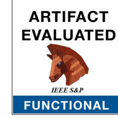

Interplay of Efficient Model Checking and Secure Processor Design: A Case Study on Secure Speculation

Tingzhen Dong*
MIT CSAIL
rogerdtz@mit.edu

Kunpeng Wang*
MIT CSAIL
kunpengw@mit.edu

Yuheng Yang
MIT CSAIL
yuhengy@mit.edu

Yu-Wei Fan
Princeton University
yf9172@princeton.edu

Qinhan Tan
Princeton University
qinhant6@gmail.com

Thomas Bourgeat
EPFL
Thomas.bourgeat@epfl.ch

Sharad Malik
Princeton University
sharad@princeton.edu

Mengjia Yan
MIT CSAIL
mengjiay@mit.edu

***Abstract*—There is increasing use of model checking to verify the security of processors, including those with speculative execution. However, model checkers face scalability challenges and are limited in the scale of processors and specific security properties they can currently handle. This research shows how exploiting domain-specific information about the design of secure processors can be used to scale model checking and make it more efficient.**

Our key observation is that proofs of non-interference properties require both information-flow invariants and functionality-based invariants, while model checkers often spend substantial effort deriving functionality invariants that do not directly contribute to the final security proof. To address this problem, we introduce Uninterpreted Functions with History (HUF), a modular abstraction for sequential modules that preserves relevant information-flow properties while abstracting away security-irrelevant functionality. We further show that the same verification insight can guide hardware design: security mechanisms become significantly easier to verify when their security depends less on hard-to-prove global functional correctness. We build a verification framework to automate the use of HUF abstractions in security verification. Practical demonstration of the verification methodology is conducted through case studies on BOOM processors for secure-speculation mitigations designed with the HUF-guided design guideline.

1. Introduction

As processor designs become increasingly complex [1] and new security vulnerabilities continue to be discovered [2], [3], [4], [5], there is an urgent need to adopt formal verification to ensure strong security guarantees. The common limitation of formal verification techniques is scalability, i.e., the ability to handle large circuits with complex module interactions. Prior work has demonstrated that automated techniques, such as model checking [6] and grammar-guided invariant synthesis [7], [8], [9], either fail to scale to out-of-order processors or are limited to handling specific types of security properties.

This paper improves model checking for verifying security properties of out-of-order processors at the RTL level. The security properties of interest are non-interference properties to check microarchitectural side channels and speculative execution vulnerabilities, including the safe instruction set property (SISP) [9] and various speculative execution contracts [10]. This improvement is based on our insight about the invariants that model checkers derive or use as part of verifying the property; invariants are essentially lemmas used in the proof. We find that to prove a non-interference property, we need two types of invariants: (i) information-flow invariants, and (ii) functionality-based invariants. The former relates to how information flow is propagated in the processors, such as which signal can be influenced by secrets under which condition. The latter relates to functionality-based behaviors, specifying conditions on the reachable states of a functionality-correct processor.

An insightful observation supported by our work is that *the security of a processor does not rely on its full functional correctness*. We find that the model checker spends a lot of time searching for functionality-based invariants that do *not* contribute to proving the final security property. This observation brings us to the question: if security does not rely on full functional correctness, what does the security depend on, and can we *abstract* the rest of the machine away and significantly improve the verification performance? As such, we aim to find an abstraction scheme that preserves the information-flow properties of a module, while abstracting away irrelevant functional details.

Manually constructing such abstractions is challenging, error-prone, and requires extensive formal-methods expertise. This raises the following research question: how to systematically and soundly abstract modules in processors to improve *security* verification efficiency?

This Paper We introduce an innovative modular abstraction for sequential modules that preserves information-

* *Both authors contributed equally to this research.*

flow behavior critical for proving non-interference properties while abstracting irrelevant functionality. This modular abstraction not only simplifies verification, but also gives us insights to formulate hardware design guideline for future processors that lead to easier verification.

The core of this abstraction scheme is a primitive we introduce called Uninterpreted Function with History (HUF). HUF extends existing Uninterpreted Function with Equality (EUF) (e.g., [11]), which was traditionally only applied to combinational circuits to alleviate the verification burden of complex arithmetic operations. We extend this abstraction to incorporate history inputs, making it stateful and applicable to sequential circuits.

On the verification side, the modular abstraction offers two appealing features. First, it is highly effective in proving non-interference properties. We show that the HUF abstraction can be applied to multiple representative sequential modules in out-of-order processors, including the reorder buffer (ROB) and rename tables. Second, ensuring that the module abstractions are sound with respect to the actual implementation is computationally easy and can be automatically handled by a model checker, as the proof obligations are fully local and mostly structural. To support the verification with the HUF abstraction scheme, we build a hierarchical verification framework called SHUFFLE, which decomposes the large monolithic proof into one property proof with significantly reduced complexity and multiple small proofs of HUF abstraction soundness.

On the design side, the verification methodology also inspires us to formulate a new hardware design guideline. The guideline is simple at a high level: design security mechanisms that achieve their goals (e.g., preventing information leakage) while reducing reliance on the processor's global functional correctness. For example, a practically useful consideration is to design defense mechanisms to not rely on the ROB's in-order commit behavior, which is a hard-to-prove global functionality property. We present small adjustments to existing defense mechanisms that substantially improve verifiability with the cost of reasonable overhead on performance and area.

We demonstrate the effectiveness of our abstraction technique through multiple case studies of secure-speculation mitigations on the BOOM processor [12]: (i) verifying the SISP property on unmodified BOOM, (ii) verifying the Sandboxing speculative contract on BOOM with a conservative scheme delaying speculative data forwarding, and (iii) verifying the contract in (ii) on an advanced mitigation with a policy of selective delay forwarding (a variant of STT [13]).

We measure the verification time of each task and analyze the invariants generated by the model checker to better understand where the proof challenges lie. First, we successfully prove the SISP property on the largest BOOM configuration, `MegaBOOM`, demonstrating a notable speedup compared to existing work [8]. Second, we show for the first time that the conservative delay-forwarding scheme is end-to-end provable using SHUFFLE, and that it also scales to `MegaBOOM`. Third, to further stress our tool, we evaluate it on the selective forwarding scheme. We find that proving the security of this defense on a reduced `SmallBOOM` configuration requires a moderate number of manual assumptions in addition to the HUF abstractions.

Contributions We make the following contributions:

- We observe that proofs of non-interference properties do not require full functional correctness, and use this insight to distinguish information-flow invariants from functionality-based invariants.
- We introduce Uninterpreted Function with History (HUF), a modular information-flow abstraction for sequential modules, and build SHUFFLE, a hierarchical verification framework that automates HUF-based security proofs.
- We propose a practical design guideline for secure speculation: security mechanisms should reduce their reliance on hard-to-prove global functionality.
- We demonstrate the verification technique and design guideline through case studies on BOOM, showing significantly improved verifiability.

Our verification framework SHUFFLE and proof-guided defense prototypes are open-sourced and available at https://github.com/MATCHA-MIT/shuffle.

2. Problem Formulation & Motivation

This section presents the formulation of the model-checking problem for verifying security properties on out-of-order processors, with some background information, before we discuss our technical contributions starting in §3.

A model checking problem is defined by a finite state machine (FSM) M and a property P, and the task is to check whether the machine satisfies the property, denoted as $M \models P$. We start by introducing the FSM model of a simplified out-of-order processor (§2.1). We then clarify the security properties we aim to verify and provide an example to show how to specify such properties on a concrete processor (§2.2). We then analyze the types of invariants required to prove these properties (§2.3), followed by a discussion of why existing model checking techniques struggle with such verification tasks (§2.4). Finally, we present a high-level overview of our verification and design approach (§2.5).

2.1. μOOM: A Toy Out-of-Order Processor

We model the hardware design using a *Finite State Machine (FSM)* defined below.

Definition 1 (Finite State Machine)**.** A finite state machine (FSM) M is a tuple $(X, \Sigma, Y, s_0, \delta, f)$, where

- X is the state space over valuations of state variables x,
- Σ is the input space over valuations of input variables σ,
- Y is the output space over valuations of outputs y,
- $s_0 \in X$ is a single initial state,
- $\delta : X \times \Sigma \rightarrow X$ is the next-state function, and
- $f : X \times \Sigma \rightarrow Y$ is the output function.

Out-of-Order Execution We provide the necessary background on out-of-order execution to frame the architectural

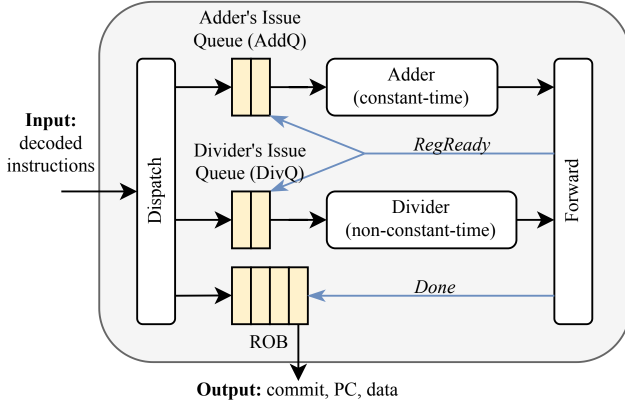

Figure 1: The simplified backend of μOOM, a toy out-of-order processor. Highlighted structures are responsible for tracking data dependency and program orders of in-flight instructions.

insights and design choices discussed later in this paper. In particular, we highlight the representative large sequential modules in out-of-order processors, the roles they play, and why they matter for verification.

Out-of-order execution allows multiple in-flight instructions to reside in the processor and begin execution as soon as their operands are ready, in contrast to being constrained by the program order. To enable out-of-order execution, two key mechanisms are required:

- *Data dependency tracking.* The goal is to track dependencies between producers and consumers of registers and eliminate false data dependencies (such as write-after-read and write-after-write hazards on the same register), enabling the processor to determine when an instruction is ready for execution. This is typically implemented through the rename logic and issue logic, often called the *rename table* and *issue queue*.
- *Program order tracking.* The goal is to maintain the original program order of instructions so that, even though they execute out of order, they can commit in order, ensuring correct exception handling and proper squashing of misspeculation. This is usually implemented via a *reorder buffer (ROB)*.

μOOM State Machine Figure 1 illustrates a sketch of a toy out-of-order processor, called μOOM. We focus on the backend, where out-of-order execution happens. The backend is modeled as a state machine composed of multiple interacting modules. The input to this backend is an instruction that has been decoded and finished reading operands from registers. It has two functional units: a constant-time Adder and a non-constant-time Divider.

The following modules work in concert to support correct out-of-order execution:

- The *dispatch* module sends each instruction to the corresponding issue queue and the ROB.
- The ROB module maintains a table to track in-flight instructions ordered by their program order. Each in-flight instruction is assigned an `robID`, which serves as an ordered, unique identifier.
- Each functional unit has an *issue queue*, denoted as `AddQ` and `DivQ`. Following Tomasulo's algorithm [14], issue queues track data dependencies among instructions and record which registers each instruction is waiting for.
- The *forward* module receives results from the two functional units, informs the ROB to mark an instruction to be committable, and notifies the issue queues that a destination register becomes ready, so as to wake up the instruction that waits for it.

