InSpectre: Breaking and Fixing Microarchitectural Vulnerabilities by Formal Analysis

Roberto Guanciale
robertog@kth.se
KTH Royal Institute of Technology
Stockholm, Sweden

Musard Balliu
musard@kth.se
KTH Royal Institute of Technology
Stockholm, Sweden

Mads Dam
mfd@kth.se
KTH Royal Institute of Technology
Stockholm, Sweden

ABSTRACT

The recent Spectre attacks have demonstrated the fundamental insecurity of current computer microarchitecture. The attacks use features like pipelining, out-of-order and speculation to extract arbitrary information about the memory contents of a process. A comprehensive formal microarchitectural model capable of representing the forms of out-of-order and speculative behavior that can meaningfully be implemented in a high performance pipelined architecture has not yet emerged. Such a model would be very useful, as it would allow the existence and non-existence of vulnerabilities, and soundness of countermeasures to be formally established.

This paper presents such a model targeting single core processors. The model is intentionally very general and provides an infrastructure to define models of real CPUs. It incorporates microarchitectural features that underpin all known Spectre vulnerabilities. We use the model to elucidate the security of existing and new vulnerabilities, as well as to formally analyze the effectiveness of proposed countermeasures. Specifically, we discover three new (potential) vulnerabilities, including a new variant of Spectre v4, a vulnerability on speculative fetching, and a vulnerability on out-of-order execution, and analyze the effectiveness of existing countermeasures including constant time and serializing instructions.

CCS CONCEPTS

• **Security and privacy** → *Formal security models*; **Side-channel analysis and countermeasures**.

KEYWORDS

microarchitecture, vulnerabilities, out-of-order, speculation, formal models, countermeasures, verification

ACM Reference Format:
Roberto Guanciale, Musard Balliu, and Mads Dam. 2020. InSpectre: Breaking and Fixing Microarchitectural Vulnerabilities by Formal Analysis. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS '20), November 9–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3372297.3417246

CCS '20, November 9–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7089-9/20/11.
https://doi.org/10.1145/3372297.3417246

1 INTRODUCTION

The wealth of vulnerabilities that have followed on from Spectre and Meltdown [32, 36] have provided ample evidence of the fundamental insecurity of current computer microarchitecture. The use of instruction level parallelism in the form of out-of-order (OoO) and speculative execution has produced designs with side channels that can be exploited by attackers to learn sensitive information about the memory contents of a process. One witness of the subtlety of the issues is the more than 50 years passed since pipelining, caching, and OoO execution, cf. IBM S/360, was first introduced.

Another witness is the fact that two years after the discovery of Spectre, a comprehensive understanding of the security implications of pipeline related microarchitecture features has yet to emerge. One result is the ongoing arms race between researchers discovering new Spectre-related vulnerabilities [9], and CPU vendors providing patches followed by informal arguments [5]. The security and effectiveness of the currently proposed countermeasures is unknown, and there are continuously new vulnerabilities appearing that exploit specific microarchitecture features.

It is important to note that side channels and functional correctness are to a large extent orthogonal. The latter is usually proved by reducing pipelined behavior to sequential behavior through some form of refinement-based argument. The past decades have seen a significant body of work in this area, cf. [1, 8, 39, 48]. Functional correctness, however, focuses on programs' input-output behavior and fails to adequately capture the differential aspects of speculation and instruction reordering that are at the root of Spectre-like vulnerabilities. For a systematic study of the latter we argue that new tools that are not necessarily tied to any specific pipeline architecture are needed.

Along this line, several recent works [12, 14, 23, 41] have proposed formal microarchitectural models using information flow analysis to identify information leaks arising from speculative execution. These models capture specific speculation features, e.g, branch prediction, and variants of Spectre, and design analyses that detect known attacks [12, 23, 54]. While these approaches illustrate the usefulness of formal models in analyzing microarchitecture leaks, features lying at the heart of modern CPUs such as OoO execution and many forms of speculation remain largely unexplored, implying that new vulnerabilities may still exist.

Contributions. This work presents InSpectre, the first comprehensive model capable of capturing OoO execution and all existing forms of speculation. Our first contribution is a novel semantics supporting microarchitectural features such as OoO execution, non-atomicity of instructions, and various forms of speculation, including branch prediction, jump target prediction, return address

prediction, and dependency prediction. Additionally, the semantics supports features such as address aliasing, dynamic references, store forward, and OoO memory commits, which are necessary to model all known variants of Spectre. The semantics implements the stages of an abstract pipeline supporting OoO (Section 4) and speculative execution (Section 5). In line with existing work [12, 23], our security condition formalizes the intuition that optimizations should not introduce additional information leaks (conditional noninterference, Section 2). We use this condition to show that InSpectre can reproduce all four variants of Spectre.

As a second contribution, we use InSpectre to discover three new potential vulnerabilities. The first vulnerability shows that CPUs supporting only OoO may leak sensitive information. We discovered the second vulnerability while attempting to validate a CPU vendor's claim that microarchitectures like Cortex A53 are immune to Spectre vulnerabilities because they support only speculative fetching [5]. Our model reveals that this may not be the case. The third vulnerability is a variant of Spectre v4 showing that speculation of a dependency, rather than speculation of a non-dependency as in Spectre v4, between a load and a store operation may also leak sensitive information.

Finally, as a third contribution, we leverage InSpectre to analyze the effectiveness of some existing countermeasures. We found that constant-time [7] analysis is unsound for processors supporting only OoO, and propose a provably secure fix that enables constant-time analysis to ensure security for such processors.

Proofs are reported in the extended version of this paper [22].

2 SECURITY MODEL

Our security model has the following ingredients: (i) an *execution* model which is given by the execution semantics of a program; (ii) an *attacker* model specifying the observations of an attacker; (iii) a *security policy* specifying the parts of the program state that contain sensitive/high information, and the parts that contain public/low information; (iv) a *security condition* capturing a program's security with respect to an execution model, an attacker model, and a security policy.

First, we consider a general model of attacker that observes the interaction between the CPU and the memory subsystem. This model has been used (e.g., [3]) to capture information leaks via cache-based side channels transparently without an explicit cache model. It can capture trace-driven attackers that can interleave with the victim's execution and indirectly observe, for instance using Flush+Reload [20], the victim's cache footprint via latency jitters. The attacker can observe the address of a memory load *dl v* (data load from memory address v), the address of a memory store *ds v* (data store to memory address v), as well as the value of the program counter *il v* (instruction load from memory address v) [43].

We assume a transition relation $\rightarrow \subseteq \mathit{States} \times \mathit{Obs} \times \mathit{States}$ to model the execution semantics of a program as a state transformer producing observations $l \in \mathit{Obs}$. The reflexive and transitive closure of $\rightarrow$ induces a set of executions $\pi \in \Pi$. The function $\mathit{trace} : \Pi \mapsto \mathit{Obs}^*$ extracts the sequence of observations of an execution.

The security policy is defined by an *indistinguishability relation* $\sim \subseteq \mathit{States} \times \mathit{States}$. The relation $\sim$ determines the security of information that is initially stored in a state, modeling the set of initial states that an attacker is not allowed to discriminate. These states represent the initial *uncertainty* of an attacker about sensitive information.

The security condition defines the security of a program on the target execution model (e.g., the speculation model) $\rightarrow_t$ conditionally on the security of the same program on the reference, i.e. sequential, model $\rightarrow_r$, by requiring that the target model does not leak more information than the reference model for a policy $\sim$.

Definition 2.1 (Conditional Noninterference). Let $\sim$ be a security policy and $\rightarrow_t$ and $\rightarrow_r$ be transition relations for the target and reference models of a system. The system is conditionally noninterferent if for all $\sigma_1, \sigma_2 \in \mathit{States}$ such that $\sigma_1 \sim \sigma_2$, if for every $\pi_1 = \sigma_1 \rightarrow_r \cdots$ there exists $\pi_2 = \sigma_2 \rightarrow_r \cdots$ such that $\mathit{trace}(\pi_1) = \mathit{trace}(\pi_2)$ then for every $\rho_1 = \sigma_1 \rightarrow_t \cdots$ there exists $\rho_2 = \sigma_2 \rightarrow_t \cdots$ such that $\mathit{trace}(\rho_1) = \mathit{trace}(\rho_2)$.

Conditional noninterference captures only the new information leaks that may be introduced by model $\rightarrow_t$, and ignores any leaks already present in model $\rightarrow_r$. The target model is constructed in two steps. First, we present an OoO model that extends the sequential model, which is deterministic, by allowing evaluation to proceed out-of-order. Then the OoO model is further extended by adding speculation. At each step the traces of the abstract model are included in the extended model, and a memory consistency result demonstrates that the per location sequence of memory stores is the same for both models. This establishes functional correctness. Conditional noninterference then establishes security of each extension. Each such step strictly increases the set of possible traces by adding nondeterminism. Since refinement is often viewed as essentially elimination of nondeterminism, one can think of the extensions as "inverse refinements". Since conditional noninterference considers a possibilistic setting, it does not account for information leaks through the number of initial indistinguishable states. Appendix A elucidates the advantages of conditional noninterference as compared to standard notions of noninterference.

3 FORMAL MICROARCHITECTURAL MODEL

We introduce a Machine Independent Language (MIL) which we use to define the semantics of microarchitectural features such as OoO and speculative execution. We use MIL as a form of abstract microcode language: A target language for translating ISA instructions and reasoning about features that may cause vulnerabilities like Spectre. Microinstructions in MIL represent atomic actions that can be executed by the CPU, emulating the pipeline phases in an abstract manner. This model is intentionally very general and provides an infrastructure to define models of real microarchitectures.

We consider a domain of values $v \in V$, a program counter $pc \in \mathcal{PC}$, a finite set of register/flag identifiers $r_0, \ldots, r_n, f, z \in \mathcal{R} \subseteq V$, and a finite set of memory addresses $a_0, \ldots, a_m \in \mathcal{M} \subseteq V$. The language can be easily extended to support other type of resources, e.g., registers for vector operations. We assume a total order $<$ on a set of names $t_0, t_1, \ldots \in N$, which we use to uniquely identify microinstructions. We write $N_1 < N_2$ if for every pair $(t_1, t_2) \in N_1 \times N_2$ it holds that $t_1 < t_2$.

Microinstructions $\iota \in \mathcal{I}$ are conditional atomic single assignments. A microinstruction $\iota = t \leftarrow c?o$ is uniquely identified by its name $t \in N$ and consists of a boolean guard c, which determines if

the assignment should be executed, and an operation $o \in Op$. The MIL language has three types of operations:

$$e ::= v \mid t \mid e_1 + e_2 \mid e_1 > e_2 \mid \cdots$$
$$o ::= e \mid ld\ \tau\ t_a \mid st\ \tau\ t_a\ t_v$$

An internal operation e is an expression over standard finite arithmetic and can additionally refer to names in N and values in V. A resource load operation $ld\ \tau\ t_a$, where $\tau \in \{\mathcal{PC}, \mathcal{R}, \mathcal{M}\}$, loads the value of resource τ addressed by t_a. We support three types of resources: The program counter $\mathcal{PC}$, registers $\mathcal{R}$, and memory locations $\mathcal{M}$. A resource store operation $st\ \tau\ t_a\ t_v$ uses the value of t_v to update the resource τ addressed by t_a.

The free names $fn(\iota)$ of an instruction $\iota = t \leftarrow c?o$ is the set of names occurring in c or o, the bound names, $bn(\iota)$, is the singleton $\{t\}$, and the names $n(\iota)$ is $fn(\iota) \cup bn(\iota)$.

To model the internal state of a CPU pipeline, we can translate an ISA instruction as multiple microinstructions. For an ISA instruction at address $v \in \mathcal{M}$ and a name $t \in N$, the function $translate(v, t)$ returns the MIL translation of the instruction at address v, ensuring that the names of the microinstructions thus generated are greater than t. Because we assume code to not be self-modifying, an instruction can be statically identified by its address in memory. We assume that the translation function satisfies the properties: (i) for all $\iota_1, \iota_2 \in translate(v, t)$, if $\iota_1 \neq \iota_2$ then $bn(\iota_1) \cap bn(\iota_2) = \emptyset$; for all $\iota \in translate(v, t)$, (ii) $fn(\iota) < bn(\iota)$, and (iii) $\{t\} < n(\iota)$.

These properties ensure that names uniquely identify microinstructions, the name parameters of a single instruction form a Directed Acyclic Graph, the translated microinstructions are assigned names greater than t, and the translation of two different ISA instructions does not have direct inter-instruction dependencies (but may have indirect ones).

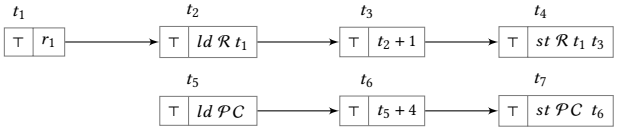
3.1 MIL Program Examples

We introduce some illustrative examples of MIL programs, using their graph representation. For clarity, we omit conditions whenever they are true and visualize only the immediate dependencies between graph elements.

Consider an ISA instruction that increments the value of register r_1, i.e., $r_1 := r_1 + 1$. The instruction can be translated in MIL as follows:

$$\left\{ \begin{array}{l} t_1 \leftarrow r_1,\ t_2 \leftarrow ld\ \mathcal{R}\ t_1,\ t_3 \leftarrow t_2 + 1,\ t_4 \leftarrow st\ \mathcal{R}\ t_1\ t_3, \\ t_5 \leftarrow ld\ \mathcal{PC}\ ,\ t_6 \leftarrow t_5 + 4,\ t_7 \leftarrow st\ \mathcal{PC}\ \ t_6 \end{array} \right\}$$

Intuitively, t_1 refers to the identifier of target register r_1, t_2 loads the current value of r_1, t_3 executes the increment, and t_4 stores the result of t_3 in the register store. The translation of an ISA instruction also updates the program counter to enable the execution of the next instruction. In this case, the program counter is increased by 4, unconditionally. Notice that we omit the program counter's address, since there is only one such resource. We can graphically represent this set of microinstructions using the following graph:

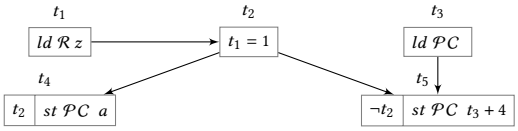


Example 1: r_1 := r_1+1

In the following we adopt syntactic sugar to use expressions, in place of names, for the address and value of load and store operations. This can be eliminated by introducing the proper intermediary internal assignments. This permits to rewrite the previous example as:

The translation of multiple ISA instructions results in disconnected graphs. This reflects the fact that inter-instruction dependencies may not be statically identified due to dynamic references and must be dynamically resolved by the MIL semantics. When translating multiple instructions, we use the following convention for generated names: the name t_{ij} identifies the j-th microinstruction resulting from the translation of the i-th instruction. Our convention induces a total (lexicographical) order over names (i.e., $t_{ij} < t_{i'j'}$ iff $(i < i') \vee (i = i' \wedge j < j')$), which respects the properties of the translation function.

