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 and axiomatic approach to specifying speculative semantics that has been used in CAT to axiomatize the concurrency semantics of processors like X86, POWER and ARM [1], [2], [4].

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
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

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 \( \text{exp}(\ell) \) to refer to the target register and the memory expression of a given instruction, respectively. Finally, programs are sequences of instructions.

Statement \( r \rightarrow e \) computes the value of expression \( e \) and locally assigns it to register \( r \). The conditional assignment \( r \leftarrow e \) 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 \( \ell \) redirects the control flow to \( \ell \). The conditional jump beqz \( r,\ell \) redirects the control flow to \( \ell \) 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 \( \ell \) as

\[
PRED(\ell) := \{ \ell' \in \mathbb{N} \mid (\ell' + 1 = \ell \land \ell' \neq \text{jmp } \ell') \lor \\
\ell' = \text{jmp } \ell \lor \ell' = \text{beqz } r,\ell \}.
\]

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

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 representated 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
we model speculative execution, we leave such properties.

There are certain properties that give for each read a unique write to the same address, i.e., \( g \) provides a location relation \( ℓ \) to memory events that access an address \( M \). By \( M_a \) we refer to memory events that access an address \( a \). Finally, if \( ℓ \) is an instruction using a register and \( e_ℓ \) is an event representing an instance of \( ℓ \), we use \( \text{val}(e_ℓ) \) to represent the value of the register for that given instance.

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

\[
\text{rf} \subseteq (W \times R) \cap \text{loc}
\]
\[
\forall r \in R : \exists w \in W : \text{rf}(w, r)
\]
\[
\text{rf}(w, r) \Rightarrow \text{val}(w) = \text{val}(r)
\]

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 \( r_i \); this is represented by the edge \( \text{rf}(e_s, e_i) \).

The coherence order \( \text{co} \) relates writes to the same address and forms a total order for each address:

\[
\text{co} \subseteq (W \times W) \cap \text{loc}
\]
\[
\forall a \in \mathbb{N}: \text{total}(\text{co}, W_a)
\]

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 initial writes to all addresses by a unique event \( e_0 : \text{init} \). The only exception is address \( \text{sec} \) for which we use \( e_s : \text{sec} \); this event models the initial value of the secret.

The program order \( \text{po} \subseteq (X \times X) \) represents the order in which instructions are written. For instructions not being part of the same loop we have \( (e_ℓ, e'_ℓ) \in \text{po} \Rightarrow ℓ < ℓ' \). For instructions belonging to the same loop, we have that \( (e_ℓ, e'_ℓ) \in \text{po} \Rightarrow ℓ < ℓ' \land i ≤ j \lor ℓ ≥ ℓ' \land i < j \).

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

\[
\text{addr} := \{(e_ℓ, e_ℓ') \subseteq (R \times M) \cap \text{po} \mid \text{exp}(ℓ) \in \text{addr}(ℓ') \land \text{reg}(ℓ') < ℓ' \land \text{reg}(ℓ'') = \text{reg}(ℓ)\}
\]

A candidate execution is a triple \( (X, \text{rf}, \text{co}) \). Each different combination of these three yields a possible behavior. Once \( X \) is fixed, relations \( po, \text{fence} \) and \( \text{addr} \) 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 \( \text{rf} \) 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 ARMRv8 is part of its official documentation [2]. In this paper we show that CAT can also be used to model the
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].

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 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:

If ℓ′ + 1 = ℓ ∧ ℓ′ ≠ beqz r, ℓ″, then eℓ ∈ X ⇒ eℓ′ ∈ X

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

If ℓ′ = jmp ℓ, then eℓ ∈ X ⇒ eℓ′ ∈ 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 ≠ 0, otherwise the jump would be taken:

If ℓ′ + 1 = ℓ ∧ ℓ′ = beqz r, ℓ″, then eℓ ∈ X ⇒ (eℓ′ ∈ X ∧ val(eℓ′))

Here, we interpret val(eℓ′) as a boolean to denote val(eℓ′) ≠ 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:

If ℓ′ = beqz r, ℓ, then eℓ ∈ X ⇒ (eℓ′ ∈ X ∧ ¬val(eℓ′))

