Cats vs. Spectre: An Axiomatic Approach to Modeling Speculative Execution Attacks

Hernán Ponce-de-León
Research Institute CODE
Bundeswehr University Munich
hernan.ponce@unibw.de

Johannes Kinder
Research Institute CODE
Bundeswehr University Munich
johannes.kinder@unibw.de

***Abstract*—The SPECTRE family of speculative execution attacks has required a rethinking of formal methods for security. Approaches based on operational speculative semantics have made initial inroads towards finding vulnerable code and validating defenses. However, with each new attack grows the amount of microarchitectural detail that has to be integrated into the underlying semantics. We propose an alternative, lightweight and axiomatic approach to specifying speculative semantics that relies on insights from memory models for concurrency. We use the CAT modeling language for memory consistency to specify execution models that capture speculative control flow, store-to-load forwarding, predictive store forwarding, and memory ordering machine clears. We present a bounded model checking framework parameterized by our speculative CAT models and evaluate its implementation against the state of the art. Due to the axiomatic approach, our models can be rapidly extended to allow our framework to detect new types of attacks and validate defenses against them.**

I. INTRODUCTION

The promise of formal methods is to provide guarantees of correctness and security in exchange for rigorous specifications and sound layering of abstractions [10]. Unfortunately, the discovery of speculation-based vulnerabilities such as SPECTRE has shown that some of the most basic abstractions are broken at the microarchitectural level in today's mainstream computing architectures [28]. Modern processors execute code speculatively and may have to roll back state if a prediction turns out to be wrong; despite being rolled back, the transient execution can leave traces in the microarchitectural state that an attacker can abuse. By causing the processor to mispredict certain conditions, such as whether a branch is taken or not, an attacker can exfiltrate data despite system-level protection.

There is ongoing work to incorporate microarchitectural effects into formal methods, in order to resolve this impasse [9], [16], [22], [23], [34], [38], [39], [40]. The proposal is appealing: if the semantics used for reasoning about code takes speculation into account, we can reliably identify code patterns vulnerable to transient execution attacks, and we can prove the effectiveness of proposed countermeasures. A key challenge here is the variety of attacks: while commonly referred to under the umbrella term of *speculative execution attacks*, the underlying mechanisms can be very different, and require different semantics.

The majority of approaches rely on defining speculative *operational* semantics. Operational semantics describe *how* a valid program is interpreted, as sequences of computational steps. This requires descriptions of microarchitectural implementation details such as buffers or recovery mechanisms for wrong predictions. For example, operational semantics for speculative execution record snapshots of the state before every prediction to enable subsequent rollbacks. This results in complex models and complications with nested speculations, which have been listed as one of the main challenges of modeling speculative execution [23], [32]. Porting an operational speculative semantics to incorporate a different class of attack is no easy task, and no such approach covers all known attacks.

Axiomatic semantics, as an alternative to the operational approach, define *which* executions are valid. The axiomatic approach has been successful in reasoning about concurrency and weak memory models. Its elegance lies in being able to succinctly express which dependencies among reads and writes are enforced. This enables a modular style of reasoning. To describe different memory models, Alglave et al. introduced the relational language *CAT*, which is expressive enough to axiomatize the concurrency semantics of processors like X86, POWER and ARM [1], [2], [4].

We argue that the axiomatic approach and CAT in particular lend themselves equally well to modeling speculative execution semantics. To this end, we develop a set of models describing speculative behaviors and their effects. We find CAT to be ideally suited to capture a variety of attacks in a simple, concise and unified manner. While the similarity between weak memory models and speculation has been noted before [13], [20], [37], we are the first to prove their flexibility by providing axiomatic weak memory-style models for several variants of speculative execution attacks and a concrete tool to detect vulnerable code and validate defenses. In a recent survey, Cauligi et al. [8] stated that "*they are only suited for analyzing particular Spectre variants [...] and are difficult to adapt to other attacks*" and that "*it remains an open problem to translate a semantics of this style into a concrete analysis tool*".

The key advantages of defining axiomatic semantics with CAT are *simplicity* and *modularity*, which lead to reliable and rapid development of verification tools. Because the

Basic Types

(Registers)	r	$\in$	*Regs*
(Values)	n	$\in$	$\mathbb{N}$

Syntax

(Expressions)	$\langle exp \rangle$	:=	$r \mid n \mid \ominus \langle exp \rangle \mid \langle exp \rangle \otimes \langle exp \rangle \mid$ **sec**
(Statements)	$\langle stm \rangle$	:=	$r \leftarrow \langle exp \rangle \mid r \xleftarrow{\langle exp \rangle ?} \langle exp \rangle$
		|	**load** $r, \langle exp \rangle$ | **store** $r, \langle exp \rangle$
		|	**jmp** ℓ | **beqz** r, ℓ
		|	**skip** | **fence**
(Labels)	ℓ	$\in$	$\mathbb{N}$
(Programs)	$\langle p \rangle$	:=	$\ell : \langle stm \rangle \mid \langle p \rangle ; \langle p \rangle$

Fig. 1. μASM syntax.

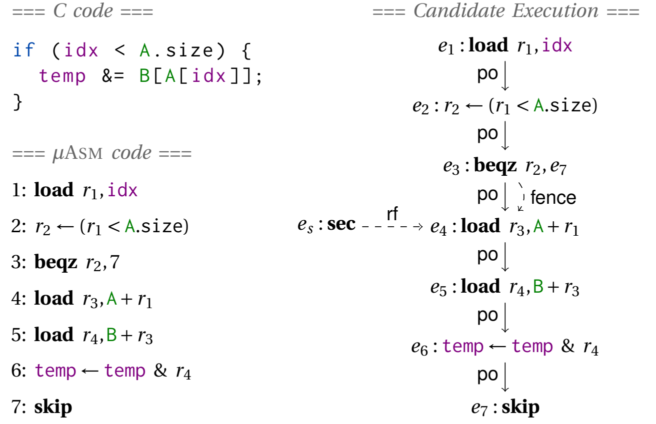


Fig. 2. SPECTRE-v1 – impossible under traditional semantics, but possible under control flow speculation.

models are simple and concise, the resulting analysis tools are less involved. Because the speculative semantics defined as CAT models are modular, we can quickly adapt to new types of attacks. A case in point is that in the final phase of preparing this paper, Ragab et al. [35] presented machine clears as a new source for transient execution. Following the paper's description, we were able to define a CAT model to handle the memory ordering machine clear in less than two hours (see §IV-E).

Overall, this paper makes the following contributions:

- We show how to use the CAT language to axiomatically describe the effects of microarchitectures and provide concrete models that capture the behaviors underlying known SPECTRE attacks: speculative control flow (§III), store-to-load forwarding, predictive store forwarding, and memory ordering machine clear (§IV).
- We propose an analysis framework that is parametric in its microarchitectural model (defined via CAT). The analysis is an instance of a Bounded Model Checking problem, implemented in the tool KAIBYO, which we use to verify software isolation (§V).
- We evaluate our framework and compare its precision, flexibility, and performance against state-of-the-art tools to detect SPECTRE vulnerabilities (§VI).

II. Preliminaries

We begin by introducing the target assembly language (§II-A) and its semantics (§II-B). We then present the CAT language (§II-C), followed by a brief description of speculative execution attacks (§II-D), the threat model we consider (§II-E), and possible extensions (§II-F).

A. Input Language

We target μASM, a core assembly language defined by Guarnieri et al. [23]. The syntax is given in Fig. 1. Registers and constants over $\mathbb{N}$ form the base expressions; more complex expressions are built using the usual unary and binary operators. Besides computing values, expressions are used as memory addresses. We introduce the special expression **sec** to represent the address of a secret (more on this in §II-E).

The statements of the μASM language are (conditional) local assignment, memory **load** and **store**, direct (**jmp**) and conditional (**beqz**) jump, **fence** and **skip**. A statement is paired with a unique label $\ell \in \mathbb{N}$. The resulting pair $\ell : s$ is called an instruction. We refer to instructions by their labels. We use $reg(\ell)$ and $exp(\ell)$ to refer to the target register and the memory expression of a given instruction, respectively. Finally, programs are sequences of instructions.

Statement $r \leftarrow e$ computes the value of expression e and locally assigns it to register r. The conditional assignment $r \xleftarrow{e'?} e$ takes effect only if e' does not evaluate to 0. Statement **load** r, e assigns the value at the address computed from e to register r. Conversely, **store** r, e assigns the value of r to the address computed from e. Statement **jmp** ℓ directly redirects the control flow to ℓ. The conditional jump **beqz** r, ℓ redirects the control flow to ℓ only if the value of register r evaluates to 0. A **fence** enforces an ordering constraint on memory operations. A **skip** statement has no computational effect. We define the set of static predecessors of label ℓ as

$$\text{PRED}(\ell) \quad := \quad \{\ell' \in \mathbb{N} \mid (\ell' + 1 = \ell \wedge \ell' \neq \textbf{jmp } \ell'') \vee \ell' = \textbf{jmp } \ell \vee \ell' = \textbf{beqz } r, \ell\}.$$

In slight abuse of notation, we write $\ell = \textbf{jmp } \ell'$ here to mean that $\ell : \textbf{jmp } \ell'$ is an instruction contained in the program. As a result of the definition, PRED(ℓ) contains the instruction immediately preceding ℓ in the program order, if that is not a **jmp** instruction, and any jump (direct or conditional) in the program targeting ℓ.

Consider the left of Fig. 2 which shows a code snippet written in C (top) and μASM (bottom). Variable `idx` is an input and `A.size` is the length of the array `A`. If $r_2 \neq 0$ (meaning that `idx` is in bounds for `A`), instruction 4 loads the value of `A[idx]` (here represented by accessing address $A + r_1$) into register r_3; if not, the program terminates by jumping to 7. Since the input is compared with the size of the array, instruction 4 cannot read from arbitrary memory. In particular, it cannot access the secret at address **sec**. The loaded value in r_3 is used by instruction 5 for accessing

a second array B. For simplicity, we assume the values of A are smaller than the size of B, so that there is no need for a second bounds check. We then can conclude that the program cannot access memory out of bounds.

B. CAT Semantics of μASM

We define the semantics of a program axiomatically in terms of its *consistent executions*, following the CAT approach introduced by Alglave et al. [1], [4] for formalizing weak memory models. Behaviors of a program are represented by graphs where nodes (called events) model occurrences of instructions and edges model relations or dependencies. Given a program, we proceed in two steps. First, we define all possible behaviors or *candidate executions*, which satisfy basic properties about control and data flow. Second, we filter out behaviors that are invalid with respect to the target semantics, using a set of assertions given as a CAT model. The remaining behaviors form the set of consistent executions and define the semantics of the program. Different assertions yield different CAT models, each of them describing one concrete semantics. Note that for a given program, different assertions might result in the same set of consistent executions. When this is true for every possible program, the CAT models are equivalent. Our CAT model for in-order semantics is given in Fig. 4.

