# Hardware-Software Contracts for Secure Speculation

Marco Guarnieri*, Boris Köpf†, Jan Reineke‡, and Pepe Vila*
*IMDEA Software Institute †Microsoft Research ‡Saarland University

2021 IEEE Symposium on Security and Privacy (SP) | 978-1-7281-8934-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/SP40001.2021.00036

***Abstract*—Since the discovery of Spectre, a large number of hardware mechanisms for secure speculation has been proposed. Intuitively, more defensive mechanisms are less efficient but can securely execute a larger class of programs, while more permissive mechanisms may offer more performance but require more defensive programming. Unfortunately, there are no hardware-software contracts that would turn this intuition into a basis for principled co-design.**

**In this paper, we put forward a framework for specifying such contracts, and we demonstrate its expressiveness and flexibility.**

**On the hardware side, we use the framework to provide the first formalization and comparison of the security guarantees provided by a representative class of mechanisms for secure speculation.**

**On the software side, we use the framework to characterize program properties that guarantee secure co-design in two scenarios traditionally investigated in isolation: (1) ensuring that a benign program does not leak information while computing on confidential data, and (2) ensuring that a potentially malicious program cannot read outside of its designated sandbox. Finally, we show how the properties corresponding to both scenarios can be checked based on existing tools for software verification, and we use them to validate our findings on executable code.**

## I. Introduction

Speculative execution avoids expensive pipeline stalls by predicting the outcome of branching (and other) decisions, and by continuing the execution based on these predictions. When a prediction turns out to be incorrect, the processor rolls back the effects of speculatively executed instructions on the architectural state consisting of registers, flags, and main memory.

However, the microarchitectural state, which includes the content of various caches and buffers, is not (or only partially) rolled back. This side effect can leak information about the speculatively accessed data and thus violate confidentiality, see Figure 1a. Spectre attacks [1], [2] demonstrate that this vulnerability affects all modern general-purpose processors and poses a serious threat for platforms with multiple tenants.

A multitude of hardware mechanisms for secure speculation have been proposed. They are based on a number of basic ideas, such as delaying load operations until they cannot be squashed [3], delaying operations that depend on speculatively loaded data [4], [5], limiting the effect of speculatively executed instructions [6], [7], [8], [9], or rolling back the microarchitectural state when a misprediction is detected [10].

Intuitively, more defensive mechanisms are less efficient but can securely execute a larger class of programs, while more permissive mechanisms offer more performance but require more defensive programming. We refer to this intuition as (*).

For example, consider the variant of Spectre v1 shown in Figure 1b, where array `A` is accessed before the bounds check.

```
1 if (y < size_A)
2  x = A[y];
3  temp &= B[x * 64];
```

(a) Program $P_1$

```
1 x = A[y];
2 if (y < size_A)
3  temp &= B[x * 64];
```

(b) Program $P_2$

Fig. 1: Program $P_1$ is the vanilla Spectre v1 example, where `A[y]` can be speculatively read and leaked into the data cache via an access to array `B`, for `y >= size_A`. Program $P_2$, is a variant where `A[y]` is accessed non-speculatively before the bounds check but the leak occurs during speculative execution.

Mechanisms delaying loads until they cannot be squashed [3] prevent speculatively leaking `A[y]`, for $y \geq$ `size_A`. In contrast, more permissive mechanisms that delay only loads depending on speculatively accessed data [4], [5] do *not* prevent the leak, because `A[y]` is accessed non-speculatively.

While the performance characteristics of secure speculation mechanisms are well-studied, there has been little work on (1) characterizing the security guarantees they provide, and in particular on (2) investigating how these guarantees can be effectively leveraged by software to achieve global security guarantees.[1] That is, we lack hardware-software contracts that support principled co-design for secure speculation, and that would formalize the intuition (*) described above.

**Contracts:** In this paper, we put forward a framework for specifying such contracts, based on three basic building blocks: an ISA language, a model of the microarchitecture, and an adversary model specifying which microarchitectural components (such as caches or branch predictor state) are observable via side-channels.

Contracts specify which program executions a side-channel adversary can distinguish. A contract in our framework is defined in terms of *executions* and *observations* made on these executions, and it is formalized in terms of a labelled ISA semantics. A CPU satisfies a contract if, whenever two program executions agree on all observations, they are guaranteed to be indistinguishable by the adversary at the microarchitectural level. The contract semantics can mandate exploration of mispredicted paths, effectively requiring agreement on observations corresponding to transient instructions.

Secrets at the program level must not affect contract obser-

[1] A notable exception to (1) is STT [5], which is backed by a security property that guarantees the confidentiality of speculatively loaded data. However, this property alone does not provide an actionable basis for (2), as preventing leakage of non-speculatively accessed data (as in Figure 1b) is declared out of scope [5, Section 4].


DOI 10.1109/SP40001.2021.00036

vations, because then they can become visible to the adversary. Hence, contracts exposing more observations correspond to hardware with weaker security guarantees, whereas contracts exposing fewer observations correspond to hardware with stronger guarantees. The extreme case is a contract with no observations, which is satisfied by an ideal side-channel resilient platform that can securely execute every program.

**Software Side:** Our framework provides a basis for deriving requirements that *software* needs to satisfy to run securely on a specific platform. For deriving such requirements, we consider two scenarios typically considered in the literature:

- In the first scenario, called "constant-time programming", the goal is to ensure that a benign program, such as a cryptographic algorithm, does not leak information while computing on confidential data.
- In the second scenario, which we call "sandboxing", the goal is to restrict the memory region that a potentially malicious program, such as a Web application, can read from.[2]

For each scenario, we identify program-level properties that guarantee security on hardware that satisfies a given contract. We stress that secure speculation approaches usually *either* consider constant-time programming [11], [12], [13], [14] *or* sandboxing [15], [16]. In contrast, our framework supports *both* goals through program-level properties.

We provide tool support for automatically checking if programs are secure in both scenarios. For this, we extend a static analysis tool for detecting speculative leaks [11] to cater for different contracts, and we use it to validate all examples used in the paper on x86 executable code.

**Hardware Side:** We use our framework to define contracts for a comprehensive set of recent hardware mechanisms for secure speculation: disabling speculation, delaying speculative load operations [3], and speculative taint tracking [4], [5].

To this end, we formalize each mechanism in the context of a variant of the simple speculative out-of-order processor from [13] and we prove that it satisfies specific contracts against an adversary that observes caches, predictors, and (part of) the reorder buffer during execution. We show that the contracts we define form a lattice, and we use this to give, for the first time, a rigorous comparison of the security guarantees offered by different secure speculation mechanisms.

Our analysis highlights that the studied mechanisms [3], [4], [5] prevent leaks of speculatively accessed data, and confirms the results of [5]. For software, this means that "sandboxing" is supported out-of-the-box, in the sense that programs only need to place appropriate bounds checks, but no speculation barriers.

Our analysis also shows that the mechanisms offer no support for "constant-time programming". This means that programs that are constant-time in the traditional sense [17] still require additional checks [11], [13] or insertion of speculation barriers [18], even if hardware mechanisms for secure speculation are deployed.

**Summary of contributions:** We propose a novel framework for expressing security contracts between hardware and software. Our framework is expressive enough to (1) characterize the security guarantees provided by recent proposals for secure speculation, and (2) provide program-level properties formalizing how to leverage these hardware guarantees to achieve global, end-to-end security for different scenarios. From a theoretical perspective, we provide the first characterization of security for a comprehensive class of hardware mechanisms for secure speculation. From a practical perspective, we show how to automate checks for programs to run securely on top of these mechanisms.

**Bonus material:** A technical report containing a full formalization and proofs of all technical results is available at [19].

[2]In the terminology of [2], sandboxing aims to block disclosure gadgets.

## II. ISA LANGUAGE, SEMANTICS, AND ADVERSARIES

We introduce the foundations for specifying hardware-software contracts: an ISA language (§II-A), its architectural semantics (§II-B), a general notion of hardware semantics (§II-C), and an adversary model capturing which aspects of the microarchitecture are observable via side channels (§II-D).

### A. ISA language

For modeling the ISA we rely on $\mu$ASM, a simple assembly language from [11] with the following syntax:

| | | | |
|---|---|---|---|
| **Basic Types** | | | |
| *(Registers)* | $x$ | $\in$ | *Regs* |
| *(Values)* | $n, \ell$ | $\in$ | $Vals = \mathbb{N} \cup \{\bot\}$ |
| **Syntax** | | | |
| *(Expressions)* | $e$ | $:=$ | $n \mid x \mid \ominus e \mid e_1 \otimes e_2 \mid \textbf{ite}(e_1, e_2, e_3)$ |
| *(Instructions)* | $i$ | $:=$ | $\textbf{skip} \mid x \leftarrow e \mid \textbf{load}\ x, e \mid \textbf{store}\ x, e \mid \textbf{jmp}\ e \mid \textbf{beqz}\ x, \ell \mid \textbf{spbarr}$ |
| *(Programs)* | $p$ | $:=$ | $i \mid p_1; p_2$ |

- $\mu$ASM expressions are built from a set of register identifiers *Regs*, which contains a designated element **pc** representing the program counter, and a set of values *Vals*, which consists of the natural numbers and $\bot$.
- $\mu$ASM instructions include assignments, load and store instructions, indirect jumps, branching instructions, and a speculation barrier **spbarr**.
- $\mu$ASM programs are sequences of instructions.

### B. Architectural semantics $\rightarrow$

The architectural semantics models the execution of $\mu$ASM programs at the architectural level. It is defined in terms of *architectural states* $\sigma = \langle m, a \rangle$ consisting of a *memory* $m$ and a *register assignment* $a$. Memories $m$ map memory addresses, represented by natural numbers, to values in *Vals*. Register assignments $a$ map register identifiers to values in *Vals*. We signal program termination by assigning the special value $\bot$ to the program counter **pc**.

The architectural semantics is a deterministic binary relation $\sigma \rightarrow \sigma'$, which we formalize in [19], mapping an architectural state $\sigma$ to its successor $\sigma'$. A *run* is a finite sequence of states $\sigma_0, \ldots, \sigma_n$ with $\sigma_0 \rightarrow \ldots \rightarrow \sigma_n$ such that $\sigma_0$ is initial (that is, all registers including **pc** have value 0) and $\sigma_n$ is final (that is, $\sigma_n(\textbf{pc}) = \bot$).

### C. Hardware semantics $\Rightarrow$

A hardware semantics models the execution of $\mu$ASM programs at the microarchitectural level. Here we describe a general notion of hardware semantics with the key aspects necessary for explaining hardware-software contracts; we provided multiple, concrete hardware semantics modeling different processors and countermeasures in §V–VI.

Hardware semantics are defined in terms of *hardware states* $\langle \sigma, \mu \rangle$ consisting of an architectural state $\sigma$ (as before) and a *microarchitectural state* $\mu$, which models the state of components like predictors, caches, and reorder buffer.

A hardware semantics is a deterministic relation $\Rightarrow$ mapping hardware states $\langle \sigma, \mu \rangle$ to their successors $\langle \sigma', \mu' \rangle$. A *hardware run* is a sequence $\langle \sigma_0, \mu_0 \rangle \Rightarrow \ldots \Rightarrow \langle \sigma_n, \mu_n \rangle$ such that $\langle \sigma_0, \mu_0 \rangle$ is initial and $\langle \sigma_n, \mu_n \rangle$ is final. For this, we assume that there is a fixed, initial microarchitectural state $\mu_0$, where, for instance, the reorder buffer is empty and all caches have been invalidated.

### D. Adversary model

We consider adversaries that can observe parts of the microarchitectural state during execution. We model hardware observations as projections to parts of the microarchitectural state. For instance, a cache adversary can be modeled as a function $\mathscr{A}$ projecting $\mu$ to its cache component. In the paper, we consider an adversary $\mathscr{A}$ that has access to the state of caches, predictors, and (part of) the reorder buffer; we formalize $\mathscr{A}$ in Section V-C.

Given a program $p$, $\{\!|p|\!\}(\sigma)$ denotes the trace $\mathscr{A}(\mu_0) \cdot \ldots \cdot \mathscr{A}(\mu_n)$ of hardware observations produced in the run $\langle \sigma, \mu_0 \rangle \Rightarrow \ldots \Rightarrow \langle \sigma_n, \mu_n \rangle$. We refer to $\{\!|p|\!\}$ as the *hardware trace semantics* (hardware semantics for short) of program $p$.

## III. Hardware-software contracts

The purpose of a contract is to split the responsibilities for preventing side channels between software and hardware.

We first formalize the general notion of contracts and we specify when a hardware platform satisfies a contract. Then we present several fundamental contracts for secure speculation.

### A. Formalizing contracts

A *contract* is a labeled, deterministic semantics $\rightharpoonup$ for the ISA. Given a program $p$ and an initial architectural state $\sigma_0$, the labels on the transitions of the corresponding run $\sigma_0 \xrightarrow{l_1} \sigma_1 \xrightarrow{l_2} \ldots \xrightarrow{l_n} \sigma_n$ define the *trace* $[\![p]\!](\sigma_0) = l_1 l_2 \ldots l_n$. In this paper, we only consider terminating programs. We leave the extension to non-terminating programs as future work.

The traces of a contract $[\![p]\!]$ capture which architectural states are guaranteed to be indistinguishable to an attacker on a hardware satisfying the contract, which is formalized below.

**Definition 1** ($\{\!|\cdot|\!\} \vdash [\![\cdot]\!]$)**.** A hardware semantics $\{\!|\cdot|\!\}$ *satisfies a contract* $[\![\cdot]\!]$ if, for all programs $p$ and all initial architectural states $\sigma, \sigma'$, if $[\![p]\!](\sigma) = [\![p]\!](\sigma')$, then $\{\!|p|\!\}(\sigma) = \{\!|p|\!\}(\sigma')$.

