Compositional Proofs of Information Flow Properties for Hardware-Software Platforms

Kevin Cheang
Adwait Godbole
Yatin A. Manerkar
Sanjit A. Seshia

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2023-204
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2023/EECS-2023-204.html

August 9, 2023

Compositional Proofs of Information Flow Properties for Hardware-Software Platforms

Kevin Cheang[1], Adwait Godbole[1], Yatin A. Manerkar[2], and Sanjit A. Seshia[1]

[1] University of California, Berkeley,
[2] University of Michigan

Abstract. While agile hardware design flows have led to performant computation platforms, hardware security vulnerabilities pose a threat to security-critical software running on these platforms. Verifying programs running on such platforms w.r.t. security properties faces two major challenges: (a) vulnerable software is often nested within large pieces of code, and (b) security properties require more fine-grained platform models (compared to functional properties), leading to complex verification queries. Our work is motivated by these challenges. To address (a) we develop *SymboTaint*, a Hoare-style proof system that allows the decomposition of monolithic security specifications over large programs into smaller verification conditions. For challenge (b), we develop *Information Flow State Machines* (IFSMs), a modeling framework that provides compositionality properties. As a case study, we develop a speculative microprocessor model called *Speculative Abstract Platform* (SAP) which captures hardware designs with a wide range of microarchitectural features. As an evaluation, we use IFSMs to model SAP and verify observational determinism on a broad class of attack programs running on SAP.

1 Introduction

The rise of user-friendly, high-level hardware design frameworks [3, 25, 30] has made agile development and optimization of hardware computation platforms more accessible. These designs range from general-purpose computers and domain-specific computation engines (accelerators), to platforms designed with security as the guiding principle [1, 2, 13, 17, 18, 26, 27, 33]. The usage of ever more efficient hardware systems, however, has been plagued by the existence of hardware execution attacks [6, 8, 9, 29, 34, 42, 51–53]. Formal methods can provide strong security guarantees about system behaviour in the context of these attacks, thus building trust in the system.

Most hardware execution attacks exploit *microarchitectural* features such as caches, branch predictors, and load/store buffers. While reasoning over the architectural state (e.g., program counter, registers) suffices for proving functional correctness of software, one also has to account for the microarchitectural state when proving security properties. Since the microarchitectural state is more detailed than the architectural state, software semantics at the microarchitectural result in especially challenging verification queries. This problem is made more severe given that vulnerable software fragments are typically nested in large pieces of code. In this work, we develop a concerted approach to make verification of information-flow-style properties scale. On the software side, our approach uses Hoare-style reasoning tailored to security properties, while on the hardware side we leverage compositionality of the platform model.

Fundamental contributions such as Hoare-logic [23], interpolants [35] have enabled techniques such as interpolation-based reasoning [37], and invariant inference (e.g. [11, 15]) which have improved the scalability of software verification. These techniques have been predominantly used for checking single-trace properties such as safety and functional correctness. Security properties such as non-interference [12] on the other hand are *hyperproperties* defined over sets of traces. While some hyperproperties can be compiled to single-trace safety properties (over the self-composition), and hence permit the above approaches, such encodings do not make use of the specialized nature of security properties. This informs our first research question:

(RQ1) How can we tailor Hoare-style proof techniques to better scale verification of information-flow-based properties such as non-interference?

We answer this question by developing a proof system called ***SymboTaint***, which combines the symbolic representation of state, with taint-like equivalences over system variables. These equivalences align with security properties such as non-interference which prescribe equivalence between two executions w.r.t. certain variables. The symbolic state allows for more precise reasoning than pure taint analysis.

Security verification of software is closely tied with the microarchitectural semantics of the underlying hardware. Additionally, the capabilities of an adversary also vary with the microarchitecture, as some microarchitectural features create new side channels [49], leading to modified security specifications. While microarchitectural semantics are much more detailed, verification can benefit from frameworks that leverage compositionality of hardware. The choice of modelling framework also impacts whether one is able to easily instrument the model with proofs. Hence, we ask:

(RQ2) What modeling formalisms allow compositionality and parameterizability, and connect better with the software-side proof techniques?

We answer this question by developing an abstract operational model called the ***Information Flow State Machine*** (IFSM). Intuitively, an information flow state machine augments the underlying platform model with the joint symbolic-taint analysis from the proof system. This enables better interoperability between the proof and the model. Additionally, under some conditions, the transition relation of an IFSM can be decomposed. This allows the proof to reason about only those components of the platform that are relevant to the security property.

To evaluate the efficacy of our approach, we introduce a speculative platform model and verify the security of several safe and vulnerable programs. These are representative of a broad class of transient execution attacks [8, 24] targeting various microarchitectural features. Our verification approach is based on IFSMs which instrument the speculative platform with the proof. We compare the performance of our technique with prior work on verifying transient execution attacks [10] and observe improved performance across safe and unsafe examples. In summary, we make the following main contributions:

1. **SymboTaint Proof System**. We introduce SymboTaint, a sound proof system that specializes pre/post-conditions from Hoare-style proofs to capture invariants common to security proofs for programs.

2. **Information Flow State Machines**. We introduce IFSMs, an operational model that allows us to connect proofs from SymboTaint with the platform model. We develop conditions under which IFSMs can be decomposed for more efficient analysis of security properties.
3. **Speculative Abstract Platform Model**. We introduce the SAP parameterized platform model which abstractly models a speculative microprocessor. The model captures a wide combination of microarchitectural features and attack vectors, beyond what models in the literature capture. We use IFSMs in this modelling, demonstrating the compositionality of microarchitectural features.
4. **Evaluation on Transient Execution Attacks**. We evaluate our methodology - the model and proof system - by verifying transient execution attacks on the SAP model. We check a broad class of transient execution attack examples against the secure speculation [10, 21] property. We observe performance improvements over the monolithic proof approach from [10].

Outline. In §2 we motivate the problem by considering an example of an attack vector. In §3 we introduce the platform and attacker model, and security properties of interest. In §4 we develop the SymboTaint proof-system which enables Floyd-Hoare-style proofs for verification of security properties. In §5, we introduce IFSMs, an operational formalism that allows instrumenting the platform model with proofs written in SymboTaint. We also develop a notion of composition for IFSMs that enable concise proofs. In §6 we present a speculative abstract platform (SAP) model that is capable of capturing a broad class of transient execution attacks from [8, 24] and perform experimentation on this model in §7. We discuss related work in §8 and conclude in §9.

2 Motivation

We motivate our methodology and abstractions with the problem of verifying classes of transient execution attacks. In recent literature, secure speculation [10] has been consistently used to capture speculation dependent vulnerabilities in micro-architectures. This property is an extension of observational determinism [55] which itself is a flavor of the non-interference property [12]. Our work is based on using non-interference style properties to identify a broad class of transient execution attacks. We begin by presenting a motivating example of such an attack.

Fig. 1 illustrates `victim_func`, a function that is owned by a victim process and is callable by an adversary process. Ignoring for the moment the first two segments labelled A and B (lines 2-5), segment C (lines 7-9) shows the first discovered transient execution attack called Spectre V1 (bounds check bypass) [29]. The way Spectre V1 works is that the adversary can train the branch predictor to mispredict the condition `(x < N)`, thereby coercing the processor to transiently execute line 9. This results in an access to a potential victim's secret using `a[x]` and then a secret dependent access `arr2[a[x] * 512]`. This access leaves observable side effects in the data cache covert channel. The adversary can observe these side-effects and hence infer the secret.

Complexity in software. A vulnerable code segment such as the one above does not often appear in isolation. It may appear alongside other complex code segments, such as

```
 1 int victim_func(int x) {
 2     // A: Init secret pointer
 3     int* a = &secret;
 4     // B: Secret dependent code
 5     ...
 6     // C: Spectre V1 / BCB
 7     a = arr1;
 8     if (x < N)
 9         tmp = arr2[a[x] * 512];
10 }
```

Fig. 1. Victim program executing in the trusted user's domain with input `x` which is adversary controlled. This function is vulnerable to Spectre V1 (BCB), Spectre V4 (store-bypass), their combination and leakage from segment B.

```
// C1: lines 7-8
addi a3, gp, -88
store a3, s0, -24
bge a0, a1, END
// C2: line 9, a[x]
load a3, s0, -24;
load a3, a3, 0
add a4, a0, a3
load a4, a4, 0
// C3: line 9, arr2[a[x]*512]
muli a4, a4, 512
addi a3, gp -48
load a3, a3, 0
add a4, a4, a3
load a5, a4, 0
```

Fig. 2. Instruction level translation of lines 4-6 of the program in Figure 1.

A and B in Fig. 1. Segment A may non-trivially interact with segment C, with potential security implications. For example, even if the adversary was unable to mistrain the branch predictor, a faulty store-to-load forwarding of the secret address `a` from line 3 to line 9 could result in a variation of the Spectre V4 (speculative store bypass) attack [8]. Similarly, segment B can contribute its own secret dependent effects to an exploitable side-channel, such as the line fill buffer [42], resulting in data leakage. This example illustrates the challenges in verifying large pieces of code monolithically and motivates approaches that decompose the proof. In §4 we develop such an approach which takes the form of a proof system. In §6 we apply this proof system to an abstract speculative microprocessor model that can execute assembly-like code.

Complexity in hardware. Another aspect that complicates the verification of software such as `victim_func` is the necessity to model the hardware at the microarchitectural level, resulting in a massive model. Hence, developing modeling approaches which facilitate compositionality is desirable. While this is true, reasoning over subsets of components in an unguarded manner can lead to false guarantees. Figure 3, illustrates such an example inspired by the CacheOut attack [43], This attack exfiltrates in-flight data (potentially containing secrets) from the line-fill-buffer (LFB) into the cache, which then serves as the side-channel. An analysis on the model in Fig. 3A, sans the LFB, can be imprecise. Hence, compositionality though useful requires care. In §5.1, we present an operational modeling framework which enables composition under certain conditions. The speculative platform model from §6.1 has a wide range of microarchitectural components, yet satisfies these conditions, resulting in efficient verification.

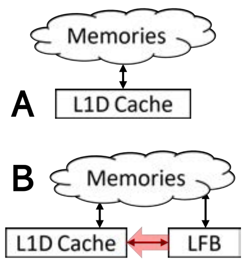

Fig. 3. Different levels of modeling detail.

