, ctrl_stalld
IBuf: io.imem.bits.pc, io.imem.bits.btb.taken,
  io.imem.bits.btb.bridx, nBufValid, nReady
CSR: io.rw.cmd, io.rw.addr, io.pc(1),
  io.exception, csr_wen, wdata(2), reg_mstatus.v,
  reg_debug, reg_bp(0).control.dmode,
  reg_pmp.cfg.a, reg_pmp.cfg.l, reg_dcsr.step,
  insn_call, insn_break
MulDiv: io.req.valid, state

// FPU
Top: divSqrt_killed
DivSqrtRawFN_small: entering, cycleNum

// DCache
Top: io.cpu.req.valid, tl_out.d.valid,
  tl_out.d.ready, tlb_port.req.valid, s0_clk_en,
  s1_valid, s1_valid_not_nacked, s1_xcpt_valid,
  s1_flush_valid, s1_vaddr, s1_write, s2_valid,
  s2_valid_no_xcpt, s2_valid_masked,
  s2_valid_hit_pre_data_ecc_and_waw,
  s2_valid_miss, s2_valid_cached_miss, s2_req.cmd,
  s2_probe, s2_dont_nack_miscio.cpu.req.bits.cmd,
  s2_victimize, s2_uncached, s2_req.addr,
  advance_pstore1, pstore1_valid_likely,
  pstore1_addr, inWriteback, grantIsUncachedData,
  release_state, release_ack_wait,
  release_ack_addr
Tag Arbiter: io.in.valid
Data Array: io.req.valid, io.req.bits.addr,
  io.req.bits.write, io.req.bits.way_en,
  io.req.bits.wordMask
Data Arbiter: io.in.valid
D-TLB: pmp.io.pmp(1).cfg.a
PMAChecker: pmp.io.pmp(1).cfg.a
```