An *event* is the representation of an occurrence or instance of an executed instruction. Let $\mathbb{X}$ be a set of events representing a behavior, i.e., the nodes of the corresponding graph. There are certain properties $\mathbb{X}$ must fulfill to guarantee that the behavior represents a possible control flow of the program. Since, on top of traditional control flow, we model speculative execution, we leave such properties underspecified here; they are formalized in §III. For loop-free programs there is a one to one correspondence between events and executed instructions: there is a unique instance of ℓ represented by $e_\ell \in \mathbb{X}$. In the presence of loops, if ℓ is in a loop that is executed n times, then there are n such instances and $\{e_\ell^1, \dots, e_\ell^n\} \subseteq \mathbb{X}$. Events coming from memory instructions form the set $\mathbb{M} \subseteq \mathbb{X}$, which is further split into $\mathbb{R}$ and $\mathbb{W}$ depending on whether instructions come from **load** or **store** statements, i.e., $\mathbb{M} = \mathbb{R} \uplus \mathbb{W}$. By $\mathbb{M}_a$ we refer to memory events that access an address $a \in \mathbb{N}$. Finally, if ℓ is an instruction using a register and e_ℓ is an event representing an instance of ℓ, we use $val(e_\ell)$ to represent the value of the register for that given instance.

Relations form the edges of execution graphs. The *location relation* loc forms equivalence classes between memory events accessing the same address, i.e., $\mathsf{loc} := \{(e_\ell, e_{\ell'}) \mid \exists a \in \mathbb{N} : e_\ell, e_{\ell'} \in \mathbb{M}_a\}$. The *reads-from relation* rf gives for each read a unique write to the same address from which the read obtains its value:

$$\mathsf{rf} \subseteq (\mathbb{W} \times \mathbb{R}) \cap \mathsf{loc}$$
$$\forall r \in \mathbb{R} : \exists! w \in \mathbb{W} : \mathsf{rf}(w, r)$$
$$\mathsf{rf}(w, r) \Rightarrow val(w) = val(r)$$

The last constraint above defines how the data flows between different instructions. The candidate execution of Fig. 2 represents a behavior where the initial value of the secret (here represented by e_s) flows to the access to A at index r_1; this is represented by the edge $\mathsf{rf}(e_s, e_4)$.

The *coherence order* co relates writes to the same address and forms a total order for each address:

$$\mathsf{co} \subseteq (\mathbb{W} \times \mathbb{W}) \cap \mathsf{loc}$$
$$\forall a \in \mathbb{N} : total(\mathsf{co}, \mathbb{W}_a)$$

Coherence models the order in which **store** instructions hit the main memory. For each address, we assume the existence of a write event assigning its initial value. Those events come first in the coherence order. In figures we represent initial writes to all addresses by a unique event $e_0 : init$. The only exception is address **sec** for which we use $e_s : \mathbf{sec}$; this event models the initial value of the secret.

The *program order* $\mathsf{po} \subseteq (\mathbb{X} \times \mathbb{X})$ represents the order in which instructions are written. For instructions not being part of the same loop we have $(e_\ell, e_{\ell'}) \in \mathsf{po} \Rightarrow \ell < \ell'$. For instructions belonging to the same loop, we have that

$$(e_\ell^i, e_{\ell'}^j) \in \mathsf{po} \Rightarrow (\ell < \ell' \wedge i \leq j) \vee (\ell \geq \ell' \wedge i < j).$$

In Fig. 2 we have po edges between all events e_1-e_7; they represent the order of the corresponding instructions in the μASM code. Relation fence contains every pair of events for which there is a **fence** instruction in between, i.e., $\mathsf{fence} := \{(e_\ell, e_{\ell'}) \subseteq \mathsf{po} \mid \exists \ell'' : \mathbf{fence} : \ell < \ell'' < \ell'\}$. Note that relations co, po and fence are transitive, although we represent them in diagrams only by their direct edges. The *address dependency* addr relates reads with memory events using the loaded value for computing their addresses:

$$\mathsf{addr} := \{(e_\ell, e_{\ell'}) \subseteq (\mathbb{R} \times \mathbb{M}) \cap \mathsf{po} \mid reg(\ell) \in exp(\ell') \wedge \nexists \ell'' : \ell < \ell'' < \ell' \wedge reg(\ell'') = reg(\ell)\}$$

A candidate execution is a triple $(\mathbb{X}, \mathsf{rf}, \mathsf{co})$. Each different combination of these three yields a possible behavior. Once $\mathbb{X}$ is fixed, relations po, fence and addr can be statically computed from the program. The right part of Fig. 2 shows one candidate execution for the program on the left (note that certain rf edges are omitted for clarity).

C. Specifying Memory Models with CAT

Each candidate execution as defined in §II-B yields a possible behavior. On a given processor, however, only some of those behaviors can occur in practice. A memory model defines which behaviors are allowed, by specifying which values a **load** instruction can read. We use CAT, the core of which is given in Fig. 3, to formalize this. CAT is concise but expressive enough to specify a wide range of memory models. It has been used to clarify and formalize the concurrency semantics not only of processors like X86, POWER and ARM, but also the memory model of C11 and the Linux kernel [3], [4], [6], [33], [36]. Moreover, the CAT model of ARMv8 is part of its official documentation [2]. In this paper we show that CAT can also be used to model the

$$\begin{aligned}
\langle MCM \rangle &:= \langle assert \rangle \mid \langle rel \rangle \mid \langle MCM \rangle \wedge \langle MCM \rangle \\
\langle assert \rangle &:= \mathsf{acyclic}\langle r \rangle \mid \mathsf{irreflexive}\langle r \rangle \mid \mathsf{empty}\langle r \rangle \\
\langle r \rangle &:= \langle b \rangle \mid \langle r \rangle \cup \langle r \rangle \mid \langle r \rangle \cap \langle r \rangle \mid \langle r \rangle \setminus \langle r \rangle \\
&\mid \langle r \rangle^{-1} \mid \langle r \rangle^{+} \mid \langle r \rangle^{*} \mid \langle r \rangle ; \langle r \rangle \\
\langle b \rangle &:= \mathsf{po} \mid \mathsf{fence} \mid \mathsf{rf} \mid \mathsf{co} \mid \mathsf{loc} \mid \mathsf{addr} \\
&\mid [\langle set \rangle] \mid \langle set \rangle \times \langle set \rangle \mid \langle name \rangle \\
\langle set \rangle &:= \mathbb{X} \mid \mathbb{M} \mid \mathbb{W} \mid \mathbb{R} \\
\langle rel \rangle &:= \langle name \rangle = \langle r \rangle
\end{aligned}$$

Fig. 3. Core of the CAT language [1].

effects of microarchitectural optimizations such as branch and alias predictors.

The role of the memory model is to filter out the candidate executions that are not consistent according to the intended semantics. In CAT, a memory model is a constraint system over so-called *derived relations*. Derived relations are built from the base relations described in §II-B, hand-defined relations that refer to the different sets of events, and named relations that we will explain in a moment. CAT supports operators like union, intersection, difference, inverse, transitive (and reflexive) closure and composition. The assertions that filter candidate executions are acyclicity, irreflexivity and emptiness constraints over derived relations. As an example, our CAT model for in-order semantics in Fig. 4 defines the derived relation com as the union of co, rf and the composition of rf^{-1} with co. The model states that the union of com and po must be acyclic. CAT also supports recursive definitions of relations. We assume a set $\langle name \rangle$ of relation names (different from the predefined relations) and require each name used in the memory model to have a defining equation $\langle name \rangle = \langle r \rangle$. Notably, $\langle r \rangle$ may again contain named relations, making the system of defining equations recursive. The relations then are defined to be the least solution to this system of equations.

D. Speculative Execution Attacks

Speculative execution uses different predictors to guess, e.g., the outcome of branching instructions or aliasing of addresses. A prediction opens a *speculation window* during which instructions are executed without knowing whether the prediction was correct. Once the window closes, the effects of the speculatively executed instructions are committed (if the prediction was correct) or rolled back (if the prediction was wrong).

Speculatively executed instructions under a misprediction are called *transient*. When speculation is over, all directly visible effects of transient instructions are discarded. However, this is only the case for the architectural state. If the cache or other microarchitectural components have been modified by these instructions, those effects are not (or only partially) discarded. This can allow sensitive data to be leaked, e.g., through cache timing [28], [31]. In these attacks, the size of the speculation window determines the number of operations an attacker can issue on a transient path before the results are squashed.

The candidate execution in Fig. 2 represents a Spectre-v1 gadget [28]. An attacker can bypass the bounds check in the following way [7]: first, during the *setup phase*, the attacker invokes the code with valid values of `idx`, thereby training the branch predictor to expect the jump not to be taken. Second, during the *transient execution phase*, the attacker invokes the program with a value of `idx` outside the bounds of `A`. Rather than waiting for the result of the comparison (e.g., if the value of `A.size` is not cached), the processor guesses that the bounds check will be true and transiently executes instructions 4 and 5 using the malicious `idx`. Note that instruction 5 loads data into the cache in an address that is dependent on `A[idx]`. When the result of the bounds check is eventually determined, the processor rolls back the effect of 4 and 5. However, changes made to the cache state are not reverted. Finally, in the *decoding phase*, the attacker can use a side channel to analyze the cache contents and retrieve the value of the secret [31].

E. Threat Model

An attacker in our setting is an arbitrary program that interacts with a program of interest (the victim). We require some interaction between the attacker and the victim that allows to start a transient execution phase. Root causes of transient execution include branch and alias mispredictions, but also machine clears [35]. For example, to exploit Spectre-v1 in Fig. 2 it is sufficient that the attacker has access to the input `idx` to mistrain the branch predictor. We also assume both the attacker and the victim share the same cache, which the attacker analyzes during the decoding phase to retrieve the secret. The attacker can access the cache after execution of any instruction and does not need to wait for termination of the victim program.

The goal of the attacker is to break software isolation by reading a secret from address **sec** outside its sandbox boundary. While the attacker cannot directly access **sec**, they can trick the victim into leaking sensitive information. We consider data leakage both under normal and speculative execution. Our attacker observes the address of executed **load** instructions to see if any match **sec**. This is a common leakage model for speculative isolation [8].

To model the effects of speculative execution, our semantics can mispredict the outcome of all branch instructions and the address of all memory instructions in the victim. This is the worst-case scenario in terms of leakage regardless of how attackers poison predictors.