Complex State Updates Both the ROB and issue queues (highlighted in Figure 1) are sequential circuits that maintain large tables to track the execution status of in-flight instructions. Stateful structures at such a complexity level, with intertwined state updates coordinated across multiple modules, are absent in in-order processors. We find that model checking algorithms encounter significant challenges in discovering invariants on these complex components.

2.2. Non-Interference Properties

We focus on verifying *information-flow security* properties of hardware processors. These properties specify how information is allowed, or disallowed, to propagate within a system and are often used to characterize microarchitectural side channels [9], [15], [16] and speculative execution vulnerabilities [6], [7], [10]. Representative examples include the safe instruction set property (SISP) [9] and various speculative execution contracts [10], including the Sandboxing and Constant-Time contracts. In this section, our discussion will primarily focus on using SISP as illustrative examples, as SISP is a relatively simpler property compared to others.

Formally, information-flow security can be expressed as *non-interference* properties [17] over a state-transition system. Since non-interference is a *hyperproperty*, as it relates a pair of execution traces, the standard way to check it with model checking is to convert it to a safety property using *self-composition* [18]. Self-composition is implemented by constructing two identical instances of the design under verification (DUV). The two instances are driven by symbolic inputs where all public inputs are constrained to be identical across instances, while secret inputs are allowed to differ. The verification then checks whether, under all possible symbolic assignments, the specified attacker-observable outputs of the two instances remain equal. This is a safety property that can be directly handled by model checkers.

Specifying SISP on μOOM To illustrate our point more clearly, we demonstrate how to specify SISP (Safe Instruction Set Property) on μOOM. SISP states that, given a safe instruction set denoted as $I_{\texttt{safe}}$, and the machine initialized with secrets residing in the register file, when assuming a program consists only of instructions from this safe set, its execution should not leak any secrets through microarchitectural side channels. μOOM supports two instructions in its ISA: `Add` and `Div`. The Adder executes in constant time,

while the Divider's latency depends on its operands. Thus, the safe instruction set is $I_{\mathtt{safe}} = \{\mathtt{Add}\}$.

Further, we need to define a leakage model specifying which processor signals are attacker-observable. In the case of μOOM, we define that an attacker can observe timing differences via the commit signal (a Boolean signal generated by ROB to indicate whether an instruction is committing at each cycle, part of the output signals in Figure 1).

To formulate the model-checking problem with self-composition, we construct a miter state machine M_{miter} consisting of two instances of μOOM and an additional state bit that accumulates the cycle-by-cycle check result of the SISP assumption. The miter machine's state space is:

$$X_{\text{miter}} := X_{\mu\text{OOM}} \times X_{\mu\text{OOM}} \times \{0, 1\}$$

with state variables $\langle x^1, x^2, \mathtt{safeinst_ok} \rangle$. The SISP property can be expressed as:

$$P_{\text{SISP}}(x) := (\mathtt{safeinst_ok} = 1) \rightarrow (x^1.\mathtt{commit} = x^2.\mathtt{commit})$$

2.3. Information-Flow and Functional Invariants

To prove a safety property, a common approach is to search for an inductive invariant. We begin with the standard definition of an inductive invariant, and then introduce our observation that, in verifying non-interference properties, these invariants naturally fall into two categories.

Definition 2 (Inductive Invariant)**.** Let $M = (X, \Sigma, Y, s_0, \delta, f)$ be an FSM, and let $P(x)$ be a safety property. A predicate $Inv(x)$ is an *inductive invariant* proving P iff it satisfies:

$$\begin{array}{ll} & Inv(s_0) = 1, \\ \forall x,\, \sigma,\, x'. & Inv(x) \,\wedge\, Tr(x, \sigma, x') \,\Rightarrow\, Inv(x'), \text{ and} \\ \forall x. & Inv(x) \,\Rightarrow\, P(x). \end{array}$$

where $Tr(x, \sigma, x') := (x' = \delta(x, \sigma))$ represents the derived transition relation with x and x' denoting the current-state and next-state variables.

An inductive invariant $Inv(x)$ is often a complex formula composed (e.g., using conjunction) of multiple invariants (or lemmas).

When proving non-interference properties, we find that the invariants involved in $Inv(x)$ can be categorized into two types: 1) information-flow invariants, and 2) functionality-based invariants.

Information-Flow Invariants An information-flow invariant specifies under which conditions a relevant signal can be influenced by a secret, or conversely, guarantees that the signal will never be affected by the secret. In the context of self-composition, such invariants usually take the form:

$$cond \rightarrow (x^1.signal = x^2.signal)$$

This states that, under condition *cond*, the signal remains equal across the two instances and thus does not depend on secret inputs. Because these invariants relate the same signal across two DUV instances, we also refer to them as *cross-instance invariants*.

Functionality-Based Invariants Although our goal is to prove security properties, these properties inevitably depend on certain aspects of the processor's functionality (but likely not full functional correctness). For example, it is often required that the dispatch logic in a processor behaves correctly; otherwise, the processor might send operations to incorrect functional units, exercising data paths that are not supposed to be activated for security reasons. We refer to such invariants as *functionality-based invariants* or *single-instance invariants*, since they relate signals within the same instance of the circuit.

Example Invariants For the μOOM example, we present several invariants that help prove SISP. Since SISP asserts that the commit signals of the two instances must always be equal, and given the complex interactions among the ROB, dispatch, and forward modules, it is critical to capture invariants about the input and output signals of these modules.

Some useful *information-flow invariants* state that a subset of the input and output signals of the issue queue for the Adder are not affected by secrets. For example, the following invariant ensures that each entry in the Add issue queue (AddQ) has the same ready bit:

$$x^1.x_{\mathtt{AddQ}}.\mathtt{table}[i].\mathtt{srcRdy} = x^2.x_{\mathtt{AddQ}}.\mathtt{table}[i].\mathtt{srcRdy}$$

On the other hand, since the Divider unit's execution time depends on its operands, and we constrain inputs to include only the Add instructions, a useful *functionality-based invariant* states that the Divider unit is never used, requiring the dispatch unit to behave correctly. Concretely, each entry in the Divider's issue queue (DivQ) satisfies:

$$x_{\mathtt{DivQ}}.\mathtt{table}[i].\mathtt{valid} = 0.$$

However, not all functionality-based invariants directly contribute to proving the final property. Even worse, they sometimes take a fairly complicated form. For example, each ROB entry in μOOM satisfies the following functionality-based invariant, which states that an in-flight instruction that has just finished its execution must have its corresponding ROB entry in a valid state and not marked as done yet:

$$\begin{aligned}(\mathtt{valid}_{\mathtt{done}} = 1) \rightarrow (&x_{\mathtt{rob}}.\mathtt{table}[\mathtt{robId}_{\mathtt{done}}].\mathtt{valid} = 1 \\ &\wedge x_{\mathtt{rob}}.\mathtt{table}[\mathtt{robId}_{\mathtt{done}}].\mathtt{done} = 0)\end{aligned}$$

2.4. Why Model Checking Struggles With Verifying Non-Interference Properties?

Model-checking algorithms aim to obtain unbounded proofs by discovering inductive invariants. Among various approaches, we focus on one widely adopted algorithm, *Property-Driven Reachability* (PDR, also known as IC3) [19], [20]. We analyze its scalability bottlenecks and find that it spends substantial effort deriving functionality-based invariants over large sequential modules, even when those invariants do not directly contribute to the final non-interference proof.

To understand the bottleneck concretely, we analyze the invariants generated by open-source implementations,

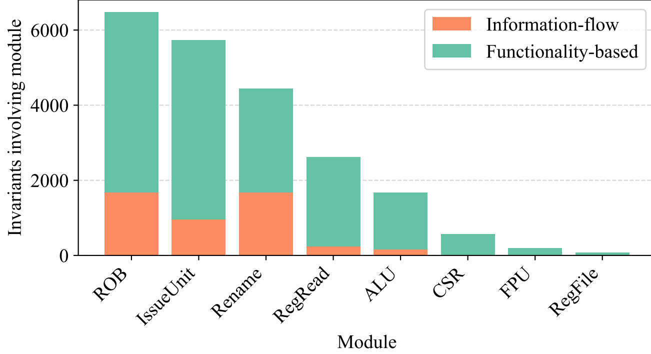

Figure 2: The distribution of candidate invariants across modules when using *rIC3* to verify SISP on `SmallBOOM`. Over 70% are functionality-based invariants. *ABC* shows a similar distribution.

rIC3 [21] and *ABC* [22], when verifying SISP on the BOOM processor [12] with the experimental configuration listed in §6. Since the off-the-shelf algorithms time out, we output and classify the intermediate candidate invariants generated by the *rIC3* model checker after running it for one hour. Figure 2 shows the invariant distribution across different modules and the breakdown between information-flow invariants and functionality-based invariants.

From the figure, we observe that the majority of invariants reference signals from the ROB, the issue module, and the rename module, all of which are complex sequential circuits consisting of large tables. Moreover, most of the invariants are functionality-based invariants that relate signals within the same instance of the design. We manually analyzed these functionality-based invariants and found that most of them do not contribute to proving the SISP property, like the example at the end of §2.3.

We believe this phenomenon arises for two main reasons. First, when supplying the self-composed state machine to the model checker, everything is blended together, and the model checker is unaware that it operates on two duplicated state machines. As such, it has no clue how to efficiently search for the information-flow invariants that are critically important for proving the final property. Second, the model checker is inherently bad at handling large sequential circuits. When encountering such structures, IC3 often falls into a "rabbit hole," repeatedly generating invariants that describe relationships among table fields.

2.5. The Interplay between Verification and Design

The analysis above reveals significant potential for speedup in model checking security properties: Since certain aspects of functionality are inessential for achieving the final security property, circuit logic related to such functionality can be abstracted away. The abstraction should satisfy two requirements:

1) The abstraction must preserve the essential information-flow properties of the module so that relevant information-flow invariants still hold on the abstracted machine;
2) The abstraction must abstract large modules with complex state updates whose functionality is irrelevant to proving the security property.

The above abstraction insight suggests that scaling security requires an *interplay* between verification and design. On the verification side, abstraction can systematically remove security-irrelevant functionality to reduce proof complexity. On the design side, reducing security's reliance on hard-to-prove global functionality leads to architectural designs that are easier to verify.

In the rest of this paper, we propose a scheme to guide users to derive *modular abstractions*, more specifically, local information-flow abstractions that apply to individual modules, ranging from combinational modules to complex sequential modules. At the core of this scheme is a new abstraction primitive, called Uninterpreted Function with History (HUF), which we formalize in §3. Next, we discuss how to apply HUF to representative modules in out-of-order processors and describe the proof framework built upon HUF in §4. Finally, we show that the idea that security can be achieved without full functional correctness extends beyond verification and allows us to formulate a verification-guided design guideline in §5.

3. Uninterpreted Function with History

We now introduce our abstraction scheme Uninterpreted Function with History (HUF). We motivate and explain at a high level how we extend existing abstractions with Uninterpreted Functions with Equality (EUF) to handle sequential modules in §3.1. We then formally define HUF and HUF-abstracted machines in §3.2 and §3.3 respectively. We present variants of HUF in §3.4, and introduce the module-level proof obligations needed to ensure the soundness of the variants in §3.5. Finally, we discuss practical considerations when implementing HUF abstraction in §3.6.

