Detecting speculative leaks with compositional semantics

XAVER FABIAN, CISPA Helmholtz Center for Information Security, Germany and University of Trento, Italy
MARCO GUARNIERI, IMDEA Software Institute, Spain
BORIS KÖPF, Azure Research, Microsoft, UK
JOSE F. MORALES, IMDEA Software Institute, Spain
MARCO PATRIGNANI, University of Trento, Italy
JAN REINEKE, Saarland University, Germany
ANDRES SANCHEZ, Amazon, Spain

Speculative execution enhances processor performance by predicting intermediate results and executing instructions based on these predictions. However, incorrect predictions can lead to security vulnerabilities, as speculative instructions leave traces in microarchitectural components that attackers can exploit. This is demonstrated by the family of Spectre attacks. Unfortunately, existing countermeasures to these attacks lack a formal security characterization, making it difficult to verify their effectiveness.

In this paper, we propose a novel framework for detecting information flows introduced by speculative execution and reasoning about software defenses. The theoretical foundation of our approach is *speculative non-interference* (SNI), a novel semantic notion of security against speculative execution attacks. SNI relates information leakage observed under a standard non-speculative semantics to leakage arising under semantics that explicitly model speculative execution. To capture their combined effects, we extend our framework with a mechanism to safely compose multiple speculative semantics, each focussing on a single aspect of speculation. This allows us to analyze the complex interactions and resulting leaks that can arise when multiple speculative mechanisms operate together. On the practical side, we develop Spectector, a symbolic analysis tool that uses our compositional framework and leverages SMT solvers to detect vulnerabilities and verify program security with respect to multiple speculation mechanisms. We demonstrate the effectiveness of Spectector through evaluations on standard security benchmarks and new vulnerability scenarios.

CCS Concepts: • **Security and privacy → Formal security models**; **Systems security**.

Additional Key Words and Phrases: Spectre; Speculative Execution; Speculative information flows; Speculative non-interference

ACM Reference Format:
Xaver Fabian, Marco Guarnieri, Boris Köpf, Jose F. Morales, Marco Patrignani, Jan Reineke, and Andres Sanchez. 2026. Detecting speculative leaks with compositional semantics. 1, 1 (July 2026), 60 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Speculative execution avoids pipeline stalls by predicting the results of intermediate computations and by speculatively executing instructions based on such predictions. Whenever a prediction turns out to be incorrect, the processor squashes

Authors' addresses: Xaver Fabian, CISPA Helmholtz Center for Information Security, Saarbrücken, Germany, xaver.fabian@cispa.de and University of Trento, Trento, Italy; Marco Guarnieri, IMDEA Software Institute, Madrid, Spain, marco.guarnieri@imdea.org; Boris Köpf, Azure Research, Microsoft, UK; Jose F. Morales, IMDEA Software Institute, Madrid, Spain; Marco Patrignani, University of Trento, Trento, Italy, marco.patrignani@unitn.it; Jan Reineke, Saarland University, Saarbrücken, Germany; Andres Sanchez, Amazon, Madrid, Spain.

Listing 1. Code vulnerable to branch speculation.

```
1 if (y < size)
2   temp &= B[A[y] * 512];
```

Listing 2. Code vulnerable to store speculation.

```
1 p = &secret;
2 p = &public;
3 temp = B[*p * 512];
```

Listing 3. Speculative leak arising from speculation over branch and store instructions combined.

```
1 x = 0;
2 p = &secret;
3 p = &public;
4 if (x != 0)
5     temp &= A[*p];
```

Fig. 1. Code snippets vulnerable to different kinds of speculation attacks.

the speculatively executed instructions, thereby rolling back their effects on the architectural state, which consists of registers, flags, and memory. However, the execution of speculative instructions leaves footprints in a CPU's microarchitectural components (like caches, predictors, and internal buffers) that may persist even after these instructions have been squashed. As shown by Spectre [49] and follow-up attacks [10, 15, 50, 53, 74, 78, 79, 81], these microarchitectural side effects can be exploited to compromise the security of programs and to leak information about speculatively accessed data.

Speculative Leaks. Modern CPUs employ a wide range of speculation mechanisms (branch predictors, memory disambiguators, etc.) that are used to speculate over different kinds of instructions and intermediate results, such as conditional branches [49], indirect jumps [49], store and load operations [43], return instructions [8, 50], and load addresses and loaded results [46, 47, 69]. All these speculation mechanisms can be exploited by attackers to leak data.

As an example, the code in Listing 1 is vulnerable to a Spectre-PHT attack [49], which exploits speculation over branch instructions. Whenever the branch prediction mispredicts the outcome of the condition `y > size` on Line 1, this results in speculatively executing Line 2, which leaks the out-of-bounds value pointed by `A[y]` through the data cache. As another example, Listing 2 depicts a code snippet vulnerable to a Spectre-STL attack [43], which exploits speculation over memory disambiguation checks. Whenever the memory write on Line 2 is predicted to have a different address than the memory read `*p` on Line 3, the content of `secret` is leaked to an attacker. Essentially, the memory write in Line 2 is bypassed because of speculation.

The majority of well-known attacks [7, 43, 49, 50, 53] only exploit leaks introduced by individual speculation mechanisms, e.g., branch predictors in Spectre-PHT (Listing 1) and memory disambiguators in Spectre-STL (Listing 2). However, some speculative leaks only arise due to the interaction of multiple mechanisms. For example, the code in Listing 3 can speculatively leak the value of `&secret` in Line 5 whenever (1) the memory write to `p` in Line 3 is predicted to have a different address then the memory read `*p` on Line 5, and (2) the branch instruction on Line 4 is mispredicted as taken. This leak, therefore, arises from the *combination* of two speculation mechanisms—branch prediction and memory disambiguation prediction—and *it cannot be detected* when considering the two mechanisms in isolation.

Limitations of existing leak-detection approaches. Since the discovery of Spectre [49], a number of approaches have been proposed for reasoning about the security of programs against speculative leaks [20, 25, 36, 39, 42, 44, 60, 61, 63, 76, 77, 80].

These approaches employ a wide array of techniques for reasoning about leaks (e.g., symbolic execution [20, 25, 36, 39, 76, 77], abstract interpretation [80], testing [42, 44, 60, 61, 63]), but they all share a critical limitation. They support only *fixed* speculation mechanisms, such as branch prediction [36, 38, 75–77] and memory disambiguation prediction [20, 25, 65], which are *hard-coded*, e.g., in the underlying formal models. As a result, extending them to reason about a new

speculation mechanism is often complicated (e.g., it might require modifying the underlying formal model and updating the corresponding security proofs), since none of these approaches has been designed in a *compositional* manner.

As shown by code snippets like Listing 3, sound reasoning about speculative leaks in a program requires accounting for *all* speculation mechanisms, since ignoring some mechanisms might lead to missed leaks. Unfortunately, (1) we lack a precise understanding of the speculation mechanisms implemented in current CPUs, as demonstrated time and again by attacks discovering previously unknown speculation mechanisms, and (2) new CPU generations often implement refined and improved speculation mechanisms to improve performance [46, 47]. Since we cannot develop an analysis that would account for future microarchitectures, approaches for reasoning about speculative leaks must be *compositional* and easy to *extend* whenever a new speculation mechanism is discovered. However, we currently lack a precise characterization of security against speculative leaks and an associated program analysis framework that is *extensible and compositional* in the underlying speculation mechanisms.

Our approach: In this paper, we develop a novel, principled approach for reasoning about leaks introduced by speculatively executed instructions and for reasoning about software defenses against Spectre-style attacks. Our approach is backed by a semantic characterization of security against speculative leaks and it comes with an algorithm, based on symbolic execution, for proving the absence of leaks. Crucially, our approach is *compositional*. It allows specifying individual operational semantics—each one capturing the effects of a different speculation mechanism—and combining them in a single composed semantics, which captures leaks arising from the interactions of all these mechanisms, thereby leading to simpler formalizations. Additionally, these semantics can be directly integrated in our verification algorithm, whose soundness proof is compositional and defined in terms of the component semantics, thereby maximizing proof reuse. Next, we describe our contributions in more detail:

(1) **Modeling speculation:** We propose a generalized template for modeling the effects of speculative execution as an operational semantics. Our template extends an architectural (non-speculative) semantics (Section 3) to a *speculative semantics* (Section 4) that captures the effects of speculatively executed instructions. In a nutshell, the speculative semantics follow mispredicted paths for a bounded number of steps before backtracking and restarting the architectural execution, while relying on a *prediction oracle* to model the speculation mechanism. Note that this template is flexible enough to capture both control-flow and data-flow speculation. Following this template, we introduce five speculative semantics (denoted $\mathcal{A}_{\mathsf{B}}$, $\mathcal{A}_{\mathsf{J}}$, $\mathcal{A}_{\mathsf{S}}$, $\mathcal{A}_{\mathsf{R}}$, and $\mathcal{A}_{\mathsf{SLS}}$) which capture speculation over branches, jumps, stores, and return instructions (Section 6). This is the most comprehensive collection of individual speculative semantics in a single framework to date.

(2) **Speculative non-interference:** We propose *speculative non-interference* (Section 5), a novel semantic notion of security against speculative execution attacks. Speculative non-interference is based on comparing a program with respect to two different semantics: (a) a standard, *non-speculative semantics*, which we use as a proxy for the intended program behavior, and (b) a *speculative semantics* that captures the effects of speculation induced by one or more speculation mechanisms, which captures the effect of speculatively executed instructions. In a nutshell, speculative non-interference requires that *speculatively executed instructions do not leak more information into the microarchitectural state than what the intended behavior does*, i.e., than what is leaked by the standard, non-speculative semantics.
To capture "leakage into the microarchitectural state", we consider an observer of the program execution that sees the locations of memory accesses and jump targets. This observer model is commonly used for characterizing "side-channel free" or "constant-time" code [4, 57] in the absence of detailed models of the microarchitecture. Under this observer model, an adversary may distinguish two initial program states if they yield different traces of memory locations and

jump targets. *Speculative non-interference* (SNI) requires that two initial program states can be distinguished under the speculative semantics only if they can also be distinguished under the standard, non-speculative semantics.
The speculative semantics, and hence SNI, depends on the decisions taken by the prediction oracle. We show that one can abstract away from the specific oracle by considering a worst-case oracle that *always mispredicts*. SNI w.r.t. this always-mispredict oracle implies SNI w.r.t. a large class of prediction strategies. This allows reasoning about speculative leaks while ignoring the details of the specific prediction strategy, which are often undocumented.

(3) **Composition framework:** We propose a framework for composing speculative semantics that capture speculation due to different mechanisms (Section 7). The framework allows specifying individual semantics for each speculation mechanism and combining them into a composed semantics. The combination yields a single operational semantics which can be used to reason about leaks involving the different kinds of speculation it comprises (as in Listing 3). We also formalize the key properties of our framework: if the individual semantics fulfill some (expected) safety conditions (which we prove for all the semantics we combine), then the composed semantics is well-formed and can be used to reason about leakage in a sound manner. That is, whenever a program is leaky under one of the individual semantics then it is leaky under the composed semantics and, vice versa, whenever a program is SNI under the composed semantics, then it is SNI w.r.t. all individual semantics. We apply the composition framework to all our individual semantics, thereby obtaining 18 combined speculative semantics (Section 8).

(4) **Checking speculative non-interference:** We propose Spectector (Section 9), an algorithm to automatically prove that programs satisfy SNI w.r.t. any speculative semantics (or composition) in our framework. Given a program p, Spectector uses symbolic execution with respect to the speculative semantics to derive a concise representation of the traces of memory accesses and jump targets during execution along all possible program paths. Based on this representation, Spectector creates a symbolic formula capturing that, whenever two initial program states produce the same (instruction and data) memory access patterns in the standard semantics, they also produce the same access patterns in the speculative semantics. Validity of this formula for each program path implies SNI.

(5) **Implementation and evaluation:** We implement a prototype of Spectector (Section 10), with a front-end parsing (a subset of) x86 assembly and a back-end that uses the Z3 SMT solver to perform symbolic execution against all our speculative semantics (and their combinations) and to determine whether programs contain speculative leaks. We validate Spectector on both existing microbenchmarks (for speculation on branches, indirect jumps, store and return instructions) and on new ones (for speculative leaks arising from combinations of speculation mechanisms) that we introduce. These microbenchmarks contain both leaky programs as well as programs patched with well-known compiler-level countermeasures against Spectre. Using Spectector, we successfully (1) detect all leaks pointed out in [48], (2) detect novel, subtle leaks that are out of the scope of existing approaches that check for known vulnerable code patterns [77], (3) detect leaks arising from speculation over multiple mechanisms that are not detected by existing symbolic approaches [14], and (4) identify cases where compilers unnecessarily inject countermeasures, i.e., opportunities for optimization without sacrificing security.

Scope of this paper. This paper provides a unified, extended, and updated version of the results presented in two conference papers by Guarnieri et al. [38] and Fabian et al. [31]. In addition to the original results from [31, 38], (1) the paper introduces a general template for speculative operational semantics that generalizes existing formalizations, (2) it adds new speculative semantics for straight-line speculation and for speculation over indirect jumps (and their combinations), and (3) it provides additional technical details about the speculative semantics as well as about composition proofs. The paper is accompanied by an updated version of Spectector (with an updated evaluation) that

Listing 4. Spectre-PHT- Assembly code

```
1  mov   size, %rax
2  mov   y, %rbx
3  cmp   %rbx, %rax
4  jbe   END
5  mov   A(%rbx), %rax
6  shl   $9, %rax
7  mov   B(%rax), %rax
8  and   %rax, temp
```

supports all new speculative semantics and combinations presented in the paper. Overall, we see this paper as a unified reference for the modeling of speculative leaks at program level and for the Spectector program analysis tool.

Additional materials. The Spectector program analysis tool is open-source and available at [37]. Given the number of different speculative semantics studied in this paper, we leave the full formalization of all semantics, technicalities, and proof details to the associated technical report, which is available at [30].

2 ILLUSTRATIVE EXAMPLE

To illustrate our approach, we show how Spectector applies to the Spectre-PHT example [49] shown in Listing 1 using a speculative semantics capturing branch speculation.

Spectre-PHT. The program checks whether the index stored in the variable `y` is less than the size of the array `A`, stored in the variable `size`. If that is the case, the program retrieves `A[y]`, amplifies it with a multiple (here: 512) of the cache line size, and uses the result as an address for accessing the array `B`.

If `size` is not cached, evaluating the branch condition requires traditional processors to wait until `size` is fetched from main memory. Modern processors instead speculate on the condition's outcome and continue the computation. Hence, the memory accesses in line 2 may be executed even if $y \geq size$.

When `size` becomes available, the processor checks whether the speculated branch is the correct one. If it is not, it rolls back the architectural (i.e., ISA) state's changes and executes the correct branch. However, the speculatively executed memory accesses leave a footprint in the microarchitectural state, in particular in the cache, which enables an adversary to retrieve `A[y]`, even for $y \geq size$, by probing the array `B`.

Detecting leaks with Spectector. Spectector automatically detects leaks introduced by speculatively executed instructions, or proves their absence. Specifically, Spectector detects a leak whenever executing the program under the speculative semantics, which captures that the execution can go down a mispredicted path for a bounded number of steps, leaks more information into the microarchitectural state than executing the program under a non-speculative semantics.

To illustrate how Spectector operates, we consider the x86 assembly[1] translation of Listing 1's program (cf. Listing 4). Spectector performs symbolic execution with respect to the speculative semantics to derive a concise representation of the concrete traces of memory accesses and program counter values along each path of the program. These symbolic traces capture the program's effect on the microarchitectural state.

[1]We use a simplified AT&T syntax without operand sizes

Listing 5. Spectre-PHT- Assembly code with speculative load hardening. Clang inserted instructions 3, 6, 9, and 11.

```
1  mov     size, %rax
2  mov     y, %rbx
3  mov     $0, %rdx
4  cmp     %rbx, %rax
5  jbe     END
6  cmovbe  $-1, %rdx
7  mov     A(%rbx), %rax
8  shl     $9, %rax
9  or      %rdx, %rax
10 mov     B(%rax), %rax
11 or      %rdx, %rax
12 and     %rax, temp
```

During speculative execution, the speculatively executed parts are determined by the predictions of the branch predictor. As shown in Section 6.1, leakage due to speculative execution is maximized under a branch predictor that mispredicts every branch. The code in Listing 4 yields two symbolic traces w.r.t. the speculative semantics that mispredicts every branch:[2]

$$\texttt{start} \cdot \texttt{rlb} \cdot \overline{\tau} \quad \text{when} \quad \texttt{y} < \texttt{size} \tag{1}$$

$$\texttt{start} \cdot \overline{\tau} \cdot \texttt{rlb} \quad \text{when} \quad \texttt{y} \geq \texttt{size} \tag{2}$$

where $\overline{\tau} = \texttt{load}\ (\texttt{A} + \texttt{y}) \cdot \texttt{load}\ (\texttt{B} + \texttt{A[y]}\ \texttt{*}\ 512)$. Here, the argument of load is visible to the observer, while start and rlb denote the start and the end of a misspeculated execution. The traces of the *non-speculative* semantics are obtained from those of the speculative semantics by removing all observations in between start and rlb.

Trace 1 shows that whenever y is in bounds (i.e., $\texttt{y} < \texttt{size}$) the observations of the speculative semantics and the non-speculative semantics coincide (i.e. they are both $\overline{\tau}$). In contrast, Trace 2 shows that whenever $\texttt{y} \geq \texttt{size}$, the speculative execution generates observations $\overline{\tau}$ that depend on A[y] whose value is not visible in the non-speculative execution. This is flagged as a leak by Spectector.

Proving Security with Spectector. The Clang 7.0.0 C++ compiler implements a countermeasure, called speculative load hardening [19], that applies conditional masks to addresses to prevent leaks into the microarchitectural state. Listing 5 depicts the protected output of Clang on the program from Listing 1.

The symbolic execution of the speculative semantics produces, as before, Trace 1 and Trace 2, but with

$$\overline{\tau} = \texttt{load}\ (\texttt{A} + \texttt{y}) \cdot \texttt{load}\ (\texttt{B} + (\texttt{A[y]}\ \texttt{*}\ 512)\ |\ \mathit{mask}),$$

where $\mathit{mask} = \textbf{ite}(\texttt{y} < \texttt{size}, \texttt{0x0}, \texttt{0xFF..FF})$ corresponds to the conditional move in line 6 and | is a bitwise-or operator. Here, $\textbf{ite}(\texttt{y} < \texttt{size}, \texttt{0x0}, \texttt{0xFF..FF})$ is a symbolic if-then-else expression evaluating to 0x0 if $\texttt{y} < \texttt{size}$ and to 0xFF..FF otherwise.

The analysis of Trace 1 is as before. For Trace 2, however, Spectector determines (via a query to Z3 [26]) that, for all $\texttt{y} \geq \texttt{size}$ there is exactly *one* observation that the adversary can make during the speculative execution, namely $\texttt{load}\ (\texttt{A} + \texttt{y}) \cdot \texttt{load}\ (\texttt{B} + \texttt{0xFF..FF})$, from which it concludes that no information leaks into the microarchitectural state, i.e., the countermeasure is effective in securing the program.

[2]For simplicity of presentation, the example traces capture only loads but not the program counter.

$$\text{(Programs) } p \coloneqq n : i \mid p_1; p_2 \qquad \text{(Functions) } \mathcal{F} \coloneqq \varnothing \mid \mathcal{F}; f \mapsto n$$
$$\text{(Expressions) } e \coloneqq n \mid x \mid \ominus e \mid e_1 \otimes e_2 \qquad \text{(Registers) } x \in \mathit{Regs} \qquad \text{(Values) } n, l \in \mathit{Vals} = \mathbb{N} \cup \{\bot\}$$
$$\text{(Instructions) } i \coloneqq \textbf{skip} \mid x \leftarrow e \mid \textbf{load}\ x, e \mid \textbf{store}\ x, e \mid \textbf{jmp}\ e \mid \textbf{beqz}\ x, l \mid x \xleftarrow{e'?} e \mid \textbf{spbarr} \mid \textbf{call}\ f \mid \textbf{ret}$$

Fig. 2. Syntax of the μAsm language

3 μASM LANGUAGE

In this section, we present μAsm, a simple assembly-style language serving as the basis for our framework. We start by describing the attacker model we consider (Section 3.1). Then, we present the syntax (Section 3.2) and the base non-speculative semantics (Section 3.3) of μAsm.

3.1 Observations and Attacker Model

We adopt a commonly-used attacker model [3, 20, 25, 35, 36, 38, 64, 75]: a passive attacker observing the execution of a program through events τ. These events, which we call *observations*, model timing leaks through cache and control flow while abstracting away low-level microarchitectural details. Specifically, we consider an attacker that observes the program counter and the locations of memory accesses during computation. This attacker model is commonly used to formalize timing side-channel free code [4, 57], without requiring microarchitectural models. In particular, it captures through data and instruction caches without requiring an explicit cache model.

$$\mathit{Obs} ::= \texttt{load}\ n \mid \texttt{store}\ n \mid \texttt{pc}\ n \mid \texttt{call}\ f \mid \texttt{ret}\ n \qquad \tau ::= \varepsilon \mid \mathit{Obs} \qquad \overline{\tau} ::= \emptyset \mid \overline{\tau} \cdot \tau$$

The `store` n and `load` n events denote read and write accesses to memory location n, so they capture leaks through the data cache. In contrast, `pc` n, `call` f, and `ret` n events record the control-flow of the program, thereby capturing leaks through, for instance, the instruction cache. An observation τ is either an event Obs or the empty observation ε. Traces $\overline{\tau}$ are sequences of observations; we indicate sequences of elements $[e_1; \cdots; e_n]$ as $\bar{e}$, and adding an element e to $\bar{e}$ as $\bar{e} \cdot e$.

3.2 Syntax of μAsm

μAsm is an assembly-like language whose syntax is presented in Figure 2. In μAsm, programs p are sequences of mappings from natural numbers n (i.e., the instruction address or label) to instructions i.

Instructions i include skipping **skip**, register assignments $x \leftarrow e$, loads **load** x, e, stores **store** x, e, indirect jumps **jmp** e, conditional branches **beqz** x, l, conditional assignments $x \xleftarrow{e'?} e$, speculation barriers **spbarr**, calls **call** f, and returns **ret**. Speculation barriers (**spbarr**) and conditional assignments ($x \xleftarrow{e'?} e$) are both commonly used to implement Spectre countermeasures. In particular, speculation barriers stop speculation outright, whereas conditional assignments are often used to replace branching instructions.

Instructions can refer to expressions e, which are constructed by combining registers and values with unary $\ominus$ and binary $\otimes$ operators. Registers come from the set *Regs*, containing register identifiers and designated registers **pc** and **sp** modelling the program counter and stack pointer respectively, whereas values come from the set *Vals*, which includes natural numbers and $\bot$. Note that we use $\bot$ to denote the termination of the program.

The function map $\mathcal{F}$ is a partial map from function names f to instruction numbers n.

We say that a program is *well-formed* if (1) it does not contain duplicate labels, (2) it contains an instruction labelled with 0, i.e., the initial instruction, (3) every function name f occuring in a **call** f instruction is in the domain of $\mathcal{F}$ and (4) it does not contain branch instructions of the form n : **beqz** $x, n+1$. Note that the last constraint ensures that all the outcomes of branch instructions are always reflected in the value of the program counter.

Example 1 (Spectre-PHT Example). The Spectre-PHT example from Listing 1 can be expressed in μAsm as follows:

$$0 : x \leftarrow \texttt{y} < \texttt{size}$$
$$1 : \textbf{beqz}\ x, \bot$$
$$2 : \textbf{load}\ z, \texttt{A} + \texttt{y}$$
$$3 : z \leftarrow z * 512$$
$$4 : \textbf{load}\ w, \texttt{B} + z$$
$$5 : \texttt{temp} \leftarrow \texttt{temp}\ \&\ w$$

Here, registers `y`, `size`, and `temp` store the respective variables. Similarly, registers `A` and `B` store the memory addresses of the first elements of the arrays `A` and `B`.

In the following, we use instruction keywords to denote the set of all instructions of a given type. For instance, **beqz** is the set of all branch instructions, i.e., $\textbf{beqz} = \{\textbf{beqz}\ x, l \mid x \in \mathit{Regs} \wedge l \in \mathit{Vals}\}$.

3.3 Non-speculative Semantics of μAsm

We now introduce the standard, non-speculative semantics of μAsm program, which models their execution on a platform without speculation. For this, we describe next the small-step operational non-speculative semantics $\rightarrow$.

The judgment for this semantics is $\langle p, \sigma\rangle \xrightarrow{\tau} \langle p, \sigma'\rangle$ and it reads: *"a program state $\langle p, \sigma\rangle$ steps to a new program state $\langle p, \sigma'\rangle$ producing observation τ"*. Program states $\langle p, \sigma\rangle \in \Omega$ consist of a program p and a configuration σ. The program p is used to look up the current instruction, whereas the configuration $\sigma = \langle m, a\rangle$ records the state of the memory m and register file a. Memories map addresses (which are natural numbers) to values, whereas register files map register identifiers to values.

$$\text{Configurations } \sigma ::= \langle m, a\rangle \qquad \text{Prog. States } \Omega ::= \langle p, \sigma\rangle$$
$$\mathit{RegisterFile}\ a ::= \emptyset \mid r; x \mapsto v \qquad \mathit{Registers}\ x \in \mathit{Regs}$$
$$\mathit{Memory}\ m ::= \emptyset \mid m; n \mapsto v \qquad \mathit{where}\ n \in \mathbb{N}$$

Figure 3 presents the rules defining the non-speculative semantics $\rightarrow$. The rules rely on the evaluation of expressions (indicated as $[\![e]\!](a) = n$) where expression e is evaluated to value n under register file a. In the rules, $a[x \mapsto n]$, where $x \in \mathit{Regs} \cup \mathbb{N}$ and $n \in \mathit{Vals}$, denotes the update of a map (memory or registers), whereas $a(x)$ denotes reading from a map. Finally, $\sigma(x)$, where $x \in \mathit{Regs}$ and $\sigma = \langle m, a\rangle$, denotes $a(x)$.