2.1 Approach Overview: Efficient Proofs with Interpolants

Security properties such as non-interference [19] are based on a notion of observation that identifies when two states are considered to be equivalent. Non-interference in particular requires that two executions which start in equivalent states must also end in equivalent states. One prominent approach [10] used to verify such properties for a program is bounded model checking (BMC) [36]. This is presented in Eq. 1, which is a relation over traces of two instances of the same transition system. It states that if the two instances of the system start in some initial states (say $q_1^{(0)}$ of the first instance and $q_2^{(0)}$ of the second) that are *low-equivalent* ($q_1^{(0)} \approx_{\mathrm{L}} q_2^{(0)}$), and each instance takes k steps (i.e., $\delta^k(q^{(0)}, q^{(k)}) = \delta(q^{(0)}, q^{(1)}) \wedge \ldots \wedge \delta(q^{(k-1)}, q^{(k)})$) to transition from the initial states to the final states ($q_1^{(k)}, q_2^{(k)}$), then the final states should be low-equivalent ($q_1^{(k)} \approx_L q_2^{(k)}$).

$$q_1^{(0)} \approx_L q_2^{(0)} \wedge \delta^k(q_1^{(0)}, q_1^{(k)}) \wedge \delta^k(q_2^{(0)}, q_2^{(k)}) \Rightarrow q_1^{(k)} \approx_L q_2^{(k)} \tag{1}$$

However, a monolithic proof of non-interference (such as a direct translation of the Eq. 1 into a BMC query) results in a large verification query. One way to address the complexity of a monolithic proof is to decompose it into smaller proofs, using intermediate properties, or what we refer to as interpolants, to connect them. This can be intuitively conceptualized through the following three equations.

$$q_1^0 \approx_L q_2^0 \wedge \delta(q_1^0, q_1^1) \wedge \delta(q_2^0, q_2^1) \Rightarrow J_1(q_1^1, q_2^1) \tag{2}$$

$$\forall i \in \{1, \ldots, k-1\}.\ J_i(q_1^i, q_2^i) \wedge \delta(q_1^i, q_1^{i+1}) \wedge \delta(q_2^i, q_2^{i+1}) \Rightarrow J_{i+1}(q_1^{i+1}, q_2^{i+1}) \tag{3}$$

$$J_k(q_1^k, q_2^k) \Rightarrow q_1^k \approx_L q_2^k \tag{4}$$

Instead of a single query, one may break down the proof into a set of smaller verification conditions. This is possible by identifying intermediate conditions $J_1, \cdots, J_k$ such that: (a) the initial conditions and transition constraints implies J_1 (Eq. 2), (b) J_i and transition constraints imply J_{i+1} (Eq. 3), and (c) the final interpolant $J_k(q_1^k, q_2^k)$ implies that the final states are low-equivalent (Eq. 4).

So far this is just standard interpolant-based reasoning. The crux of effectively using this approach to address the complexity issues mentioned lies in the shape of the interpolants J_i used, and how the system model (i.e., δ) is represented. One type of property that can naturally serve as part of these interpolants J_i for non-interference-style properties, is the class of relational properties of variables between the two system instances in a non-interference proof. Namely, this is the information described by the set of equality constraints $\{q_1^i(v) = q_2^i(v)\}_{v \in V_i}$ for $V_i \subseteq V$, where $q(v)$ denotes the value of variable v in state q. One can view this as a summary of which variables between the two instances are still low-equivalent after the i-th step. This intuition is formalized in §4 as the *SymboTaint* proof system. On the modeling side, it is highly desirable to be able to decompose or separate a system into simpler components. Exploiting the simpler proof obligations from Eq. 2-4, one can then hope that a smaller system recomposed from the decomposed components is sufficient for sound analysis of the property. This intuition is materialized in §5.1 as *Information State Flow Machines*.

3 Security Model

In this section we begin by introducing our programming model in §3.1 followed by the attacker model in §3.2. We provide background on security properties in §3.3. We consider in this work the following security properties: non-interference [19], observational-determinism [55], and trace-property observational determinism [10].

3.1 Programming Model

We start by developing the system model based on which security properties are defined. We adopt a standard state-transition system, $M = \langle Q, I, \delta \rangle$ with states Q, initial states $I \subset Q$ and transition relation $\delta \subseteq Q \times \mathsf{Op} \times Q$ with transitions labelled by operations from Op. It is often useful to view states as assignments to the variables in the system. In this view, $q \in Q$ is map from variables $v \in V$ to values from some domain $\mathbb{D}$: $q : V \rightarrow \mathbb{D}$.

A program is a word over Op (i.e. $P \in \mathsf{Op}^*$) which generates an execution. The execution on a program $P = \mathsf{op}_1 \cdots \mathsf{op}_n$ is a trace of states $\pi_P = q^{(0)}...q^{(n)} \in \mathit{Tr}(M)$, where the initial state belongs to I and (b) consecutive states $q^{(i)}, q^{(i+1)}$ have a valid transition under op_{i+1}: $\delta(q^{(i)}, \mathsf{op}_{i+1}, q^{(i+1)})$. For trace π, we write $\pi^{(i)}$ for the i-th state of the trace. $\mathit{Tr}(M)$ represents valid traces of M (across programs in Op^*), while $\mathit{Tr}_P(M)$ represents traces corresponding to program P. A program can lead to several traces (e.g. if the system is *non-deterministic*).

```
// Core variables and operations
core {
    // System state variables
    var pc : word_t;
    var regs : [regindex_t]word_t;
    ...

    // Operations
    operation add (rs1, rs2, rd) {
        regs[rd] = regs[rs1] + regs[rs2];
    }

    operation load(rs1, rd, imm) {
        // regs[rd] = mem[regs[rs1]+imm];
        var addr = regs[rs1]+imm;
        regs[rd] = mem.load(addr);
    }
}
```

Fig. 4. A simple platform

Example 1 (Simple platform model). Figure 4 illustrates a simple platform model. The model state variables consist of the register file and the memory. The model consists of operations `add` and `load`. The `load` operation accesses the memory through the `mem.load` function. Throughout our exposition, we add more detail to this model (e.g. a cache, branch predictor, etc.), leading up to the SAP model in §6.1. The modelling syntax loosely follows UCLID5 [40] which we use to implement the verification techniques discussed later.

3.2 Adversary Model

In this section, we characterize the *capabilities* of the adversary/attacker by defining a parameterized adversary model. The adversary model determines which behaviours constitute a vulnerability and hence influence the security specification. A common adversary model [10,31,47] is one that can passively observe and actively write to subsets of variables. Our approach additionally endows the adversary with the ability to transmit data between variables. This choice is motivated by microarchitectural mechanisms that move data between variables without necessarily making it visible, as illustrated in Example 2.

Example 2. In Fig. 5, we illustrate how a transmit operation can abstractly model the effect of adversary code in the case of the Lazy-FP [46] vulnerability. In Lazy-FP, the flow of information from the secret to cache state made possible by the adversary's capability to leak information from the `xmm` register to the general purpose register `rax` using the $\mathsf{adv}_{\mathit{flow}}(\mathtt{xmm}, \mathtt{rax})$ operation (and eventually the observable cache) indicated by the solid red arrows. The victim program enables information flow from the secret to the `xmm` register (indicated by the dotted line).

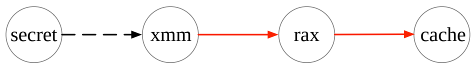

Fig. 5. Information flow in the Lazy-FP vulnerability.

Formally, we characterize our adversary as a triple, $\mathcal{A} = \langle V_O, V_T, F\rangle$. The components indicate the set of observable state variables $V_O \subseteq V_{\mathrm{L}}$, tamperable variables $V_T \subseteq V$ and a set of *transmitting pairs* $F \subseteq V \times V$. Observable variables determine when two states are considered to be distinguishable. The tampering operations $\mathsf{Op}_T = \{\mathsf{op}_{\mathit{tamp}}(v) \mid v \in V_T\}$ change the value of the tampered variable to an arbitrary value: $\delta(q, \mathsf{op}_{\mathit{tamp}}(v), q') \iff \exists x \in \mathbb{D}.\ q' = q[v \leftarrow x]$. Our adversary model augments these tampering operations from [32, 47] with transmitting operations $\mathsf{Op}_{\mathit{flow}} = \{\mathsf{adv}_{\mathit{flow}}(v_1, v_2) \mid (v_1, v_2) \in F\}$. A transmitting operation $\mathsf{adv}_{\mathit{flow}}(v_1, v_2)$ establishes a flow of data from v_1 to v_2: $\delta(q, \mathsf{adv}_{\mathit{flow}}(v_1, v_2), q') \iff q' = q[v_2 \leftarrow q(v_1)]$. These adversary operations are a subset of the complete operation set: $\mathsf{Op}_O \cup \mathsf{Op}_T \cup \mathsf{Op}_{\mathit{flow}} \subset \mathsf{Op}$. The adversary executes asynchronously with the system using interleaving semantics[3].

3.3 Security Properties

Observation. We base our security properties on a notion of observation that dictates when two states lead to different observations (e.g. timing/power-based side-channels [48]). Our instantiation of observation identifies a subset of variables $V_{\mathrm{L}} \subseteq V$ denoted as *low* variables. These are required to have equivalent values in the two states:

$$q_1 \approx_{V_{\mathrm{L}}} q_2 \doteq \forall v \in V_{\mathrm{L}}.\ q_1(v) = q_2(v) \tag{5}$$

In the context of an entire execution, the adversary-visible §3.2 should be tagged as low. Using this definition we define the standard non-interference property.

Definition 1 (Non-Interference). *A system M executing program P satisfies non-interference (w.r.t. low variables V_L) if*

$$\forall \pi_1, \pi_2 \in \mathit{Tr}_P(M).\ \pi_1^{(0)} \approx_{V_{\mathrm{L}}} \pi_2^{(0)} \Rightarrow \pi_1^{(n)} \approx_{V_{\mathrm{L}}} \pi_2^{(n)} \tag{6}$$

[3] Note that the adversary and system can take an arbitrary number of steps.

In words, this property requires that states which start out being observationally equivalent should end up being observationally equivalent after executing P.