MIL is expressive enough to support conditional instructions like conditional move. Conditional branches can be modeled in MIL via microinstructions that are guarded by complementary conditions. For instance, the `beq a` instruction, which jumps to address a if the z flag is set, can be translated as in Example 2.



Example 2: `beq a`

4 OUT-OF-ORDER SEMANTICS

This section presents an OoO semantics for MIL programs, which is extended in Section 5 to account for speculation. To show the adequacy of the OoO and speculative semantics we prove that they are memory consistent with respect to the in-order (sequential) semantics, namely that writes to the same memory location are seen in the order. Full details of memory consistency and in-order semantics are reported in Appendix B.

4.1 States, Transitions, Observations

We formalize the semantics via a transition relation $\sigma \xrightarrow{l} \sigma'$, which maps a state σ to a state σ', and produces a (possibly empty, represented by a dot $(\cdot)$) observation $l \in Obs$, eliding the dot when convenient. As in Section 2, $Obs = \{\cdot, dl\ v, ds\ v, il\ v\}$ captures the attacker model.

States σ are tuples (I, s, C, F) where: (i) I is a set of MIL microinstructions, (ii) $s \in Stores = N \rightharpoonup V$ is a (partial) storage function from names to values recording microinstructions' execution results, (iii) $C \subseteq N$ is a set of names of store operations that have been committed to the memory subsystem, (iv) $F \subseteq N$ is a set of names of program counter store operations that have been processed, causing the ISA instruction at the stored location to be fetched and decoded.

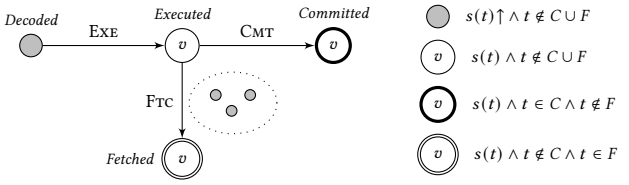


Figure 1: OoO semantics: Microinstruction lifecycle

In the following we write $s[t \mapsto v]$ for substitution of value v for name t in store s. We use $f(x)\downarrow$ to represent that the partial function f is defined on x, and $f(x)\uparrow$ if not $f(x)\downarrow$. We write $\mathrm{dom}(f)$ for the domain of a partial function f. We also use $f|_D$ to represent the restriction of function f to domain D. The semantics of expressions is $[e] : \mathit{Stores} \rightharpoonup V$ and is defined as expected. An expression is undefined if at least one name is undefined in a storage, i.e., $[e](s)\uparrow \Leftrightarrow \mathit{fn}(e) \not\subseteq \mathrm{dom}(s)$. For $\sigma = (I, s, C, F)$ we use $[e]\sigma$ for $[e]s$, $\sigma(t)\uparrow$ for $s(t)\uparrow$, and $\iota \in \sigma$ for $\iota \in I$.

4.2 Microinstruction Lifecycle

Figure 1 represents the microinstruction lifecycle in the OoO semantics. For a given state (I, s, C, F), a microinstruction $\iota = t \leftarrow c?o \in I$ can be in one of four different states. A microinstruction in state Decoded (represented by a gray circle) has not been executed, committed or fetched ($s(t)\uparrow$, $t \notin C$, $t \notin F$), and its guard is either true ($[c]s$) or undefined ($[c](s)\uparrow$). If the guard is false, i.e, $\neg[c]s$, the instruction is considered as Discarded. A microinstruction moves to state Executed (denoted by a simple circle containing $s(t)$) if its guard evaluates to true and all dependencies have been executed. An Executed store microinstruction can either be committed to the memory (Committed: $t \in C$, denoted by a bold circle), or, if it is a PC store, assign the program counter, causing a new ISA instruction to be fetched and decoded (Fetched: $t \in F$, denoted by a double circle). Accordingly, the transition of a PC store to state Fetched leads to the spawn of a collection of newly decoded microinstructions (i.e., the translation of the subsequent ISA instruction) in state Decoded. The labels of the edges in the diagram correspond to the names of the transition rules of Section 4.4.

4.3 Semantics of Single Microinstructions

The semantics is defined in two steps: we first define the semantics of single microinstructions, then introduce the operational semantics of MIL programs. The semantics of a microinstruction $[\iota] : \mathit{States} \rightarrow (V \times \mathit{Obs}) \cup \{\bot\}$ returns either a value and an observation, or $\bot$ if the microinstruction cannot be executed.

(Internal operations) The semantics of internal operations is straightforward:

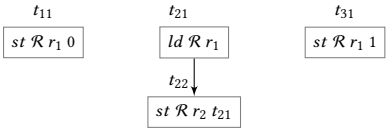
$$[t \leftarrow c?e]\sigma = \begin{cases} ([e]\sigma, \cdot) & \text{if } [e](\sigma)\downarrow \\ \bot & \text{otherwise} \end{cases}$$

An internal operation can be executed as soon as its dependencies are available. In Example 1, the semantics of internal operation t_1 is defined for the empty storage $\emptyset$, since it does not refer to any names. However, the semantics of t_3 is undefined in $\emptyset$, since it depends on the value of t_2 that is not available in $\emptyset$.

(Store operations) The semantics of store operations is defined as follows:

$$[t \leftarrow c?st\ \tau\ t_a\ t_v]\sigma = \begin{cases} ([t_v]\sigma, \cdot) & \text{if } [t_v](\sigma)\downarrow \wedge [t_a](\sigma)\downarrow \\ \bot & \text{otherwise} \end{cases}$$

A resource update can be executed as soon as both the address of the resource and the value are available. Observe that this rule models the *internal* execution of a resource update and not its commit to the memory subsystem. These internal updates are not observable by a programmer, therefore there is no restriction on their execution order. As an example, the ISA program $r_1 := 0;\ r_2 := r_1;\ r_1 := 1$ can be implemented by the following microinstructions:



The semantics of t_{11}, i.e., $st\ \mathcal{R}\ r_1\ 0$, and t_{31}, i.e., $st\ \mathcal{R}\ r_1\ 1$, is defined in $\emptyset$, and yields $(0, \cdot)$ and $(1, \cdot)$, respectively. As we will see, the operational semantics is in charge of ordering resource updates to preserve consistency and dependencies.

(Load operations) While the semantics of internal operations and store operations only depends on the execution of their operands, load operations may depend on past store operations. This requires identifying the previous resource update that determines the correct value to be loaded. We use the following definitions to compute the set of store operations that may affect a load operation.

Definition 4.1. Consider a load operation $t \leftarrow c?ld\ \tau\ t_a$

- $\mathit{str\text{-}may}(\sigma, t) = \{t' \leftarrow c'?st\ \tau\ t'_a\ t'_v \in \sigma \mid t' < t \wedge ([c']\sigma \vee [c'](\sigma)\uparrow) \wedge ([t'_a]\sigma = [t_a]\sigma \vee \sigma(t'_a)\uparrow \vee \sigma(t_a)\uparrow)\}$ is the set of stores that *may* affect the load address of t in state σ.
- $\mathit{str\text{-}act}(\sigma, t) = \{t' \leftarrow c'?st\ \tau\ t'_a\ t'_v \in \mathit{str\text{-}may}(\sigma, t) \mid \neg\exists t'' \leftarrow c''?st\ \tau\ t''_a\ t''_v \in \mathit{str\text{-}may}(\sigma, t).\ t'' > t' \wedge [c'']\sigma \wedge [t''_a]\sigma \in \{[t_a]\sigma, [t'_a]\sigma\}\}$ is the set of *active* stores.

The stores that may affect the address of t are the stores that: (i) have not been discarded, namely they can be executed ($[c]\sigma$) or may be executed ($c(\sigma)\uparrow$), and (ii) the store address in t'_a may result in the same address as the load address in t_a, namely either they both evaluate to the same address ($\sigma(t'_a) = \sigma(t_a)$), or the store address is unknown ($\sigma(t'_a)\uparrow$), or the load address is unknown ($\sigma(t_a)\uparrow$).

The active stores of t are the stores that may affect the load address computed by t_a, and, there are no subsequent stores t'' on the same address as the load address in t_a, or on the same address as the store address in t'_a. This set determines the "minimal" set of store operations that may affect a load operation from address t_a.

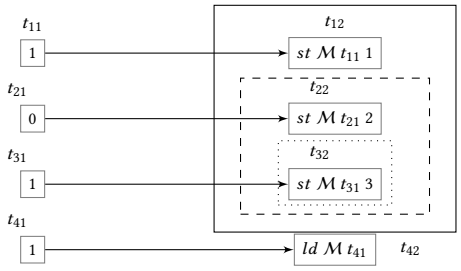
The definitions of $\mathit{str\text{-}act}(\sigma, t)$ and $\mathit{str\text{-}may}(\sigma, t)$ are naturally extended to stores $t \leftarrow c?st\ \tau\ t_a\ t_v$. These definitions allow us to define the semantics of loads:

$$[t \leftarrow c?ld\ \tau\ t_a]\sigma = \begin{cases} ([t_s]\sigma, l) & \text{if } bn(\mathit{str\text{-}act}(\sigma, t)) = \{t_s\} \wedge \\ & \sigma(t_a)\downarrow \wedge \sigma(t_s)\downarrow \\ \bot & \text{otherwise} \end{cases}$$

where $l = \begin{cases} dl\ \sigma(t_a) & \text{if } t_s \in C \wedge \tau = \mathcal{M} \\ \cdot & \text{otherwise} \end{cases}$

A load operation can be executed if the set of active stores consists of a singleton set with bound name t_s, i.e., the store causing t_a to be assigned is uniquely determined, and both the address t_a of the load and the address t_s of the store can be evaluated in state σ.

Note that the semantics allows forwarding the result of a store to another microinstruction before it is committed to memory. In fact, if the active store is yet to be committed to memory, i.e., $t_s \notin C$, it is possible for the store to forward its data to the load, without causing an interaction with the memory subsystem (i.e., $l = \cdot$). Otherwise, the load yields an observation of a data load from address $\sigma(t_a)$.



Example 3: `*(1):=1; *(0):=2; *(1):=3; *(1);`

Example 3 illustrates the semantics of loads. The program writes 1 into address 1, then writes 2 in 0, overwrites address 1 with 3, and finally loads from address 1. We use active stores to dynamically compute the dependencies of load operations. Let σ_0 be a state containing microinstructions as in the example, and having empty storage. For this state, the active store for the load t_{42}, i.e., $str\text{-}act(\sigma_0, t_{42})$, consists of all stores of the example, as depicted by the solid rectangle. Since none of microinstructions that compute the addresses have been executed, the address t_{41} of the load is unknown, hence, we cannot exclude any store from affecting the address that will be used by t_{42}. Therefore, the load cannot be executed in σ_0. This set of active stores will shrink during execution as more information becomes available through the storage.

Let the storage of σ_1 be $\{t_{11} \mapsto 1; t_{31} \mapsto 1\}$, i.e., the result of executing t_{11} and t_{31}. The active stores $str\text{-}act(\sigma_1, t_{42})$ consist of microinstructions depicted by the dashed rectangle. Observe that the store t_{12} is in $str\text{-}may(\sigma_1, t_{42})$, however there exists a subsequent store, namely t_{32}, that overwrites the effects of t_{12} on the same memory address. Therefore, t_{12} is no longer an active store and it can safely be discarded.

Let the storage of σ_2 be $\{t_{11} \mapsto 1; t_{31} \mapsto 1, t_{41} \mapsto 1\}$, i.e., the result of executing t_{11}, t_{31} and t_{41}. The active stores $str\text{-}act(\sigma_2, t_{42})$ now consist of the singleton set $\{t_{32}\}$ as depicted by the dotted rectangle. This is because the address t_{41} of the load can be computed in state σ_2. Although t_{22} is still in $str\text{-}may(\sigma_2, t_{42})$, there is a subsequent store, t_{32}, that will certainly affect the address of the load. Therefore, t_{22} is no longer an active store.

Finally, let the storage of σ_3 be $\{t_{11} \mapsto 1; t_{31} \mapsto 1, t_{41} \mapsto 1, t_{32} \mapsto 3\}$, i.e., the result of executing t_{11}, t_{31}, t_{41}, and t_{32}. Once $str\text{-}act$ has been reduced to a singleton set ($\{t_{32}\}$), and the active-store has been executed ($\sigma_3(t_{32})\downarrow$), the semantics of the load is defined. This yields the same value as the store in t_{32}. If the store t_{32} has been committed to memory, the execution of the load yields the observation $dl\ 1$.

4.4 Operational Semantics

We can now define the microinstructions' transition relation $\sigma \xrightarrow{l} \!\!\!\!\rightarrow\ \sigma'$, implementing the lifecycle of Section 4.2.

(Execute) A microinstruction can be executed if it hasn't already been executed ($s(t)\uparrow$), the guard holds ($[c]s$), and the dependencies have been resolved ($[\iota](s)\downarrow$):

$$(\textsc{Exe})\quad \frac{\iota = t \leftarrow c?o \in I \qquad s(t)\uparrow \qquad [c]s \qquad [\iota]\sigma = (v, l)}{\sigma = (I, s, C, F) \xrightarrow{l} \!\!\!\!\rightarrow\ (I, s[t \mapsto v], C, F)}$$

Observe that if ι is a load from the memory subsystem, the rule can produce the observation of a data load.