Most of the rules of the semantics in Figure 3 are standard; we describe selected rules below. Branch instructions emit observations recording the outcome of the branch (Rule Beqz-Sat, Rule Beqz-Unsat), while memory operations emit observations recording the accessed memory (Rule Store, Rule Load). A call to function f is a jump to the function's starting line number n, as indicated by the function map $\mathcal{F}$. A call stores the return address on the stack at the value of

Expression evaluation

$$[\![n]\!](a) = n \qquad [\![x]\!](a) = a(x) \qquad [\![\ominus e]\!](a) = \ominus[\![e]\!](a) \qquad [\![e_1 \otimes e_2]\!](a) = [\![e_1]\!](a) \otimes [\![e_2]\!](a)$$

$$\boxed{\langle p, \sigma\rangle \xrightarrow{\tau} \langle p, \sigma'\rangle}$$

$$\text{(Skip)}\ \frac{p(a(\mathbf{pc})) = \mathbf{skip}}{\langle p, \langle m, a\rangle\rangle \to \langle p, \langle m, a[\mathbf{pc} \mapsto a(\mathbf{pc}) + 1]\rangle\rangle}$$

$$\text{(Barrier)}\ \frac{p(a(\mathbf{pc})) = \mathbf{spbarr}}{\langle p, \langle m, a\rangle\rangle \to \langle p, \langle m, a[\mathbf{pc} \mapsto a(\mathbf{pc}) + 1]\rangle\rangle} \qquad \text{(Assign)}\ \frac{p(a(\mathbf{pc})) = x \leftarrow e \quad x \neq \mathbf{pc}}{\langle p, \langle m, a\rangle\rangle \to \langle p, \langle m, a[\mathbf{pc} \mapsto a(\mathbf{pc}) + 1, x \mapsto [\![e]\!](a)]\rangle\rangle}$$

$$\text{(ConditionalUpdate-Sat)}\ \frac{p(a(\mathbf{pc})) = x \xleftarrow{e'?} e \quad [\![e']\!](a) = 0 \quad x \neq \mathbf{pc}}{\langle p, \langle m, a\rangle\rangle \to \langle p, \langle m, a[\mathbf{pc} \mapsto a(\mathbf{pc}) + 1, x \mapsto [\![e]\!](a)]\rangle\rangle} \qquad \text{(ConditionalUpdate-Unsat)}\ \frac{p(a(\mathbf{pc})) = x \xleftarrow{e'?} e \quad [\![e']\!](a) \neq 0 \quad x \neq \mathbf{pc}}{\langle p, \langle m, a\rangle\rangle \to \langle p, \langle m, a[\mathbf{pc} \mapsto a(\mathbf{pc}) + 1]\rangle\rangle}$$

$$\text{(Terminate)}\ \frac{p(a(\mathbf{pc})) = \bot}{\langle p, \langle m, a\rangle\rangle \to \langle p, \langle m, a[\mathbf{pc} \mapsto \bot]\rangle\rangle}$$

$$\text{(Load)}\ \frac{p(a(\mathbf{pc})) = \mathbf{load}\ x, e \quad x \neq \mathbf{pc} \quad n = [\![e]\!](a)}{\langle p, \langle m, a\rangle\rangle \xrightarrow{\texttt{load}\ n} \langle p, \langle m, a[\mathbf{pc} \mapsto a(\mathbf{pc}) + 1, x \mapsto m(n)]\rangle\rangle}$$

$$\text{(Store)}\ \frac{p(a(\mathbf{pc})) = \mathbf{store}\ x, e \quad n = [\![e]\!](a)}{\langle p, \langle m, a\rangle\rangle \xrightarrow{\texttt{store}\ n} \langle p, \langle m[n \mapsto a(x)], a[\mathbf{pc} \mapsto a(\mathbf{pc}) + 1]\rangle\rangle}$$

$$\text{(Beqz-Sat)}\ \frac{p(a(\mathbf{pc})) = \mathbf{beqz}\ x, \ell \quad a(x) = 0}{\langle p, \langle m, a\rangle\rangle \xrightarrow{\texttt{pc}\ \ell} \langle p, \langle m, a[\mathbf{pc} \mapsto \ell]\rangle\rangle} \qquad \text{(Beqz-Unsat)}\ \frac{p(a(\mathbf{pc})) = \mathbf{beqz}\ x, \ell \quad a(x) \neq 0}{\langle p, \langle m, a\rangle\rangle \xrightarrow{\texttt{pc}\ a(\mathbf{pc})+1} \langle p, \langle m, a[\mathbf{pc} \mapsto a(\mathbf{pc}) + 1]\rangle\rangle}$$

$$\text{(Jmp)}\ \frac{p(a(\mathbf{pc})) = \mathbf{jmp}\ e \quad \ell = [\![e]\!](a)}{\langle p, \langle m, a\rangle\rangle \xrightarrow{\texttt{pc}\ \ell} \langle p, \langle m, a[\mathbf{pc} \mapsto \ell]\rangle\rangle}$$

$$\text{(Call)}\ \frac{p(a(\mathbf{pc})) = \mathbf{call}\ f \quad \mathcal{F}(f) = n \quad a' = a[\mathbf{pc} \mapsto n, \mathbf{sp} \mapsto a(\mathbf{sp}) - 8] \quad m' = m[a'(\mathbf{sp}) \mapsto a(\mathbf{pc}) + 1]}{\langle p, \langle m, a\rangle\rangle \xrightarrow{\texttt{call}\ f} \langle p, \langle m', a'\rangle\rangle} \qquad \text{(Ret)}\ \frac{p(a(\mathbf{pc})) = \mathbf{ret} \quad l = m(a(\mathbf{sp})) \quad a' = a[\mathbf{pc} \mapsto l, \mathbf{sp} \mapsto a(\mathbf{sp}) + 8]}{\langle p, \langle m, a\rangle\rangle \xrightarrow{\texttt{ret}\ l} \langle p, \langle m, a'\rangle\rangle}$$

Fig. 3. The non-speculative small-step semantics of μAsm.

the stack pointer **sp** and decreases **sp** (Rule Call). A return does the inverse: it looks up the return address via the stack pointer **sp** and then increases the stack pointer (Rule Ret).

Figure 4 presents the rules for deriving the trace of observations associated with an execution according to the non-speculative semantics $\to$. We denote by $(\langle p, \sigma\rangle) \Lsh_{NS} \overline{\tau}$ the trace $\overline{\tau}$ associated with executing the program p starting from an initial configuration σ until reaching a final configuration σ' (Rule NS-Trace) using the reflexive-transitive-closure $\Downarrow$ of the non-speculative semantics $\to$ (Rule NS-Reflection and Rule NS-Single). To simplify our notation, we introduce the shorthand $Beh_{NS}(p, \sigma)$ to denote the specific trace $\overline{\tau}$ generated by this execution. Finally, the *non-speculative behaviour* $Beh_{NS}(p)$ of a program p is the set of all traces generated by all possible initial states for program p (Rule NS-Beh).

$$\boxed{(\langle p,\sigma\rangle)\,\text{👻}_{NS}\overline{\tau}}$$

$$\frac{}{\langle p,\sigma\rangle \Downarrow_{\varepsilon} \langle p,\sigma\rangle}\ \text{(NS-Reflection)} \qquad \frac{\langle p,\sigma\rangle \Downarrow_{\overline{\tau}} \langle p,\sigma''\rangle \quad \langle p,\sigma''\rangle \xrightarrow{\tau} \langle p,\sigma'\rangle}{\langle p,\sigma\rangle \Downarrow_{\overline{\tau}\cdot\tau} \langle p,\sigma'\rangle}\ \text{(NS-Single)}$$

$$\frac{\exists \sigma' \quad \sigma' \in \mathit{FinalConf} \quad \langle p,\sigma\rangle \Downarrow_{\overline{\tau}} \langle p,\sigma'\rangle}{(\langle p,\sigma\rangle)\,\text{👻}_{NS}\overline{\tau}}\ \text{(NS-Trace)} \qquad \frac{}{\mathit{Beh}_{NS}(p) = \{\overline{\tau} \mid \exists \sigma \in \mathit{InitConf}.\ (\langle p,\sigma\rangle)\,\text{👻}_{NS}\overline{\tau}\}}\ \text{(NS-Beh)}$$

Fig. 4. The Reflexive-transitive-closure of $\rightarrow$ and behaviour of μAsm programs.

4 MODELLING SPECULATIVE EXECUTION

In this section, we propose a general approach for modeling, at program-level, the effects of speculative execution induced by different kinds of speculation mechanisms. For this, we extend a non-speculative (i.e., architectural) semantics to a *speculative semantics* that is equipped with a *prediction oracle* (which captures how predictions are made) used to explore mispredicted paths. For this, we first informally explain the core concepts behind our model (Section 4.1). Then, we formalize prediction oracles and prediction histories (Section 4.2). We conclude by presenting the states and the evaluation relation associated with the (speculative) oracle semantics (Section 4.3).

We remark that here we present only a general template for defining the oracle semantics. Such template needs to be instantiated to capture specific speculation mechanisms, and we do so in Section 6 for a large class of mechanisms. In the following, we indicate a speculative semantics as 👻_x, where the metavariable x is used as a placeholder indicating the type of speculation mechanism.

4.1 High-level Description

The core idea behind our approach is to extend a non-speculative semantics—like the one from Section 3—to a *speculative semantics* that captures the effect of speculatively executed instructions. For this, our speculative semantics is parametric in two components: (1) a set $\mathit{instr}_{\text{👻}}$ of μAsm instructions that might trigger speculation, and (2) a *prediction oracle* O capturing the prediction strategy associated with the modeled speculation mechanism.

In a nutshell, our speculative semantics works as follows. All instructions not in $\mathit{instr}_{\text{👻}}$ are executed as in the standard non-speculative semantics. In contrast, whenever an instruction in $\mathit{instr}_{\text{👻}}$ is reached, the oracle O is queried to obtain the next predicted state, which records the effects of the prediction. For instance, for branch prediction, the program counter in the predicted state is updated to decide which of the two branches to execute speculatively.

To enable a subsequent rollback in case of a misprediction, a snapshot of the current program state is taken, before starting a *speculative transaction*. In this speculative transaction, the program is executed speculatively starting from the predicted state for a bounded number w of computation steps, called the speculative window. To abstract from the complexity of estimating the actual speculative window (which depends on the details of a CPU's microarchitecture), in our model the speculation window w is also provided by the prediction oracle.

At the end of a speculative transaction, the correctness of the prediction is evaluated: (1) If the prediction was *correct*, the transaction is committed and the computation continues using the current configuration. (2) In contrast, if the prediction was *incorrect*, the transaction is aborted, the original configuration is restored, and the computation continues on the correct branch.

In the following sections, we formalize the behavior intuitively described above in the general structure of the *speculative oracle semantics*, whereas in Section 6 we will present instantiations of the oracle semantics (and $instr_{\text{🔔}}$) for different speculation mechanisms.

4.2 Prediction Oracles

In our semantics, the prediction oracle serves two distinct purposes: (1) predicting the next program state (i.e., modeling how the speculation mechanism works), and (2) determining the length of the speculative transactions (i.e., determining for how long the speculative execution lasts). Our notion of prediction oracle is general enough to account for both control-flow speculation, such as branch prediction, and data-flow speculation, such as value prediction.

A *prediction oracle* O is a partial function that takes as input a program p, a prediction history h, and the current program state σ such that $p(\sigma(\mathbf{pc}))$ is an instruction triggering speculation, i.e., $p(\sigma(\mathbf{pc})) \in instr_{\text{🔔}}$. The prediction oracle returns as output a triple $\langle \delta, w, \tau \rangle$, where $\delta \in Conf$ is a partial configuration indicating the predicted values, $w \in \mathbb{N}$ represents the speculative transaction's length, and τ is a (potentially empty) observation. One can then join the returned partial configuration δ with the current configuration σ, indicated as $\sigma \uplus \delta$, to obtain the next speculative configuration. Informally, $\sigma \uplus \delta$ simply updates σ with the values in δ at the same places (registers, memory locations, etc). We say that a prediction oracle O has *speculative window at most* w if the length of the transactions generated by its predictions is at most w, i.e., for all programs p, histories h, and configurations σ, $O(p, h, \sigma) = \langle \delta, w', \tau \rangle$, for some δ, τ such that $w' \leq w$.

Taking into account the prediction history enables us to capture history-based predictors, a general class of predictors that base their decisions on past behaviors. Formally, a *prediction history* is a sequence of triples $\langle \ell, id, \delta \rangle$, where $\ell \in Vals$ is the label of an instruction related to speculation, $id \in \mathbb{N}$ is the unique identifier of the transaction that triggered speculation, and δ are the predicted values.

Example 2 (Control-flow Prediction)**.** The "backward taken forward not taken" (BTFNT) branch predictor, implemented in early CPUs [40], predicts the branch as taken if the target instruction address is lower than the program counter. It can be formalized by the *BTFNT* oracle below, for a fixed speculative window w, as follows:

$$BTFNT(p, h, \sigma) = \langle [\mathbf{pc} \mapsto \ell''], w, \mathsf{pc}\ \ell'' \rangle, \text{ where } \ell'' = min(\sigma(\mathbf{pc}) + 1, \ell') \text{ and } p(\sigma(\mathbf{pc})) = \mathbf{beqz}\ x, \ell'$$

BTFNT speculatively updates the program counter to the minimum of the branch target ℓ' and the branch-not-taken target (i.e., $\sigma(\mathbf{pc}) + 1$). Note that the *BTFNT* oracle is static, that is, it is independent of the prediction history h. Dynamic branch predictors, such as simple 2-bit predictors and more complex correlating or tournament predictors [40], can also be formalized using prediction oracles.

Example 3 (Data-flow Prediction)**.** Oleksenko et al. [61] discovered that certain Intel CPUs speculate on division operations, called Zero Dividend Injection (ZDI), thereby resulting in a restricted form of value prediction. We model this behavior with the following oracle, which predicts that the upper bits of a 128-bit by 64-bit division operation are 0:

$$ZDI(p, h, \sigma) = \langle \delta, w, \epsilon \rangle, \text{ where } p(\sigma(\mathbf{pc})) = k \leftarrow (x : y) \div z \text{ and } \delta = [k \mapsto (0 : y) \div z].$$

Above, $(x : y)$ denotes the 128-bit value obtained by concatenating registers x and y whereas $\div$ is the division operator.

4.3 Oracle Semantics

Next, we formalize the speculative oracle semantics. We start by formalizing the states over which the semantics operates and continue by presenting the evaluation relation given a prediction oracle O.

Speculative States. The speculative oracle semantics operates on speculative states X_x, each one consisting of a stack of speculative instances Ψ_x. The state is a stack for two reasons. First, speculation can be nested: a new (mis)prediction may occur while the processor is already executing speculatively, pushing a further transaction on top of the current one. Second, the stack records, for each ongoing transaction, the architectural state from which that speculation started, so that the effects of a mispredicted transaction can be discarded and execution resumed from that state upon rollback

Each instance Ψ_x contains the program p, a counter ctr that uniquely identifies the speculative transaction associated with the instance, a configuration σ, the prediction history h, and the remaining speculation window n describing the number of instructions that can still be executed speculatively (or $\perp$ when no speculation is happening). Depending on the specific speculation that is modelled, additional data might be tracked (indicated with $\cdots$ below as well as in the evaluation rules).

$$\textit{Spec. States } X_x ::= \overline{\Psi}_x \qquad\qquad \textit{Spec. Instances } \Psi_x ::= \langle p, ctr, \sigma, h, n, \cdots \rangle$$

Evaluation rules. We describe the speculative oracle semantics, given a prediction oracle O_x, with the relation $\rightsquigarrow_x^{O_x} \subseteq X_x \times Obs^* \times X_x$, which describes how speculative states are modified along the computation while producing observations capturing leaks. To denote the start and end of speculative transactions, we extend the set of observations Obs to include three new observations $\texttt{start}_x\ n$, $\texttt{rlb}_x\ n$, and $\texttt{commit}_x\ n$ marking the start and end (with a commit or a rollback) of speculative transaction with id n respectively.

$$Obs ::= \cdots \mid \texttt{start}_x\ n \mid \texttt{rlb}_x\ n \mid \texttt{commit}_x\ n$$

Next, we describe in detail the rules formalizing the oracle semantics, which capture the intuition from Section 4.1 and are depicted in Figure 5. Whenever none of the speculative instances in the current state has a speculative window of 0 or is stuck (denoted by $canStep(\overline{\Psi}_x)$), the computation proceeds by doing one step for the instance at the top of the stack (Rule O-Context) and by updating the speculative window of the other instances. In particular, if the instruction to be executed in the topmost instance is a speculative barrier, then all speculative windows are set to 0 ($\overline{\Psi}_x{}'' = zeroes(\overline{\Psi}_x)$), otherwise all windows are decremented by 1 ($\overline{\Psi}_x{}'' = decr(\overline{\Psi}_x)$).

In the topmost speculative instance, instructions that do not trigger speculation (i.e., they are not in $instr_{\text{🕵}}$) are executed following the non-speculative semantics while also decrementing the current speculative window by 1 (Rule O-NoSpec). Speculation barriers are handled similarly except for setting the speculative window in the topmost instance to 0 (Rule O-barr). Finally, Rule O-Spec handles instructions that trigger speculation (i.e., those that belong to $instr_{\text{🕵}}$). In this case, the oracle is queried to obtain the prediction δ and the next speculative window w, and a new speculative instance (starting from the predicted state $\sigma \uplus \delta$ and with an updated prediction history h') is pushed on top of the stack to start the new speculative transaction. This fresh speculative instance is decorated with the original program state σ and with the predicted values δ, which will later on be used to determine whether the transaction should be committed or rolled back. The rule also appends the observation $\texttt{start}_x\ n$ to the trace to record the beginning of a speculative transaction.

A speculative transaction must be finalized (committed or rolled back) whenever its speculation window reaches 0 or the execution cannot proceed further. We refer to the latter case as the instance being 'stuck', formally indicated by

$$\boxed{\overline{\Psi}_x \overset{\tau}{\rightsquigarrow}_x^{O_x} \overline{\Psi}'_x}$$

(O-Context)

$$\frac{\Psi_x \overset{\tau}{\rightsquigarrow}_x^{O_x} \overline{\Psi}'_x \qquad canStep(\overline{\Psi}_x \cdot \Psi_x) \qquad \text{if } p(\Psi_x.\sigma(\mathbf{pc})) = \mathbf{spbarr} \text{ then } \overline{\Psi}_x'' = zeroes(\overline{\Psi}_x) \text{ else } \overline{\Psi}_x'' = decr(\overline{\Psi}_x)}{\overline{\Psi}_x \cdot \Psi_x \overset{\tau}{\rightsquigarrow}_x^{O_x} \overline{\Psi}_x'' \cdot \overline{\Psi}'_x}$$

(O-Rollback)

$$\frac{\langle p, \sigma'' \rangle \xrightarrow{\tau} \langle p, \sigma''' \rangle \qquad (\sigma''' \neq \sigma'' \uplus \delta \text{ or } p, \sigma' \vdash \text{fin})}{\overline{\Psi}_x \cdot \langle p, ctr, \sigma, h, n, \cdots \rangle \cdot \langle p, ctr', \sigma', h', 0, \cdots \rangle^{\langle \sigma'', \delta \rangle} \cdot \overline{\Psi}_x{}' \overset{\mathtt{rlb}_x\ ctr \cdot \tau}{\rightsquigarrow}_x^{O_x} \overline{\Psi}_x \cdot \langle p, ctr', \sigma''', h', n-1, \cdots \rangle}$$

(O-Commit)

$$\frac{\langle p, \sigma'' \rangle \xrightarrow{\tau} \langle p, \sigma''' \rangle \qquad (\sigma''' = \sigma'' \uplus \delta \text{ or } p, \sigma' \vdash \text{fin})}{\overline{\Psi}_x \cdot \langle p, ctr, \sigma, h, n, \cdots \rangle \cdot \langle p, ctr', \sigma', h', 0, \cdots \rangle^{\langle \sigma'', \delta \rangle} \cdot \overline{\Psi}_x{}' \overset{\mathtt{commit}_x\ ctr}{\rightsquigarrow}_x^{O_x} \overline{\Psi}_x \cdot \langle p, ctr', \sigma', h', n, \cdots \rangle \cdot \overline{\Psi}_x{}'}$$

$$\boxed{\Psi_x \overset{\tau}{\rightsquigarrow}_x^{O_x} \overline{\Psi}'_x}$$

(O-NoSpec)

$$\frac{p(\sigma(\mathbf{pc})) \notin instr_{\text{👻}} \cup \{\mathbf{spbarr}\} \qquad \langle p, \sigma \rangle \xrightarrow{\tau} \langle p, \sigma' \rangle}{\langle p, ctr, \sigma, h, n+1, \cdots \rangle \overset{\tau}{\rightsquigarrow}_x^{O_x} \langle p, ctr, \sigma', h, n, \cdots \rangle}$$

(O-barr)

$$\frac{p(\sigma(\mathbf{pc})) = \mathbf{spbarr} \qquad \sigma \xrightarrow{\tau} \sigma'}{\langle p, ctr, \sigma, h, n, \cdots \rangle \overset{\tau}{\rightsquigarrow}_x^{O_x} \langle p, ctr, \sigma', h, 0, \cdots \rangle}$$

(O-Spec)

$$\frac{\begin{array}{c} p(\sigma(\mathbf{pc})) \in instr_{\text{👻}} \qquad O(p, n, h, \sigma) = \langle \delta, w, \tau \rangle \\ h' = h \cdot \langle \sigma(\mathbf{pc}), n+1, \delta \rangle \qquad \overline{\tau} = \mathtt{start}_x\ ctr \cdot \tau \end{array}}{\langle p, ctr, \sigma, h, n+1, \cdots \rangle \overset{\overline{\tau}}{\rightsquigarrow}_x^{O_x} \langle p, ctr, \sigma, h, n+1, \cdots \rangle \cdot \langle p, ctr+1, \sigma \uplus \delta, h', w, \cdots \rangle^{\langle \sigma, \delta \rangle}}$$

Fig. 5. Evaluation rules for the speculative oracle semantics $\rightsquigarrow_x^{O_x}$ given an oracle O_x

the judgment $p, \sigma' \vdash \text{fin}$. This judgment holds whenever the current instruction is undefined, i.e., $p(sigma(\mathbf{pc})) = \bot$ (see Rule Terminate). To determine whether to commit or rollback, the semantics inspects the snapshot state σ and prediction δ that decorate the instance $\Psi_x^{\langle \sigma, \delta \rangle}$ whose window has reached 0.

- If the prediction was *correct* (i.e., doing one step from σ results indeed in the predicted state $\sigma \uplus \delta$), the transaction is committed and the computation continues using the current configuration (Rule O-Commit).
- If the prediction was *incorrect* (i.e., doing one step from σ produces a state different from the predicted state $\sigma \uplus \delta$), the transaction is aborted and the computation continues on the correct branch (Rule O-Rollback).

Thus, committing and rolling back speculative transactions happen along the stack of states. Rolling back deletes all the instances above the rolled back instance, whereas committing updates the configuration, the counter, the prediction history h and additional data tracked by the semantics of the instance below and the committed instance is deleted. Rule O-Commit and Rule O-Rollback also record the end of speculative transactions on the trace with the $\mathtt{commit}_x\ n$ and $\mathtt{rlb}_x\ n$ observations respectively.

Analogously to the non-speculative case, the behaviour $Beh_x^O(p)$ of a program p under the oracle semantics is the set of all traces generated from an initial state until termination.

Non-Speculative Consistency. Fundamentally, a valid speculative semantics must preserve the program's architectural behaviour. Speculation should only introduce observational side-effects during transient execution, without altering

the program's non-speculative behaviour. We formalize this correctness requirement in Property 1, stating that the standard non-speculative behaviour can be exactly recovered from the oracle behaviour by applying the non-speculative projection $\restriction_{ns}$, which (1) removes from the trace τ any sub-trace enclosed between $\texttt{start}_x\ n$ and $\texttt{rlb}_x\ n$ for some n, and then (2) drops all remaining $\texttt{start}_x\ n$ and $\texttt{commit}_x\ n$ observations. In Section 6, we prove that all our speculative semantics instances satisfy Property 1.

Property 1 (NS Consistency Oracle). $Beh_{NS}(p) = Beh^{\mathcal{O}}_x(p)\restriction_{ns}$

5 SPECULATIVE NON-INTERFERENCE

In this section, we introduce *speculative non-interference* (SNI, Section 5.1), a semantic notion characterizing the leaks introduced by speculatively-executed instructions. Next, we introduce the notion of *always-mispredict speculative semantics* (Section 5.2), that is, a speculative semantics that facilitates reasoning about leaks w.r.t. *any* prediction oracle.

5.1 Speculative Non-Interference

Speculative non-interference (SNI) is a semantic notion of security characterizing those information leaks that are introduced by speculative execution. Intuitively, SNI requires that programs do not leak more information under the speculative semantics than what is leaked under the non-speculative semantics.

SNI is parametric in a policy ϕ and in the speculative semantics x, which models how the program executes. The policy ϕ describes which parts of the program are public/low information, i.e., those that are known by an adversary. Formally, a security policy P is a finite subset of $Regs \cup \mathbb{N}$ specifying the low register identifiers and memory addresses. Two configurations σ, σ' are called *low-equivalent* for a policy ϕ, written $\sigma \backsim_\phi \sigma'$, if they agree on all register and memory locations in ϕ.

Example 4 (Example Policy for Example 1). A policy P for the program from Example 1 may state that the content of the registers `y`, `size`, `A`, and `B` is non-sensitive, i.e., $P = \{\texttt{y}, \texttt{size}, \texttt{A}, \texttt{B}\}$.

Policies need not be manually specified but can in principle be inferred from the context in which a piece of code executes, e.g., whether a variable is reachable from public input or not.

A program p satisfies SNI (Definition 1) for a speculative semantics x if any pair of low-equivalent initial configurations σ and σ' that generate the same observations under the non-speculative semantics also generate the same observations under the speculative semantics.

Definition 1 (Speculative Non-Interference). *Program p satisfies SNI (denoted $p \vdash_x SNI$) for a speculative semantics x if for all σ, σ', if $\sigma \backsim_\phi \sigma'$ and $Beh_{NS}(p, \sigma) = Beh_{NS}(p, \sigma')$ then $Beh_x(p, \sigma) = Beh_x(p, \sigma')$.*

SNI is parametric in the speculative semantics x. For the oracle semantics, which additionally depends on a prediction oracle $\mathcal{O}$, we make the oracle explicit and write $p \vdash^{\mathcal{O}}_x$ SNI. The always-mispredict semantics, introduced in Section 5.2, does not depend on an oracle; for it we keep the plain notation $p \vdash_x$ SNI

Speculative non-interference is a variant of non-interference. While non-interference compares what is leaked by a program with a policy specifying the allowed leaks, speculative non-interference compares the program leakage under two semantics, the non-speculative and the speculative one. The security policy and the non-speculative semantics, together, specify what the program may leak under the speculative semantics.[3]

[3] Conceptually, the non-speculative semantics induces declassification assertions for the speculative semantics [66].

Example 5 (SNI for Example 1)**.** The program p from Example 1 does not satisfy speculative non-interference for the BTFNT oracle from Example 2 and the policy P from Example 4. Consider two initial configurations $\sigma := \langle m, a\rangle$, $\sigma' := \langle m', a'\rangle$ that agree on the values of y, `size`, A, and B but disagree on the value of B[A[y] ∗ 512]. Say, for instance, that $m(a(\mathtt{A}) + a(\mathtt{y})) = 0$ and $m'(a'(\mathtt{A}) + a'(\mathtt{y})) = 1$. Additionally, assume that $\mathtt{y} \geq \mathtt{size}$.

Executing the program under the non-speculative semantics produces the trace $\mathtt{pc}\ \bot$ when starting from σ and σ'. Moreover, the two initial configurations are indistinguishable with respect to the policy P. However, executing p under the speculative semantics produces two distinct traces:

$$\tau = \mathtt{start}\ 0 \cdot \mathtt{pc}\ 3 \cdot \mathtt{load}\ v_1 \cdot \mathtt{load}\ (a'(\mathtt{B}) + 0) \cdot \mathtt{rlb}\ 0 \cdot \mathtt{pc}\ \bot$$

$$\tau' = \mathtt{start}\ 0 \cdot \mathtt{pc}\ 3 \cdot \mathtt{load}\ v_1 \cdot \mathtt{load}\ (a'(\mathtt{B}) + 1) \cdot \mathtt{rlb}\ 0 \cdot \mathtt{pc}\ \bot$$

where $v_1 = a(\mathtt{A}) + a(\mathtt{y}) = a'(\mathtt{A}) + a'(\mathtt{y})$. Therefore, p does not satisfy speculative non-interference.

Note that $p \vdash^{O}_{x}$ SNI requires a specific oracle. We restrict attention to oracles whose predictions do not themselves depend on secrets.

