Automatic Detection of Speculative Execution Combinations

Xaver Fabian
Cispa Helmholtz Center for Information Security
Saarbrücken, Germany
xaver.fabian@cispa.de

Marco Guarnieri
IMDEA Software Institute
Madrid, Spain
marco.guarnieri@imdea.org

Marco Patrignani
University of Trento
Trento, Italy
marco.patrignani@unitn.it

ABSTRACT
Modern processors employ different speculation mechanisms to speculate over different kinds of instructions. Attackers can exploit these mechanisms simultaneously in order to trigger leaks of speculatively-accessed data. Thus, sound reasoning about such speculative leaks requires accounting for all potential speculation mechanisms. Unfortunately, existing formal models only support reasoning about fixed, hard-coded speculation mechanisms, with no simple support to extend said reasoning to new mechanisms.

In this paper we develop a framework for reasoning about composed speculative semantics that capture speculation due to different mechanisms and implement it as part of the SPECTECTOR verification tool. We implement novel semantics for speculating over store and return instructions and combine them with the semantics for speculating over branch instructions. Our framework yields speculative semantics for speculating over any combination of these instructions that are secure by construction, i.e., we obtain these security guarantees for free. The implementation of our novel semantics in SPECTECTOR let us verify programs that are vulnerable to Spectre v1, Spectre v4, and Spectre v5 vulnerabilities as well as new snippets that are only vulnerable to their compositions.

CCS CONCEPTS
• Security and privacy → Formal security models; Systems security.

KEYWORDS
Spectre; Speculative Execution; Speculative information flows; Speculative non-interference; Combinations of speculative semantics

ACM Reference Format:
Xaver Fabian, Marco Guarnieri, and Marco Patrignani. 2022. Automatic Detection of Speculative Execution Combinations. In Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS ’22), November 7–11, 2022, Los Angeles, CA, USA. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3548606.3560555

1 INTRODUCTION
Speculative execution avoids pipeline stalls by predicting intermediate results and by speculatively executing instructions based on such predictions. When a prediction turns out to be incorrect, the processor squashes the speculative instructions, thereby rolling back their effect on the architectural state. Speculative instructions, however, leave footprints in microarchitectural components (like caches) that persist even after speculative execution terminates. As shown by Spectre [24], attackers can exploit these side effects to leak information about speculatively accessed data.

Modern general-purpose processors have different speculation mechanisms (branch predictors, memory disambiguators, etc.) that are used to speculate over different kinds of instructions: conditional branching [24], indirect jumps [24], store and load operations [23], and return instructions [25]. While well-known attacks target only individual speculation mechanism (e.g., Spectre-PHT [24] targets branch predictors), some speculative leaks only arise due to the interaction of multiple mechanisms.