(Commit) Once a memory store has been executed ($s(t)\downarrow$), it can be committed to memory, yielding an observation. The rule ensures that stores can only be committed once ($t \notin C$) and that stores on the same address are committed in program order, by checking that all past stores are in C, i.e., $bn(str\text{-}may(\sigma, t)) \subseteq C$.

$$(\textsc{Cmt})\quad \frac{t \leftarrow c?st\ \mathcal{M}\ t_a\ t_v \in I \qquad s(t)\downarrow \qquad t \notin C \qquad bn(str\text{-}may(\sigma, t)) \subseteq C}{\sigma = (I, s, C, F) \xrightarrow{ds\ s(t_a)} \!\!\!\!\rightarrow\ (I, s, C \cup \{t\}, F)}$$

In summary, stores can be executed internally in any order, however, they are committed in order. In Example 3, if σ has storage $s = \{t_{11} \mapsto 1; t_{12} \mapsto 1; t_{31} \mapsto 1; t_{32} \mapsto 3\}$ and commits $C = \emptyset$, then only t_{12} can be committed, since t_{22} has not been executed and $bn(str\text{-}may(\sigma, t_{32})) \not\subseteq C$. Notice that t_{22} is in the may stores since its address has not been resolved. Therefore, t_{32} can be committed only after t_{12} has been committed and t_{21} has been executed. However, the commit of t_{32} does not have to wait for the commit or execution of t_{22}. In fact, if σ' has storage $s' = s \cup \{t_{21} \mapsto 0\}$ then $bn(str\text{-}may(\sigma', t_{32})) = \{t_{12}\}$. That is, the order of store commits is only enforced per location, as expected.

(Fetch-Decode) A program counter store enables the fetching and decoding (i.e., translating) of a new ISA instruction. The rule for fetching is similar to the rule for commit, since instructions are fetched in order. The set F keeps track of program counter updates whose resulting instruction has been fetched and ensures that instructions are not fetched or decoded twice. Fetching the result of a program counter update yields the observation of an instruction load from address a.

$$(\textsc{Ftc})\quad \frac{t \leftarrow c?st\ \mathcal{PC}\ t_v \in I \qquad s(t) = a \qquad t \notin F \qquad bn(str\text{-}may(\sigma, t)) \subseteq F}{\sigma = (I, s, C, F) \xrightarrow{il\ a} \!\!\!\!\rightarrow\ (I \cup I', s, C, F \cup \{t\}) \quad \text{where } I' = translate(a, max(I))}$$

Write $max(I)$ for the largest name t in I and $translate(a, max(I))$ for the translation of the instruction at address a, ensuring that the names of the microinstructions thus generated are greater than $max(I)$.

(Remarks on OoO semantics) The three rules of the semantics reflect the atomicity of MIL microinstructions: A transition can affect a single microinstruction by either assigning a value to the storage, extending the set of commits, or extending the set of fetches. In the following, we use $step\text{-}param(\sigma, \sigma') = (\alpha, t)$ to identify the rule $\alpha \in \{\textsc{Exe}, \textsc{Cmt}(a, v), \textsc{Ftc}(I)\}$ that enables $\sigma \twoheadrightarrow \sigma'$ and the name t of the affected microinstruction. In case of commits we also extract

the modified address a and the saved value v, in case of fetches we extract the newly decoded microinstructions I. The semantics preserves several invariants: Let $(I, s, C, F) = \sigma$ if $\alpha = \textsc{Exe}$ then $\sigma(t)\uparrow$; if $\alpha = \textsc{Cmt}(a, v)$ then $t \notin C$ and free names (i.e., address and value) of the corresponding microinstruction are defined in s; if $\alpha = \textsc{Ftc}(I')$ then $t \notin F$; all state components are monotonic.

(Initial state) In order to bootstrap the computation, we assume that the set of microinstructions of the initial state contains one store for each memory address and register, the value of these stores is the initial value of the corresponding resource, and that these stores are in the storage and commits of the initial state.

5 SPECULATIVE SEMANTICS

We now extend the OoO semantics to support speculation. We add two new components to the states: a set of names $P \subseteq n(I)$ whose values have been predicted as result of speculation, and a partial function $\delta : N \rightharpoonup S$ recording, for each name t, the storage dependencies at time of execution of the microinstruction identified by t. Therefore, a state in the speculative semantics is a tuple $h = (I, s, C, F, \delta, P)$ where $\sigma = (I, s, C, F)$ is the corresponding state in the OoO semantics. Abusing notation we write (σ, δ, P) to denote a state in the speculative semantics, and use $h, h_1, \ldots$ to range over these states. Informally, $\delta(t)$ is a *snapshot* of the storage that affects the value of t due to speculative predictions. As we will see, these snapshots are needed in order to match speculative states with non-speculative states, and to restore the state of the execution in case of misspeculation.

5.1 Managing Microinstruction Dependencies

The execution of a microinstruction may depend on local (intra-) instruction dependencies, the names appearing freely in a microinstruction, as well as cross (inter-) instruction dependencies, caused by memory or register loads.

Definition 5.1. Let $t \leftarrow c?o \in \sigma$. The dependencies of t in σ are

$$deps(t, \sigma) = fn(t \leftarrow c?o) \cup depsX(t, \sigma)$$

where the cross-instruction dependencies are defined as

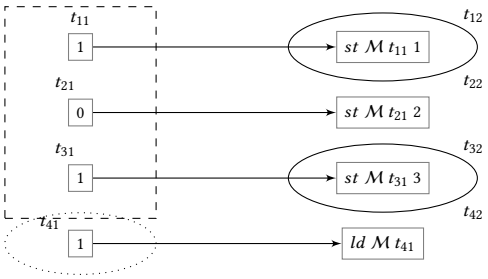
$$depsX(t, \sigma) = \begin{cases} \emptyset, & \text{if } t \text{ is not a load} \\ asn(\sigma, t) \cup srcs(\sigma, t), & \text{otherwise.} \end{cases}$$

Cross-dependencies are nonempty only for loads and consist of the names of active stores affecting t in state σ, $asn(\sigma, t) = bn(str\text{-}act(\sigma, t))$, plus, the names of stores potentially intervening between the earliest active store and t (we call $srcs(\sigma, t)$ the *potential sources of* t), which are defined as

$$srcs(\sigma, t) = \bigcup \{fn(c'), \{t'_a\} \quad | \quad \begin{array}{l} min(asn(\sigma, t)) \le t' < t, \\ t' \leftarrow c'?st\ \tau\ t'_a\ t'_v \in \sigma\} \end{array}$$

Intuitively, a load depends on the execution of active stores that may affect the address of that load. Moreover, the fact that a name t^* is in the set of active stores asn depends on the addresses and guards of all stores between t^* and t. This is because their values will determine the actual store that affects the address of the load t. Thanks to our ordering relation $<$ between names, we can use the minimum name $min(asn)$ in asn to compute all stores between any name in asn and t, thus extracting the free names of their guards and addresses.

The following figure illustrates dependencies of the load from Example 3:



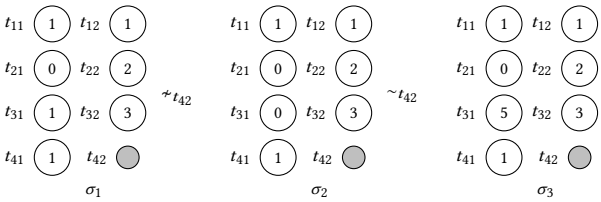
If $s = \{t_{11} \mapsto 1; t_{21} \mapsto 0; t_{41} \mapsto 1\}$ then the set of active stores names asn for t_{42} is $bn(str\text{-}act(\sigma, t_{42})) = \{t_{12}, t_{32}\}$, as depicted by the solid ellipses. In particular, $min(asn) = t_{12}$. We consider all stores between t_{12} and the load t_{42} (i.e., t_{12}, t_{22}, and t_{32}), and add to the set of cross-dependencies the names in their guards and addresses, namely t_{11}, t_{21} and t_{31}, as depicted by the dashed rectangle. Observe that t_{21} is in the set of cross-dependencies, although t_{22} is not an active store. This is because membership of t_{12} in the active stores' set depends on the address t_{21} being set to 0, i.e., $s(t_{21}) = 0$. Therefore, the set of cross-dependencies $depsX(t_{42}, \sigma) = \{t_{12}, t_{32}, t_{11}, t_{21}, t_{31}\}$. Finally, the local dependencies of the load t_{42} consist of its parameter t_{41} (the dotted ellipsis), such that $deps(t_{42}, \sigma) = \{t_{12}, t_{32}, t_{11}, t_{21}, t_{31}, t_{42}\}$.

We verify that the dependencies $deps$ are computed correctly.

Definition 5.2 (t-equivalence). Let σ_1 and σ_2 be states with storage s_1 and s_2, and ι_1 and ι_2 be the microinstructions identified by t. Then σ_1 and σ_2 are t-equivalent, $\sigma_1 \sim_t \sigma_2$, if $\iota_1 = \iota_2$, $s_1|_{fn(\iota_1)} = s_2|_{fn(\iota_2)}$, and if t's microinstruction is a load with dependencies $T_i = deps(t, \sigma_i)$ and active stores $SA_i = str\text{-}act(\sigma_i|_{T_i}, t)$ for $i \in \{1, 2\}$ then $SA_1 = SA_2$ and $s_1|_{SA_1} = s_2|_{SA_2}$.

Intuitively, t-equivalence states that, if the microinstruction named with t depends (in the sense of $deps$) in both states on the same active stores and these stores assign the same value to t, then the microinstruction has the same dependencies, it is enabled, and it produces the same result in both states.

We use three possible states of the example above to illustrate t-equivalence: σ_1 is a state reachable in the OoO semantics, σ_2 and σ_3 may result from misspeculating the value of t_{31} to be 0 and 5 respectively.



The states σ_1 and σ_2 are not t_{42}-equivalent. In particular, $T_1 = deps(t_{42}, \sigma_1) = \{t_{31}, t_{32}\}$ (notice that t_{12} and t_{22} are not in the dependencies because by we know that $t_{31} \mapsto 1$ and $t_{41} \mapsto 1$) , $T_2 = deps(t_{42}, \sigma_2) = \{t_{11}, t_{21}, t_{31}, t_{12}\}$. Notice that $\sigma_1|_{T_1}$ and $\sigma_2|_{T_2}$ contain all the information needed to evaluate the semantics of t_{42} in σ_1 and σ_2 respectively. In this case $SA_1 = str\text{-}act(\sigma_1|_{T_1}, t_{42}) = \{t_{32}\}$, and $SA_2 = str\text{-}act(\sigma_2|_{T_2}, t_{42}) = \{t_{12}\}$ hence $SA_1 \neq SA_2$: the two

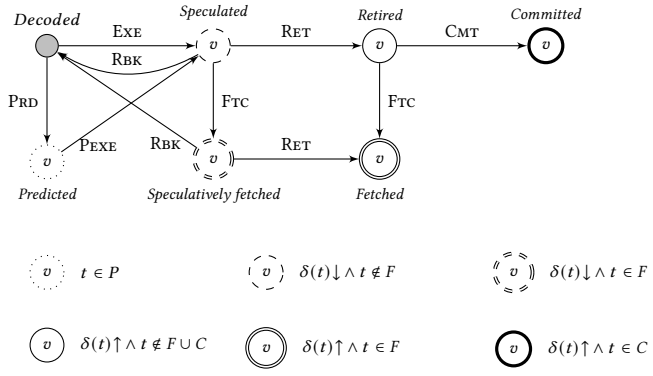


Figure 2: Speculative semantics: Microinstruction lifecycle

states lead the load t_{42} to take the result produced by two different memory stores.

The states σ_2 and σ_3 are t_{42}-equivalent. In fact, $T_3 = deps(t_{42}, \sigma_3) = \{t_{11}, t_{21}, t_{31}, t_{12}\}$ and $SA_3 = str\text{-}act(\sigma_3|_{T_3}, t_{42}) = \{t_{12}\}$. Therefore, $SA_2 = SA_3$ and $s_2|_{SA_2} = s_3|_{SA_3}$: The two states lead the load t_{42} to take the result produced by the same memory stores.

Lemma 5.3. *If $\sigma_1 \sim_t \sigma_2$ and t's microinstruction in σ_1 is $\iota = t \leftarrow c?o$, then $deps(t, \sigma_1) = deps(t, \sigma_2)$, $[c]\sigma_1 = [c]\sigma_2$, and if $[\iota]\sigma_1 = (v_1, l_1)$ and $[\iota]\sigma_2 = (v_2, l_2)$ then $v_1 = v_2$.*

5.2 Microinstruction Lifecycle

Figure 2 depicts the microinstruction lifecycle under speculative execution. Compared to the OoO lifecycle of Section 4.2, states Decoded, Predicted, Speculated, and Speculatively Fetched correspond to state Decoded, state Retired corresponds to Executed, otherwise states Fetched and Committed are the same. As depicted in the legend, transitions between states set different properties of a microinstruction's lifecycle, which we will model in the semantics.

State Predicted (dotted circle) models microinstructions that have not yet been executed, but whose result values have been predicted. A Decoded microinstruction can transition to state Predicted by predicting its result value, thus recording that the value was predicted and causing the state of the microinstruction to be defined. A microinstruction that is ready to be executed (in Decoded), possibly relying on predicted values, can be executed and transition to state Speculated (dashed circle), recording its dependencies in the snapshot. Notice that state Speculated models both speculative and non-speculative execution of a microinstruction.

From state Speculated, a microinstruction can: (*a*) roll back to Decoded (if the predicted values were wrong); (*b*) speculatively fetch the next ISA instruction to be executed, thus moving to state Speculatively Fetched, doubled dashed circle) and generating newly decoded microinstructions; or (*c*) retire in state Retired (single circle) if it no longer depends on speculated values.

Microinstructions in state Speculatively Fetched can either be rolled back due to misspeculation, otherwise move to state Fetched (double circle). Finally, in state Retired, as in the OoO case, a PC store microinstruction can be (non-speculatively) fetched and generate newly decoded microinstructions, or, if it is a memory store, it can be committed to the memory subsystem (bold circle).

5.3 Microinstruction Semantics

We now present a speculative semantics, denoted by the transition relation $(\sigma, \delta, P) \twoheadrightarrow\!\!\!\rightarrow (\sigma', \delta', P')$, that reflects the microinstructions' lifecycle in Figure 2. We illustrate the rules of our semantics using the graph in Example 4 and the interpretation of states (circles) in Figure 2. Additionally, for two microinstruction identifiers t and t' in speculative state $h = (I, s, C, F, \delta, P)$, we draw an edge from t to t' labeled with v whenever $\delta(t)(t') = v$.