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 X must satisfy this. Combining the four cases above, a given instance of an instruction ℓ is executed if we have that
eℓ ∈ X ⇒ \( CDF(ℓ, ℓ′) \) (1)

where the CDF (control flow dependency) predicate is defined as

\[
CDF(ℓ, ℓ′) := \begin{cases} 
  eℓ′ ∈ X & \text{if } ℓ′ + 1 = ℓ ∧ ℓ′ ≠ beqz r, ℓ″ \\
  eℓ′ ∈ X & \text{if } ℓ′ = jmp ℓ \\
  eℓ′ ∈ X ∧ val(eℓ′) & \text{if } ℓ′ + 1 = ℓ ∧ ℓ′ = beqz r, ℓ″ \\
  eℓ′ ∈ X ∧ ¬val(eℓ′) & \text{if } ℓ′ = beqz r, ℓ \\
\end{cases}
\]

Note that (2) does not need to consider the case where ℓ′ + 1 = ℓ ∧ ℓ′ ≠ jmp ℓ″ 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. X = {e1,...,e7}. For this execution, (2) requires that val(e2) ≠ 0, otherwise the jump would have been taken and e3 ∈ X. As val(e2) ≠ 0, we have r1 < A.size and A + r1 is in bounds. This means the dashed relation is not possible because e5:sec and e4 access different addresses ((e5,e4) ⊈ loc) and the reads-from relation requires r ⊈ loc. The second path has val(e2) = 0. In this execution the jump is taken and the control flow is redirected from 3 to 7, i.e. X = {e1,e2,e3,e7}. Since only e1 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 C ⊆ 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 T ⊆ X. We consider as executed any instruction that is either committed or transiently executed and have that X = C ∪ T.

For each event eℓ representing a conditional jump at ℓ we use proposition bpc_eℓ 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_eℓ unconstrained.

In §III-A we formalized the properties 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 have been taken) and that the branch predictor is correct:

\[
\text{If } \ell' + 1 = \ell \land \ell' = \text{beqz } r, \ell'', \\
\text{then } e_{\ell} \in C \Rightarrow (e_{\ell'} \in C \land \text{val}(e_{\ell'}) \land \text{bpc}_{e_{\ell'}})
\]

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

\[
\text{If } \ell' = \text{beqz } r, \ell, \\
\text{then } e_{\ell} \in C \Rightarrow (e_{\ell'} \in C \land \text{val}(e_{\ell'}) \land \text{bpc}_{e_{\ell'}})
\]

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

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

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

\[
\text{CFD}(\ell, \ell') := \begin{cases} 
    e_{\ell'} \in C & \text{if } \ell' + 1 = \ell \land \ell' \neq \text{beqz } r, \ell'' \\
    e_{\ell'} \in C & \text{if } \ell' = \text{jmp } \ell \\
    e_{\ell'} \in C \land \text{val}(e_{\ell'}) \land \text{bpc}_{e_{\ell'}} & \text{if } \ell' + 1 = \ell \land \ell' = \text{beqz } r, \ell'' \\
    e_{\ell'} \in C \land \text{val}(e_{\ell'}) \land \text{bpc}_{e_{\ell'}} & \text{if } \ell' = \text{beqz } r, \ell 
\end{cases}
\]

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 \(\ell'\) is a conditional jump immediately preceding \(\ell\) but targeting a different label, then \(\ell\) executing transiently implies that \(\ell'\) 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:

\[
\text{If } \ell' + 1 = \ell \land \ell' = \text{beqz } r, \ell'', \\
\text{then } e_{\ell} \in \mathcal{T} \Rightarrow (e_{\ell'} \in C \land \text{val}(e_{\ell'}) \land \neg \text{bpc}_{e_{\ell'}})
\]

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

\[
\text{If } \ell' = \text{beqz } r, \ell, \\
\text{then } e_{\ell} \in \mathcal{T} \Rightarrow (e_{\ell'} \in C \land \text{val}(e_{\ell'}) \land \neg \text{bpc}_{e_{\ell'}})
\]

We capture speculative control flow dependencies with the constraint

\[
\text{CFD}(\ell, \ell') := \bigvee_{\ell' \in \text{PRED}(\ell')} \text{SCFD}(\ell, \ell')
\]

where the speculative control flow dependency \(\text{SCFD}\) is defined as

\[
\text{SCFD}(\ell, \ell') := \begin{cases} 
    e_{\ell'} \in \mathcal{T} & \text{if } \ell' + 1 = \ell \land \ell' \neq \text{beqz } r, \ell'' \\
    e_{\ell'} \in \mathcal{T} & \text{if } \ell' = \text{jmp } \ell \\
    e_{\ell'} \in C \land \text{val}(e_{\ell'}) \land \neg \text{bpc}_{e_{\ell'}} & \text{if } \ell' + 1 = \ell \land \ell' = \text{beqz } r, \ell'' \\
    e_{\ell'} \in C \land \text{val}(e_{\ell'}) \land \text{bpc}_{e_{\ell'}} & \text{if } \ell' = \text{beqz } r, \ell
\end{cases}
\]