We also consider observational determinism [55] which states that executing program P from indistinguishable states should result in indistinguishable states at every step. We first extend the definition of low-equivalence to traces: $\pi_1 \approx_{V_{\mathtt{L}}} \pi_2 \doteq \forall i \in \mathbb{N}.\ \pi_1^{(i)} \approx_{V_{\mathtt{L}}} \pi_2^{(i)}$. Observational determinism can be formalized as follows:

Definition 2 (Observational Determinism). *A system M executing program P satisfies observational-determinism (OD) w.r.t low variables $V_{\mathtt{L}}$ if*

$$\forall \pi_1, \pi_2 \in \mathit{Tr}_P(M).\ \pi_1^{(0)} \approx_{V_{\mathtt{L}}} \pi_2^{(0)} \Rightarrow \pi_1 \approx_{V_{\mathtt{L}}} \pi_2 \tag{7}$$

An extension of observational determinism [10] captures trace property-dependent violations of observational determinism. While OD requires that any two traces that are initially equivalent be always equivalent, Trace-Property Observational Determinism (TPOD) relaxes this condition in two ways. First, TPOD restricts these traces (π_3, π_4) to a trace set T_2. Secondly, TPOD only enforces their equality when two other traces (π_1, π_2) from trace set T_1 are equivalent. We refer the reader to [10] for details.

Definition 3 (TPOD). *Given trace properties $T_1 \subset \mathit{Tr}_P(M)$ and $T_2 \subset \mathit{Tr}_P(M)$, a system M satisfies trace property-dependent observational determinism when executing program P if*

$$\forall \pi_1, \pi_2 \in T_1 . \pi_3, \pi_4 \in T_2 . \big(\pi_1 \approx_{V_{\mathtt{L}}} \pi_2 \wedge \pi_3^{(0)} \approx_{V_{\mathtt{L}}} \pi_4^{(0)}\big) \Rightarrow \pi_3 \approx_{V_{\mathtt{L}}} \pi_4 \tag{8}$$

Taint contexts Taint analysis [44] is an approach that can perform an approximate (typically over-approximate) analysis of the system w.r.t. security properties. These approaches generally consider a set $\mathcal{L}$ of security labels [14, 41]. A *taint-context* Γ maps variables to taint-labels, $\Gamma : V \to \mathcal{L}$. When the label set consists of two labels low and high, $\{\mathtt{L}, \mathtt{H}\}$, we can view $\Gamma(v) = \mathtt{L}$ to mean that the variable v is untainted (by any high - $\mathtt{H}$ - values). Consequently, this interpretation of taint can be related to the notion of equivalence of variables. In particular, when the taint assigned to v is low at some point in the execution, $\Gamma(v) = \mathtt{L}$, v only depends on variables that had initially had $\mathtt{L}$ taint. If the latter were observationally equal then v is as well. Hence, the taint can be thought of as a relational property marking variables that take identical values across executions. We make use of this when developing the SymboTaint proof system. We can define indistinguishability in terms of the taint-context:

$$q_1 \approx_{\Gamma} q_2 \doteq \forall v \in V.\ \Gamma(v) = l \Rightarrow q_1(v) = q_2(v) \tag{9}$$

Eq. 5 relates to this as: $q_1 \approx_{\{v \mid \Gamma(v)=\mathtt{L}\}} q_2 \iff q_1 \approx_{\Gamma} q_2$. We also use δ_τ as the transition relation over taint-contexts: $\delta_\tau \subseteq Q \times (V \to \mathcal{L}) \times \mathsf{Op} \times (V \to \mathcal{L})$.

4 The SymboTaint Proof System

In this section, we develop a methodology to verify a given program running on a platform model with respect to security properties from §3.3. Our technique is a proof-system based on Hoare-logic [23] and interpolant-based reasoning [37]. Standard Hoare-logic inductively builds a proof for a program by composing Hoare-triples defined for atomic statements to get a pre-post condition for the full program. The pre-post conditions are typically state sets (predicates) that over-approximate the actual set of states reached. This is adequate when one is concerned with proving properties over individual executions. However, the security properties of interest (§3.3) are *hyperproperties* defined over pairs of executions. This requires us to augment the shape of the interpolants used in the proof-system, which we now discuss.

4.1 Joint Symbolic-Taint Interpolants

Our proof system uses interpolants of the form $\{S, \Gamma\}$ where S is a set of states and Γ is a collection of relation equality constraints. While standard Hoare-logic/interpolant-based approaches use formulae that hold over individual states, $\{S, \Gamma\}$ holds on *a pair of states*. This is defined through a judgement $(q_1, q_2) \models \{S, \Gamma\}$ (where $q_1, q_2 \in Q$):

$$(q_1, q_2) \models \{S, \Gamma\} \doteq (q_1 \in S) \wedge (q_2 \in S) \wedge q_1 \approx_\Gamma q_2$$

We define the SymboTaint-triple $\{S, \Gamma\}\ M(\mathsf{op})\ \{S', \Gamma'\}$ over an individual operation op similar to Hoare-logic. If two states satisfying $\{S, \Gamma\}$ transition on op, then any pair of post-states satisfy $\{S', \Gamma'\}$:

$$\begin{aligned}&\{S, \Gamma\}\ M(\mathsf{op})\ \{S', \Gamma'\} \doteq \forall q_1, q_2, q_1', q_2'.\\ &\qquad((q_1, q_2) \models \{S, \Gamma\} \wedge \delta(q_1, \mathsf{op}, q_1') \wedge \delta(q_2, \mathsf{op}, q_2')) \implies (q_1', q_2') \models \{S', \Gamma'\}\end{aligned}$$

We observe that the symbolic-taint interpolants *can be composed*, giving us proof rules for sequential composition and iteration similar to Hoare-logic. We provide the full set of proof rules in the Appendix. Then starting with the triple for a single operation as the base case, we can build proofs for larger programs. This key feature allows us to decompose proofs for large programs into a set of verification conditions (VC) over small sequences of operations. Each of these VCs can be discharged more effectively than a single VC for the full program.

Self-composition vs. SymboTaint. Hyper-properties such as non-interference can be thought of as (single trace) safety properties over the self-composition of the system [12]. Following this observation, one could build a self-composition of the system and use interpolants defined over two copies of the variables. However, this approach defines symbolic constraints on twice the variables, putting strain on the underlying model-checker. We develop a different route.

Our choice of interpolants is based on the observation that the properties in §3.3 mention equality over pairs of variables from the copies of the system. We encode this as a taint-context $\Gamma : V \to \mathcal{L}$ which marks whether a variable takes equal values. When $\Gamma(v) = \mathtt{L}$ then v takes the same value across both executions.

Example 3. In Fig. 6 we depict a snippet from a model extending Ex. 1. In this model, before a `load` operation invokes `mem.load` to fetch an address from the memory it checks for the address in the cache (`cache.is_hit`). The model performs cache partitioning [28] by dividing the cache into partitions each of which is only accessible

```
 1 core {
 2  ...
 3  operation load(rs1, rd, imm) {
 4    // mem[regs[rs1]+imm] = rd;
 5    var addr = regs[rs1]+imm;
 6    // Redefined model semantics
 7    if (cache.is_hit(addr, dom))
 8      regs[rd] = cache.load(addr, dom)
 9    else
10      regs[rd] = mem.load(addr);
11  }
12  ...
13 }
14
15 // Cache component
16 cache {
17  var mdata : [set_index_t]word_t;
18  var data : [set_index_t]word_t;
19  // Internal functions
20  get_index(addr, dom): set_index_t =
21    addr[63:22] ++ dom;
22  get_tag(addr) : tag_t = addr[21:8];
23  is_hit(addr,dom): bool = get_tag(addr
       )
24    == mdata[get_index(addr, dom)];
25 }
```

Fig. 6. Modified platform model from Fig. 4 with a partitioned cache.

by a single process domain (dom). The model partitions the cache by set-index (line 19); the index depends on the address and the process domain. Suppose the victim domain 0 is allocated indices $\leq k$ while the attacker domain 1 is allocated the rest.

An access made by the victim (domain 0) on address addr, results in the satisfaction of the Hoare-triple $\{Q, \Gamma_0\}\ M(\texttt{load})\ \{S_1, \Gamma_1\}$ where $\Gamma_0 = [\lambda i > k.i \leftarrow \texttt{L}]$ is the taint-context with domain 1 accessible indices marked low. Additionally, the symbolic state captures the fact that domain 0 is making the access. Then following the partitioning semantics encoded in $M(\texttt{load})$, we can infer a post-judgement where $\Gamma_1 = \Gamma_0$. That is after performing the load operation, domain 1 visible state remains low-equivalent. This proves that a victim executed load does not modify attacker-visible state.

4.2 Connecting the Proof System with Security Properties

In this section, we connect the proof system developed in §4.1 with security properties. This connection is based on the fact that we can develop SymboTaint-triples which, under some conditions, are sound with respect to the properties defined in §3.3. We now discuss these conditions, starting with non-interference.

Non-interference: When proving non-interference with respect to the low variables $V_{\texttt{L}}$ and program P, we generate a valid triple of the form: $\{I, \Gamma_0\}\ M(P)\ \{Q, \Gamma_f\}$ with the following condition, denoted as CondNI:

$$\text{CondNI}: \quad \forall v \notin V_{\texttt{L}}.\ \Gamma_0(v) \neq \texttt{L} \ \wedge\ \forall v \in V_{\texttt{L}}.\ \Gamma_f(v) = \texttt{L} \tag{10}$$

In the pre-condition, we allow a low ($\texttt{L}$) assignment to variables in $V_{\texttt{L}}$. This ensures that the antecedent of the non-interference property is implied. We require that the final taint-context Γ_f assign $\texttt{L}$ to $V_{\texttt{L}}$, which implies the consequent of non-interference.

Observational-determinism: The difference between observational determinism and non-interference is that in the former, all taint-contexts must assign $\texttt{L}$ to $V_{\texttt{L}}$. Hence, in this case, we want to generate valid triples where CondObsDet holds:

$$\{I, \Gamma_0\}\ M(\mathsf{op}_1)\ \{S_1, \Gamma_1\}\ M(\mathsf{op}_2)\ \cdots \{S_n, \Gamma_n\}$$
$$\text{CondObsDet}: \forall v \notin V_{\texttt{L}}.\ \Gamma_0(v) \neq \texttt{L} \ \wedge\ \forall i \in [1..n].\ \forall v \in V_{\texttt{L}}.\ \Gamma_i(v) = \texttt{L} \tag{11}$$

The first judgement (pre-condition) is identical to the case of non-interference. However, in this case, we need to choose the intermediate interpolants such that each of the taint-contexts assign the $\mathtt{L}$ label to the variables in $V_{\mathtt{L}}$.