(Predict) The semantics allows to predict the value of an internal operation choosing a value $v \in V$. The rule updates the storage and records the predicted name, while ensuring that the microinstruction has not been executed already.

$$(\textsc{Prd}) \quad \frac{t \leftarrow c?e \in I \qquad s(t)\uparrow \qquad \delta' = \delta \cup \{t \mapsto \emptyset\}}{(I, s, C, F, \delta, P) \twoheadrightarrow\!\!\!\rightarrow (I, s[t \mapsto v], C, F, \delta', P \cup \{t\})}$$

We remark that the semantics can predict a value only for an internal operation ($t \leftarrow c?e$) that has not been already executed ($s(t)\uparrow$). As we will see, this choice does not hinder expressiveness while it avoids the complexity in modeling speculative execution of program counter updates and loads. Concretely, the rule assigns an arbitrary value to the name of the predicted microinstruction ($s[t \mapsto v]$) and records that the result is speculated ($\delta \cup \{t \mapsto \emptyset\}$). Observe that the snapshot $\delta'(t)$ is $\emptyset$ because the prediction does not depend on the results of other microinstructions.

Consider state h_0 in Example 4 containing all microinstructions of our running program, which have just been decoded (gray circles). The CPU can predict that the value of arithmetic operation t_2 is 0. Rule Prd updates the storage with $t_2 \mapsto 0$ (dotted circle), the snapshot for t_2 with an empty mapping, and adds t_2 to the prediction set.

(Execute) The rules for execution, commit, and fetch reuse the OoO semantics. First for the case when the instruction has not been predicted already:

$$(\textsc{Exe}) \quad \frac{\sigma \xrightarrow{l}\!\!\!\!\rightarrow \sigma' \qquad step\text{-}param(\sigma, \sigma') = (\textsc{Exe}, t)}{(\sigma, \delta, P) \xrightarrow{l}\!\!\!\!\!\rightarrow (\sigma', \delta \cup \{t \mapsto s|_{deps(t,\sigma)}\}, P)}$$

The rule executes a microinstruction t using the OoO semantics and updates the snapshot δ, recording that the execution of t was determined by the value of its dependencies in $deps(t, \sigma)$ in storage s of state σ. Notice that the premise $step\text{-}param(\sigma, \sigma') = (\textsc{Exe}, t)$ ensures that microinstruction t has not been predicted. In fact, $step\text{-}param(\sigma, \sigma') = (\textsc{Exe}, t)$ only if $\sigma(t)\uparrow$, while rule Prd would update the storage with a value for name t, hence $t \notin P$.

Consider now the state h_2 resulting from the execution of t_1 and t_3 in Example 4. In h_2 the CPU can execute the PC update t_6, updating the storage with $t_6 \mapsto 36$. The rule additionally updates the snapshot for t_6 with the current values of its dependencies, i.e., $\{t_2 \mapsto 0, t_3 \mapsto 32\}$. Since the executed microinstruction t_6 is a store, its dependencies are the free names occurring in the microinstruction. These snapshots are used by rules Cmt and Rbk to identify mispredictions. Similarly, the rule enables the execution of the memory store t_4 in h_3, which updates the storage with $t_4 \mapsto 1$ and the snapshot for t_4 with the values of its dependencies $\{t_1 \mapsto 1\}$.

The following rule enables the execution of microinstructions whose result has been previously predicted:

$$\text{(Pexe)}\quad \frac{\sigma = (I, s, C, F) \quad t \in P \qquad (I, s \setminus \{t\}, C, F) \xrightarrow{l}\!\!\!\!\rightarrow \sigma' \quad \textit{step-param}(\sigma, \sigma') = (\textsc{Exe}, t)}{(\sigma, \delta, P) \xrightarrow{l}\!\!\!\!\rightarrow\!\!\!\!\rightarrow (\sigma', \delta \cup \{t \mapsto s|_{deps(t,\sigma)}\}, P \setminus \{t\})}$$

The rule removes the value predicted for t from the storage $(s \setminus \{t\})$ to enable the actual execution of t in the OoO semantics. It also removes t from the set of predicted names P and updates the snapshot with the new dependencies of t.

In our example, rule Pexe computes the actual value of t_2 in state h_4, which was previously mispredicted as 0. The rule corrects the misprediction updating the storage with $t_2 \mapsto 1$ and the snapshot for t_2 with the values of its dependencies, i.e., $t_1 \mapsto 1$. Notice that in case of a misprediction, the rule does not immediately roll back all other speculated microinstructions that are affected by the mispredicted values, e.g., t_6.

(Commit) To commit a microinstruction it is sufficient to ensure that there are no dependencies left ($\delta(t)\uparrow$), i.e., the microinstruction has been retired. Since memory commits have observable side effects outside the processor pipeline, only retired memory stores can be sent to the memory subsystem.

$$\text{(Cmt)}\quad \frac{\sigma \xrightarrow{l}\!\!\!\!\rightarrow \sigma' \quad \textit{step-param}(\sigma, \sigma') = (\textsc{Cmt}(a, v), t) \quad \delta(t)\uparrow}{(\sigma, \delta, P) \xrightarrow{l}\!\!\!\!\rightarrow\!\!\!\!\rightarrow (\sigma', \delta, P)}$$

Consider the state h_4 and the memory store t_4 in our example. Since t_4 has not been retired (i.e., $\delta(t_4) = \{t_1 \mapsto 1\}$) it cannot be committed as $\delta(t_4)\downarrow$. By contrast, the commit of t_4 is allowed in state h_8 where $\delta(t_4)\uparrow$.

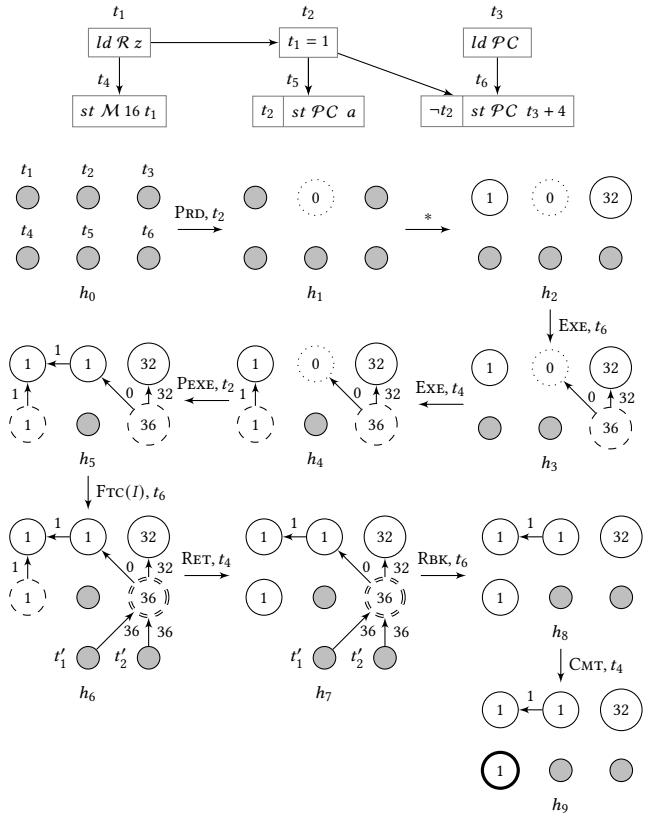
(Fetch) Finally, for the case of (speculative or non-speculative) fetching, the snapshot must be updated to record the dependency of the newly added microinstructions:

$$\text{(Ftc)}\quad \frac{\sigma \xrightarrow{l}\!\!\!\!\rightarrow \sigma' \qquad \textit{step-param}(\sigma, \sigma') = (F(I), t)}{(\sigma, \delta, P) \xrightarrow{l}\!\!\!\!\rightarrow\!\!\!\!\rightarrow (\sigma', \delta \cup \{t' \mapsto s|_{\{t\}} \mid t' \in I\}, P)}$$

Following the OoO semantics, if $\textit{step-param}(\sigma, \sigma') = (F(I), t)$ then t is a PC update and $s(t)$ is the new value of the PC. For every newly added microinstruction in $t' \in I$, we extend the snapshot δ recording that t' was added as result of updating the PC microinstruction t with the value $s(t)$ (formally, we project the storage s on t, i.e., $s|_{\{t\}}$). The new snapshot may be used later to roll back the newly added microinstructions in I if the value of the PC is misspeculated.

For example, in state h_5 the CPU can speculatively fetch the PC update t_6, which sets the program counter to 36. Suppose that the newly added microinstructions in I (i.e., the microinstructions resulting from the translation of the ISA instruction at address 36) are t'_1 and t'_2. Following the OoO semantics, I is added to existing microinstructions in σ' . The rule additionally updates the snapshot for t'_1 and t'_2 recording the PC store that generated the new microinstructions, i.e., $t_6 \mapsto 36$.

(Retire) The following transition rule allows to retire a microinstruction in case of correct speculation:



Example 4: Execution trace of speculative semantics.

$$\text{(Ret)}\quad \frac{s(t)\downarrow \qquad \mathrm{dom}(\delta(t)) \cap \mathrm{dom}(\delta) = \emptyset \qquad (I, s, C, F) \sim_t (I, \delta(t), C, F) \qquad t \notin P}{(I, s, C, F, \delta, P) \longrightarrow\!\!\!\!\rightarrow (I, s, C, F, \delta \setminus \{t\}, P)}$$

The map $\delta(t)$ contains the snapshot of t's dependencies at time of t's execution. A microinstruction can be retired only if all its dependencies have been retired ($\mathrm{dom}(\delta(t)) \cap \mathrm{dom}(\delta) = \emptyset$), the microinstruction has been executed (i.e. its value has not been just predicted $s(t)\downarrow \wedge\, t \notin P$), and the snapshot of t's dependencies is $\sim_t$ equivalent with the current state, hence the semantics of t has been correctly speculated (see Lemma 5.3). Retiring a microinstruction results in removing the state of its dependencies from δ, as captured by $\delta \setminus \{t\}$.

For instance, in state h_6 the PC store t_6 cannot be retired for two reasons: one of its dependencies has not been retired (i.e., $\delta(t_6) = \{t_2 \mapsto 0, t_3 \mapsto 32\}$ and $\delta(t_2) = \{t_1 \mapsto 1\}$, hence $\mathrm{dom}(\delta(t_6)) \cap \mathrm{dom}(\delta) = \{t_2\}$), and the snapshot for t_6 differs with respect to the storage (i.e., $\delta(t_6)(t_2) \neq s(t_2)$). Instead, the microinstruction t_4 can be retired because its dependencies (i.e., t_1) have been retired (i.e., $\delta(t_1)\uparrow$) and the snapshot for t_4 (i.e., $t_1 \mapsto 1$) exactly matches the values in the storage. Notice that retiring t_4 would simply remove the mapping for t_4 from δ (not shown).

Notice that in case of a load, $(I, s, C, F) \sim_t (I, \delta(t), C, F)$ may hold even if some dependencies of t differ in s and $\delta(t)$. In fact, a load may have been executed as a result of misspeculating the address

of a previous store. In this case, $\sim_t$ implies that the misspeculation has not affected the calculation of *str-act* of the load (i.e., it does not cause a store bypass), hence there is no reason to re-execute the load. This mechanism is demonstrated in examples later in this section.

(Rollback) A microinstruction t can be rolled back when it is found to transitively reference a value that was wrongly speculated. This is determined by comparing t's dependencies at execution time ($\delta(t)$) with the current storage assignment (s). In case of a discrepancy, if t is not a program counter store, the assignment to t can simply be undone, leaving speculated microinstructions t' that reference t to be rolled back later, if necessary.

$$(\textsc{Rbk}) \quad \frac{t \notin P \qquad t \notin F \qquad (I, s, C, F) \nsim_t (I, \delta(t), C, F)}{(I, s, C, F, \delta, P) \longrightarrow\!\!\!\!\rightarrow (I, s \setminus \{t\}, C, F, \delta \setminus \{t\}, P)}$$

However, if t *is* a program counter store, the speculative evaluation using rule Ftc will have caused a new microinstruction to be speculatively fetched. This fetch needs to be undone. To that end let $t' \prec t$ (t' refers to t) if $t \in \mathrm{dom}(\delta(t'))$, let $\prec^+$ be the transitive closure of $\prec$. As expected $\prec^+$ is antisymmetric and its the reflexive closure is a partial order. Define then the set Δ^+ as $\{t' \mid t' \prec^+ t\}$: i.e., Δ^+ is the set of names that reference t, not including t itself. Finally, let $\Delta^* = \Delta^+ \cup \{t\}$.

$$(\textsc{Rbk}) \quad \frac{t \notin P \qquad t \in F \qquad (I, s, C, F) \nsim_t (I, \delta(t), C, F)}{(I, s, C, F, \delta, P) \longrightarrow\!\!\!\!\rightarrow (I \setminus \Delta^+, s \setminus \Delta^*, C, F \setminus \Delta^*, \delta \setminus \Delta^*, P \setminus \Delta^*)}$$

For example, in state h_7 the program counter update t_6 can be rolled back because $s(t_2) = 1 \neq 0 = \delta(t_6)(t_2)$. The transition moves the microinstruction t_6 back to the decoded state (i.e., the storage and snapshot h_8 are undefined for t_6) and removes every microinstruction that have been decoded by t_6 (i.e., t_1' and t_2').

Notice that rollbacks can be performed out of order and that loads can be retired even in case of mispredictions if their dependencies have been enforced. This permits to model advanced recovery methods used by modern processors, including concurrent and partial recovery in case of multiple mispredictions.

Speculation of load/store dependencies Since the predicted values of internal operations (cf. rule Prd) can affect conditions and targets of program counter stores, the speculative semantics supports speculation of control flow, as well as speculative execution of cross-dependencies resulting from prediction of load/store's addresses. We illustrate these features with an example (Figure 3), which depicts one possible execution of the program in Example 3.

Consider the state h_0 after the CPU has executed and retired microinstructions $t_{11}, t_{12}, t_{21}, t_{22}$, and t_{41}, thus resolving the first two stores and the load's address. In state h_0 the CPU can predict the address (i.e., the value of t_{31}) of the third store as 0 and modify the state as in h_1 (rule Prd).