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 \(\text{bpc}_{e_{\ell}}\) 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:

\[
\neg \text{val}(e_{2}), \neg \text{bpc}_{e_{3}}, C = \{e_{1}, e_{2}, e_{3}\}, T = \{e_{4}, e_{5}, e_{6}, e_{7}\} \quad (4)
\]

\[
\text{val}(e_{2}), \neg \text{bpc}_{e_{3}}, C = \{e_{1}, e_{2}, e_{3}\}, T = \{e_{7}\}
\]

Note that (4) has \(\neg \text{val}(e_{2})\) and thus \(\text{idx} \geq \text{A.size}\). Because \(\text{idx}\) is not in bounds, it is possible that \(A + r_{1} = \text{sec}\). In this scenario, both \(e_{3}\) spec and \(e_{4}\) access the same address \((e_{s}, e_{d}) \in \text{loc}\) and thus the \(r\) edge is possible, showing that the secret can be read and the program is vulnerable to \(\text{SPECTRE-V1}\).

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 \(\text{sw}\) 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:

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

We model that an instruction can only execute transiently during the speculation window using the following constraint to restrict the set \(\mathcal{T}: \text{po} \subseteq \text{po}; ((\mathcal{T}); \text{po})^{\leq \text{sw}}\). The constraint imposes that the number of \(\text{po}\)-consecutive transiently executed events is smaller than \(\text{sw}\). Note that \(\mathcal{T}\) can still have more than \(\text{sw}\) elements if several mispredictions occur and the corresponding transient events are not \(\text{po}\)-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 \(\text{SPECTRE-V1}\). We model this by enforcing that fences cannot be executed transiently:

\[
\bigwedge_{\ell': \text{fence}} e_{\ell} \notin \mathcal{T}
\]

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 \(\text{SPECTRE-V1}\) attacks, but also to prove that common mitigations work.
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 e5 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 com ∪ po and forces it to be acyclic (otherwise instructions could not be scheduled to satisfy their dependencies). From rf⁻¹ ∪ co we have that if a read r gets its value from a write w (rf⁻¹(r, w)), then any other write w’ coming after w in the coherence order (co(w, w’)) must come after r (rf⁻¹; 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 e4 must read from the last store to that address (here e5), otherwise there would be a cycle. Having rf(e5, e4) enforces that r2 gets the masked value of idx. Because of the masking, we have r2 < A.size. Finally, since the address of instruction e5 is in-bounds, e5 can only read the initial value of the array.

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 e3 is not yet known, the processor might predict that e3 and e4 will not alias and speculatively execute e4 before e3. 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 po\([ \mathbb{W} \times \mathbb{R} ]\) 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 win = [\mathbb{W}]; po; [([\mathbb{W}]; po)^{sw-1}]; [\mathbb{R}] where sw is the size of the store buffer, with sw \geq 1. Note that sw can differ from the speculation window sw created by the branch predictor. For r^2k 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) | e \in \mathbb{W}\} and [\mathbb{R}] = \{(e, e) | 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 po; ([[\mathbb{W}]; po)^{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}]; po; [\mathbb{W}] on the right 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 \muASM 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 (e3, e4) 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 e3). Since it is possible to reorder e3 and e4 because the initial value of idx is not transiently executed, the program is still vulnerable to PECTRE-PSF. AMD recently confirmed that its Zen 3 processor is vulnerable to this attack [19]. We present a CAT model allowing this kind of prediction and modeling such an attack.

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 e4 \rightarrow e0 \rightarrow e3 \rightarrow fence \rightarrow e4, 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 e4 will get the masked value in r2 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 processor 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 e4 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 e3 and e4 alias. In this scenario, register r2 gets the value 64 which is then multiplied by 1 (the value of idx) to compute the address of e5. Since the size of A is smaller than 64,
### Predictive Store Forwarding

#### SPECTRE-v4 – predictive store forwarding (unsafe).

![Diag](...)