Listing 1: 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];
```

For example, the code in Listing 1 can speculatively leak the value of $\texttt{secret}$ 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. Hence, leaks like the one in Listing 1 are missed by sound analyses for speculative leaks that consider speculation over only one of these speculation mechanisms.

Sound reasoning about speculative leaks requires accounting for all potential speculation mechanisms. However, existing formal models (also called speculative semantics) support multiple speculation mechanisms poorly. Some of them support only fixed speculation mechanisms: branch prediction [20, 21, 38–40] and (in addition) memory disambiguation prediction [9, 14, 32]. Furthermore, the different speculation mechanisms are hard-coded into the formal semantics [9, 14, 21]. Extending these semantics with new speculation mechanisms (e.g., speculation over return addresses or value prediction) requires changes to the formal model and to any security proof relying on it. This is not a scalable approach for developing comprehensive formal models and analyses for speculative leaks.

In this paper we develop a framework for composing speculative semantics that capture speculation due to different mechanisms and implement it as part of the SPECTECTOR verification tool. The combination yields a single operational semantics that can be used
to reason about leaks involving all speculation mechanisms of the components (as in Listing 1). Our framework lets us define the speculative semantics of each mechanism independently, which leads to simpler formalisation. Additionally, the security of the composed semantics is derived automatically from the security of its sub-parts, maximising proof reuse. Finally, the composed semantics can be easily implemented in Spectector, which can be used to verify the absence of leaks like those in Listing 1.

Concretely, this paper makes the following contributions:

- It introduces $\mathcal{Q}_1$ and $\mathcal{Q}_R$, two novel semantics for speculation over store and return instructions (Section 3).
- It defines the framework for composing different speculative semantics and formalises its key properties: if the individual semantics fulfil some (expected) security conditions (which we prove for all the semantics we combine), then the composed semantics is also secure (Section 4).
- It instantiates the framework with $\mathcal{Q}_2$, $\mathcal{Q}_R$ and $\mathcal{Q}_b$, the semantics for speculation over branch instructions from [21], creating all the possible compositions ($\mathcal{Q}_{b+s} \triangleq \mathcal{Q}_b \circ \mathcal{Q}_R$, $\mathcal{Q}_{s+b} \triangleq \mathcal{Q}_s \circ \mathcal{Q}_R$, and $\mathcal{Q}_{b+s+r}$) and proving their security (Section 5). All these semantics are mechanised in Coq, and we write $\mathcal{Q}$ to indicate when traces are calculated mechanically.
- It extends the Spectector verification tool with all these semantics and validates this extension on both existing benchmarks (for speculation on store and return instructions) as well as on new snippets (for combined speculation) that we define (Section 6).

The rest of the paper first presents background notions, such as the security notion we rely on, and the formal language we extend with the novel speculative semantics (Section 2) and then related work (Section 8) and conclusions (Section 9).

Additional material: Full details of the semantics and proofs can be found in the technical report available at [18]. The extended version of Spectector is available at [16], whereas the mechanisation of our speculative semantics in Coq are available at [17].

2 BACKGROUND: $\mu$ASM, SPECULATIVE SEMANTICS AND SECURITY DEFINITION

This section first describes the attacker model and the security definition we consider (Section 2.1). Then, it presents the syntax (Section 2.2) and the semantics (Section 2.3) of $\mu$ASM, a simple assembly-style language, followed by $\mathcal{Q}_2$, the semantics for speculation over branch instructions (Section 2.4). Most of the notions that we overview next are taken from Guarnieri et al. [21].

2.1 Attacker Model and Security Definition

We adopt a commonly-used attacker model [3, 9, 14, 19–21, 31, 38]: a passive attacker observing the execution of a program through events $\tau$. These events, which we call observations, model timing leaks through cache and control flow while abstracting away low-level microarchitectural details.

$$\text{Obs} ::= \text{load } n \mid \text{store } n \mid \text{pc } n \mid \text{call } f \mid \text{ret } n \mid \text{start}_x n \mid \text{rlb}_y n \mid \tau ::= \epsilon \mid \text{Obs} \mid \text{start}_x n \mid \text{rlb}_y n$$

The $\text{store } n$ and $\text{load } n$ events denote read and write accesses to memory location $n$, so they model cache leakage. In contrast, $\text{pc } n$, $\text{call } f$, and $\text{ret } n$ events record the control-flow of the program.

The $\text{start}_x n$ and $\text{rlb}_y n$ observations denote the start and the finish of a speculative transaction [21] (with identifier $n$) produced by the speculative semantics $x$ (we use $x$ and $y$ to range over the speculative semantics we define later).

An observation $\tau$ is either an event $\text{Obs}$ or the empty observation $\epsilon$. Traces $\mathcal{T}$ 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 $\bar{e} \cdot e$.

The non-speculative projection $\mathcal{T}_{ns}$ [21] of a trace $\mathcal{T}$ deletes all speculative observations by removing all sub-traces enclosed between $\text{start}_x n$ and $\text{rlb}_y n$. The remaining trace, then, captures all non-speculative observations.