This prediction enables speculative execution of the load t_{42} in state h_1: the active store's bounded names $bn(str\text{-}act(\sigma_1, t_{42}))$ consist of the singleton set $\{t_{12}\}$, since $s_1(t_{21}) = s_1(t_{31}) = 0$, while $s_1(t_{41}) = 1$. Hence, we can apply rule Exe to execute t_{42}, thus updating the storage with $t_{42} \mapsto 1$, and recording the snapshot $\{t_{11} \mapsto 1, t_{21} \mapsto 0, t_{31} \mapsto 0, t_{41} \mapsto 1, t_{12} \mapsto 1\}$ for t_{42}. Concretely, t_{42}'s dependencies in state h_1 consists of the local dependencies (i.e., the load's address t_{41}), and the cross dependencies containing

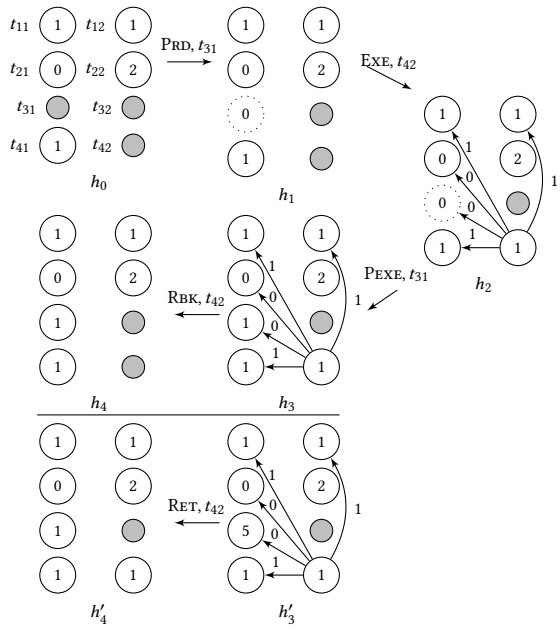


Figure 3: Speculation of load/store dependencies

t_{12} (i.e., active store it loads the value from), as well as the potential sources of t_{42}, that is, the addresses of all stores between the active store t_{12} and the load t_{42}, namely t_{11}, t_{21} and t_{31}.

At this point, load t_{42} cannot be retired by rule Ret in state h_2 since its dependencies, e.g., t_{31}, are yet to be retired. However, we can execute t_{31} by applying rule Pexe. The execution updates the state by removing t_{31} from the prediction set and storing its correct value, as well as extending the snapshot with $t_{31} \mapsto \emptyset$, as depicted in state h_3.

The execution of t_{31} enables the premises of rule Rbk to capture that the dependency misprediction led to misspeculation of the address of the load t_{42}. Specifically, the set *asn* at the time of t_{42}'s execution $bn(str\text{-}act((I_3, \delta_3(t_{42}), C_3, F_3), t_{41})) = \{t_{12}\}$ differs from the active store set $bn(str\text{-}act(\sigma_3, t_{41}))) = \{t_{32}\}$ in the current state. Therefore, we roll back the execution removing the mappings for t_{42} from the storage and the snapshot as in h_4.

Finally, we remark that the speculative execution of loads is rolled back *only if* a misprediction causes a violation of load/store dependencies. For instance, if the value of t_{31} was 5 instead of 1, as depicted in h'_3, the misprediction of t_{31}'s value as 0 in h_1 does not enable a rollback of the load. This is because the actual value of t_{31} does not change the set of active stores. In fact, the set of active stores at the time of t_{42}'s execution $bn(str\text{-}act((I'_3, \delta'_3(t_{42}), C'_3, F'_3), t_{41})) = \{t_{12}\}$ is the same as the active store's set $bn(str\text{-}act(\sigma'_3, t_{41}))) = \{t_{12}\}$ in the current state.

6 ATTACKS AND COUNTERMEASURES

InSpectre can be used to model and analyze (combinations of) microarchitectural features underpinning Spectre attacks [9, 32, 38], and, importantly, to discover new vulnerabilities and to reason about the security of proposed countermeasures. Observe that these

results hold for our generic microarchitectural model, while specific CPUs would require instantiating InSpectre to model their microarchitectural features. We remark that real-world feasibility of our new vulnerabilities falls outside the scope of this work.

Specifically, we use the following recipe: We model a specific prediction strategy in InSpectre and try to prove conditional noninterference for arbitrary programs. Failure to complete the security proof results in new classes of counterexamples as we report below.

Concretely, prediction strategies and countermeasures are modeled by constraining the nondeterminism in the microinstruction scheduler and in the prediction semantics (see rule PRD). The prediction function $pred_p : \Sigma \to N \to 2^V$ captures a prediction strategy p by computing the set of predicted values for a name $t \in N$ and a state $\sigma \in \Sigma$. We assume the transition relation satisfies the following property: If $(\sigma, \delta, P) \xrightarrow{l} (\sigma', \delta', P \cup \{t\})$ then $t \in \mathrm{dom}(pred_p(\sigma))$ and $\sigma'(t) \in pred_p(\sigma)(t)$. This property ensures that the transition relation chooses predicted values from function $pred_p$.

Following the security model in Section 2, we check conditional noninterference by: (*a*) using the in-order transition relation $\to$ as reference model and speculative (OoO) transition relation $\twoheadrightarrow$ ($\hookrightarrow$) as target model; (*b*) providing the security policy $\sim$ for memory and registers. To invalidate conditional noninterference it is sufficient to find two $\sim$-indistinguishable states that yield the same observations in the reference model and different observations in the target model. We use the classification by Canella et al. [9] to refer to existing attacks. We refer to Appendix C for the models and countermeasures for Spectre-PHT, Spectre-BTB, Spectre-RSB, and Retpoline.

6.1 Spectre-STL

Spectre-STL [27] (Store-To-Load) exploits the CPUs mechanism to predict load-to-store data dependencies. A load cannot be executed before executing all the past (in program order) stores that affect the same memory address. However, if the address of a past store has not been resolved, the CPU may execute the load in speculation without waiting for the store, predicting that the target address of the store is different from the load's address. Mispredictions cause store bypasses leading to information leaks and access to stale data. This behavior can be modeled as $pred_{STL}(\sigma, \delta, P) =$

$$\left\{ t_a \mapsto a \;\middle|\; \begin{array}{l} t' \leftarrow c'?ld\ \mathcal{M}\ t'_a \in \sigma \wedge \sigma(t'_a) \neq a \\ t \leftarrow c?st\ \mathcal{M}\ t_a\ t_v \in \textit{str-act}(\sigma, t') \wedge \\ \sigma(t_a)\uparrow \end{array} \right\}$$

A prediction occurs whenever a memory store (t) is waiting an unresolved address ($\sigma(t_a)\uparrow$), while the address ($s(t'_a)$) of a subsequent load (t') has been resolved, and the load may depend on the store ($t \in bn(\textit{str-act}(\sigma, t'))$). Prediction guesses that the store's address (t_a) differs with the load's address.

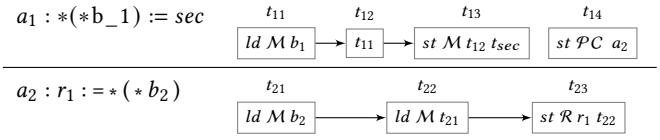
6.1.1 Hardware countermeasures to Store Bypass. The specification of proposed hardware countermeasures oftentimes comes with no precise semantics and is ambiguous. ARM introduced the Speculative Store Bypass Safe (SSBS) configuration to prevent store bypass vulnerabilities. The specification of SSBS [4] is: *Hardware is not permitted to load ... speculatively, in a manner that could ... give rise to a ... side channel, using an address derived from a register value that has been loaded from memory ... (L) that speculatively reads an entry from earlier in the coherence order from that location being loaded from than the entry generated by the latest store (S) to that location using the same virtual address as L.*

InSpectre provides a ground to formalize the behavior of these hardware mechanisms. We formalize SSBS as follows. Let $\sigma = (I, s, C, F, \delta, P)$ and $t \leftarrow c?ld\ \tau\ t_a \in \sigma$. If $\sigma \xrightarrow{l} \sigma'$, $\sigma(t)\uparrow$, and $\sigma'(t)\downarrow$, then for every $t' \in \textit{srcs}(t, \sigma)$, if $\sigma(t') \neq \sigma(t_a)$ then $t' \notin P$.

The reason why SSBS prevents Spectre-STL is simple. The rule forbids the execution of a load t if any address used to identify the last store affecting t_a has been predicted to differ from t_a.

6.1.2 New Vulnerability: Spectre-STL-D. Our model reveals that if a microarchitecture mispredicts *the existence of a Store-To-Load Dependency* (hence Spectre-STL-D), e.g., in order to forward temporary store results, a similar vulnerability may be possible. To model this behavior it is enough to substitute $\sigma(t'_a) \neq a$ with $\sigma(t'_a) = a$ in $pred_{STL}$. We consider this a new form of Spectre because the implementation of this microarchitectural feature can be substantially different from the one required for Spectre-STL (e.g., Feiste et al. [17] patented a mechanism to implement this feature) and because the vulnerable programs are different.

This feature may cause Spectre-STL-D if a misspeculated dependency is used to perform subsequent memory accesses. Consider the following program:



If the CPU executes and fetches t_{14}, predicts that $t_{12} = b_2$ (i.e., it mispredicts the alias $*b_1 == *b_2$), executes t_{13}, forwards the result of t_{13} to t_{21}, and executes t_{22} before the load t_{11} is retired, then the address accessed by t_{22} depends on t_{sec}. This can produce the secret-dependent sequence of observations $il\ a_1 :: il\ a_2 :: dl\ sec$, while the sequential semantics always produces the secret-independent sequence of observations $il\ a_1 :: dl\ b_1 :: ds\ *b_1 :: il\ a_2 :: dl\ b_2 :: dl\ *b_2$. Notice that SSBS may not be effective against Spectre-STL-D.

6.2 New Vulnerability: Spectre-OoO

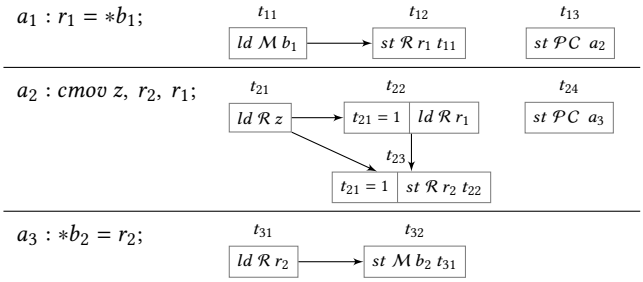
A popular countermeasure to prevent sensitive data from affecting the execution time and caches is "constant time programming", also known as "data oblivious algorithms". This mechanism ensures that branch conditions and memory addresses are independent of sensitive data. The following definition formalizes "ISA constant time" while abstracting from the specific ISA:

Definition 6.1. A program is "ISA constant time" if for every pair of states $\sigma_1 \sim \sigma_2$ and every pair of in-order executions of length n, $\sigma_1 \to^n \sigma'_1$ and $\sigma_2 \to^n \sigma'_2$, it is the case that $\sigma'_1 \approx_{ISA} \sigma'_2$, where $(I, s, C, F) \approx_{ISA} (I', s', C', F')$ iff

(1) $I = I'$, $C = C'$, $F = F'$: the sets of microinstructions, commits and decodes are equal.
(2) If $t \leftarrow c?ld\ \mathcal{M}\ t_a \in I$ or $t \leftarrow c?st\ \mathcal{M}\ t_a\ t_v \in I$ then $[c]\sigma = [c]\sigma'$, $s(t)\downarrow = s'(t)\downarrow$, (whenever defined, guards evaluate the same, and memory operations execute in lockstep) and $[c]\sigma \Rightarrow (\sigma(t_a) = \sigma'(t_a))$ (the same values are used to address memory)

(3) If $t \leftarrow c?st\ \mathcal{PC}\ t_v \in I$ then $[c]\sigma = [c]\sigma'$, $s(t)\downarrow = s'(t)\downarrow$, and $[c]\sigma \Rightarrow (\sigma(t_v) = \sigma'(t_v))$ (the same values are used to update the PC)

The following program (and its MIL translation) exemplifies this policy. It loads register r_1 from address b_1, copies the value of r_1 in r_2 if the flag z is set, and saves the result into b_2.



Suppose that flag z contains sensitive information and the attacker observes only the data cache. The "conditional move" instruction in a_2 executes in constant time [28] and is used to re-write branches that may leak information via the execution time or the instruction cache. This allows the program to always access address b_1 and b_2 unconditionally and execute always the same ISA instructions: In the sequential model the program always produces the sequence of observations $dl\ b_1 :: ds\ b_2$.

Programs that are ISA constant time could be insecure in presence of speculation, as demonstrated by Spectre-PHT [32]. Perhaps surprisingly, it turns out that ISA constant time is not secure even for the OoO model, in absence of speculation. In fact, our analysis of conditional noninterference for ISA constant time programs in the OoO model led to the identification of a class of vulnerable programs, where secrets influence the existence of data dependency between registers. The above program exemplifies this problem: the data dependency between t_{11} and t_{32} exists only if z is set. Concretely, consider two states σ_0 and σ_1 in which $z = 0$ and $z = 1$, respectively. Then, $str\text{-}act(\sigma_1, t_{31}) = \{t_{23}\}$ and $str\text{-}act(\sigma_1, t_{23}) = \{t_{12}\}$, while $str\text{-}act(\sigma_0, t_{31})$ is the microinstruction representing the initial value of r_2. Therefore, state σ_0 may produce the observation sequence $ds\ b_2 :: dl\ b_1$ only if the flag $z = 0$, thus leaking its value through the data cache.

6.2.1 MIL Constant Time. Spectre-OoO motivates the need for a new microarchitecture-aware definition of constant time.

Definition 6.2. A program is "MIL constant time" if for every pair of states $\sigma_1 \sim \sigma_2$ and every pair of in-order executions of length n, $\sigma_1 \rightarrow^n \sigma_1'$ and $\sigma_2 \rightarrow^n \sigma_2'$, it is the case that $\sigma_1' \approx_{MIL} \sigma_2'$, where $(I, s, C, F) \approx_{MIL} (I', s', C', F')$ iff

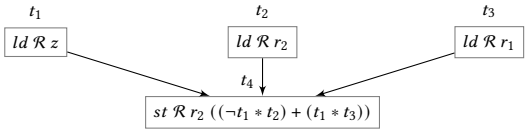
(1) $(I, s, C, F) \approx_{ISA} (I', s', C', F')$
(2) If $t \leftarrow c?ld\ \mathcal{R}\ t_a \in I$ or $t \leftarrow c?st\ \mathcal{R}\ t_a\ t_v \in I$ then $[c]\sigma = [c]\sigma'$, $s(t)\downarrow = s'(t)\downarrow$, and $[c]\sigma \Rightarrow (\sigma(t_a) = \sigma'(t_a))$

Notice that in addition to standard requirements of constant time, MIL constant time requires that starting from two $\sim$-indistinguishable states the program makes the same accesses to registers. MIL constant time is sufficient to ensure security in the OoO model:

Theorem 6.3. *If a program P is MIL constant time then P is conditionally noninterferent in the OoO model.*

The theorem enables the enforcement of conditional noninterference for the OoO model by verifying MIL constant time in the sequential model. This strategy has the advantage of performing the verification in the sequential model, which is deterministic, thus making it easier to reuse existing tools for binary code analyses [6].