Definition 2 (Non-Interfering oracle)**.** *A prediction oracle O is non-interfering w.r.t. a policy ϕ if, for every program p and history h and for all configurations $\sigma \backsim_\phi \sigma'$, $O(p, h, \sigma) = O(p, h, \sigma')$.*

Intuitively, a non-interfering oracle bases its predictions only on the public part of the configuration. This costs no generality in the speculation mechanisms we can model: prediction in real hardware is driven by the program's execution history and by microarchitectural state, never by secret architectural data. The only oracles excluded are physically unrealizable ones that inspect secrets to decide whether or how to speculate. Such an oracle would itself be a secret-dependent control-flow channel. For the remainder of the paper we therefore assume that all prediction oracles are non-interfering.

5.2 Always-Mispredict (AM) Semantics

The oracle speculative semantics from Section 4.3 and, as a result, SNI are parametric in the prediction oracle O. Often, however, it is desirable to obtain security guarantees w.r.t *any* prediction oracle, since the details about speculation mechanisms might differ between different CPUs or may even be unknown. To this end, we introduce a variant of the speculative semantics, which we call the *always-mispredict speculative semantics*, that facilitates reasoning about leaks w.r.t. any prediction oracle. We formally define the AM semantics as a speculative template rather than a concrete mechanism. This template abstracts the core logic of exploring mispredicted paths and serves as a unified foundation for the specific instantiations defined in Section 6.

For simplicity, consider the case of branch prediction. In this case, leakage due to speculative execution is maximized under a predictor that mispredicts every time. This intuition holds true unless speculative transactions are nested, where a correct prediction of a nested branch sometimes yields more leakage than a misprediction.

Example 6. Consider the following variation of the Spectre-Pht example [49] from Figure 1, and assume that the function `benign()` runs for longer than the speculative window and does not leak any information.

Then, under a branch predictor that mispredicts every branch, the speculative transaction corresponding to the outer branch will be rolled back before reaching line 4. On the other hand, given a correct prediction of the inner branch, line 4 would be reached and a speculative leak would be present.

$$\boxed{\overline{\Phi}_x \overset{\tau}{\dashrightarrow}_x \overline{\Phi}'_x}$$

$$\frac{p(\sigma(\mathbf{pc})) \notin \mathit{instr}_{\text{👻}} \cup \{\mathbf{spbarr}\} \quad \langle p, \sigma\rangle \xrightarrow{\tau} \langle p, \sigma'\rangle}{\overline{\Phi}_x \cdot \langle p, ctr, \sigma, n+1, \cdots\rangle \overset{\tau}{\dashrightarrow}_x \overline{\Phi}_x \cdot \langle p, ctr, \sigma', n, \cdots\rangle}\ \text{(AM-NoSpec)}$$

$$\frac{p(\sigma(\mathbf{pc})) = \mathbf{spbarr} \quad \sigma \xrightarrow{\tau} \sigma'}{\overline{\Phi}_x \cdot \langle p, ctr, \sigma, n, \cdots\rangle \overset{\tau}{\dashrightarrow}_x \overline{\Phi}_x \cdot \langle p, ctr, \sigma', 0, \cdots\rangle}\ \text{(AM-barr)}$$

$$\frac{p(\sigma(\mathbf{pc})) = \mathit{instr}_{\text{👻}} \quad \langle p, \sigma\rangle \xrightarrow{\tau} \langle p, \sigma'\rangle \quad j = min(\omega, n) \quad \overline{\tau} = \tau \cdot \mathtt{start}_x\ ctr \cdot \cdots}{\overline{\Phi}_x \cdot \langle p, ctr, \sigma, n+1\rangle \overset{\overline{\tau}}{\dashrightarrow}_x \overline{\Phi}_x \cdot \langle p, ctr, \sigma', n\rangle \cdot \Phi'_x}\ \text{(AM-Spec)}$$

$$\frac{\Phi'_x.n = 0 \text{ or } p, \Phi'_x.\sigma \vdash \text{fin}}{\overline{\Phi}_x \cdot \langle p, ctr, \sigma, n\rangle \cdot \Phi'_x \overset{\mathtt{rlb}_x\ ctr}{\dashrightarrow}_x \overline{\Phi}_x \cdot \langle p, \Phi'_x.ctr', \sigma, n\rangle}\ \text{(AM-Rollback)}$$

Fig. 6. Always-mispredict speculative semantics

```
1  if (y < size)
2     if (y-1 < size)
3        benign();
4     temp &= B[A[y] * 512];
```

A simple but inefficient approach to deal with this challenge would be to consider both cases, correct and incorrect predictions, upon every branch. This, however, would result in an exponential explosion of the number of paths to consider. Furthermore, it would not be applicable to speculation mechanisms where there might be more than one incorrect prediction, e.g., indirect branch speculation or value speculation.

Intuition. To address these issues, we introduce the *always-mispredict speculative semantics* that differs from the oracle speculative semantics in three key ways:

(1) It mispredicts every time, hence its name. For every instruction that might trigger speculation (i.e., the instruction belongs to $\mathit{instr}_{\text{👻}}$), the always-mispredict semantics first speculatively explores all possible *wrong* paths for a bounded number of steps and then continues with the correct one. Thus, the semantics is not parametric in the prediction oracle.

(2) It initializes the length of every *non-nested* transaction to w, and the length of every *nested* transaction to the remaining length of its enclosing transaction, decremented by 1.

(3) Upon executing instructions, only the remaining length of the innermost transaction is decremented.

The consequence of these modifications is that nested transactions do not reduce the number of steps that the semantics may explore the correct path for, after the nested transactions have been rolled back. In Example 6, after rolling back the nested speculative transaction, the outer transaction continues as if the nested branch had been correctly predicted in the first place, and thus the speculative leak in line 4 is reached.

Formalization. The state Σ_x of the AM semantics is a stack of speculative instances Φ_x. As shown in all rules in Figure 6, reductions in the always-mispredict semantics happen only on top of the stack. Each instance Φ_x contains the program p, a counter ctr that identifies the speculative transaction , a configuration σ, and the remaining speculation window n describing the number of instructions that can still be executed speculatively (or $\perp$ when no speculation is happening). Depending on the specific speculation that is modelled, additional data is tracked (indicated with $\cdots$

in the rules). Throughout the paper, we fix the maximal speculation window, i.e., the maximum number of speculative instructions, to a global constant ω.

$$\textit{Spec. States } \Sigma_x ::= \overline{\Phi}_x \qquad\qquad \textit{Spec. Instances } \Phi_x ::= \langle p, ctr, \sigma, n, \cdots \rangle$$

Modifications (1)–(3) are captured in the four rules given in Figure 6. The judgement for the AM semantics is: $\Sigma_x \overset{\tau}{\Longrightarrow}_{\mathcal{J}_x} \Sigma'_x$. If the instruction is not related to speculation and it is not a speculation barrier, then the speculative instance on the top of the stack is updated according to the non-speculative semantics $\to$ (Rule AM-NoSpec). In contrast, whenever the current instruction is a speculation barrier **spbarr**, the remaining speculation window of the topmost instance is set to 0.

Whenever speculation starts, one or more speculative instances are pushed on top of the stack (Rule AM-Spec) and when speculation ends, the speculative instance is then popped (Rule AM-Rollback). Finally, we note that how exactly the new speculative instances are created w.r.t. the auxiliary data depends on the specific speculative semantics, as we show in Section 6. This is reflected in Rule AM-Spec not specifying how the new speculative instances $\overline{\Phi}'_x$ are defined. In all cases, however, the new instances $\overline{\Phi}'_x$ are pushed with remaining speculation window $j = min(\omega, n)$; only the auxiliary data is instance-specific.

The always-mispredict behaviour $Beh^{\omega}_x(p)$ of a program p is the set of all traces generated from an initial state until termination using the reflexive-transitive closure of $\Longrightarrow_{\mathcal{J}_x}$. Crucially, it is possible to connect the AM semantics and the non-speculative semantics through the non-speculative projection $\restriction_{ns}$, which removes from a trace $\overline{\tau}$ all events related to speculation. We lift the projection $\restriction_{ns}$ to the behaviour of a program p in the natural way.

We require that any instantiation of this template (e.g., for branches or stores) preserves the non-speculative behaviour of the program (Similar to the Oracle Semantics in Section 4.3). We formalize this as a correctness requirement that all our instances must satisfy:

Property 2 (NS Consistency AM)**.** $Beh_{NS}(p) = Beh^{\omega}_x(p)\restriction_{ns}$

The goal of the AM semantics is to derive security guarantees that are independent of the choice of a prediction oracle. Property 3 formalizes this intuition by precisely connecting the oracle and the AM semantics of any speculative semantics $\mathcal{J}_x$. In particular, Property 3 states that checking SNI w.r.t. the AM semantics is sufficient to obtain security guarantees w.r.t. *all* prediction oracles. That is, if a program is SNI w.r.t. the always-mispredict semantics, then it is SNI irrespectively of the choice of prediction oracle. Similarly, if a program violates SNI w.r.t. the always-mispredict semantics, then there is one prediction oracle for which SNI is violated. As we show in Section 6, this holds for all instances studied in this paper.

Property 3 (Oracle Overapproximation)**.** $p \vdash_x \textit{SNI iff } \forall O.\, p \vdash^{O}_{x} \textit{SNI}$

6 INSTANCES OF SPECULATIVE SEMANTICS

Here we present specific instances of speculative semantics modelling the effect of speculative execution over branch instructions (Section 6.1), **store** instructions (Section 6.2), **ret** instructions (Section 6.3, Section 6.4) and indirect jump instructions (Section 6.5) using the semantics templates presented in Section 4.

Before detailing these specific instances, we must establish the properties that well-formed semantics must satisfy in our framework. We summarise these properties in a single definition:

Definition 3 (Well-Formed Speculative Semantics)**.** *A speculative semantics $\mathcal{J}_x$ is well-formed (denoted $\vdash \mathcal{J}_x$ WFSS) if:*

- *Oracle Overapproximation:* $p \vdash_x$ *SNI iff* $\forall O.\ p \vdash_x^O$ *SNI*
- *NS Consistency:* $Beh_x^\omega(p)\restriction_{ns} = Beh_{NS}(p) = Beh_x^O(p)\restriction_{ns}$
- *Symbolic Consistency:* $Beh_x^\omega(p) = \mu(Beh_x^S(p))$

Intuitively, a well-formed speculative semantics is made of three components: an AM semantics, an oracle semantics, and a symbolic AM semantics.

First, the AM semantics must overapproximate the oracle semantics (for any oracle), guaranteeing that it is sufficient to check a program p for SNI w.r.t. the AM semantics. Next, both the AM and the Oracle semantics must preserve the non-speculative behaviour of a program p. Applying the non-speculative projection ($\restriction_{ns}$) to their traces exactly recovers the standard non-speculative behaviour. Thus, we can execute p only once to get the (non-)speculative behaviour of that program run.

Finally, to enable automated verification, we require the Symbolic AM semantics. Both the symbolic semantics and the concretization function μ (which maps symbolic values to concrete ones) are defined in Section 9, as they are used only by our analysis tool. The Symbolic Consistency property mandates that the concrete AM traces exactly match the concretization of the symbolic traces, where $\mu(Beh_x^S(p))$ conceptually denotes the instantiation of the symbolic traces with concretizations that satisfy the collected path conditions.

For conciseness' sake, here we only present the concrete AM semantics in full detail and refer to the technical report for the full details of the oracle/symbolic semantics [30], and defer the detailed discussion of how the symbolic semantics is utilized in practice to the implementation of Spectector in Section 9. For each of the speculative semantics, we prove that they satisfy all validity requirements, that is, that they are well-formed speculative semantics according to Definition 3.

6.1 $\not\!\!\Box_B$: Speculation on Branch Instructions

Modern hardware uses a branch predictor to predict the outcome of branching decisions since this speeds up program execution. However, mispredictions can be steered and exploited by attackers leading to Spectre-PHT attacks [49].

6.1.1 The AM Semantics. At every branch instruction, the always-mispredict semantics first speculatively executes the wrong branch for a fixed number of steps and then continues with the correct one. As a result, this semantics is deterministic and agnostic to implementation details of the branch predictor.

The state Σ_B of the AM semantics is a stack of speculative instances Φ_B where reductions happen only on top of the stack. Each instance Φ_B contains the program p, a counter ctr that uniquely identifies the speculation instance, a configuration σ, and the remaining speculation window n describing the number of instructions that can still be executed speculatively (or $\bot$ when no speculation is happening). In this semantics, we have $instr_{\not\!\!\Box} =$ **beqz** since branches are the source of speculation.

$$\textit{Spec. States } \Sigma_B ::= \overline{\Phi}_B \qquad\qquad \textit{Spec. Instances } \Phi_B ::= \langle p, ctr, \sigma, n\rangle$$

The judgement for the AM semantics is: $\Sigma_B \overset{\tau}{\dashrightarrow}_{\not\!\!\Box_B} \Sigma'_B$.

$$\boxed{\overline{\Phi}_B \overset{\tau}{\dashrightarrow}_{\not\!\!\Box_B} \overline{\Phi}'_B}$$

$$\frac{p(\sigma(\mathbf{pc})) \notin \mathbf{beqz} \cup Z_B \cup \{\mathbf{spbarr}\} \quad \langle p, \sigma\rangle \xrightarrow{\tau} \langle p, \sigma'\rangle}{\langle p, ctr, \sigma, n+1\rangle \overset{\tau}{\rightsquigarrow}_B \langle p, ctr, \sigma', n\rangle} \text{(B:AM-NoSpec)}$$

$$\frac{p(\sigma(\mathbf{pc})) = \mathbf{spbarr} \quad \sigma \xrightarrow{\tau} \sigma'}{\langle p, ctr, \sigma, n\rangle \overset{\tau}{\rightsquigarrow}_B \langle p, ctr, \sigma', 0\rangle} \text{(B:AM-Barr)}$$

$$\frac{n' = 0 \text{ or p is stuck}}{\langle p, ctr, \sigma, n\rangle \cdot \langle p, ctr', \sigma', n'\rangle \overset{\mathtt{rlb}_B\ ctr}{\rightsquigarrow}_B \langle p, ctr', \sigma, n\rangle} \text{(B:AM-Rollback)}$$

$$\frac{\begin{array}{c} p(\sigma(\mathbf{pc})) = \mathbf{beqz}\ x, \ell \quad \langle p, \sigma\rangle \xrightarrow{\tau} \langle p, \sigma'\rangle \quad j = min(\omega, n) \\ \sigma'' = \sigma[\mathbf{pc} \mapsto l'] \qquad \overline{\tau} = \tau \cdot \mathtt{start}_B\ ctr \cdot \mathtt{pc}\ l \\ l' = \begin{cases} \sigma(\mathbf{pc}) + 1 & \text{if } \sigma'(\mathbf{pc}) = l \\ l & \text{if } \sigma'(\mathbf{pc}) \neq l \end{cases} \end{array}}{\langle p, ctr, \sigma, n+1\rangle \overset{\overline{\tau}}{\rightsquigarrow}_B \langle p, ctr, \sigma', n\rangle \cdot \langle p, ctr+1, \sigma'', j\rangle} \text{(B:AM-Spec)}$$

As mentioned, Rule B:AM-Spec pushes a new speculative state with the wrong branch, followed by the state with the correct one. When speculation ends, Rule B:AM-Rollback pops the related state. All other instructions are handled by delegating back to the non-speculative semantics (Rule B:AM-NoSpec).

Rule B:AM-NoSpec differs slightly from what was presented before in Section 5.2: it applies to instructions that are not branch or barrier instructions **and** are not in the metaparameter Z_B (in gray). The latter is a set of instructions and is part of our composition framework (which we explain in Section 7.1). Instantiating Z_B allows us to restrict when to apply non-speculative steps in composed semantics. When we consider $\rightsquigarrow_B$ in isolation, Z_B is the empty set (so, Rule B:AM-NoSpec applies to everything except branch and barrier instructions). However, we will instantiate Z_B in different manners when building the composed semantics. In the following, we write $\rightsquigarrow_B^{Z_B}$ to stress the value of Z_B when needed but we often omit Z_B for simplicity.

The always-mispredict behaviour $Beh_B^{\omega}(p)$ of a program p is the set of all traces generated from an initial state until termination using the reflexive-transitive closure of $\overset{\tau}{\rightsquigarrow}_B$.

Note that Rule B:AM-NoSpec, Rule B:AM-Barr and Rule B:AM-Rollback are direct instantiations of the general rules described in Section 5.2. This holds true for all the other speculative semantics that we will present, thus, we omit them.

6.1.2 Oracle Semantics. At every branch instruction, the oracle semantics queries the explicit oracle O_B for the branching decision.

Here, we summarize the key differences with the AM semantics. First, speculative instances are extended to track the branching history h, which records the outcomes of prior branch instructions. Second, when executing a **beqz** instruction, the oracle predicts the branch outcome (based on the branching history h) and a new speculative instance is pushed on top of the stack (Rule B:Spec). Finally, whenever the speculation window of an instance anywhere on the stack reaches 0, the execution needs to be rolled back or committed.

$$\Psi_B \overset{\tau}{\rightsquigarrow}_B^{O} \Psi'_B$$

$$\frac{p(\sigma(\mathbf{pc})) \notin \cup\, Z_{\text{B}} \qquad \sigma \xrightarrow{\tau} \sigma'}{\langle p, ctr, \sigma, h, n+1\rangle \overset{\tau}{\rightsquigarrow}_{\text{B}}^{O} \langle p, ctr, \sigma', h, n\rangle} \text{(B:NoBranch)}$$

$$\frac{\begin{array}{c} p(\sigma(\mathbf{pc})) = \mathbf{beqz}\ x, \ell \qquad \sigma \xrightarrow{\tau} \sigma' \qquad O(p, n, h, \sigma) = (\delta, \omega, \texttt{pc}\ \ell'') \\ \delta = [\mathbf{pc} \mapsto l''] \qquad\qquad h' = h \cdot \langle \sigma(\mathbf{pc}), ctr, \delta\rangle \\ \overline{\tau} = \tau \cdot \texttt{start}_{\text{B}}\ ctr \cdot \texttt{pc}\ l'' \end{array}}{\langle p, ctr, \sigma, h, n+1\rangle \overset{\overline{\tau}}{\rightsquigarrow}_{\text{B}}^{O} \langle p, ctr, \sigma', h', n\rangle \cdot \langle p, ctr+1, \sigma \uplus \delta, h', \omega\rangle^{\langle \sigma, \delta\rangle}} \text{(B:Spec)}$$

As in the oracle-semantics template, rolling back deletes all the instances above the rolled back instance, whereas committing updates the configuration, the counter and the branching history h of the instance below and the committed instance is deleted. These rules are not shown since they are direct instantiations of the general oracle rules presented in Section 4.3.

6.2 $\lightning_{\text{S}}$: Speculation on Store Instructions

Modern processors write **store**s to main memory asynchronously to reduce delays caused by the memory subsystem. Processors employ a *Store Queue* where not-yet-committed **store** instructions are stored before being permanently written to memory. When executing a **load** instruction, the processor first inspects the store queue for a matching memory address. If there is a match, the value is retrieved from the store queue (called *store-to-load forwarding*), and otherwise, the memory request is issued to the memory subsystem. To speed up computation, processors employ memory disambiguation predictors to predict if the memory addresses of loads and stores match. Since the prediction can be incorrect, processors may speculatively bypass a **store** instruction in the store queue, leading to a **load** instruction retrieving a stale value [43].

Example 7 (Spectre-STL)**.** Consider the code in Listing 6, which is the μAsm translation of Listing 2:

Listing 6. Spectre-STL in μAsm.

```
1 store secret, p
2 store public, p
3 load eax, p
4 load edx, B + eax
```

Assume that the **store** instructions in Lines 1 and 2 are still in the *store queue* and not yet committed to main memory. A misprediction of the memory disambiguator for the **load** instruction in Line 3 causes it to bypass the **store** instruction in Line 2 and retrieve the value from the stale **store** instruction in Line 1. The speculative access of the memory is then leaked into the microarchitectural state by the array access into B in Line 4.

Here, we present the speculative AM semantics (Section 6.2.1) and the corresponding oracle semantics (Section 6.2.2).

6.2.1 Speculative Semantics. The overall structure of the $\lightning_{\text{S}}$ semantics is similar to that of $\lightning_{\text{B}}$: speculative execution is modelled using a stack of speculative states, instructions that do not start speculative transactions are executed by delegating back to the non-speculative semantics, and speculative transactions are rolled back whenever the speculative window reaches 0. The key difference between $\lightning_{\text{S}}$ and $\lightning_{\text{B}}$ is the differing source of speculation $instr_{\lightning}$. Here $instr_{\lightning}$ = **store** for $\lightning_{\text{S}}$ instead of $instr_{\lightning}$ = **beqz** as in $\lightning_{\text{B}}$.

The states used in $\lightning_{\text{S}}$ are similar to those of $\lightning_{\text{B}}$:

$$\textit{Spec. States}\ \Sigma_{\text{S}} ::= \overline{\Phi}_{\text{S}} \qquad\qquad \textit{Spec. Instance}\ \Phi_{\text{S}} ::= \langle p, ctr, \sigma, n\rangle$$

We also add a bypass n observation denoting that the **store** instruction at program counter n was speculatively bypassed.

$$Obs_S ::= Obs \mid \texttt{bypass}\ n$$

The judgement $\Sigma_S \overset{\tau}{\dashrightarrow}_S \Sigma'_S$ describes how Σ_S steps to Σ'_S emitting observation τ. As in $\_\!\!\nearrow_B$, reductions only happen on top of the stack.

$$\boxed{\overline{\Phi}_S \overset{\tau}{\dashrightarrow}_S \overline{\Phi}'_S}$$

$$\frac{\begin{array}{c}p(\sigma(\mathbf{pc})) = \mathbf{store}\ x, e \qquad \langle p, \sigma\rangle \xrightarrow{\tau} \langle p, \sigma'\rangle \qquad j = min(\omega, n) \\ \sigma'' = \sigma[\mathbf{pc} \mapsto \sigma(\mathbf{pc}) + 1] \qquad \tau' = \tau \cdot \texttt{bypass}\ \sigma(\mathbf{pc}) \cdot \texttt{start}_S\ ctr\end{array}}{\langle p, ctr, \sigma, n+1\rangle \overset{\tau'}{\dashrightarrow}_S \langle p, ctr, \sigma', n\rangle \cdot \langle p, ctr+1, \sigma'', j\rangle}\ (\text{S:AM-Spec})$$

To model the effect of bypassing a **store** instruction, Rule S:AM-Spec bypasses the **store** instruction by increasing the program counter without updating the memory and starts a new speculative transaction by pushing a new speculative instance on top of the state. A **load** instruction loading from the same memory location as the bypassed **store** instruction, therefore, retrieves a stale value. Our semantics models store-to-load forwarding as all-or-nothing: a **load** reads either the bypassed (stale) value or the stored value. We do not capture partial or size-mismatched forwarding, where only part of a store is forwarded.

The set $Beh^{\omega}_S(p)$ contains all traces generated from an initial state until termination using the reflexive-transitive closure of $\dashrightarrow_S$.

6.2.2 Oracle Semantics. Instead of bypassing **store**, the oracle semantics employs an oracle O that decides if the **store** instruction should be speculatively bypassed or not. As before, the behaviour $Beh^{O}_S(p)$ of a program p is the set of all traces starting from an initial state until termination using the reflexive-transitive closure of the oracle semantics.

6.3 $\_\!\!\nearrow_R$: Speculation on Return Instructions

The return-stack buffer (RSB) is a small stack the CPU uses to save return addresses upon **call** instructions. These saved return addresses are speculatively used when the function returns because accessing the RSB is faster than looking up the return address on the stack (stored in main memory). This works well because return addresses rarely change during function execution. However, mispredictions can be exploited by an attacker [50, 53].

Example 8 (Return Speculation Vulnerability)**.** Consider the example in Listing 7 and recall that register **sp** is used to find return addresses saved on the stack.

Each function call pushes a return address on the stack and decrements the **sp** register. After reaching the function *Manip_Stack*, the **sp** register is incremented (line 2). Thus, **sp** points to the previous return address on the stack (i.e., line 11), and the non-speculative execution continues in *Main* and terminates. However, the return address of the call in line 5 is line 6 and it is on top of the RSB. Thus, the CPU speculatively executes lines 6–7 and leaks the secret.

This section describes the AM semantics (Section 6.3.1) and the oracle semantics (Section 6.3.2), Then, it discusses formalising different implementations of the RSB in the CPU (Section 6.3.3).

6.3.1 Speculative Semantics. Unlike before, the state of $\_\!\!\nearrow_R$ contains a model of the RSB which is used to retrieve return addresses instead of relying on the stack.

Listing 7. A program exploiting RSB speculation.

```
1  Manip_Stack:
2      sp ← sp + 8
3      ret
4  Speculate:
5      call Manip_Stack
6      load eax, secret
7      load edx, eax
8      ret
9  Main:
10     call Speculate
11     skip
```

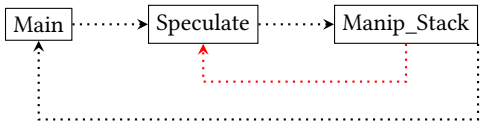

Fig. 7. Control Flow of the vulnerable program in Listing 7. A red arrow indicates a speculative control flow that happens because of misprediction with the RSB.

Thus, speculative instances of $\not\Box_R$ are extended with an additional entry $\mathbb{R}$ for tracking the RSB, whose size is limited by a global constant $\mathbb{R}_{size}$ denoting the maximal RSB size. A speculative instance Φ_R now consists of the program p, the counter ctr, the configuration σ, the speculation window ω, and the RSB $\mathbb{R}$. As before, a state Σ_R is a stack of speculative instances $\overline{\Phi}_R$.

$$\textit{Spec. States } \Sigma_R ::= \overline{\Phi}_R \qquad\qquad \textit{Spec. Instance } \Phi_R ::= \langle p, ctr, \sigma, \mathbb{R}, n\rangle$$

As before, in $\Sigma_R \overset{\tau}{\to}_{\not\Box_R} \Sigma_R$ reductions happen on the top of the stack and in this semantics we have $instr_{\not\Box} = \textbf{call} \cup \textbf{ret}$ since both **call** and **ret** instructions interact with the RSB and **ret** instructions are the source of speculation.

$$\boxed{\overline{\Phi}_R \overset{\tau}{\to}_{\not\Box_R} \overline{\Phi}'_R}$$

(R:AM-Spec)

$$\frac{\begin{array}{ccc} p(\sigma(\mathbf{pc})) = \mathbf{ret} & \sigma = \langle m, a\rangle & \langle p,\sigma\rangle \xrightarrow{\tau} \langle p,\sigma'\rangle \\ \mathbb{R} = \mathbb{R}' \cdot l & j = min(\omega, n) & l \neq m(a(\mathbf{sp})) \\ \sigma'' = \sigma[\mathbf{pc} \mapsto l, \mathbf{sp} \mapsto a(\mathbf{sp}) + 8] & & \overline{\tau} = \tau \cdot \mathtt{start}_R\ ctr \cdot \mathtt{ret}\ l \end{array}}{\langle p, ctr, \sigma, \mathbb{R}, n+1\rangle \overset{\overline{\tau}}{\to}_{\not\Box_R} \langle p, ctr, \sigma', \mathbb{R}', n\rangle \cdot \langle p, ctr+1, \sigma'', \mathbb{R}', j\rangle}$$

(R:AM-Call)

$$\frac{\begin{array}{cc} p(\sigma(\mathbf{pc})) = \mathbf{call}\ f & \langle p,\sigma\rangle \xrightarrow{\tau} \langle p,\sigma'\rangle \\ \mathbb{R}' = \mathbb{R} \cdot \langle a(\mathbf{pc})+1\rangle & |\mathbb{R}| < \mathbb{R}_{size} \end{array}}{\langle p, ctr, \sigma, \mathbb{R}, n+1\rangle \overset{\tau}{\to}_{\not\Box_R} \langle p, ctr, \sigma', \mathbb{R}', n\rangle}$$

During **call** instructions (Rule R:AM-Call), the return address is pushed on top of the RSB (if there is space available) and during **ret** instructions, the return address stored on the RSB is used if the entry on top of the RSB is different from the one stored on the stack (Rule R:AM-Spec). Then, the rule creates a new speculative instance that uses the return address from the RSB $\mathbb{R}$. Note that speculation only happens when the return address from the RSB differs from the one on the stack (stored in $m(a(\mathbf{sp}))$).