Different contracts correspond to different divisions of security obligations between software and hardware: secrets at the program level must not affect contract observations, because then they can become visible to the adversary. Hence, contracts exposing more observations correspond to hardware with weaker security guarantees, whereas contracts exposing fewer observations correspond to hardware with stronger security guarantees. A degenerate case is a contract with no observations, which is satisfied by an ideal side-channel resilient platform that securely executes every program.

### B. Contracts for secure speculation

We now define four fundamental contracts that characterize the security guarantees offered by mechanisms for secure speculation. We derive our contracts, which we fully formalize in [19], as the combination of two kinds of building blocks.

*1) Building blocks for contracts:* The first building block are *observer modes*, which govern what information a contract exposes. We define them via labels on the contract semantics.

- The *constant-time* observer mode (ct for short) is commonly used when reasoning about side channels in cryptographic algorithms. It uses labels `pc` $\ell$, `load` $n$, and `store` $n$ to expose the value $\ell$ of the program counter and the addresses $n$ of load and store operations. The ct observer mode can be augmented with support for variable-latency instructions by additionally exposing the operands of those instructions as observations, or refined to capture adversaries that can infer addresses of memory accesses only up to the granularity of cache banks, lines, or pages [20]. We forgo both extensions for simplicity.
- The *architectural* observer mode (arch for short) additionally exposes the *value* $v$ that is loaded from memory location $n$ via the label `load` $n = v$ upon each load instruction. As registers are set to zero in the initial architectural state, arch traces effectively determine the values of all registers during execution.

The second building block are *execution modes* that characterize which paths need to be explored to collect observations. For processors with speculative execution, depending on the presence and effectiveness of hardware-level countermeasures, it is necessary to go beyond paths covered by the architectural semantics.

- In the *sequential* execution mode (seq for short), programs are executed sequentially and in-order following the architectural semantics.
- In the *always-mispredict* execution mode (spec for short), programs are executed sequentially, but incorrect branches are also executed for a bounded number of steps before backtracking. This execution mode is based on [11] and can be used to explore the effects of speculatively executed instructions at the ISA level.

*2) Contract* $[\![\cdot]\!]^{\text{seq}}_{\text{ct}}$*:* This contract exposes the program counter and the locations of memory accesses on sequential, non-speculative paths; see Figure 2. $[\![\cdot]\!]^{\text{seq}}_{\text{ct}}$ is a fundamental baseline that is often implicitly assumed in practice, and that has also been formalized in [17], [21].

In Section VI-A we show that $[\![\cdot]\!]^{\text{seq}}_{\text{ct}}$ is satisfied by a simple in-order processor without speculation. However, modern out-of-order processors do *not* satisfy $[\![\cdot]\!]^{\text{seq}}_{\text{ct}}$, as shown below.

$$\textsc{Load}\quad \frac{p(a(\mathbf{pc})) = \mathbf{load}\ x,e \qquad \langle m,a\rangle \to \langle m',a'\rangle}{\langle m,a\rangle \xrightarrow{\texttt{load}\ (\!|e|\!)(a)}{}^{\text{seq}}_{\text{ct}} \langle m',a'\rangle}$$

$$\textsc{Store}\quad \frac{p(a(\mathbf{pc})) = \mathbf{store}\ x,e \qquad \langle m,a\rangle \to \langle m',a'\rangle}{\langle m,a\rangle \xrightarrow{\texttt{store}\ (\!|e|\!)(a)}{}^{\text{seq}}_{\text{ct}} \langle m',a'\rangle}$$

$$\textsc{Beqz-Sat}\quad \frac{p(a(\mathbf{pc})) = \mathbf{beqz}\ x,\ell \qquad \langle m,a\rangle \to \langle m',a'\rangle}{\langle m,a\rangle \xrightarrow{\texttt{pc}\ a'(\mathbf{pc})}{}^{\text{seq}}_{\text{ct}} \langle m',a'\rangle}$$

Fig. 2: $[\![\cdot]\!]^{\text{seq}}_{\text{ct}}$ contract for a program $p$ - selected rules (here $(\!|e|\!)(a)$ is the result of expression $e$ given assignment $a$). The contract is obtained by augmenting the architectural semantics with observations `load` $n$, `store` $n$, and `pc` $\ell$ exposing the addresses of loads, stores, and the program counter, respectively.

**Example 1.** Consider the vanilla Spectre v1 snippet from Figure 1a, compiled to $\mu$ASM:

```
1 x ← y < size_A
2 beqz x, ⊥ //checking y < size_A
3 load z,A + y //accessing A[y]
4 z ← z*64
5 load w, B+z //accessing B[A[y]*64]
```

Consider architectural states $\sigma$ and $\sigma'$ that agree on the observations on trace `pc` $3 \cdot$ `load (A+y)` $\cdot$ `load (B+z)` (and hence on the content of array `A` within bounds), but for which $\sigma($`A + y`$) = 0$ and $\sigma'($`A + y`$) = 1$ for some `y>size_A`. On processors with speculation, an adversary with cache access can distinguish $\sigma$ and $\sigma'$, as shown by Spectre attacks [1].

Perhaps surprisingly, processors deploying recent proposals for secure speculation still violate $[\![\cdot]\!]^{\text{seq}}_{\text{ct}}$, see §VI.

*3) Contract $[\![\cdot]\!]^{\text{spec}}_{\text{ct}}$:* This contract additionally exposes the program counter and the locations of all memory accesses on speculatively executed paths. It is based on the speculative semantics from [11] and formalized in Figure 3.

In Section VI, we show that speculative out-of-order processors (with and without mechanisms for secure speculation) satisfy $[\![\cdot]\!]^{\text{spec}}_{\text{ct}}$.

Consider again Example 1: by exposing observations on mispredicted paths, $[\![\cdot]\!]^{\text{spec}}_{\text{ct}}$ makes the states $\sigma,\sigma'$ distinguishable at the contract level, effectively delegating the responsibility of ensuring that `A + y` does not carry secret information for `y >= size_A` to software.

*4) Contract $[\![\cdot]\!]^{\text{seq}}_{\text{arch}}$:* This contract exposes the program counter, the location of all loads and stores, and the values of all data loaded from memory on standard, i.e., non-speculative, program paths. The contract is obtained by modifying the `LOAD` rule from Figure 2 as follows:

$$\textsc{Load}\quad \frac{p(a(\mathbf{pc})) = \mathbf{load}\ x,e \qquad \langle m,a\rangle \to \langle m',a'\rangle}{\langle m,a\rangle \xrightarrow{\texttt{load}\ (\!|e|\!)(a)=m((\!|e|\!)(a))}{}^{\text{seq}}_{\text{arch}} \langle m',a'\rangle}$$

As we assume that register values are zeroed in the initial state, the $[\![\cdot]\!]^{\text{seq}}_{\text{arch}}$ trace effectively exposes the contents of registers during execution. While this does not seem to guarantee any kind of security, $[\![\cdot]\!]^{\text{seq}}_{\text{arch}}$ *does* guarantee the confidentiality of data that is *only transiently* loaded, thus effectively preventing speculative disclosure gadgets. In that sense, the contract $[\![\cdot]\!]^{\text{seq}}_{\text{arch}}$ is a simple and clean formulation of the idea behind *transient noninterference* [5], making it comparable to the guarantees offered by other contracts, and providing an actionable interface to software.

*5) Special contracts:* We informally present a number of contracts that illustrate our framework's expressiveness:

- $[\![\cdot]\!]_{\top}$ is the contract that does not expose any observations and corresponds to a hypothetical side-channel resilient processor that can securely execute every program.
- $[\![\cdot]\!]^{\text{seq-spec}}_{\text{ct-pc}}$ exposes program counter and addresses of loads during sequential execution, and only the program counter during speculative execution. That is, it may intuitively be understood as $[\![\cdot]\!]^{\text{seq}}_{\text{ct}} + [\![\cdot]\!]^{\text{spec}}_{\text{pc}}$. This contract corresponds, for instance, to processors vulnerable to speculative port-contention attacks like [22].
- $[\![\cdot]\!]^{\text{spec}}_{\text{arch}}$ exposes the values of data loaded from memory also during speculatively executed instructions. It corresponds to a processor that does not offer any confidentiality guarantees for accessed data.
- $[\![\cdot]\!]_{\bot}$ exposes all architectural state and corresponds to a hypothetical processor that provides no confidentiality guarantees whatsoever.

### C. A lattice of contracts

Contracts can be compared in terms of the security guarantees that they offer to software. Intuitively, a contract is stronger than another, if it guarantees to leak less information to a microarchitectural adversary. For instance, $[\![\cdot]\!]_{\top}$, which exposes no observations, is stronger than $[\![\cdot]\!]_{\bot}$, which exposes the entire architectural state, written $[\![\cdot]\!]_{\top} \sqsupseteq [\![\cdot]\!]_{\bot}$.

**Definition 2** ($[\![\cdot]\!]_1 \sqsupseteq [\![\cdot]\!]_2$)**.** A contract $[\![\cdot]\!]_1$ is *stronger* than a contract $[\![\cdot]\!]_2$ if $[\![p]\!]_2(\sigma) = [\![p]\!]_2(\sigma') \Rightarrow [\![p]\!]_1(\sigma) = [\![p]\!]_1(\sigma')$ for all programs $p$ and all initial architectural states $\sigma,\sigma'$.

Equivalently, $[\![\cdot]\!]_1 \sqsupseteq [\![\cdot]\!]_2$ holds whenever two architectural states that can be distinguished by $[\![\cdot]\!]_1$'s traces can also be distinguished by $[\![\cdot]\!]_2$'s traces.

Note that if $[\![\cdot]\!]_1$ exposes only a subset of the labels of $[\![\cdot]\!]_2$, then $[\![\cdot]\!]_1$ is stronger than $[\![\cdot]\!]_2$ according to Definition 2. For example, the instructions explored by spec are also explored by seq, and the observations of ct are contained in the observations of arch. This enables us to arrange all contracts defined in §III-B in the lattice [23] shown in Figure 4.

Finally, as expected, a hardware platform that satisfies a contract $[\![\cdot]\!]_1$ also satisfies all weaker contracts $[\![\cdot]\!]_2$.

$$\textsc{Step}\quad \frac{p(\sigma(\mathbf{pc})) \neq \mathbf{beqz}\ x,\ell \qquad \sigma \xrightarrow{\tau}{}^{\text{seq}}_{\text{ct}}\ \sigma'}{\langle \sigma, \omega+1\rangle \cdot s \xrightarrow{\tau}{}^{\text{spec}}_{\text{ct}}\ \langle \sigma', \omega\rangle \cdot s}$$

$$\textsc{Rollback}\quad \frac{s = \langle \sigma', \omega'\rangle \cdot s'}{\langle \sigma, 0\rangle \cdot s \xrightarrow{\text{pc}\ \sigma'(\mathbf{pc})}{}^{\text{spec}}_{\text{ct}}\ s}$$

$$\textsc{Barrier}\quad \frac{p(\sigma(\mathbf{pc})) = \mathbf{spbarr} \qquad \sigma \xrightarrow{\tau}{}^{\text{seq}}_{\text{ct}}\ \sigma'}{\langle \sigma, \omega+1\rangle \cdot s \xrightarrow{\tau}{}^{\text{spec}}_{\text{ct}}\ \langle \sigma', 0\rangle \cdot s}$$

$$\textsc{Branch}\quad \frac{p(\sigma(\mathbf{pc})) = \mathbf{beqz}\ x,\ell \quad \ell_{correct} = \begin{cases} \ell & \text{if } \sigma(x) = 0 \\ \sigma(\mathbf{pc})+1 & \text{otherwise} \end{cases} \quad \ell_{mispred} \in \{\ell, \sigma(\mathbf{pc})+1\} \setminus \ell_{correct} \quad \omega_{mispred} = \begin{cases} \mathrm{w} & \text{if } \omega = \infty \\ \omega & \text{otherwise} \end{cases}}{\langle \sigma, \omega+1\rangle \cdot s \xrightarrow{\text{pc}\ \ell_{mispred}}{}^{\text{spec}}_{\text{ct}}\ \langle \sigma[\mathbf{pc} \mapsto \ell_{mispred}], \omega_{mispred}\rangle \cdot \langle \sigma[\mathbf{pc} \mapsto \ell_{correct}], \omega\rangle \cdot s}$$

Fig. 3: Definition of $[\![\cdot]\!]^{\text{spec}}_{\text{ct}}$ contract. Configurations are stacks of $\langle \sigma, \omega\rangle$, where $\omega \in \mathbb{N} \cup \{\infty\}$ is the speculative window denoting how many instructions are left to be executed. (initial architectural states $\sigma$ are treated as $\langle \sigma, \infty\rangle$). At each computation step, the $\omega$ at the top of the stack is reduced by 1 (rules STEP and BRANCH). When executing a branch instruction (rule BRANCH), the state $\langle \sigma[\mathbf{pc} \mapsto \ell_{mispred}], \omega_{mispred}\rangle$ is pushed on top of the stack, thereby allowing the exploration of the mispredicted branch for $\omega_{mispred}$ steps. The correct branch $\langle \sigma[\mathbf{pc} \mapsto \ell_{correct}], \omega\rangle$ is also recorded on the stack; allowing to later roll back speculatively executed statements. When the $\omega$ at the top of the stack reaches 0, we pop it (i.e., we backtrack and discard the changes) and we continue the computation (rule ROLLBACK). Speculation barriers trigger a roll back by setting $\omega$ to 0 (rule BARRIER).