TPOD: While non-interference and observational determinism are properties over two traces, TPOD is over four traces. This requires us to consider a *self-composition* of the platform transition system. Additionally, TPOD only enforces observational-equivalence when the traces belong to T_1 and T_2. This allows us to strengthen the proof system for TPOD with auxiliary invariants that over-approximate T_1 and T_2. We call these invariants *cover-invariants*. We now briefly discuss these concepts.

Self composition of M The self-composition of M is the transition system $M^2 = \langle Q^2, \delta^2, I^2 \rangle$. The new state space is $Q^2 = Q \times Q$, the transition relation δ^2 enforces δ on both the first and second copies of the state, and $I^2 = I \times I$. We also consider the paired state as an assignment to two copies of variables: $V^1 = \{v^1\}_{v \in V}$ and $V^2 = \{v^2\}_{v \in V}$.

Cover invariants for trace-properties In order to capture the trace-properties T_1, T_2 that TPOD enforces, we allow invariants $I_1^{tpod}, I_2^{tpod} \subseteq Q$ that are implied by T_1 and T_2 respectively. That is, if $\pi \in T_1$ then $\forall i.\ \pi^i \in I_1^{tpod}$, and similarly for T_2 and I_2^{tpod}. While cover invariants can just be *True*, tighter invariants can lead to stronger proofs.

For TPOD, we require the following valid triples to hold over the self-composed system M^2 and program $P = \mathsf{op}_1 \cdots \mathsf{op}_n$ such that CondTPOD holds:

$$\{S_0, \Gamma_0\}\ M(\mathsf{op}_1)\ \{S_1, \Gamma_1'\}, \{S_1, \Gamma_1\}\ M(\mathsf{op}_2)\ \{S_2, \Gamma_2'\}, \cdots, \{S_{n-1}, \Gamma_{n-1}\}\ M(\mathsf{op}_n)\ \{S_n, \Gamma_n'\}$$

$$\begin{aligned} &\mathrm{CondTPOD} : S_0 = (I \cap I_1^{tpod}) \times (I \cap I_2^{tpod})\ \wedge \qquad (12)\\ &\quad \forall i.\ \forall v \notin V_{\mathtt{L}}^1.\ \Gamma_i(v) \neq \mathtt{L}\ \wedge\ \forall v \notin V_{\mathtt{L}}^2.\ \Gamma_0(v) \neq \mathtt{L}\ \wedge \\ &\quad \forall i \in [1..n].\ \forall v \in V_{\mathtt{L}}^2.\ \Gamma_i'(v) = \mathtt{L}\ \wedge\ \forall i \in [1..(n-1)].\ \forall v \in V^2.\ \Gamma_i(v) = \Gamma_i'(v) \end{aligned}$$

Theorem 1 (Soundness). *If there is a Hoare-triple for M under program P that is valid w.r.t. conditions* CondNI *(resp.* CondObsDet*,* CondTPOD*) then the system M satisfies non-interference (resp. observational-determinism, TPOD) on program P.*

5 IFSMs: Operational Encoding of SymboTaint

In this section, we discuss an operational approach to encode the proof-based reasoning developed in §4. This allows us to represent proofs in the form of executions of a standard symbolic transition system. This has the following prominent advantages. Firstly, it is easier to connect (by way of instrumentation) an operationally encoded proof with a platform model that is represented as a transition system. Then, off-the-shelf model-checking tools can be used to perform verification on the proof-instrumented platform model (e.g. bounded/unbounded model checking, invariant inference, etc.). We apply this to secure and insecure cases in §7 where we analyze an abstract platform model. Additionally, an operational encoding allows us to perform structural composition (§5.2) of parts of the platform. In particular, this allows projecting away components that are not relevant to the proof of a certain property. This is advantageous since platforms (over which we evaluate our techniques) are built hierarchically.

5.1 Information Flow State Machine

At a high level, an IFSM is a state-transition system that encodes a joint symbolic-taint ($\{S, \mathit{\Gamma}\}$) judgement in its state. This encoding is performed by augmenting the state from the system §3.1 with a taint-context ($\mathit{\Gamma} : V \to \mathcal{L}$). Consequently, IFSM creation can be thought of as instrumenting a platform model with taint-tracking variables.

The transitions of the IFSM update the system state following the transition relation δ from §3.1, and update the taint-context following its transition relation, δ_τ. If a transition is allowed in the IFSM, then the corresponding Hoare-triple holds in the proof system of §4. This key feature implies the soundness of safety proofs that use IFSMs. The IFSM is also parameterized by the initial taint-context, $\mathit{\Gamma}_0$. The choice of initial taint context depends on the property being proved. For example, if one is concerned with proving non-interference with $V_{\mathtt{L}}$ as the set of low variables, the initial taint-context assigning $\mathtt{H}$ to all non-low variables provides the correct antecedent: $\mathit{\Gamma}_0 = [V_{\mathtt{L}} \to \mathtt{L}, V_{\mathtt{H}} \to \mathtt{H}]$. We now formally define an IFSM.

Definition 4 (Information Flow State Machine). *An IFSM is a transition system $\langle \mathsf{Q}, \mathit{\Delta}, \mathsf{I} \rangle$, with a set of configurations Q, the transition relation $\mathit{\Delta}$, and a set of initial states I. Each configuration pairs a platform state with a taint context: $\mathsf{Q} = Q \times (V \to \mathcal{L})$. The transition relation combines the platform variable updates with taint transitions: $\mathit{\Delta}((q_1, \mathit{\Gamma}_1), \mathsf{op}, (q_2, \mathit{\Gamma}_2)) \iff \delta(q_1, \mathsf{op}, q_2) \wedge \delta_\tau((q_1, \mathit{\Gamma}_1), \mathsf{op}, \mathit{\Gamma}_2)$. where δ is the platform transition relation and δ_τ is the taint-context transition relation. Finally the set of initial configurations is defined as: $\mathsf{I} = I \times \mathit{\Gamma}_0$.*

5.2 Composing IFSMs

In this section, we discuss structural compositionality of IFSMs. Our notion of composition is based on the observations that (a) hardware platforms are typically hierarchical in nature and (b) only some components in the design hierarchy transition for certain operations. We define a composition of two IFSMs as follows.

Definition 5 (Composition of IFSMs). *The composition of $M_1 = \langle \mathsf{Q}, \mathit{\Delta}_1, \mathsf{I}_1 \rangle$ and $M_2 = \langle \mathsf{Q}, \mathit{\Delta}_2, \mathsf{I}_2 \rangle$ is $M_1 || M_2 = \langle \mathsf{Q}, \mathit{\Delta}, \mathsf{I} \rangle$, where: $\mathit{\Delta} = \mathit{\Delta}_1 \wedge \mathit{\Delta}_2$ and $I = I_1 \wedge I_2$.*

The composition conjoins transition relations and starting states of M_1 and M_2. The new transition relation enforces constraints from both component IFSMs. This may lead to the new IFSM not having any valid transitions (e.g. when M_1, M_2 require conflicting updates to a variable). We identify the conditions under which this does not happen.

Separability. To allow composability, we require that the overall transition relation is *separable* into per-variable components: $\delta(q, \mathsf{op}, q') \iff \wedge_v \, \delta^v(q, \mathsf{op}, q'(v))$. Intuitively, the relation $\delta^v \subseteq Q \times \mathsf{Op} \times \mathbb{D}$ localizes the effect of δ on an individual variable v. The post-values admitted by all individual δ^vs can be combined to generate valid next-state assignments. We define δ^v_τ similarly, by replacing δ with δ_τ, q with $(q, \mathit{\Gamma})$, and q' with $\mathit{\Gamma}'$ in the above equation: $\delta_\tau((q, \mathit{\Gamma}), \mathsf{op}, \mathit{\Gamma}') \iff \bigwedge_v \, \delta^v_\tau((q, \mathit{\Gamma}), \mathsf{op}, \mathit{\Gamma}'(v))$.

Projection. In addition to separating the effects of a transition, we also need to identify when a particular component drives a certain variable. We denote this by the *guard predicate* $C^v(q, \mathsf{op})$ which is true when v is driven by the component and false otherwise. Formally, we can write this as follows:

$$C^v(q, \mathsf{op}) \lor \forall x \in \mathbb{D}.\ \delta^v(q, \mathsf{op}, x) \tag{13}$$

When C^v is true, the component enforces specific next values for variable v. Otherwise, δ^v holds for all next values, i.e. the component does not enforce any constraints on the new value. When composing two or more components, we require that at each step the guard of at most one component be true. This ensures non-conflicting updates.

5.3 Compositional Verification with IFSM

Compositionality allows us to view the full platform model as a set of separable constraints imposed on each variable. We can make full use of this separability, by composing constraints from only necessary models. Consider the task of verifying the composition $M = M_1||...||M_N$. Then it suffices to compose only those components that drive some variable v. For operation op, we denote such components as $\mathbf{I}(\mathsf{op}) \doteq \{i \in [N] \mid \exists q \in Q, v \in V.\ C_i^v(q, \mathsf{op})\}$. Then we have the following corollary.

Corollary 1 (Minimal Composition). *If each IFSM of a composition satisfies Eq. 13 and the guards of each IFSM model are disjoint, we have (where* $\mathbf{I}(\mathsf{op}) = \{j_1, \cdots, j_k\}$*):*

$$\{S, \Gamma\}\ M(\mathsf{op})\ \{S', \Gamma'\} \iff \{S, \Gamma\}\ (M_{j_1}||M_{j_2}||\cdots||M_{j_k})\,(\mathsf{op})\ \{S', \Gamma'\} \tag{14}$$

In the following sections, we utilize this notion of minimal composition when verifying transient execution attacks on platform model of a microprocessor.

6 Verifying Speculative Platforms with IFSMs

Existing works on modeling and verifying security properties for processor platforms commonly develop the platform model based on *speculative semantics* [10, 16, 21, 22]. These platform models specify instruction semantics at the microarchitectural-level. These mappings from instructions to micro-architectural effects are often hard-coded making extensions to these semantics difficult. A modeling language that allows flexible specification of micro-architectural features can address these challenges. We present Speculative Abstract Platform (SAP) that models a general class of microprocessor designs. The SAP model is implemented using the modeling and verification language UCLID5 [40] which allows for parameterization and composition of model components. While we model a wide range of microarchitectural features, for space reasons, we only present the cache and branch prediction components of the SAP model in this section. For full details, we refer the reader to the repository at `https://github.com/ifsm-sp2023/sap` and Appendix §B.