Finally, we remark that MIL constant time is microarchitecture aware. This means that the same ISA program may or may not satisfy MIL constant time when translated to a given microarchitecture. In fact, the MIL translation of conditional move above is not MIL constant time because of the dependency between the sensitive value in t_{21} and conditional store in t_{23}. However, if a microarchitecture translates the same conditional move as below, the translation is clearly MIL constant time.



7 RELATED WORK

Speculative semantics and foundations Several works have recently addressed the formal foundations of specific forms of speculation to capture Spectre-like vulnerabilities. Cheang et al. [12], Guarnieri et al. [23], and Mcilroy et al. [41] propose semantics that support branch prediction, thus modeling only Spectre v1. Neither work supports speculation of target address, speculation of dependencies, or OoO execution. Disselkoen at al. [14] propose a pomset-based semantics that supports OoO execution and branch prediction. Their model targets a higher abstraction level modeling memory references using logical program variables. Hence, the model cannot support dynamic dependency resolution, dependency prediction, and speculation of target addresses.

Like us, Cauligi et al. [11] propose a model that captures existing variants of Spectre and independently discover a vulnerability similar to our Spectre-STL-D. Remarkably, they demonstrate the feasibility of the attack on Intel Broadwell and Skylake processors. A key difference between the two models is that Cauligi et al. impose sequential order to instruction retire and memory stores. While simplifying the proof of memory consistency and verification, it does not reflect the inner workings of modern CPUs, which reorder memory stores and implement a relaxed consistency model. These features are required to capture Spectre-OoO in Section 6.2. Moreover, our model provides a clean separation between the general speculative semantics and microarchitecture-specific features, where the latter is obtained by reducing the nondeterminism of the former. This enables a modular analysis of (combinations of) predictive strategies, as in Spectre-PHT ICache in Section C.1.3.

Cache side channels In line with prior works [11, 12, 23], our attacker model abstracts away the mechanism used by an attacker to profile the sequence of a victim's memory accesses, providing a general account of trace-driven attacks [46]. Complementary works [15, 20, 37, 59] show that cache profiling is becoming increasingly steady and precise. Performance jitters caused by cache usage have been widely exploited to leak sensitive data [2, 21, 33, 40, 44, 45, 60], e.g., in cryptography software. Miller [42], and Fogh and Ertl [18] propose a taxonomy for mitigating speculative execution

vulnerabilities. We refer to a recent survey by Canella et al. [19] on cache-based countermeasures.

Spectre vs Meltdown Recent attacks that use microarchitectural effects of speculative execution have been generally distinguished as Spectre and Meltdown attacks [9]. We focus on the former [13, 16, 24, 25, 31, 32, 34, 38, 58], which exploits speculation to cause a victim program to transiently access sensitive memory locations that the attacker is not authorized to read. Meltdown attacks [36] transiently bypass the hardware security mechanisms that enforce memory isolation. Importantly, Meltdown attacks can be easily countered in hardware, while Spectre attacks require hardware-software co-design, which motivates our model. We remark that the vulnerability in Section 6.1 is different from the recent Microarchitectural Data Sampling attacks [10, 49, 53], since it only requires the CPU to predict memory aliases with no need of violating memory protection mechanisms. Microarchitectures supporting this feature have been proposed, e.g., in Feiste et al. [17].

Tool support Several prototypes have been developed to reproduce and detect known Spectre-PHT attacks [12, 23, 54, 55]. Checkmate [51] synthesizes proof-of-concept attacks by using models of speculative and OoO pipelines. Tool support for vulnerabilities beyond Spectre-PHT requires dealing with a large number of possible predictions and instruction interleavings. In fact, current tools mainly focus on Spectre-PHT ignoring OoO execution.

Functional Pipeline Correctness A number of authors, cf. [1, 8, 29, 39, 48], have studied the orthogonal problem of functional correctness in the context of concrete pipeline architectures involving features such as OoO and speculation, usually using a complex refinement argument based on Burch-Dill style flushing [8] in order to align OoO executions with their sequential counterparts. Our correlate is the serialization proofs for OoO and speculation provided is the full version of the paper. It is of interest to mechanize these proofs and to examine if a generic account of serialization using, e.g., InSpectre can help also in the functional verification of concrete pipelines.

Hardware countermeasures While CPU vendors and researchers propose countermeasures, it is hard to validate their effectiveness without a model. InSpectre can help modeling and reasoning about their security guarantees, as in Section C.1.1. Similarly, InSpectre can model the hardware configurations and fences designed by Intel [26] to stall (part of) an instruction stream in case of speculation. Several works [30, 50, 56, 57, 61] propose security-aware hardware that prevent Spectre-like attacks. InSpectre can help formalizing these hardware features and analyzing their security.

8 CONCLUDING REMARKS

This paper presented InSpectre, the first comprehensive model capable of capturing out-of-order execution and different forms of speculation that could be implemented in a high-performance pipeline. We used InSpectre to model existing vulnerabilities, to discover three new potential vulnerabilities, and to reason about the security of existing countermeasures proposed in the literature. There are a number of interesting directions left open in this work.

Foundations of microarchitecture security We argue that InSpectre pushes the boundary on foundations of microarchitecture security with respect to the current state-of-the-art substantially. Existing models [11, 12, 14, 23, 41] miss features like dynamic inter-instruction dependency (except [11]]), instruction non-atomicity, OoO memory commits, and partial misprediction of rollbacks. These features were essential to discover the vulnerabilities, as well as to reason about countermeasures like retpoline or memory fences for data dependency. For instance, InSpectre would not have captured our Spectre-OoO vulnerability if the memory stores and instruction retire are performed in the *sequential* order. Similarly, *static* computation of active stores would not have exposed Store-To-Load variants of Spectre. Moreover, forcing the rollback of all subsequent microinstructions as soon as a value is mispredicted prevents modeling advanced recovery methods used by modern processors, including concurrent and partial recovery in case of multiple mispredictions.

A novel feature of our approach is to decompose instructions into smaller microinstruction-like units. We argue that the modeling of pipelines using ISA level instructions as atomic units is in the long run the wrong approach, not reflecting well the behavior at the hardware level, and inelegant as a foundation for real pipeline information flow. Non-atomicity is needed to handle, for instance, intra-instruction dependencies and interactions between I/D-caches. Therefore, decomposing instructions into smaller microinstructions, as we do, appears convenient.

InSpectre lacks explicit support of Meltdown-like vulnerabilities, multicore and hyperthreading, fences, TLBs, cache eviction policies, and mechanisms used to update branch predictor tables. Our model can already capture many of these features. In the paper we give Intel's *lfence* as an example. We focus here on core aspects of out-of-order and speculation, but there is nothing inherent in the framework that prevents modeling the above additional features. Also, by providing a general model we cannot currently argue if a concrete architecture is secure. For that we need to specialize the model to a given architecture, by adding detail and eliminating nondeterminism.

Tooling Tooling is needed to explore more systematically the utility of the model for exploit search and countermeasure proof, and the framework needs to be instantiated to different concrete pipeline architectures and be experimentally validated.

One can envisage MIL-based analysis tools like Spectector [23], Pitchfork [11], and oo7 [55]. However, the large nondeterminism introduced by out-of-order and speculation will make such an approach inefficient. We are currently taking a different route by modeling concrete microarchitectures within a theorem prover. This allows verifying conditional noninterference if the microarchitecture is inherently secure. A failing security proof gives a basis for proving countermeasure soundness as in Section 6.2, and the identification of sufficient conditions that can be verified in the (more tractable) sequential model.

ACKNOWLEDGMENTS

We thank our shepherd Andrew Myers and anonymous reviewers for the helpful feedback. The work was partially supported by the Swedish Foundation for Strategic Research (SSF) through framework project TrustFull, by the Swedish Civil Contingencies Agency through framework project CERCES, and by the Swedish Research Council (VR) through project JointForce.

REFERENCES

[1] Mark D. Aagaard, Byron Cook, Nancy A. Day, and Robert B. Jones. 2001. A Framework for Microprocessor Correctness Statements. In *Correct Hardware Design and Verification Methods*. 433–448.

[2] Onur Acıiçmez and Çetin Kaya Koç. 2006. Trace-driven cache attacks on AES (short paper). In *International Conference on Information and Communications Security*. Springer, 112–121.

[3] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016. Verifying constant-time implementations. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*. 53–70.

[4] ARM. 2018. SSBS, Speculative Store Bypass Safe. https://developer.arm.com/docs/ddi0595/d/aarch64-system-registers/ssbs Accessed: 2020-01-16.

[5] ARM. 2019. Cache Speculation Side channels v2.4.

[6] Musard Balliu, Mads Dam, and Roberto Guanciale. 2014. Automating Information Flow Analysis of Low Level Code. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*. 1080–1091.

[7] Daniel J. Bernstein. 2005. Cache-timing attacks on AES. http://cr.yp.to/antiforgery/cachetiming-20050414.pdf.

[8] Jerry R. Burch and David L. Dill. 1994. Automatic verification of pipelined microprocessor control. In *Computer Aided Verification*, David L. Dill (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 68–80.

[9] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*. 249–266.

[10] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. 2019. Fallout: Leaking Data on Meltdown-resistant CPUs. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM.

[11] Sunjay Cauligi, Craig Disselkoen, Klaus v Gleissenthall, Deian Stefan, Tamara Rezk, and Gilles Barthe. 2019. Towards Constant-Time Foundations for the New Spectre Era. *arXiv preprint arXiv:1910.01755* (October 2019).

[12] Kevin Cheang, Cameron Rasmussen, Sanjit Seshia, and Pramod Subramanyan. 2019. A Formal Approach to Secure Speculation. In *CSF 2019*.

[13] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. 2018. Sgxpectre attacks: Stealing intel secrets from sgx enclaves via speculative execution. *arXiv preprint arXiv:1802.09085* (2018).

[14] Craig Disselkoen, Radha Jagadeesan, Alan Jeffrey, and James Riely. 2019. The Code That Never Ran: Modeling Attacks on Speculative Evaluation. In *S&P 2019*.

[15] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. 2017. Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel {TSX}. In *26th USENIX Security Symposium*. 51–67.

[16] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, and Dmitry Ponomarev. [n.d.]. et almbox. 2018. BranchScope: A new side-channel attack on directional branch predictor. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems. ACM*.

[17] Kurt A. Feiste, John S. Muhich, Larry E. Thatcher, and Steven W. White. 2000. Forwarding store instruction result to load instruction with reduced stall or flushing by effective/real data address bytes matching. (February 2000). https://patents.google.com/patent/US6021485 US6021485A.

[18] Anders Fogh and Christopher Ertl. 2018. Wrangling with the Ghost An Inside Story of Mitigating Speculative Execution Side Channel Vulnerabilities.

[19] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. 2018. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering* 8, 1 (2018), 1–27.

[20] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+ Flush: a fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 279–299.

[21] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. Cache template attacks: Automating attacks on inclusive last-level caches. In *24th USENIX Security Symposium*. 897–912.

[22] Roberto Guanciale, Musard Balliu, and Mads Dam. 2019. InSpectre: Breaking and Fixing Microarchitectural Vulnerabilities by Formal Analysis. arXiv:1911.00868 [cs.CR]

[23] Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. 2020. Spectector: Principled detection of speculative information flows. In *Proceedings of the 41st IEEE Symposium on Security and Privacy*. IEEE.

[24] Jann Horn. 2018. speculative execution, variant 4: speculative store bypass.

[25] Jann Horn et al. 2018. Reading privileged memory with a side-channel. *Project Zero* 39 (2018).

[26] Intel. 2018. Speculative Execution Side Channel Mitigations, Revision 3.0.

[27] Intel. 2018. Speculative Execution Side Channel Update.

[28] Intel. 2019. Guidelines for Mitigating Timing Side Channels Against Cryptographic Implementations. https://software.intel.com/security-software-guidance/insights/guidelines-mitigating-timing-side-channels-against-cryptographic-implementations

[29] Ranjit Jhala and Kenneth L. McMillan. 2001. Microarchitecture Verification by Compositional Model Checking. In *Proceedings of the 13th International Conference on Computer Aided Verification*. Springer-Verlag, 396–410.

[30] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer. 2018. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[31] Vladimir Kiriansky and Carl Waldspurger. 2018. Speculative buffer overflows: Attacks and defenses. *arXiv preprint arXiv:1807.03757* (2018).

[32] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. 1–19.

[33] Paul C Kocher. 1996. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Annual International Cryptology Conference*. Springer, 104–113.

[34] Esmaeil Mohammadian Koruyeh, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2018. Spectre returns! speculation attacks using the return stack buffer. In *12th USENIX Workshop on Offensive Technologies (WOOT)*.

[35] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2018. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *12th USENIX Workshop on Offensive Technologies (WOOT 18)*. USENIX Association.

[36] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. 2018. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium*. 973–990.

[37] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. 2015. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 605–622.

[38] Giorgi Maisuradze and Christian Rossow. 2018. ret2spec: Speculative execution using return stack buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2109–2122.

[39] P. Manolios and S. K. Srinivasan. 2005. Refinement maps for efficient verification of processor models. In *Design, Automation and Test in Europe*. 1304–1309 Vol. 2.

[40] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. 2017. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud.. In *NDSS*, Vol. 17. 8–11.

[41] Ross Mcilroy, Jaroslav Sevcik, Tobias Tebbi, Ben L Titzer, and Toon Verwaest. 2019. Spectre is here to stay: An analysis of side-channels and speculative execution. *arXiv preprint arXiv:1902.05178* (2019).

[42] Matt Miller. 2018. Mitigating speculative execution side channel hardware vulnerabilities. https://msrc-blog.microsoft.com/2018/03/15/mitigating-speculative-execution-side-channel-hardware-vulnerabilities/

[43] David Molnar, Matt Piotrowski, David Schultz, and David A. Wagner. 2005. The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks. In *Information Security and Cryptology - ICISC 2005, 8th International Conference, Seoul, Korea, December 1-2, 2005, Revised Selected Papers*. 156–168.