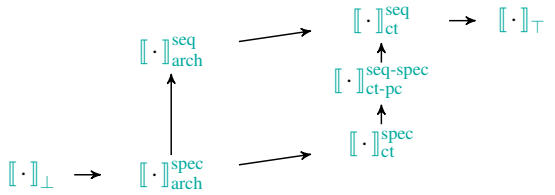


Fig. 4: Lattice of contracts. An edge from $[\![\cdot]\!]_2$ to $[\![\cdot]\!]_1$ means that $[\![\cdot]\!]_1 \sqsupseteq [\![\cdot]\!]_2$, that is, $[\![\cdot]\!]_1$ is stronger than $[\![\cdot]\!]_2$. The top element $[\![\cdot]\!]_{\top}$ of the lattice exposes no observations, while its bottom element $[\![\cdot]\!]_{\bot}$ exposes the entire architectural state.

**Proposition 1.** *If* $\{\!|\cdot|\!\} \vdash [\![\cdot]\!]_1$ *and* $[\![\cdot]\!]_1 \sqsupseteq [\![\cdot]\!]_2$*, then* $\{\!|\cdot|\!\} \vdash [\![\cdot]\!]_2$.

This implies that processors with stronger contracts $[\![\cdot]\!]_1$ are backward-compatible in the sense that they can securely execute any side-channel resilient legacy code that was already secure under weaker contracts $[\![\cdot]\!]_2$.

Full proofs of Proposition 1 and of the results in Figure 4 are available in [19].

## IV. Programming against contracts

Contracts are the basis for secure programming. Here, we consider two scenarios that are both instances of secure programming: In the first, which we call "constant-time programming", the goal is to ensure that a benign program does not leak confidential data to an adversary while computing on this data. In the second, which we call "sandboxing", the goal is to prevent a potentially malicious program from accessing confidential data. Proofs of this section's results are in [19].

### A. Secure programming

We begin by framing secure programming as an information-flow property. To distinguish confidential data from public data, we rely on a policy $\pi$: *Vals* $\to \{L, H\}$ that labels memory locations as high ($H$) or low ($L$), encoding whether locations store confidential data or not. Two architectural states $\sigma, \sigma'$ are *low-equivalent*, written $\sigma \simeq_\pi \sigma'$, iff the values of all low memory locations are the same.

**Definition 3** ($p \vdash NI(\pi, [\![\cdot]\!])$)**.** Program $p$ is *non-interferent* w.r.t. contract $[\![\cdot]\!]$ and policy $\pi$ if for all initial architectural states $\sigma, \sigma'$: $\sigma \simeq_\pi \sigma' \Rightarrow [\![p]\!](\sigma) = [\![p]\!](\sigma')$.

That is, a program is non-interferent w.r.t. a contract and a policy, if low-equivalent architectural states are indistinguishable under the contract, i.e., no information about high memory locations leaks into the contract's traces.

Similarly to Definition 3, one can define a notion of non-interference w.r.t. a hardware semantics $\{\!|\cdot|\!\}$, written $p \vdash NI(\pi, \{\!|\cdot|\!\})$, where information about high memory locations cannot flow into hardware observations.

The following proposition, capturing leakage at the hardware level, follows by composition of Definitions 1 and 3:

**Proposition 2.** *If* $p \vdash NI(\pi, [\![\cdot]\!])$ *and* $\{\!|\cdot|\!\} \vdash [\![\cdot]\!]$*, then* $p \vdash NI(\pi, \{\!|\cdot|\!\})$.

### B. Sandboxing

The goal of sandboxing is to enable the safe execution of untrusted, potentially malicious code. This is achieved by ensuring that the untrusted code is confined to a set of tightly controlled resources. Here we focus on one important aspect: preventing code from reading outside of its own subset of the address space. To achieve this, just-in-time compilers enforce access-control policies by inserting checks to ensure that all memory accesses happen within the sandbox's bounds.

We describe sandboxes using policies $\pi$, where memory outside of the sandbox is declared high. To account for programs that may escape the sandbox by exploiting speculation across access-control checks, we make the following distinction:

- Traditional sandboxing approaches [24], [25] check/enforce *vanilla sandboxing*: A program $p$ is *vanilla-sandboxed* w.r.t. $\pi$ if $p$ never accesses high memory locations when executing under the architectural semantics $\to$. In our framework,

being vanilla-sandboxed is equivalent to $p \vdash NI(\pi, [\![\cdot]\!]^{seq}_{arch})$, i.e., being non-interferent w.r.t. $[\![\cdot]\!]^{seq}_{arch}$. This follows from $[\![\cdot]\!]^{seq}_{arch}$ exposing the value of accessed high memory locations.

• To faithfully reason about sandboxing on out-of-order and speculative processors, one needs to go beyond vanilla sandboxing and make sure that the program does not leak any information that is outside of its sandbox through a covert channel. We say that a program is *generally-sandboxed* w.r.t. contract $[\![\cdot]\!]$, if it is vanilla-sandboxed and in addition non-interferent w.r.t $[\![\cdot]\!]$, i.e., $p \vdash NI(\pi, [\![\cdot]\!])$. General sandboxing together with Proposition 2 guarantees that no data outside of the sandbox affects what a microarchitectural adversary (including the sandboxed program $p$ itself, via probing) can observe on any platform satisfying $[\![\cdot]\!]$.

Definition 4 enables to bridge the gap between vanilla sandboxing and general sandboxing for a given program.

**Definition 4.** Program $p$ satisfies *weak speculative non-interference* (wSNI) with respect to $[\![\cdot]\!]$ if for all initial architectural states $\sigma, \sigma'$: $[\![p]\!]^{seq}_{arch}(\sigma) = [\![p]\!]^{seq}_{arch}(\sigma') \Rightarrow [\![p]\!](\sigma) = [\![p]\!](\sigma')$.

Weak speculative non-interference is a variant of *speculative non-interference*, the security property checked by Spectector [11]. Proposition 3 shows how wSNI bridges the gap between vanilla and general sandboxing.

**Proposition 3.** *If program $p$ is vanilla-sandboxed w.r.t. $\pi$ and wSNI w.r.t. $[\![\cdot]\!]$, then $p$ is generally-sandboxed w.r.t. $\pi$ and $[\![\cdot]\!]$.*

Hence, to check whether a program $p$ is generally-sandboxed w.r.t. $[\![\cdot]\!]$ and $\pi$ one can: (1) check/enforce that $p$ is vanilla-sandboxed w.r.t. $\pi$, and (2) verify whether $p$ is wSNI.

### C. Constant-time programming

Constant-time programming is a coding discipline for the implementation of code like cryptographic algorithms that needs to compute over secret data without leaks. Code without (1) secret-dependent control flow, (2) secret-dependent memory accesses, and (3) secret-dependent inputs to variable-latency instructions is traditionally understood as "constant time". As discussed before this corresponds to $[\![\cdot]\!]^{seq}_{ct}$, which exposes control flow and memory accesses.

Again, considering only $[\![\cdot]\!]^{seq}_{ct}$ is insufficient to reason about constant-time on modern processors. For this, we make the following distinction:

• Existing constant-time approaches (type systems [26], static analyses [17], [27], and techniques for secure compilation [28], [29]) check/enforce *vanilla-constant-time*. In our framework, a program $p$ is *vanilla-constant-time* w.r.t. $\pi$ if $p \vdash NI(\pi, [\![\cdot]\!]^{seq}_{ct})$, i.e., $p$ non-interferent w.r.t. $[\![\cdot]\!]^{seq}_{ct}$.

• More generally, a program $p$ is *generally-constant-time* w.r.t. contract $[\![\cdot]\!]$ iff $p \vdash NI(\pi, [\![\cdot]\!])$, i.e., constant-time coincides with non-interference w.r.t. a contract.

One possibility for checking general-constant-time is devising dedicated tools [13]. Alternatively, one can reuse vanilla-constant-time tools [17], [27] and then bridge the gap between vanilla and general-constant-time. To bridge this gap, one can rely on the following generalization of *speculative non-interference* from [11]:

```
1 if (y < size_A)
2   x = A[y];
3   if (x)
4     temp &= B[0];
```

(a) Program $P'_1$

```
1 x = A[y];
2 if (y < size_A)
3   if (x)
4     temp &= B[0];
```

(b) Program $P'_2$

Fig. 5: Variants of Spectre v1 that leak information through the control-flow statement in line 3.

**Definition 5** (Speculative non-interference [11])**.** Program $p$ is *speculatively non-interferent* (SNI) w.r.t. policy $\pi$ and contract $[\![\cdot]\!]$ if for all initial architectural states $\sigma, \sigma'$:

$$\sigma \simeq_\pi \sigma' \wedge [\![p]\!]^{seq}_{ct}(\sigma) = [\![p]\!]^{seq}_{ct}(\sigma') \Rightarrow [\![p]\!](\sigma) = [\![p]\!](\sigma').$$

Proposition 4 shows how SNI bridges the gap between vanilla and general constant-time.

**Proposition 4.** *If program $p$ is vanilla-constant-time w.r.t. $\pi$ and SNI w.r.t. $\pi$ and $[\![\cdot]\!]$, then $p$ is generally-constant-time w.r.t. $\pi$ and $[\![\cdot]\!]$.*

Thus to check whether a program $p$ is generally-constant-time w.r.t. $[\![\cdot]\!]$ and $\pi$ one can (1) check vanilla-constant-time, and (2) verify whether $p$ is SNI w.r.t. $[\![\cdot]\!]$ and $\pi$.

Observe, however, that not all contracts are useful for general-constant-time. Remarkably, the $[\![\cdot]\!]^{seq}_{arch}$ contract, which naturally corresponds to the guarantees provided by state-of-the-art hardware-level countermeasures like STT [5] and NDA [4] is inherently inadequate for constant-time programming: A program that is non-interferent w.r.t. $[\![\cdot]\!]^{seq}_{arch}$ may not access any secret data. However, accessing and computing on secret data is the whole point of constant-time programming.

### D. Experiments

In this section, we illustrate how our framework can be used to support secure programming, for both the sandboxing and constant-time scenarios, w.r.t. the contracts from §III.

**Tooling:** To automate our analysis we adapted Spectector [11], which can already check SNI for the $[\![\cdot]\!]^{spec}_{ct}$ contract, to support checking SNI and wSNI w.r.t. all the contracts from §III, i.e., $[\![\cdot]\!]^{seq}_{arch}$, $[\![\cdot]\!]^{seq}_{ct}$, $[\![\cdot]\!]^{spec}_{ct}$, $[\![\cdot]\!]^{seq\text{-}spec}_{ct\text{-}pc}$.

Propositions 3–4 present a clear path to check (general) sandboxing/constant-time: (1) use existing tools to verify vanilla sandboxing/constant-time, and (2) verify wSNI/SNI using Spectector.

**Experimental setup:** We analyze 4 different programs:

• $P_1$ and $P_2$ are the Spectre v1 snippet from Figure 1a and its variant from Figure 1b, respectively.

• $P'_1$ and $P'_2$ are modifications of $P_1$ and $P_2$ that leak information through control-flow statements. The programs are shown in Figure 5.

We compile each program with Clang at `-O2` optimization level. We also compile each program with a countermeasure that automatically injects `lfence` speculation barriers after

Table I: Sandboxing analysis w.r.t. different contracts.

| | $[\![\cdot]\!]^{\text{seq}}_{\text{ct}}$ | $[\![\cdot]\!]^{\text{seq}}_{\text{arch}}$ | $[\![\cdot]\!]^{\text{spec}}_{\text{ct}}$ | $[\![\cdot]\!]^{\text{seq-spec}}_{\text{ct-pc}}$ |
|---|---|---|---|---|
| $P_1$ | Y, $\sqsupseteq$ | Y, $\sqsupseteq$ | N | Y, wSNI |
| $P_1^f$ | Y, $\sqsupseteq$ | Y, $\sqsupseteq$ | Y, wSNI | Y, wSNI |
| $P_1'$ | Y, $\sqsupseteq$ | Y, $\sqsupseteq$ | N | N |
| $P_1'^f$ | Y, $\sqsupseteq$ | Y, $\sqsupseteq$ | Y, wSNI | Y, wSNI |

Table II: Constant-time analysis results w.r.t. diff. contracts.

| | $[\![\cdot]\!]^{\text{seq}}_{\text{ct}}$ | $[\![\cdot]\!]^{\text{seq}}_{\text{arch}}$ | $[\![\cdot]\!]^{\text{spec}}_{\text{ct}}$ | $[\![\cdot]\!]^{\text{seq-spec}}_{\text{ct-pc}}$ |
|---|---|---|---|---|
| $P_2$ | Y, $\sqsupseteq$ | N | N | Y, SNI |
| $P_2^f$ | Y, $\sqsupseteq$ | N | Y, SNI | Y, SNI |
| $P_2'$ | Y, $\sqsupseteq$ | N | N | N |
| $P_2'^f$ | Y, $\sqsupseteq$ | N | Y, SNI | Y, SNI |

each branch instruction.[3] We denote by $P^f$ the program $P$ with `lfences`.

As a result, we have eight small x86 programs that we analyze with the help of our enhanced version of Spectector.

**Sandboxing:** We analyze programs $P_1, P_1', P_1^f, P_1'^f$ w.r.t. the policy $\pi$ that declares the contents of `A[i]` as *low* for all `i` that are within the array bounds, and as *high* otherwise.

Our goal is to determine whether these programs satisfy the general-sandboxing property w.r.t. the contracts in §III. We remark that all variants of $P_1$ are vanilla-sandboxed w.r.t. $\pi$: they never access out-of-bound locations under the architectural semantics $\rightarrow$ thanks to the bounds check.