6.1 The Speculative Abstract Platform

The SAP model consists of several abstract components represented as IFSMs. The model includes a CPU core, a data cache, a line fill buffer, load and store buffers, a page table, a translation lookaside buffer, a pattern history table, a branch target buffer and a power state. We note that the SAP model serves as an initial abstraction which one may use for the analysis of speculative programs. To exemplify our methodology, we describe the following three IFSM components of the SAP in more detail and describe their composition: the CPU, cache and branch predictor models. We omit details about virtualization from this example for simplicity.

Model	**State Var.**	Description
CPU	`pc`	The program counter.
	`regs`	Registers.
	`mem`	Physical memory.
	`excp`	Exception register.
	`pid`	Current executing process.
Cache	`cache_valid`	Cache index to valid bit.
	`cache_tag`	Cache index to entry tag.
Branch prediction	`pht`	Pattern history table.
	`btb`	Branch target buffer.

Fig. 7. CPU, cache, and branch prediction components of the SAP model.

Abstract CPU model. The CPU model M_{core}, is an abstraction of architectural states, which include the variables in the first row of Table 7. The program counter, registers, physical memory, and exception registers[4] are denoted by `pc`, `regs`, `mem`, `excp` respectively. We use `pid` to denote the domain of the executing process (adversary or victim). The set of operations (Op) define the transition relation semantics δ. For the CPU, these operations are load (`load`), store (`store`), conditional branch (`bge`), add (`add`), jump (`jmp`) and other instructions containing only the ISA level semantics. This model was described earlier as Fig. 4.

Abstract cache model. The second row in Table 7 describes the cache model M_{cache}, which contains a map of cache indexes to valid bits and tags of cache lines (this is abstracted away in Figure 6). In addition to the internal functions as shown in Figure 6, the cache model contains the `load` operation which reads from `data` variable. The cache model redefines the semantics of the CPU's `load` operation and thus we associate the guard $C^{\texttt{regs[rd]}}(q, \texttt{load}) :=$ `is_hit(addr,pid)`, where $\texttt{addr} := \texttt{q.regs[rs1]} + \texttt{imm}$. Fig. 8 expands on Fig. 6 to illustrate this.

```
// Cache model continued
cache {
  ...
  operation load(addr, dom) {
    return data[get_index(addr, dom)];
  }
  guard load(addr, dom) {
    return is_hit(addr, dom);
  }
}
```

Fig. 8. Continued cache model from Fig. 6 with the load operation and guard.

[4] We note that the number of exceptions is not comprehensive and only includes page faults, abort pages and device-not-available exceptions to accommodate the variants we verify.

Abstract branch predictor model. The abstract branch predictor M_{BP}, redefines branch instructions with speculative semantics by using a pattern history table (pht). It takes the conventional *always mispredict* semantics common in existing models [16, 21]. For example, the branch-if-greater-equal (bge) operation makes a branch decision based on the pht using an uninterpreted function ctr_to_dir. This takes as argument the state of the counter for a given address and returns a branch direction. The guard associated to this operation, $C^{pc}_{BP}(q, \texttt{bge}) := *$, allows the model to non-deterministically choose (indicated by $*$) between executing bge in the branch_prediction or the core.

```
// Branch prediction
branch_prediction {
  var pht : [addr_t]counter_state_t;

  // internal functions
  ctr_to_dir(ctr_state) : boolean;
  predict_dir(addr) : boolean =
    ctr_to_dir(pht[addr]);
  ...

  operation bge(rs1, rs2, addr) {
    // speculatively predict branch
    if (predict_dir(pc)) {
      pc = pc + 4;
    } else {
      pc = addr;
    }
  }

  guard bge() { return *; }
  ...
}
```

Fig. 9. Branch prediction in the SAP model.

6.2 Composing IFSM Models.

Composing SAP components. To verify our properties efficiently, we compose the abstract models from §6.1 using Def. 5. In addition, the models are designed such that the transitions satisfy Eq. 13 and the collection of guards are disjoint. Subsequently, we make use of Cor. 1 to verify "*minimal*" compositions with the interpolant-based approach. To illustrate this, consider the task of verifying non-interference for Figure 2. Figure 10 shows the program component that contains three operations corresponding to the blocks in Figure 2. Each instruction level operation within the block corresponds to a composition of the operations from M_{core}, M_{cache} and M_{BP}.

Executing the victim program on the SAP model. First, by construction, our models M_{core}, M_{cache} and M_{BP} satisfy Eq. 13 because when the guard of a variable v evaluates to false, we do not update variable v. Second, each guard is written to be disjoint. This allows us to use Corollary 1 to compose only the necessary models for computing the symbolic-taint interpolants. For example, computing the post-interpolant for block_C1(), the composition $M_{core}||M_{cache}||M_{BP}$ is used because of the store and bge instructions. On the otherhand, computing the post-interpolant of block_C3() only requires $M_{core}||M_{cache}$. This results in a shorter compilation time of the models for each computation of an interpolant and smaller model to reason about.

```
// Branch prediction
program {
  operation block_C1() {
    addi(a3, gp, -88);
    store(a3, s0, -24);
    bge(a0, a1, END);
  }
  ...
  operation block_C3() {
    ...
    load(a3, a3, 0);
    add(a4, a4, a3);
    load(a5, a4, 0);
  }
}
```

Fig. 10. Program composed with the SAP model.

7 Case Studies

While several approaches [10,21,38] perform verification w.r.t. secure speculation, vulnerabilities such as ÆPIC [7] transcend secure speculation, due to which we check observational determinism (Eq. 11), which in turn implies secure speculation.

Transient execution attacks and their adversary model parameterization.

	Execution		V_O		V_T							F		Spec. Feature Exploited				
Vulnerability	***Asynchronous***	***Entry Points***	***L1D***	***AVX2 Power***	***PHT***	***RSB/BTB***	***Page Tables***	***Store Buffer***	***L1D Cache***	***Load Port***	***LFB***	v_1	v_2	***Branch Pred.***	***BTB/RSB Pred.***	***STL Forward***	***Reg. Perm.***	***Mem. Perm.***
Spectre v1 [29]		×	×		×				×					×				
Spectre v2 [29]		×	×			×			×						×			
Spectre v4 [29]		×	×					×	×							×		
Meltdown [34]		×	×						×									×
Foreshadow [51]		×	×				×		×									×
LVI [52]		×	×		×	×	×	×	×	×	×					×		×
NetSpectre [45]		×	×	×	×				×					×				
LazyFP [46]		×	×						×			`XMM`	`L1D`				×	
RIDL [42]	×		×						×	×	×	`LFB`	`L1D`				×	

Table 1. The execution column indicates whether the adversary is allowed to execute asynchronously or only before and after the victim program (at entry points). V_O, V_T and F represent the observable states, tamperable states and the flow transmit pairs (§3.2) of the adversary.

7.1 Speculative Examples

Table 1 shows the list of transient execution attacks that we check for observational determinism when executing on the SAP model. The execution column indicates the setting in which the vulnerabilities may occur: either when the adversary executes asynchronously (e.g., using simultaneous multi-threading) or only before and after the victim program execution (entry points). We try to choose asynchronous execution whenever possible because it is a more restrictive model. The three columns V_O, V_T and F describe our parameterization of the adversary model (§3.2) used to capture the attacks. V_O shows two covert channels that are used in the list of attacks, which include the L1D cache and the AVX2 power state. V_T indicates the states that we consider tamperable. Lastly, F describes the adversary's capability to leak information from the state in column v_1 to the state in column v_2. For example, in the case of the LazyFP vulnerability, we assume the setting in which the adversary can leak information from the XMM registers to the L1D cache, and for the MDS-based attacks, we assume that the adversary can leak information from the line fill buffer (`LFB`) to the L1D cache. The remaining column indicates the speculative features being exploited at a high level. We note that these columns are not intended to be exhaustive of the speculative features, buffers and covert channels that can potentially be exploited. We emphasize that the table stresses the need for a more holistic approach that considers all components of the platform and an adversary model that can parameterize the system state. Each vulnerability shown only exploits a specific combination of speculative features and covert channels, yet there is potentially a combinatorial space of features one could exploit and thus sound analysis would require reasoning about all these combinations in a scalable manner.

Verifying Observational Determinism on the SAP Model						
Vulnerability	**OD** (BMC)		**TOD** (BMC)		**TOD** (Interpolants)	
	Insecure	Part. $	Insecure	Part. $	Insecure	Part. $
Spectre v1	150.7	125.9	7.1	35.3	5.2	6.2
Spectre v2	223.0	242.8	8.2	25.3	4.2	4.3
Spectre v4	20.8	49.2	5.6	6.872	4.2	4.0
Meltdown	12.7	92.1	4.1	4.1	3.8	3.7
Foreshadow	11.9	81.7	4.5	9.3	3.9	3.8
LVI	15.1	33.6	7.0	5.1	4.0	4.3
NetSpectre	17.6	-	7.0	-	3.7	-
LazyFP	6	-	3.7	-	3.8	-
RIDL	14.37	20.2	4.3	3.5	4.3	3.9
Spectre v1 (St.=4)	TLE	TLE	94.7	155.0	4.8	4.3
Spectre v1 (St.=5)	TLE	TLE	466.2	297.2	5.8	4.7
Spectre v1 (St.=6)	TLE	TLE	874.8	TLE	6.1	5.7

Table 2. Time (in seconds) to verify OD using the 2-safety encoding with BMC, the trace property encoding of OD (TOD) with BMC and TOD with interpolants. Examples are checked using UCLID5, and marked with TLE (time limit exceeded) if it takes longer than 15 minutes.