3.1. From EUF to HUF

Abstraction substitutes parts of the design with a model that over-approximates the concrete behavior. In hardware verification, a common form of abstraction is to substitute complex modules with *uninterpreted functions* (UFs), effectively hiding solver-expensive arithmetic logic [11], [23], [24], [25]. A UF is a function symbol with fixed arity and type signature whose internal behavior is left unspecified.

UF with Equality (EUF) The theory of *Uninterpreted Function with Equality (EUF)* augments first-order logic with equality, resulting in the *congruence* principle for all such symbols: if two sets of arguments are pairwise equal, then applying the same UF to both sets yields equal results.

Concretely, consider abstracting two instances of the module implementing a 64-bit adder f in a given design. We create a fresh UF symbol F that takes two 64-bit inputs (denoted as σ) and produces one 64-bit output. We can then

substitute every instance of the adder with applications of F. In this setting, the model checker treats each invocation of F as a symbolic function application. Instead of reasoning about bit-level addition, it only enforces the equality and congruence constraints defined below:

$$\forall \sigma^1, \sigma^2.\ \ \sigma^1 = \sigma^2 \Rightarrow F(\sigma^1) = F(\sigma^2)$$

UF with History (HUF) As discussed in §2.4, many modules that bottleneck processor security proofs are sequential modules (e.g., ROB and rename table). The outputs of sequential modules depend not only on current inputs, but also on the internal states that evolve over time.

We propose an extension of UF for the sequential context. The idea is to enforce equality over input histories: if the full input histories through (and including) cycle t are equal, then the outputs of two sequential modules at cycle t are also equal. This approach captures historical dependence without exposing complex internal state machines, enabling UF-style abstraction on sequential circuits.

3.2. HUF Formalization

Conceptually, *Uninterpreted Function with History* (HUF) is an abstraction scheme that allows us to abstract both the output function and complex state transition function of sequential modules. It consists of (i) a UF symbol F for the output, (ii) a 1-bit predicate that summarizes the state equivalence relation, and (iii) the *HUF constraint* as will be explained in detail below.

Without loss of generality, we focus on two instances of the same module with its FSM M, indexed by $i \in \{1, 2\}$. Let σ_t^i denote inputs to M at cycle t, x_t^i its state, and y_t^i its output. The functionality of M is characterized by an output function f and a next-state function δ:

$$y_t^i = f\big(x_t^i, \sigma_t^i\big), \qquad x_{t+1}^i = \delta\big(x_t^i, \sigma_t^i\big)$$

Our goal is to abstract the output of the module by applying EUF abstraction to f with UF F.

A per-cycle EUF abstraction states:

$$(x_t^1 = x_t^2) \wedge (\sigma_t^1 = \sigma_t^2) \Rightarrow F\big(x_t^1, \sigma_t^1\big) = F\big(x_t^2, \sigma_t^2\big) \quad (1)$$

This abstracts the combinational output logic but leaves the sequential next-state logic δ intact since it's implicitly involved in checking the state equivalence relation $(x_t^1 = x_t^2)$.

To avoid reasoning about δ, we introduce a 1-bit state variable `hist_eq` that encodes the state equivalence relation. Specifically, it requires that the two instances have identical input histories before the current cycle.

$$\texttt{hist_eq}_0 := 1;\ \texttt{hist_eq}_{t+1} := \texttt{hist_eq}_t \wedge (\sigma_t^1 = \sigma_t^2) \quad (2)$$

Intuitively, $\texttt{hist_eq}_t = 1$ implies that the two instances have identical internal states $(x_t^1 = x_t^2)$ because all input histories from cycle 0 to $t-1$ are equal. As such, we claim the following lemma:

Lemma 1 (State Equivalence with History)**.** *For all $t \geq 0$:*

$$\texttt{hist_eq}_t \Rightarrow (x_t^1 = x_t^2)$$

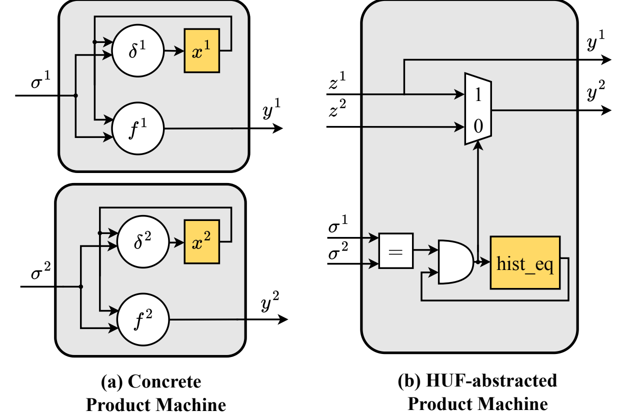

Figure 3: HUF-abstracted product machine construction. The state elements are highlighted.

The proof is included in Appendix A.1.

To constrain the output at cycle t, in addition to the state equivalence, we require the current inputs to be equal. By Lemma 1, we can safely replace the state equivalence in the premise of Equation (1) with $\texttt{hist_eq}_t$, yielding the following equation:

$$\texttt{hist_eq}_t \wedge (\sigma_t^1 = \sigma_t^2) \Rightarrow F(x_t^1, \sigma_t^1) = F(x_t^2, \sigma_t^2) \quad (3)$$

which we called the *HUF constraint*.

3.3. HUF-Abstracted Product Machine

We now describe how to utilize HUF to perform abstraction. At a high level, the abstraction replaces the large concrete state with a 1-bit summary of cross-instance history equivalence, while leaving the outputs unconstrained except when the *HUF constraint* requires them to be equal. We begin by illustrating this process in Figure 3, followed by a formal definition.

Consider a concrete product machine M (left in Figure 3) consisting of two instances, with state variables $x = (x^1, x^2)$, inputs $\sigma = (\sigma^1, \sigma^2)$, and outputs $y = (y^1, y^2)$. The state elements are highlighted and can be substantially large. We use HUF to abstract the concrete product machine M into an abstract product machine, denoted by M_{abs} (right in Figure 3).

We start by replacing the concrete state variables x with a 1-bit state variable `hist_eq`, which compactly summarizes whether the states of the two instances are equal. We then construct logic to compute `hist_eq` and the outputs y.

To compute `hist_eq`, we compare the inputs σ^1 and σ^2 at each cycle and accumulate the comparison result according to Equation (2). To leave the outputs unconstrained while enforcing equality when needed, we introduce two fresh symbolic inputs z^1 and z^2 and wire them such that the two instances either produce the same output or are allowed to differ. Specifically, if the equivalence condition holds, both outputs are driven by the same fresh input z^1, so the outputs are equal. Otherwise, the two outputs are driven by different fresh inputs, z^1 and z^2, so they may differ.

We formally define HUF-abstracted product machine.

Definition 3 (HUF-abstracted Product Machine)**.** Let $M = (X, \Sigma, Y, s_0, \delta, f)$ be a concrete product machine for two instances, with state variables $x = (x^1, x^2)$, inputs $\sigma = (\sigma^1, \sigma^2)$, and outputs $y = (y^1, y^2)$. The two instances share the same initial state s_0, next-state function δ, and output function f.

Let $\texttt{HistUpd}$ denote the next-state function of $\texttt{hist_eq}$ from Equation (2) and Z denote the domain of fresh symbolic inputs $z = (z^1, z^2)$. The HUF-abstracted product machine is the tuple

$$M_{\text{abs}} = (X', \Sigma', Y, s'_0, \delta', f')$$

where

$$\begin{aligned} X' &:= \{0, 1\} && (\texttt{hist_eq}), \\ \Sigma' &:= \Sigma \times Z && (\text{add fresh inputs } z), \\ s'_0 &:= (1) && (\texttt{hist_eq}_0 = 1), \\ \delta' &:= \texttt{HistUpd}(\texttt{hist_eq}, \sigma), && (\text{use Equation (2)}) \\ f' &:= (f^1, f^2), \end{aligned}$$

with the following for the output functions of the two instances:

$$f^1 := z^1, \quad f^2 := \left(\texttt{hist_eq}_t \wedge (\sigma_t^1 = \sigma_t^2)\right) \ ? \ z^1 : z^2 \quad (4)$$

Blackboxing The construction of the abstracted machine M_{abs} *blackboxes* the module M, as it removes the concrete output function f, the internal transition function δ, and the state variables x. As a consequence, M_{abs} contains a 1-bit state variable and simple logic, significantly reducing the state space and improving model checking performance.

Besides, since f' is defined per output pair, it is feasible to apply HUF only to a subset of a module's outputs and keep other outputs concrete. We refer to this as *partially abstracting* (or *partially blackboxing*) a module.

Soundness The above HUF construction, which we call *default HUF*, is an over-approximation of the concrete product machine, and thus sound by construction. This is because *default HUF* only enforces output equality when all inputs are the same across the two instances for the current and all previous cycles, in which case, the two instances in the concrete product machine also generate identical outputs. As a result, any safety property proved under *default HUF* abstractions holds for the concrete design.

3.4. Variants of HUF

The *default HUF* above can be overly conservative when used to prove non-interference properties, and can induce false counterexamples due to the behaviors allowed by the abstraction, but not present in the original design. We now introduce two HUF variants that capture the information-flow properties of a module more precisely.

Consider a module whose inputs contain 2 variables: $\sigma = \langle valid, data \rangle$. This is commonly seen in processors where modules communicate with each other via handshake interfaces. When abstracting such a module with *default HUF*, we have

$$\begin{aligned} &\texttt{hist_eq}_{t+1} := \texttt{hist_eq}_t \wedge (valid_t^1 = valid_t^2) \wedge \\ &\qquad (data_t^1 = data_t^2) \\ &f^1 := z^1 \\ &f^2 := \left(\texttt{hist_eq}_t \wedge (valid_t^1 = valid_t^2) \wedge (data_t^1 = data_t^2)\right) \ ? \\ &\qquad z^1 : z^2 \end{aligned}$$

We discuss how the HUF variants modify the computation of $\texttt{hist_eq}$ and the output functions. For brevity, we omit the update rule for $\texttt{hist_eq}$, since it uses the same condition as f^2. We also omit f^1, since it remains unchanged.

Subset HUF The *subset HUF* variant computes the equivalence condition using only a subset of the inputs, rather than the full set of inputs. When used on a given output signal, intuitively, this abstraction states that only a subset of the inputs may influence this output. This could be useful when abstracting the timing signal (e.g., `out_valid`) of a constant-time functional unit, where the timing signal depends only on whether the input is valid, but not on the actual data values. In this case, only $valid_t$ matters:

$$f^2 := \left(\texttt{hist_eq}_t \wedge (valid_t^1 = valid_t^2)\right) \ ? \ z^1 : z^2$$

Guarded HUF The *guarded HUF* variant further masks input signals before using them to compute the equivalence condition. This abstraction is useful to capture the behavior of a module that processes an input packet only when its associated valid bit is one, and ignores the packet otherwise. More concretely, when applying *guarded HUF* to the example above, we get:

$$\begin{aligned} f^2 := \big(&\texttt{hist_eq}_t \wedge (valid_t^1 = valid_t^2) \wedge \\ &(valid_t^1 \ ? \ data_t^1 : 0 \ = \ valid_t^2 \ ? \ data_t^2 : 0)\big) \ ? \ z^1 : z^2 \end{aligned}$$

Notice that these variants make the *HUF constraint* in Equation (3) stronger since the precondition $(\texttt{hist_eq}_t \wedge (\sigma_t^1 = \sigma_t^2))$ becomes weaker. As a result, these HUF variants lose the capability to over-approximate arbitrary modules. Instead, it introduces an additional *refinement proof obligation*: the concrete module must be proven to refine its corresponding HUF abstraction.