Table I summarizes our findings, which we discuss below:

- For $[\![\cdot]\!] \in \{[\![\cdot]\!]^{\text{seq}}_{\text{arch}}, [\![\cdot]\!]^{\text{seq}}_{\text{ct}}\}$, the fact that $[\![\cdot]\!]^{\text{seq}}_{\text{arch}}$ and $[\![\cdot]\!]^{\text{seq}}_{\text{ct}}$ are stronger than $[\![\cdot]\!]^{\text{seq}}_{\text{arch}}$ (see §III-C) directly implies wSNI w.r.t. these contracts for *any* program (denoted by "Y, $\sqsupseteq$" in the table). Therefore, programs $P_1, P_1', P_1^f, P_1'^f$ all satisfy general-sandboxing (see Proposition 3) without further analysis.
- For $[\![\cdot]\!] \in \{[\![\cdot]\!]^{\text{spec}}_{\text{ct}}, [\![\cdot]\!]^{\text{seq-spec}}_{\text{ct-pc}}\}$, we check whether wSNI holds using Spectector. Table entries "Y, wSNI" denote a successful check, which implies (via Proposition 3) that the program is generally-sandboxed w.r.t. $[\![\cdot]\!]$. In several cases, denoted by "N", the wSNI check fails. While this is not generally the case, the counterexamples to wSNI show that the respective programs are indeed not sandboxed w.r.t. $[\![\cdot]\!]$.

– Program $P_1$ fails the wSNI check w.r.t. $[\![\cdot]\!]^{\text{spec}}_{\text{ct}}$, due to the speculative secret-dependent load (line 3 in Figure 1b), but it satisfies wSNI w.r.t. the stronger contract $[\![\cdot]\!]^{\text{seq-spec}}_{\text{ct-pc}}$ that ensures confidentiality of secret-dependent speculative loads.

– In contrast, program $P_1'$ violates wSNI due to the speculative branch on line 3 in Figure 5 w.r.t. $[\![\cdot]\!]^{\text{spec}}_{\text{ct}}$ and $[\![\cdot]\!]^{\text{seq-spec}}_{\text{ct-pc}}$.

– Finally, programs $P_1^f$ and $P_1'^f$, where `lfences` are inserted after the branch, satisfy wSNI w.r.t. $[\![\cdot]\!]^{\text{spec}}_{\text{ct}}$ and $[\![\cdot]\!]^{\text{seq-spec}}_{\text{ct-pc}}$.

**Constant-time:** We analyze programs $P_2, P_2', P_2^f, P_2'^f$ w.r.t. the same policy $\pi$ as before.

This time, our goal is to determine whether these programs are constant-time w.r.t. the contracts in §III. We remark that $P_2, P_2', P_2^f, P_2'^f$ are vanilla-constant-time w.r.t. $\pi$, while none of these programs is vanilla-sandboxed w.r.t. $\pi$.

Table II summarizes our findings, which we discuss below:

- For $[\![\cdot]\!]^{\text{seq}}_{\text{ct}}$, all programs are constant-time w.r.t. $[\![\cdot]\!]^{\text{seq}}_{\text{ct}}$ as they are vanilla-constant-time (denoted by "$Y, \sqsupseteq$" in the table).
- For $[\![\cdot]\!]^{\text{seq}}_{\text{arch}}$, constant-time is violated for all programs, with and without `lfence`, due to the non-speculative load of a secret into the architectural state.
- For $[\![\cdot]\!] \in \{[\![\cdot]\!]^{\text{spec}}_{\text{ct}}, [\![\cdot]\!]^{\text{seq-spec}}_{\text{ct-pc}}\}$, Table entries "Y, SNI" denote a successful check using Spectector, which implies (via Proposition 4) that the program is constant-time w.r.t. $[\![\cdot]\!]$. Again, while this is not true in general, the counterexamples to SNI for these particular programs turn out to be proofs that the programs are not constant-time w.r.t. $[\![\cdot]\!]$.

Program $P_2$ violates SNI w.r.t. $[\![\cdot]\!]^{\text{spec}}_{\text{ct}}$ but satisfies it under the stronger contract $[\![\cdot]\!]^{\text{seq-spec}}_{\text{ct-pc}}$ that does not expose the address of the speculative load (line 3 in Figure 1b). In contrast, $P_2'$ violates SNI against both contracts. Finally, the programs with fences ($P_2^f$ and $P_2'^f$) satisfy SNI w.r.t. $[\![\cdot]\!]^{\text{spec}}_{\text{ct}}$ and $[\![\cdot]\!]^{\text{seq-spec}}_{\text{ct-pc}}$.

[3]The countermeasure is enabled with the `-x86-speculative-load-hardening -x86-slh-lfence` flags.

## V. Modeling microarchitecture and adversaries

This section presents a hardware semantics for $\mu$ASM programs. The semantics is based on the semantics from [13], [18] and it models the execution of $\mu$ASM programs by a simple out-of-order processor with a unified cache for data and instructions and a branch predictor for speculative execution over branch instructions. The purpose of this semantics is to allow us to model and reason about hardware-level Spectre countermeasures; see §VI. To this end, it strives to achieve the following design goals: (1) To faithfully capture the key features of speculative and out-of-order execution, while (2) keeping it simple, and (3) supporting large classes of microarchitectural features like caches and branch predictors. The latter aspect allows us to focus on hardware-level countermeasures in the context of arbitrary caching algorithms and branch-prediction strategies.

We start by formalizing hardware configurations (Section V-A) that extend architectural states with the state of the microarchitectural components, i.e., cache, reorder buffer, and branch predictor. Next, we formalize the semantics of the pipeline steps (Section V-B). This semantics describes how instructions are fetched, executed, and retired under our semantics as well as how hardware configurations are updated during the execution. We conclude by formalizing the adversary that we consider in our security analysis (Section V-C).

### A. Hardware configurations

Each *hardware configuration* $\langle \sigma, \mu \rangle$ consists of its architectural state $\sigma$, recording the memory and register assignments, and of its microarchitectural state $\mu$, which we formalize next.

The microarchitectural state consists of a reorder buffer, which stores the state of in-flight instructions, a cache, a branch predictor, and a scheduler, which orchestrates the pipeline during the computation. Note that, in our model, cache states track which memory blocks are stored in the cache (i.e., they store metadata) but they do *not* store the data itself. While we fix the behavior of the reorder buffer in §V-A1, our semantics is parametric in the models of caches, branch pre-

dictors, and the pipeline scheduler; see §V-A2. Theorem statements in §VI (except where explicitly stated) hold for *all* possible choices of cache, predictor, and scheduler in our model.

*1) Reorder buffers:* Reorder buffers store the state of in-flight, i.e., not yet retired, instructions. Initially instructions are *unresolved*, e.g., a load **load** $x, y+z$ that has not yet been performed or an assignment $z \leftarrow 2+k$ whose right-hand side has not yet been evaluated. Executing an unresolved instruction can transform it into a *resolved* instruction, where all expressions are replaced with their values. Additionally, to model speculative control flow, reorder buffer entries may be *tagged* with the address of a branch instruction $\ell$. We write $\mathbf{pc} \leftarrow v@\ell$, whenever the assignment of $v$ to the **pc** is the result of a call to the branch predictor when fetching the branch at address $\ell$ (only assignments to the program counter register **pc** are tagged since branch prediction is the sole source of speculation in our semantics). Instructions are *untagged*, written $i@\varepsilon$, if they are not the result of a prediction.

We model reorder buffers as sequences of *commands* of length at most $\mathbf{w}$ denoting the buffer's maximal length:

| | | | |
|---|---|---|---|
| *(Tags)* | $T$ | $:=$ | $\varepsilon \mid \ell$ |
| *(Commands)* | $c$ | $:=$ | $i@T$ |
| *(Reorder buffers)* | $buf$ | $:=$ | $\varepsilon \mid c \cdot buf$ |

A reorder buffer captures the state of execution of in-flight instructions. Consider the buffer $buf := k \leftarrow 25@\varepsilon \cdot \mathbf{load}\ x, y+z@\varepsilon \cdot z \leftarrow 2+k@\varepsilon$. It records that there are three in-flight instructions: one of them ($k \leftarrow 25@\varepsilon$) has been resolved and is ready to be retired, while the remaining two are still unresolved. Executing the third command would result in the new buffer $buf' := k \leftarrow 25@\varepsilon \cdot \mathbf{load}\ x, y+z@\varepsilon \cdot z \leftarrow 27@\varepsilon$.

Given a buffer $buf$, its data-independent projection $buf{\downarrow}$ is obtained by replacing all resolved (respectively unresolved) expressions in instructions with `R` (respectively `UR`). For instance, the data-independent projection of the buffer $buf$ from above is $k \leftarrow \mathtt{R}@\varepsilon \cdot \mathbf{load}\ x, \mathtt{UR}@\varepsilon \cdot z \leftarrow \mathtt{UR}@\varepsilon$.

*2) Caches, Branch predictors, and Schedulers:* Rather than providing a fixed model for caches, branch predictors and schedulers, our semantics is parametric in such components. To this end, we only fix the interface to these components, which is given in Table III, constraining how the semantics may interact with these components. Each of these components is defined by a set of states, an initial state, and uninterpreted functions modeling their relevant behavior:

- Caches are equipped with a function $access(cs, \ell) \in \{\mathtt{Hit}, \mathtt{Miss}\}$ that captures whether accessing memory address $\ell$ in cache state $cs$ results in a cache hit (`Hit`) or miss (`Miss`), and a function $update(cs, \ell) = cs'$ that updates the state of the cache based on the access to address $\ell$. We stress that cache states $cs$ track only the memory addresses of the blocks in the cache, *not* the blocks themselves.
- Branch predictors are equipped with a function $update(bp, \ell, b)$ that updates the state $bp$ of the branch predictor by recording that the branch at program counter $\ell$ has been resolved to value $b$, and $predict(bp, \ell)$ that, given a predictor state $bp$, predicts the outcome of the branch at address $\ell$.
- Schedulers determine which pipeline stages to activate next. Following [13], [18], we model this choice using three types of directives: (a) **fetch** is used to fetch and decode the next instruction pointed by the program counter register **pc**, (b) **execute** $i$ is used to execute the $i$-th command in the reorder buffer $buf$, and (c) **retire** is used to retire (i.e., apply the changes to the memory and register file) the first command in the buffer. Schedulers are equipped with an $next(sc)$ function that produces the next directive given the scheduler's state $sc$, and an $update(sc, buf)$ function that updates the scheduler's state based on the state of the reorder buffer.

*3) Microarchitectural states:* A *microarchitectural state* $\mu$ is a 4-tuple $\langle buf, cs, bp, sc \rangle$ where $buf$ is a reorder buffer, $cs$ is the state of the unified cache (for data and instructions), $bp$ is the branch predictor state, and $sc$ is the scheduler state.

A microarchitectural state $\mu$ is *initial* if $buf = \varepsilon$ and the microarchitectural components are in their initial states. Similarly, $\mu$ is *final* if $buf = \varepsilon$. Hence, a hardware configuration $\langle \sigma, \mu \rangle$ is initial (respectively final) if $\sigma$ and $\mu$ are so.

For simplicity, we write $\langle m, a, buf, cs, bp, sc \rangle$ to represent the hardware configuration $\langle \langle m, a \rangle, \langle buf, cs, bp, sc \rangle \rangle$.

## B. Hardware semantics

We formalize the hardware semantics of a $\mu$ASM program $p$ using a binary relation $\Rightarrow \subseteq HwStates \times HwStates$ that maps hardware states to their successors:

STEP

$$\frac{\langle m, a, buf, cs, bp \rangle \overset{d}{\Rightarrow} \langle m', a', buf', cs', bp' \rangle \qquad d = next(sc) \qquad sc' = update(sc, buf'{\downarrow})}{\langle m, a, buf, cs, bp, sc \rangle \Rightarrow \langle m', a', buf', cs', bp', sc' \rangle}$$

The rule captures one execution step at the microarchitectural level. The scheduler is queried to determine the directive $d = next(sc)$ indicating which pipeline step to execute. Next, the microarchitectural state is updated by performing one step of the auxiliary relation $\langle m, a, buf, cs, bp \rangle \overset{d}{\Rightarrow} \langle m', a', buf', cs', bp' \rangle$, which depends on the directive $d$ and is formalized below. Finally, the scheduler state is updated based on the data-independent projection of the reorder buffer, i.e., $sc' = update(sc, buf'{\downarrow})$. This formalizes the crucial assumption that the scheduler's decisions may depend upon the dependencies between the instructions in the reorder buffer, but not on the values computed thus far.

For each directive, i.e., **fetch**, **execute** $i$, and **retire**, we sketch below the rules that govern the definition of the auxiliary relations $\overset{\textbf{fetch}}{\Longrightarrow}$, $\overset{\textbf{execute}\ i}{\Longrightarrow}$, and $\overset{\textbf{retire}}{\Longrightarrow}$. We provide a full formalization of the rules in [19].