7.2 Verification Results

We model instruction-level program snippets representative of the transient execution attack vulnerabilities from Table 1 and verify the programs by composing it with the SAP. The results of the approaches used are described in Table 2. Specifically, we first verify that the program snippets violate the secure speculation property using (1) bounded model checking (BMC) with the 2-safety observational determinism-based encoding (OD), (2) using bounded model checking with the symbolic taint analysis (TOD) encoding (which is a trace property with taint contexts modeled using ghost variables as explained in §5.1) and (3) using the interpolant-based approach with the symbolic taint analysis encoding. For the interpolant-based approach, we prove pre-post properties corresponding to the taint context. These results are listed under the *insecure* columns. We also verify that the programs are secure with an abstract partitioned cache [28] model, with the exception of NetSpectre and LazyFP because the model of the former leaks to the AVX2 side-channel and the latter leaks secrets into cache after the execution of the victim. These results are listed under the columns labeled *Part. $*. We note that the disparity in runtimes between the different vulnerabilities can be explained by the number of atomic blocks we separate the program P into (and hence require more steps for BMC), the number of instructions and varying platform model complexity. The last three examples are extensions of the Spectre v1 attack where the system takes a varying number of steps (annotation (St.=i) means the system takes i steps). As expected, the interpolant-based approach outperforms BMC because each check is localized to small sequences of instructions. The experiments were run on a 2.6 GHz 6-Core Intel Core i7 machine with 16 GB RAM.

8 Related Work

Our main contribution in this work is demonstrating two forms of composition: (a) temporal composition which builds Hoare-style proofs for sequences of instructions, and (b) spatial composition which allows reasoning only over relevant slices of the hardware design. This is most closely related to work on information flow checking; more specifically, lazy self-composition [54]. The SymboTaint proof system used in temporal composition combines symbolic state with taint-based relational atoms. Lazy self-composition develops an abstraction-refinement approach, also using relational atoms. However, their core focus is on performing symbolic reasoning lazily (by default relying only on taint-based relational atoms). In contrast, our focus is on proof decomposition. Consequently, a lazy self-composition-based approach can be used with ours for identifying optimal interpolants. Additionally, lazy self-composition does not consider the modeling and compositional reasoning of the hardware platform.

Other related works that combine program and platform models to verify security include Covern [39] which defines composition over a specific type of shared resource system with locks, compositional information flow-aware refinement [5] which introduces the notion of ignorance-preservation, is developed over an abstract system which can be used our formalisms to develop more accurate models. Lastly, work on modeling hardware platforms using happens-before graphs [38, 50] proposes a pattern-based approach for checking security, however, non-interference is beyond the scope of this work.

The emergence of transient execution attacks has also led to the use of information flow checking for proving the security of programs [20–22], but they are limited in their ability to extend to different attacker and platform models and lack a systematic method of spatial composition. While previous work [16] provides a systematic approach to combine speculative attacks, they are attack-centric and retrospective, requiring knowledge about the precise attack mechanisms, and are limited in their ability to combine attacks.

9 Conclusion

In this work, we considered the problem of verifying information-flow-based security properties for software running on hardware platforms. This is challenging owing to complex microarchitectural-level system models and vulnerable code fragments nestled within large software. We introduced *SymboTaint*, a proof-system that specializes Hoare-style reasoning to properties such as non-interference and observational determinism. We developed *Information Flow State Machines* as an operational framework that allows parameterizable modeling of microarchitectural features. Additionally, IFSMs allow instrumenting the platform model with *SymboTaint* based proofs. We presented an abstract model of a speculative microprocessor called the *Speculative Abstract Platform* (SAP) with several microarchitectural features. We use our methodology to verify observational- determinism for a broad class of transient execution attacks beyond what is possible with existing approaches.

References

1. ARM TrustZone. https://www.arm.com/products/security-on-arm/trustzone (2013)
2. Intel Trust Domain Extensions. https://www.intel.com/content/dam/develop/external/us/en/documents/tdx-whitepaper-v4.pdf (2020)
3. Bachrach, J., Vo, H.D., Richards, B.C., Lee, Y., Waterman, A., Avizienis, R., Wawrzynek, J., Asanović, K.: Chisel: Constructing hardware in a scala embedded language. DAC Design Automation Conference 2012 pp. 1212–1221 (2012)
4. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org (2016)
5. Baumann, C., Dam, M., Guanciale, R., Nemati, H.: On compositional information flow aware refinement. In: 2021 IEEE 34th Computer Security Foundations Symposium (CSF). pp. 1–16 (2021). https://doi.org/10.1109/CSF51468.2021.00010
6. Borrello, P., Kogler, A., Schwarzl, M., Lipp, M., Gruss, D., Schwarz, M.: ÆPIC Leak: Architecturally leaking uninitialized data from the microarchitecture. In: 31st USENIX Security Symposium (USENIX Security 22) (2022)
7. Borrello, P., Kogler, A., Schwarzl, M., Lipp, M., Gruss, D., Schwarz, M.: ÆPIC Leak: Architecturally leaking uninitialized data from the microarchitecture. In: 31st USENIX Security Symposium (USENIX Security 22) (2022)
8. Canella, C., Bulck, J.V., Schwarz, M., Lipp, M., von Berg, B., Ortner, P., Piessens, F., Evtyushkin, D., Gruss, D.: A systematic evaluation of transient execution attacks and defenses. In: 28th USENIX Security Symposium (USENIX Security 19). pp. 249–266. USENIX Association, Santa Clara, CA (Aug 2019), https://www.usenix.org/conference/usenixsecurity19/presentation/canella
9. Canella, C., Genkin, D., Giner, L., Gruss, D., Lipp, M., Minkin, M., Moghimi, D., Piessens, F., Schwarz, M., Sunar, B., Van Bulck, J., Yarom, Y.: Fallout: Leaking data on meltdown-resistant cpus. In: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS). ACM (2019)
10. Cheang, K., Rasmussen, C., Seshia, S., Subramanyan, P.: A formal approach to secure speculation. In: 2019 IEEE 32nd Computer Security Foundations Symposium (CSF). pp. 288–28815 (2019). https://doi.org/10.1109/CSF.2019.00027
11. Cimatti, A., Griggio, A., Mover, S., Tonetta, S.: Infinite-state invariant checking with ic3 and predicate abstraction. Formal Methods in System Design **49**, 190–218 (2016)
12. Clarkson, M.R., Schneider, F.B.: Hyperproperties. In: Proceedings of the 2008 21st IEEE Computer Security Foundations Symposium. p. 51–65. CSF '08, IEEE Computer Society, USA (2008). https://doi.org/10.1109/CSF.2008.7, https://doi.org/10.1109/CSF.2008.7
13. Costan, V., Devadas, S.: Intel sgx explained. Cryptology ePrint Archive, Report 2016/086 (2016)
14. Denning, D.E.: A lattice model of secure information flow. Commun. ACM **19**, 236–243 (1976)
15. Dillig, I., Dillig, T., Li, B., McMillan, K.L.: Inductive invariant generation via abductive inference. Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications (2013)
16. Fabian, X., Patrignani, M., Guarnieri, M.: Automatic detection of speculative execution combinations. In: Proceedings of the 29th ACM Conference on Computer and Communications Security. CCS 2022, ACM (2022)

17. Feng, E., Lu, X., Du, D., Yang, B., Jiang, X., Xia, Y., Zang, B., Chen, H.: Scalable memory protection in the PENGLAI enclave. In: 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21). pp. 275–294. USENIX Association (Jul 2021), https://www.usenix.org/conference/osdi21/presentation/feng
18. Ferraiuolo, A., Baumann, A., Hawblitzel, C., Parno, B.: Komodo: Using verification to disentangle secure-enclave hardware from software. In: Proc. of Symposium on Operating Systems Principles (SOSP) (2017)
19. Goguen, J.A., Meseguer, J.: Unwinding and inference control. 1984 IEEE Symposium on Security and Privacy pp. 75–75 (1984)
20. Guanciale, R., Balliu, M., Dam, M.: Inspectre: Breaking and fixing microarchitectural vulnerabilities by formal analysis. In: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security. p. 1853–1869. CCS '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3372297.3417246, https://doi.org/10.1145/3372297.3417246
21. Guarnieri, M., Köpf, B., Morales, J.F., Reineke, J., Sánchez, A.: Spectector: Principled detection of speculative information flows. In: 2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020. pp. 1–19. IEEE (2020)
22. Guarnieri, M., Köpf, B., Reineke, J., Vila, P.: Hardware-software contracts for secure speculation. In: 2021 IEEE Symposium on Security and Privacy (SP). pp. 1868–1883 (2021). https://doi.org/10.1109/SP40001.2021.00036
23. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**, 576–580 (1969)
24. Hu, G., He, Z., Lee, R.B.: Sok: Hardware defenses against speculative execution attacks. In: 2021 International Symposium on Secure and Private Execution Environment Design (SEED). pp. 108–120 (2021). https://doi.org/10.1109/SEED51797.2021.00023
25. Ikarashi, Y., Bernstein, G.L., Reinking, A., Genç, H., Ragan-Kelley, J.: Exocompilation for productive programming of hardware accelerators. Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (2022)
26. Kaplan, D.: AMD SEV-ES. http://support.amd.com/TechDocs/ProtectingVMRegisterStatewithSEV-ES.pdf (2017)
27. Kaplan, D., Powell, J., Woller, T.: http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf (2016)
28. Kiriansky, V., Lebedev, I., Amarasinghe, S., Devadas, S., Emer, J.: Dawg: A defense against cache timing attacks in speculative execution processors. In: 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). pp. 974–987 (2018). https://doi.org/10.1109/MICRO.2018.00083
29. Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., Yarom, Y.: Spectre attacks: Exploiting speculative execution. In: 2019 IEEE Symposium on Security and Privacy (SP). pp. 1–19 (2019). https://doi.org/10.1109/SP.2019.00002
30. Koul, K., Melchert, J., Sreedhar, K., Truong, L., Nyengele, G., Zhang, K., Liu, Q., Setter, J., Chen, P.H., Mei, Y., Strange, M., Daly, R.G., Donovick, C., Carsello, A., Kong, T., Feng, K., Huff, D., Nayak, A., Setaluri, R., Thomas, J.J., Bhagdikar, N., Durst, D., Myers, Z., Tsiskaridze, N., Richardson, S., Bahr, R., Fatahalian, K., Hanrahan, P., Barrett, C.W., Horowitz, M., Torng, C., Kjolstad, F., Raina, P.: Aha: An agile approach to the design of coarse-grained reconfigurable accelerators and compilers. ACM Transactions on Embedded Computing Systems (TECS) (2022)
31. Kozyri, E., Chong, S., Myers, A.C.: Expressing information flow properties. Foundations and Trends in Privacy and Security **3**(1), 1–102 (2022). https://doi.org/10.1561/3300000008, https://doi.org/10.1561/3300000008