Speculative Non-Interference: With this trace model we can define the security property we use in this paper: Speculative Non-Interference (SNI) [21]. Intuitively, SNI requires that programs do not leak more information under the speculative semantics than under the non-speculative semantics.

SNI is parametric in a policy $\phi$, which describes public/low information for the program, and in the used speculative semantics $x$, which models how the program executes. Following Guarnieri et al. [21], a policy $\phi$ consists of a list of public registers and public memory locations. Two configurations $\sigma^1, \sigma^2$ are called low-equivalent for a policy $\phi$, written $\sigma^1 \sim_\phi \sigma^2$, if they agree on all register and memory locations in $\phi$. The speculative semantics $x$ defines how (speculative) traces describing the program behaviour are generated. We indicate that program $p$ generates trace $\mathcal{T}$ from state $s$ with semantic $x$ as $\text{Beh}^x(p, s, \mathcal{T}) = \mathcal{T}$. We formalise multiple speculative semantics in later sections, each one instantiating $\text{Beh}^x(p, s)$.

A program $p$ satisfies SNI (Definition 1) for a speculative semantics $x$ if any pair of low-equivalent initial configurations $\sigma^1$ and $\sigma^2$ that generate the same observations without speculative events also generate the same observations with speculative events too.

Definition 1 (SNI). Program $p$ satisfies SNI (denoted $p \models_x \text{SNI}$) if for all $\sigma^1, \sigma^2$, if $\sigma^1 \sim_\phi \sigma^2$ and $\text{Beh}^x(p, \sigma^1)_{ns} = \text{Beh}^x(p, \sigma^2)_{ns}$ then $\text{Beh}^x(p, \sigma^1) = \text{Beh}^x(p, \sigma^2)$.

2.2 $\mu$ASM

(Executable) $p ::= n : i \mid p_1 ; p_2$ (Functions) $\mathcal{F} ::= \emptyset \mid \mathcal{F} : f \mapsto n$

(Registers) $x \in \text{Regs}$ (Values) $n \in \text{Vals} = \mathbb{N} \cup \{\bot\}$

(Expressions) $e ::= n \mid x \mid \text{load } x \mid \text{store } x \mid \text{jump } e \mid \text{beq } x \mid l \mid \text{call } f \mid \text{ret }$

$\mu$ASM is an assembly-like language whose syntax is presented above. Programs $p$ in $\mu$ASM are sequences of mappings from natural numbers $n$ (i.e., the instruction address) to instructions $i$ or $\bot$. Instructions include skipping, register assignments, loads, stores, indirect jumps, conditional branches, conditional assignments, speculation barriers, calls, and returns. Instructions can refer to expressions, which are constructed by combining registers and values with unary and binary operators. Registers come from the set $\text{Regs}$, containing register identifiers and designated registers $\text{pc}$ and $\text{sp}$, modelling the program counter and stack pointer respectively, while values come from the set $\text{Vals}$, which includes natural numbers and $\bot$. 
In the following, we use instruction keywords to denote the set of all instructions of a given type. For instance, \texttt{beqz} is the set of all branch instructions, i.e., \( \texttt{beqz} = \{ \texttt{beqz} \ x, \ell | \ x \in \text{Regs} \wedge \ell \in \text{Vals} \} \).

### 2.3 Non-speculative Semantics of \( \mu \text{ASM} \)

\( \mu \text{ASM} \) has a small-step operational non-speculative semantics \( \Rightarrow \) that describes how programs execute without speculative execution. The judgment for this semantics is \( (p, \sigma) \xrightarrow{} (p', \sigma') \) and it reads: “a program state \( (p, \sigma) \) steps to a new program state \( (p', \sigma') \) producing observation \( \tau \)”. Programs states \( (p, \sigma) \) consist of the program \( p \) and the configuration \( \sigma \). The program \( p \) is used to look up the current instruction, whereas the configuration \( \sigma = (m, a) \) is used to read from/write to the memory \( m \) and register file \( a \). Memories map addresses (which are natural numbers) to values, whereas register files map register identifiers to values.

Most of the rules of the semantics are standard and thus omitted; we present selected rules below (see \[21\] for all rules). 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 \( x \in \text{Regs} \cup \mathbb{N} \) and \( n \in \text{Vals} \), denotes the update of a map (memory or registers), whereas \( a(x) \) denotes reading from a map. Finally, \( \sigma(x) \), where \( x \in \text{Regs} \) and \( \sigma = (m, a) \), denotes \( a(x) \).

#### Branch instructions emit observations recording the outcome of the branch (Rule \texttt{Beqz-Sat}), while memory operations emit observations recording the accessed memory (Rule Store). 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 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 Return).

#### Non-speculative behaviour \( \text{Beh}_{\text{NS}}(p) \) of a program \( p \) is the set of all traces generated from an initial state until termination using the reflexive-transitive-closure of the non-speculative semantics.

#### Symbolic semantics. Following \[21\], we introduce a symbolic non-speculative semantics \( \Rightarrow^S \) that is based on the program analysis tool spectre’s analysis. This symbolic semantics differs from \( \Rightarrow \) in two key ways: (1) concrete configurations \( \sigma \) are replaced with symbolic configurations \( \sigma^S \), and (2) path condition constraints are generated in the standard way and they are encoded as part of the symbolic trace \( \tau \). Given a symbolic trace \( \tau \), \( \mu(\tau) \) denotes the set of all concrete traces that can be obtained by concretizing \( \tau \) with values consistent with \( \tau \)’s path condition. The symbolic non-speculative behavior \( \text{Beh}_{\text{NS}}^S(p) \) of a program \( p \) consists of all symbolic traces derived by applying \( \Rightarrow^S \), and \( \mu(\text{Beh}_{\text{NS}}^S(p)) \) is the set of all concrete traces derived from \( p \)'s symbolic traces. As proved by Guarnieri et al. \[21\], \( \text{Beh}_{\text{NS}}(p) = \mu(\text{Beh}_{\text{NS}}^S(p)) \).