*1) Fetch:* Instructions are fetched in-order. Here we present selected rules modeling instruction fetch:

FETCH-BRANCH-HIT

$$\frac{a' = apl(buf, a) \qquad |buf| < \mathbf{w} \qquad a'(\mathbf{pc}) \neq \bot \qquad p(a'(\mathbf{pc})) = \mathbf{beqz}\ x, \ell \qquad \ell' = predict(bp, a'(\mathbf{pc})) \qquad access(cs, a'(\mathbf{pc})) = \mathtt{Hit} \qquad update(cs, a'(\mathbf{pc})) = cs'}{\langle m, a, buf, cs, bp \rangle \overset{\textbf{fetch}}{\Longrightarrow} \langle m, a, buf \cdot \mathbf{pc} \leftarrow \ell'@a'(\mathbf{pc}), cs', bp \rangle}$$

FETCH-MISS

$$\frac{|buf| < \mathbf{w} \qquad a' = apl(buf, a) \qquad a'(\mathbf{pc}) \neq \bot \qquad access(cs, a'(\mathbf{pc})) = \mathtt{Miss} \qquad update(cs, a'(\mathbf{pc})) = cs'}{\langle m, a, buf, cs, bp \rangle \overset{\textbf{fetch}}{\Longrightarrow} \langle m, a, buf, cs', bp \rangle}$$

Table III: Signatures of the microarchitectural components

| Component | States | Initial state | Functions | |
|---|---|---|---|---|
| *Cache* | *CacheStates* | $cs_0$ | $access : CacheStates \times Vals \to \{\texttt{Hit},\texttt{Miss}\}$ | $update : CacheStates \times Vals \to CacheStates$ |
| *Branch predictor* | *BpStates* | $bp_0$ | $predict : BpStates \times Vals \to Vals$ | $update : BpStates \times Vals \times Vals \to BpStates$ |
| *Pipeline scheduler* | *ScStates* | $sc_0$ | $next : ScStates \to Dir$ | $update : ScStates \times Bufs \to ScStates$ |

In these rules, and in those described later, $apl(buf, a)$ denotes the assignment $a'$ obtained by updating $a$ with the changes performed by the commands in $buf$. Concretely, $apl(buf, a)$ iteratively applies the pending changes for all commands in $buf$ as follows: (a) Assignments $x \leftarrow e@T$ set the value of $a'(x)$ to $e$ if the assignment is resolved (i.e., $e \in Vals$) and to $\bot$ otherwise (denoting unresolved values). (b) Load operations **load** $x, e@T$ set the value of $a'(x)$ to $\bot$ (since the load operation has not been performed yet). (c) Whenever $buf$ contains a speculation barrier **spbarr**$@T$, $apl(buf, a) = \lambda x \in Regs.\ \bot$. (d) Other instructions are ignored.

The rule FETCH-BRANCH-HIT models the fetch of a branch instruction **beqz** $x, \ell$. Whenever the reorder buffer $buf$ is not full ($|buf| < \mathtt{w}$), **pc** is defined ($a'(\mathbf{pc}) \neq \bot$), and the instruction is in the cache ($access(cs, a'(\mathbf{pc})) = \texttt{Hit}$), the branch predictor is queried to obtain the next program counter $\ell' = predict(bp, a'(\mathbf{pc}))$. Next, the cache and the reorder buffer states are updated. The latter is updated by appending the command $\mathbf{pc} \leftarrow \ell'@a'(\mathbf{pc})$, which records the change to the program counter as well as the label of the branch instruction whose target was predicted. The semantics also contains rules for fetching jumps **jmp** $e$, which append the command $\mathbf{pc} \leftarrow e@\varepsilon$ to the buffer, and other instructions $i$, which append the commands $i@\varepsilon \cdot \mathbf{pc} \leftarrow a'(\mathbf{pc}) + 1@\varepsilon$ to the buffer.

The rule FETCH-MISS models a cache miss when loading the next instruction. In this case, the cache is updated while the reorder buffer is not modified. A subsequent **fetch** triggered by the scheduler would result in a cache hit and a corresponding change to the reorder buffer.

*2) Execute:* Commands in-flight are executed out-of-order, where the **execute** $i$ directive triggers the execution of the $i$-th command in the buffer. Selected rules are given in Figure 6.

The rule EXECUTE-LOAD-HIT models the successful execution of a load (**load** $x, e@T$) that results in a cache hit. In the rule, $(\!|e|\!)(a')$ denotes the result of evaluating $e$ in the context of the assignment $a'$ obtained by applying to $a$ all earlier in-flight commands in $buf$. Whenever the address is resolved, i.e., $(\!|e|\!)(a') \neq \bot$, and accessing the address results in a cache hit ($access(cs, (\!|e|\!)(a')) = \texttt{Hit}$), the reorder buffer is updated by replacing **load** $x, e@T$ with $x \leftarrow m((\!|e|\!)(a'))@T$, thereby recording that the load operation has been executed and that the value of $x$ is now $m((\!|e|\!)(a'))$. The cache state is also updated to account for the memory access to $(\!|e|\!)(a')$.

In contrast, the EXECUTE-BRANCH-ROLLBACK rule models the resolution of a mis-speculated branch instruction that results in rolling back the speculatively executed instructions by dropping their entries from the reorder buffer. Whenever the predicted value $\ell$ disagrees with the outcome $\ell'$ of the instruction **beqz** $x, \ell''$ at address $\ell_0$, the buffer is updated by (1) recording the new value of **pc** (by replacing $\mathbf{pc} \leftarrow \ell@\ell_0$ with $\mathbf{pc} \leftarrow \ell'@\varepsilon$), and (2) squashing all later buffer entries (by discarding the buffer suffix $buf'$). Moreover, the branch predictor's state is updated by recording that the branch at address $\ell_0$ has been resolved to $\ell'$.

*3) Retire:* Instructions are retired in-order. This is done by retiring only commands $i@T$ at the head of the reorder buffer where the instruction $i$ has been resolved and the tag $T$ is $\varepsilon$ indicating that there are no unresolved predictions. Selected rules for the **retire** directive are given below:

RETIRE-ASSIGNMENT

$$\frac{buf = x \leftarrow v@\varepsilon \cdot buf' \qquad v \in Vals}{\langle m, a, buf, cs, bp\rangle \xRightarrow{\textbf{retire}} \langle m, a[x \mapsto v], buf', cs, bp\rangle}$$

RETIRE-STORE

$$\frac{buf = \textbf{store}\ v, n@\varepsilon \cdot buf' \qquad v, n \in Vals \qquad update(cs, n) = cs'}{\langle m, a, buf, cs, bp\rangle \xRightarrow{\textbf{retire}} \langle m[n \mapsto v], a, buf', cs', bp\rangle}$$

The rule RETIRE-ASSIGNMENT models the retirement of a command $x \leftarrow v@\varepsilon$, where the assignment $a$ is permanently updated by recording that $x$'s value is now $v$. In contrast, RETIRE-STORE models the retirement of store commands **store** $v, n@\varepsilon$. In this case, the memory $m$ is permanently updated by writing the value $v$ to address $n$ and the cache state is updated. Finally, we have rules RETIRE-SKIP and RETIRE-BARRIER modeling the retirement of **skip** and **spbarr** instructions, which are removed from the reorder buffer without modifying the architectural state.

### C. Formalizing the adversary model

We conclude by formalizing the adversary model that we use in the security analysis in Section VI.

In our analysis, we consider an adversary $\mathscr{A}$ that can observe almost the entire microarchitectural state. Specifically, it can observe (1) the data-independent projection of the reorder buffer (i.e., which instructions are in-flight, but not to what values they are resolved), (2) the state of cache (which stores only the addresses of the blocks in the cache, not the blocks themselves), branch predictor, and scheduler. We formalize this as $\mathscr{A}(\langle m, a, buf, cs, bp, sc\rangle) = \langle buf\downarrow, cs, bp, sc\rangle$.

## VI. MECHANISMS FOR SECURE SPECULATION

In this section, we show how several recent proposals for hardware-level secure speculation can be cast within our framework and we study their security.

We analyze three countermeasures: (1) disabling speculation (**seq** in §VI-A), (2) delaying *all* speculative loads (**loadDelay** in §VI-B), and (3) employing hardware-level taint tracking and selectively delaying tainted instructions (**tt** in §VI-C). For each countermeasure **ctx**, we formalize its semantics using a

EXECUTE-LOAD-HIT

$$\frac{|buf| = i-1 \qquad a' = apl(buf,a) \qquad \textbf{spbarr} \notin buf \qquad \textbf{store}\ x', e' \notin buf \qquad x \neq \textbf{pc} \qquad (\!|e|\!)(a') \neq \bot \qquad access(cs,(\!|e|\!)(a')) = \texttt{Hit} \qquad update(cs,(\!|e|\!)(a')) = cs'}{\langle m,a,buf \cdot \textbf{load}\ x,e@T \cdot buf', cs, bp\rangle \xRightarrow{\textbf{execute}\ i} \langle m,a,buf \cdot x \leftarrow m((\!|e|\!)(a'))@T \cdot buf', cs', bp\rangle}$$

EXECUTE-BRANCH-ROLLBACK

$$\frac{|buf| = i-1 \qquad a' = apl(buf,a) \qquad \textbf{spbarr} \notin buf \qquad \ell_0 \neq \varepsilon \qquad p(\ell_0) = \textbf{beqz}\ x,\ell'' \qquad (a'(x) = 0 \wedge \ell \neq \ell'') \vee (a'(x) \in Vals \setminus \{0,\bot\} \wedge \ell \neq \ell_0 + 1) \qquad \ell' \in \{\ell'', \ell_0+1\} \setminus \{\ell\} \qquad bp' = update(bp,\ell_0,\ell')}{\langle m,a,buf \cdot \textbf{pc} \leftarrow \ell@\ell_0 \cdot buf', cs, bp\rangle \xRightarrow{\textbf{execute}\ i} \langle m,a,buf \cdot \textbf{pc} \leftarrow \ell'@\varepsilon, cs, bp'\rangle}$$

Fig. 6: Selected rules for **execute** $i$

relation $\Rightarrow_{\textbf{ctx}}$ obtained by modifying the hardware semantics from §V (which induces the corresponding trace semantics $\{\!|\cdot|\!\}_{\textbf{ctx}}$ in the usual way). Additionally, we characterize their security guarantees by showing which of the contracts from §III they satisfy; see Figure 8 for a summary of the results. An overview of the proofs is available in Appendix A, whereas detailed proofs of all results are given in [19].

Unless otherwise specified, all theorems hold for any instantiation of cache, branch predictor, and scheduler.

Before analyzing the countermeasures, we observe that *all* possible instances of the hardware semantics satisfy the $[\![\cdot]\!]^{\text{spec}}_{\text{ct}}$ contract, as stated in Theorem 1.

THEOREM 1. $\{\!|\cdot|\!\} \vdash [\![\cdot]\!]^{\text{spec}}_{\text{ct}}$.

From this, it immediately follows that *all* countermeasures presented below satisfy the $[\![\cdot]\!]^{\text{spec}}_{\text{ct}}$ contract as well.

### A. seq: Disabling speculation

A first, drastic countermeasure against speculative execution attacks is disabling speculative and out-of-order execution. To model this, we instantiate the hardware semantics by providing a sequential scheduler that produces directives in a **fetch** – **execute** 1 – **retire** order. The sequential scheduler, formalized in [19], works as follows:

- Whenever the reorder buffer is empty, the scheduler selects the **fetch** directive that adds entries to the buffer.
- If the first entry in the buffer is not resolved, the scheduler selects the **execute** 1 directive. Thus, the instruction is executed and, potentially, resolved.
- If the first entry in the buffer is resolved, the scheduler selects the **retire** directive. Therefore, the instruction is retired and its changes are written into the architectural state.

That is, the sequential scheduler ensures that instructions are executed in an in-order, non-speculative fashion.

As expected, instantiating the hardware semantics with the sequential scheduler (denoted with **seq**) results in strong security guarantees. As stated in Theorem 2, **seq** implements the $[\![\cdot]\!]^{\text{seq}}_{\text{ct}}$ interface that exposes only the program counter and the location of memory accesses under sequential execution.

THEOREM 2. $\{\!|\cdot|\!\}_{\textbf{seq}} \vdash [\![\cdot]\!]^{\text{seq}}_{\text{ct}}$.

### B. loadDelay: Delaying all speculative loads

Sakalis et al. [3] propose a family of countermeasures that delay memory loads to avoid leakage. In the following, we analyze the *eager delay of (speculative) loads* countermeasure. This countermeasure consists in delaying loads until all sources of mis-speculation have been resolved. We remark that the hardware semantics of Section V supports speculation only over branch instructions. Therefore, we model the **loadDelay** countermeasure by preventing loads whenever there are preceding, unresolved branch instructions in the reorder buffer. Using the terminology of [3], loads are delayed as long as they are under a so-called *control-shadow*.

We formalize the **loadDelay** countermeasure by modifying the STEP rule of the hardware semantics as follows (changes are highlighted in blue):

STEP-OTHERS

$$\frac{\langle m,a,buf,cs,bp\rangle \overset{d}{\Rightarrow} \langle m',a',buf',cs',bp'\rangle \qquad d = next(sc) \qquad sc' = update(sc, buf'\downarrow) \qquad d \in \{\textbf{fetch},\textbf{retire}\} \vee (d = \textbf{execute}\ i \wedge buf|_i \neq \textbf{load}\ x,e)}{\langle m,a,buf,cs,bp,sc\rangle \Rightarrow_{\textbf{loadDelay}} \langle m',a',buf',cs',bp',sc'\rangle}$$

STEP-EAGER-DELAY

$$\frac{\langle m,a,buf,cs,bp\rangle \overset{d}{\Rightarrow} \langle m',a',buf',cs',bp'\rangle \qquad d = next(sc) \qquad sc' = update(sc, buf'\downarrow) \qquad d = \textbf{execute}\ i \qquad buf|_i = \textbf{load}\ x,e \qquad \forall \textbf{pc} \leftarrow \ell@T \in buf[0..i-1].\ T = \varepsilon}{\langle m,a,buf,cs,bp,sc\rangle \Rightarrow_{\textbf{loadDelay}} \langle m',a',buf',cs',bp',sc'\rangle}$$

Fetching, retiring, and executing all instructions that are not loads work as before (see STEP-OTHERS rule). However, load instructions are executed only if all prior branch instructions are resolved (see STEP-NAIVE-DELAY rule). This is captured by requiring that all branch instructions in the buffer prefix have tag $\varepsilon$, i.e., $\forall \textbf{pc} \leftarrow \ell@T \in buf[0..i-1].\ T = \varepsilon$.

Thus, loads are delayed until they are guaranteed to be executed, while other instructions may be freely executed speculatively and out-of-order. Hence, no data memory accesses are performed on mis-speculated paths. However, maybe surprisingly, parts of the architectural state can still be leaked on mis-speculated paths as nested conditional branches may modify the instruction cache and the branch predictor state.

As a consequence, **loadDelay** violates the $[\![\cdot]\!]^{\text{seq}}_{\text{ct}}$ contract capturing the standard constant-time requirements.