- `e0: init` performs an out-of-bounds access allowing to read the secret.
- `e1: r1 ↦ idx < C.size` stores `C[0] = 64;`
- `e2: beqz r1, ⊥`
- `e3: store C + 0, 64`
- `e4: load r2, C + idx`
- `e5: load r2, r1 * idx`
- `e6: load rf0, temp`
- `e7: load rf0, temp`
- `e8: load rf0, temp`
- `e9: store temp, r3 & rf0`

### 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 com ∪ (po ∩ 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, rf 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 to represent any implementation of it 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 `x = 1`, then it should also observe `y = 1`. Since the TSO semantics guarantees `r0 * (r0 – r1)` is always 0, the access to `L` is safe. The CAT model captures this since it forbids the (unsafe) candidate execution where `rf0 = 1 ∧ rf1 = 0` due to the cycle `e1 → e2 → e6 → e7 → rf0`.

Ragab et al. [35] noticed that if `e1` is a slow load (due to a cache miss) and `e2` a fast one (cache hit), event `e2` can be transiently executed before `e1`. Since the load 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 used to mount an attack.

We can use CAT to capture this speculation window by replacing `(R × M)` with `(R × 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 `e1, e2` neither access the same address nor have an address dependency, with this modification, `(e1, e2) ∉ po-tso` and the candidate execution becomes consistent. It is then possible that `r0 * (r0 – r1) ≠ 0` and thus, `e3` can read the secret if `A + 1 = sec`.

### 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
we implemented the new 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 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 SPECTRE [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 correct 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-VI. Both KAIBYO and SPECTRE 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 fail to check for violations (see NONE column), but they cannot prove programs correct, because loops cannot be fully unrolled. While the same limitation applies to SPECTRE 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.

SPECTRE 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

\[\begin{align*}
\text{com} &= \text{co} \lor \text{rf} \lor \text{rf}^- \lor \text{co} \\
\text{com-tso} &= \text{co} \lor \text{rf} \lor \text{rf}^- \lor \text{co} \\
\text{po-tso} &= (\text{po} \cap [\text{RF} \times \text{RF}] \cup (\text{WM} \times \text{WM})) \lor \text{fence} \\
\text{acyclic} &= \text{com-tso} \cup \text{po-tso}
\end{align*}\]

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

\[\begin{align*}
\text{uint8_t A}[1]; \\
\text{thread}_1: & \quad \text{r}_0 = \text{x}; & \quad \text{e}_4: & \quad \text{r}_2 \leftarrow 1 \\
\text{r}_1 = \text{y}; & \quad \text{temp} \text{ &= B[A[r_1-r_0]]}; & \quad \text{e}_5: & \quad \text{r}_3 \leftarrow 1 \\
\text{thread}_2: & \quad \text{y = 1; } & \quad \text{e}_6: & \quad \text{r}_2, \text{y} \\
\text{x = 1; } & \quad \text{e}_7: \text{load } r_0, x & \quad \text{e}_8: \text{store } r_2, y
\end{align*}\]
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 ri. 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.
While there are four possible outcomes for branch prediction verify non-interference-style properties. Since this requires scalable non-BMC tools based on axiomatic semantics [29].

The memory model community recently made progress on axiomatic semantics is an active area of research; the weak reasoning about pairs of executions, doing so would affect scalability. Improving performance of tools based on axiomatic semantics into a concrete analysis tool, but it provides security foundations to build tools that help software developers write microarchitecturally secure programs. On the other side, for hardware developers, contracts are 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.

VII. DISCUSSION

KAIIBYO analyzes programs w.r.t software isolation. Currently, it cannot analyze non-interference-style properties, and our evaluation shows the consequences of this. KAIIBYO 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.

KAIIBYO 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].

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

The solving times of SPECTECTOR and BNSSEC 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 KAIIBYO. 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).

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. SPECTCTOR 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 speculative 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-of-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.

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

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

Fig. 10. A variant of SPECTRE-v4 from [14] written in C (left) and compiled to x86 with GCC 8.3.0 (right).
```
without enforcing a specific implementation. Contracts are 2-hypersafety properties [11] and thus require appropriate techniques to efficiently model pairs of traces. ReSE is a promising approach to extend symbolic execution for analyzing security properties of two execution traces [15]. In fact, an extension to ReSE 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. Disselkön 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 used 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.