F. Beyond Software Isolation

While we focus on software isolation, our semantics can also serve as a building block for other properties. We show in Sections III and IV that the proposed semantics captures behaviors that are unobservable from the architectural point of view. Guarnieri et al. [24] recently proposed a framework

to specify hardware-software contracts and guarantee non-interference-style properties, such as constant time. In this framework, contracts are formed of an execution mode (the semantics) and an observation mode (capturing the threat model). While the CAT models we present in the following can serve as an execution mode, defining an observation mode in axiomatic semantics is an open problem and left for future work.

III. Speculative Control Flow

In this section, we show how we model control flow (§III-A) and its speculation (§III-B) axiomatically to detect Spectre-v1 (also known as Spectre-PHT). We also discuss the speculation window (§III-C) and how to mitigate the attack (§III-D).

A. Traditional Control Flow

In a traditional model of computation, instructions are fetched, executed and retired in order (see §IV-A). In this setting, the set $\mathbb{X}$ (see §II-B) is entirely defined by the value of conditions in jump instructions. The following cases relate instructions ℓ, ℓ' along the same path. If ℓ immediately follows ℓ' in the program order and ℓ' is not a conditional jump targeting ℓ, then ℓ can only execute if ℓ' does:

$$\textit{If } \ell' + 1 = \ell \wedge \ell' \neq \textbf{beqz } r, \ell'', \textit{ then } e_\ell \in \mathbb{X} \Rightarrow e_{\ell'} \in \mathbb{X}$$

When ℓ is the target of the direct jump ℓ', then ℓ can only execute if ℓ' does:

$$\textit{If } \ell' = \textbf{jmp } \ell, \textit{ then } e_\ell \in \mathbb{X} \Rightarrow e_{\ell'} \in \mathbb{X}$$

If ℓ' is both the immediate predecessor of ℓ and a conditional jump **beqz** r, ℓ'' targeting a different label, the dependency requires that $r \neq 0$, otherwise the jump would be taken:

$$\textit{If } \ell' + 1 = \ell \wedge \ell' = \textbf{beqz } r, \ell'', \textit{then } e_\ell \in \mathbb{X} \Rightarrow (e_{\ell'} \in \mathbb{X} \wedge \textit{val}(e_{\ell'}))$$

Here, we interpret $\textit{val}(e_{\ell'})$ as a boolean to denote $\textit{val}(e_{\ell'}) \neq 0$. The dependency requires that $r = 0$ when ℓ' is a conditional jump **beqz** r, ℓ targeting ℓ. This corresponds to the case where the jump is taken:

$$\textit{If } \ell' = \textbf{beqz } r, \ell, \textit{ then } e_\ell \in \mathbb{X} \Rightarrow (e_{\ell'} \in \mathbb{X} \wedge \neg \textit{val}(e_{\ell'}))$$

The predicate PRED (see §II-A) captures the static predecessors of an instruction. In a given execution however, each event has a unique predecessor. The set $\mathbb{X}$ must satisfy this. Combining the four cases above, a given instance of an instruction ℓ is executed if we have that

$$e_\ell \in \mathbb{X} \Rightarrow \bigvee_{\ell' \in \text{PRED}(\ell)} \text{CFD}(\ell, \ell') \quad (1)$$

where the CFD (*control flow dependency*) predicate is defined as

$$\text{CFD}(\ell, \ell') := \begin{cases} e_{\ell'} \in \mathbb{X} & \text{if } \ell' + 1 = \ell \wedge \ell' \neq \textbf{beqz } r, \ell'' \\ e_{\ell'} \in \mathbb{X} & \text{if } \ell' = \textbf{jmp } \ell \\ e_{\ell'} \in \mathbb{X} \wedge \textit{val}(e_{\ell'}) & \text{if } \ell' + 1 = \ell \wedge \ell' = \textbf{beqz } r, \ell'' \\ e_{\ell'} \in \mathbb{X} \wedge \neg\textit{val}(e_{\ell'}) & \text{if } \ell' = \textbf{beqz } r, \ell \end{cases} \quad (2)$$

Note that (2) does not need to consider the case where $\ell' + 1 = \ell \wedge \ell' = \textbf{jmp } \ell''$ because such instructions are not part of PRED(ℓ).

As we mentioned before, the program in Fig. 2 cannot access out-of-bounds memory because, before accessing `A`, `idx` is compared to the size of the array. To confirm this claim, let us analyze the two possible control flow paths of this program. If `idx` < `A.size`, the body of the if statement is executed. Following (1), this corresponds to executing every single instruction, i.e. $\mathbb{X} = \{e_1, \ldots, e_7\}$. For this execution, (2) requires that $\textit{val}(e_2) \neq 0$, otherwise the jump would have been taken and e_3–$e_6 \not\in \mathbb{X}$. As $\textit{val}(e_2) \neq 0$, we have $r_1 <$ `A.size` and `A` $+ r_1$ is in bounds. This means the dashed rf relation is not possible because $e_s : \textbf{sec}$ and e_4 access different addresses ($(e_s, e_4) \not\in \mathsf{loc}$) and the reads-from relation requires $\mathsf{rf} \subseteq \mathsf{loc}$. The second path has $\textit{val}(e_2) = 0$. In this execution the jump is taken and the control flow is redirected from 3 to 7, i.e. $\mathbb{X} = \{e_1, e_2, e_3, e_7\}$. Since only e_1 is a read event and the addresses of `idx` and **sec** are different, no **load** can read from out-of-memory. We conclude this execution does not read from **sec** either and the whole program is safe.

B. Speculative Control Flow

Modern processors implement branch speculation. Suppose a conditional jump is fetched but the value of its condition is not yet known. Instead of stalling, the processor makes a prediction on which branch will be taken and continues executing speculatively. Our semantics needs to consider the effects of the branch predictor and speculative execution. Instructions following a correct prediction are eventually committed; mispredictions lead to transient executions that are eventually rolled back.

Let $\mathbb{C} \subseteq \mathbb{X}$ represent instructions that are eventually committed. This set captures the case where instructions are executed *(i)* under normal semantics or, *(ii)* speculatively, but under a correct prediction. Transient instructions are represented by the set $\mathbb{T} \subseteq \mathbb{X}$. We consider as executed any instruction that is either committed or transiently executed and have that $\mathbb{X} = \mathbb{C} \uplus \mathbb{T}$.

For each event e_ℓ representing a conditional jump at ℓ we use proposition bpc_{e_ℓ} to represent that the predicted branch direction was correct or the value of the condition known. Following the always mispredict semantics [23] (which has been shown sufficient to obtain security guarantees w.r.t. all branch predictors), we leave bpc_{e_ℓ} unconstrained.

In §III-A we formalized the properties $\mathbb{X}$ must fulfill for traditional semantics with predicates (1) and (2). We extend this now to capture the effects of the branch predictor. The first two cases of (2) neither involve conditional jumps nor branch predictors and thus remain unchanged. The two cases below are possible when instructions execute along the correct control flow (i.e., correct prediction or known value of the condition) and at least one of them is a conditional jump. If ℓ' is not only the immediate predecessor of ℓ, but also a conditional jump targeting a different label, the

dependency requires that $r \neq 0$ (otherwise the jump would been taken) and *that the branch predictor is correct*:

$$\textit{If } \ell' + 1 = \ell \wedge \ell' = \textbf{beqz } r, \ell'', \\ \textit{then } e_\ell \in \mathbb{C} \Rightarrow (e_{\ell'} \in \mathbb{C} \wedge \textit{val}(e_{\ell'}) \wedge \mathsf{bpc}_{e_{\ell'}})$$

If ℓ' is a conditional jump targeting ℓ, the dependency requires that $r = 0$ and that *the branch direction is correctly predicted*. This corresponds to the case where the jump is taken:

$$\textit{If } \ell' = \textbf{beqz } r, \ell, \\ \textit{then } e_\ell \in \mathbb{C} \Rightarrow (e_{\ell'} \in \mathbb{C} \wedge \neg \textit{val}(e_{\ell'}) \wedge \mathsf{bpc}_{e_{\ell'}})$$

In the presence of branch predictors, committed instructions must follow the correct control flow:

$$e_\ell \in \mathbb{C} \Rightarrow \bigvee_{\ell' \in \text{PRED}(\ell)} \text{CFD}(\ell, \ell')$$

The control flow dependency definition from (2) needs to be updated with the two cases from above as follows:

$$\text{CFD}(\ell, \ell') := \begin{cases} e_{\ell'} \in \mathbb{C} & \text{if } \ell'+1 = \ell \wedge \ell' \neq \textbf{beqz } r, \ell'' \\ e_{\ell'} \in \mathbb{C} & \text{if } \ell' = \textbf{jmp } \ell \\ e_{\ell'} \in \mathbb{C} \wedge \textit{val}(e_{\ell'}) \wedge \mathsf{bpc}_{e_{\ell'}} & \text{if } \ell'+1 = \ell \wedge \ell' = \textbf{beqz } r, \ell'' \\ e_{\ell'} \in \mathbb{C} \wedge \neg \textit{val}(e_{\ell'}) \wedge \mathsf{bpc}_{e_{\ell'}} & \text{if } \ell' = \textbf{beqz } r, \ell \end{cases}$$

We capture dependency between two instructions with transient execution in a similar way. There are four possible cases; the first two cases are analogous to the traditional control flow case. The differences with the remaining two cases are the following: If ℓ' is a conditional jump immediately preceding ℓ but targeting a different label, then ℓ executing transiently implies that ℓ' was executed (*transiently or not*) and the jump should have been taken *but the branch predictor made the wrong guess* and thus altered the control flow:

$$\textit{If } \ell' + 1 = \ell \wedge \ell' = \textbf{beqz } r, \ell'', \\ \textit{then } e_\ell \in \mathbb{T} \Rightarrow (e_{\ell'} \in \mathbb{X} \wedge \neg \textit{val}(e_{\ell'}) \wedge \neg \mathsf{bpc}_{e_{\ell'}})$$

If ℓ is the target of the conditional jump ℓ', then ℓ executing transiently implies ℓ' was executed *under any semantics* and the jump was taken due to a *wrong prediction*:

$$\textit{If } \ell' = \textbf{beqz } r, \ell, \\ \textit{then } e_\ell \in \mathbb{T} \Rightarrow (e_{\ell'} \in \mathbb{X} \wedge \textit{val}(e_{\ell'}) \wedge \neg \mathsf{bpc}_{e_{\ell'}})$$