**Example 2.** This program illustrates that $\{\!|\cdot|\!\}_{\textbf{loadDelay}} \not\vdash [\![\cdot]\!]^{\text{seq}}_{\text{ct}}$:

```
1 x = A[10]
2 y = not (A[20] | 1)
```

```
3  if (y) //branch always unsatisfied
4    if (x) //only executed speculatively
5      skip
```

Consider two configurations $\sigma$ and $\sigma'$ such that $\sigma(\texttt{A+10}) = 0$ and $\sigma'(\texttt{A+10}) = 1$. Then, $[\![p]\!]^{\text{seq}}_{\text{ct}}(\sigma) = [\![p]\!]^{\text{seq}}_{\text{ct}}(\sigma') = \texttt{load A+10} \cdot \texttt{load A+20} \cdot \texttt{pc}\ \bot$. However, the hardware can leak information through, e.g., the instruction cache if the branch at line 3 is speculatively taken. Then, the result of branch at line 4, which determines whether or not **skip** at line 5 is fetched, leaks whether `A[10]` (stored in `x`) is 0 or not, thereby distinguishing $\sigma$ and $\sigma'$.

To capture the guarantees offered by the eager-delay countermeasure, we can use the $[\![\cdot]\!]^{\text{seq-spec}}_{\text{ct-pc}}$ contract, which may intuitively be understood as $[\![\cdot]\!]^{\text{seq}}_{\text{ct}} + [\![\cdot]\!]^{\text{spec}}_{\text{pc}}$, i.e., control-flow and memory accesses are leaked under sequential execution, and in addition, the program counter is leaked during speculative execution. This new contract is satisfied by the countermeasure, leading to Theorem 3.

THEOREM 3. $\{\!|\cdot|\!\}_{\textbf{loadDelay}} \vdash [\![\cdot]\!]^{\text{seq-spec}}_{\text{ct-pc}}$.

As the control flow during speculative execution may only depend upon data previously loaded non-speculatively, the security of the countermeasure can also be captured by $[\![\cdot]\!]^{\text{seq}}_{\text{arch}}$.

THEOREM 4. $\{\!|\cdot|\!\}_{\textbf{loadDelay}} \vdash [\![\cdot]\!]^{\text{seq}}_{\text{arch}}$.

### C. tt: Taint tracking of speculative values

Recent work [4], [5] propose to track transient computations and to selectively delay instructions involving tainted information. While these proposals slightly differ in how instructions are labelled and on the effects of different labels, they share the same building blocks and provide similar guarantees.

For this reason, we start by presenting an overview of the Speculative Taint Tracking (STT) [5] and Non-speculative Data Access (NDA) [4] countermeasures. Next, we introduce a general extension to the hardware semantics from Section V for supporting taint tracking schemes. We continue by formalizing a countermeasure inspired by STT and we discuss its security guarantees, and we conclude by discussing NDA.

*1) Overview:* STT [5] and NDA [4] are two recent taint tracking proposals for secure speculation. These countermeasures extend a processor with hardware-level taint tracking to track whether data has been retrieved by a speculatively executed instruction. The taint tracking mechanism propagates taint through the computation and whenever operations are no longer transient, the taint is removed. Finally, both NDA and STT selectively delay tainted operations to avoid leaks.

The main difference between the two approaches is that while STT delays the *execution* of tainted transmit instructions (that is, instructions like loads that might leak information), NDA adopts a more conservative approach that delays the *propagation* of data from tainted instructions.

*2) Supporting taint tracking:* To support taint tracking, we label entries in the reorder buffer with sets of reorder buffer's indexes. A labeled command is of the form $\langle i@T \rangle_L$ where $i@T$ is a reorder buffer entry and $L \subset \mathbb{N}$ is a label, i.e., the set of indexes of the entries $i@T$ depends on.

Existing proposals differ in (1) how labels are assigned and propagated, and (2) how labels affect the processor's execution. To accommodate different variants for (1) and (2), we formalize these aspects using two functions:

- The *labeling function* $lbl(buf_{ul}, buf, d)$ computes the new labels associated with the (unlabeled) buffer $buf_{ul}$ given the old labeled buffer $buf$ and the directive $d$ determining the activated pipeline step. This function models how the tracking works, i.e., how labels are assigned to new instructions and how they are propagated.
- The *unlabeling function* $unlbl(buf, d)$ produces an unlabeled buffer $buf_{ul}$ starting from a labeled buffer $buf$ and a directive $d$. This function models how labels affect the processor's semantics in terms of changes to the reorder buffer (and these changes might depend on the executed pipeline step modeled by $d$).

We describe later how these functions can be instantiated to model STT and NDA.

We formalize the **tt** countermeasure by modifying the STEP rule as follows (changes are highlighted in blue):

$$\textsc{Step}\quad \frac{d = next(sc) \qquad buf_{ul} = unlbl(buf, d) \qquad \langle m, a, buf_{ul}, cs, bp\rangle \overset{d}{\Rightarrow} \langle m', a', buf'_{ul}, cs', bp'\rangle \qquad buf' = lbl(buf'_{ul}, buf, d) \qquad sc' = update(sc, buf'\downarrow)}{\langle m, a, buf, cs, bp, sc\rangle \Rightarrow_{\textbf{tt}} \langle m', a', buf', cs', bp', sc'\rangle}$$

The rule differs from the standard STEP rule in three ways:

- Entries in the reorder buffer are labelled.
- Before activating a step in the pipeline, i.e., before applying one step of $\overset{d}{\Rightarrow}$, we use the unlabeling function to derive an unlabeled buffer $buf_{ul} = unlbl(buf, d)$ representing how labels affect the reorder buffer entries.
- The buffer produced by the application of $\overset{d}{\Rightarrow}$ is labeled by invoking the labeling function $buf' = lbl(buf'_{ul}, buf, d)$. Therefore, the labels in $buf'$ are updated to track the information flows through the computation.

*3) Speculative taint tracking:* Here we present how to model a countermeasure inspired by STT [5]. As mentioned above, STT tracks whether data depends on speculatively accessed data and delays the execution of transient transmit instructions. These features are reflected in our model:

- In $\mu$ASM, there are three kinds of *transmit instructions*: loads **load** $x, e$, stores **store** $x, e$, and assignments to the program counter **pc** $\leftarrow e$. We write $transmit(i@T)$ whenever the instruction $i$ is a transmit instruction.
- The labeling function, formalized in [19], specifies how newly fetched instructions are labeled as well as how labels are updated during computation, and it works as follows:

– Newly fetched **load** $x, e$ instructions are labelled with the indexes of the unresolved branch instruction in the buffer. That is, non-transient loads are labelled with $\emptyset$, whereas potentially transient loads have a non-empty label. In contrast, newly fetched assignments $x \leftarrow e$ are labelled with the union of the labels associated with the registers occurring in $e$. That is, assignments that depend only on non-transient values are labelled with $\emptyset$, whereas those depending on potentially

$$
\begin{aligned}
unlbl(buf, \textbf{fetch}) &= mask(buf)\\
unlbl(buf, \textbf{retire}) &= drop(buf)\\
unlbl(buf, \textbf{execute } i) &= \begin{cases} mask(buf) & \text{if } transmit(buf|_i)\\ drop(buf) & \text{otherwise}\end{cases}\\
drop(\varepsilon) &:= \varepsilon\\
drop(\langle i@T\rangle_L \cdot buf) &:= i@T \cdot drop(buf)\\
mask(\varepsilon) &:= \varepsilon\\
mask(\langle i@T\rangle_L \cdot buf) &:= \begin{cases} x \leftarrow \bot@T \cdot mask(buf) & \text{if } L = \emptyset \wedge\\ & \quad i = x \leftarrow e\\ i@T \cdot mask(buf) & \text{otherwise}\end{cases}
\end{aligned}
$$

Fig. 7: Unlabeling function $unlbl(buf, d)$ for STT

transient values have a non-empty label. All other newly fetched instructions are labelled with an empty label.

– When we retire an instruction, all indexes in labels are decremented by 1 and indexes reaching 0 are removed. This ensures that indexes are still consistent with the, now shorter, reorder buffer.

– When we execute non-branch instructions, labels are preserved.

– When we execute and resolve a branch instruction (thereby eliminating one of the sources of speculation), we remove its index from later commands' labels. As a result, some later commands may now have an empty label, i.e., they are certainly non-transient.

Overall, the labeling function ensures that reorder buffer entries that may depend on transiently retrieved data are labelled with a non-empty label at every point of the computation.

- To delay *only* transmit instructions, the unlabeling function, defined in Figure 7, replaces assignments $x \leftarrow e$ whose label is non-empty with $x \leftarrow \bot$ for **fetch** and **execute** $i$ directives when the $i$-th entry in the buffer is a transmit instruction. This ensures that transmit instructions are not executed whenever they depend on possibly transient data, which are now mapped to $\bot$. In contrast, the unlabeling function simply strips the taint tracking labels for **retire** and **execute** $i$ directives whenever the $i$-the entry is not a transmit instruction; thereby allowing the hardware to freely execute non-transmit instructions.

Concretely, **tt** delays all transmit instructions that depend on transiently retrieved data. However, **tt** does not delay transient loads that depend on non-transient data, as acknowledged also in [5]. This means that parts of the architectural state can be leaked using speculatively executed instructions. As Example 3 shows, **tt** violates the $[\![\cdot]\!]^{\text{seq}}_{\text{ct}}$ contract.

**Example 3.** Consider the Spectre v1 variant from Figure 1b, compiled to $\mu$ASM:

```
1 load z,A + y //accessing A[y]
2 x ← y < size_A
3 beqz x, ⊥ //checking y < size_A
4 z ← z*64
5 load w, B+z //accessing B[A[y]*64]
```

Consider two configurations $\sigma$ and $\sigma'$ that agree on the values of `A`, `B`, `y`, and `size_A` and for which $\sigma(\texttt{y}) > \sigma(\texttt{size\_A})$, i.e., the array `A` is speculatively accessed out of bounds. Furthermore, assume that $\sigma(\texttt{A + y}) = 0$ and $\sigma'(\texttt{A + y}) = 1$. Then, $[\![p]\!]^{\text{seq}}_{\text{ct}}(\sigma) = [\![p]\!]^{\text{seq}}_{\text{ct}}(\sigma') = \texttt{load A+y} \cdot \texttt{pc } \bot$. However, the hardware semantics can potentially leak information through the data cache if the hardware speculatively executes the load on line 5. Indeed, the load on line 1 is labeled with $\emptyset$ since it is *not* transient. Hence, the load operation on line 5, which depends on the result of line 1, is not delayed (even though transient operations relying on its result would be delayed since line 5 is labeled with the index corresponding to the unresolved branch in line 3). By probing the state of the cache an attacker can determine whether `A[y] = 0` or `A[y] = 1`, thereby distinguishing $\sigma$ and $\sigma'$.

One way to characterize the guarantees provided by the **tt** countermeasure is with the $[\![\cdot]\!]^{\text{spec}}_{\text{ct}}$ contract.

THEOREM 5. $\{\!|\cdot|\!\}_{\textbf{tt}} \vdash [\![\cdot]\!]^{\text{spec}}_{\text{ct}}$.

However, we remark that this contract is already satisfied by the baseline hardware defined in Section V without any countermeasures. A more meaningful characterization of **tt**'s guarantees, stated in Theorem 6, is via the $[\![\cdot]\!]^{\text{seq}}_{\text{arch}}$ contract. Intuitively, **tt** satisfies $[\![\cdot]\!]^{\text{seq}}_{\text{arch}}$ as it prevents the execution of transmit instructions based on transiently retrieved data.

THEOREM 6. $\{\!|\cdot|\!\}_{\textbf{tt}} \vdash [\![\cdot]\!]^{\text{seq}}_{\text{arch}}$.

Theorem 6 confirms the results of [5] and provides a clean characterization of the *transient noninterference* [5] guarantees in terms of the $[\![\cdot]\!]^{\text{seq}}_{\text{arch}}$ contract.

*4) Non-speculative data access:* Weisse et al. [4] propose NDA, a family of countermeasures for secure speculation that also relies on hardware taint tracking. In a nutshell, NDA delays the propagation of speculatively executed instructions until the corresponding speculation sources have been resolved. NDA comes with two different propagation strategies—*strict* and *permissive* propagation—that can be modeled as follows:

- For both propagation strategies, the unlabeling function simply replaces all assignments $x \leftarrow e$ whose label is non-empty with $x \leftarrow \bot$, thereby preventing the propagation of potentially transient data. This differs from STT where labels are sometimes removed to allow the propagation of potentially transient data, as long as the propagation does not lead to leaks.
- The labeling function differs from the one in **tt** in how newly fetched instructions are labeled. For the strict strategy, *all* newly fetched transient instructions are labelled with the indexes of unresolved branch instructions. In contrast, *only* newly fetched transient **load**s are labeled with the indexes of unresolved branch instructions under the permissive strategy.

Despite these differences, NDA provides similar guarantees to **tt**. That is, it satisfies the $[\![\cdot]\!]^{\text{spec}}_{\text{ct}}$ and $[\![\cdot]\!]^{\text{seq}}_{\text{arch}}$ contracts.

### D. Summary

Figure 8 summarizes the results of this section in the lattice structure established in §III-C. This yields the first rigorous comparison of the security guarantees of mechanisms for

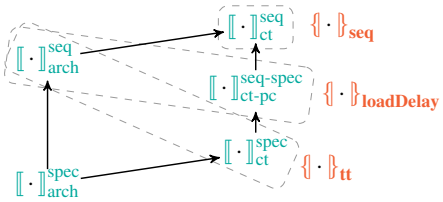

Fig. 8: Security guarantees of secure-speculation mechanisms.

secure speculation, and it translates the results from §IV into a principled basis for programming them securely.