[44] Michael Neve and Jean-Pierre Seifert. 2006. Advances on access-driven cache attacks on AES. In *International Workshop on Selected Areas in Cryptography*. Springer, 147–162.

[45] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache attacks and countermeasures: the case of AES. In *Cryptographers' track at the RSA conference*. Springer, 1–20.

[46] Dan Page. 2002. Theoretical use of cache memory as a cryptanalytic side-channel. *IACR Cryptology ePrint Archive* 2002, 169 (2002).

[47] Andrew Pardoe. 2018. Spectre mitigations in MSVC.

[48] Jun Sawada and Warren A Hunt. 2002. Verification of FM9801: An out-of-order microprocessor model with speculative execution, exceptions, and program-modifying capability. *Formal Methods in System Design* 20, 2 (2002), 187–222.

[49] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *CCS*.

[50] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. 2019. Context-sensitive fencing: Securing speculative execution via microcode customization. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 395–410.

[51] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. 2018. CheckMate: Automated Synthesis of Hardware Exploits and Security Litmus Tests. In *51st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2018, Fukuoka, Japan, October 20-24, 2018*. 947–960.

[52] Paul Turner. 2018. Retpoline: a software construct for preventing branch-target-injection. https://support.google.com/faqs/answer/7625886
[53] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2019. RIDL: Rogue In-flight Data Load. In *S&P*.
[54] Guanhua Wang, Sudipta Chattopadhyay, Arnab Kumar Biswas, Tulika Mitra, and Abhik Roychoudhury. 2019. KLEESPECTRE: Detecting Information Leakage through Speculative Cache Attacks via Symbolic Execution. *arXiv preprint arXiv:1909.00647* (2019).
[55] G. Wang, S. Chattopadhyay, I. Gotovchits, T. Mitra, and A. Roychoudhury. 2019. oo7: Low-overhead Defense against Spectre attacks via Program Analysis. *IEEE Transactions on Software Engineering* (2019).
[56] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas F Wenisch, and Baris Kasikci. 2019. NDA: Preventing Speculative Execution Attacks at Their Source. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 572–586.
[57] Jonathan Woodruff, Robert NM Watson, David Chisnall, Simon W Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G Neumann, Robert Norton, and Michael Roe. 2014. The CHERI capability model: Revisiting RISC in an age of risk. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE, 457–468.
[58] Yuan Xiao, Yinqian Zhang, and Mircea-Radu Teodorescu. 2020. SPEECHMINER: A Framework for Investigating and Measuring Speculative Execution Vulnerabilities. In *NDSS*.
[59] Yuval Yarom and Katrina Falkner. 2014. FLUSH+ RELOAD: a high resolution, low noise, L3 cache side-channel attack. In *23rd USENIX Security Symposium*. 719–732.
[60] Yuval Yarom, Daniel Genkin, and Nadia Heninger. 2017. CacheBleed: a timing attack on OpenSSL constant-time RSA. *Journal of Cryptographic Engineering* 7, 2 (2017), 99–112.
[61] Drew Zagieboylo, G. Edward Suh, and Andrew C. Myers. 2019. Using Information Flow to Design an ISA that Controls Timing Channels. In *32nd IEEE Computer Security Foundations Symposium, CSF*. 272–287.

A SECURITY CONDITION

We now elucidate the advantages of conditional noninterference as compared to standard notions of noninterference and declassification. Suppose we define the security condition directly on the target model, in the style of standard noninterference.

Definition A.1 (Noninterference). Let P be a program with transition relation $\rightarrow$ and $\sim_P$ a security policy. P satisfies noninterference up to $\sim_P$ if for all $\sigma_1, \sigma_2 \in \mathit{States}$ such that $\sigma_1 \sim_P \sigma_2$ and executions $\pi_1 = \sigma_1 \rightarrow \cdots$, there exists an execution $\pi_2 = \sigma_2 \rightarrow \cdots$ such that $\mathit{trace}(\pi_1) = \mathit{trace}(\pi_2)$.

Noninterference ensures that if the observations do not enable an attacker to refine his knowledge of sensitive information beyond what is allowed by the policy $\sim_P$, the program can be considered secure. Noninterference can accommodate partial release of sensitive information by refining the definition of the indistinguishability relation $\sim_P$. In our context, a precise definition of $\sim_P$ can be challenging to define. However, we ultimately aim at showing that the OoO/speculative model does not leak more information than the in-order (sequential) model, thus capturing the intuition that microarchitectural features like OoO and speculation should not introduce additional leaks. Therefore, instead of defining the policy $\sim_P$ explicitly, we split it into two relations $\sim$ (as in Def. 2.1) and $\sim_D$, where the former models information of the initial state that is known by the attacker, i.e., the public resources, and the latter models information that the attacker is allowed to learn during the execution via observations. Hence, $\sim_P = \sim \cap \sim_D$. This characterization allows for a simpler formulation of the security condition that is transparent on the definition of $\sim_D$, as described in Def. 2.1.

B IN-ORDER SEMANTICS

We define the in-order (i.e., sequential) semantics by restricting the scheduling of the out-of-order semantics and enforcing the execution of microinstructions in program order.

Say that a microinstruction $\iota = t \leftarrow c?o$ is *completed* in state σ (written $C(\sigma, \iota)$) if one of the following conditions hold:

- The instruction's guard evaluates to false in σ, i.e. $\neg[c](\sigma)$.
- The instruction has been executed and is not a memory or a program counter store, i.e. $o \neq \mathit{st}\ \mathcal{M}\ t_a\ t_v \wedge o \neq \mathit{st}\ \mathit{pc}\ \ t_v \wedge \sigma(t)\downarrow$.
- The instruction is a committed memory store or a fetched and decoded program counter store, i.e. $t \in C \cup F$

The in-order transition rule allows an evaluation step to proceed only if program-order preceding microinstructions have been completed.

$$\frac{\sigma \overset{l}{\twoheadrightarrow} \sigma' \qquad \mathit{step\text{-}param}(\sigma, \sigma') = (\alpha, t) \qquad \forall \iota \in \sigma \text{ if } \mathit{bn}(\iota) < t \text{ then } C(\sigma, \iota)}{\sigma \overset{l}{\rightarrow} \sigma'}$$

It is easy to show that the sequential model is deterministic. In fact, the OoO model allows each transition to modify one single name t, while the precondition of the in-order rule forces all previous instructions to be completed, therefore only one transition is enabled.

Definition B.1. Let $\sigma_1 :: \cdots :: \sigma_n$ be the sequence of states of execution π, then $\mathit{commits}(\pi, a)$ is the list of memory commits at address a in π, and is empty if $n < 2$; $v :: \mathit{commits}(\sigma_2 :: \cdots :: \sigma_n, a)$ if $\mathit{step\text{-}param}(\sigma_1, \sigma_2) = (\textsc{Cmt}(a, v), t)$; and $\mathit{commits}(\sigma_2 :: \cdots :: \sigma_n, a)$ otherwise.

We say that two models are memory consistent if writes to the same memory location are seen in the same order.

Definition B.2. The transition systems $\rightarrow_1$ and $\rightarrow_2$ are *memory consistent* if for any program and initial state σ_0, for all executions $\pi = \sigma_0 \rightarrow_1^* \sigma$, there exists $\pi' = \sigma_0 \rightarrow_2^* \sigma'$ such that for all $a \in \mathcal{M}$ $\mathit{commits}(\pi, a)$ is a prefix of $\mathit{commits}(\pi', a)$.

Intuitively, two models that are memory consistent yield the same sequence of memory updates for each memory address. This ensures that the final result of a program is the same in both models. Notice that since we do not assume any fairness property for the transition systems then an execution π of $\rightarrow_1$ may indefinitely postpone the commits for a given address. For this reason we only require to find an execution such that $\mathit{commits}(\pi, a)$ is a prefix of $\mathit{commits}(\pi', a)$. We obtain memory consistency of both the OoO and the speculative semantics against the in-order semantics.

Theorem B.3. *$\twoheadrightarrow$ and $\rightarrow$ are memory consistent.* □

Theorem B.4. *$\twoheadrightarrow$ and $\rightarrow$ are memory consistent.* □

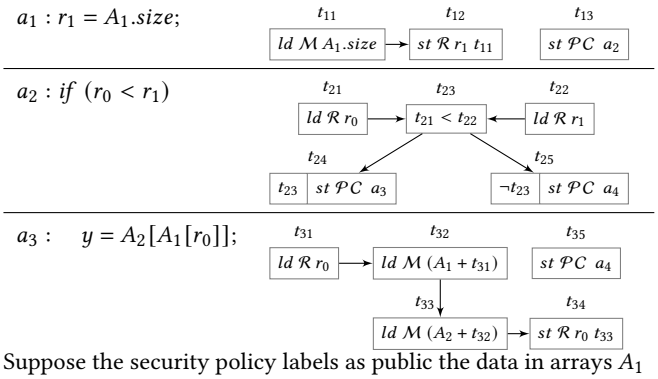
For reasons of space the proofs are deferred to the full version of the paper.

C ATTACKS AND COUNTERMEASURES

C.1 Spectre-PHT

Spectre-PHT [32] exploits the prediction mechanism for the outcome of conditional branches. Modern CPUs use *Pattern History*

Tables (PHT) to record patterns of past executions of conditional branches, i.e., whether the *true* or the *false* branch was executed, and then use it to predict the outcome of that branch. By poisoning the PHT to execute one direction (say the *true* branch), an attacker can fool the prediction mechanism to execute the *true* branch, even when the actual outcome of the branch is ultimately *false*. The following program (and the corresponding MIL) illustrates information leaks via Spectre-PHT:



Suppose the security policy labels as public the data in arrays A_1 and A_2, and in register r_0, and that the attacker controls the value of r_0. This program is secure at the ISA level as it ensures that r_0 always lies within the bounds of A_1. However, an attacker can fool the prediction mechanism by first supplying values of r_0 that execute the *true* branch, and then a value that exceeds the size of A_1. This causes the CPU to perform an out-of-bounds memory access of sensitive data, which is later used as index for a second memory access of A_2, thus leaving a trace into the cache.

Branch prediction predicts values for MIL instructions that block the evaluation of the guard of a PC store whose target address has been already resolved. For $\sigma = (I, s, C, F, \delta, P)$, we model it as:

$$pred_{br}(\sigma) = \left\{t' \mapsto v \mid t \leftarrow c?st\ \mathcal{PC}\ \ t_a \in I \wedge t' \in fn(c) \wedge s(t_a)\downarrow\right\}$$

Let σ_0 be the state where only the instruction in a_1 has been translated. Then $pred_{br}(\sigma_0)$ is empty, since σ_0 contains a single unconditional PC update (the guard of t_{13} has no free names). The CPU may apply rules Exe, Ret, and Ftc on t_{13} without waiting the result of t_{11}. This leads to a new state σ_1 which is obtained by updating the storage with $s_1 = \{t_{13} \mapsto a_2\}$, extending the microinstructions' list with the translation of a_2, and the snapshot with $\delta_1 = \{t_{2i} \mapsto t_{13} \mapsto a_2 \text{ for } 1 \le i \le 5\}$, while producing the observation *il* a_2. In this state $pred_{br}(\sigma_1) = \{t_{23} \mapsto 0, t_{23} \mapsto 1\}$ since the conditions of the two PC stores (i.e., t_{24} and t_{25}) depend on t_{23} which is yet to be resolved. The CPU can now apply rule Prd using the prediction $t_{23} \mapsto 1$, thus guessing that the condition is true. The new state σ_2 contains $s_2 = s_1 \cup \{t_{23} \mapsto 1\}$, $\delta_2 = \delta_1$, and $P_2 = \{t_{23}\}$.

The CPU can follow the speculated branch by applying rules Exe and Ftc on t_{24}, which results in state σ_3 with $s_3 = s_2 \cup \{t_{24} \mapsto a_3\}$, $\delta_3 = \delta_2 \cup \{t_{24} \mapsto \{t_{23} \mapsto 1\}, t_{3i} \mapsto \{t_{24} \mapsto a_3\} \text{ for } 1 \le i \le 5\}$, and $F_3 = \{t_{13}, t_{24}\}$. Additionally, it produces the observation *il* a_3.

Applying rule Exe on t_{31} and t_{32} results in a buffer overread and produces state σ_4 with $s_4 = s_3 \cup \{t_{31} \mapsto r_0, t_{32} \mapsto A_1[r_0]\}$, and observation $dl\ r_0$. Similarly, rule Exe on t_{33} produces the observation $dl\ A_2 + A_1[r_0]$.

Clearly, if $r_0 \ge A_1.size$, the observation reveals memory content outside A_1, allowing an attacker to learn sensitive data. Observe that this is rejected by the security condition, since such observation is not possible in the sequential semantics.

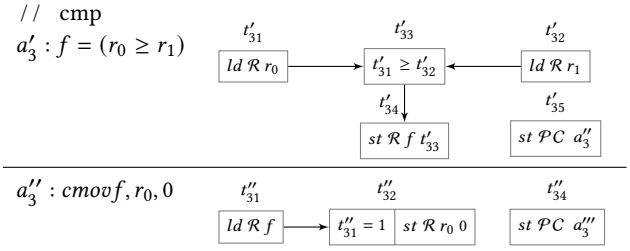
C.1.1 Countermeasure: Serializing Instructions. Serializing instructions can be modeled by constraining the scheduling of microinstructions. For example, we can model the Intel's *lfence* instruction via a function $lfence(I)$ that extracts all microinstructions resulting from the translation of lfence.

Concretely, for $\sigma = (I, s, C, F, \delta, P)$, $t \in lfence(I)$ and $\sigma \twoheadrightarrow \sigma'$, it holds that: (i) if $\sigma(t)\uparrow$ and $\sigma'(t)\downarrow$ then for each $t' \leftarrow c?ld\ \mathcal{M}\ t_a \in \sigma$ such that $t' < t\ fn(c) \subseteq \mathrm{dom}(s) \setminus \mathrm{dom}(\delta)$ and $c \Rightarrow (\sigma(t')\downarrow \wedge \delta(t')\uparrow)$; (ii) if $\sigma(t')\uparrow$, $\sigma'(t')\downarrow$, $t' > t$, and $t' \leftarrow c?ld\ \mathcal{M}\ t_a \in \sigma$, or $t' \leftarrow c?st\ \mathcal{M}\ t_a\ t_v \in \sigma$, or $t' \leftarrow c?st\ \mathcal{R}\ t_a\ t_v \in \sigma$, then $\sigma(t)\downarrow \wedge\ \delta(t)\uparrow$.