Thankfully, these refinement proof obligations are local to the corresponding modules and can be checked independently and in parallel.

3.5. Refinement Proof Obligation

To ensure soundness, we require the abstract machine M_{abs} *over-approximates* the behaviors of the concrete machine M, or equivalently, the concrete machine M *refines* the abstract machine M_{abs}. Intuitively, this means that the abstract machine admits all behaviors of the concrete machine. Formally, following our construction of the concrete and abstract product machines in §3.3, we say that M *refines* M_{abs}, if for every input trace and M's corresponding output

trace, there exists an extension to the same input trace with some valuations over z that produces an identical output trace in M_{abs}.

Definition 4 (Refinement)**.** Let the concrete FSM be $M = (X, \Sigma, Y, s_0, \delta, f)$ and the HUF-abstracted FSM be $M_{\text{abs}} = (X', \Sigma', Y, s'_0, \delta', f')$. We say that M *refines* M_{abs} iff for all $k \geq 0$,

$$
\begin{aligned}
&\forall \sigma_0, \dots, \sigma_{k-1}.\ \exists z_0, \dots, z_{k-1}. \\
&\text{with } x_0 = s_0 \text{ and } x'_0 = s'_0, \\
&\text{for all } t < k: \\
&x_{t+1} = \delta(x_t, \sigma_t), \qquad\qquad y_t = f(x_t, \sigma_t), \\
&x'_{t+1} = \delta'(x'_t, (\sigma_t, z_t)), \qquad y'_t = f'(x'_t, (\sigma_t, z_t)), \\
&y_t = y'_t.
\end{aligned}
\tag{5}
$$

To check the above refinement relation, we build a serial composition C of M and M_{abs}, which connects z, the free inputs of M_{abs}, with the outputs of M and uses the same driver for σ and σ'. We claim that M *refines* M_{abs} if the property $y = y'$ holds on C, as stated formally in the following theorem:

Theorem 1 (Refinement Check)**.** *Given a concrete FSM M, a HUF-abstracted FSM M_{abs}, and their serial composition C, if $P_{refine} \coloneqq y = y'$ holds on C, then M refines M_{abs}.*

The proof is included in Appendix A.2 for interested readers.

3.6. HUF Implementation

We briefly discuss a few practical issues that arise when implementing HUF and how we address them.

Spurious Combinational Loops The first issue arises when the original module is replaced by its HUF abstraction and reconnected to the rest of the circuit. In the abstracted module, the output is computed from the `hist_eq` register and the current inputs. As a result, the abstraction may introduce a new combinational path from the input to the output. In practice, there may already exist external combinational paths that connect the output back to the input. When these paths are composed, they may create a spurious combinational loop that does not exist in the original design.

To address this issue, we perform a module-level local static path analysis. For each output, we backtrack along all combinational paths and identify the subset of inputs that belong to its combinational cone of influence, meaning that there exists at least one combinational path from the input to the given output. We then only include these inputs when computing the output, while noting that all the inputs are still involved in computing the `hist_eq` bit.

Hierarchical Abstraction and Refinement Proof The second issue is the scalability of the refinement proof. Although each refinement proof is local to one module, it may still take prohibitively long or even time out, if the module has large states. For example, the floating point module (`FpPipeline`) in `SmallBOOM` synthesizes to around 120k gates and 13k registers. In our experimental setup, the refinement proof for this module times out, as proof complexity scales roughly exponentially with module size.

To address this issue, we introduce *hierarchical refinement proof*. Given a large-state module that is composed of a collection of inner submodules, we apply HUF abstraction recursively to these submodules. As a result, we decompose one monolithic refinement proof into multiple smaller proof obligations. The same soundness guarantee in §3.5 continues to hold under hierarchical abstraction.

4. Shuffle: Verification Using HUF

Having fully presented the HUF abstraction, including its formalization, variants, and refinement proof obligations, we now discuss how users can apply this abstraction in practice. Using the HUF abstraction naturally helps "guide" model checkers away from "rabbit holes" and enables more efficient discovery of useful invariants for multiple reasons. First, the `hist_eq` bit in the *HUF constraint* preserves information-flow relationships and provides a compact encoding of information-flow invariants. Second, by abstracting away (i.e., blackboxing) the internal functionality of a module, HUF prevents the model checker from exploring security-irrelevant internal logic in large sequential circuits. Together, these effects substantially reduce the state space and simplify proof search.

4.1. Leveraging Design Knowledge for Abstraction

At a high level, defining a HUF abstraction is equivalent to specifying, *at the module level*, which input signals may influence which outputs. Such a *modular information-flow specification* is often intuitive for designers to construct.

We use the model-checking problem formulation introduced in §2, i.e., verifying SISP on μOOM, to demonstrate that applying HUF is both intuitive and effective through three concrete examples.

Example 1: Abstracting the ROB Recall from §2.4, the model checker spends a substantial amount of time discovering invariants that refer to ROB internal signals, making it a good candidate for applying abstraction.

We apply the *default HUF* abstraction, computing the `hist_eq` bit by checking equality across all ROB inputs. The abstraction captures information-flow behavior needed for the proof: if the two ROB instances have received identical input histories, then their outputs, such as `commit`, should also match. Meanwhile, by blackboxing the ROB, invariants about its internal signals are no longer explored, saving considerable exploration time and allowing the model checker to focus more on more relevant invariants. For example, when proving SISP, the abstraction enables the model checker to focus on ROB's contribution to the equality of the `commit` signals across the two instances.

As shown later in §7.4, by abstracting two sequential modules, namely ROB and rename tables, with the *default HUF*, we can prove SISP on `SmallBOOM`.

Example 2: Abstracting the Adder We use the Adder as an example to illustrate how to abstract a constant-time functional unit. The Adder receives data inputs guarded by a valid bit from its associated issue queue and produces a computation result, also guarded by a valid bit. We refer to the input and output valid bits as *control signals*, and the input and output data as *data signals*.

To abstract the Adder while preserving its constant-time property, we apply the *default HUF* to the data output, and the *subset HUF* to generate the valid output using only its control inputs. This captures the following information-flow details: the output valid signal, representing computation completion, cannot be influenced by the data values on which the Adder operates. Our refinement proof will conclude that the Adder implementation is indeed constant-time and ensures that the abstraction preserves the soundness of the final SISP proof.

Example 3: Abstracting the Divider We illustrate how to abstract a non-constant-time functional unit. If the *default HUF* abstraction is applied directly, false counterexamples may arise. This is because, when the valid bits are 0, the two Divider instances are not actually processing input packets, but their data inputs may still differ due to default values on the signals. The *default HUF* would nevertheless compare these data inputs and treat the difference as observable, even though it should be ignored.

To address this issue, we apply the *guarded HUF*, computing `hist_eq` by checking equality of the valid bits and additionally using the valid bit to guard the comparison of the data signals. This abstraction provides a more precise information-flow specification than the *default HUF* and helps to exclude false counterexamples when used to prove the final security property.

4.2. The Verification Flow

We now describe the end-to-end verification flow when using the HUF abstractions. Overall, users need to supply some manual information to the verification framework SHUFFLE, and SHUFFLE takes care of the rest of the proof process. We start by describing the inputs and working mechanisms inside SHUFFLE, and then discuss how a user can supply the information.

SHUFFLE The framework takes the following inputs:

1) an RTL-level design under verification (DUV),
2) a security property to be verified,
3) HUF abstractions for selected modules in the DUV, and
4) any additional *assumptions*, which may either remain unproven, representing verification gaps, or be proven separately and supplied to the model checker as auxiliary lemmas to accelerate verification.

Given these inputs, SHUFFLE discharges two types of proof obligations and solves them with a model checker:

- *Module-level refinement proof obligations.* For each abstracted module, prove that its abstracted HUF instance is a conservative over-approximation of the concrete module under the declared history inputs and guards. For large modules, the proof can be further decomposed hierarchically, as detailed in §3.6.
- *Top-level property proof.* After substituting each listed instance with its HUF abstraction, prove the target security property on the self-composed DUV.

Selecting and Refining HUF Abstractions There are two questions to answer: (i) which modules to abstract, and (ii) which HUF variants to use, which determines whether we partially abstract a module or completely blackbox it.

For module selection, the user can choose modules that lie in the intersection of (a) modules whose functionality is not relevant to achieving the final property, and (b) modules that consume substantial model-checking time. The property-irrelevant aspect can often be reasoned about at a high level. For example, proving SISP requires invariants describing the activation of correct data paths (e.g., the Divider is unused because it is not constant-time). This requires the dispatch module to behave correctly; thus, it cannot be abstracted. The model-checker-heavy aspect can be analyzed as in §2.4, by plotting which modules are frequently referenced in partially generated invariants.

In terms of which HUF variants to use, our examples in §4.1 show that specifying local information-flow constraints is typically straightforward and intuitive for designers. Designers generally know whether a module implements features that introduce timing variability. If the implementation deviates from the intended behavior, the refinement proof will fail and catch such module-level timing vulnerabilities.

For large design projects, this level of manual verification effort is quite tractable and worthwhile. Alternatively, one could explore automating this process through trial and feedback, especially in cases where high-level design insights are unavailable. When a coarse-grained abstraction is used, the top-level property proof may fail and generate a counterexample. Analyzing the counterexample can then guide the verifier to either unblackbox a module or use a more fine-grained variant of the HUF abstraction.

5. HUF-Guided Design Guideline for Secure Speculation

The same abstraction insight above that enables scalable verification also inspires us to formulate a verification-guided design guideline: security mechanisms can achieve better verifiability by reducing their reliance on hard-to-prove global functionality. We illustrate this idea through case studies on secure-speculation mitigations and show how small design adjustments can substantially improve verifiability.

Since the SISP property is relatively simple, as it can be handled by SHUFFLE out of the box (shown in §7), we now shift our focus to a more complex property, commonly referred to as the *secure-speculation contract* [10]. In this section, we start by formulating the verification problem in §5.1, and presenting existing defense designs in §5.2. We then give examples of design adjustments that follow our

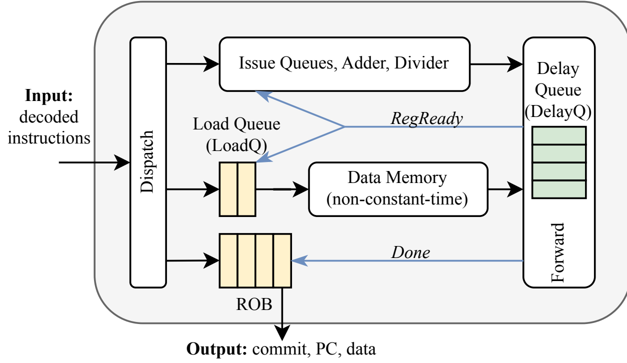

Figure 4: Extending the μOOM backend with a load data path and the *DelayFwd-All* defense.

HUF-guided design guideline and substantially improve the verifiability in §5.3.