Here, we overview how our semantics behaves with empty and full RSB. Whenever the RSB is empty, executing a **ret** instruction does not cause speculation and we return to the address pointed by **sp**. In contrast, whenever the RSB is full, executing a **call** instruction does not add entries to the RSB, i.e., we model an *acyclic* RSB.[4]

[4]We follow the way AMD processors handle this kind of speculation [53].

The behaviour $Beh^{\omega}_{\mathsf{R}}(p)$ is the set of all traces generated from an initial state until termination using $\overset{\tau}{\dashrightarrow}_{\mathsf{R}}$.

6.3.2 Oracle Semantics. Unlike before, the oracle cannot decide the outcome of the **ret** instruction, because the CPU always uses the return address stored in the RSB (if there is one) and it does not speculate otherwise [20]. The only thing the oracle decides here is the size of the speculation window ω.

6.3.3 Different Behaviours of Empty and Full RSBs. Modern CPUs use different RSB implementations that differ in the way they handle underflows and overflows, i.e., when the RSB is empty or full [53]. For example, cyclic RSB implementations overwrite old entries when the RSB is full. Alternatively, CPUs can fallback to other predictors (like the indirect branch predictor) to predict return addresses whenever the RSB is empty.

In our model, the RSB is not cyclic and there is no speculation when the RSB is empty (Rule R:AM-Ret-Empty).

$$\frac{p(\sigma(\mathbf{pc})) = \mathbf{ret} \qquad \langle p, \sigma\rangle \xrightarrow{\tau} \langle p, \sigma'\rangle}{\langle p, ctr, \sigma, \emptyset, n+1\rangle \overset{\tau}{\dashrightarrow}_{\mathsf{R}} \langle p, ctr, \sigma', \emptyset, n\rangle} \text{(R:AM-Ret-Empty)}$$

We remark that extending $\dashrightarrow_{\mathsf{R}}$ to support different RSBs implementations can be done with minimal effort.

6.4 $\dashrightarrow_{\mathsf{SLS}}$: Straightline Speculation

Modern CPUs use the Branch Target Buffer (BTB) to assist with branch prediction. The BTB is indexed by possible jump instructions and predicts the next program counter. However, it also stores information about the type of branch (i.e. no branch, direct branch, indirect branch, or return) encountered at that location. Thus, it can happen that the BTB correctly predicts the location of a branch but mispredicts the type of the branch. For example, a **ret** instruction can be mispredicted to have the type no branch which in turn means that the CPU speculatively executes code right past the **ret** instruction. This kind of speculation is called straight-line speculation (SLS) [5, 8]. Note that a non-branch instruction can also be predicted as a branch which results in ghost jumps. Here, however, we focus only on straight-line speculation.

Example 9 (Straightline Speculation Vulnerability)**.** Consider the example in Listing 8:

Listing 8. Code vulnerable to straightline speculation.

```
1 ret
2 load eax, p
3 load edx, B + eax
```

Assume that p is an attacker-controlled value during the execution. After the execution of the **ret** instruction, the BTB of the processor predicts a no branch for the **ret** instruction. Thus, the processor speculatively bypasses the **ret** instruction and executes the following **load** instruction in Line 2. The speculative access of the memory is then leaked into the microarchitectural state by the array access into B in Line 3.

This section describes the AM semantics (Section 6.4.1) and the oracle semantics (Section 6.4.2).

6.4.1 Speculative Semantics.

$$\overline{\Phi}_{\mathsf{SLS}} \overset{\tau}{\dashrightarrow}_{\mathsf{SLS}} \overline{\Phi}'_{\mathsf{SLS}}$$

(SLS:AM-Spec)

$$\frac{\begin{array}{ccc} p(\sigma(\mathbf{pc})) = \mathbf{ret} & \sigma \xrightarrow{\tau} \sigma' & j = min(\omega, n) \\ \sigma'' = \sigma[\mathbf{pc} \mapsto \sigma(\mathbf{pc}) + 1] & \tau' = \tau \cdot \mathtt{bypass}\ \sigma(\mathbf{pc}) \cdot \mathtt{start}_{\mathsf{SLS}}\ ctr \end{array}}{\langle p, ctr, \sigma, n+1\rangle \overset{\tau'}{\dashrightarrow}_{\mathsf{SLS}} \langle p, ctr, \sigma', n\rangle \cdot \langle p, ctr+1, \sigma'', j\rangle}$$

Judgement $\Sigma_{\mathsf{SLS}} \overset{\tau}{\dashrightarrow}_{\mathsf{SLS}} \Sigma'_{\mathsf{SLS}}$ describes how Σ_{SLS} steps to Σ'_{SLS} emitting observation τ. As in $\mathcal{B}$, reductions only happen on top of the stack. Here we have $instr_{\mathcal{B}} = \mathbf{ret}$ because **ret** instructions are the source of speculation.

To model the effect of bypassing a **ret** instruction, Rule SLS:AM-Spec bypasses the **ret** instruction by increasing the program counter instead of using the return address to update the program counter and starts a new speculative transaction by pushing a new speculative instance on top of the state. This is in contrast to speculation on **ret** in $\mathcal{R}$, which uses the RSB to update the program counter accordingly.

The set $Beh^{\omega}_{\mathsf{SLS}}(p)$ contains all traces generated from an initial state until termination using the reflexive-transitive closure of $\dashrightarrow_{\mathsf{SLS}}$.

6.4.2 Oracle Semantics. Instead of bypassing every **ret** instruction, the oracle semantics employs an oracle O that decides if the **ret** instruction should be speculatively bypassed or not. As before, the behaviour $Beh^{O}_{\mathsf{SLS}}(p)$ of a program p is the set of all traces starting from an initial state until termination using the reflexive-transitive closure of the oracle semantics.

6.5 $\mathcal{J}$: Speculation on (Indirect) Jump Instructions

Modern processors use an indirect branch predictor to predict the outcome of indirect branches. Indirect branches are branches where the outcome is not known until the execution of the program. For example, **jmp** r1, which jumps to the location specified by register r1. Instead of waiting for the value of $r1$ to be available, the processor predicts the jump target based on the branch target buffer. However, attackers can poison the content of the branch target buffer, which allows the attacker to speculatively divert the control flow of the program [49].

Example 10 (A Program Exploiting Jump Speculation)**.** Consider the example in Listing 9 implementing a small jump table. In Line 2 until Line 7 the program assigns the jump target to r1 depending on the value of x. Next, the indirect jump in Line 8 executes and non-speculative execution continues at either J1 or J2. In both cases, execution terminates by jumping to the end in Line 17, By exploiting jump speculation, an attacker could guide the indirect jump in Line 8 to Line 15 thus leaking the contents of the private variable p in Line 16.

This section describes the AM semantics (Section 6.5.1), the oracle semantics (Section 6.5.2).

6.5.1 Speculative Semantics. The attacker can inject any address as the new jump target, making analysis especially tricky. Consider a hypothetical speculation rule capturing the speculative behaviour of indirect jumps (note, this is not the version we use, which is presented below):

(J:AM-Spec (Non-Final-Version))

$$\frac{\begin{array}{cc} p(\sigma(\mathbf{pc})) = \mathbf{jmp}\ e & \sigma \xrightarrow{\tau} \sigma' \\ \exists x \subset e.\, x \in Regs & j = min(\omega, n) \\ S = \{l \mid l \in p \setminus \{\sigma'.\mathbf{pc}\}\} & \langle p, ctr, \sigma, j\rangle \vdash^{S} \overline{\Phi}_{\mathcal{J}} \end{array}}{\langle p, ctr, \sigma, n+1\rangle \overset{\tau}{\dashrightarrow}_{\mathcal{J}} \langle p, ctr, \sigma, n\rangle \cdot \overline{\Phi}_{\mathcal{J}}}$$

Listing 9. Code vulnerable to indirect jump speculation.

```
Main:
    beqz x, L1
    r1 <- J1
    jmp End
L1:
    r1 <- J2
End:
    jmp r1
J1:
    x <- 0
    jmp Fin
J2:
    x <- 1
    jmp Fin
    load eax, p
    load edx, B + eax
Fin:
```

Listing 10. Example addition of endbr instructions.

```
Main:
    beqz x, L1
    r1 <- J1
    jmp End
L1:
    r1 <- J2
End:
    jmp r1
J1:
    endbr
    x <- 0
    jmp Fin
J2:
    endbr
    x <- 1
    jmp Fin
    load eax, p
    load edx, B + p
Fin:
```

Fig. 8. Example program vulnerable to jump speculation (Listing 9) and a version restricting the jump speculation using **endbr** instructions (Listing 10).

$$\frac{}{\langle p, ctr, \sigma, n\rangle \vdash^{\varnothing} \varnothing}\ \text{(Helper-Base)} \qquad \frac{l \in S \quad \langle p, ctr+1, \sigma, n\rangle \vdash^{S\setminus l} \overline{\Phi}_J}{\langle p, ctr, \sigma, n\rangle \vdash^{S} \langle p, ctr+1, \sigma[\mathbf{pc} \mapsto l], n\rangle \cdot \overline{\Phi}_J}\ \text{(Helper-Ind)}$$

Rule J:AM-Spec (Non-Final-Version) allows the execution to speculatively jump to any location and we would need to create a speculative instance for **each** of these locations. Here, Rule Helper-Ind and Rule Helper-Base are helpers creating the speculative transactions and we use $x \subset e$ to denote the subexpressions of e, thereby ensuring that only indirect jumps are used for speculation.

However, creating a speculative instance for each location in the program makes the analysis of the program infeasible because of the amount of states that need to be explored. Furthermore, against such a model, almost any "interesting" program would be considered insecure since speculation is, in practice, unrestricted.

One of the techniques employed by modern CPUs to restrict the scope of indirect-jump speculation is (hardware) control-flow-integrity (CFI) [1]. CFI leverages tags to mark valid jump targets in the code and ensures that the code can only jump to these tagged targets. There are different ways tagging and enforcement mechanisms can be implemented (see Burow et al. [17] for a comprehensive survey). We focus on the hardware implementations of CFI of Intel (Intel-CET [68]) and ARM (branch target identification [9]) because they are available on new hardware and constrain *speculative* control flow: when enabled, a predicted indirect-branch target that does not begin with an **endbr** instruction limits or halts transient execution [68].

Both implementations add a new instruction **endbr** (in the case of ARM this instruction is called **bti**) that marks the legal targets of indirect jumps in a program and enables coarse-grained forward edge control flow integrity. Enforcement is done by a state machine in hardware that ensures only valid jumps are allowed. Furthermore, all available standard

compilers (i.e. Gcc, Clang) already emit these **endbr** instructions and these instructions are handled as no-ops if the current hardware does not support CFI, making it backwards-compatible. Thus, we similarly add a **endbr** instruction to μAsm, which marks legal targets for indirect jumps in the program and otherwise behaves like a **skip** instruction (Rule Endbr). This restricts the scope of indirect-jump speculation and allows for a tractable analysis.

$$\text{(Instructions) } i ::= \cdots \mid \textbf{endbr} \qquad \frac{p(a(\textbf{pc})) = \textbf{endbr}}{\langle p, \langle m, a\rangle\rangle \rightarrow \langle p, \langle m, a[\textbf{pc} \mapsto a(\textbf{pc}) + 1]\rangle\rangle}\ \text{(Endbr)}$$

First, we keep track of all possible allowed jump targets in the program. We collect the instruction number for each **endbr** instruction into a labelset L defined as follows: $L_p = \{i \mid \forall i \in \mathbb{N}.\, p(i) = \textbf{endbr}\}$. We will omit p in L_p if it is clear from context. One possible way to modify the program in Listing 9 with **endbr** instruction is the program in Listing 10. Then, the labelset is defined as: $L = \{Line\ 10, Line\ 14\}$. Next, we modify the rule for non-speculative jumps by requiring that the jump target is part of the allowed jump targets L (Rule Jmp-Mod):

$$\frac{\begin{array}{cc} p(a(\textbf{pc})) = \textbf{jmp}\ e & \ell = [\![e]\!](a) \\ \exists x \subset e.\, x \in Regs & l \in L \end{array}}{\langle p, \langle m, a\rangle\rangle \xrightarrow{\text{pc } \ell} \langle p, \langle m, a[\textbf{pc} \mapsto \ell]\rangle\rangle}\ \text{(Jmp-Mod)}$$

Note that for all other semantics, we can define the set of possible jump targets to be all instruction labels in the program to recover the original rule for non-speculative jumps.

Next, we define the new speculative semantics:

$$\boxed{\Phi_{\jmath} \overset{\tau}{\dashrightarrow}_{\jmath} \overline{\Phi}'_{\jmath}}$$

$$\frac{\begin{array}{ccc} p(\sigma(\textbf{pc})) = \textbf{jmp}\ x & \sigma \xrightarrow{\tau} \sigma' & j = min(\omega, n) \\ \exists x \subset e.\, x \in Regs & \tau' = \tau \cdot \texttt{pc}\ \sigma'(\textbf{pc}) \cdot \texttt{start}_{\jmath}\ ctr & \\ \multicolumn{3}{c}{\langle p, ctr, \sigma, j\rangle \vdash^{L \setminus \{\sigma'.\textbf{pc}\}} \overline{\Phi}_{\jmath}} \end{array}}{\langle p, ctr, \sigma, n+1\rangle \overset{\tau'}{\dashrightarrow}_{\jmath} \langle p, ctr, \sigma', n\rangle \cdot \overline{\Phi}_{\jmath}}\ \text{(J:AM-Spec)}$$

We replace the unmitigated Rule J:AM-Spec (Non-Final-Version) with the constrained Rule J:AM-Spec. The main difference lies in the definition of the allowed jump target set. The set S in the unmitigated rule allows jumps to arbitrary addresses in the program. In the new rule this is restricted to L, allowing jumps only to targets designated by **endbr** instructions. When an indirect branch is encountered, Rule J:AM-Spec creates a speculative instance for each target in L, excluding the correct non-speculative target. Thus, the rule creates exactly $|L| - 1$ speculative instances. Consider the example in Listing 10 where $L = \{Line\ 10, Line\ 14\}$. Because Rule J:AM-Spec can only speculatively jump to targets in L, execution cannot speculatively reach Line 17, successfully preventing the leakage of the private variable p. Our $\dashrightarrow_{\jmath}$ semantics models speculation over indirect jumps; we do not model BTB-driven speculation triggered by **call** instructions.

Crucially, because this rule creates a set of instances rather than a single misprediction (diverging from the standard single-target template in Section 5.2), it introduces an asymmetry between start and rollback observations: A single $\texttt{start}_{\jmath}\ ctr$ observation is now associated with multiple rollback observations (exactly $|L| - 1$ many).[5] However, because the speculative instances are pushed onto the execution stack sequentially, their execution is strictly nested. The resulting trace exhibits the following structure, where inner transactions are rolled back before the outer transaction

[5] We could have decided to add matching $\texttt{start}_{\jmath}\ ctr$ observations to Rule J:AM-Spec to balance the $\texttt{start}_{\jmath}\ i$ and the $\texttt{rlb}_{\jmath}\ i$.

concludes:

$$\texttt{start}_{\mathcal{J}}\ ctr \cdots \texttt{rlb}_{\mathcal{J}}\ ctr + |L| - 2 \cdots \texttt{rlb}_{\mathcal{J}}\ ctr$$

This nesting guarantees the correctness of the non-speculative projection function $\restriction_{ns}$. Since $\restriction_{ns}$ erases the trace segment between a start event and its matching rollback (here, the final $\texttt{rlb}_{\mathcal{J}}\ ctr$), all intermediate speculative events—including the nested rollbacks—are correctly removed from the trace.

6.5.2 Oracle Semantics. Instead of predicting all of the possible indirect **jmp** targets, the oracle chooses one of the possible jump targets in L and continues only with this one choice. As before, the behaviour $Beh^{O}_{\mathcal{J}}(p)$ of a program p is the set of all traces starting from an initial state until termination using the reflexive-transitive closure of the oracle semantics.

6.6 Safety of Semantics

We conclude this section by proving the core properties satisfied by all semantics. Theorem 1 characterizes that all the speculative semantics presented in the paper are well-formed speculative semantics, i.e., (1) their always-mispredict version over-approximates the corresponding oracle version, (2) they are consistent under non-speculative trace projection, and (3) their symbolic version is consistent with the corresponding non-symbolic version.

Theorem 1 (Well-Formed Speculative Semantics). *The following statements hold:*

- $(\text{🔔}_{\text{B}}$ *is WFSS)* $\vdash \text{🔔}_{\text{B}}$ *WFSS*
- $(\text{🔔}_{\text{S}}$ *is WFSS)* $\vdash \text{🔔}_{\text{S}}$ *WFSS*
- $(\text{🔔}_{\text{R}}$ *is WFSS)* $\vdash \text{🔔}_{\text{R}}$ *WFSS*
- $(\text{🔔}_{\text{SLS}}$ *is WFSS)* $\vdash \text{🔔}_{\text{SLS}}$ *WFSS*
- $(\text{🔔}_{\mathcal{J}}$ *is WFSS)* $\vdash \text{🔔}_{\mathcal{J}}$ *WFSS*

7 A FRAMEWORK FOR COMPOSING SPECULATIVE SEMANTICS

Although the instances of speculative semantics presented in Section 6 each capture one specific aspect of speculation (e.g., 🔔_{B} captures branch speculation) , they do not capture the vulnerability in Listing 3 (restated here for clarity) as the traces of Example 11 show.

Example 11 (SNI for Listing 11)**.** The traces generated are:

$$\overline{\tau}^1_{\text{B}} = \overline{\tau}^2_{\text{B}} := \texttt{store}\ p \cdot \texttt{store}\ p \cdot \texttt{start}_{\text{B}}\ 0 \cdot \texttt{load}\ p \cdot \texttt{load}\ A + public \cdot \texttt{rlb}_{\text{B}}\ 0 \cdot \texttt{pc}\ 9$$

$$\overline{\tau}^1_{\text{S}} = \overline{\tau}^2_{\text{S}} := \ldots \cdot \texttt{store}\ p \cdot \texttt{start}_{\text{S}}\ 1 \cdot \texttt{bypass}\ 1 \cdot \texttt{pc}\ \bot \cdot \texttt{rlb}_{\text{S}}\ 1 \cdot \texttt{pc}\ \bot$$

The program in Listing 11 seems secure since there is no secret value leaked in the speculative transaction; thus the program satisfies SNI for 🔔_{B} and 🔔_{S} in isolation. However, this program speculatively leaks when considering speculation over **beqz** and **store** instructions, but we would need a combined semantics to detect this vulnerability.

The vulnerability only appears when the branch predictor (Section 6.1) and the memory disambiguator (Section 6.2) are used *together*. Intuitively, we know that CPUs use all the speculation mechanisms described here (and many others as well) at the same time. Thus, we should not only focus on these individual speculation mechanisms in *isolation* but we need to look at their *combinations* as well. That is, we need a way to compose the different semantics into new semantics that can reason about these "combined" leaks.

Listing 11. $\mathcal{A}_{\mathsf{B+S}}$ example.

```
x = 0;
p = &secret;
p = &public;
if (x != 0)
    temp &= A[*p];
```

This section presents a novel, general framework for composing two speculative semantics x and y, each one capturing the effects of a single speculation mechanism, to allow for speculation from both mechanisms x and y. The semantics x and y are also called the *source* semantics of the composition. Next, we first introduce the new composed semantics, which consists of an always-mispredict semantics, an oracle semantics, and a symbolic semantics (Section 7.1). Then, we present the notion of *well-formed* composition which we use to study the properties of composed semantics (Section 7.2).

New Notation. The states Σ_{xy}, instances Φ_{xy}, and the trace model Obs_{xy} are defined as the union of the source parts. Furthermore, we define a projection function $\restriction_{xy}$ and two projections $\restriction^x_{xy}$ and $\restriction^y_{xy}$ that return the first and second projection of the pair from $\restriction_{xy}$. These functions are lifted to states by applying them pointwise:

$$Obs_{xy} \coloneqq Obs_x \cup Obs_y \qquad \Phi_{xy} \coloneqq \Phi_x \cup \Phi_y \qquad \Sigma_{xy} \coloneqq \Sigma_x \cup \Sigma_y$$
$$\restriction_{xy} \colon \Phi_{xy} \mapsto (\Phi_x, \Phi_y) \qquad \restriction^x_{xy} \colon \Phi_{xy} \mapsto \Phi_x \qquad \restriction^y_{xy} \colon \Phi_{xy} \mapsto \Phi_y$$

For example, the $\Phi_{\mathsf{S+R}}$ state resulting from the union of Φ_{S} and Φ_{R} states (from Section 6.2.1 and Section 6.3.1 respectively) is $\langle p, ctr, \sigma, \mathbb{R}, n\rangle$, as it contains all common elements (the program p, the counter ctr, the state σ, and the speculation count n) plus the return stack buffer $\mathbb{R}$ from Φ_{R} only. Taking the $\cdot\restriction^{\mathsf{S}}_{\mathsf{S+R}}$ of a $\Phi_{\mathsf{S+R}}$ state returns the Φ_{S} subpart, i.e., all but the return stack buffer.

We overload $\restriction^x_{xy}$ and $\restriction^y_{xy}$ to also work on traces $\overline{\tau}$. The projection $\tau\restriction^x_{xy}$ deletes all speculative transactions (marked by $\mathtt{start}_y\ id$ and $\mathtt{rlb}_y\ id$) not generated by the source semantics x. The definition of $\restriction^y_{xy}$ is similar by replacing x with y:

$$\varepsilon\restriction^x_{xy} = \varepsilon \qquad (\tau \cdot \overline{\tau})\restriction^x_{xy} = \tau \cdot (\overline{\tau})\restriction^x_{xy}$$
$$(\mathtt{start}_y\ id \cdot \cdots \mathtt{rlb}_y\ id \cdot \overline{\tau})\restriction^x_{xy} = \overline{\tau}\restriction^x_{xy}$$

We indicate source semantics for x and y as $\mathcal{A}_x$ and $\mathcal{A}_y$ respectively and use $\mathcal{A}_{xy}$ to indicate the composed semantics.

7.1 Combined Speculative Semantics

The combined semantics delegates back to the source semantics of x and y to model the effects of both speculation mechanisms (modeled by x and y). This is captured in the two core rules below:

$$\text{(AM-x-Step)}\ \frac{\Phi_{xy}\restriction^x_{xy} \overset{\tau}{\dashrightarrow}_{\mathcal{A}_x}^{Z_{xy}\restriction^x_{xy}} \overline{\Phi}'_{xy}\restriction^x_{xy}}{\Phi_{xy} \overset{\tau}{\dashrightarrow}_{\mathcal{A}_{xy}}^{Z_{xy}} \overline{\Phi}'_{xy}} \qquad \text{(AM-y-Step)}\ \frac{\Phi_{xy}\restriction^y_{xy} \overset{\tau}{\dashrightarrow}_{\mathcal{A}_y}^{Z_{xy}\restriction^y_{xy}} \overline{\Phi}'_{xy}\restriction^y_{xy}}{\Phi_{xy} \overset{\tau}{\dashrightarrow}_{\mathcal{A}_{xy}}^{Z_{xy}} \overline{\Phi}'_{xy}}$$

The combined semantics does a step by either delegating back to the x source semantics (Rule AM-x-Step) or to the y one (Rule AM-y-Step).[6] The rules rely on metaparameter Z_{xy}, which is a pair of two metaparameters $Z_{xy} \coloneqq (Z_x, Z_y)$ — one for x and one for y. We overload the projections $\restriction^x_{xy}$ and $\restriction^y_{xy}$ to extract the corresponding metaparameter from Z_{xy}, e.g., $Z_{xy}\restriction^x_{xy} = Z_x$.

[6] To simplify notation, we omit that the $\Phi_x \setminus \Phi_y$ parts of state Φ_{xy} in x-step (similar $\Phi_y \setminus \Phi_x$ in y-step) do not change between Φ_{xy} and $\overline{\Phi}'_{xy}$.

The role of Z is central to making the composed semantics work as expected. It restricts how the combined semantics delegates execution to the components to ensure that the correct rule is applied.

With $Z = (\emptyset, \emptyset)$, consider the execution of the **beqz** instruction in Line 4 in Listing 11. The combined semantics $\mathcal{A}_{B+S}$ can use Rule AM-x-Step to delegate back to $\mathcal{A}_B$ for **beqz** instructions, creating a new speculative transaction (Rule B:AM-Spec). However, $\mathcal{A}_{B+S}$ can also use Rule AM-y-Step, because **beqz** instructions are also handled by $\mathcal{A}_S$. Unfortunately, this does not start speculation, which happens only on **store** instructions (Rule S:AM-Spec).

Intuitively, $\mathcal{A}_{B+S}$ should delegate back to $\mathcal{A}_B$, so Rule AM-y-Step should not be applicable. This can be obtained by instantiating $Z_{B+S} = (\textbf{store}, \textbf{beqz})$, so that its projections are $Z_B = \textbf{store}$ and $Z_S = \textbf{beqz}$. Now, $\mathcal{A}_{B+S}$ can only apply Rule AM-x-Step on the **beqz** of Line 4, because Z_S ensures that $\mathcal{A}_S$ cannot execute **beqz** instructions, as depicted in the full rule for $\mathcal{A}_S^{\textbf{beqz}}$ below (where we indicate the instructions derived from $Z_S = \textbf{beqz}$ in blue):

$$\frac{p(\sigma(\mathbf{pc})) \notin \textbf{store} \cup \textbf{beqz} \qquad \langle p, \sigma\rangle \xrightarrow{\tau} \langle p, \sigma'\rangle}{\langle p, ctr, \sigma, n+1\rangle \xrightarrow{\tau} \mathcal{A}_S^{\textbf{beqz}} \langle p, ctr, \sigma', n\rangle} \text{(S:AM-NoSpec)}$$

Having clarified the intuition behind the semantics, we can define the behaviour $Beh^{\omega}_{xy}()$ as the set of all traces generated from initial states until termination using $\xrightarrow{\tau} \mathcal{A}_{xy}$.

7.1.1 Oracle Combination. Instead of using one oracle, the combination uses a pair of oracles, one from each source semantics. When delegating back to either source, the correct oracle of the source is handed over to the source semantics.

7.1.2 Symbolic Combination. Instead of using the AM semantics for delegation, the combined symbolic semantics $\mathcal{A}^{S}_{xy}$ uses the symbolic source semantics for delegation. Furthermore, the new notation (union, projections) is lifted to the symbolic combination to create the symbolic states Σ^{S}_{xy}. The behaviour $Beh^{S}_{xy}(p)$ of program p is the set of all traces generated using the symbolic semantics.