## VII. Discussion

### A. Scope of the model

With our modeling of a generic microarchitecture and corresponding side-channel adversaries (§V), we aim to strike a balance between capturing the central aspects of attacks on speculative and out-of-order processors, while obtaining a general and tractable model.

As a consequence, we simplify many aspects of modern processors. For instance, we model only a simple 3-stage pipeline, single threaded, and with conditional branch prediction as the only source of speculation. Likewise, we consider an adversary that can observe instructions in the reorder buffer and memory blocks in the cache, but not the data they carry.

This modelling is adequate for reasoning about protections against variants of Spectre v1. However, it does not encompass features such as store-to-load forwarding or prediction over memory aliasing, or adversaries that can observe leaks from internal processor buffers, such as those exploited in data-sampling attacks [30], [31].

As a consequence, Theorems 1–6 need not extend to these scenarios. However, our framework for expressing contracts is not limited to this simple model, as we discuss next.

### B. Beyond Spectre v1

We now discuss how to extend our framework to other transient execution attacks. For each attack, we discuss how to (1) extend our contracts, and (2) adjust our hardware semantics:

- *Spectre-BTB and Spectre-RSB*: These variants speculate respectively over indirect jumps and return instructions. To support them, the spec-contracts can be extended to explore all possible mispredicted paths for a bounded number of steps before rolling back (similarly to the Branch rule in Figure 3). Moreover, our hardware semantics $\{|\cdot|\}$ can also easily be extended to handle these new forms of speculation. For instance, speculation over indirect jumps could be modeled similarly to the Fetch-Branch-Hit rule in §V.
- *Spectre-STL*: This variant speculates over memory aliasing over in-flight store and load operations. Extending our contracts to handle this new kind of speculation requires to modify the spec-contracts to model the effects of store-to-load forwarding resulting from memory aliasing predictions. This could be done similarly to Pitchfork [13]. That is, the spec semantics could keep track of the issued **store** $x,e$ instructions. Then, whenever a **load** $y,e'$ instruction is executed, one could explore multiple paths representing all possible aliasing predictions for a fixed number of steps and later roll back. Finally, the $\{|\cdot|\}$ semantics can be extended to support Spectre-STL similarly to other semantics [13], [14], [32].
- *Straight-line speculation*: Some CPUs can speculatively execute instructions that follow straight after unconditional jumps or function returns [33], which may result in speculative leaks. A corresponding contract can be captured by an execution mode (similar to spec) that explores instructions following unconditional changes in control flow up to a bounded depth and exposes the corresponding observations, whereas our hardware semantics $\{|\cdot|\}$ can be extended to support this form of speculation similarly to Fetch-Branch-Hit rule in §V.
- *Meltdown and MDS*: In Spectre-type attacks, transient execution is caused by control- and data-flow mispredictions. In Meltdown-type [2] attacks (encompassing both Meltdown [34] and data sampling attacks [30], [31]), transient execution is caused by instruction faults or microcode assists. Contracts for processors that are vulnerable to this kind of attacks would need to expose large portions of the memory space, which makes secure programming challenging.

### C. Uses of contracts

The contracts we propose in this paper are designed to adequately capture the security guarantees offered by existing mechanisms for secure speculation, while exposing tractable verification conditions for software. We envision hardware vendors to produce such contracts for their CPUs, to enable users to reason about software security without exposing details of the microarchitecture, and to provide a baseline against which to validate the vendors' security claims.

Moreover, rather than trying to infer contracts for a microarchitecture that has not been designed with security in mind, our framework can serve as a basis for a clean-slate approach, where one starts from a desired security contract and aims to design microarchitectures that optimize performance within these constraints.

## VIII. Related work

**Speculative execution attacks:** These attacks exploit microarchitectural side-effects of speculatively executed instructions to leak information. There exist many Spectre [1] variants that differ in the exploited speculation sources [35], [36], [37], the covert channels [38], [39], [40] used, or the target platforms [41]. We refer to [2], [42] for a survey.

**Hardware-level countermeasures:** Here, we review proposals that we have not formalized in §VI:

- "Redo"-based countermeasures [6], [7], [9] execute speculative memory operations on shadow cache structures. Once a memory operation becomes non-speculative, its effects are replicated on the standard cache hierarchy by re-executing the operation. While these countermeasures satisfy $\llbracket\cdot\rrbracket^{\text{spec}}_{\text{ct}}$, they likely violate $\llbracket\cdot\rrbracket^{\text{seq}}_{\text{arch}}$ as they still modify other parts of the microarchitectural state such as the reorder buffer. This intuition is confirmed by speculative interference attacks [43], which demonstrate how to convert seemingly transient changes to the reorder buffer into persistent cache-state changes.

- In contrast, "Undo"-based countermeasures [10] mitigate Spectre attacks by rolling back the effects of speculatively executed instructions on the cache. Such countermeasures provide security against adversaries that observe the final cache state, but they likely do not provide guarantees against the trace-based attackers we consider in this paper.
- Delay-based mitigations selectively delay the execution of some instructions to prevent speculative leaks. In addition to the **loadDelay** countermeasure studied in §VI-B, Sakalis et al. [3] propose a more permissive scheme, similar to conditional speculation [44], where only loads resulting in cache misses are delayed. These countermeasures, however, would violate the $[\![\cdot]\!]^{\text{seq}}_{\text{arch}}$ and $[\![\cdot]\!]^{\text{seq-spec}}_{\text{ct-pc}}$ contracts because cache hits would still leak information.

SpecShield [45] proposes two countermeasures: one similar to eager-delay and the other similar to NDA's permissive strategy, with similar guarantees as those of **loadDelay** and **tt**.

Finally, some proposals, like [46], [47], improve efficiency by only delaying instructions that may leak program-level sensitive information. This is achieved by either considering all user-provided data as untrusted [46] or by allowing the specification of program-level policies [47].

**Formal microarchitectural models:** While several works [48], [49], [50] present formal architectural models for (parts of) the ARMv8-A, RISC-V, MIPS, and x86 ISAs, only recently researchers started to focus on formal models of microarchitectural aspects. For instance, Coppelia [51] is a tool to automatically generate software exploits for hardware designs.

The speculative semantics from [11] forms the basis for the $[\![\cdot]\!]^{\text{spec}}_{\text{ct}}$ contract that exposes the effects of speculatively executed instructions. In contrast to [11], other semantics [13], [14], [18], [32] more closely resemble the actual microarchitectural behavior of out-of-order processors with multiple pipeline stages, rather than concisely capturing the resulting leakage. Specifically, the hardware semantics $\{\!|\cdot|\!\}$ in §V extends [13], [18]'s semantics by making explicit the dependencies with caches, predictors, and pipeline scheduler. Finally, Disselkoen et al. [52] presents a speculative semantics based on ideas from recent advances on relaxed memory models.

Fadiheh et al. [53] propose a SAT-based boundel model checking methodology to check whether a given register-transfer level (RTL) processor design exhibits covert channel vulnerabilities. In the terminology of our work, they check whether the processor design satisfies the $[\![\cdot]\!]^{\text{seq}}_{\text{arch}}$ contract.

**HW-SW contracts for side channels:** Recently, researchers [54], [55] have been calling for new hardware-software contracts that expose security-relevant microarchitectural details. We answer this call by providing contracts for secure speculation and by showing how they can be leveraged at the software level.

Recent work [56], [57] presents extensions to the RISC-V ISA where data is labeled, e.g., as *Public* or *Secret*; labels are tracked during the computation; and the microarchitecture ensures that secret data does not leak. This work is orthogonal to ours in that we characterize the security of different hardware-level countermeasures for a standard ISA.

## IX. Conclusions

Motivated by a lack of hardware-software contracts that support principled co-design for secure speculation, we presented a framework for specifying such contracts.

On the hardware side, we used our framework to provide the first uniform characterization of guarantees provided by a representative set of mechanisms for secure speculation.

On the software side, we used our framework to characterize secure programming in two scenarios—"constant-time programming" and "sandboxing"—and we show how to automate checks for programs to run securely on top of these mechanisms.

*Acknowledgments:* Pepe Vila's work was done while at Microsoft Research. We would like to thank David Chisnall, Muntaquim Chowdhury, Matthew Fernandez, Cédric Fournet, Carlos Rozas, and Gururaj Saileshwar for feedback and discussions. This work was supported by a grant from Intel Corporation, Atracción de Talento Investigador grant 2018-T2/TIC-11732A, Juan de la Cierva-Formación grant FJC2018-036513-I, Spanish project RTI2018-102043-B-I00 SCUM, and Madrid regional project S2018/TCS-4339 BLOQUES.

## References

[1] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution," in *Proceedings of the 40th IEEE Symposium on Security and Privacy*, ser. S&P '19. IEEE, 2019.

[2] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A Systematic Evaluation of Transient Execution Attacks and Defenses," in *Proceedings of the 28th USENIX Security Symposium*, ser. USENIX Security '19. USENIX Association, 2019.

[3] C. Sakalis, S. Kaxiras, A. Ros, A. Jimborean, and M. Själander, "Efficient invisible speculative execution through selective delay and value prediction," in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA '19. ACM, 2019.

[4] O. Weisse, I. Neal, K. Loughlin, T. F. Wenisch, and B. Kasikci, "NDA: Preventing speculative execution attacks at their source," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-52. IEEE/ACM, 2019.

[5] 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*, ser. MICRO-52. IEEE/ACM, 2019.

[6] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. 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/ACM, 2018.

[7] K. N. Khasawneh, E. M. Koruyeh, C. Song, D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "SafeSpec: Banishing the spectre of a meltdown with leakage-free speculation," in *Proceedings of the 56th Annual Design Automation Conference 2019*, ser. DAC '19. ACM, 2019.

[8] V. Kiriansky, I. A. Lebedev, S. P. Amarasinghe, S. Devadas, and J. S. Emer, "DAWG: A defense against cache timing attacks in speculative execution processors," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, ser. ISCA '18. IEEE, 2018.

[9] S. Anisworth and T. M. Jones, "Muontrap: Preventing cross-domain spectre-like attacks by capturing speculative state," in *Proceedings of the 47th International Symposium on Computer Architecture*, ser. ISCA '20. ACM, 2020.

[10] G. Saileshwar and M. K. Qureshi, "Cleanupspec: An "Undo" approach to safe speculation," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-52. IEEE/ACM, 2019.

[11] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez, "SPECTECTOR: Principled detection of speculative information flows," in *Proceedings of the 41st IEEE Symposium on Security and Privacy*, ser. S&P '20. IEEE, 2020.

[12] G. Barthe, G. Betarte, J. Campo, C. Luna, and D. Pichardie, "System-level non-interference for constant-time cryptography," in *Proceedings of the 21st ACM Conference on Computer and Communications Security*, ser. CCS '14. ACM, 2014.

[13] S. Cauligi, C. Disselkoen, K. v. Gleissenthall, D. Stefan, T. Rezk, and G. Barthe, "Towards constant-time foundations for the new spectre era," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '20. ACM, 2020.

[14] M. Balliu, M. Dam, and R. Guanciale, "InSpectre: Breaking and fixing microarchitectural vulnerabilities by formal analysis," in *Proceedings of the 27th ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '20. ACM, 2020.

[15] C. Carruth, "Speculative load hardening," 2018. [Online]. Available: http://releases.llvm.org/8.0.0/docs/SpeculativeLoadHardening.html

[16] M. Miller, "Mitigating speculative execution side channel hardware vulnerabilities," https://blogs.technet.microsoft.com/srd/2018/03/15/mitigating-speculative-execution-side-channel-hardware-vulnerabilities/, 2018.

[17] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying constant-time implementations," in *Proceedings of the 26th USENIX Security Symposium*, ser. USENIX Security '16. USENIX Association, 2016.

[18] M. Vassena, K. v. Gleissenthall, R. G. Kici, D. Stefan, and R. Jhala, "Automatically eliminating speculative leaks with Blade," *CoRR*, vol. abs/2005.00294, 2020.

[19] M. Guarnieri, B. Köpf, J. Reineke, and P. Vila, "Hardware-software contracts for secure speculation," *CoRR*, vol. abs/2006.03841v2, 2020. [Online]. Available: https://arxiv.org/abs/2006.03841v2

[20] G. Doychev and B. Köpf, "Rigorous Analysis of Software Countermeasures against Cache Attacks," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementations*, ser. PLDI '17. ACM, 2017.

[21] G. Barthe, G. Betarte, J. D. Campo, and C. Luna, "System-level non-interference of constant-time cryptography. part I: model," *Journal of Automatic Reasoning*, vol. 63, no. 1, 2019.

[22] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "SMoTherSpectre: Exploiting speculative execution through port contention," in *Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19. ACM, 2019.

[23] J. Landauer and T. Redmond, "A lattice of information," in *Proceedings of the 6th IEEE Computer Security Foundations Workshop*, ser. CSFW '93. IEEE, 1993.

[24] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native client: A sandbox for portable, untrusted x86 native code," *Communications of the ACM*, vol. 53, no. 1, Jan. 2010.

[25] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with WebAssembly," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '17. ACM, 2017.

[26] B. Rodrigues, F. M. Quintão Pereira, and D. F. Aranha, "Sparse representation of implicit flows with applications to side-channel detection," in *Proceedings of the 25th International Conference on Compiler Construction*, ser. CC '16. ACM, 2016.

[27] D. Molnar, M. Piotrowski, D. Schultz, and D. A. Wagner, "The program counter security model: Automatic detection and removal of control-flow side channel attacks," in *Proceedings of the 8th International Conference on Information Security and Cryptology*, ser. ICISC '05. Springer, 2005.

[28] S. Cauligi, G. Soeller, B. Johannesmeyer, F. Brown, R. S. Wahby, J. Renner, B. Grégoire, G. Barthe, R. Jhala, and D. Stefan, "FaCT: A DSL for timing-sensitive computation," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '19. ACM, 2019.