5.1. Verification Problem Formulation

5.1.1. Extending μOOM with A Load Data Path. To facilitate our discussion of information leakage caused by speculative execution, we extend the μOOM processor (§2.1) with a load data path that interacts with a data memory, shown in Figure 4. The extension introduces the key ingredients needed for secure-speculation reasoning: a speculative data-access path and a mis-speculation handling mechanism. When specifying security properties later, we assume that secret data resides in memory in the initial state.

The extended μOOM supports three instructions: `Add`, `Div`, and `Ld`. We classify instructions into two categories: (i) *access instructions*, which can obtain secret data (in our case, the `Ld` instruction); and (ii) *transmitter instructions*, also referred to as unsafe instructions, which can leak operand values through microarchitectural side channels (the `Div` and `Ld` instructions).

The processor operates similarly to the original design, with a few minor modifications. First, the dispatch and forwarding modules need to interact with three issue queues, rather than two in the previous case. Second, both the `Div` and `Ld` instructions can trigger exceptions (e.g., divide-by-zero or misaligned address access). When an exception-triggering instruction reaches the head of the ROB, the ROB squashes all in-flight instructions, capturing the same speculation mechanism exploited by Meltdown [3].

5.1.2. Specifying the Sandboxing Contract. The Sandboxing contract specifies that if, during sequential execution (i.e., without speculation), a program never accesses a secret, meaning it never loads secret data into the register file, then its execution on a speculative hardware should not leak secrets through microarchitectural side channels.

Following prior work [6], [7], we adopt a two-machine self-composition formulation. Specifically, we construct a miter circuit by duplicating the extended μOOM processor and introducing an additional state bit to track whether the contract assumption holds. The state variables are $\langle x^1, x^2, \texttt{nonspec_safe}\rangle$, and the property is defined as:

$$P_{\text{contract}}(x) \;:=\; \texttt{nonspec_safe} \rightarrow (x^1.\texttt{commit} = x^2.\texttt{commit})$$

where `nonspec_safe` accumulates per-cycle assumption checks. At a high level, it remains true if the data values exposed by non-speculative execution match between the two instances, and is set to false otherwise.

5.2. Defense Design Variants

We look into a class of defenses designed to satisfy the Sandboxing contract property. These defenses work by delaying speculative data forwarding from *access instructions* to their dependent instructions. Representative designs include STT [13] and DOLMA [26], where STT has multiple RTL implementations [27], [28].

Figure 4 illustrates how the baseline, insecure design can be modified to delay speculative data forwarding. The forwarding logic receives computed results from the execution stage, informs the issue queues which registers are *ready*, and wakes up dependent instructions to consume the produced data. Instead of forwarding data immediately when it is ready, these defenses delay forwarding until it is safe to do so, that is, when releasing the data does not leak secrets through microarchitectural side channels.

We perform case studies on two forwarding policies:

- *DelayFwd-All*. A conservative policy that delays forwarding to *any* dependent instructions.
- *DelayFwd-Transmitter*. A more efficient policy that only delays forwarding to *transmitter instructions*, i.e., those that may leak their operands through side channels.

The two schemes illustrate a tradeoff between verifiability and design complexity. *DelayFwd-All* is more conservative and therefore easier to verify, while *DelayFwd-Transmitter* is more efficient but harder to verify, since it requires proving the correctness of secret-data dependency tracking.

5.3. HUF-Guided Design Adjustments

We now illustrate our HUF-guided design guideline through two case studies. In both studies, we aim to reduce the security mechanisms' reliance on hard-to-prove global functionality properties. This enables more aggressive HUF abstractions and makes the corresponding proof substantially more tractable.

5.3.1. Case Study 1: *DelayFwd-All*. We present a design of the *DelayFwd-All* scheme that satisfies the Sandboxing contract property without relying on the ROB's in-order commit behavior, a hard-to-prove global functionality property.

Design To support the delayed-forwarding mechanism, the defense introduces a buffer, named Delay Queue (`DelayQ`), inside the forwarding module. The `DelayQ` stores speculative data that cannot yet be released (Figure 4). The

critical design choice here is to organize the `DelayQ` similarly to the ROB, indexing it by the `robID` of the producer instruction. It releases data when it receives a notification from the ROB that the corresponding producer instruction has committed.

How Does It Help Verification? The design above can achieve the security property even if the ROB commits instructions in arbitrary order. Recall that, under the contract assumption `nonspec_safe`, the data produced by committed instructions are equal across the two instances. Since the `DelayQ` and ROB use the same indexing mechanism, this equality transfers to the released `DelayQ` entries, which is the critical invariant needed to establish the proof. As a result, the proof no longer depends on reasoning about the ROB's in-order commit behavior. HUF abstraction preserves the capability to propagate such equality relationships while abstracting away the ROB's complex internal logic.

5.3.2. Case Study 2: *DelayFwd-Transmitter*. Unlike *DelayFwd-All*, which delays all speculative forwarding, *DelayFwd-Transmitter* only delays forwarding whose destination is a *transmitter instruction*, and therefore needs to track whether a value depends (directly or indirectly) on a prior *access instruction*. To do so, STT [13] augments the physical register file with metadata that records the youngest *access instruction* on which each register depends, referred to as *Youngest Root of Taint (YRoT)*. This design efficiently encodes the minimal amount of dependency information and is area-efficient.

Design Adjustment *YRoT*, by definition, requires reasoning about the relative order among instructions, which is a hard-to-prove global functionality property. We propose an alternative design that simplifies the proof by reducing this dependency.

The key adjustment is to track all dependent *access instructions*, rather than only the youngest one. Accordingly, we record an *All Roots of Taint* vector for each register. We encode these vectors into a table, called *RoT*, that captures the dependencies between each register and the *access instructions* on which it depends. By removing the obligation to reason about relative ordering among instructions, our design enables verification with ROB abstracted by HUF.

Remaining Verification Challenges As we show later in §7, even with this adjustment, *DelayFwd-Transmitter* still presents other challenges that cannot be fully addressed with HUF and require a moderate number of manually identified assumptions. Nevertheless, our design adjustment helps improve verifiability with a significantly reduced number of manual assumptions compared to prior work [28], [29]. This exercise also helps pinpoint the remaining challenges in automating the generation of cross-module invariants, which we leave for future work.

6. Experiment Setup

Implementation SHUFFLE is implemented as a Yosys [30] plugin together with supporting scripts. The Yosys plugin takes user-specified HUF abstractions in YAML format and, for each specified module, generates the corresponding HUF-abstracted product machine in Verilog. By manipulating the RTLIL representation, the plugin further generates (1) the module-level refinement proof obligations and (2) the top-level security proof obligation on the miter circuit in which selected modules are abstracted. The proof obligations are sent to the rIC3 model checker [21] as parallel verification tasks.

In addition to the capabilities discussed in §3.4 and §3.6, our implementation also supports abstracting a module's internal signals, not only its interface signals. Furthermore, we implemented a plugin for rIC3 that maps its internal state variables to Verilog signals to assist invariant analysis.

BOOM Configurations BOOM [12] is an out-of-order processor widely used in academic research. We evaluate SHUFFLE for the following BOOM configurations: `SmallBOOM`, `MediumBOOM`, `LargeBOOM`, and `MegaBOOM`. All except `SmallBOOM` support superscalar execution and can commit more than one instruction per cycle. They share the same pipeline structure and differ only in resource sizes (e.g., ROB size, physical register count, and execution unit count).

Following the design adjustments described in §5.3, we implement *DelayFwd-All* and *DelayFwd-Transmitter* on BOOM, with 626 and 2129 lines of Chisel code changed, respectively. The changes are available as patches in our open-source repository.

To enable proof exploration on the most challenging tasks, we also introduce `TinyBOOM`, a downgraded version of `SmallBOOM` with four ROB entries, four integer registers, four floating-point registers, and one issue slot for each issue queue. In addition, the data bypassing logic is disabled.

Verification Setup We follow the miter-circuit construction used in prior work [8]. The miter circuit is built with self-composed `BOOMCore` modules. When verifying SISP, we specify secrets residing in the register file by driving the output data from the register file with fresh symbolic inputs. When verifying the Sandboxing contract, we specify secrets residing in memory by driving the response data from the load-store-unit module with fresh symbolic inputs. We evaluate three verification tasks:

1) verifying SISP on unmodified BOOM,
2) verifying the Sandboxing contract on BOOM with the *DelayFwd-All* mitigation,
3) verifying the Sandboxing contract on BOOM with the *DelayFwd-Transmitter* mitigation.

Table 1 lists the user-specified HUF abstractions and additional manual assumptions for each verification task.

7. Evaluation Results

We evaluate the effectiveness of our verification scheme from multiple perspectives: (i) overall verification time (§7.1), (ii) discharged refinement proofs (§7.2), (iii) invariants found by the model checker (§7.3), (iv) how the

Property	Mitigation	Abstracted Modules	Manual Assumption(s)
SISP	None	Rename, ROB, IssueUnit, ALU, etc. (12)	No
Sandboxing	*DelayFwd-All*	Rename, ROB, RegFile, IssueUnit, Decode, RegisterRead, ALU, etc. (17)	No
Sandboxing	*DelayFwd-Transmitter*	Rename, ROB, CSR, data path of functional units. (9)	Yes

TABLE 1: User-specified HUF abstractions and whether manual assumptions are required, for the three verification tasks. The numbers in parentheses report the number of abstracted modules among the 20 top-level modules in `BOOMCore` (we consider a module abstracted if any of its inner modules has been abstracted).

verification time changes with different HUF abstractions (§7.4), and (v) overhead of the proof-guided mitigation designs (Appendix D). Our experiments are performed on a platform equipped with an Intel® Core™ i9-14900K CPU and 64 GB of memory.

7.1. Verification Time

Figure 5 shows the top-level security proof time and the maximum refinement proof time for the first two verification tasks. Since the top-level security proof and refinement proofs are fully parallelizable, the overall runtime is determined by the longest proof among them, as also reported in the figure. Overall, our verification scheme is highly effective and scales the model checker to prove SISP and *DelayFwd-All* across all BOOM configurations, including the largest, `MegaBOOM`.

For both verification tasks, the verification time increases exponentially as the core size grows (the verification time is plotted on a logarithmic scale). The refinement proof time remains moderate compared to the top-level security proof, suggesting that refinement proof obligations do not dominate the verification complexity.

Specifically, for the SISP verification task, model checking the unabstracted DUV times out (> 24 hours) even on `SmallBOOM`. With the HUF abstractions, the SISP verification on `MegaBOOM` completes in around 15 minutes. This is about 12× faster than the reported VeloCT [8] result, which takes over 3 hours. Note that VeloCT [8] achieves its performance by utilizing 160 parallel cores, while our verification involves one top-level property proof and 10 refinement proofs, requiring only 11 parallel cores.

Limitation in Verifying *DelayFwd-Transmitter* We also candidly note that we have not yet achieved end-to-end proof for *DelayFwd-Transmitter* on BOOM. We made some partial progress in that we proved this defense on `TinyBOOM` in 12.5 hours with extra manual assumptions detailed in Appendix B. In this process, HUF helps identify the critical cross-module, functionality-based invariants needed to establish the security property. Without HUF, this investigation would have been far less structured and considerably more tedious.

7.2. Refinement Proofs

We now examine the refinement proofs separately, focusing on which modules require them and where the proof cost comes from.