7.2 Properties of Composition

We now illustrate the benefits of our composition framework. For this, we first introduce a notion of well-formed composition (Section 7.2.1), which intuitively tells when a combined semantics "makes sense". Then, we show that for well-formed compositions, if the source semantics are *WFSS*, so is the combined semantics (Section 7.2.2). Since we proved this property for *any* well-formed composition in our framework, all (well-formed) compositions we present in Section 8 are *WFSS for free*. This proof reuse and extensibility is our framework's key advantage over having ad-hoc semantics combining multiple speculation mechanisms, which requires one to manually prove the *WFSS* results we instead obtain for free.

7.2.1 Well-formed Compositions. The well-formedness conditions for the composition in Definition 4 ensures that the delegation between the source semantics is done properly. Note that these conditions are the *minimal* set of assumptions that let us derive *WFSS* of the combined semantics for free:

Definition 4 (Well-Formed Composition)**.** *A composition $\mathcal{A}_{xy}$ of two speculative semantics $\mathcal{A}_x$ and $\mathcal{A}_y$ is* well-formed*, written $\vdash \mathcal{A}_{xy} : WFC$, if:*

(1) (Confluence) *Whenever $\Sigma_{xy} \xrightarrow{\tau} \mathcal{A}_{xy}\ \Sigma'_{xy}$ and $\Sigma_{xy} \xrightarrow{\tau} \mathcal{A}_{xy}\ \Sigma''_{xy}$, then $\Sigma'_{xy} = \Sigma''_{xy}$.*

(2) (Projection preservation) *For all p, $Beh^{\omega}_{x}(p) = Beh^{\omega}_{xy}((p))\restriction^{x}_{xy}$ and $Beh^{\omega}_{y}(p) = Beh^{\omega}_{xy}((p))\restriction^{y}_{xy}$.*

(3) (Relation preservation) *If $\Sigma_{xy} \approx_{xy} X_{xy}$ and $\Sigma_{xy} \xrightarrow{\overline{\tau}} \mathcal{A}^{*}_{xy}\ \Sigma'_{xy}$ then $X_{xy} \overset{\overline{\tau}'}{\rightsquigarrow}{}^{O_{xy}}_{xy*}\ X'_{xy}$ and $\Sigma'_{xy} \approx_{xy} X'_{xy}$.*

(4) (Symbolic preservation) *If* $\Sigma^{\mathcal{S}}_{xy} \xrightarrow{\tau_S} \mathcal{S}_{xy} \Sigma^{\mathcal{S}'}_{xy}$ *and* $\mu\,\Sigma^{\mathcal{S}}_{xy} = \Sigma_{xy}$*, then there is* Σ'_{xy} *s.t.* $\Sigma_{xy} \xrightarrow{\mu\,\tau_S}_{xy} \Sigma'_{xy}$ *and* $\mu\,\Sigma^{\mathcal{S}'}_{xy} = \Sigma'_{xy}$.

Next, we explain the well-formedness conditions:

- Confluence (point 1) ensures that the non-determinism of the combined semantics (that non-deterministically delegates back to its sources) is not harmful. Consider the assignment in Line 1 in Listing 3. $\mathcal{S}_{\mathsf{B+S}}$ can delegate to either $\mathcal{S}_{\mathsf{B}}$ or $\mathcal{S}_{\mathsf{S}}$ to reduce the assignment. If the combined semantics is *confluent*, then it does not matter which source rule executes the assignment in Line 1 in Listing 3, the semantics reaches the same state either way.
- Projection preservation (point 2) ensures that the combined semantics is not hiding or forgetting traces of its sources. Any observable emitted by a source semantics must be propagated to the combined one, this is also the reason why Obs_{xy} is defined as the union of the source *Obs*.
- To explain relation preservation (point 3), we need to mention a technical detail: the state relation (denoted $\approx_{xy}$ and defined in our technical report) between the AM states (Σ_{xy}) and the Oracle ones (X_{xy}). Intuitively, two states are related if they are the same or if one is waiting on a speculation of the other to end. Then, point (3) ensures that whenever we start from related states ($\Sigma_{xy} \approx_{xy} X_{xy}$) and we do one or more steps of the AM composed semantics ($\Sigma_{xy} \xrightarrow{\overline{\tau}}{}^{*}_{xy} \Sigma'_{xy}$), then we can *always* find a related state ($\Sigma'_{xy} \approx_{xy} X'_{xy}$) that is reachable by performing one or more steps of the composed oracle semantics ($X_{xy} \xrightarrow{\overline{\tau}'}{}^{O_{xy}*}_{xy} X'_{xy}$). This fact is used when proving that SNI of a program under the composed AM semantics implies SNI under the composed oracle semantics (point 1 of Definition 3). Thus, it is not important for the AM and the Oracle semantics to produce the same traces, just that the two AM traces and the two Oracle traces are pairwise equivalent – which follows from the state relation.
- Finally, symbolic preservation (point 4) ensures that any step of the always-mispredict composed semantics corresponds to the concretization of a step of the symbolic composed semantics (and vice versa[7]). Note that proving symbolic preservation is almost trivial whenever both source semantics enjoy the same property (like our semantics $\mathcal{S}_{\mathsf{B}}$, $\mathcal{S}_{\mathsf{J}}$, $\mathcal{S}_{\mathsf{S}}$, $\mathcal{S}_{\mathsf{R}}$ and $\mathcal{S}_{\mathsf{SLS}}$).

7.2.2 WFSS Preservation. The key result of this section is that well-formed compositions whose sources are well-formed speculative semantics (*WFSS*) are also *WFSS* (Theorem 2). Note that our proof of Theorem 2, available in the companion technical report [30], holds for *any* well-formed composition in our framework and, therefore, it applies *for free* to all the compositions in Section 8.

Theorem 2 ($\mathcal{S}_{xy}$ is *WFSS*). *If* $\vdash \mathcal{S}_x$ *WFSS and* $\vdash \mathcal{S}_y$ *WFSS and* $\vdash \mathcal{S}_{xy}$: *WFC, then* $\vdash \mathcal{S}_{xy}$ *WFSS.*

As a corollary of Theorem 2, we obtain that the security of well-formed compositions is related to the security of their components (Theorem 3). In particular, whenever a program is insecure w.r.t. one of the components, then it is insecure w.r.t. the composed semantics. Dually, if a program is secure w.r.t. the composed semantics, then it is secure w.r.t. the single components. Note, however, that there are programs that are secure for the single components but insecure w.r.t. the composed semantics like Listing 3.

Theorem 3 (Combined SNI Preservation). *Whenever* $\vdash \mathcal{S}_{xy}$: *WFC holds:*

- *If* $p \nvdash_x$ *SNI or* $p \nvdash_y$ *SNI, then* $p \nvdash_{xy}$ *SNI.*
- *If* $p \vdash_{xy}$ *SNI, then* $p \vdash_x$ *SNI and* $p \vdash_y$ *SNI.*

[7] For space reasons, Definition 4 only reports one direction (with a simplified notation).

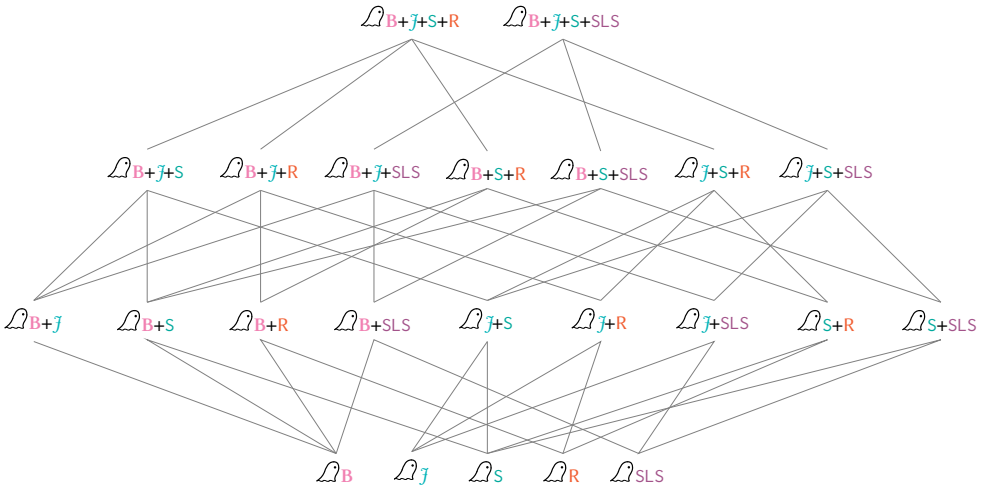

Fig. 9. Partial order of the different combinations that we instantiate using our framework

These results have an immediate practical impact on Spectector: (1) the analysis of Spectector relies on the (symbolic) speculative semantics, (2) the source semantics $\mathcal{A}_{\mathsf{B}}$, $\mathcal{A}_{\mathsf{J}}$, $\mathcal{A}_{\mathsf{S}}$, $\mathcal{A}_{\mathsf{R}}$ and $\mathcal{A}_{\mathsf{SLS}}$ are *WFSS*, (3) well-formed compositions are also *WFSS*, and (4) the composition of $\mathcal{A}_{\mathsf{B}}$, $\mathcal{A}_{\mathsf{J}}$, $\mathcal{A}_{\mathsf{S}}$, $\mathcal{A}_{\mathsf{R}}$ and $\mathcal{A}_{\mathsf{SLS}}$ are well-formed. So, Spectector equipped with any combination of the $\mathcal{A}_{\mathsf{B}}$, $\mathcal{A}_{\mathsf{J}}$, $\mathcal{A}_{\mathsf{S}}$, $\mathcal{A}_{\mathsf{R}}$ and $\mathcal{A}_{\mathsf{SLS}}$ produces sound results, i.e., whenever the tool proves that a program is leak-free then the program satisfies SNI. In the next section, we describe all the compositions and prove they are well-formed (this implies that they are *WFSS* thanks to Theorem 2).

8 INSTANTIATING OUR FRAMEWORK

We instantiate our combination framework with all possible combinations of $\mathcal{A}_{\mathsf{B}}$, $\mathcal{A}_{\mathsf{J}}$, $\mathcal{A}_{\mathsf{S}}$, $\mathcal{A}_{\mathsf{R}}$ and $\mathcal{A}_{\mathsf{SLS}}$, yielding 18 combinations depicted in Figure 9. Combinations of $\mathcal{A}_{\mathsf{SLS}}$ and $\mathcal{A}_{\mathsf{R}}$ are impossible because they speculate on the same instructions (**ret**). Thus, we cannot set the metaparameter Z in a sensible way (more details in Section 11). Since we cannot describe all of the combinations in detail, we focus on two representative combinations: $\mathcal{A}_{\mathsf{S+R}}$ (Section 8.1) and $\mathcal{A}_{\mathsf{B+J+S+R}}$ (Section 8.2); the other combinations can be instantiated in a similar way.

For each of these, we overview the combined AM semantics using examples and we prove that the combined semantics is well-formed, i.e., it satisfies Definition 4.

Full details and well-formedness proofs of all other combinations are available in the companion technical report [30].

8.1 $\mathcal{A}_{\mathsf{S+R}}$ Composition

To combine semantics using our framework, we need to define the states, observations, and metaparameter $Z_{\mathsf{S+R}}$ for the composed semantics $\mathcal{A}_{\mathsf{S+R}}$. The combined state $\Sigma_{\mathsf{S+R}}$ is the union of the states Σ_{S} and Σ_{R}; thus it contains the RSB $\mathbb{R}$ as well.

$$\textit{Spec. States } \Sigma_{\mathsf{S+R}} ::= \overline{\Phi}_{\mathsf{S+R}} \qquad \textit{Spec. Instance } \Phi_{\mathsf{S+R}} ::= \langle p, ctr, \sigma, \mathbb{R}, n \rangle$$

Listing 12. $\lightning_{S+R}$ example.

```
1  Manip_Stack:
2      sp ← sp + 8
3      ret
4  Speculate:
5      call Manip_Stack
6      store secret, p
7      store pub, p
8      load eax, p
9      load edi, eax
10     ret
11 Main:
12     call Speculate
13     skip
```

The union Obs_{S+R} of the trace models Obs_S and Obs_R is defined as:

$$Obs_{S+R} ::= \texttt{start}_S\ n \mid \texttt{start}_R\ n \mid \texttt{rlb}_S\ n \mid \texttt{rlb}_R\ n \mid \texttt{bypass}\ n \mid \ldots$$

To define the metaparameter Z_{S+R}, we need to identify the instructions that are related with speculative execution for each component semantics. For $\lightning_S$, the only instruction associated with speculative execution is **store**, since the semantics can only speculatively bypass stores. For $\lightning_R$, even though the semantics speculates only over **ret** instructions, **call** instructions also affect speculative execution since $\lightning_R$ pushes return addresses onto the RSB $\mathbb{R}$ when executing **call**s. Therefore, we set the metaparameter Z_{S+R} to (**call** $\cup$ **ret**, **store**). This ensures that in $\lightning_{S+R}$, **store** instructions are only executed by delegating back to $\lightning_S^{\textbf{call}\cup\textbf{ret}}$ whereas **call** and **ret** instructions are only executed by delegating back to $\lightning_R^{\textbf{store}}$.

Theorem 4 states the combination of $\lightning_S$ and $\lightning_R$ described above is well-formed. Given that $\lightning_S$ and $\lightning_R$ are *WFSS* (Theorem 1), we can derive "for free" that $\lightning_{S+R}$ is *WFSS* (Theorem 2).

Theorem 4 ($\lightning_{S+R}$ is well-formed). $\vdash \lightning_{S+R} : \mathit{WFC}$

Listing 12 presents a program that contains a leak that can be detected only by $\lightning_{S+R}$ but not by its components $\lightning_S$ and $\lightning_R$. Execution starts on Line 12 by calling the function *Speculate* and it continues at Line 5. Next, the function *Manip_Stack* is called and the stack pointer **sp** is incremented (Line 2). This modifies the return address of the function *Manip_Stack* to now point to Line 13 (the return address of the **call** to *Speculate*). Under $\lightning_R$, mispredicting the return address of *Manip_Stack* using the RSB leads to continuing the execution at Line 6. However, the **store** instructions in Line 7 overwrites the secret value stored in Line 6. Then, the **load** instructions in Line 8 and Line 9 emit only public values. As a result, no secret is leaked and speculation ends. Similarly, under $\lightning_S$, speculation over store bypasses has no effect in Listing 12 because the **store** instruction in Line 6 is never reached and function *Manip_Stack* returns to Line 13. Therefore, the leak is missed under $\lightning_S$ and $\lightning_R$, i.e., Listing 12 $\vdash_S$ SNI and Listing 12 $\vdash_R$ SNI.

However, under the combined semantics $\lightning_{S+R}$, the **store** instruction on Line 7 is now speculatively bypassed and when returning from function *Manip_Stack* the execution speculatively continues from Line 8. Now, the **load** instructions are executed and the secret is leaked, as shown in the traces below. Since *secret* is a high value, there are low-equivalent configurations σ^1, σ^2 that differ in the value of *secret*. Thus, there are two traces that differs in the observation $\texttt{load}\ \mathit{secret}$ (highlighted in gray). Hence, the program is not secure under the combined semantics, i.e.,

Listing 12 $\not\sim_{S+R}$ SNI.

$$\tau^2_{S+R} \neq \tau^1_{S+R} \stackrel{\text{def}}{=} \texttt{call}\ \mathit{Speculate} \cdots \texttt{start}_R\ 0 \cdots \texttt{start}_S\ 1 \cdots \texttt{rlb}_S\ 1 \cdots \texttt{start}_S\ 2 \cdot \texttt{bypass}\ 7 \cdot \texttt{load}\ p \cdot \texttt{load}\ \mathit{secret} \cdots$$

The relation between the source semantics and their composition is visualised in Figure 10, which shows the insecure programs (with respect to SNI) detected under the different semantics. The combined semantics encompasses all vulnerable programs of $\mathcal{A}_S$ and $\mathcal{A}_R$ *and* additional programs like Listing 12. These additional programs are the reason why the semantics $\mathcal{A}_{S+R}$ is "stronger than the sum of its parts" $\mathcal{A}_S$ and $\mathcal{A}_R$.

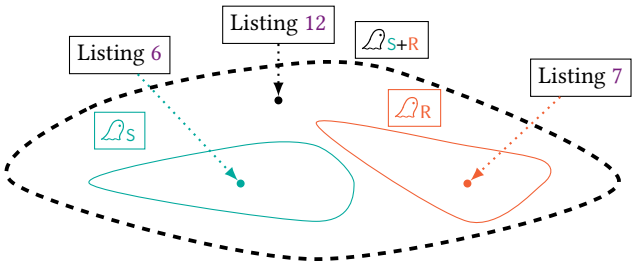

Fig. 10. Relating $\mathcal{A}_S$, $\mathcal{A}_R$ and $\mathcal{A}_{S+R}$ w.r.t. SNI.

8.2 $\mathcal{A}_{B+J+S+R}$ Composition

We conclude this section by combining four semantics $\mathcal{A}_B$, $\mathcal{A}_J$, $\mathcal{A}_S$ and $\mathcal{A}_R$ (as mentioned, a combination with five semantics is not possible because $\mathcal{A}_{SLS}$ and $\mathcal{A}_R$ speculate on the same instruction). Our framework (Section 7) allows us to only combine a pair of source semantics into a combined one. For simplicity, we present $\mathcal{A}_{B+J+S+R}$ as a direct combination of the four source semantics (technically, we obtain $\mathcal{A}_{B+J+S+R}$ by combining $\mathcal{A}_{B+J}$ with $\mathcal{A}_{S+R}$).

The metaparameter $Z_{B+J+S+R}$ (which we represent as a quadruple of values) is :

$$Z_B \stackrel{\text{def}}{=} (\textbf{call} \cup \textbf{ret} \cup \textbf{store} \cup \textbf{jmp})$$

$$Z_J \stackrel{\text{def}}{=} (\textbf{call} \cup \textbf{ret} \cup \textbf{store} \cup \textbf{beqz})$$

$$Z_S \stackrel{\text{def}}{=} (\textbf{call} \cup \textbf{ret} \cup \textbf{beqz} \cup \textbf{jmp})$$

$$Z_R \stackrel{\text{def}}{=} (\textbf{beqz} \cup \textbf{store} \cup \textbf{jmp})$$

$$Z_{B+J+S+R} \stackrel{\text{def}}{=} (Z_B, Z_J, Z_S, Z_R)$$

As a result, the combined semantics $\mathcal{A}_{B+J+S+R}$ can only delegate to the corresponding speculative semantics for the appropriate speculation sources.

As stated in Theorem 5, $\mathcal{A}_{B+J+S+R}$ is well-formed and we derive that $\mathcal{A}_{B+J+S+R}$ is *WFSS* through Theorem 2.

Theorem 5 ($\mathcal{A}_{B+J+S+R}$ is well-formed). $\vdash \mathcal{A}_{B+J+S+R} : \mathit{WFC}$

Figure 11 depicts a leaky program that can be detected only under $\mathcal{A}_{B+J+S+R}$, since the program satisfies SNI under $\mathcal{A}_B$, $\mathcal{A}_J$, $\mathcal{A}_S$ and $\mathcal{A}_R$. Under $\mathcal{A}_{B+J+S+R}$, the **store** instruction in Line 18 is bypassed. Therefore, when returning from the *Manip_Stack* function, the program mispredicts the return address and speculatively returns to Line 6. Here, the indirect jump in Line 7 mispredicts and execution continues speculatively in Line 9. Finally, the **beqz** instruction in Line 11 is mispredicted and the **load** instructions are executed, which leaks the secret value.

```
1  Manip_Stack:
2      sp <- sp + 8
3      ret
4  Speculate:
5      call Manip_Stack
6      r1 <- L2
7      jmp r1
8  L1:
9      endbr
10     x <- 0
11     beqz x, L2
12     load eax, p
13     load edi, eax
14 L2:
15     ret
16 Main:
17     store secret, p
18     store pub, p
19     call Speculate
```

Fig. 11. $\not\Delta_{B+J+S+R}$ example.

The resulting traces, differing in the value of *secret* exposed by the load *secret* observation (highlighted in gray), are given below:

$$\tau^2_{B+J+S+R} \neq \tau^1_{B+J+S+R} \stackrel{\text{def}}{=} \cdots \texttt{start}_S\ 1 \cdot \texttt{bypass}\ 18 \cdot \texttt{call}\ \mathit{Speculate} \cdot \texttt{call}\ \mathit{Manip\_Stack} \cdot \texttt{start}_R\ 2 \cdot \texttt{ret}\ 6 \cdot \texttt{start}_J\ 3 \cdot \texttt{pc}\ 9 \cdot \texttt{start}_B\ 4 \cdot \texttt{pc}\ 12 \cdot \texttt{load}\ p \cdot \boxed{\texttt{load}\ \mathit{secret}} \cdot \texttt{rlb}_B\ 4 \cdot \texttt{rlb}_J\ 3 \cdot \texttt{rlb}_R\ 2 \cdot \texttt{rlb}_S\ 1 \cdots$$

Thus, the program is not secure, i.e., Figure 11 $\not\vdash_{B+J+S+R}$ SNI.

9 DETECTING SPECULATIVE INFORMATION FLOWS : THE SPECTECTOR ALGORITHM

This section presents Spectector, a program analysis for detecting speculative leaks or proving their absence. The core idea behind Spectector is to formally compare a program's behavior under speculative execution against its standard, non-speculative execution. A leak is identified if the speculative execution reveals information that is not revealed by the non-speculative execution. To make this comparison comprehensive, analyzing every possible concrete execution is infeasible. Therefore, Spectector leverages symbolic execution to represent all possible program behaviors concisely as a set of symbolic traces. Our approach requires two key formalisms:

(1) A symbolic non-speculative semantics to establish the program's intended, baseline behavior (Section 9.1).
(2) A symbolic architectural model (AM) semantics to model the processor's speculative behavior (Section 9.2).

By analyzing the symbolic traces generated from both semantics with an SMT solver, Spectector can identify discrepancies that correspond to memory or control-flow leaks. If no such discrepancies are found, the program is proven secure against the modeled speculative behaviors (Section 9.3). Finally, we discuss the implementation of Spectector within the Ciao logic programming system [41] (Section 9.4).

9.1 Symbolic Non-Speculative Semantics for μAsm

To establish a ground truth for our analysis, we first define the program's intended behavior using a symbolic non-speculative semantics, denoted by the relation $\langle p, \sigma^{\mathcal{S}} \rangle \xrightarrow{\tau_S}_{\mathcal{S}} \langle p, \sigma^{\mathcal{S}'} \rangle$.

This semantics lifts the concrete non-speculative semantics $\langle p, \sigma \rangle \xrightarrow{\tau} \langle p, \sigma' \rangle$ to operate over symbolic values instead of concrete ones. We now detail the key extensions.

Symbolic program states consist of a program p and a symbolic configuration $\sigma^{\mathcal{S}} ::= \langle sm, sa \rangle$.

Concrete memories m are replaced with symbolic memories sm, modeled as symbolic memories using the standard theory of arrays [16]. We model memory updates as triples of the form $\mathbf{write}(sm, se, se')$, which update the symbolic memory sm by assigning the symbolic value se' to the symbolic location se. Memory reads $\mathbf{read}(sm, se)$ retrieve the value at symbolic memory location se in sm. $\mathbf{read}(sm, se)$ and $\mathbf{write}(sm, se, se')$ are used in Rule Load-Symb and Rule Store-Symb respectively, which is the only difference to the concrete non-speculative semantics. Concrete register assignments a are replaced with symbolic register assignments sa, which are functions mapping registers to symbolic expressions.

Symbolic expressions represent computations over symbolic values. A *symbolic expression* se is a concrete value $n \in Vals$, a symbolic value $s \in SymbVals$, an if-then-else expression $\mathbf{ite}(se, se', se'')$, or the application of a unary $\ominus$ or a binary operator $\otimes$.

$$se := n \mid s \mid \mathbf{ite}(se, se', se'') \mid \ominus se \mid se \otimes se'$$

Most operational rules of $\rightarrow_{\mathcal{S}}$ are straightforward extensions of the concrete semantics, and we omit them. The primary differences w.r.t. the concrete μAsm semantics arise when handling memory operations and control-flow statements with symbolic values, which are formalized in the operational semantics rules below.

$$\boxed{\sigma^{\mathcal{S}} \xrightarrow{\tau}_{\mathcal{S}} \sigma^{\mathcal{S}\prime}}$$

(Load-Symb)
$$\frac{p(sa(\mathbf{pc})) = \mathbf{load}\ x, e \qquad x \neq \mathbf{pc} \qquad se = [\![e]\!](sa) \qquad se' = \mathbf{read}(sm, se)}{\langle p, sm, sa \rangle \xrightarrow{\texttt{load}\ se}_{\mathcal{S}} \langle p, sm, sa[\mathbf{pc} \mapsto sa(\mathbf{pc}) + 1, x \mapsto se'] \rangle}$$

(Store-Symb)
$$\frac{p(sa(\mathbf{pc})) = \mathbf{store}\ x, e \qquad se = [\![e]\!](sa) \qquad sm' = \mathbf{write}(sm, se, sa(x))}{\langle p, sm, sa \rangle \xrightarrow{\texttt{store}\ se}_{\mathcal{S}} \langle p, sm', sa[\mathbf{pc} \mapsto a(\mathbf{pc}) + 1] \rangle}$$

(Beqz-Conc-Sat)
$$\frac{p(sa(\mathbf{pc})) = \mathbf{beqz}\ x, \ell \qquad sa(x) = 0}{\langle p, sm, sa \rangle \xrightarrow{\texttt{pc}\ \ell \cdot \texttt{symPc}(\top)}_{\mathcal{S}} \langle p, sm, sa[\mathbf{pc} \mapsto \ell] \rangle}$$

(Beqz-Conc-Unsat)
$$\frac{p(sa(\mathbf{pc})) = \mathbf{beqz}\ x, \ell \qquad sa(x) \neq 0}{\langle p, sm, sa \rangle \xrightarrow{\texttt{pc}\ sa(\mathbf{pc})+1 \cdot \texttt{symPc}(\top)}_{\mathcal{S}} \langle p, sm, sa[\mathbf{pc} \mapsto a(\mathbf{pc}) + 1] \rangle}$$

(Beqz-Symb-Sat)
$$\frac{p(sa(\mathbf{pc})) = \mathbf{beqz}\ x, \ell \qquad sa(x) \notin Vals}{\langle p, sm, sa \rangle \xrightarrow{\texttt{pc}\ \ell \cdot \texttt{symPc}(sa(x)=0)}_{\mathcal{S}} \langle p, sm, sa[\mathbf{pc} \mapsto \ell] \rangle}$$

(Beqz-Symb-Unsat)
$$\frac{p(sa(\mathbf{pc})) = \mathbf{beqz}\ x, \ell \qquad sa(x) \notin Vals}{\langle p, sm, sa \rangle \xrightarrow{\texttt{pc}\ sa(\mathbf{pc})+1 \cdot \texttt{symPc}(sa(x)\neq 0)}_{\mathcal{S}} \langle p, sm, sa[\mathbf{pc} \mapsto sa(\mathbf{pc}) + 1] \rangle}$$

(Jmp-Conc)
$$\frac{p(sa(\mathbf{pc})) = \mathbf{jmp}\ e \qquad \ell = [\![e]\!](sa) \qquad \ell \in Vals}{\langle p, sm, sa \rangle \xrightarrow{\texttt{pc}\ \ell \cdot \texttt{symPc}(\top)}_{s} \langle p, sm, sa[\mathbf{pc} \mapsto \ell] \rangle}$$

(Jmp-Symb)
$$\frac{p(sa(\mathbf{pc})) = \mathbf{jmp}\ e \qquad [\![e]\!](sa) \notin Vals \qquad \ell \in Vals}{\langle p, sm, sa \rangle \xrightarrow{\texttt{pc}\ \ell \cdot \texttt{symPc}([\![e]\!](sa)=\ell)}_{s} \langle p, sm, sa[\mathbf{pc} \mapsto \ell] \rangle}$$

When symbolically executing a program, we may produce observations whose values are symbolic. To account for this, we introduce symbolic observations of the form `load` se and `store` se for symbolic load and store instructions.