We capture *speculative control flow dependencies* with the constraint

$$e_\ell \in \mathbb{T} \Rightarrow \bigvee_{\ell' \in \text{PRED}(\ell)} \text{SCFD}(\ell, \ell') \tag{3}$$

where the speculative control flow dependency SCFD is defined as

$$\text{SCFD}(l, l') := \begin{cases} e_{\ell'} \in \mathbb{T} & \text{if } \ell'+1=\ell \wedge \ell' \neq \textbf{beqz } r, \ell'' \\ e_{\ell'} \in \mathbb{T} & \text{if } \ell' = \textbf{jmp } \ell \\ e_{\ell'} \in \mathbb{X} \wedge \neg \textit{val}(e_{\ell'}) \wedge \neg \mathsf{bpc}_{e_{\ell'}} & \text{if } \ell'+1=\ell \wedge \ell' = \textbf{beqz } r, \ell'' \\ e_{\ell'} \in \mathbb{X} \wedge \textit{val}(e_{\ell'}) \wedge \neg \mathsf{bpc}_{e_{\ell'}} & \text{if } \ell' = \textbf{beqz } r, \ell \end{cases}$$

Let us analyze the behavior of the program in Fig. 2 in the presence of speculative execution. The two execution paths from §III-A are still possible if bpc_{e_3} is true, i.e., if the branch predictor is correct. Additionally, the program has two executions where the branch predictor is wrong and some instructions execute transiently:

$$\neg \textit{val}(e_2), \neg \mathsf{bpc}_{e_3}, \mathbb{C} = \{e_1, e_2, e_3\}, \mathbb{T} = \{e_4, e_5, e_6, e_7\} \tag{4} \\ \textit{val}(e_2), \neg \mathsf{bpc}_{e_3}, \mathbb{C} = \{e_1, e_2, e_3\}, \mathbb{T} = \{e_7\}$$

Note that (4) has $\neg \textit{val}(e_2)$ and thus `idx` $\geq$ `A.size`. Because `idx` is not in bounds, it is possible that `A` $+ r_1 =$ **sec**. In this scenario, both e_s : **sec** and e_4 access the same address ($(e_s, e_4) \in \mathsf{loc}$) and thus the rf edge is possible, showing that the secret can be read and the program is vulnerable to SPECTRE-v1.

C. Speculation Window

Speculative execution attacks are only effective if there is a large enough speculation window for the transient execution phase. Thus, we need not only to consider the effects of the branch predictor, but also the speculation window it creates. Let sw be the size of the speculation window created by a branch prediction. The theoretical upper limit of instructions that can be transiently executed during this window is given by the size of the reorder buffer. Given a relation r, the definition below computes the pairs that are related by composing r with itself at most k times:

$$\mathsf{r}^{\leq k} := \begin{cases} \mathsf{r} & \text{if } k = 0 \\ \mathsf{r}; \mathsf{r}^{\leq k-1} & \text{otherwise} \end{cases}$$

We model that an instruction can only execute transiently during the speculation window using the following constraint to restrict the set $\mathbb{T}$: $\mathsf{po} \subseteq \mathsf{po}; ([\mathbb{T}]; \mathsf{po})^{\leq \mathsf{sw}-1}$. The constraint imposes that the number of po-consecutive transiently executed events is smaller than sw. Note that $\mathbb{T}$ can still have more than sw elements if several mispredictions occur and the corresponding transient events are not po-consecutive. The constraint also directly allows *nested speculations* within the same window, which is challenging to achieve with operational semantics.

D. Mitigating SPECTRE-v1

Serializing instructions can be used to stop the speculation and mitigate SPECTRE-v1. We model this by enforcing that fences cannot be executed transiently:

$$\bigwedge_{\ell : \textbf{fence}} e_\ell \not\in \mathbb{T} \tag{5}$$

Together, constraints (3) and (5) imply that instructions following a **fence** cannot execute transiently unless a new speculation is started by another conditional jump. For example, adding a **fence** after instruction 3 in Fig. 2 forbids the execution in (4) even in the presence of speculation. This shows that our semantics does not only allow to detect SPECTRE-v1 attacks, but also to prove that common mitigations work.

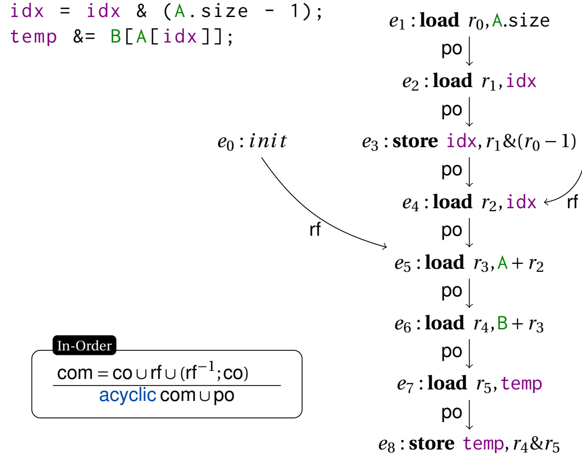


Fig. 4. Safe array access due to index masking.

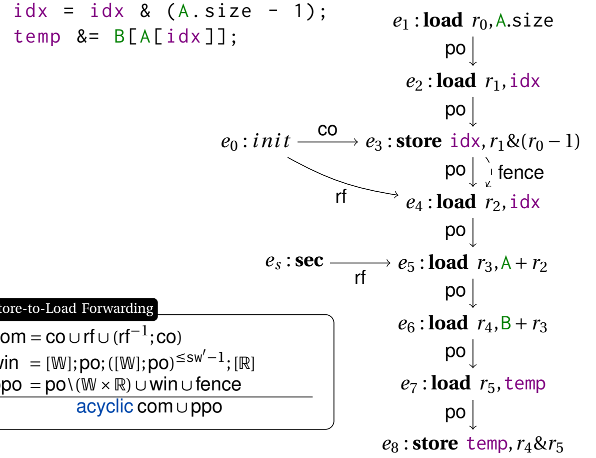


Fig. 5. SPECTRE-v4 – impossible under *in-order* semantics, but possible under *store-to-load forwarding* semantics.

IV. Instruction Reordering

In this section, we show how to model the logical reordering of instructions (e.g., due to aliasing or value forwarding), the reason underlying SPECTRE-v4. We begin with explaining in-order execution as a baseline (§IV-A) and then present our model for store-to-load forwarding (§IV-B) and possible defenses (§IV-C). Next, we introduce our model for predictive store forwarding (§IV-D). Finally, we show how CAT naturally handles the interaction between speculation, concurrency, and weak memory models (§IV-E) due to the composability of axiomatic models (§IV-F).

A. In-order Execution

In the simplest model of execution, the processor fetches an instruction, stalls until its operands are available, and finally executes. This model of computation, called *in-order execution*, is straightforward to understand and analyze: instructions are executed one-by-one following the program order. Consider the program in Fig. 4. Since `idx` is masked using `A.size`, accessing array `A` is safe. This safe behavior is captured by the candidate execution where event e_5 reads from the initial values of the array and there are no out-of-bounds accesses. Since this program has only one control flow path (regardless of whether the processor supports speculative control flow or not), this is the only possible execution of the program under in-order semantics.

Our CAT model of the in-order semantics is shown in Fig. 4. As usual, rf edges only relate write-read pairs accessing the same address. The model defines a causal dependency $\mathsf{com} \cup \mathsf{po}$ and forces it to be acyclic (otherwise instructions could not be scheduled to satisfy their dependencies). From $\mathsf{rf}^{-1};\mathsf{co}$ we have that if a read r gets its value from a write w ($\mathsf{rf}^{-1}(r,w)$), then any other write w' coming after w in the coherence order ($\mathsf{co}(w,w')$) must come after r ($\mathsf{rf}^{-1};\mathsf{co}(r,w')$), otherwise r would get its value from w' instead. Setting com to be acyclic imposes that there is a single view of how instructions hit memory. Since instructions are executed in order, po must be part of the causal dependency. All these imply event e_4 must read from the last **store** to that address (here e_3), otherwise there would be a cycle. Having $\mathsf{rf}(e_3, e_4)$ enforces that r_2 gets the masked value of `idx`. Because of the masking, we have $r_2 <$ `A.size`. Finally, since the address of instruction e_5 is in-bounds, e_5 can only read the initial value of the array.

Instruction e_3 cannot directly read the initial value of `idx` (represented here by the $e_0 : \mathit{init}$ event). Reading the initial (not masked) value of `idx`, it would be possible for e_5 to access the secret. The combination of rf and co given in Fig. 5 forms the candidate execution representing this unsafe scenario, but it forms a cycle of dependencies $e_4 \xrightarrow{\mathsf{rf}^{-1}} e_0 \xrightarrow{\mathsf{co}} e_3 \xrightarrow{\mathsf{po}} e_4$. Since the CAT model forbids cycles involving those relations, this unsafe scenario is not possible and we can conclude the program is safe under in-order semantics.

B. Store-to-load Forwarding

Modern processors use a combination of speculative and out-of-order execution optimizations, so some of the guarantees from in-order execution do not hold. Although not observable at the architectural level, the microarchitecture allows more executions, which might leave traces in the microarchitectural state. In Fig. 5, if the address used by e_3 is not yet known, the processor might predict that e_3 and e_4 will not alias and speculatively execute e_4 before e_3. This is the basis of SPECTRE-STL (one instance of SPECTRE-v4). Our CAT model for *store-to-load forwarding* still imposes a single view of how instructions hit memory (com is acyclic), but uses a weaker notion of preserved program order (ppo) than in-order semantics (which uses the whole po) and thus allows the SPECTRE-STL behavior. There are three possible scenarios where the order of events is preserved by ppo:

- Events are in program order, but they are not a write-read pair.

- They are "far away" in the program order in such a way that events in between fill the store buffer.
- The corresponding instructions are separated by a **fence** and thus the events are related by a fence edge.

The relation $\mathsf{po} \setminus (\mathbb{W} \times \mathbb{R})$ tells us that some read events can be speculatively executed before a po-previous write. This is allowed, e.g., if the processor knows or predicts the addresses do not alias. Apart from write-read pairs, our CAT model preserves every other pair of events in program order since they are irrelevant to model store-to-load forwarding.