32. Lee, D., Cheang, K., Thomas, A., Lu, C., Gaddamadugu, P., Vahldiek-Oberwagner, A., Vij, M., Song, D., Seshia, S.A., Asanovic, K.: Cerberus: A formal approach to secure and efficient enclave memory sharing. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. p. 1871–1885. CCS '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3548606.3560595, https://doi.org/10.1145/3548606.3560595
33. Lee, D., Kohlbrenner, D., Shinde, S., Asanović, K., Song, D.: Keystone: An open framework for architecting trusted execution environments. In: Proceedings of the Fifteenth European Conference on Computer Systems. EuroSys '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3342195.3387532, https://doi.org/10.1145/3342195.3387532
34. Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., Hamburg, M.: Meltdown: Reading kernel memory from user space. In: 27th USENIX Security Symposium (USENIX Security 18) (2018)
35. Lyndon, R.: An interpolation theorem in the predicate calculus. Pacific Journal of Mathematics **9**, 129–142 (1959)
36. McMillan, K.L.: Symbolic model checking. In: International Conference on Computer Aided Verification (1993)
37. McMillan, K.L.: Interpolation and sat-based model checking. In: International Conference on Computer Aided Verification (2003)
38. Mosier, N., Lachnitt, H., Nemati, H., Trippel, C.: Axiomatic hardware-software contracts for security. In: Proceedings of the 49th Annual International Symposium on Computer Architecture. p. 72–86. ISCA '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3470496.3527412, https://doi.org/10.1145/3470496.3527412
39. Murray, T., Sison, R., Engelhardt, K.: Covern: A logic for compositional verification of information flow control. In: 2018 IEEE European Symposium on Security and Privacy (EuroS&P). pp. 16–30 (2018). https://doi.org/10.1109/EuroSP.2018.00010
40. Polgreen, E., Cheang, K., Gaddamadugu, P., Godbole, A., Laeufer, K., Lin, S., Manerkar, Y.A., Mora, F., Seshia, S.A.: Uclid5: Multi-modal formal modeling, verification, and synthesis. In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification. pp. 538–551. Springer International Publishing, Cham (2022)
41. Sandhu, R.S.: Lattice-based access control models. Computer **26**, 9–19 (1993)
42. van Schaik, S., Milburn, A., Österlund, S., Frigo, P., Maisuradze, G., Razavi, K., Bos, H., Giuffrida, C.: RIDL: Rogue in-flight data load. In: S&P (May 2019)
43. van Schaik, S., Minkin, M., Kwong, A., Genkin, D., Yarom, Y.: Cacheout: Leaking data on intel cpus via cache evictions. 2021 IEEE Symposium on Security and Privacy (SP) pp. 339–354 (2020)
44. Schwartz, E.J., Avgerinos, T., Brumley, D.: All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). 2010 IEEE Symposium on Security and Privacy pp. 317–331 (2010)
45. Schwarz, M., Schwarzl, M., Lipp, M., Gruss, D.: Netspectre: Read arbitrary memory over network. ArXiv **abs/1807.10535** (2018)
46. Stecklina, J., Prescher, T.: Lazyfp: Leaking FPU register state using microarchitectural side-channels. CoRR **abs/1806.07480** (2018), http://arxiv.org/abs/1806.07480
47. Subramanyan, P., Sinha, R., Lebedev, I., Devadas, S., Seshia, S.A.: A formal foundation for secure remote execution of enclaves. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. p. 2435–2450. CCS '17, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3133956.3134098, https://doi.org/10.1145/3133956.3134098

48. Szefer, J.: Survey of microarchitectural side and covert channels, attacks, and defenses. Journal of Hardware and Systems Security pp. 1–16 (2016)
49. Szefer, J.: Survey of microarchitectural side and covert channels, attacks, and defenses. Journal of Hardware and Systems Security pp. 1–16 (2018)
50. Trippel, C., Lustig, D., Martonosi, M.: Checkmate: Automated synthesis of hardware exploits and security litmus tests. In: 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). pp. 947–960 (2018). https://doi.org/10.1109/MICRO.2018.00081
51. Van Bulck, J., Minkin, M., Weisse, O., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Wenisch, T.F., Yarom, Y., Strackx, R.: Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In: Proceedings of the 27th USENIX Security Symposium. USENIX Association (August 2018), see also technical report Foreshadow-NG
52. Van Bulck, J., Moghimi, D., Schwarz, M., Lipp, M., Minkin, M., Genkin, D., Yuval, Y., Sunar, B., Gruss, D., Piessens, F.: LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In: 41th IEEE Symposium on Security and Privacy (S&P'20) (2020)
53. Weisse, O., Van Bulck, J., Minkin, M., Genkin, D., Kasikci, B., Piessens, F., Silberstein, M., Strackx, R., Wenisch, T.F., Yarom, Y.: Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution. Technical report (2018), see also USENIX Security paper Foreshadow
54. Yang, W., Vizel, Y., Subramanyan, P., Gupta, A., Malik, S.: Lazy Self-composition for Security Verification: 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II, pp. 136–156 (07 2018)
55. Zdancewic, S., Myers, A.C.: Observational determinism for concurrent program security. 16th IEEE Computer Security Foundations Workshop, 2003. Proceedings. pp. 29–43 (2003)

Supplementary Material

In this section we provide additional examples and proofs that support the main text.

A Proof System

A.1 Proof rules

We provide the full set of proof rules in Fig. 11. In addition to sequential composition of the proof rules, we can also have a proof rule for for strengthening the precondition/weaking the postcondition. This is analogous to the consequence rule in the classical Floyd-Hoare system (e.g. [23]). We denote this as rule (3) in Fig. 11.

$$\text{(RBase)}\quad \frac{\forall q_1, q_2, q_1', q_2'.\ ((q_1, q_2) \models \{S, \mathit{\Gamma}\} \wedge \delta(q_1, \mathsf{op}_1 \cdots \mathsf{op}_n, q_1') \wedge \delta(q_2, \mathsf{op}_1 \cdots \mathsf{op}_n, q_2')) \implies (q_1', q_2') \models \{S', \mathit{\Gamma}'\}}{\{S, \mathit{\Gamma}\}\ M(\mathsf{op}_1 \cdots \mathsf{op}_n)\ \{S', \mathit{\Gamma}'\}}$$

$$\text{(RSeq)}\quad \frac{\{S, \mathit{\Gamma}\}\ M(\mathsf{op}_1 \cdots \mathsf{op}_k)\ \{S'', \mathit{\Gamma}''\} \qquad \{S'', \mathit{\Gamma}''\}\ M(\mathsf{op}_{k+1} \cdots \mathsf{op}_n)\ \{S', \mathit{\Gamma}'\}}{\{S, \mathit{\Gamma}\}\ M(\mathsf{op}_1 \cdots \mathsf{op}_n)\ \{S', \mathit{\Gamma}'\}}$$

$$\text{(RCons)}\quad \frac{(S_1 \subseteq S_1' \ \wedge\ \mathit{\Gamma}_1 \sqsubseteq \mathit{\Gamma}_1') \qquad (S_2' \subseteq S_2 \ \wedge\ \mathit{\Gamma}_2' \sqsubseteq \mathit{\Gamma}_2) \qquad \{S_1', \mathit{\Gamma}_1'\}\ M(\mathsf{op}_1 \cdots \mathsf{op}_n)\ \{S_2', \mathit{\Gamma}_2'\}}{\{S_1, \mathit{\Gamma}_1\}\ M(\mathsf{op}_1 \cdots \mathsf{op}_n)\ \{S_2, \mathit{\Gamma}_2\}}$$

Fig. 11. Proof rules for joint symbolic-taint judgements.

A.2 Proof of Theorem 1

We now provide a proof for Theorem 1.

Proof (Proof sketch). In the case of non-interference (NI), if 10 holds then, we also have

$$\{I, \mathit{\Gamma}^*\}\ M(P)\ \{Q, \mathit{\Gamma}_f\} \tag{15}$$

where $\mathit{\Gamma}^* = [V_{\mathtt{L}} \rightarrow \mathtt{L}, V_{\mathtt{H}} \rightarrow \mathtt{H}]$ (by RCons). Now consider any pair of traces $\pi_1, \pi_2 \in \mathit{Tr}_P(M)$. If $\pi_1^{(0)} \approx_{V_{\mathtt{L}}} \pi_2^{(0)}$ (precondition of NI, Defn. 6) then $(\pi_1^{(0)}, \pi_2^{(0)}) \models \{I, \mathit{\Gamma}^*\}$. Consequently (by Eq. 15), every pair of final states satisfy $(\pi_1^{(n)}, \pi_2^{(n)}) \models \{Q, \mathit{\Gamma}_f\}$. Then, by the condition on $\mathit{\Gamma}_f$ (in Eq. 10), we get $\pi_1^{(n)} \approx_{V_{\mathtt{L}}} \pi_2^{(n)}$ as desired. The proof for OD is similar, however, we use the intermediate taint-contexts to show equivalence $\pi_1^{(i)} \approx_{V_{\mathtt{L}}} \pi_2^{(i)}$ for each step i.

We now provide a proof sketch for TPOD. Suppose there exist traces $\pi_1, \pi_2, \pi_3, \pi_4 \in \mathit{Tr}_P(M)$ that satisfy the preconditions of TPOD. That is (A) $\pi_1, \pi_2 \in T_1$, (B) $\pi_3, \pi_4 \in T_2$, (C) $\pi_1 \approx_{V_{\mathtt{L}}} \pi_2$ and (D) $\pi_3^{(0)} \approx_{V_{\mathtt{L}}} \pi_4^{(0)}$. Then we proceed by induction to show that $((\pi_1^{(i)}, \pi_3^{(i)}), (\pi_2^{(i)}, \pi_4^{(i)})) \models \{S_i, \mathit{\Gamma}_i\}$ holds for each i. The base case follows by (Eq. 12) since $\pi_1^{(0)}, \pi_2^{(0)} \in I \cap I_1^{\mathit{tpod}}$ (since I_1^{tpod} is a cover invariant for T_1), and similarly $\pi_3^{(0)}, \pi_4^{(0)} \in I \cap I_2^{\mathit{tpod}}$.