Furthermore, because of the symbolic execution, we need to keep track if certain paths in our program are feasible. We encode this information using `symPc`(se) observations, which record the choices made at branches and jumps. We can see these observations in action in Rule Beqz-Symb-Unsat and Rule Beqz-Symb-Sat for branches. Rule Jmp-Conc and Rule Jmp-Symb record them analogously for indirect jumps. Because x cannot be evaluated to a value, the symbolic semantics decides if the branch is taken or not and records this decision in the observation `symPc`$(sa(x) = 0)$ Without these observations, we could later concretize x to a value unequal to 0 even though the branch was not taken, making the path infeasible in a concrete program execution. To explore all paths of the program, our symbolic tool will negate these

symbolic branching observations to explore the other branches as well. The path condition $pthCnd(\overline{\tau_S})$=$\bigwedge_{\texttt{symPc}(se)\in\tau} se$ of trace τ is the conjunction of all symbolic branching conditions in τ.

The value of a symbolic expression se depends on a *valuation function* $\mu : SymbVals \to Vals$ mapping symbolic values to concrete ones. The evaluation $\mu(se)$ is also standard. We write $\mu \vDash se$ to denote that symbolic expression se is *satisfiable* for a valuation μ, i.e., $\mu(se) \neq 0$. Every valuation that satisfies a symbolic run's path condition maps the symbolic run to a concrete one. Finally, we write $\vDash se$ to denote that there exists a valuation μ such that $\mu \vDash se$.

The *symbolic non-speculative behaviour* $Beh^{\mathcal{S}}_{NS}(p)$ of a program p is the set of all traces generated by all possible initial states for program p.

Theorem 6 connects the symbolic and concrete non-speculative semantics via the concretization function μ:

Theorem 6 (NS: Symbolic Consistency). $Beh_{NS}(p) = \mu(Beh^{\mathcal{S}}_{NS}(p))$.

This allows us to use the symbolic non-speculative semantics in our tool Spectector with confidence that it behaves equivalently to the concrete non-speculative semantics.

9.2 Symbolic Always-Mispredict Semantics

To enable automated verification, Spectector implements a symbolic always-mispredict semantics. Structurally, this semantics perfectly mirrors the concrete always-mispredict semantics introduced earlier, with the sole distinction that it operates over symbolic expressions and relies on the symbolic non-speculative step relation $\rightarrow_{\mathcal{S}}$ rather than its concrete counterpart. Thus, we omit details related to this semantics and refer to the technical report [30].

We denote by $\mu((\langle p, \sigma^{\mathcal{S}}\rangle)\!\!\oint^{\omega}_{x}\overline{\tau_S})$ the set $\{(\langle p, \sigma^{\mathcal{S}}\rangle)\!\!\oint^{\omega}_{x}(\mu(\overline{\tau_S})) \mid \mu \vDash pthCnd(\overline{\tau_S})\}$ and lift it to $Beh^{\mathcal{S}}_{x}(p)$. Here $(\langle p, \sigma^{\mathcal{S}}\rangle)\!\!\oint^{\omega}_{x}\overline{\tau_S}$ denotes the run of the program p with initial configuration $\sigma^{\mathcal{S}}$ generating $\overline{\tau_S}$ for the symbolic always-mispredict semantics x.

Symbolic Consisteny. Fundamentally, a valid symbolic analysis must faithfully capture all possible concrete executions of a program. The symbolic always-mispredict semantic should explore the exact same set of execution paths and generate the same observations as its concrete counterpart, merely operating over symbolic values and constraints. We formalize this correctness requirement in Property 4, stating that the concrete always-mispredict behaviour can be exactly recovered from the symbolic behaviour by applying the concretization function.

Property 4 (Symbolic Consistency AM). $Beh^{\omega}_{x}(p) = \mu(Beh^{\mathcal{S}}_{x}(p))$

Because we have proven that all our speculative semantics and their combinations satisfy the requirements of our framework (i.e., they are well-formed speculative semantics), they inherently satisfy Property 4. This allows us to use all of the symbolic semantics in our tool Spectector with confidence that they behave equivalently to the concrete semantics.

Example 12 (Symbolic Trace for Example 1)**.** Executing the program from Example 1 under the symbolic speculative semantics $\oint_{\mathrm{B}}$ with speculative window 2 yields the following two symbolic traces:

$$\tau_1 := \texttt{symPc}(\texttt{y} < \texttt{size}) \cdot \texttt{start}_{\mathrm{B}}\ 0 \cdot \texttt{pc}\ 2 \cdot \texttt{pc}\ 10 \cdot \texttt{rlb}_{\mathrm{B}}\ 0 \cdot \texttt{pc}\ 3 \cdot \texttt{load}\ \texttt{A} + \texttt{y} \cdot \texttt{load}\ \texttt{B} + \mathbf{read}(sm, (\texttt{A} + \texttt{y})) * 512$$

$$\tau_2 := \texttt{symPc}(\texttt{y} \geq \texttt{size}) \cdot \texttt{start}_{\mathrm{B}}\ 0 \cdot \texttt{pc}\ 3 \cdot \texttt{load}\ \texttt{A} + \texttt{y} \cdot \texttt{load}\ \texttt{B} + \mathbf{read}(sm, (\texttt{A} + \texttt{y})) * 512 \cdot \texttt{rlb}_{\mathrm{B}}\ 0 \cdot \texttt{pc}\ 2 \cdot \texttt{pc}\ 10$$

Algorithm 1 SPECTECTOR program analysis

Input: A program p, a security policy P, a speculative window $\omega \in \mathbb{N}$.
Output: SECURE if p satisfies speculative non-interference with respect to the policy P and speculative semantics $\circlearrowleft_x$; INSECURE otherwise

1: **procedure** SPECTECTOR(p, P, ω)
2: **for** each symbolic run $\overline{\tau} \in Beh_x^S(p)$ **do**
3: **if** MEMLEAK($\overline{\tau}, P$) $\vee$ CTRLLEAK($\overline{\tau}, P$) **then**
4: **return** INSECURE
5: **return** SECURE

6: **procedure** MEMLEAK($\overline{\tau}, P$)
7: $\psi \leftarrow pthCnd(\overline{\tau})_{1\wedge 2} \wedge polEqv(P) \wedge$
$obsEqv(\overline{\tau}\restriction_{ns}) \wedge \neg obsEqv(\overline{\tau}\restriction_{se})$
8: **return** SATISFIABLE(ψ)

9: **procedure** CTRLLEAK($\overline{\tau}, P$)
10: **for** each prefix $\nu \cdot$ `symPc`(se) of $\overline{\tau}\restriction_{se}$ **do**
11: $\psi \leftarrow pthCnd(\overline{\tau}\restriction_{ns} \cdot \nu)_{1\wedge 2} \wedge polEqv(P) \wedge$
$obsEqv(\overline{\tau}\restriction_{ns}) \wedge \neg sameSymbPc(se)$
12: **if** SATISFIABLE(ψ) **then**
13: **return** $\top$
14: **return** $\bot$

9.3 The Spectector Algorithm

The SPECTECTOR program analysis approach is presented in Algorithm 1. It relies on two procedures: MEMLEAK and CTRLLEAK, to detect leaks resulting from memory and control-flow instructions, respectively. We start by discussing the SPECTECTOR algorithm and next explain the MEMLEAK and CTRLLEAK procedures.

SPECTECTOR takes as input a program p, a policy P specifying the non-sensitive information, and a speculative window ω. The algorithm iterates over all symbolic runs produced by the symbolic always-mispredict speculative semantics (lines 2-4). For each trace $\overline{\tau} \in Beh_x^S(p)$, the algorithm checks whether $\overline{\tau}$ speculatively leaks information through memory accesses or control-flow instructions. If this is the case, then SPECTECTOR has found a witness of a speculative leak and it reports p as INSECURE. If none of the traces contains speculative leaks, the algorithm terminates returning SECURE (line 5).

Detecting leaks caused by memory accesses. The procedure MEMLEAK takes as input a trace $\overline{\tau}$ and a policy P, and it determines whether $\overline{\tau}$ leaks information through symbolic `load` and `store` observations. The check is expressed as a satisfiability check of a constraint ψ. The construction of ψ is inspired by self-composition [13], which reduces reasoning about *pairs* of program runs to reasoning about single runs by replacing each symbolic variable x with two copies x_1 and x_2. We lift the subscript notation to symbolic expressions.

The constraint ψ is the conjunction of four formulas:

- $pthCnd(\tau)_{1\wedge 2}$ stands for $pthCnd(\overline{\tau})_1 \wedge pthCnd(\overline{\tau})_2$, which ensures that both runs follow the path associated with $\overline{\tau}$.
- $polEqv(P)$ introduces constraints $x_1 = x_2$ for each register $x \in P$ and $\mathbf{read}(sm_1, n) = \mathbf{read}(sm_2, n)$ for each memory location $n \in P$, which ensure that both runs agree on all non-sensitive inputs.

- $obsEqv(\overline{\tau}\restriction_{ns})$ introduces a constraint $se_1 = se_2$ for each `load` se or `store` se in $\tau\restriction_{ns}$, which ensures that the non-speculative observations associated with memory accesses are the same in both runs.
- $\neg obsEqv(\overline{\tau}\restriction_{se})$ ensures that at least one speculative observation associated with memory accesses differs among the two runs.

If ψ is satisfiable, there are two P-indistinguishable configurations that produce the same non-speculative traces (since $pthCnd(\overline{\tau})_{1\wedge 2} \wedge polEqv(P) \wedge obsEqv(\overline{\tau}\restriction_{ns})$ is satisfied) and whose speculative traces differ in a memory access observation (since $\neg obsEqv(\overline{\tau}\restriction_{se})$ is satisfied), i.e. a violation of SNI.

Detecting leaks caused by control-flow instructions. To detect leaks caused by control-flow instructions, CtrlLeak checks whether there are two traces in τ's concretization that agree on the outcomes of all non-speculative branch, jump and return instructions, while differing in the outcome of at least one speculatively executed branch, jump or return instruction.

In addition to $pthCnd(\overline{\tau})$, $obsEqv(\overline{\tau})$, and $polEqv(P)$, the procedure relies on the function $sameSymbPc(se)$ that introduces the constraint $se_1 \leftrightarrow se_2$ ensuring that se is satisfied in one concretization iff it is satisfied in the other.

CtrlLeak checks, for each prefix $v \cdot$ `symPc`(se) in $\overline{\tau}$'s speculative projection $\overline{\tau}\restriction_{se}$, the satisfiability of the conjunction of $pthCnd(\overline{\tau}\restriction_{ns} \cdot v)_{1\wedge 2}$, $polEqv(P)$, $obsEqv(\overline{\tau}\restriction_{ns})$, and $\neg sameSymbPc(se)$. Whenever the formula is satisfiable, there are two P-indistinguishable configurations that produce the same non-speculative traces, but whose speculative traces differ on control-flow observations, i.e. a violation of SNI.

Example 13 (Example Application of Spectector)**.** Consider the trace from Example 12:

$$\overline{\tau} := \texttt{symPc}(\texttt{y} \geq \texttt{size}) \cdot \texttt{start}_{\texttt{B}}\ 0 \cdot \texttt{pc}\ 3 \cdot \texttt{load}\ \texttt{A} + \texttt{y} \cdot \texttt{load}\ \texttt{B} + \textbf{read}(sm, (\texttt{A} + \texttt{y})) * 512 \cdot \texttt{rlb}_{\texttt{B}}\ 0 \cdot \texttt{pc}\ 2 \cdot \texttt{pc}\ 10$$

MemLeak detects a leak caused by the observation `load` $\texttt{B} + \textbf{read}(sm, (\texttt{A} + \texttt{y})) * 512$. Specifically, it detects that there are distinct symbolic valuations that agree on the non-speculative observations but disagree on the value of `load` $\texttt{B} + \textbf{read}(sm, (\texttt{A} + \texttt{y})) * 512$. That is, the observation depends on sensitive information that is not disclosed by $\overline{\tau}$'s non-speculative projection.

Soundness and Completeness. Theorem 7 states that Spectector deems secure only speculatively non-interferent programs, and all detected leaks are actual violations of SNI.

Theorem 7 (Correctness Spectector). *If* Spectector(p, P, ω) *terminates, then* Spectector(p, P, ω) = Secure *iff the program p satisfies speculative non-interference w.r.t. the policy P and all prediction oracles $\mathcal{O}$ with speculative window at most ω.*

The theorem follows from Oracle Overapproximation and Symbolic Consistency of the speculative semantics. Importantly, both conditions are part of our well-formed speculative semantics definition (Definition 3). Since $\mathcal{O}_{\texttt{B}}$, $\mathcal{O}_{\texttt{J}}$, $\mathcal{O}_{\texttt{S}}$, $\mathcal{O}_{\texttt{R}}$ and $\mathcal{O}_{\texttt{SLS}}$ are well-formed speculative semantics $\vdash \mathcal{O}_x$ *WFSS* and our combinations are well-formed (Definition 4), we have that all the combinations are *WFSS* as well. Thus, we get Theorem 7 for free for all the combinations.

Theorem 7 characterizes the Spectector *algorithm*, under the assumptions that concolic exploration is exhaustive, the SMT solver resolves every query, and the procedure terminates. Whenever our *implementation* returns a verdict, it respects these guarantees and never reports an incorrect result. On programs with loops or too many paths, or when the SMT solver cannot resolve a query within its time budget, Spectector reports a timeout instead. Because the SMT theories we use are decidable, such a timeout reflects resource limits rather than a fundamental obstacle.

9.4 Spectector Implementation

We implement our approach in our tool Spectector, which is available at [37]. The tool, which is implemented on top of the Ciao logic programming system [41], consists of three components: a front end that translates x86 assembly programs into μAsm, a core engine implementing Algorithm 1, and a back end handling SMT queries.

x86 Front End. The front end translates AT&T/GAS and Intel-style assembly files into μAsm[8]. It currently supports over 120 instructions: data movement instructions (**mov**, etc.), logical, arithmetic, and comparison instructions (**xor**, **add**, **cmp**, etc.), branching and jumping instructions (**jae**, **jmp**, etc.), conditional moves (**cmovae**, etc.), stack manipulation (**push**, **pop**, etc.), and function calls[9] (**call**, **ret**).

It currently does not support privileged x86 instructions, e.g., for handling model specific registers and virtual memory. Further, it does not support sub-registers (like `eax`, `ah`, and `al`) and unaligned memory accesses, i.e., we assume that only 64-bit words are read/written at each address without overlaps. Finally, the translation currently maps symbolic address names to μAsm instruction addresses, limiting arithmetic on code addresses.

Core Engine. The core engine implements Algorithm 1. It relies on a concolic approach to implement symbolic execution that performs a depth-first exploration of the symbolic runs. Starting from a concrete initial configuration, the engine executes the program under the symbolic always-mispredict speculative semantics while keeping track of the symbolic configuration and path condition. It discovers new runs by iteratively negating the last (not previously negated) conjunct in the path condition until it finds a new initial configuration, which is then used to re-execute the program concolically. In our current implementation, indirect jumps and returns are *not* included in the path conditions, and thus new symbolic runs and corresponding inputs are only discovered based on negated branch conditions.[10] This process is interleaved with the MemLeak and CtrlLeak checks and iterates until a leak is found or all paths have been explored.

SMT Back End. The Z3 SMT solver [26] acts as a back end for checking satisfiability and finding models of symbolic expressions using the Bitvector and Array theories, which are used to model registers and memory. The implementation currently does not rely on incremental solving, since it was less efficient than one-shot solving for the selected theories.

Implementation of the Speculative Semantics and the Combinations in Spectector. We implemented all our semantics (the symbolic versions of $\mathcal{A}_{\mathsf{B}}$, $\mathcal{A}_{\mathsf{S}}$, $\mathcal{A}_{\mathsf{J}}$, $\mathcal{A}_{\mathsf{R}}$ and $\mathcal{A}_{\mathsf{SLS}}$ plus all 18 compositions from Section 8) in Spectector. The implementation of compositions closely follows the structure of our framework. As in Section 8, selecting one of the composed semantics in Spectector sets the metaparameter Z, which is used to delegate back to the correct individual semantics.

10 EVALUATION

This section reports on two case studies in which we apply Spectector to analyze the security of programs. The goals of the first case study are (1) to determine whether speculative non-interference realistically captures speculative leaks and (2) to assess Spectector 's precision Therefore, we analyze various examples targeting the different Spectre versions we want to capture with our speculative semantics (Section 10.1).

The goal of our second case study is to investigate if the combined speculative semantics allow us to capture stronger attacks that are not captured by individual semantics. Thus, we create code snippets that are only exploitable using a combination of different Spectre attacks and check if Spectector using the combined semantics can detect leaks (Section 10.2).

[8] This translation is currently unverified.
[9] We model the so-called "near calls", where the callee is in the same code segment as the caller.
[10] We plan to remove this limitation in a future release of our tool.

All code snippets as well as all scripts to reproduce our results are available in our public repository at https://spectector.github.io/.

10.1 Case Study: Detecting Leaks w.r.t. Individual Speculative Semantics

Using Spectector, we analyze a corpus of 41 microbenchmarks containing speculative leaks generated by different speculation mechanisms in isolation (Section 6). With these experiments, we aim to show that speculative non-interference and our individual speculative semantics can correctly identify speculative leaks associated with speculation over branches, indirect jumps, store-bypasses and return instructions.

10.1.1 Benchmarks. We start from 41 snippets of code containing leaks resulting from speculation over **beqz**, **store**, **load**, **jmp**, and **ret** instructions (and their combinations). We compile these snippets using various compilers, compiler options, and mitigations, thereby obtaining a corpus of 291 x64 assembly programs, which we analyse with Spectector. Next, we describe next in detail the composition of our benchmark corpus:

- **Spectre-PHT:** 15 snippets are variants of the Spectre-PHT vulnerability by Kocher [48]. For each of the 15 snippets, we analyse the assembly programs obtained using different compilers and compiler options. For compilers, we rely on three state-of-the-art compilers: Microsoft Visual C++ versions v19.15.26732.1 and v19.20.27317.96, Intel Icc v19.0.0.117, and Clang v7.0.0 and compile each snippet using two different *optimization levels* (`-O0` and `-O2`) and three *mitigation levels*: (a) Unp: we compile without any Spectre mitigations. (b) Fen: we compile with automated injection of speculation barriers.[11] (c) Slh: we compile using speculative load hardening.[12] Compiling each of the 15 examples from [48] with each of the 3 compilers, each of the 2 optimization levels, and each of the 2-3 mitigation levels, yields a corpus of 240 x64 assembly programs. For each program, we specify a security policy that flags as "low" all registers and memory locations that can either be controlled by the adversary or can be assumed to be public. This includes variables `y` and `size`, and the base addresses of the arrays `A` and `B` as well as the stack pointer.
- **Spectre-BTB:** 2 snippets are variants of the Spectre-BTB vulnerability. They exploit speculation over indirect jump instructions and were created by ourselves. For each snippet, we also analyze manually patched versions obtained by inserting lfences after every endbr instruction and automatically patched versions using the retpoline [45] countermeasure.[13] This results in a corpus of 6 x64 assembly programs.
To make the analysis tractable, we use the `endbr` instruction for CFI (see Section 6.5). Note that these `endbr` instructions were automatically added by the compiler.
- **Spectre-STL:** 13 snippets are variants of the Spectre-STL vulnerability. They exploit speculation over memory disambiguation, and they have been used as benchmarks in prior work [25, 65]. For each snippet, we also analyze a patched version where a manually inserted lfence instruction stops speculation over store-bypasses and prevents the leak. This results in a corpus of 26 x64 assembly programs.
- **Spectre-RSB:** 5 snippets are variants of the Spectre-RSB vulnerability. They exploit speculation over return instructions, and they are obtained from the `safeside` [34] and `transientfail` [18] projects[14]. For each snippet, we also analyze manually patched versions obtained by (1) inserting lfences after call instructions (i.e., at the instruction

[11] Fences are supported by Clang with the flag `-x86-speculativeload-hardening-lfence`, by Icc with `-mconditionalbranch=all-fix`, and by Visual C++ with `/Qspectre`.
[12] Speculative load hardening is supported by Clang with the flag `-x86-speculative-load-hardening`.
[13] Retpoline is supported by Gcc with the flag `-mindirect-branch=thunk`.
[14] Out of the three Spectre-RSB examples from `safeside` [34], we analyze the only one that works against an acyclic RSB like the one supported by $\mathcal{A}_R$. Programs *ca_ip*, *ca_oop*, and *sa_ip* from `transientfail` [18] rely on concurrent execution. Since Spectector does not support concurrency, we hardcode the worst-case interleaving in terms of speculative leakage in our benchmark.

address where **ret** speculatively returns), and (2) using the modified `retpoline` defense proposed in [53, Section 6.1]. This results in a corpus of 15 x64 assembly programs.

- **Spectre-SLS:** 2 snippets are variants of the Spectre-SLS vulnerability. They exploit speculation over return instructions and were created by ourselves.

For each snippets, we also analyze a manually patched version obtained by inserting LFENCEs after every return instruction, resulting in a corpus of 4 x64 assembly programs.[15]

10.1.2 Experimental Setup. The benchmarks for **Spectre-PHT** are compiled as described in the section above. The benchmarks for **Spectre-STL**, **Spectre-RSB**, and **Spectre-SLS** are implemented in C and compiled with GCC 11.1.0 and we manually inserted `lfence`/`modified retpoline` countermeasures in the patched versions. The benchmarks for **Spectre-BTI** are also implemented in C and `lfences` were inserted manually while retpoline was added automatically by the compiler into the patched programs.

The benchmarks for **Spectre-PHT** were run on a Linux machine (kernel 4.9.0-8- amd64) with Debian 9.0, a Xeon Gold 6154 CPU, and 64 GB of RAM. All other experiments were run on a laptop with a Dual Core Intel Core i5-7200U CPU and 8GB of RAM.

The speculation window ω bounds how many instructions may execute speculatively before a misprediction resolves. We use $\omega = 200$ by default, motivated by typical sizes of the reorder buffer [28], which limits the lengths of speculative transactions in modern microarchitectures.

Ex.	VCC						ICC				CLANG					
	UNP		FEN 19.15		FEN 19.20		UNP		FEN		UNP		FEN		SLH	
	-O0	-O2	-O0	-O2	-O0	-O2	-O0	-O2	-O0	-O2	-O0	-O2	-O0	-O2	-O0	-O2
01	○	○	●	●	●	●	○	○	●	●	○	○	●	●	●	●
02	○	○	●	●	●	●	○	○	●	●	○	○	●	●	●	●
03	○	○	●	○	●	●	○	○	●	●	○	○	●	●	●	●
04	○	○	○	○	●	●	○	○	●	●	○	○	●	●	●	●
05	○	○	●	○	●	○	○	○	●	●	○	○	●	●	●	●
06	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●
07	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●
08	○	●	○	●	○	●	○	●	●	●	○	●	●	●	●	●
09	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●
10	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	○
11	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●
12	○	○	○	○	●	●	○	○	●	●	○	○	●	●	●	●
13	○	○	○	○	○	○	○	○	●	●	○	○	●	●	●	●
14	○	○	○	○	●	●	○	○	●	●	○	○	●	●	●	●
15	○	○	○	○	○	○	○	○	●	●	○	○	●	●	○	●

Fig. 12. Analysis of Kocher's examples [48], compiled with different compilers and options. For each of the 15 examples, we analyzed the unpatched version (denoted by UNP), the version patched with speculation barriers (denoted by FEN), and the version patched using speculative load hardening (denoted by SLH). Programs have been compiled without optimizations (`-O0`) or with compiler optimizations (`-O2`) using the compilers VISUAL C++ (two versions), ICC, and CLANG. ○ denotes that SPECTECTOR detects a speculative leak, whereas ● indicates that SPECTECTOR proves the program secure.

[15] We note that compilers like CLANG have the option `mharden-sls-all` to automatically protect the code. However, this countermeasure inserts an `int3` instruction after every return on x86. Since interrupts and exceptions are not handled by SPECTECTOR, they cannot be translated to μASM. An `lfence` behaves similarly in this case.

10.1.3 Results for Spectre-PHT. Figure 12 depicts the results of applying Spectector to the 240 **Spectre-PHT** examples. We highlight the following findings:

- Spectector detects the speculative leaks in almost all unprotected programs, for all compilers (see the Unp columns). The exception is Example #8, which uses a conditional expression instead of the if statement of Listing 1:

```
temp &= B[A[y<size?(y+1):0]*512];
```

At optimization level `-O0`, this is translated to a (vulnerable) branch instruction by all compilers, and at level `-O2` to a (safe) conditional move, thus closing the leak. See Appendix A.1 for the corresponding Clang assembly.

- The Clang and Intel Icc compilers defensively insert fences after each branch instruction, and Spectector can prove security for all cases (see the Fen columns for Clang and Icc). In Example #8 with options `-O2` and Fen, Icc inserts an **lfence** instruction, even though the baseline relies on a conditional move, see line 10 below. This **lfence** is unnecessary according to our semantics, but may close leaks on processors that speculate over conditional moves.

```
1  mov   y, %rdi
2  lea   1(%rdi), %rdx
3  mov   size, %rax
4  xor   %rcx, %rcx
5  cmp   %rax, %rdi
6  cmovb    %rdx, %rcx
7  mov   temp, %r8b
8  mov   A(%rcx), %rsi
9  shl   $9, %rsi
10 lfence
11 and   B(%rsi), %r8b
12 mov   %r8b, temp
```

- For the Visual C++ compiler, Spectector automatically detects all leaks pointed out in [48] (see the Fen 19.15 `-O2` column for Vcc). Our analysis differs from Kocher's only on Example #8, where the compiler v19.15.26732.1 introduces a safe conditional move, as explained above. Moreover, without compiler optimizations (which is not considered in [48]), Spectector establishes the security of Examples #3 and #5 (see the Fen 19.15 `-O0` column). The latest Vcc compiler additionally mitigates the leaks in Examples #4, #12, and #14 (see the Fen 19.20 column).
- Spectector can prove the security of speculative load hardening in Clang (see the Slh column for Clang), except for Example #10 with `-O2` and Example #15 with `-O0`. We stress that this is a translation-validation result: Spectector certifies the security of individual hardened programs, not of the hardening pass itself. Indeed, Spectector flags two hardened programs that remain insecure, matching the known result that Clang's speculative load hardening is not secure in general [82].