When proving SISP, we abstract 42 modules in total. Among these abstractions, 32 are *default HUF* and therefore sound-by-construction, as discussed in §3.3, while the other 10 abstractions require refinement proofs. Their proof times in `SmallBOOM` are shown in Figure 6, sorted in descending order.

Note that the `FpPipeline` module benefits from hierarchical abstractions and *hierarchical refinement proof* (§3.6). Directly abstracting `FpPipeline` and discharging a monolithic refinement proof leads to a timeout. Instead, we recursively abstract its inner submodules, such as `FRegFile` and `FPUUnit`, generating multiple smaller proof obligations and reducing the refinement proof time to 2.1 seconds.

When proving *DelayFwd-All*, we abstracted 38 modules in total. Only three modules use HUF variants and therefore require refinement proofs. In `SmallBOOM` configuration, the refinement proof for `IRegFile` takes the longest time, at 59 seconds.

7.3. Invariants Analysis

We analyze the invariants found by the model checker when verifying SISP and *DelayFwd-All* on `SmallBOOM` and *DelayFwd-Transmitter* on `TinyBOOM`. Different security properties and mitigation designs lead to distinct invariant profiles. We hope that this under-the-hood analysis illuminates the challenges in each verification task and helps guide future work.

Figure 7 presents the distribution of invariants based on their lengths (i.e., number of literals). We further classify them into information-flow (cross-instance) and functionality-based (single-instance) invariants. For SISP and *DelayFwd-All*, the majority of invariants are short, containing fewer than five literals. The invariants are dominated by information-flow invariants, stating that a large number of signals are always equal across the two instances.

Manual inspection shows that, when verifying SISP, most one-literal functionality-based invariants state that certain signals are always 0, indicating the corresponding modules (e.g., the floating-point unit) or branch-related signals are never activated. When verifying *DelayFwd-All*, there

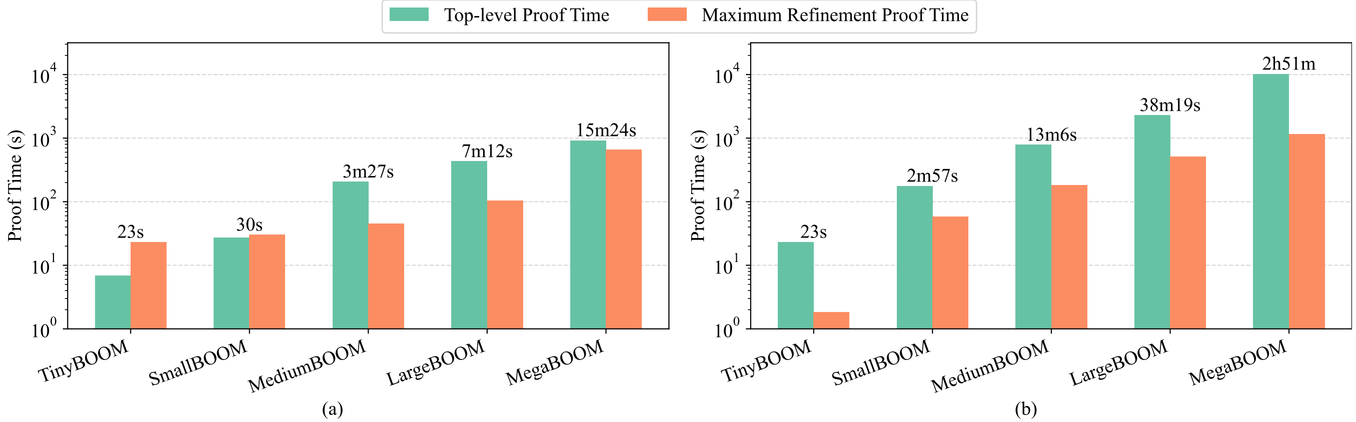

Figure 5: The top-level proof time and maximum refinement proof time when verifying (a) SISP and (b) the Sandboxing contract on *DelayFwd-All*. All times are shown on a logarithmic scale.

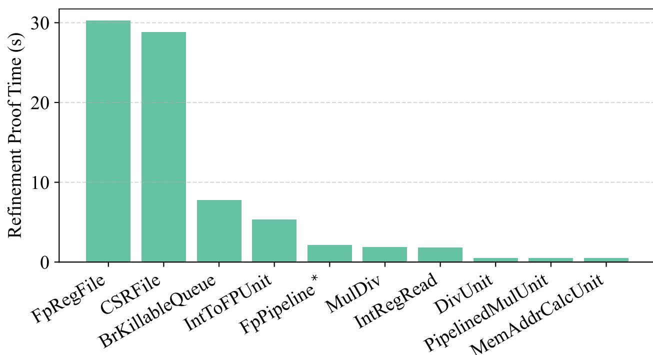

Figure 6: Verification time of refinement proofs when verifying SISP on `SmallBOOM`. Modules annotated with * have their submodules abstracted with *hierarchical refinement proof*.

are important functionality-based invariants that capture the relationships between the ROB and the `DelayQ` introduced by our defense.

In both SISP and *DelayFwd-All*, the number of invariants required for the proof increases roughly linearly as the core size grows, indicating that the invariants are well-structured and scalable.

In contrast, the invariants required for proving *DelayFwd-Transmitter* are much more complex, with many containing more than five literals and numerous functionality-based invariants. Manual examination shows that most invariants attempt to prove that the taint propagation is performed correctly, enumerating conditions that ensure the *RoT* table accurately tracks data dependencies among registers.

We provide further analysis to show how these invariants distribute across modules in Appendix C.

7.4. Abstraction Selection and Verification Time

For SISP, we study how different HUF abstractions influence the verification time. In Figure 8, we select six modules from BOOM and incrementally abstract each one of them. We then plot how the verification time changes as more modules are abstracted. The order in which modules are selected for abstraction follows the analysis presented in §2.4. We give priority to the modules whose signals are more frequently referenced by intermediate invariants found by the model checker before abstraction.

In general, verification time decreases as more modules are abstracted with HUF, though not strictly monotonically. Abstracting the `ROB`, `Rename`, and `ALU` contributes most to the speedup, as the first two contain large tables and the last has highly complex combinational logic.

8. Discussion & Future Work

We now discuss our work's limitations and potential future directions. First, our modular information-flow specifications, though easy to define and prove sound, often ignore interactions with other modules. We benefit from simplified refinement proofs where module inputs are unconstrained. However, in fact, modules operate within constrained environments that restrict the input traces they can observe. This limitation manifests in verifying *DelayFwd-Transmitter*, where the remaining proof challenges arise from complex cross-module invariants.

A promising direction for future work is to integrate *environment assumptions* into both the abstraction constraints and refinement checking, similar to prior work on functional correctness proofs [31]. This approach has the potential to further decompose global invariants into several local and presumably easier-to-prove invariants in a compositional manner.

Second, we have so far demonstrated the use of the HUF abstraction with the IC3 algorithm. However, there are other model-checking algorithms, such as *k-induction* [32] and

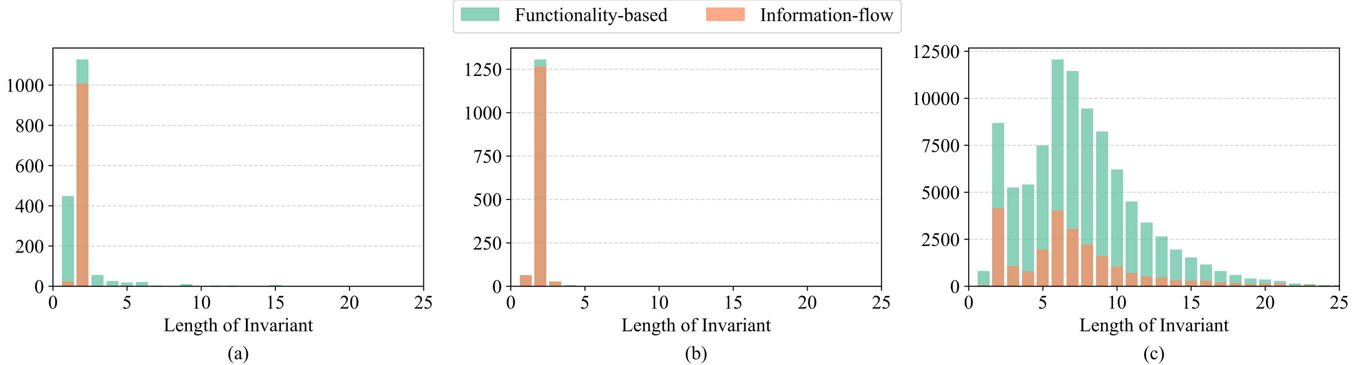

Figure 7: Invariant distribution grouped by the invariant length and types of invariants for different verification tasks: (a) verifying SISP on `SmallBOOM`, (b) verifying Sandboxing contract on `SmallBOOM` with *DelayFwd-All*, and (c) verifying Sandboxing contract on `TinyBOOM` with *DelayFwd-Transmitter*.

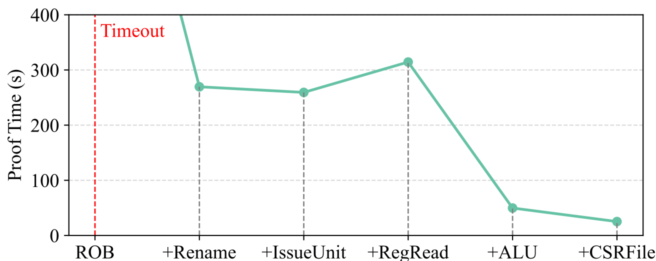

Figure 8: Verification time of SISP on `SmallBOOM` as HUF abstractions are incrementally applied.

interpolation-based approaches [33], which differ in how they search for inductive invariants. Although our techniques are instantiated to improve and guide invariant discovery in IC3, they are largely orthogonal to the underlying model-checking algorithm. A promising direction for future work is to evaluate how the HUF abstraction can also complement other model-checking approaches.

Third, we illustrate our proof-guided design guideline on a small set of forwarding-based defenses. Many hardware defense proposals, such as InvisiSpec [34] and ProSpeCT [35], were designed primarily from performance and area-efficiency perspectives. It would be an exciting direction to explore how the proposed proof-guided design guideline can be applied to them and inspire new design adjustments that ease their verification, or even lead to entirely new verification-friendly design guideline.

9. Related Work

We first discuss related work that is closest to us, in that they aim to verify the same security properties. ConjunCT [9] and VeloCT [8] verify the SISP property using grammar-guided invariant synthesis, which searches for invariants within the space defined by invariant-template grammars. Our approach is more efficient in proving these properties, as shown in our evaluation section. A more appealing aspect of our abstraction is that it can be reused to verify different properties. In contrast, it is unclear whether the invariant-template grammars (as well as the required expert annotations) used for SISP in these works can be effectively reused for other verification problems.

LeaVE [7], Contract Shadow Logic [6], and UPEC [28], [29] verify the speculative execution contracts. The scalability of LeaVE and Contract Shadow Logic is limited, with LeaVE restricted to in-order processors and Contract Shadow Logic to an in-house toy out-of-order core. Neither scales to large-scale out-of-order cores like BOOM. UPEC does scale to BOOM, but relies on substantial, manually identified assumptions.

We next discuss related work along other dimensions.