Now assume (inductive case) that it holds for some i. The $((\pi_1^{(i+1)}, \pi_3^{(i+1)}), (\pi_2^{(i+1)}, \pi_4^{(i+1)})) \models \{S_{i+1}, \Gamma'_{i+1}\}$ holds by (F) and RBASE. Now, the fact that Γ_i and Γ'_i agree on $V_{\mathtt{L}}^2$ and that $(\pi_1^{(i)} \approx_{V_{\mathtt{L}}^1} \pi_2^{(i)})$ implies $((\pi_1^{(i+1)}, \pi_3^{(i+1)}), (\pi_2^{(i+1)}, \pi_4^{(i+1)})) \models \{S_{i+1}, \Gamma_{i+1}\}$. This shows the inductive case. Finally, this implies $\pi_3^{(i)} \approx_{V_{\mathtt{L}}^2} \pi_4^{(i)}$ for each i since $\Gamma_i(v) = \mathtt{L}$ for $v \in V_{\mathtt{L}}^2$.

B SAP Model

B.1 Assumptions

We first state some of the assumptions of the SAP model that affected some of the design decisions of the model.

Program Reachability Transient execution depends on the execution path that a program takes both non-speculatively and speculatively. Existing methods have used an *always mispredict* [16] model to explore the possible paths of program execution. We assume the same model and assume direct branches can speculatively execute in either direction. As for indirect branches, we assume that an arbitrary location is chosen as an (over-)approximation.

Page Tables and Memory Accesses Our model assumes static page table mappings. In addition, there are two partitions of memory initially, one partition contains only low-labelled data fragments and the second partition contains one high-labelled data fragment (representative of secret memory). The idea behind this is that if the adversary is able to exfiltrate information about a single secret, then the same program with additional secrets still results in a leak. Lastly, our model assumes memory consistency.

Forgoing Speculative Save States Lastly, our model does not include a notion of *save states* unlike prior formal models [10, 16, 21]. We find that even without speculative save states, a wide range of vulnerabilities can still be captured by our model. Moreover, speculative save states are often encoded as arrays of the architectural state, further increasing the complexity of the formal model. We note that we use symbolic model checking in our case study and these assumptions allow us to model a speculative microprocessor abstractly and simplify the model in a way that avoids quantification in the underlying SMT [4] formula. Under these assumptions, we formulate the speculative semantics of our model as the following.

In this section we describe the components of the SAP model in more detail. Figure 12 illustrates the components of the SAP model and their operations.

B.2 The Speculative Abstract Platform Model

In this section, we describe the SAP model in more detail.

Abstract CPU Model The CPU model is an abstraction of architectural states, which include the variables in the first row of Table 13. The `pc` is the program counter which is a virtual address type VA. The registers `regs` is a map from integers $\mathbb{N}$ to words W. The memory `mem` is a map from physical addresses PA to words W. `excp` is the exception register that is of type $\mathcal{E}$ [5], representing a set of exception types. In addition to the state variables there are a set of operations (Op) that define the transition relation semantics Δ. For the CPU, these operations are the load (`load`), store (`store`), conditional branch (`bge`), add (`add`), multiply (`mul`, and jump (`jmp`) instructions containing only the ISA level semantics. As an example, the semantics of the load operation (without virtualization) is described in Example 1.

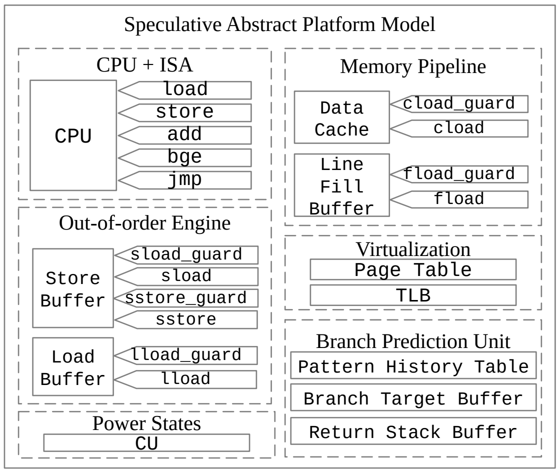

Fig. 12. The Speculative Abstract Platform. Each rectangle represents an IFSM module along with their operation wrappers indicated by tags, of which only a subset have been displayed.

Abstract Cache Model The second row in Table 13 describes the cache model, which extends the CPU model with a map of cache indexes (of type CA) to cache valid bits (of type boolean $Bool$) and a map of cache indexes (of type CA) to cache tags (modelled as physical addresses PA). Here, the cache index is an uninterpreted type and an uninterpreted function `lidx` is used to index into the cache line based on the address. We choose this formulation to capture a class of cache replacement policies. Note that the model assumes cache coherency, thus there is no need to create a separate variable for the data value in the cache.

Abstract Line Fill Buffer Model The line fill buffer is kept fairly abstract and only contains a hit map `lfb_hit` that returns whether a given physical address results in a

[5] We note that the number of exceptions is not comprehensive and only includes page faults, abort pages and device-not-available exceptions to accommodate the variants we verify.

Model	State Var.	Type	Description
CPU	`pc`	VA	The program counter.
	`regs`	$\mathbb{N} \rightarrow W$	Registers.
	`mem`	$PA \rightarrow W$	Physical memory.
	`excp`	$\mathcal{E}$	Exception register.
	`pid`	$\mathcal{P}_{id}$	Current executing process.
Cache	`cache_valid`	$CA \rightarrow Bool$	Cache index to valid bit.
	`cache_tag`	$CA \rightarrow PA$	Cache index to entry tag.
LFB	`lfb_hit`	$PA \rightarrow Bool$	PA to hit/miss value.
OoO Engine	`sb_valid`	$SI \rightarrow Bool$	SB index to valid bit.
	`sb_data`	$SI \rightarrow W$	SB index to data.
	`lb_valid`	$LI \rightarrow Bool$	LB index to valid bit.
	`lb_data`	$LI \rightarrow W$	LB index to data.
Page table	`addr_map`	$VA \rightarrow PA$	Virtual to physical addr. map.
	`addr_perm`	$VA \rightarrow ACL$	VA permissions and pres. bit.
	`owner`	$PA \rightarrow P_{id}$	PA to owner (PID).
TLB	`tlb_valid`	$TI \rightarrow Bool$	TLB set index to valid bit.
	`tlb_tag`	$TI \rightarrow PA$	TLB set index to tag.
Branch predictors	`pht`	$VA \rightarrow Bool$	Pattern history table.
	`btb`	$VA \rightarrow VA$	Branch target buffer.
Power states	`cu`	$Bool$	Power state of a computational unit.

Fig. 13. The Speculative Abstract Platform Model: State variables, along with their types and description, separated by individual IFSM components.

hit (which then returns the appropriate data value) or a miss. Despite the simplicity of the model, it is sufficient to capture the ineffectiveness of cache flushes in preventing micro-architectural data sampling attacks (MDS). Similar to the cache model, the line fill buffer contains wrappers for the `load` operation as shown in Figure 12.

Abstract Out-of-Order Engine Model The out-of-order (OoO) engine contains the store buffer and load buffer variables. `sb_valid` and `sb_data` are maps from the store buffer indices SI (of uninterpreted type) to the entry's valid bit and data value (of type word W) respectively. The load buffer is modelled similarly with the variables `lb_valid` and `lb_data`.

Abstract Page Table Model The page table model contains a virtual to physical address map `addr_map` and a permission map `addr_perm` where each virtual address is associated with a permission bit for read, write and execute and a present bit (e.g. $ACL \doteq \{R, W, X, P\}$, where R, W, X, P are boolean types). It also contains an owner map `owner` that determines which process owns a physical address. More interestingly, the operations for this model include address translation operations which are used in the `load` and `store` operations.

Abstract Translation Lookaside Buffer Model The translation lookaside buffer is modelled similarly to the cache, which includes a map from TLB indices TI (of uninterpreted type) to a valid bit and physical address (of type PA). The operations of the TLB model contain wrappers for the address translation operations defined in the abstract page table model.

Abstract Branch Predictor Model The abstract branch predictor uses an uninterpreted function to determine the branch direction of conditional direct branches based on a pattern history table state `pht`, which stores the previously predicted direction for each address. It also contains a branch target buffer state `btb`, which is a map from virtual addresses to the target address prediction. In other words, it takes the conventional *always mispredict* semantics common in existing speculative semantics [16, 21].

Abstract Power State Model Lastly, a boolean valued state `cu` is used to represent the power state of a computational unit such as the AVX2 unit. The state is set to true whenever an associated instruction (which is in the set of operations) that uses the unit is executed.

C Discussion

In this section, we describe some of the limitations of our proof system, verification methodology, and formal models.

C.1 Limitations

Compositional Verification of CondTPOD. While CondNI (Eq. 10) and $\mathrm{CondObsDet}$ (Eq. 11) can be checked by checking each Floyd-Hoare triple locally, $\mathrm{CondTPOD}$ (Eq. 12) requires checking trace properties over entire traces. Consequently, this means we lose the ability to check each Floyd-Hoare triple locally for general trace properties. However, trace properties that can be expressed as invariants over the triples naturally does allow our proof system to also check whether the system satisfies the trace properties in a localized manner. In fact, many useful properties, including the ones used to instantiate TPOD to derive secure speculation [10], can indeed be written as invariants (e.g. whether a program is allowed to speculate). Thus we defer the exploration of compositionally checking trace properties to existing and future work.

In-order Execution of Programs. Similar to existing approaches [10, 16, 21] that use symbolic execution, our verification methodology using the SAP model only considers in-order program instruction fetch. Thus, there may exist vulnerabilities on an out-of-order microprocessor that are not captured using our model alone. Proving properties about fully out-of-order processors would thus require modeling a component akin to a reorder buffer. Alternatively, one could potentially synthesize sound abstractions such that any violation of an information flow property in the out-of-order implementation model is preserved by the abstraction.

C.2 Soundness of Abstractions

While the SAP model is capable of capturing a broad class of attacks, we emphasize that every component is necessary for sound verification of any system. Thus while our SAP model is capable of capturing a broad class of vulnerabilities on a class of micro-architectures, ideally each micro-architecture should tailor the model to accommodate all components that could potentially be exploited. For more fine-grain analysis, one should use sound abstract models derived from the RTL implementation, as direct formal verification of RTL often does not scale.