However, not all write-read pairs can be reordered. Once a write event has been committed (it is no longer in the store buffer), its address is known, alias speculation is not possible, and the store cannot be passed over. The size of the store buffer defines a speculation window that can be modeled in CAT by tracking the number of write events between the potentially reordered pair. To model the speculation window, we preserve the program order between a write-read pair if the number of write events between the pair fills the buffer, forcing to retire and commit the write event. We model this in CAT using the relation $\mathsf{win} = [\mathbb{W}];\mathsf{po};([\mathbb{W}];\mathsf{po})^{\leq \mathsf{sw}'-1};[\mathbb{R}]$ where sw' is the size of the store buffer, with $\mathsf{sw}' \geq 1$. Note that sw' can differ from the speculation window sw created by the branch predictor. For $\mathsf{r}^{\leq k}$ to be well defined, $k \geq 0$ must hold. Relations $[\mathbb{W}]$ and $[\mathbb{R}]$ are the identity relation restricted to write and read events respectively, i.e., $[\mathbb{W}] = \{(e,e) \mid e \in \mathbb{W}\}$ and $[\mathbb{R}] = \{(e,e) \mid e \in \mathbb{R}\}$. Using $[\mathbb{W}]$ on the left of the composition guarantees that the first event in every pair is a write (those not to be reordered). Analogously, using $[\mathbb{R}]$ on the right forces the second event to be a read. The inner part $\mathsf{po};([\mathbb{W}];\mathsf{po})^{\leq \mathsf{sw}'-1}$ represents the writes between the potentially reordered pair that fit in the buffer. For example, if the size of the store buffer is two, win reduces to $[\mathbb{W}];\mathsf{po};[\mathbb{W}];\mathsf{po};[\mathbb{W}];\mathsf{po};[\mathbb{R}]$, which relates all write-read pairs in po for which there are at most two write events in between.

Since **fence** instructions stop speculation, the ppo also preserves the order of events coming from instructions for which there is a fence in between. If we add a **fence** between instructions 3 and 4 in the μASM program for Fig. 5, this results in the dashed fence edge being part of the graph.

The preserved program order ppo is defined as the union of the three relations described above. Let us see how this semantics affect the possible behavior of the program. The pair (e_3, e_4) is not part of ppo: it is a write-read pair from po which can be reordered because the buffer might not be full and there are no fences in between (even if the buffer was full, the processor can decide to commit some of the buffered events, making room for e_3). Since it is possible to reorder e_3 and e_4 and because the initial value of idx is controlled by the attacker, the value loaded to r_2 can be bigger than A.size. The address of e_5 depends on r_2, allowing to potentially read out of bounds and access the secret. The candidate execution shown in Fig. 5 represents this scenario and it is consistent according to our CAT model, i.e., our semantics models SPECTRE-STL.

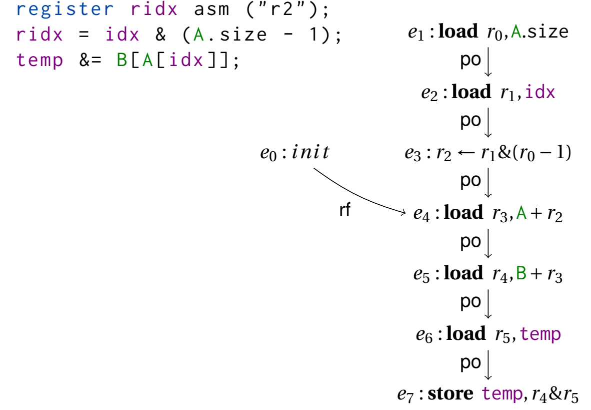


Fig. 6. SPECTRE-v4 – store replaced by register assignment (safe).

C. *Mitigating* SPECTRE-STL

At the software level, there are at least two possible ways to mitigate SPECTRE-STL. One alternative is to add a **fence** instruction resulting in the dashed fence edge in Fig. 5. Since pairs of events related by fence are part of the ppo order, adding this edge results in the cycle $e_4 \xrightarrow{\mathsf{rf}^{-1}} e_0 \xrightarrow{\mathsf{co}} e_3 \xrightarrow{\mathsf{fence}} e_4$, which is not allowed by the CAT model. This shows that the mitigation makes the program safe under our semantics. Another alternative is given in Fig. 6. Here, the compiler is instructed to store the masked value in a register, resulting in a different assembly program. Since the attack relies on bypassing a store to memory, but the masking is now done by a local computation using only registers, our semantics guarantees that e_4 will get the masked value in r_2 and thus the access to A will be in bounds.

D. *Predictive Store Forwarding*

The SPECTRE variant from §IV-B exploits the fact that processors may predict that the addresses of two instructions do not alias. Some processors implement the opposite, allowing them to predict that two addresses alias even if they finally do not match. This type of prediction was used as the basis of a theoretical new version of SPECTRE-v4 [9], [22] which we denote SPECTRE-PSF. AMD recently confirmed that its Zen 3 process is vulnerable to this attack [19]. We present a CAT model allowing this kind of prediction and modeling such an attack.

The unsafe scenario is shown in Fig. 7. Note that this program accesses array C using an attacker-controlled index idx. Even if the index is compared with the size of C, the program is vulnerable to SPECTRE-v1 as we have seen in §III-B. Unfortunately, even if we use the mitigation from §III-D to guarantee that e_4 is not transiently executed, the program is still vulnerable in the presence of alias speculation. Consider the scenario where idx is 1. Despite having different offsets, the processor can speculate that e_3 and e_4 alias. In this scenario, register r_2 gets the value 64 which is then multiplied by 1 (the value of idx) to compute the address of e_5. Since the size of A is smaller than 64,

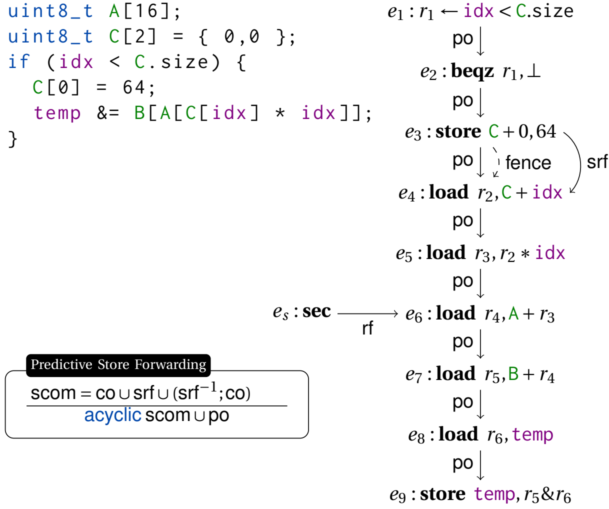


Fig. 7. SPECTRE-v4 – predictive store forwarding (unsafe).

instruction e_6 performs an out-of-bounds access allowing to read the secret.

Our CAT model for *predictive store forwarding* is given in Fig. 7. The formalization closely follows the in-order semantics with the exception that instead of using relation rf, we use a new relation that supports alias speculation. While events related by rf must access the same address, srf relaxes this and only imposes that the addresses of those two instructions must alias:

$$\mathsf{srf} \subseteq (\mathbb{W} \times \mathbb{R}) \cap \mathsf{alias}$$
$$\forall r \in \mathbb{R} : \exists! w \in \mathbb{W} : \mathsf{srf}(w, r)$$
$$\mathsf{srf}(w, r) \Rightarrow val(w) = val(r)$$

We leave the alias relation unconstrained to support arbitrary predictors, i.e., $\mathsf{alias} := \{(m_1, m_2) \mid m_1, m_2 \in \mathbb{M}\}$.

The candidate execution from Fig. 7 fulfills the assertion of our predictive store forwarding CAT model showing that the program is vulnerable to SPECTRE-PSF.

Since **fence** instructions stop speculation, we must enforce that if two events are related by srf and there is a **fence** between the corresponding instructions, then the events must access the same address, i.e., $\mathsf{srf} \cap \mathsf{fence} \subseteq \mathsf{loc}$. Note that if the dashed fence edge in Fig. 7 is part of the graph, the edge $e_3 \xrightarrow{\mathsf{srf}} e_4$ would not be allowed because $\mathtt{C}{+}0 \neq \mathtt{C}{+}\mathtt{idx}$ (in this scenario $\mathtt{idx} = 1$), forcing e_4 to read its value from $e_0 : init$ and forbidding e_6 to access **sec**.

E. Concurrency and Weak Memory Models

While all the attacks we described previously have an inherent notion of concurrency (the attacker resides in a sandbox different from the victim), so far we only considered single-threaded victims. Besides branch misprediction, the Intel Optimization Reference Manual [25] lists *machine clear* (i.e., data misprediction) as a category of *bad speculation*. One possible cause of such machine clears is memory ordering in the presence of concurrency. Intel and AMD processors implement the *Total-Store-Order* memory model in which all cores see operations in program order except in one case: a **store** followed by a **load** on a different address may be reordered. This is because cores use local buffers to hide the latency of **store** operations.

The CAT model for TSO [36] is given in Fig. 8. The assertion acyclic $\mathsf{com} \cup (\mathsf{po} \cap \mathsf{loc})$ enforces local consistency within the same core. Due to the use of buffers, only the *external reads-from relation* is part of the global view. Here, rfe is the subset of rf such that the events from the pair belong to different cores. Relation po-tso keeps all pairs in po except write-read pairs (which from a global point of view might be reordered), unless there is a **fence** instruction between them.

Consider the concurrent program in Fig. 8 running on the TSO memory model. Here we assume μASM is extended with threads running on different cores: a program is composed by a set of threads and a thread is a sequence of instructions. The only impact this change has in the CAT semantics is that we must guarantee po only relates events within the same thread. The program shows a *message passing* pattern where if `thread_1` observes $\mathtt{x} = 1$, then it should also observe $\mathtt{y} = 1$. Since the TSO semantics guarantees $r_0 * (r_0 - r_1)$ is always 0, the access to A is safe. The CAT model captures this since it forbids the (unsafe) candidate execution where $r_0 = 1 \land r_1 = 0$ due to the cycle $e_1 \xrightarrow{\mathsf{po}} e_2 \xrightarrow{\mathsf{rf}^{-1};\mathsf{co}} e_6 \xrightarrow{\mathsf{po}} e_7 \xrightarrow{\mathsf{rfe}} e_1$.

Ragab et al. [35] noticed that if e_1 is a slow **load** (due to a cache miss) and e_2 a fast one (cache hit), event e_2 can be transiently executed before e_1. Suppose that while the **load** from x is pending, both stores from `thread_2` are executed as represented by the candidate execution in Fig. 8. If this is the case, the coherence controller notifies that the values of x and y have changed, leading to a *memory ordering machine clear* which guarantees that the behavior is not observable at the architectural level. Abusing this kind of behavior is non-trivial due to the strict synchronization requirements, however there exists a speculation window which could be use to mount an attack.

We can use CAT to capture this speculation window by replacing $(\mathbb{R} \times \mathbb{M})$ with $(\mathbb{R} \times \mathbb{W})$ and adding addr to the union in the definition of po-tso. These changes allow some read-read pairs to be reordered. The first assertion still guarantees the reordering is not possible if both reads access the same address; and by adding addr to po-tso, we forbid to revert the order when the address of the second read depends on the value loaded by the first one. Since e_1, e_2 neither access the same address nor have an address dependency, with this modification, $(e_1, e_2) \notin \mathsf{po\text{-}tso}$ and the candidate execution becomes consistent. It is then possible that $r_0 * (r_0 - r_1) \neq 0$ and thus, e_3 can read the secret if $\mathtt{A} + 1 = \mathbf{sec}$.