Example 10 with Speculative Load Hardening. Example #10 differs from Listing 1 in that it leaks sensitive information into the microarchitectural state by conditionally reading the content of `B[0]`, depending on the value of `A[y]`.

```
if (y < size)
  if (A[y] == k)
```

```
3       temp &= B[0];
```

SPECTECTOR proves the security of the program produced with CLANG -O0 and speculative load hardening. However, at optimization level -O2, CLANG outputs the following code that SPECTECTOR reports as insecure.

```
1  mov     size, %rdx
2  mov     y, %rbx
3  mov     $0, %rax
4  cmp     %rbx, %rdx
5  jbe     END
6  cmovbe  $-1, %rax
7  or      %rax, %rbx
8  mov     k, %rcx
9  cmp     %rcx, A(%rbx)
10 jne     END
11 cmovne  $-1, %rax
12 mov     B, %rcx
13 and     %rcx, temp
14 jmp     END
```

The reason for this is that CLANG masks only the register %rbx that contains the index of the memory access A[y], cf. lines 6–7. However, it does *not* mask the value that is read from A[y]. As a result, the comparison at line 9 speculatively leaks (via the jump target) whether the content of A[0xFF...FF] is k. SPECTECTOR detects this subtle leak and flags a violation of speculative non-interference.

While this example nicely illustrates the scope of SPECTECTOR, it is likely not a problem in practice: First, the leak may be mitigated by how data dependencies are handled in modern out-of-order CPUs. Specifically, the conditional move in line 6 relies on the comparison in Line 4. If executing the conditional leak effectively terminates speculation, the reported leak is spurious. Second, the leak can be mitigated at the OS-level by ensuring that 0xFF...FF is not mapped in the page tables, or that the value of A[0xFF...FF] does not contain any secret. [16]Such contextual information can be expressed with policies (see Section 5.1) to improve the precision of the analysis.

Results for Spectre-BTB Figure 13c reports the analysis of the programs in the **Spectre-BTB** benchmark. Using the $\lightning_J$ semantics, SPECTECTOR detected leaks (i.e., violations of SNI) in all unpatched programs. Furthermore, SPECTECTOR proved that the manually patched programs using lfence and the automatically patched programs using the retpoline countermeasure satisfy SNI, i.e., they are free of speculative leaks.

10.1.4 Results for Spectre-STL. Figure 13a reports the results of analysing the programs in the **Spectre-STL** benchmark.[17] Using the $\lightning_S$ semantics, SPECTECTOR successfully detected leaks (i.e., violations of SNI) in all unpatched programs, except programs 03, 09, and 12 which do not contain speculative leaks (consistently with other analysis results [25, 65]). Observe that Binsec/Haunted [25] flags program 13 as secure since the program can *only* speculatively

[16]Personal Communication with C. Marinas

[17]We had to slightly modify programs 02, 05, and 06 due to limitations of SPECTECTOR's x86 front-end when dealing with global values (programs 05 and 06) and 32-bit addressing (program 02). We had to limit the speculation window, due to vanilla SPECTECTOR's limitations in symbolic execution, when analyzing program 09, which contains a loop.

Test case		$\lightning_S$ None	$\lightning_S$ Fence
case01	(I)	○	●
case02	(I)	○	●
case03	(S)	●	●
case04	(I)	○	●
case05	(I)	○	●
case06	(I)	○	●
case07	(I)	○	●
case08	(I)	○	●
case09	(S)	●	●
case10	(I)	○	●
case11	(I)	○	●
case12	(S)	●	●
case13	(I)	○	●

(a) Results for the Spectre-STL programs under the $\lightning_S$ semantics against unpatched programs (column "None") and programs patched with `lfence` (column "Fence").

Test case		$\lightning_R$ None	$\lightning_R$ Fence	$\lightning_R$ Mod. Retpoline
ret2spec_c_d	(I)	○	●	●
ca_ip	(I)	○	●	●
ca_oop	(I)	○	●	●
sa_ip	(I)	○	●	●
sa_oop	(I)	○	●	●

(b) Results for the Spectre-RSB programs under the $\lightning_R$ semantics against unpatched programs (column "None"), programs patched with `lfence` (column "Fence"), and programs patched with the modified `retpoline` defense proposed in [53, S6.1] (column "Mod. Retpoline").

Test case		$\lightning_J$ None	$\lightning_J$ Fence	$\lightning_J$ Retpoline
caseJ01	(I)	○	●	●
caseJ02	(I)	○	●	●

(c) Results for the Spectre-BTB programs under the $\lightning_J$ semantics against unpatched programs (column "None"), programs patched with `lfence` (column "Fence"), and programs patched with the `retpoline` defense proposed in [45] (column "Retpoline").

Test case		$\lightning_{SLS}$ None	$\lightning_{SLS}$ Fence
caseSLS01	(I)	○	●
caseSLS02	(I)	○	●

(d) Results for the Spectre-SLS programs under the $\lightning_{SLS}$ semantics against unpatched programs (column "None"), programs patched with `lfence` (column "Fence").

Fig. 13. Result of the analysis of our benchmarks for $\lightning_J$, $\lightning_S$, $\lightning_R$ and $\lightning_{SLS}$. For each program, ○ denotes that Spectector finds a violation of SNI under the corresponding semantics, whereas ● denotes that Spectector proves the program secure under the semantics. Next to each program, we report if the program is Secure or Insecure in its unpatched version.

leak initial values from the stack, which Binsec/Haunted treats as public by default [14]. Since we assume initial memory values to be secret (like Ponce de León and Kinder [65]), Spectector correctly detected the leak in program 13. Spectector also successfully proved that all patched programs (where an `lfence` is added between **store** instructions) satisfy SNI and are free of speculative leaks.

10.1.5 Results for Spectre-RSB. Figure 13b reports the analysis results on the **Spectre-RSB** programs. Using $\lightning_R$, Spectector successfully detected leaks in all unpatched programs. Moreover, Spectector successfully proved that the patched programs where a `lfence` instruction is added after every **call** satisfy SNI, i.e., they are free of speculative leaks. Spectector also successfully proved secure the programs patched using the modified `retpoline` defense proposed by Maisuradze and Rossow [53], which replaces return instructions with a construct that traps the speculation in an infinite loop.

10.1.6 Results for Spectre-SLS. Figure 13d reports the analysis results on the **Spectre-SLS** programs. Using $\lightning_{SLS}$, Spectector successfully detected leaks in all unpatched programs and proved the security (i.e. SNI) of the patched programs where an lfence instruction was added after every return instruction.

10.2 Case Study: Detecting Leaks w.r.t. Combined Speculative Semantics

Using Spectector, we analyze a corpus of 18 microbenchmarks containing speculative leaks generated by a combination of speculation mechanisms (for all composed semantics from Section 8). In particular, we consider one microbenchmark for each of all possible combinations of the $\mathcal{A}_{\mathsf{B}}$, $\mathcal{A}_{\mathsf{J}}$, $\mathcal{A}_{\mathsf{S}}$, $\mathcal{A}_{\mathsf{R}}$ and $\mathcal{A}_{\mathsf{SLS}}$ semantics, i.e., 18 combined semantics (as mentioned, combinations containing both $\mathcal{A}_{\mathsf{R}}$ and $\mathcal{A}_{\mathsf{SLS}}$ are not possible). With these experiments, we aim to show that our combined semantics can detect novel leaks that are otherwise undetectable when considering single speculation mechanisms in isolation.

10.2.1 Benchmarks. For each of the 18 combined semantics, we manually crafted an example μAsm program that has a speculative leak under that specific combination. For each program, we also analyze a manually patched version where `lfence` instructions prevent speculative leaks. We refer to this benchmark, consisting of 36 μAsm programs, as **Spectre-Comb**.

10.2.2 Experimental Setup. For each μAsm program in **Spectre-Comb**, we run Spectector with all individual speculative semantics as well as all their combinations. All experiments were run on a laptop with a Dual Core Intel Core i5-7200U CPU and 8GB of RAM.

10.2.3 Results for Spectre-Comb. Table 1 reports the results of our analysis on the **Spectre-Comb** programs, which involve leaks arising from a combination of multiple speculation mechanisms.

Spectector equipped with the individual semantics $\mathcal{A}_{\mathsf{B}}$, $\mathcal{A}_{\mathsf{J}}$, $\mathcal{A}_{\mathsf{S}}$, $\mathcal{A}_{\mathsf{R}}$ and $\mathcal{A}_{\mathsf{SLS}}$ is not able to detect the speculative leaks in any of the 18 programs and, therefore, proves them secure. This is expected since the programs contain leaks that arise from a combination of semantics.

Spectector can successfully identify leaks in all programs when using the semantics corresponding to that example. For example, **comb25** is proven insecure by the respective semantics $\mathcal{A}_{\mathsf{S+R}}$.

Furthermore, as Figure 9 suggests, all 'stronger' semantics (higher in the combination lattice) also detect the vulnerability in the programs made for a 'weaker' semantics (lower in the combination lattice) and the 'weaker' ones do not detect the vulnerability of programs made for 'stronger' semantics. For example, **comb25** is also proven to be insecure by Spectector when using $\mathcal{A}_{\mathsf{B+J+R}}$, $\mathcal{A}_{\mathsf{J+S+R}}$, $\mathcal{A}_{\mathsf{B+J+S+R}}$, while the program **comb1245** can only be proven insecure by $\mathcal{A}_{\mathsf{B+J+S+R}}$.

Semantics that are in no particular order in the lattice like $\mathcal{A}_{\mathsf{B+J+S+R}}$ and $\mathcal{A}_{\mathsf{B+J+S+SLS}}$ are also not able to detect the vulnerability in **comb1246** and **comb1245** respectively. This is expected since they each miss one speculative mechanism necessary to detect the vulnerability.

Because there is no combination of all our source speculative semantics $\mathcal{A}_{\mathsf{B}}$, $\mathcal{A}_{\mathsf{J}}$, $\mathcal{A}_{\mathsf{S}}$, $\mathcal{A}_{\mathsf{R}}$ and $\mathcal{A}_{\mathsf{SLS}}$, there is also no unique combination that can find the leaks in all programs in **Spectre-Comb**. However, using $\mathcal{A}_{\mathsf{B+J+S+R}}$ and $\mathcal{A}_{\mathsf{B+J+S+SLS}}$, Spectector is able to successfully detect leaks in all programs.

We also analyzed programs manually patched with `lfence` statements. The examples are not shown here for brevity and can be found with the other code snippets at [37]. As before, Spectector successfully proves the security of all patched programs. Even for leaks that arise from multiple speculation mechanisms, it is often sufficient to insert a single `lfence` to secure the entire program, e.g., an `lfence` after the **beqz** instruction in **comb15** is enough to make the program SNI with respect to $\mathcal{A}_{\mathsf{B+S+R}}$.

Program	B	J	S	R	SLS	B+J	B+S	B+R	B+SLS	J+S	J+R	J+SLS	S+R	S+SLS
comb12	●	●	●	●	●	○	●	●	●	●	●	●	●	●
comb14	●	●	●	●	●	●	○	●	●	●	●	●	●	●
comb15	●	●	●	●	●	●	●	○	●	●	●	●	●	●
comb16	●	●	●	●	●	●	●	●	○	●	●	●	●	●
comb24	●	●	●	●	●	●	●	●	●	○	●	●	●	●
comb25	●	●	●	●	●	●	●	●	●	●	○	●	●	●
comb26	●	●	●	●	●	●	●	●	●	●	●	○	●	●
comb45	●	●	●	●	●	●	●	●	●	●	●	●	○	●
comb46	●	●	●	●	●	●	●	●	●	●	●	●	●	○
comb124	●	●	●	●	●	●	●	●	●	●	●	●	●	●
comb125	●	●	●	●	●	●	●	●	●	●	●	●	●	●
comb126	●	●	●	●	●	●	●	●	●	●	●	●	●	●
comb145	●	●	●	●	●	●	●	●	●	●	●	●	●	●
comb146	●	●	●	●	●	●	●	●	●	●	●	●	●	●
comb245	●	●	●	●	●	●	●	●	●	●	●	●	●	●
comb246	●	●	●	●	●	●	●	●	●	●	●	●	●	●
comb1245	●	●	●	●	●	●	●	●	●	●	●	●	●	●
comb1246	●	●	●	●	●	●	●	●	●	●	●	●	●	●

Program	B+J+S	B+J+R	B+J+SLS	B+S+R	B+S+SLS	J+S+R	J+S+SLS	B+J+S+R	B+J+S+SLS
comb12	○	○	○	●	●	●	●	○	○
comb14	○	●	●	○	○	●	●	○	○
comb15	●	○	●	○	●	●	●	○	●
comb16	●	●	○	●	○	●	●	●	○
comb24	○	●	●	●	●	○	○	○	○
comb25	●	○	●	●	●	○	●	○	●
comb26	●	●	○	●	●	●	○	●	○
comb45	●	●	●	○	●	○	●	○	●
comb46	●	●	●	●	○	●	○	●	○
comb124	○	●	●	●	●	●	●	○	○
comb125	●	○	●	●	●	●	●	○	●
comb126	●	●	○	●	●	●	●	●	○
comb145	●	●	●	○	●	●	●	○	●
comb146	●	●	●	●	○	●	●	●	○
comb245	●	●	●	●	●	○	●	○	●
comb246	●	●	●	●	●	●	○	●	○
comb1245	●	●	●	●	●	●	●	○	●
comb1246	●	●	●	●	●	●	●	●	○

Table 1. Results of the analysis. For each program, ○ denotes that Spectector finds a violation of SNI whereas ● denotes that Spectector proves the program secure under the corresponding semantics. The different programs were devised to be vulnerable under a specific combination. The numbers correspond to that specific version 1 = B, 2 = J, 4 = S, 5 = R and 6 = SLS. For example, comb125 was devised to be vulnerable under B+J+R.

11 DISCUSSION

11.1 Scope of the Models

Lifting the results of the security analysis for our speculative semantics to real-world CPUs is only possible to the extent that these semantics capture the information flows in the target system. Thus, Spectector's result may incorrectly classify programs as secure (if our semantics do not capture information flows happening in real-world CPUs) or insecure (if our semantics admit speculations that are impossible on real systems).

We capture "leakage into the microarchitectural state" using the relatively powerful observer of the program execution that sees the location of memory accesses and the jump targets. This observer could be replaced by a weaker one, which accounts for more detailed models of a CPU's memory hierarchy, and Spectector could be adapted accordingly, e.g. by adopting the cache models from CacheAudit [29]. We believe, however, that highly detailed models are not actually desirable for several reasons: (a) they encourage brittle designs that break under small changes to the model, (b) they have to be adapted frequently, and (c) they are hard to understand and reason about for compiler developers and hardware engineers. The "constant-time" observer model adopted in this paper has proven to offer a good tradeoff between precision and robustness [4, 57].

11.2 Other Speculation Mechanisms

There are many speculation mechanisms beyond those modeled in the speculative semantics $\lightning_{\mathsf{B}}$, $\lightning_{\mathsf{S}}$, $\lightning_{\mathsf{J}}$, $\lightning_{\mathsf{R}}$ and $\lightning_{\mathsf{SLS}}$ from Section 6:

- CPUs speculate over **ret** instructions in different ways. For instance, there are many different ways of implementing return stack buffers (e.g., cyclic versus acyclic RSBs [53] or RSBs that fall back to indirect branch prediction [78]). This kind of speculation can be modeled by modifying the Rule R:AM-Spec rule in $\lightning_{\mathsf{R}}$.
- Many proposals for value prediction over different kinds of instructions exist [51, 56, 69]. While naive speculative semantics might have to explore *all* possible values as prediction, semantics that model specific prediction mechanisms might restrict the set of predicted values (thereby leading to a more tractable analysis).

We expect that most of these mechanisms can be modeled as speculative semantics satisfying our well-formedness conditions. Hence, they could work with our composition framework.

11.3 Limitations of Composition

Our composition framework has two main limitations:

(1) The metaparameter Z is expressed in terms of μAsm instructions, i.e., the smallest unit of computation in our framework. Since Z restricts how the composed semantics delegates execution to its sources, this limit the expressiveness of composed semantics. For instance, $\lightning_{\mathsf{S+R}}$ cannot speculate over the *implict* **store** writing the return address to the stack that happens as part of **call** instructions.

(2) Our framework does not support combinations where a single instruction performs speculation-relevant changes in both source semantics. For instance, consider a combination of $\lightning_{\mathsf{R}}$ with $\lightning_{\mathsf{SLS}}$. Here, both semantics start different speculative transactions on executing **ret** instructions. However, instantiating Z as $(\emptyset, \emptyset)$, which enables both speculations, violates the confluence well-formedness condition for the composed semantics, whereas setting $Z = (x, y)$ so that only one of x and y is **ret** would only capture one of the two speculation mechanisms.

The first limitation is inherent to the instruction granularity of Z, and we leave it to future work. The second can be lifted with a moderate extension, which we sketch here. When a single instruction (e.g., **ret**) triggers speculation in both

source semantics, the composed semantics—rather than delegating to one source and blocking the other via Z—creates two speculative instances at that instruction, one per source semantics, and pushes both onto the speculation stack. Since a speculative instance is popped when its window ends, execution resumes with the second instance, so both mechanisms are explored. The same construction applies to any pair of mechanisms acting on a single instruction, such as the load-address (LAP [47]) and load-value (LVP [46]) predictors of recent Apple processors. We leave a full formalization to future work.

11.4 Scalability

Spectector is a proof-of-concept aimed at detecting speculative leaks and their combinations rather than at analyzing large code bases, and three factors currently limit its scalability. First, the front end supports only a fraction of the x64 ISA (cf. Section 9.4); extending it to the full ISA is conceptually straightforward but engineering-intensive. Second, Spectector relies on manually specified policies partitioning configurations into public and secret parts; precise manual policies do not scale to large code bases, and automatic policy inference from the calling context is not yet supported. Third, Spectector is based on symbolic execution and therefore inherits its path explosion. That said, checking SNI itself adds little on top of symbolic execution. Although SNI is a relational property—and relational analyses are known to scale poorly—Spectector only ever compares executions that follow the *same* symbolic path: because the program counter is observable, speculative non-interference never requires relating two executions that disagree on their path conditions. The cost of the analysis is thus dominated by the underlying symbolic execution rather than by the relational SNI check itself.

11.5 Different Flavours of Non-Interference

Here, we discuss the relation of SNI with other well-known notions of security against side-channel leaks.

11.5.1 General Non-Interference. We start by introducing General Non-Interference (GNI), a notion of security capturing that secrets do not leak through side channels. This notion is adapted from the hardware-software contracts framework of Guarnieri et al. [36] into our setting.

Definition 5 (General Non-Interference (GNI))**.** *Program p satisfies GNI (denoted $p \vdash_x GNI$) for a semantics x iff for all σ, σ', if $\sigma \backsim_\phi \sigma'$ then $Beh^\omega_x(p, \sigma) = Beh^\omega_x(p, \sigma')$.*

In a nutshell, a program p satisfies GNI for a given semantics x if any pair of low-equivalent initial configurations σ and σ' results in the same observations, i.e., $Beh^\omega_x(p, \sigma) = Beh^\omega_x(p, \sigma')$.

Observe that GNI is an *absolute security property* [21], that is, it ensures that *all* observations are the same for any two executions starting from low-equivalent configurations. In contrast, SNI is a *relative security property* [21]. That is, it ensures that speculatively executed instructions (and, thus, speculative observations) do not leak more information than what is already leaked by the program's non-speculative execution. Naturally, satisfying GNI implies satisfying SNI for the same semantics x:

Corollary 1 (GNI implies SNI)**.** *If $p \vdash_x GNI$ then $p \vdash_x SNI$.*

11.5.2 Relation with constant-time variants. A common defense against several side-channel timing attacks is the constant-time programming discipline [4, 57]. This programming discipline requires that a program's control flow and memory access patterns are strictly independent of sensitive data.

Following [4], constant-time programming can be modeled as a non-interference property by requiring that low-equivalent executions result in the same architectural control-flow and in the same sequence of memory accesses. That is, we can instantiate constant-time security by requiring the equivalence of the non-speculative trace, as indicated next:

Definition 6 (Non-Speculative Constant-Time (CT)). *Program p satisfies CT (denoted $p \vdash SeqCT$) iff for all σ, σ', if $\sigma \backsim_\phi \sigma'$ then $Beh_{NS}(p, \sigma) = Beh_{NS}(p, \sigma')$.*

Observe that CT is an instantiation of GNI w.r.t. the non-speculative semantics, i.e., $p \vdash \text{SeqCT} \Leftrightarrow p \vdash_{NS} \text{GNI}$. Note also that we named Definition 6 as "Sequential Constant-Time" rather than the more standard "Constant-Time" to stress that CT traditionally applies only to the architectural, non-speculative semantics.

To account for the effects of speculatively executed instructions, several works [3, 12, 20, 25, 75] proposed security conditions that also restrict leaks through speculatively executed instructions. That is, all these works enforce variants of GNI, differing in the underlying speculative semantics or in what observations are part of the trace. For instance, in Cauligi et al. [20] the low-projection of the final configuration is also part of the trace (in addition to the usual observations associated with the constant-time attacker) to ensure the absence of explicit leaks.

Crucially, we can precisely connect CT, SNI, and GNI. That is, general non-interference w.r.t. a speculative semantics x is exactly the conjunction of constant-time and speculative non-interference w.r.t. x [36, Proposition 4]:

Proposition 1 (GNI Decomposition). *Given a policy ϕ and a program p: $p \vdash SeqCT \wedge p \vdash_x SNI \Leftrightarrow p \vdash_x GNI$*

Proposition 1 allows us to decompose the check for GNI into two independent checks. An analysis only needs to prove that the program is constant-time (SeqCT) and that it satisfies speculative non-interference (SNI).

11.5.3 Extending Spectector with support for GNI. We conclude by discussing how Spectector can be extended to check GNI. As stated above, SeqCT can be obtained by instantiating GNI with the non-speculative semantics, so this extension can be used also to check the classic constant-time property.

Recall that Spectector algorithm from Section 9.1 searches for pairs of execution traces that have equivalent non-speculative behavior but differing speculative observations.

To support GNI, we need to modify MemLeak and CtrlLeak as indicated in Figure 14. Specifically, we enforce that the entire symbolic trace $\overline{\tau}$ can only produce identical concrete observations for all low-equivalent inputs.

For this, we modify MemLeak to check the satisfiability of the following formula:

$$pthCnd(\overline{\tau})_{1\wedge 2} \wedge polEqv(P) \wedge \neg obsEqv(\overline{\tau}).$$

In particular, $pthCnd(\overline{\tau})_{1\wedge 2}$ ensures that the same symbolic path is followed, $polEqv(P)$ ensures that the initial states are low-equivalent, and $\neg obsEqv(\overline{\tau})$ asserts that at least one observation (either architectural or speculative) associated with memory accesses differ among the two runs. That is, this formula is satisfiable only if there are two concrete traces associated with the symbolic trace $\overline{\tau}$ where memory operation accesses a different address.

The CtrlLeak procedure is modified analogously to ensure that all control-flow observations (architectural or speculative) are identical between traces.

Thus, with minor modifications to the Spectector algorithm, we are able to check for GNI-style properties for all the semantics discussed in the paper. The full algorithm for checking GNI can be found in Appendix B.

Algorithm 2 MemLeak for GNI

1: **procedure** MemLeak($\overline{\tau}, P$)
2: $\quad\psi \leftarrow pthCnd(\overline{\tau})_{1\wedge 2} \wedge polEqv(P) \wedge \neg obsEqv(\overline{\tau})$
3: $\quad$**return** Satisfiable(ψ)

Algorithm 3 CtrlLeak for GNI

1: **procedure** CtrlLeak($\overline{\tau}, P$)
2: $\quad$**for** each prefix $v \cdot$ `symPc`(se) of $\overline{\tau}$ **do**
3: $\quad\quad\psi \leftarrow pthCnd(\overline{\tau}\restriction_{ns} \cdot v)_{1\wedge 2} \wedge polEqv(P) \wedge \neg sameSymbPc(se)$
4: $\quad\quad$**if** Satisfiable(ψ) **then**
5: $\quad\quad\quad$**return** $\top$
6: $\quad$**return** $\bot$

Fig. 14. The modified versions of MemLeak and CtrlLeak for GNI

12 RELATED WORK

Speculative Execution Attacks: After Spectre [49] has been disclosed to the public in 2018, researchers have identified many other speculative execution attacks [10, 15, 50, 53, 79, 81]. These attacks differ in the exploited speculation sources [43, 50, 53], the covert channels [2, 52, 67, 72, 73] used, or the target platforms [23]. We refer the reader to Canella et al. [18] for a survey of existing attacks.

Security Properties for Speculative Leaks: Researchers have proposed many program-level properties for security against speculative leaks, which can be classified in three main groups [21]:

(1) Non-interference definitions ensure the security of speculative *and* non-speculative instructions. For instance, speculative constant-time [3, 12, 20, 25, 75] extends the constant-time security condition to account also for transient instructions.

(2) Relative non-interference definitions [22, 35, 39, 70] ensure that transient instructions do not leak more information than what is leaked by non-transient instructions. The notion of Speculative Non-Interference (SNI) introduced in this paper falls into this category, as it restricts the information leaked by speculatively executed instructions relative to the program's non-speculative behavior.

(3) Definitions that formalise security as a safety property [64, 65], which may over-approximate definitions from the groups above.

Finally, we highlight the relationship between Speculative Constant-Time (SCT) and our properties SNI and GNI. As shown by Cauligi et al. [20] if the program is sequentially constant-time, SNI, and the resulting architectural states are low-equivalent, then the program is also SCT. In relation to GNI, SCT enforces the exact same trace equivalence but imposes the additional constraint that final architectural states must be low-equivalent. We can verify SCT on top of our Spectector-GNI check by explicitly exposing the final architectural state as a terminal observation within the execution trace.

Operational Semantics for Speculative Leaks: In the last few years, there has been a growing interest in developing formal models and principled program analyses for detecting leaks caused by speculatively executed instructions. We refer the reader to [21] for a comprehensive survey. In the following, we discuss the approaches that are particularly relevant to our paper.

Some approaches explicitly model microarchitectural components like multiple pipeline stages, caches, and branch predictors. For instance, KLEESpectre [76] and SpecuSym [39] consider a semantics that explicitly models the cache, enabling reasoning about the cache content. McIlroy et al. [55] go a step further and model a multi-stage pipeline with

explicit cache and branch predictor. Their semantics can only model speculation over branch instructions since it lacks store-forwarding or RSB.

Cauligi et al. [20]'s semantics model speculation over branch instructions, store-bypasses, and return instructions. Differently from our semantics, their 3-stage pipeline semantics explicitly models several microarchitectural components like a reorder buffer and an RSB. Their tool detects violations of speculative constant-time induced by speculation over branch instructions and store-bypasses.

Barthe et al. [12] extend the Jasmin [3] cryptographic verification framework to reason about speculative constant-time and supports speculation over store-bypasses and branch instructions. Shivakumar et al. [71] equip Jasmin with an information flow control type system that enforces SCT. However, only against Spectre-PHT attacks.

Binsec/Haunted [25] detect violations of speculative constant-time due to speculation over store-bypasses and branch instructions. For this, they explicitly model the store buffer, which ν_{S} abstracts away. Blade [75] uses a JIT-step semantics that translates high-level commands (from a WHILE-language) to low-level machine instructions while tracking control-flow predictions. This allows for source-level reasoning and their semantics captures speculation over branch instructions.

While several of these models support multiple speculation mechanisms, these mechanisms are *hard-coded* and no existing approach provides a composition framework or extensible ways of extending the main theoretical results to new mechanisms "for free". Moreover, while we could have used other semantics as a basis for our framework, this would have resulted in more difficult proofs (since semantics like the one in [20] are significantly more complex than ours).

Axiomatic Semantics for Speculative Leaks: A few approaches formalise the effects of speculatively executed instructions using axiomatic semantics inspired by work on weak memory models. For instance, Colvin and Winter [24] and Disselkoen et al. [27] capture the effects of branch speculation but both lack program analyses.