Verifying Non-Interference Properties There is extensive work that aims to verify non-interference properties at the RTL level, but not targeting out-of-order processors or the specific properties in this paper. Taint analysis [36], [37], [38] offers an alternative to self-composition by encoding information flow with taint propagation logic added to the design. SecVerilog [39] and SpecVerilog [40] express taint status through security types in Verilog. Knox [41], [42] proves information-preserving refinement across software and hardware layers. MTIsolation [43] defines and verifies strong isolation between programs co-located on the same machine. Commercial model-checking tools also provide built-in support for verifying non-interference properties, such as JasperGold Secure Path Verification (SPV) [44].

Using UF for Verification In processor verification, EUF has been used in functional verification [11], [23], [24], [25] to separate control and ordering correctness from data-path complexity. These UFs are primarily applied to combinational modules, such as ALUs.

In the context of security evaluation and verification, several works, including Cheang et al. [45] and Pensieve [46], use UF abstraction on functionality-irrelevant units (e.g., branch predictors) and on observable timing signals of side-channel structures (e.g., caches). None of these works performs aggressive UF abstraction as we do on data signals and sequential modules that are critical for

functional correctness, such as the ROB, issue units, and rename tables.

Verification-Guided Hardware Design Jauch et al. [28] integrate the UPEC formal verification framework [29] into the development of the Speculative Taint Tracking (STT) defense [13]. They use formal verification to iteratively discover instruction types that may introduce timing side channels. Their approach focuses on leveraging formal analysis to refine the defense, but it does not modify the design itself to improve verification scalability, as we do.

Many hardware description languages have been proposed to make verification an integral part of the design process, including SecVerilog [39], SpecVerilog [40], Caisson [47], Sapper [48], Koika [49], and Filament [50]. They aim to make hardware descriptions inherently amenable to formal reasoning.

10. Conclusion

This paper presents our research on advancing both model checking and secure processor design through insight into how they influence each other. We observe that the security of a processor does not depend on the full functional correctness of all its modules. Building on this insight, we introduce a modular abstraction primitive, the Uninterpreted Function with History (HUF), which preserves a module's information-flow properties while abstracting away irrelevant functionality. This abstraction enables scalable verification of large and complex out-of-order processors. We further show that this verification insight translates into design insight that improves the verifiability of secure processors. Overall, this research opens the door for research exploring the beneficial interplay between model checking and secure processor design.

Acknowledgment

The authors thank the Matcha Group (MIT) for their help and the anonymous S&P reviewers for their feedback. This work was supported in part by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA; by NSF under grants CNS 2046359 and CCF 2422052. Sharad Malik was supported in part by NSF grant 2422053.

References

[1] A. Fog, "The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers," 2025. [Online]. Available: https://www.agner.org/optimize/microarchitecture.pdf

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

[3] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium*. USENIX Association, 2018.

[4] J. Wikner, D. Trujillo, and K. Razavi, "Phantom: Exploiting decoder-detectable mispredictions," in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2023. [Online]. Available: https://doi.org/10.1145/3613424.3614275

[5] B. Chen, Y. Wang, P. Shome, C. Fletcher, D. Kohlbrenner, R. Paccagnella, and D. Genkin, "GoFetch: Breaking Constant-Time cryptographic implementations using data Memory-Dependent prefetchers," in *33rd USENIX Security Symposium*, 2024.

[6] Q. Tan, Y. Yang, T. Bourgeat, S. Malik, and M. Yan, "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, Volume 1*, 2025. [Online]. Available: https://doi.org/10.1145/3669940.3707243

[7] Z. Wang, G. Mohr, K. von Gleissenthall, J. Reineke, and M. Guarnieri, "Specification and verification of side-channel security for open-source processors via leakage contracts," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3576915.3623192

[8] S. Dinesh, Y. Zhu, and C. W. Fletcher, "H-HOUDINI: Scalable invariant learning," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: Association for Computing Machinery, 2025. [Online]. Available: https://doi.org/10.1145/3669940.3707263

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

[10] M. Guarnieri, B. Köpf, J. Reineke, and P. Vila, "Hardware-software contracts for secure speculation," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021.

[11] M. Velev and R. Bryant, "EVC: A validity checker for the logic of equality with uninterpreted functions and memories, exploiting positive equality, and conservative transformations," in *Computer Aided Verification*, 2001.

[12] J. Zhao, B. Korpan, A. Gonzalez, and K. Asanovic, "SonicBOOM: The 3rd generation Berkeley Out-of-Order Machine," in *Fourth Workshop on Computer Architecture Research with RISC-V*. International Symposium on Computer Architecture Valencia, Spain, 2020.

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

[14] J. Hennessy, D. Patterson, and C. Kozyrakis, *Computer Architecture: A Quantitative Approach*, ser. The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Science, 2025. [Online]. Available: https://books.google.com/books?id=JVfY0AEACAAJ

[15] Q. Tan, Y. Fisseha, S. Chen, L. Biernacki, J.-B. Jeannin, S. Malik, and T. Austin, "Security verification of low-trust architectures," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2023. [Online]. Available: https://doi.org/10.1145/3576915.3616643

[16] A. Ferraiuolo, M. Zhao, A. C. Myers, and G. E. Suh, "HyperFlow: A processor architecture for nonmalleable, timing-safe information flow security," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018. [Online]. Available: https://doi.org/10.1145/3243734.3243743

[17] M. R. Clarkson and F. B. Schneider, "Hyperproperties," in *2008 21st IEEE Computer Security Foundations Symposium*, 2008.

[18] G. Barthe, P. R. D'Argenio, and T. Rezk, "Secure information flow by self-composition," *Mathematical Structures in Computer Science*, 2011.

[19] N. Eén, A. Mishchenko, and R. Brayton, "Efficient implementation of property directed reachability," in *2011 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 2011.

[20] A. R. Bradley, "SAT-based model checking without unrolling," in *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer-Verlag, 2011.

[21] Y. Su, Q. Yang, Y. Ci, T. Bu, and Z. Huang, "The rIC3 hardware model checker," in *Computer Aided Verification: 37th International Conference (CAV)*. Springer-Verlag, 2025.

[22] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *International Conference on Computer Aided Verification*. Springer, 2010, pp. 24–40.

[23] J. R. Burch and D. L. Dill, "Automatic verification of pipelined microprocessor control," in *Computer Aided Verification*, D. L. Dill, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994.

[24] Y.-S. Ho, P. Chauhan, P. Roy, A. Mishchenko, and R. Brayton, "Efficient uninterpreted function abstraction and refinement for word-level model checking," in *2016 Formal Methods in Computer-Aided Design (FMCAD)*, 2016.

[25] S. Lahiri and R. Bryant, "Deductive verification of advanced out-of-order microprocessors," in *Computer Aided Verification*, 2003.

[26] K. Loughlin, I. Neal, J. Ma, E. Tsai, O. Weisse, S. Narayanasamy, and B. Kasikci, "DOLMA: Securing speculation with the principle of transient Non-Observability," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[27] B. Chen, R. Choudhary, K. Khulbe, A. Lee, A. Morrison, and C. W. Fletcher, "µSTT: Microarchitecture design for speculative taint tracking," in *2025 IEEE/ACM International Conference on Computer Aided Design (ICCD)*, 2025.

[28] T. Jauch, A. Wezel, M. R. Fadiheh, P. Schmitz, S. Ray, J. M. Fung, C. W. Fletcher, D. Stoffel, and W. Kunz, "Secure-by-construction design methodology for CPUs: Implementing secure speculation on the RTL," in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, 2023.

[29] M. R. Fadiheh, A. Wezel, J. Müller, J. Bormann, S. Ray, J. M. Fung, S. Mitra, D. Stoffel, and W. Kunz, "An exhaustive approach to detecting transient execution side channels in RTL designs of processors," *IEEE Transactions on Computers*, 2022.

[30] C. Wolf and J. Glaser, "Yosys Open SYnthesis Suite," https://yosyshq.net/yosys/, 2016.

[31] K. L. McMillan, "Verification of an implementation of Tomasulo's algorithm by compositional model checking," in *Proceedings of the 10th International Conference on Computer Aided Verification*, ser. CAV '98. Berlin, Heidelberg: Springer-Verlag, 1998, p. 110–121.

[32] M. Sheeran, S. Singh, and G. Stålmarck, "Checking safety properties using induction and a SAT-solver," vol. 1954, 11 2000, pp. 108–125.

[33] K. L. McMillan, "Interpolation and SAT-based model checking," in *Computer Aided Verification*, W. A. Hunt and F. Somenzi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 1–13.

[34] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. W. Fletcher, and J. Torrellas, "InvisiSpec: Making speculative execution invisible in the cache hierarchy," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-51. IEEE Press, 2018, p. 428–441. [Online]. Available: https://doi.org/10.1109/MICRO.2018.00042

[35] L.-A. Daniel, M. Bognar, J. Noorman, S. Bardin, T. Rezk, and F. Piessens, "ProSpeCT: Provably secure speculation for the Constant-Time policy," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 7161–7178.

[36] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood, "Complete information flow tracking from the gates up," *SIGARCH Comput. Archit. News*, 2009. [Online]. Available: https://doi.org/10.1145/2528521.1508258

[37] F. Solt, B. Gras, and K. Razavi, "CellIFT: Leveraging cells for scalable and precise dynamic information flow tracking in RTL," in *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, 2022. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/solt

[38] F. Solt and K. Razavi, "HybriDIFT: Scalable memory-aware dynamic information flow tracking for hardware," in *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*, ser. ICCAD '24. New York, NY, USA: Association for Computing Machinery, 2025. [Online]. Available: https://doi.org/10.1145/3676536.3676658

[39] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, "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)*, 2015.

[40] D. Zagieboylo, C. Sherk, A. C. Myers, and G. E. Suh, "SpecVerilog: Adapting information flow control for secure speculation," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3576915.3623074

[41] A. Athalye, M. F. Kaashoek, and N. Zeldovich, "Verifying hardware security modules with Information-Preserving refinement," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2022. [Online]. Available: https://www.usenix.org/conference/osdi22/presentation/athalye

[42] A. Athalye, H. Corrigan-Gibbs, F. Kaashoek, J. Tassarotti, and N. Zeldovich, "Modular verification of secure and leakage-free systems: From application specification to circuit-level implementation," in *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles (SOSP)*. Association for Computing Machinery, 2024.

[43] S. Lau, T. Bourgeat, C. Pit-Claudel, and A. Chlipala, "Specification and verification of strong timing isolation of hardware enclaves," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3658644.3690203

[44] Cadence, "Jasper Security Path Verification App," https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification.html, [Accessed 02-Nov-2025].

[45] K. Cheang, C. Rasmussen, S. Seshia, and P. Subramanyan, "A formal approach to secure speculation," in *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*, 2019.

[46] Y. Yang, T. Bourgeat, S. Lau, and M. Yan, "Pensieve: Microarchitectural modeling for security evaluation," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3579371.3589094

[47] X. Li, M. Tiwari, J. K. Oberg, V. Kashyap, F. T. Chong, T. Sherwood, and B. Hardekopf, "Caisson: A hardware description language for secure information flow," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Association for Computing Machinery, 2011.

[48] X. Li, V. Kashyap, J. K. Oberg, M. Tiwari, V. R. Rajarathinam, R. Kastner, T. Sherwood, B. Hardekopf, and F. T. Chong, "Sapper: A language for hardware-level security policy enforcement," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Association for Computing Machinery, 2014.