F. Composability of Axiomatic Models

One of the main advantages of axiomatic models is that they are easily composable. Any of the control flow semantics in §III can be combined with any of the CAT models in this

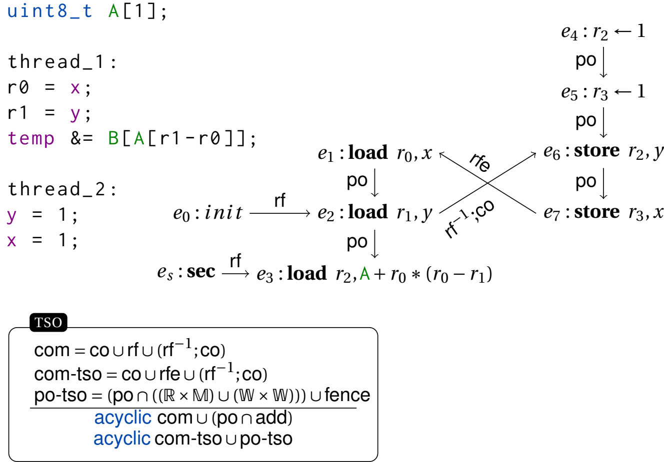


Fig. 8. Transient execution due to invalid memory ordering.

section. For instance, if an attack requires to mistrain both the branch predictor and the memory alias predictor, this would be detected by combining the SCFD definition in §III-B and the Store-to-Load Forwarding CAT model in Fig. 5.

V. Implementation

We use Bounded Model Checking (BMC) parametrized by our axiomatic semantics defined as CAT models for verifying software isolation. All predicates from Sections III and IV can be encoded using first order logic over the domain of booleans and integers. In fact, a compact BMC encoding to test reachability using the CAT language with traditional control flow was developed in [17], [21]. We implemented Kaibyo[1], a prototype tool to test software isolation, as an extension of Dartagnan[2]. Apart from the definition of the new CAT models, we added an x86 parser and the speculative control flow encoding from §III-B, we modified the property being checked (from reachability to software isolation), and we implemented the new srf relation.

Kaibyo takes as inputs a program written in x86 assembly, a CAT model, an unrolling bound k and an address **sec**. It generates a formula which is satisfiable if and only if there is a consistent execution (according to the CAT model and where loops where unrolled up to k iterations) where some read event reads from e_s : **sec**. As any other BMC technique, we only analyze bounded (finite) executions of programs containing loops. Thus, Kaibyo only provides security guarantees up to the given bound. Nevertheless, any transient execution gadget found by the tool can be exploited under the conditions laid out in §II-E.

VI. Evaluation

We use Kaibyo to evaluate our approach and answer the following research questions:

RQ1: *Do our semantics cover known attacks?*

RQ2: *Do our semantics prove effectiveness of proposed countermeasures?*

RQ3: *What effort is required to support new semantics in the analysis?*

RQ4: *How complex are the generated formulae for state-of-the-art SMT solvers?*

In the following, we first discuss our experimental setup (§VI-A) and then evaluate the ability of our models to capture Spectre variants (§VI-B), the flexibility of the tool to support different semantics (§VI-C), and the performance of SMT solvers on the generated formulae (§VI-D).

A. Experimental Setup

We compare the results of Kaibyo against Spectector [23] and Binsec [16] using the following benchmarks

(PHT) Fifteen benchmarks by Kocher [27] exploiting branch prediction.

(STL) Thirteen benchmarks from the Binsec repository [14] exploiting store-to-load forwarding.

(PSF) The example program from Fig. 7 (adapted from [19]) exploiting predictive store forwarding.

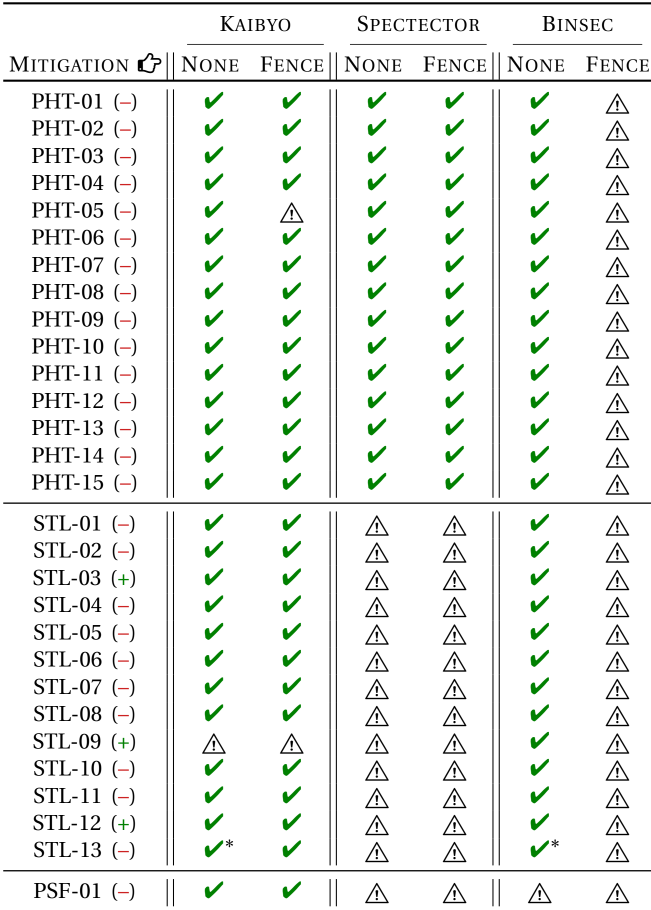
All benchmarks are written in C and compiled using GCC 8.3.0. The results of our evaluation are given in Fig. 9. The expected result w.r.t software isolation when no mitigation is used (None column) is either Safe (+) or Unsafe (-). For each benchmark there is a variant (Fence column) using a **fence** instruction to stop branch or alias speculation; all such variants are Safe. A ✔ entry means the tool returns the corrected expected result. We show ⚠ if the tool cannot analyze the program and detail the reasons below.

B. Precision of the CAT Models

All three tools support branch prediction and correctly report all PHT benchmarks as vulnerable to Spectre-v1. Both Kaibyo and Spectector also prove that adding fences after each conditional jump stops speculation. Binsec has no support for **fence** instructions and thus it cannot analyze the benchmarks using the mitigation. Kaibyo returns Safe for all benchmarks except PHT-05 which contains an input dependent loop. In the presence of input dependent loops, BMC techniques can find violations (see None column), but they cannot prove programs correct, because loops cannot be fully unrolled. While the same limitation applies to Spectector and Binsec (which are based on symbolic execution), they return Safe even if the analysis is incomplete. Kaibyo returns Unknown when it cannot find a violation up to the given bound; at the same time, it cannot prove the bound is large enough to explore all executions, hence the ⚠ entry in the table.

Spectector has no support for store-to-load forwarding and thus it cannot analyze any of the STL benchmarks. A stack overflow occurs when running Kaibyo to generate the formula for STL-09 which requires a large (>200) unrolling bound. Kaibyo proves that adding fences between reordered pairs makes the programs Safe. Binsec cannot analyze the mitigated benchmarks due to not supporting fences. Both

[1]https://github.com/unibw-patch/Kaibyo
[2]https://github.com/hernanponcedeleon/Dat3M



	KAIBYO		SPECTECTOR		BINSEC	
MITIGATION ☞	NONE	FENCE	NONE	FENCE	NONE	FENCE
PHT-01 (–)	✔	✔	✔	✔	✔	⚠
PHT-02 (–)	✔	✔	✔	✔	✔	⚠
PHT-03 (–)	✔	✔	✔	✔	✔	⚠
PHT-04 (–)	✔	✔	✔	✔	✔	⚠
PHT-05 (–)	✔	⚠	✔	✔	✔	⚠
PHT-06 (–)	✔	✔	✔	✔	✔	⚠
PHT-07 (–)	✔	✔	✔	✔	✔	⚠
PHT-08 (–)	✔	✔	✔	✔	✔	⚠
PHT-09 (–)	✔	✔	✔	✔	✔	⚠
PHT-10 (–)	✔	✔	✔	✔	✔	⚠
PHT-11 (–)	✔	✔	✔	✔	✔	⚠
PHT-12 (–)	✔	✔	✔	✔	✔	⚠
PHT-13 (–)	✔	✔	✔	✔	✔	⚠
PHT-14 (–)	✔	✔	✔	✔	✔	⚠
PHT-15 (–)	✔	✔	✔	✔	✔	⚠
STL-01 (–)	✔	✔	⚠	⚠	✔	⚠
STL-02 (–)	✔	✔	⚠	⚠	✔	⚠
STL-03 (+)	✔	✔	⚠	⚠	✔	⚠
STL-04 (–)	✔	✔	⚠	⚠	✔	⚠
STL-05 (–)	✔	✔	⚠	⚠	✔	⚠
STL-06 (–)	✔	✔	⚠	⚠	✔	⚠
STL-07 (–)	✔	✔	⚠	⚠	✔	⚠
STL-08 (–)	✔	✔	⚠	⚠	✔	⚠
STL-09 (+)	⚠	⚠	⚠	⚠	✔	⚠
STL-10 (–)	✔	✔	⚠	⚠	✔	⚠
STL-11 (–)	✔	✔	⚠	⚠	✔	⚠
STL-12 (+)	✔	✔	⚠	⚠	✔	⚠
STL-13 (–)	✔*	✔	⚠	⚠	✔*	⚠
PSF-01 (–)	✔	✔	⚠	⚠	⚠	⚠

Fig. 9. Evaluation with different SPECTRE behaviors. Expected result when no mitigation is used is SAFE (+) or UNSAFE (–). The result of the tool is correct ✔ or it cannot analyze the program ⚠.

KAIBYO and BINSEC prove that instructing the compiler to use registers instead of **store** instructions (as in Fig. 6) makes STL-03 and STL-12 SAFE. Both tools agree in all remaining results except for STL-13. The reason is that, as we describe below, the tools do not consider the same threat model.