Ponce de León and Kinder [65] illustrate how one can model leaks resulting from speculation over branch instructions and store-bypasses using the CAT modeling language for memory consistency, and they present a bounded model checking analysis for detecting speculative leaks. Interestingly, they talk about composing several of their semantics [65, SIV.F], which should allow them to detect vulnerabilities like Listing 3 (which we detect under $\nu_{\mathsf{B+S}}$). However, they do not formally characterize compositions and, therefore, they cannot derive interesting results "for free" about the composed semantics (like we do in Theorem 2). Moreover, even though they state that composability is an advantage of axiomatic models, our framework (and tool implementation) shows that composability can be done with operational semantics as well.

Secure Compilation for Speculative Leaks: In addition to program analyses like the ones described above, researchers have proposed compiler passes to prevent and mitigate speculative leaks. For instance, Patrignani and Guarnieri [64] and Fabian et al. [32] study the security of compiler-level countermeasures implemented in major compilers using Secure Compilation theory, whereas Vassena et al. [75] propose a compiler pass for automatically patching speculative leaks using a type system to track speculative leaks. Shivakumar et al. [70] show that speculative execution can leak secrets even at declassification sites and propose selective speculative load hardening (selSLH), a compiler-level countermeasure they prove sound via a relative non-interference property. In a similar vein, Ultimate SLH [**?**] strengthens SLH to cover a broader class of speculative leaks.

Mosier et al. [58] propose a set of compiler passes that, together with hardware support (e.g. Intel CET-IBT, a shadow stack for return addresses) offers protection against Spectre-PHT, Spectre-BTB, Spectre-STL, Spectre-RSB, and predictive store forwarding [6]. We note that Mosier et al. [58, Section 3.3] showed a counterexample that our

semantics $\lightbulb_S$ does not catch. When devising our speculative semantics, we encountered a trade-off: keeping the related Spectector implementation tractable (albeit imprecise) or making the semantics precise (but not implementable due to state explosion of the tool). We decided to model the speculation in that way to keep the analysis tractable. Furthermore, our case study in Section 10.1 was used by other detection tools as well [25, 65] and our semantics $\lightbulb_S$ detects all vulnerabilities there. Thus, we think this is a reasonable trade-off between precision and performance.

Detecting Leaks through Testing: The Revizor testing tool [42, 60–62] and SpecFuzz [63] use fuzz testing to find vulnerabilities caused by speculation in CPUs. Similarly, SpecDoctor by Hur et al. [44] uses differential fuzzing to fuzz for transient vulnerabilities on the RTL level. Marinaro et al. [54] extend Scam-V [59] to optimize software mitigations for Spectre-PHT using a relational testing approach.

AMuLeT [33] adapts Revizor to the design phase. AMuLeT instruments microarchitectural simulators (e.g., gem5) to check secure speculation countermeasures against leakage contracts. LMTest [11] provides a framework for testing cryptographic implementations against parameterized leakage models (defined in a DSL called LmSpec). It generates test cases to detect secret-dependent leaks in the presence of proposed microarchitectural optimizations.

Unlike Spectector, these approaches cannot detect the absence of leaks.

13 CONCLUSION

This paper presented an approach to automatically detect speculative leaks in programs. We introduced speculative non-interference, the first semantic notion of security against speculative execution attacks, and we defined multiple speculative semantics to capture 5 classes of Spectre attacks. Next, we defined a general framework to reason about the composition of different speculative semantics and instantiated the framework with our speculative semantics. Our framework yields safety of the composed semantics (almost) for free, given the safety of its parts. We utilize our theoretical development and implement a program analysis tool called Spectector that automatically detects speculative leaks or proves their absence, for our speculative semantics and their combinations.

ACKNOWLEDGMENTS

This work is supported by the Spanish Ministry of Science, Innovation, and University under the Ramón y Cajal grant RYC2021-032614-I; the Spanish Ministry of Science, Innovation, and University under the project PID2022-142290OB-I00 ESPADA; the Spanish Ministry of Science, Innovation, and University under the Europa Excelencia project EUR2025-164828 SINTRAZAS; the Spanish Ministry of Science, Innovation, and University under the project CEX2024-001471-M; the European Union under the Horizon Europe projects 101230068 PRIMULA and 101020415 SafeSecS.

REFERENCES

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-Flow Integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security* (Alexandria, VA, USA) *(CCS '05)*. Association for Computing Machinery, New York, NY, USA, 340–353. https://doi.org/10.1145/1102120.1102165

[2] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. 2019. Port Contention for Fun and Profit. In *2019 IEEE Symposium on Security and Privacy (SP)*. 870–887. https://doi.org/10.1109/SP.2019.00066

[3] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. 2017. Jasmin: High-Assurance and High-Speed Cryptography. In *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. ACM.

[4] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016. Verifying constant-time implementations. In *Proceedings of the 25th USENIX Conference on Security Symposium (SEC'16)*. USENIX, USA.

[5] AMD. 2020. Whitepaper Straight-line Speculation. https://www.amd.com/system/files/documents/technical-guidance-for-mitigating-branch-type-confusion.pdf.

[6] AMD. 2021. Security analysis of AMD predictive store forwarding. https://www.amd.com/system/files/documents/security-analysis-predictive-store-forwarding.pdf. Accessed: 2024-03-11.

[7] arm. 2020. Straight-line Speculation. https://developer.arm.com/documentation/102825/0100/?lang=en. Accessed: 2022-09-05.

[8] ARM. 2020. Whitepaper Straight-line Speculation. https://developer.arm.com/documentation/102825/0100/.

[9] ARM. 2021. Arm Armv9-A A64 Instruction Set Architecture. https://developer.arm.com/documentation/100076/0100/A64-Instruction-Set-Reference/A64-General-Instructions/BTI?

[10] Enrico Barberis, Pietro Frigo, Marius Muench, Herbert Bos, and Cristiano Giuffrida. 2022. Branch history injection: On the effectiveness of hardware mitigations against cross-privilege Spectre-v2 attacks. In *Proceedings of the 31st USENIX Security Symposium (USENIX Security '22)*. USENIX Association.

[11] Gilles Barthe, Marcel Böhme, Sunjay Cauligi, Chitchanok Chuengsatiansup, Daniel Genkin, Marco Guarnieri, David Mateos Romero, Peter Schwabe, David Wu, and Yuval Yarom. 2024. Testing Side-channel Security of Cryptographic Implementations against Future Microarchitectures. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security (CCS '24)*. ACM. https://doi.org/10.1145/3658644.3670319

[12] Gilles Barthe, Sunjay Cauligi, Benjamin Grégoire, Adrien Koutsos, Kevin Liao, Tiago Oliveira, Swarn Priya, Tamara Rezk, and Peter Schwabe. 2021. High-Assurance Cryptography in the Spectre Era. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (S&P '21)*. IEEE.

[13] Gilles Barthe, Pedro R D'argenio, and Tamara Rezk. 2004. Secure information flow by self-composition. In *Proceedings of the 17th IEEE Computer Security Foundations Workshop (CSF '04)*. IEEE.

[14] Binsec/Haunted Benchmark. 2021. Result of case_13. https://github.com/binsec/haunted_bench/issues/2.

[15] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. 2019. SMoTherSpectre: Exploiting Speculative Execution through Port Contention. In *Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security (CCS '19)*. ACM.

[16] Aaron R. Bradley and Zohar Manna. 2007. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer.

[17] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. 2017. Control-Flow Integrity: Precision, Security, and Performance. *ACM Comput. Surv.* 50, 1 (apr 2017). https://doi.org/10.1145/3054924

[18] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security '19)*. USENIX Association.

[19] C. Carruth. 2018. Speculative load hardening. https://llvm.org/docs/SpeculativeLoadHardening.html

[20] Sunjay Cauligi, Craig Disselkoen, Klaus v. Gleissenthall, Dean Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. 2020. Constant-Time Foundations for the New Spectre Era. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '20)*. ACM.

[21] Sunjay Cauligi, Craig Disselkoen, Daniel Moghimi, Gilles Barthe, and Deian Stefan. 2022. SoK: Practical Foundations for Software Spectre Defenses. In *Proceedings of the 43rd IEEE Symposium on Security and Privacy (S&P '22)*. IEEE.

[22] Kevin Cheang, Cameron Rasmussen, Sanjit Seshia, and Pramod Subramanyan. 2019. A Formal Approach to Secure Speculation. In *Proceedings of the 32nd IEEE Computer Security Foundations Symposium (CSF '19)*. IEEE.

[23] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. 2019. Stealing Intel Secrets from SGX Enclaves via Speculative Execution. In *Proceedings of the 4th IEEE European Symposium on Security and Privacy (EuroS&P '19)*. IEEE.

[24] Robert J. Colvin and Kirsten Winter. 2019. An Abstract Semantics of Speculative Execution for Reasoning About Security Vulnerabilities. In *Proceedings of the 19th Refinement Workshop (Refine '19)*. Springer.

[25] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. 2021. Hunting the Haunter — Efficient relational symbolic execution for Spectre with Haunted RelSE. In *Proceedings of the 28th Annual Network and Distributed System Security Symposium (NDSS '21)*. The Internet Society.

[26] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

[27] Craig Disselkoen, Radha Jagadeesan, Alan Jeffrey, and James Riely. 2019. The Code That Never Ran: Modeling Attacks on Speculative Evaluation. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P '19)*. IEEE.

[28] Jack Doweck, Wen-Fu Kao, Allen Kuan-yu Lu, Julius Mandelblat, Anirudha Rahatekar, Lihu Rappoport, Efraim Rotem, Ahmad Yasin, and Adi Yoaz. 2017. Inside 6th-Generation Intel Core: New Microarchitecture Code-Named Skylake. *IEEE Micro* (2017). https://doi.org/10.1109/MM.2017.38

[29] Goran Doychev, Boris Köpf, Laurent Mauborgne, and Jan Reineke. 2015. CacheAudit: A Tool for the Static Analysis of Cache Side Channels. *ACM Trans. Inf. Syst. Secur.* 18 (2015).

[30] Xaver Fabian, Marco Guarnieri, Boris Köpf, José F. Morales, Marco Patrignani, Jan Reineke, and Andrés Sánchez. 2025. [Technical Report]. Supplementary Material. Submitted to ACM TOPLAS with this manuscript.

[31] Xaver Fabian, Marco Patrignani, and Marco Guarnieri. 2022. Automatic Detection of Speculative Execution Combinations. In *Proceedings of the 29th ACM Conference on Computer and Communications Security (CCS 2022)*. ACM.

[32] Xaver Fabian, Marco Patrignani, Marco Guarnieri, and Michael Backes. 2025. Do You Even Lift? Strengthening Compiler Security Guarantees against Spectre Attacks *(POPL'25)*. ACM, New York, NY, USA. https://doi.org/10.1145/3704867

[33] Bo Fu, Leo Tenenbaum, David Adler, Assaf Klein, Arpit Gogia, Alaa R. Alameldeen, Marco Guarnieri, Mark Silberstein, Oleksii Oleksenko, and Gururaj Saileshwar. 2025. AMuLeT: Automated Design-Time Testing of Secure Speculation Countermeasures. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '25)*. ACM. https://doi.org/10.1145/3676641.3716247

[34] Google. 2019. SafeSide. https://github.com/google/safeside

[35] Roberto Guanciale, Musard Balliu, and Mads Dam. 2020. InSpectre: Breaking and Fixing Microarchitectural Vulnerabilities by Formal Analysis. In *Proceedings of the 27th ACM SIGSAC Conference on Computer and Communications Security (CCS '20)*. ACM.

[36] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. 2021. Hardware-Software Contracts for Secure Speculation. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (S&P '21)*. IEEE.

[37] Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. 2020. Spectector – Automatic detection of speculative information flows. https://spectector.github.io/

[38] Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. 2020. Spectector: Principled Detection of Speculative Information Flows. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P '20)*.

[39] Shengjian Guo, Yueqi Chen, Peng Li, Yueqiang Cheng, Huibo Wang, Meng Wu, and Zhiqiang Zuo. 2020. SpecuSym: Speculative Symbolic Execution for Cache Timing Leak Detection. In *Proceedings of the 42nd ACM/IEEE International Conference on Software Engineering (ICSE '20)*. ACM.

[40] John L. Hennessy and David A. Patterson. 2011. *Computer Architecture, Fifth Edition: A Quantitative Approach* (5th ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[41] Manuel V. Hermenegildo, Francisco Bueno, Manuel Carro, Pedro López-García, Edison Mera, José F. Morales, and German Puebla. 2012. An overview of Ciao and its design philosophy. *Theory and Practice of Logic Programming* 12, 1-2 (2012), 219–252.

[42] Jana Hofmann, Emanuele Vannacci, Cedric Fournet, Boris Kopf, and Oleksii Oleksenko. 2023. Speculation at Fault: Modeling and Testing Microarchitectural Leakage of CPU Exceptions. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association. https://www.usenix.org/conference/usenixsecurity23/presentation/hofmann

[43] J. Horn. 2018. Speculative execution, variant 4: Speculative store bypass. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528. Accessed: 2021-04-11.

[44] Jaewon Hur, Suhwan Song, Sunwoo Kim, and Byoungyoung Lee. 2022. SpecDoctor: Differential Fuzz Testing to Find Transient Execution Vulnerabilities. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS '22)*. Association for Computing Machinery, New York, NY, USA, 1473–1487. https://doi.org/10.1145/3548606.3560578

[45] Intel. 2018. Retpoline: A Branch Target Injection Mitigation. https://www.intel.com/content/dam/develop/external/us/en/documents/retpoline-a-branch-target-injection-mitigation.pdf

[46] Jason Kim, Jalen Chuang, Daniel Genkin, and Yuval Yarom. 2025. FLOP: Breaking the Apple M3 CPU via False Load Output Predictions. In *USENIX Security*.

[47] Jason Kim, Daniel Genkin, and Yuval Yarom. 2025. SLAP: Data Speculation Attacks via Load Address Prediction on Apple Silicon. In *S&P*.

[48] P. Kocher. 2018. Spectre Mitigations in Microsoft's C/C++ Compiler. https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html. Accessed: 2021-04-11.

[49] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P '19)*.

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

[51] Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. 1996. Value locality and load value prediction. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '96)*. ACM.

[52] Moritz Lipp, Andreas Kogler, David Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. 2021. PLATYPUS: Software-based Power Side-Channel Attacks on x86. In *2021 IEEE Symposium on Security and Privacy (SP)*. 355–371. https://doi.org/10.1109/SP40001.2021.00063

[53] Giorgi Maisuradze and Christian Rossow. 2018. Ret2spec: Speculative Execution Using Return Stack Buffers. In *Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. ACM.

[54] Tiziano Marinaro, Pablo Buiras, Andreas Lindner, Roberto Guanciale, and Hamed Nemati. 2024. Beyond Over-Protection: A Targeted Approach to Spectre Mitigation and Performance Optimization. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security (ASIA CCS '24)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3634737.3637651

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

[56] Sparsh Mittal. 2017. A survey of value prediction techniques for leveraging value locality. *Concurrency and computation: practice and experience* (2017).

[57] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. 2005. The program counter security model: automatic detection and removal of control-flow side channel attacks. In *Proceedings of the 8th International Conference on Information Security and Cryptology* (Seoul, Korea) *(ICISC'05)*. Springer.

[58] N. Mosier, H. Nemati, J. C. Mitchell, and C. Trippel. 2024. Serberus: Protecting Cryptographic Code from Spectres at Compile-Time. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 48–48. https://doi.org/10.1109/SP54263.2024.00048

[59] Hamed Nemati, Pablo Buiras, Andreas Lindner, Roberto Guanciale, and Swen Jacobs. 2020. Validation of Abstract Side-Channel Models for Computer Architectures. In *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part I*. Springer-Verlag, Berlin, Heidelberg. https://doi.org/10.1007/978-3-030-53288-8_12

[60] Oleksii Oleksenko, Christof Fetzer, Boris Köpf, and Mark Silberstein. 2022. Revizor: Testing Black-Box CPUs against Speculation Contracts. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22)*. ACM.

[61] Oleksii Oleksenko, Marco Guarnieri, Boris Köpf, and Mark Silberstein. 2023. Hide and Seek with Spectres: Efficient discovery of speculative information leaks with random testing. In *Proceedings of the 44th IEEE Symposium on Security and Privacy (S&P 2023)*. IEEE.

[62] Oleksii Oleksenko, Flavien Solt, Cédric Fournet, Jana Hofmann, Boris Köpf, and Stavros Volos. 2026. Enter, Exit, Page Fault, Leak: Testing Isolation Boundaries for Microarchitectural Leaks. In *Proceedings of the 47th IEEE Symposium on Security and Privacy (S&P)*. https://arxiv.org/abs/2507.06039 To appear.

[63] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. 2020. SpecFuzz: Bringing Spectre-type vulnerabilities to the surface. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 1481–1498. https://www.usenix.org/conference/usenixsecurity20/presentation/oleksenko

[64] Marco Patrignani and Marco Guarnieri. 2021. Exorcising Spectres with Secure Compilers. In *Proceedings of the 28th ACM Conference on Computer and Communications Security (CCS '21)*. ACM.

[65] Hernán Ponce de León and Johannes Kinder. 2022. Cats vs. Spectre: An Axiomatic Approach to Modeling Speculative Execution Attacks. In *Proceedings of the 43rd IEEE Symposium on Security and Privacy (S&P '22)*. IEEE.

[66] A. Sabelfeld and D. Sands. 2005. Dimensions and principles of declassification. In *18th IEEE Computer Security Foundations Workshop (CSFW'05)*. 255–269. https://doi.org/10.1109/CSFW.2005.15

[67] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. 2019. NetSpectre: Read Arbitrary Memory over Network. In *Proceedings of the 24th European Symposium on Research in Computer Security (ESORICS '19)*. Springer.

[68] Vedvyas Shanbhogue, Deepak Gupta, and Ravi Sahita. 2019. Security Analysis of Processor Instruction Set Architecture for Enforcing Control-Flow Integrity. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy (HASP '19)*. ACM.

[69] Rami Sheikh, Harold W. Cain, and Raguram Damodaran. 2017. Load value prediction via path-based address prediction: Avoiding mispredictions due to conflicting stores. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '17)*. ACM.

[70] Basavesh Ammanaghatta Shivakumar, Jack Barnes, Gilles Barthe, Sunjay Cauligi, Chitchanok Chuengsatiansup, Daniel Genkin, Sioli O'Connell, Peter Schwabe, Rui Qi Sim, and Yuval Yarom. 2023. Spectre Declassified: Reading from the Right Place at the Wrong Time. In *Proceedings of the 44th IEEE Symposium on Security and Privacy (S&P '23)*. IEEE.

[71] Basavesh Ammanaghatta Shivakumar, Gilles Barthe, Benjamin Gregoire, Vincent Laporte, Tiago Oliveira, Swarn Priya, Peter Schwabe, and Lucas Tabary-Maujean. 2023. Typing High-Speed Cryptography against Spectre v1. In *2023 IEEE Symposium on Security and Privacy (SP)*.

[72] Julian Stecklina and Thomas Prescher. 2018. LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels. *CoRR* (2018). arXiv:1806.07480

[73] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. 2018. MeltdownPrime and SpectrePrime: Automatically-Synthesized Attacks Exploiting Invalidation-Based Coherence Protocols. *CoRR* (2018). arXiv:1802.03802

[74] Daniël Trujillo, Johannes Wikner, and Kaveh Razavi. 2023. Inception: Exposing New Attack Surfaces with Training in Transient Execution. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA. https://www.usenix.org/conference/usenixsecurity23/presentation/trujillo

[75] Marco Vassena, Craig Disselkoen, Klaus von Gleissenthall, Sunjay Cauligi, Rami Gökhan Kıcı, Ranjit Jhala, Dean Tullsen, and Deian Stefan. 2021. Automatically Eliminating Speculative Leaks from Cryptographic Code with Blade. *Proceedings of the ACM on Programming Languages* 5, POPL (2021).

[76] Guanhua Wang, Sudipta Chattopadhyay, Arnab Kumar Biswas, Tulika Mitra, and Abhik Roychoudhury. 2020. KLEESpectre: Detecting Information Leakage through Speculative Cache Attacks via Symbolic Execution. *ACM Transactions on Software Engineering and Methodology* 29, 3 (2020).

[77] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. 2021. oo7: Low-Overhead Defense Against Spectre Attacks via Program Analysis. *IEEE Transactions on Software Engineering* 47, 11 (2021).

[78] Johannes Wikner and Kaveh Razavi. 2022. RETBLEED: Arbitrary Speculative Code Execution with Return Instructions. In *Proceedings of the 31st USENIX Security Symposium (USENIX Security '22)*. USENIX Association.

[79] Johannes Wikner, Daniël Trujillo, and Kaveh Razavi. 2023. Phantom: Exploiting Decoder-detectable Mispredictions. In *MICRO*. Paper=https://comsec.ethz.ch/wp-content/files/phantom_micro23.pdfURL=https://comsec.ethz.ch/research/microarch/inception

[80] Meng Wu and Chao Wang. 2019. Abstract interpretation under speculative execution. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI' 19)*. ACM, New York, NY, USA. https://doi.org/10.1145/3314221.3314647

[81] Tao Zhang, Kenneth Koltermann, and Dmitry Evtyushkin. 2020. Exploring Branch Predictors for Constructing Transient Execution Trojans. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. ACM.

[82] Zhiyuan Zhang, Gilles Barthe, Chitchanok Chuengsatiansup, Peter Schwabe, and Yuval Yarom. 2023. Ultimate SLH: Taking Speculative Load Hardening to the Next Level. In *Proceedings of the 32nd USENIX Security Symposium*. USENIX.

A CODE FROM CASE STUDIES

A.1 Example #8

In Example #8, the bounds check of Listing 1 is implemented using a conditional operator:

```
1  temp &= B[A[y<size?(y+1):0]*512];
```

When compiling the example without countermeasures or optimizations, the conditional operator is translated to a branch instruction (cf. line 4), which is a source of speculation. Hence, the resulting program contains a speculative leak, which Spectector correctly detects.

```
1  mov    size, %rcx
2  mov    y, %rax
3  cmp    %rcx, %rax
4  jae    .L1
5  add    $1, %rax
6  jmp    .L2
7
8  xor    %rax, %rax
9  jmp    .L2
10
11 mov    A(%rax), %rax
12 shl    $9, %rax
13 mov    B(%rax), %rax
14 mov    temp, %rcx
15 and    %rax, %rcx
16 mov    %rcx, temp
```

In the Unp -O2 mode, the conditional operator is translated as a conditional move (cf. line 6), for which Spectector can prove security.

```
1  mov    size, %rax
2  mov    y, %rdx
3  xor    %rcx, %rcx
4  cmp    %rdx, %rax
5  lea    1(%rdx), %rax
6  cmova  %rax, %rcx
7  mov    A(%rcx), %rax
8  shl    $9, %rax
9  mov    B(%rax), %rax
10 and    %rax, temp
```

A.2 Example #15 in SLH mode

Here, the adversary provides the input via the pointer `*y`:

```
1  if (*y < size)
2     temp &= B[A[*y] * 512];
```

In the `-O0` SLH mode, CLANG hardens the address used for performing the memory access `A[*y]` in lines 8–12, but not the resulting value, which is stored in the register `%cx`. However, the value stored in `%cx` is used to perform a second memory access at line 14. An adversary can exploit the second memory access to speculatively leak the content of `A[0xFF...FF]`. In our experiments, SPECTECTOR correctly detected such leak.

```
1  mov     $0, %rax
2  mov     y, %rdx
3  mov     (%rdx), %rsi
4  mov     size, %rdx
5  cmp     %rdx, %rsi
6  jae      END
7  cmovae  $-1, %rax
8  mov     y, %rcx
9  mov     (%rcx), %rcx
10 mov     %rax, %rdx
11 or      %rcx, %rdx
12 mov     A(%rdx), %rcx
13 shl     $9, %rcx
14 mov     B(%rcx), %rcx
15 mov     temp, %rdx
16 and     %rcx, %rdx
17 mov     %rdx, temp
```

In contrast, when Example #15 is compiled with the `-O2` flag, CLANG correctly hardens `A[*y]`'s result (cf. line 10). This prevents information from flowing into the microarchitectural state during speculative execution. Indeed, SPECTECTOR proves that the program satisfies speculative non-interference.

```
1  mov     $0, %rax
2  mov     y, %rdx
3  mov     (%rdx), %rdx
4  mov     size, %rsi
5  cmp     %rsi, %rdx
6  jae     END
7  cmovae  $-1, %rax
8  mov     A(%rdx), %rcx
9  shl     $9, %rcx
10 or      %rax, %rcx
```

```
11  mov     B(%rcx), %rcx
12  or      %rax, %rcx
13  and     %rcx, temp
```

B VARIANTS OF SPECTECTOR

In Section 11.5 we discussed the variation of the SPECTECTOR algorithm to check for GNI. Here, we will present the full algorithm as well as the algorithm for checking sequential constant-time:

Algorithm 4 SPECTECTOR SeqCT

Input: A program p, a security policy P, a speculative window $\omega \in \mathbb{N}$.
Output: SECURE if p satisfies SeqCT with respect to the policy P; INSECURE otherwise

1: **procedure** SPECTECTOR(p, P, ω)
2: **for** each symbolic run $\overline{\tau} \in Beh_x^S(p)$ **do**
3: **if** MEMLEAK($\overline{\tau}, P$) $\vee$ CTRLLEAK($\overline{\tau}, P$) **then**
4: **return** INSECURE
5: **return** SECURE

6: **procedure** MEMLEAK($\overline{\tau}, P$)
7: $\psi \leftarrow pthCnd(\overline{\tau})_{1\wedge 2} \wedge polEqv(P) \wedge \neg obsEqv(\overline{\tau}\restriction_{ns})$
8: **return** SATISFIABLE(ψ)

9: **procedure** CTRLLEAK($\overline{\tau}, P$)
10: **for** each prefix $v \cdot$ `symPc`(se) of $\overline{\tau}\restriction_{ns}$ **do**
11: $\psi \leftarrow pthCnd(\overline{\tau}\restriction_{ns} \cdot v)_{1\wedge 2} \wedge polEqv(P) \wedge \neg sameSymbPc(se)$
12: **if** SATISFIABLE(ψ) **then**
13: **return** $\top$
14: **return** $\bot$

Algorithm 5 SPECTECTOR GNI

Input: A program p, a security policy P, a speculative window $\omega \in \mathbb{N}$.
Output: SECURE if p satisfies general non-interference with respect to the policy P and speculative semantics $\lightning_x$; INSECURE otherwise

procedure SPECTECTOR(p, P, ω)
  for each symbolic run $\overline{\tau} \in Beh_x^{\mathcal{S}}(p)$ **do**
    if MEMLEAK($\overline{\tau}, P$) $\vee$ CTRLLEAK($\overline{\tau}, P$) **then**
      return INSECURE
  return SECURE

procedure MEMLEAK($\overline{\tau}, P$)
  $\psi \leftarrow pthCnd(\overline{\tau})_{1\wedge 2} \wedge polEqv(P) \wedge \neg obsEqv(\overline{\tau})$
  return SATISFIABLE(ψ)

procedure CTRLLEAK($\overline{\tau}, P$)
  for each prefix $v \cdot \texttt{symPc}(se)$ of $\overline{\tau}$ **do**
    $\psi \leftarrow pthCnd(\overline{\tau}\restriction_{ns} \cdot v)_{1\wedge 2} \wedge polEqv(P) \wedge \neg sameSymbPc(se)$
    if SATISFIABLE(ψ) **then**
      return $\top$
  return $\bot$