[49] T. Bourgeat, C. Pit-Claudel, A. Chlipala, and Arvind, "The essence of Bluespec: A core language for rule-based hardware design," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Association for Computing Machinery, 2020.

[50] R. Nigam, P. H. Azevedo de Amorim, and A. Sampson, "Modular hardware design with timeline types," *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2023.

[51] RISC-V International, "Benchmarks for RISC-V processor," https://github.com/riscv-software-src/riscv-tests/tree/master/benchmarks, 2025, [Accessed 16-Aug-2025].

[52] W. Snyder, P. Wasson, D. Galbi, G. Lore *et al.*, "Verilator," https://verilator.org, open-source Verilog/SystemVerilog simulator. Source code available at https://github.com/verilator/verilator.

Appendix A. HUF Formalization

A.1. Proof of Lemma 1

Lemma 1 (State Equivalence with History)**.** *For all* $t \geq 0$*:*

$$\texttt{hist_eq}_t \implies (x_t^1 = x_t^2)$$

Proof. Since the transition functions of the two instances are deterministic, if the inputs are pairwise equal across the two instances up to cycle $t-1$ ($\texttt{hist_eq}_t = 1$), it follows that their current state should also be equal ($x_t^1 = x_t^2$). □

A.2. Proof of Theorem 1

Theorem 1 (Refinement Check)**.** *Given a concrete FSM M, a HUF-abstracted FSM M_{abs}, and their serial composition C, if $P_{refine} \coloneqq y = y'$ holds on C, then M refines M_{abs}.*

The serial composition C for M and M_{abs} is shown in Figure 9, where the inputs of M and M_{abs} are connected with each other, and the y outputs of M are connected to the z inputs of M_{abs}.

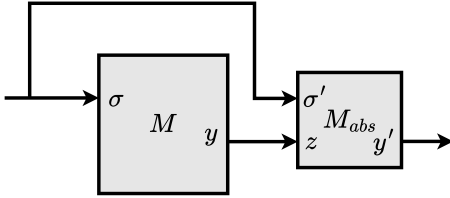

Figure 9: The serial composition C for checking the refinement relation.

We provide the proof of Theorem 1 as follows.

Proof. Assume C satisfies P_{refine}. Fix any $k \geq 0$ and any input sequence $\sigma_0, \ldots, \sigma_{k-1}$. Consider the run of C on this sequence. By construction,

$$\begin{aligned} x_{t+1} &= \delta(x_t, \sigma_t), \\ z_t = y_t &= f(x_t, \sigma_t), \\ x'_{t+1} &= \delta'(x'_t, (\sigma_t, z_t)), \end{aligned}$$

for all $t < k$. Because $C \models P_{\text{refine}}$, along this same run we also have $y_t = y'_t$ for all $t < k$. Therefore, for this input sequence, the sequence $z_0, \ldots, z_{k-1}$ defined by $z_t = y_t$ satisfies all constraints in Equation (5). □

Appendix B. Manual Assumptions and Assertions for Proving *DelayFwd-Transmitter*

As discussed in §7, we are unable to prove that *DelayFwd-Transmitter* satisfies the Sandboxing contract property by only using the HUF abstraction. To identify where our tool falls short, we manually analyze the proof process to identify the assumptions essential for establishing the property. This analysis helps explain why proving this defense is challenging and highlights that reasoning about complex cross-module invariants remains an open problem for future work.

Manually Identified Functionality Assumptions We note that, although the HUF abstraction does not enable us to complete the end-to-end final proof, it helps reveal key invariants by providing counterexamples generated by the model checker with selected modules abstracted. Moreover, our design adjustments in §5.3 remove the need to reason about the total ordering between instructions, reducing the number of assumptions compared to prior work.

All the following assumptions are functionality-based and reference signals from a single instance of the DUV.

- *Rename functional correctness:* No write-after-read (WAR) or write-after-write (WAW) hazards.
- *ROB index uniqueness:* No two in-flight instructions share the same ROB index.
- `BranchMask` *allocation correctness*: A producer instruction's `BranchMask` is a subset of the consumer instruction's `BranchMask`.

Auxiliary Assertions Help Accelerate Verification We also observe that the proof can be accelerated by asserting some cross-module specifications to guide the model checker to explore the final property in a more structured way. These helpful assertions come from two sources: (i) design insights about what properties the defense should achieve, and (ii) patterns observed from the intermediate invariants generated by the model checker.

- *Taint specification:* when a register has been untainted (its *All Roots of Taint* vector is cleared), the data is equal across two instances.
- *Data dependency tracking consistency:* Data dependencies used by the data paths involving modules, such as register read, ALU modules, and writeback, need to be consistent with the data dependency recorded in *RoT*.
- *Execution status between producer and consumer instructions:* For any pair of dependent instructions, the producer's destination's *All Roots of Taint* vector is a subset of the consumer's destination's *All Roots of Taint* vector, and the producer must not be in a squashed state.

Appendix C. Invariant Distribution Across Modules

Figure 10 presents the distribution of invariants over modules. When verifying SISP and *DelayFwd-All*, most invariants are local to specific modules. The invariants for proving *DelayFwd-All* include cross-module invariants that relate the `WritebackQ` (an auxiliary data structure in ROB to track the writeback data of each instruction) and the `DelayQ` introduced by the defense. The invariants for proving *DelayFwd-Transmitter* are mostly cross-module, since reasoning about taint-tracking correctness requires relating multiple pipeline modules involved in propagating data throughout the processor.

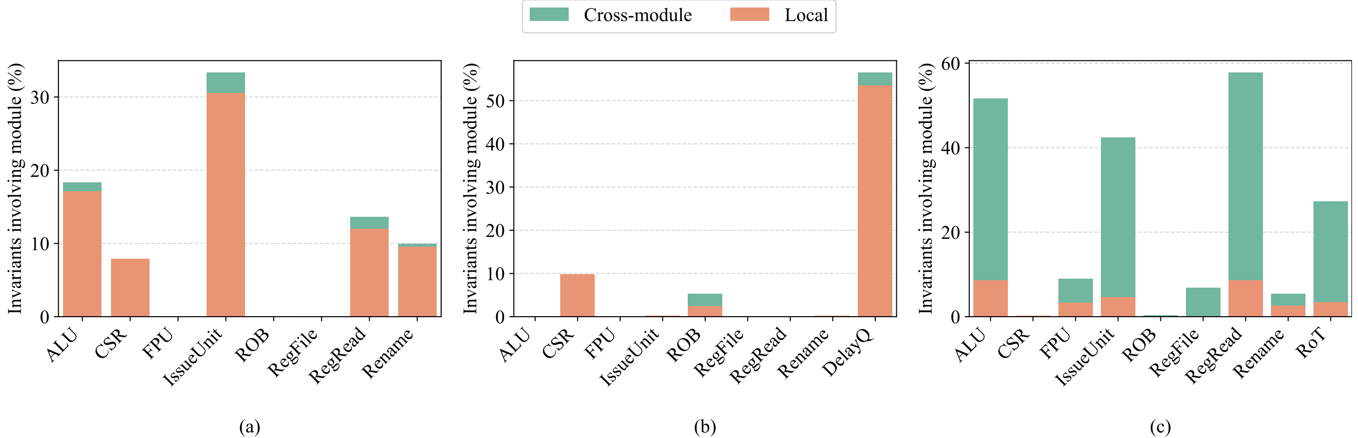

Figure 10: Invariants distribution over modules for different verification tasks: (a) verifying SISP on `SmallBOOM`, (b) verifying Sandboxing contract on `SmallBOOM` with *DelayFwd-All*, and (c) verifying Sandboxing contract on `TinyBOOM` with *DelayFwd-Transmitter*.

Appendix D. Defense Overhead

As we aim to make design adjustments to improve verifiability with minimal area and performance overhead, we present the evaluation results on `SmallBOOM` to support this claim.

In Table 2, we estimate area overhead by measuring the flip-flop and gate counts of the added logic for each defense, measured relative to the rename module. This relative metric aligns with the evaluation metric used in recent STT RTL implementations [27]. The `DelayQ` module added by *DelayFwd-All* incurs approximately 95% of the rename module's flip-flops and 117% of its gates. The *RoT* module introduced by *DelayFwd-Transmitter* incurs around 4× the rename module's flip-flops and 3× its gates. For comparison, the referenced STT implementations with *YRoT* encoding incur an area overhead of roughly 2–2.5× that of the rename module.

	# Flip-Flops	# Gates
Baseline `rename` Module	2.6k	22k
DelayFwd-All `DelayQ` Module	2.4k (95%)	26k (117%)
DelayFwd-Transmitter *RoT* Module	10k (391%)	70k (318%)

TABLE 2: Area overhead.

Performance evaluation is conducted with a set of standard RISC-V benchmarks [51] on Verilator [52]. We use the default configuration of `SmallBOOM` with 16KiB instruction and data caches. We compute the average latency across all the applications, normalized to the insecure baseline. The average latency overheads of *DelayFwd-All* and *DelayFwd-Transmitter* are 28.0% and 22.7%, respectively, which align with the magnitudes of latency overhead reported in prior defense work [13], [26].

Appendix E.
Meta-Review

The following meta-review was prepared by the program committee for the 2026 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

E.1. Summary

The paper introduces Uninterpreted Functions with History (HUF), an abstraction technique that improves the scalability of model checking for relational security properties in hardware. HUF replaces complex stateful modules with abstractions that capture only the input-output dependencies relevant to information flow. Verification is split into proving the abstraction's correctness and checking the security property on the abstracted model. The approach enables successful verification of several security properties on BOOM processor configurations where baseline methods timed out.

E.2. Scientific Contributions

- 3. Creates a New Tool to Enable Future Science.
- 6. Provides a Valuable Step Forward in an Established Field.

E.3. Reasons for Acceptance

1) The HUF-approach presented in this paper provides a clean way of separating functional correctness reasoning from security reasoning, which may have the potential to significantly reduce verification complexity.
2) Experimental results show that HUF enables verification of security properties on BOOM processor configurations where previous model-checking attempts timed out or took significantly longer.
3) The paper is well written and the technique is easy to understand. It provides interesting insights into the interplay of verifying security and functional correctness properties.

E.4. Noteworthy Concerns

The evaluation reports data on the invariants generated for a proof that did not finish (*DelayFwd-Transmitter* mitigation against the Sandboxing contract); these results might change for an end-to-end proof.

Appendix F.
Response to the Meta-Review

We acknowledge the limitation that the proof for *DelayFwd-Transmitter* is not end-to-end. However, we intentionally include its analysis because we believe it constitutes a meaningful contribution. To our knowledge, this is the first attempt to use open-source model checkers to push the boundary of verifying such a challenging property on a complex processor.

Beyond the result itself, the analysis provides concrete and novel insights into this verification task: proving such properties requires long, cross-module invariants that correlate signals across different components. Transitioning to an end-to-end proof by removing manual assumptions would not mitigate this challenge.

While the immediate strength of the paper lies in (1) the clear performance improvement on verifying SISP over prior work and (2) the verification on *DelayFwd-All*, we believe its longer-term impact comes from establishing a foundation that informs and inspires future work toward fully end-to-end proofs of relatively more challenging properties and mitigations, such as *DelayFwd-Transmitter*.