The original C code of STL-13 and the corresponding x86 assembly are given in Fig. 10. After masking the index and accessing `A[ridx]`, function `load_value` moves the loaded value to the stack and back before returning. If the instruction `mov eax, [esp+15]` happens before the previous **store** (i.e. they are reordered), then it can read from uninitialized memory. If the value from the initialized memory is bigger than the size of `B`, then the access in `case_13` to `B[eax]` is out of bounds and can access the secret. This access brings the secret to the cache and this, according to our threat model from §II-E, might have security consequences; thus KAIBYO reports it as UNSAFE. However, even if the secret is loaded into `edx`, it does not influence the control flow (e.g., conditional branches) or memory addresses (e.g., offsets into arrays). Therefore the program does not violate speculative constant time, the property BINSEC verifies.

KAIBYO is able to detect that PSF-01 from Fig. 7 is vulnerable to SPECTRE-v1. It also reports that even if adding a **fence** after the conditional jumps stops the control flow speculation, the program remains vulnerable due to predictive store forwarding. Finally, by inserting a unique **fence** between instructions 3 and 4, to stop both branch and alias speculation, it proves the program SAFE. To the best of our knowledge, KAIBYO is the only tool having support for predictive store forwarding.

The results above answer **RQ1** and **RQ2** by showing that CAT can be used to define semantics covering SPECTRE-v1 and v4 and proving that common mitigations work.

C. Flexibility of the Analysis

To answer **RQ3**, we report on our development efforts to support all vulnerabilities discussed in this paper. KAIBYO is based on the tool DARTAGNAN [18] which implements the traditional control flow encoding from §III-A and supports the core of the CAT language from Fig. 3. DARTAGNAN verifies Boogie code [30] and most of our development effort was spent in parsing x86 assembly and encoding the stack. Extending the tool to support speculative control flow required around 100 lines of Java code. Adding support to store-to-load forwarding and memory order machine clear only required to develop the CAT models from Fig. 5 and the variation of TSO described in §IV-E. The tool then automatically detected both UNSAFE behaviors. Detecting the attack based on predicative store forwarding required implementing the axioms of the new relation srf. The SMT encoding of srf enforces that both events should alias instead of access the same address, as in rf. To implement the alias relation we used an unconstrained boolean variable for every pair of memory events.

D. SMT Performance

To answer **RQ4**, we compare the performance of four different SMT solvers (Z3, CVC4, YICES2 and MATHSAT5) on the generated formulae from Fig. 9. The results are given in Fig. 11. Times are shown in seconds using a logarithmic scale. We used a 90 min timeout. For entries ⚠ in the table, we treat the result as a timeout. Since SMT solvers tend to perform differently for SAT and UNSAT instances, we divided the results in two. The top of the figure shows the results for the original benchmarks (no mitigation). All formulae are SAT except STL-03 and STL-12 (no formula is generated for STL-09 due to a stack overflow). The bottom of the figure present the results for the benchmarks in the FENCE column which are all UNSAT instances.

For the SAT instances YICES2 generally wins, with Z3 and MATHSAT5 being competitive. CVC4 shows always the worst performance for these SAT formulae, sometimes three orders of magnitude slower than the other solvers. However, CVC4 shows the overall best performance for the UNSAT instances. In particular, for benchmarks STL and PSF where reasoning about instruction reordering is needed, most solvers timeout.

```c
uint8_t load_value(uint32_t idx) {
  register uint32_t ridx asm ("edx");
  ridx = idx & (A.size - 1);
  uint8_t to_leak = A[ridx];
  return to_leak;
}

void case_13(uint32_t idx) {
  register uint8_t to_leak asm ("edx");
  to_leak = load_value(idx);
  temp &= B[to_leak * 512];
}
```

```
case_13:
        push    [esp+4]
        call    load_value
        add     esp, 4
        mov     edx, eax
        mov     eax, edx
        movzx   eax, al
        movzx   edx, B[eax]
        movzx   eax, temp
        and     eax, edx
        mov     temp, al
        ret
```

```
load_value:
        sub     esp, 16
        mov     eax, A.size
        sub     eax, 1
        and     eax, [esp+20]
        mov     edx, eax
        mov     eax, edx
        movzx   eax, A[eax]
        mov     [esp+15], al
        movzx   eax, [esp+15]
        add     esp, 16
        ret
```

Fig. 10. A variant of SPECTRE-v4 from [14] written in C (left) and compiled to x86 with GCC 8.3.0 (right).

CVC4 is able to solve all such benchmarks except STL-01 for which MATHSAT5 has the best performance.

The bottom of the figure shows a clear difference in the search space (which the solver needs to completely explore since these instances are UNSAT) of PHT w.r.t STL and PSF. While there are four possible outcomes for branch prediction (following the true or false branch combined with correctly or incorrectly predicted), alias misprediction allows a **load** to draw data from *any* prior **store**, making the search space much bigger. This suggests that it is beneficial for tools to have a portfolio of SMT backends; this is possible using libraries like JavaSMT, for instance [5], [26].

The solving times of SPECTECTOR and BINSEC for each of the benchmarks in Fig. 9 are between 0.1 and 10 seconds, meaning they are one or two orders of magnitude faster than KAIBYO. This is not surprising since those tools are specialized for concrete models and thus less flexible. Also, despite branch misprediction and instruction reordering, the benchmarks have few execution, favoring symbolic execution approaches (which use a *simple SMT query for each execution*) over BMC ones (which use a *complex SMT query for all executions*).

VII. DISCUSSION

KAIBYO analyzes programs w.r.t software isolation. Currently, it cannot analyze non-interference-style properties, and our evaluation shows the consequences of this. KAIBYO considers a program unsafe if it accesses the secret, even if this does not violate, e.g., constant time, because it neither influences the control flow nor the memory addresses. As we discussed in §II-F, given a notion of observation for axiomatic models, our semantics could be used to also verify non-interference-style properties. Since this requires reasoning about pairs of executions, doing so would affect the performance of our implementation, however.

KAIBYO demonstrates that it is possible to translate axiomatic semantics into a concrete analysis tool, but it is a proof-of-concept implementation with clear limits to scalability. Improving performance of tools based on axiomatic semantics is an active area of research; the weak memory model community recently made progress on scalable non-BMC tools based on axiomatic semantics [29].

VIII. RELATED WORK

This section describes the work that has been done on security foundations since the disclosure of SPECTRE. Formal microarchitectural models are capable of representing out-of-order and speculative behavior. Side channels are modeled by different notions of observations which over-approximate the attacker capabilities and abstract from the memory subsystem and cache. Security guarantees are formalized as properties comparing such observations under two different semantics: a reference execution model (generally in-order execution) and a target execution model (e.g., speculative and/or out-of-order executions).

Currently, the target execution model of Guarnieri et al. [23] only covers branch speculation and captures SPECTRE-v1 behaviors by the notion of *speculative non-interference*. SPECTECTOR is a symbolic execution tool for testing speculative non-interference. To extend its semantics to other SPECTRE versions, one would have to adapt the notion of microarchitectural state. Cauligi et al. extend constant time in the presence of speculation, leading to the new notion of *speculative constant time* [9]. Even though their semantics models most of the behaviors underlying SPECTRE (including indirect jumps, return stack buffers and predictive store forwarding), their analysis tool PITCHFORK does not support any of these. Doing so *"would require to generate a prohibitively large number of possible schedules"*. Guanciale et al. [22] define an out-of-order execution model using microarchitectural instructions rather than the ISA. They show traditional constant time (which is defined at the ISA level) is not secure enough *even in the absence of speculation* and propose a constant time property for the out-or-order execution model. The proposed semantics is very expressive and even allows to mispredict arbitrary values. Unfortunately, there is no tool based on such semantics.

The approaches above can be formalized as *hardware-software contracts* specifying which program executions an attacker can differentiate [24]. On the one side, contracts provide security foundations to build tools that help software developers write microarchitecturally secure programs. On the other side, for hardware developers, contracts are specifications that describe allowed microarchitectural effects

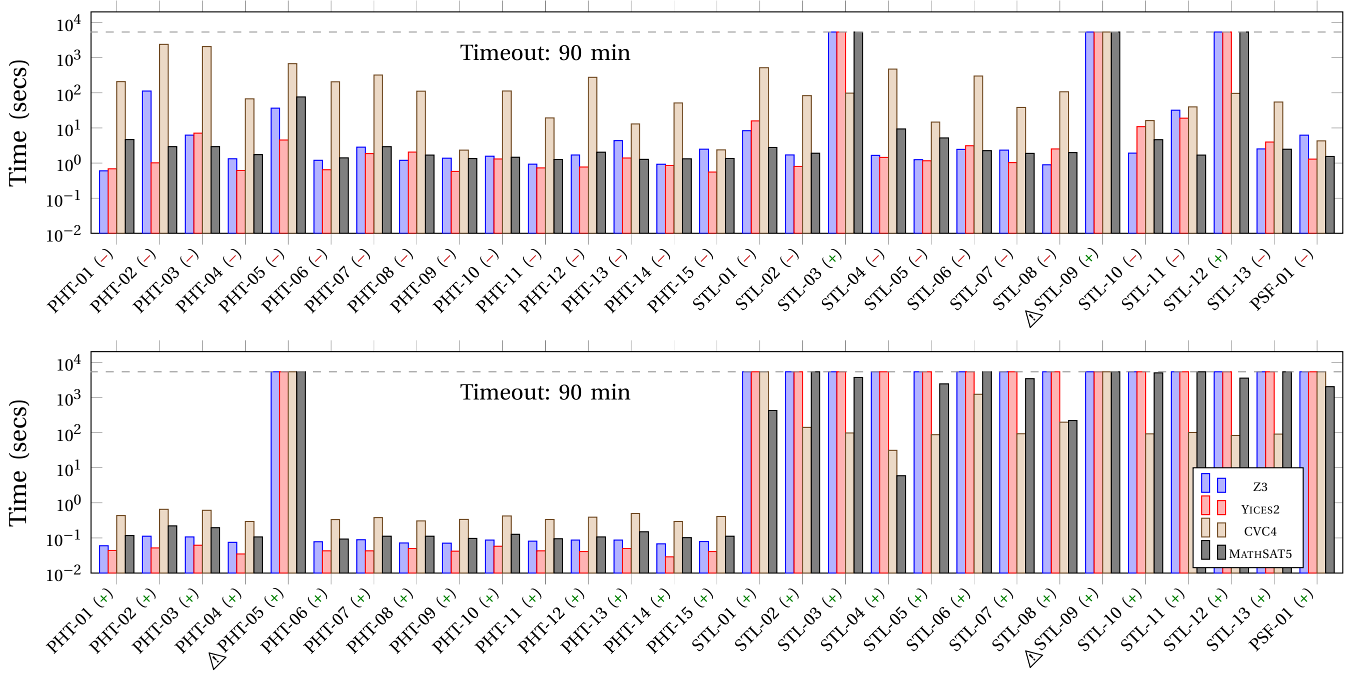


Fig. 11. Benchmarks from columns NONE (top) and FENCE (bottom). Labels show if the formula is SAT (–) or UNSAT (+). KAIBYO could not fully unroll the program (PHT-05) or generate the formula (STL-09) for entries marked with ⚠.

without enforcing a specific implementation. Contracts are 2-hypersafety properties [11] and thus require appropriate techniques to efficiently model pairs of traces. RelSE is a promising approach to extend symbolic execution for analyzing security properties of two execution traces [15]. In fact, an extension to RelSE has recently been proposed to test speculative constant time [16].