[29] G. Barthe, S. Blazy, B. Grégoire, R. Hutin, V. Laporte, D. Pichardie, and A. Trieu, "Formal verification of a constant-time preserving C compiler," *Proceedings of the ACM on Programming Languages*, vol. 4, no. POPL, 2019.

[30] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "RIDL: Rogue in-flight data load," in *Proceedings of the 40th IEEE Symposium on Security and Privacy*, ser. S&P '19. IEEE, 2019.

[31] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "ZombieLoad: Cross-privilege-boundary data sampling," in *Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19. ACM, 2019.

[32] R. McIlroy, J. Sevcík, T. Tebbi, B. L. Titzer, and T. Verwaest, "Spectre is here to stay: An analysis of side-channels and speculative execution," *CoRR*, vol. abs/1902.05178, 2019.

[33] ARM, "Whitepaper: Straight-line speculation," https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability/downloads/straight-line-speculation, 2020.

[34] 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 *Proceedings of the 27th USENIX Security Symposium*, ser. USENIX Security'18. USENIX Association, 2018.

[35] G. Maisuradze and C. Rossow, "Ret2Spec: Speculative Execution Using Return Stack Buffers," in *Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. ACM, 2018.

[36] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre returns! speculation attacks using the return stack buffer," in *Proceedings of the 12th USENIX Workshop on Offensive Technologies*, ser. WOOT '18. USENIX Association, 2018.

[37] J. Horn, "CVE-2018-3639 - speculative store bypass," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-3639, 2018.

[38] C. Trippel, D. Lustig, and M. Martonosi, "MeltdownPrime and SpectrePrime: Automatically-synthesized attacks exploiting invalidation-based coherence protocols," *CoRR*, vol. abs/1802.03802, 2018.

[39] M. Schwarz, M. Schwarzl, M. Lipp, and D. Gruss, "NetSpectre: Read arbitrary memory over network," in *Proceedings of the 24th European Symposium on Research in Computer Security*, ser. ESORICS '19. Springer, 2019.

[40] J. Stecklina and T. Prescher, "LazyFP: Leaking FPU register state using microarchitectural side-channels," *CoRR*, vol. abs/1806.07480, 2018.

[41] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "Stealing Intel secrets from SGX enclaves via speculative execution," in *Proceedings of the 4th IEEE European Symposium on Security and Privacy*, ser. EuroS&P '19. IEEE, 2019.

[42] W. Xiong and J. Szefer, "Survey of transient execution attacks," *CoRR*, vol. abs/2005.13435, 2020.

[43] M. Behnia, P. Sahu, R. Paccagnella, J. Yu, Z. Zhao, X. Zou, T. Unterluggauer, J. Torrellas, C. V. Rozas, A. Morrison, F. McKeen, F. Liu, R. Gabor, C. W. Fletcher, A. Basak, and A. R. Alameldeen, "Speculative interference attacks: Breaking invisible speculation schemes," *CoRR*, vol. abs/2007.11818, 2020.

[44] P. Li, L. Zhao, R. Hou, L. Zhang, and D. Meng, "Conditional speculation: An effective approach to safeguard out-of-order execution against spectre attacks," in *Proceedings of the 25th IEEE International Symposium on High Performance Computer Architecture*, ser. HPCA '19, 2019.

[45] K. Barber, A. Bacha, L. Zhou, Y. Zhang, and R. Teodorescu, "SpecShield: Shielding speculative data from microarchitectural covert channels," in *Proceedings of the 28th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '19. IEEE, 2019.

[46] M. Taram, A. Venkat, and D. Tullsen, "Context-sensitive fencing: Securing speculative execution via microcode customization," in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. ACM, 2019.

[47] M. Schwarz, M. Lipp, C. Canella, R. Schilling, F. Kargl, and D. Gruss, "ConTExT: A generic approach for mitigating spectre," in *Proceedings of the 27th Annual Network and Distributed System Security Symposium*, ser. NDSS '20. Internet Society, 2020.

[48] A. Armstrong, T. Bauereiss, B. Campbell, A. Reid, K. E. Gray, R. M. Norton, P. Mundkur, M. Wassell, J. French, C. Pulte, S. Flur, I. Stark, N. Krishnaswami, and P. Sewell, "ISA semantics for ARMv8-A, RISC-V, and CHERI-MIPS," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, 2019.

[49] U. Degenbaev, "Formal specification of the x86 instruction set architecture," Ph.D. dissertation, Universität des Saarlandes, 2012.

[50] S. Goel, W. A. Hunt, and M. Kaufmann, "Engineering a formal, executable x86 ISA simulator for software verification," in *Provably Correct Systems*. Springer, 2017.

[51] R. Zhang, C. Deutschbein, P. Huang, and C. Sturton, "End-to-end automated exploit generation for validating the security of processor designs," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-51. IEEE/ACM, 2018.

[52] C. Disselkoen, R. Jagadeesan, A. Jeffrey, and J. Riely, "Code that never ran: modeling attacks on speculative evaluation," in *Proceedings of the 40th IEEE Symposium on Security and Privacy*, ser. S&P '19. IEEE, 2019.
[53] M. R. Fadiheh, D. Stoffel, C. W. Barrett, S. Mitra, and W. Kunz, "Processor hardware security vulnerabilities and their detection by unique program execution checking," in *Proceedings of the 19th Conference on Design, Automation & Test in Europe*, ser. DATE'19. IEEE, 2019.
[54] G. Heiser, "For safety's sake: We need a new hardware-software contract!" *IEEE Design and Test*, vol. 35, 2018.
[55] Q. Ge, Y. Yarom, and G. Heiser, "No security without time protection: We need a new hardware-software contract," in *Proceedings of the 9th Asia-Pacific Workshop on Systems*, ser. APSys '18. ACM, 2018.
[56] J. Yu, L. Hsiung, M. E. Hajj, and C. W. Fletcher, "Data oblivious ISA extensions for side channel-resistant and high performance computing," in *Proceedings of the 27th Annual Network and Distributed System Security Symposium*, ser. NDSS '20. Internet Society, 2020.
[57] D. Zagieboylo, G. E. Suh, and A. C. Myers, "Using information flow to design an ISA that controls timing channels," in *Proceedings of the 32nd IEEE Symposium on Computer Security Foundations*, ser. CSF '19. IEEE, 2019.

## APPENDIX A
## PROOF OVERVIEW

Theorems 1–6 are statements about contract satisfaction of the form $\{\!|\cdot|\!\} \vdash [\![\cdot]\!]$. For their proof, we need to show that, given an arbitrary program $p$ and two arbitrary architectural states $\sigma$, $\sigma'$ such that $[\![p]\!](\sigma) = [\![p]\!](\sigma')$, then $\{\!|p|\!\}(\sigma) = \{\!|p|\!\}(\sigma')$. The proofs follow a common structure, which we outline in this section. For full details, see [19].

**Notation:** In the following, $\mathrm{cr}$, $\mathrm{cr}'$ denote the contract runs corresponding to the traces $[\![p]\!](\sigma)$ and $[\![p]\!](\sigma')$. Similarly, $\mathbf{hr}$, $\mathbf{hr}'$ denote the hardware runs corresponding to $\{\!|p|\!\}(\sigma)$ and $\{\!|p|\!\}(\sigma')$. Moreover, $\mathbf{hr}(i)$ and $\mathrm{cr}(i)$ respectively denote $\mathbf{hr}$'s and $\mathrm{cr}$'s $i$-th state. All proofs rely on four main components:

- an indistinguishability relation $\approx$ between hardware states,
- relations $\equiv_j$ between hardware and contract states,
- a correspondence function $corr_{\mathrm{cr},\mathbf{hr}}(\cdot)$ that maps hardware states in $\mathbf{hr}$ to $\equiv_j$-related contract states in $\mathrm{cr}$, and
- an indistinguishability lemma capturing under which conditions one step of the hardware semantics $\Rightarrow$ preserves state indistinguishability $\approx$.

We next describe each of these components and outline how they are combined for proofs of $\{\!|\cdot|\!\} \vdash [\![\cdot]\!]$.

**State indistinguishability relation $\approx$:** Two hardware states related by $\approx$ must be indistinguishable by the microarchitectural adversary from §V-C. In particular, this implies that they execute the same pipeline step. Additionally, $\approx$ captures proof invariants that are associated with specific hardware semantics. For instance, for Theorem 1, $\approx$ requires agreement on **pc**-values, whereas for Theorem 6, $\approx$ requires agreement on register assignments and reorder buffers entries with empty labels.

**Hardware-contract relation $\equiv_j$:** This is a family of relations between contract states $\mathrm{c}$ and hardware states $\mathbf{h}$, where, intuitively, $\mathrm{c} \equiv_j \mathbf{h}$ holds if contract state $\mathrm{c}$ is related to hardware state $\mathbf{h}$ when considering the prefix of $\mathbf{h}$'s reorder buffer of length $j$.

The specific definition of $\equiv_j$ depends on the considered contract: For instance, in Theorems 2, 4, and 6, which consider contracts with $\mathrm{seq}$ execution mode, $\mathrm{c} \equiv_j \mathbf{h}$ iff $\mathrm{c}$ is equivalent to the architectural state obtained by applying all commands in $\mathbf{h}$'s reorder buffer up to the $j$-th instruction to the memory and registers in $\mathbf{h}$. In contrast, for Theorems 1, 3, and 5, which consider contracts with $\mathrm{spec}$ execution mode, $\mathrm{c} \equiv_j \mathbf{h}$ additionally require that invariants related with speculative execution are satisfied. For instance, $\mathrm{c} \equiv_j \mathbf{h}$ ensures that all mispredicted branch instructions in $\mathbf{h}$'s buffer correspond to configurations whose speculative window is not $\infty$ (denoting non-speculative execution) in $\mathrm{c}$.

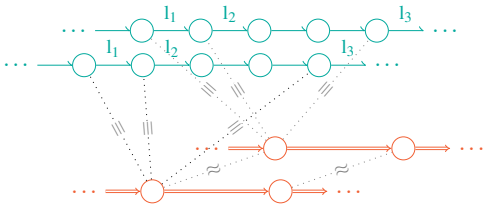

Fig. 9: Indistinguishability lemma

**Correspondence $corr_{\mathrm{cr},\mathbf{hr}}(\cdot)$:** The reorder buffer of a hardware state can contain data corresponding to multiple contract-level instructions. We use the function $corr_{\mathrm{cr},\mathbf{hr}}(\cdot)$ to map *prefixes* of the reorder buffer of hardware states to unique contract states. Specifically, $corr_{\mathrm{cr},\mathbf{hr}}(i)(j) = k$ indicates that the reorder buffer prefix of length $j$ of $\mathbf{hr}(i)$ is mapped to $\mathrm{cr}(k)$. In [19], we construct correspondences for the $\mathrm{seq}$ and $\mathrm{spec}$ execution modes such that:

- if $corr_{\mathrm{cr},\mathbf{hr}}(i)(j) = k$, then $\mathrm{cr}(k) \equiv_j \mathbf{hr}(i)$, and
- $corr_{\mathrm{cr},\mathbf{hr}}(i) = corr_{\mathrm{cr}',\mathbf{hr}'}(i)$ for all $i$.

**Indistinguishability lemma:** The indistinguishability lemma is the key intermediate result in our contract satisfaction proofs. The lemma states that, given two reachable and indistinguishable hardware states $\mathbf{h}$, $\mathbf{h}'$, if for all prefixes *buf* of $\mathbf{h}$'s reorder buffer, there are contract states $\mathrm{c}$ and $\mathrm{c}'$ such that (a) $\mathrm{c} \equiv_{|buf|} \mathbf{h}$ and $\mathrm{c}' \equiv_{|buf|} \mathbf{h}'$, and (b) $\mathrm{c}$ and $\mathrm{c}'$ produce the same observation $\mathrm{l}$ when executing one step of the contract semantics $\rightharpoonup$, then either $\mathbf{h}$ and $\mathbf{h}'$ are stuck or doing one step of the hardware semantics $\Rightarrow$ preserves the indistinguishability. See Figure 9 for a visualization. In [19], we prove indistinguishability lemmata for each combination of hardware semantics and contract.

**Contract satisfaction outline:** To prove $\{\!|\cdot|\!\} \vdash [\![\cdot]\!]$, we need to show that $\{\!|p|\!\}(\sigma) = \{\!|p|\!\}(\sigma')$ if $[\![p]\!](\sigma) = [\![p]\!](\sigma')$. To this end, we prove by induction that hardware states $\mathbf{hr}(i)$ and $\mathbf{hr}'(i)$ are indistinguishable for all $i$, which by definition of $\approx$ implies $\{\!|p|\!\}(\sigma) = \{\!|p|\!\}(\sigma')$.

(Induction basis): The initial hardware states $\mathbf{hr}(0)$ and $\mathbf{hr}'(0)$ are indistinguishable by definition as they agree on their microarchitectural components.

(Inductive step): Assume that $\mathbf{hr}(i) \approx \mathbf{hr}'(i)$. Note that (1) $\mathrm{cr}$, $\mathrm{cr}'$ agree on observations since $[\![p]\!](\sigma) = [\![p]\!](\sigma')$, and (2) $corr_{\mathrm{cr},\mathbf{hr}}(i) = corr_{\mathrm{cr}',\mathbf{hr}'}(i)$ holds by construction. Therefore, the correspondence mappings $corr_{\mathrm{cr},\mathbf{hr}}(i)$ and $corr_{\mathrm{cr}',\mathbf{hr}'}(i)$ provide contract states that satisfy conditions (a) and (b) of the indistinguishability lemma for the indistinguishable hardware states $\mathbf{hr}(i)$ and $\mathbf{hr}'(i)$, and we can conclude $\mathbf{hr}(i+1) \approx \mathbf{hr}'(i+1)$.