Intuitively, the conditions restrict the scheduling of microinstructions to ensure that: (i) whenever a fence is executed, all previous loads have been retired, and (ii) subsequent memory operations or register stores can be executed only if the fence has been retired.

In order to reduce the performance overhead, several works (e.g. [47]) use static analysis to identify necessary serialization points in a program. In the previous example, it is sufficient to place lfence after t_{32} and before t_{33}. This does not prevent the initial buffer overread of t_{32}, however, it suspends t_{33} until t_{32} is retired. In case of misprediction, t_{32} and t_{33} will be rolled back, preventing the observation $dl\ A_2 + A_1[r_0]$ which causes the information leak.

C.1.2 Countermeasure: Implicit Serialization. An alternative countermeasure to prevent Spectre-PHT is to use instructions that introduce implicit serialization [18, 42]. For instance, adding the following gadget between instructions a_2 and a_3 in the previous example prevents Spectre-PHT on existing Intel CPUs:



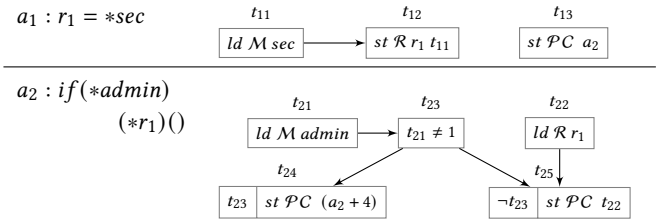
Intuitively, this gadget forces mispredictions to always access $A_1[0]$. Consider the extension of the previous example with the gadget and suppose $pred_{br}$ mispredicts $t_{23} \mapsto 1$. The instruction in a''_3 introduces a data dependency between t_{11} and t_{32} since *str-act* of t_{31} includes t''_{32} until t''_{31} has been executed; *str-act* of t''_{31} includes t'_{34}; and *str-act* of t'_{32} includes t_{12}. These names (and intermediate intra-instruction dependencies) are in the free names of some condition of a PC store, hence they cannot be predicted by $pred_{br}$ and their dependencies are enforced by the semantics. In particular, when t_{23} is mispredicted as 1, t''_{32} is executed after that t_{11} has obtained the value from the memory. This ensures that t''_{32} sets r_0 to 0 every time a buffer overread occurs. Therefore misspeculations generate the observations $dl\ A_1 + 0$ and $dl\ A_2 + A_1[0]$, which do not violate the security condition (since A_1 is labeled as public).

C.1.3 New Vulnerability: Spectre-PHT ICache. When the first Spectre attack was published, some microarchitectures (e.g., Cortex A53) were claimed immune to the attack because of "allowing

speculative fetching but not speculative execution" [5]. The informal argument was that mispredictions cannot cause buffer overreads or leave any footprint on the cache in absence of speculative loads. To check this claim, we constrain the semantics to only allow speculation of PC values. Specifically, we require for any transition $(\sigma, \delta, P) \twoheadrightarrow (\sigma', \delta', P')$ that executes a microinstruction ($step\text{-}param(\sigma, \sigma') = (\text{EXE}, t)$) which is either a load ($t \leftarrow c?ld\ \tau\ t_a \in \sigma$) or a store ($t \leftarrow c?st\ \tau\ t_a\ t_v \in \sigma$) of a resource other than the program counter ($\tau \neq \mathcal{PC}$) to have an empty snapshot on past microinstructions ($\text{dom}(\delta) \cap \{t' \mid t' < t\} = \emptyset$).

The analysis of conditional noninterference for this model led to the identification of a class of counterexamples, which we call Spectre-PHT ICache, where branch prediction causes leakage of sensitive data via an ICache disclosure gadget.

Consider a program that jumps to the address pointed to by `sec` if a user has `admin` privileges, otherwise it continues to address a_3.



In the sequential model, an attacker that only observes the instruction cache can see the sequence of observations $il\ a_1 :: il\ a_2 :: il\ a_2$ if $*admin \neq 1$, otherwise the sequence $il\ a_1 :: il\ a_2 :: il\ sec$.

A CPU that supports only speculative fetching may first complete all microinstructions in a_1, and then predict the result of t_{23} to enable the execution of t_{25}. As a result the PC speculatively fetches the instruction at location *sec* although $*admin \neq 1$. The transition yields the observation sequence $il\ a_1 :: il\ a_2 :: il\ sec$ which was not possible in the sequential model, thus violating the security condition and leaking the value of *sec* via the instruction cache.

Intel's lfence does not stop all microarchitectural operations, like instruction fetching. For this reason lfence may be ineffective against leakage via ICache. In fact, InSpectre reveals that placing a lfence between t_{21} and t_{23} does not prevent the leakage: t_{22}, t_{23}, t_{25} are neither memory operations nor register stores, hence they can be speculated before the execution of the lfence.

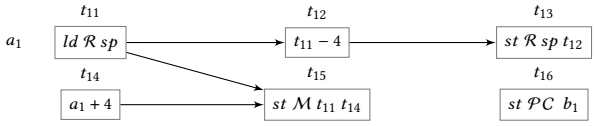
C.2 Spectre-BTB and Spectre-RSB

Two variants of Spectre attacks [9] exploit a CPU's prediction mechanism for jump targets to leak sensitive data. In particular, Spectre-BTB [32] (Branch Target Buffer) poisons the prediction of indirect jump targets. To model this prediction strategy we assume a function $ijmps(I)$ that extracts all PC stores resulting from the translation of indirect jumps. This can be accomplished by making the translation of these instructions syntactically distinguishable from other control flow updates. As a result, prediction is possible for all indirect jumps whose address is yet to be resolved: Namely, $pred_{BTB}(I, s, C, F, \delta, P) =$

$$\{t_a \mapsto v \mid t \leftarrow c?st\ \mathcal{PC}\ \ t_a \in ijmps(I) \wedge s(t_a)\uparrow\}$$

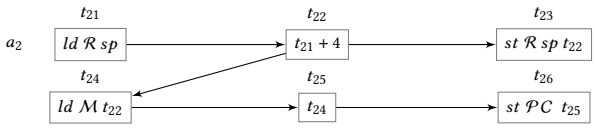
We do not restrict the possible predicted values v, since an accurate model of jump prediction requires knowing the strategy used by the CPU to update the BTB buffer.

Spectre-RSB [35, 38] poisons the Return Stack Buffer (RSB), which is used to temporally store the N most recent return addresses: `call` instructions push the return address on the RSB, while `ret` instructions pop from the RSB to predict the return target. A misprediction can happen if: (*i*) a return address on the stack has been explicitly overwritten, e.g., when a program handles a software exception using `longjmp` instructions, or, (*ii*) returning from a call stack deeper than N, the RSB is empty and the CPU uses the same prediction as for the other indirect jumps. We model `call` and `ret` instructions via program counter stores. A `call` to address b_1 from address a_1 can be modeled as



The `call` instruction saves (e.g. t_{15}) the return address (e.g. $a_1 + 4$) into the stack, decreases the stack pointer (e.g. t_{13}), and jumps to address b_1 (e.g. t_{16}).

A `ret` instruction from address a_2 can be modeled as



The instruction loads the return address from the stack (t_{24}), increases the stack pointer (t_{23}), and returns (t_{26}).

We assume functions $calls(I)$ and $rets(I)$ to extract the PC stores that belong to a `call` and `ret` respectively. Moreover, if $t \in bn(calls(I))$, we use $ret\text{-}ra(I, t)$ to retrieve name of the microinstruction that saves the return address (e.g t_{15}) of the corresponding `call`. We model return address prediction as

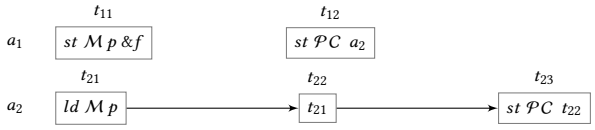
$$\begin{aligned} &pred_{RSB}(I, s, C, F, \delta, P) = \\ &\quad \{t_a \mapsto v \mid t \leftarrow c?st\ \mathcal{PC}\ \ t_a \in rets(I) \wedge s(t_a)\uparrow \wedge \\ &\qquad \exists t' \in bn(calls(I)).\ t' < t \wedge\ s(ret\text{-}ra(I, t')) = v \wedge \\ &\qquad RSB\text{-}depth(I, t', t) \subseteq \{1 \ldots N\}\} \end{aligned}$$

Prediction is possible only for `ret` microinstructions t that have a prior matching `call` t', provided that the size of intermediary stack depth is between 1 and N. We define the latter as the set $RSB\text{-}depth(I, t', t) = \{\#(bn(calls(I)) \cap \{t' \ldots t''\}) - \#(bn(rets(I)) \cap \{t' \ldots t''\}) \mid t' \leq t'' < t\}$, where $\{t' \ldots t''\}$ is an arbitrary continuous sequence of names starting from t' and ending before t'', and $\#(bn(calls(I)) \cap \{t' \ldots t''\})$ and $\#(bn(rets(I)) \cap \{t' \ldots t''\})$ count the number of `call`s and `ret`s in the sequence respectively. The prediction consists in assuming the target address (e.g. t_a) of the `ret` to be equal to the return address (e.g. v) that has been pushed into the stack by the matching `call`.

In some microarchitectures (e.g. [**?**]), RSB prediction falls back to the BTB in case of underflow, i.e., when returing from a function with nested stack deeper than N. This case can be modeled by considering the prediction strategy $pred_{RSB/BTB}$ defined as

$$\begin{aligned} &pred_{RSB} \bigcup \{t_a \mapsto v \mid t \leftarrow c?st\ \mathcal{PC}\ \ t_a \in rets(I) \wedge s(t_a)\uparrow \wedge \\ &\quad \exists t' \in bn(calls(I)).\ t' < t \wedge\ RSB\text{-}depth(I, t', t) = \{1 \ldots N'\} \wedge\ N' > N\} \end{aligned}$$

The following example shows how jump target prediction may violate the security condition. Consider the program `*p:=&f; (*p)()` that saves the address of a function (i.e., $\&f$) in a function pointer at constant address p and immediately invokes the function. Assuming that these instructions are stored at addresses a_1 and a_2, their MIL translation is:

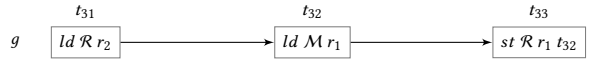


Example 5: `*p:=&f; (*p)()`

Because our semantics can predict only internal operations (see rule Prd), the translation function introduces an additional internal operation, i.e., t_{22} which allows predicting the value of the load t_{21}.

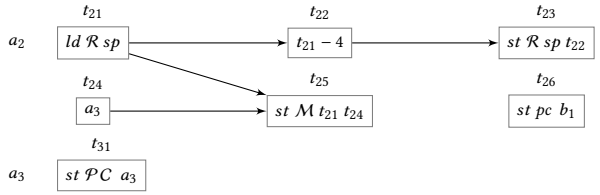
Suppose that the function f simply returns and the security policy labels all data, except the program counter, as sensitive. The program is secure (at the ISA level) as it always transfers control to f, producing the sequence of observations $il\ a_1 :: ds\ p :: il\ a_2 :: dl\ p :: il\ \&f$ independently of the initial state.

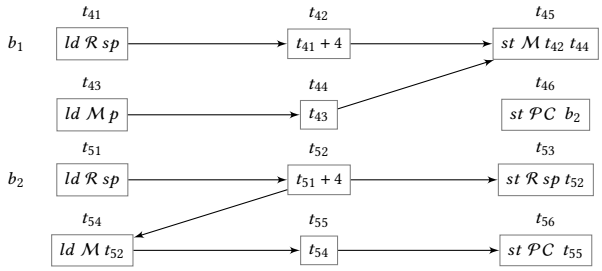
Jump target prediction produces a different behavior. Let σ_0 be the state containing only the translation of the instruction in a_1. Initially, $pred_{BTB}(\sigma_0)$ is empty since the state contains no PC updates (e.g. t_{12}) that result from translating indirect jumps. The CPU may execute and fetch t_{12}, thus adding t_{21}, t_{22}, and t_{23} to the set of microinstructions I. In the resulting state $pred_{BTB}$ is $\{t_{22} \mapsto v \mid v \in V\}$, since t_{23} models an indirect jump and t_{22} has not been executed. The CPU can therefore predict the value of t_{22} without waiting for the result of the load t_{21}. If the predicted value is the address g of the instruction $r_1 := *(r_2)$ the misprediction can use g as gadget to leak sensitive information.



In fact, the speculative semantics can produce the sequence of observations $il\ a_1 :: ds\ p :: il\ a_2 :: dl\ p :: il\ g :: dl\ v$, where v is the initial value of register r_2. The last observation of the sequence allows an attacker to learn sensitive data. Observe that this leak is readily captured by the security condition, since such observation is not possible in the sequential semantics.

C.2.1 Countermeasure: Retpoline. A known countermeasure to Spectre-BTB is the *Retpoline* technique developed by Google [52]. In a nutshell, retpolines are instruction snippets that isolate indirect jumps from speculative execution via call and return instructions. Retpoline has the effect of transforming indirect jumps at address a_2 of Example 5 as:





Instruction at a_2 calls a *trampoline* starting at address b_1 and instruction at a_3 loops indefinitely. The first instruction of the trampoline overwrites the return address on the stack with the value of at address p and its second instruction at b_2 returns.

We leverage our model to analyze the effectiveness of Retpoline for indirect jumps. Since address b_1 is known at compile time, t_{26} does not trigger jump target prediction. While executing the trampoline, the value of t_{55} may be mispredicted, especially if the load from p has not been executed and the store t_{45} is postponed. However, b_2 is a `ret`, hence the value of t_{55} is predicted via $pred_{RSB}$. Since there is no call between a_1 and b_2, then prediction can only assign the address a_3 to t_{55} (i.e., $pred_{RSB}|_{t_{55}} \subseteq \{t_{55} \rightarrow a_3\}$). Therefore, the RSB entry generated by a_2 is used and mispredictions are captured with the infinite loop in a_3. Ultimately, when the value of t_{55} is resolved, the correct return address is used and the control flow is redirected to the value in of $*p$, as expected.