We were not the first to provide insights about the relation between SPECTRE attacks and the weak memory models which characterize modern hardware. Disselkoen et al. use pomsets (where edges also represent dependencies) to model executions [20]. The novel aspect of this model is that events have preconditions which can captured failed branch predictions. Their semantics cover SPECTRE-v1 and models out-of-order executions, but it is not clear if they capture SPECTRE-v4. There is also no tool based on this semantics. CHECKMATE [37] uses graphs to capture the subtle orderings of hardware execution events when programs run on a microarchitecture. Their "micro-architecturally happens-before" notion is based on po, rf and co, but their executions are less abstract than ours: a single instruction is represented by many events modeling fetching, execution, commit and completion. They even explicitly model when a value is brought to and flushed from the L1 cache. One of the novelties of their work is the use of graphs to represent exploit patterns like FLUSH+RELOAD and PRIME+PROBE. It remains an open question if CAT can be use to capture such patterns. Closest to our work are Colvin and Winter [13]. They integrate speculative control flow within a more general verification framework using `IMP-ro`, a language for reasoning about weak memory models. While `IMP-ro` can handle weak memory models such as TSO, POWER and ARM [12], their paper focuses on sequential consistency instead of exploring the interaction between speculation and weak memory models as we did in §IV-E. Finally, CAT has seen far wider adoption based on the large body of literature [3], [4], [6], [33], [36] and its use in industry [2].

IX. CONCLUSION

We studied the use of axiomatically defined semantics in the presence of speculative and out-of-order execution, a domain that, contrary to its operational counterpart, had not been sufficiently explored. We showed that CAT, a domain specific language initially developed to clarify the concurrency semantics of weak memory models, can also be used to analyze the consequences of microarchitectural optimizations. Challenging aspects of speculative execution, such as speculation nesting, are naturally modeled using our framework. Although our axiomatic BMC-based prototype is slower than its operational symbolic execution-based counterparts, the pluggable abstract semantics allow for a cleaner and more general implementation that can easily accommodate new models, as demonstrated by the ability to quickly add support for new attacks such as that based on memory ordering machine clears [35].

ACKNOWLEDGMENTS

We would like to thank our anonymous shepherd and the anonymous reviewers for their comments and feedback. We would also like to thank Sébastien Bardin, Lesly-Ann Daniel, and Marco Guarnieri for valuable discussions on this work.

References

[1] Jade Alglave, Patrick Cousot, and Luc Maranget. Syntax and semantics of the weak consistency model specification language CAT. *CoRR*, abs/1608.07531, 2016.

[2] Jade Alglave, Will Deacon, Richard Grisenthwaite, Antoine Hacquard, and Luc Maranget. Armed cats: Formal concurrency modelling at arm. *ACM Trans. Program. Lang. Syst.*, 43(2):8:1–8:54, 2021.

[3] Jade Alglave, Luc Maranget, Paul E. McKenney, Andrea Parri, and Alan S. Stern. Frightening small children and disconcerting grown-ups: Concurrency in the Linux kernel. In *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 405–418. ACM, 2018.

[4] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.*, 36(2):7:1–7:74, 2014.

[5] Daniel Baier, Dirk Beyer, and Karlheinz Friedberger. JavaSMT 3: Interacting with SMT solvers in Java. In *Proc. Int. Conf. Computer Aided Verification (CAV)*, volume 12760 of *LNCS*, pages 195–208. Springer, 2021.

[6] Mark Batty, Alastair F. Donaldson, and John Wickerson. Overhauling SC atomics in C11 and OpenCL. In *Proc. Symp. Principles of Programming Languages (POPL)*, pages 634–648. ACM, 2016.

[7] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. In *Proc. 28th USENIX Security Symposium (USENIX Security)*, pages 249–266, 2019.

[8] Sunjay Cauligi, Craig Disselkoen, Daniel Moghimi, Gilles Barthe, and Deian Stefan. SoK: Practical foundations for Spectre defenses. *CoRR*, abs/2105.05801v2, 2021.

[9] Sunjay Cauligi, Craig Disselkoen, Klaus von Gleissenthall, Dean M. Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. Constant-time foundations for the new spectre era. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, pages 913–926. ACM, 2020.

[10] Stephen Chong, Joshua D. Guttman, Anupam Datta, Andrew C. Myers, Benjamin C. Pierce, Patrick Schaumont, Tim Sherwood, and Nickolai Zeldovich. Report on the NSF workshop on formal methods for security. *CoRR*, abs/1608.00678, 2016.

[11] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.

[12] Robert J. Colvin and Graeme Smith. A wide-spectrum language for verification of programs on weak memory models. In *Proc. 22nd Int. Symp. on Formal Methods (FM)*, volume 10951 of *LNCS*, pages 240–257. Springer, 2018.

[13] Robert J. Colvin and Kirsten Winter. An abstract semantics of speculative execution for reasoning about security vulnerabilities. In *Formal Methods. FM 2019 Int. Workshops*, volume 12233 of *LNCS*, pages 323–341. Springer, 2019.

[14] Lesly-Ann Daniel. Binsec/haunted benchmark. https://github.com/binsec/haunted_bench, 2021. Last accessed: 10.12.2021.

[15] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. Binsec/Rel: Efficient relational symbolic execution for constant-time at binary-level. In *Proc. IEEE Symp. Security and Privacy (S&P)*, pages 1021–1038. IEEE, 2020.

[16] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. Hunting the haunter — Efficient relational symbolic execution for Spectre with haunted RelSE. In *Annu. Network and Distributed System Security Symp. (NDSS)*, 2021.

[17] Hernán Ponce de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. Portability analysis for weak memory models. PORTHOS: one tool for all models. In *24th Int. Symp. Static Analysis (SAS)*, volume 10422 of *LNCS*, pages 299–320. Springer, 2017.

[18] Hernán Ponce de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. Dartagnan: Bounded model checking for weak memory models (competition contribution). In *Proc. 26th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 12079 of *LNCS*, pages 378–382. Springer, 2020.

[19] Advanced Micro Devices. Security analysis of AMD predictive store forwarding. https://www.amd.com/system/files/documents/security-analysis-predictive-store-forwarding.pdf, 2020.

[20] Craig Disselkoen, Radha Jagadeesan, Alan Jeffrey, and James Riely. The code that never ran: Modeling attacks on speculative evaluation. In *Proc. IEEE Symp. Security and Privacy (S&P)*. IEEE, 2019.

[21] Natalia Gavrilenko, Hernán Ponce de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. BMC for weak memory models: Relation analysis for compact SMT encodings. In *Proc. 31th Int. Conf. Computer Aided Verification (CAV)*, volume 11561 of *LNCS*, pages 355–365. Springer, 2019.

[22] Roberto Guanciale, Musard Balliu, and Mads Dam. InSpectre: Breaking and fixing microarchitectural vulnerabilities by formal analysis. In *Proc. ACM SIGSAC Conf. Computer and Communications Security (CCS)*, pages 1853–1869. ACM, 2020.

[23] Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. Spectector: Principled detection of speculative information flows. In *Proc. IEEE Symp. Security and Privacy (S&P)*, pages 1–19. IEEE, 2020.

[24] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. Hardware/-software contracts for secure speculation. In *Proc. IEEE Symp. Security and Privacy (S&P)*. IEEE, 2021.

[25] Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Order Number 248966-044b, June 2021.

[26] Egor George Karpenkov, Karlheinz Friedberger, and Dirk Beyer. JavaSMT: A unified interface for SMT solvers in Java. In *8th Int. Conf. Verified Software, Theories, Tools, and Experiments (VSTTE)*, volume 9971 of *LNCS*, pages 139–148, 2016.

[27] Paul Kocher. Spectre mitigations in Microsoft's C/C++ compiler. https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html, 2018.

[28] 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. Spectre attacks: Exploiting speculative execution. In *Proc. IEEE Symp. Security and Privacy (S&P)*, pages 1–19. IEEE, 2019.

[29] Michalis Kokologiannakis and Viktor Vafeiadis. Genmc: A model checker for weak memory models. In *Proc. Int. Conf. Computer Aided Verification (CAV)*, pages 427–440. Springer, 2021.

[30] K. Rustan M. Leino. This is Boogie 2. Technical Report KRML 178, Microsoft Research, 2008.

[31] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *Proc. IEEE Symp. Security and Privacy (S&P)*, pages 605–622. IEEE, 2015.

[32] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. Specfuzz: Bringing spectre-type vulnerabilities to the surface. In *Proc. 29th USENIX Security Symposium (USENIX Security)*, pages 1481–1498. USENIX Association, 2020.

[33] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. *Proc. ACM Program. Lang.*, 2(POPL):19:1–19:29, 2018.

[34] Zhenxiao Qi, Qian Feng, Yueqiang Cheng, Mengjia Yan, Peng Li, Heng Yin, and Tao Wei. SpecTaint: Speculative taint analysis for discovering spectre gadgets. In *Annu. Network and Distributed System Security Symp. (NDSS)*, 2021.

[35] Hany Ragab, Enrico Barberis, Herbert Bos, and Cristiano Giuffrida. Rage Against the Machine Clear: A Systematic Analysis of Machine Clears and Their Implications for Transient Execution Attacks. In *Proc. 30th USENIX Security Symposium (USENIX Security)*, 2021.

[36] Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus O. Myreen, and Jade Alglave. The semantics of x86-CC multiprocessor machine code. In *Proc. 41st Symp. Principles of Programming Languages (POPL)*. ACM, 2009.

[37] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. Checkmate: Automated synthesis of hardware exploits and security litmus tests. In *MICRO*, pages 947–960. IEEE Computer Society, 2018.

[38] Guanhua Wang, Sudipta Chattopadhyay, Arnab Kumar Biswas, Tulika Mitra, and Abhik Roychoudhury. KLEESpectre: Detecting information leakage through speculative cache attacks via symbolic execution. *ACM Trans. Softw. Eng. Methodol.*, 29(3):14:1–14:31, 2020.

[39] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. oo7: Low-overhead defense against spectre attacks via binary analysis. *CoRR*, abs/1807.05843, 2018.

[40] Meng Wu and Chao Wang. Abstract interpretation under speculative execution. In *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, pages 802–815. ACM, 2019.