### 2.4 \( \bigwedge_B \) : Speculating Over Branch Instructions

To model and reason about the effects of speculation over branch instructions, Guarnieri et al. \[21\] propose three related semantics: an always-mispredict semantics (Section 2.4.1), an oracle semantics (Section 2.4.2), and a symbolic semantics (Section 2.4.3). The always-mispredict semantics, our main focus, is a safe overapproximation of the oracle semantics, which explicitly models the behavior of the branch predictor using a prediction oracle. Finally, the symbolic semantics, which is used in the Spectre program analysis tool, is the symbolic version of the always-mispredict semantics. We summarize the properties of these semantics in Section 2.4.4. With a slight abuse of notation, \( \bigwedge_B \) indicates both the three speculative semantics, and the AM one alone (since it is the most relevant one).

#### Always-mispredict (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 \[21\].

The state \( \Sigma_B \) of the AM semantics is a stack of speculative instances \( \Phi_B \) where reductions happen only on top of the stack. Whenever we start speculating, a new instance is pushed on top of the stack (Rule B:AM-branch). The instance is then popped when speculation ends (Rule B:AM-Rollback). Each instance \( \Phi_B \) contains the program \( p \), a counter \( ctr \) that uniquely identifies the speculation instance, a configuration \( \sigma \), and the remaining speculation window \( n \) describing the number of instructions that can still be executed speculatively (or \( \bot \), when no speculation is happening). Throughout the paper, we fix the maximal speculation window, i.e., the maximum number of speculative instructions, to a global constant \( \omega \).

#### Spec. States \( \Sigma_B \) and Spec. Instances \( \Phi_B \) of the AM Semantics.

This judgement for the AM semantics is: \( \Sigma_B \xrightarrow{r} \Phi_B \Sigma'_B \).
A speculative semantics \( \mathcal{L}_S \) is secure (denoted \( \mathcal{L}_S \perp_{SSS} \)) if:

- Oracle Overapproximation: \( p \vdash_x \text{SNI} \) iff \( \forall O. p \vdash^O_x \text{SNI} \)
- Symbolic Consistency: \( \text{Beh}^A(x)(p) = \mu(\text{Beh}^A_x(p)) \)
- NS Consistency: \( \text{Beh}^A_x(p) |_{ns} = \text{Beh}_{\text{NS}}(p) = \text{Beh}^O_{x}(p) |_{ns} \)

Intuitively, a secure speculative semantics is made of three components: an AM semantics, an oracle semantics, and a symbolic semantics. First, the AM semantics must overapproximate the oracle semantics (for any oracle), so it is enough to check a program \( p \) for SNI w.r.t. the AM semantics \([21, \text{Theorem 1}]\). Then, since SPECTECtor uses the symbolic semantics in the implementation, the symbolic semantics must be consistent w.r.t. the AM one \([21, \text{Proposition 2}]\). Finally, both the AM and the Oracle semantics can recover the non-speculative behaviour of a program \( p \) by applying the non-speculative projection on their traces \([21, \text{Propositions 1,3}]\). So we can execute \( p \) only once to get the (non-)speculative behaviour of that program run.

Theorem 1 states that \( \mathcal{L}_B \) is a secure speculative semantics.

**Theorem 1** \( \mathcal{L}_B \) is SSS ([21]). \( \vdash \mathcal{L}_B \perp_{SSS} \)

### 3 SPECULATION ON STORES AND RETURNS

This section defines \( \mathcal{L}_S \) and \( \mathcal{L}_R \), two novel speculative semantics that model the effects of speculative execution over store instructions (Section 3.1) and ret instructions (Section 3.2). Similarly to \( \mathcal{L}_B \), for each speculation mechanism we define three semantics: an always-mispredict semantics, an oracle semantics, and a symbolic semantics. As before, we will mostly focus on the always-mispredict semantics, which safely over-approximates the oracle one, and we will use its symbolic version to reason about leaks using SPECTECtor. Most formal details, as well as proofs, can be found in the companion technical report \([18]\).

#### 3.1 \( \mathcal{L}_S \): Speculation on Store Instructions

Modern processors write stores to main memory asynchronously to reduce delays caused by the memory subsystem. For this, 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 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.

**Example 1** (Store Speculation Vulnerability). Consider the example in Listing 2:

**Listing 2: Code vulnerable to store speculation.**

\begin{verbatim}
1. p = &secret;
2. p = &public;
3. temp = B[*p * 512];
\end{verbatim}
Assume that the store instructions in Line 1 and Line 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.

This section first introduces the extended trace model required to talk about speculation over store instructions (Section 3.1.1). Next, it presents the speculative AM semantics (Section 3.1.2) and the corresponding oracle semantics (Section 3.1.3) and symbolic semantics (Section 3.1.4). This semantics is a secure speculative semantics (Theorem 2).

**Theorem 2** $\mathcal{Q}_5$ is SSS. $\vdash \mathcal{Q}_5$ SSS

### 3.1.1 Extended Trace Model

We extend the trace model $\text{Obs}$ with startts $n$ and rlb$n$ observations to mark start and end of a speculative transaction $n$ started by a store bypass. Furthermore, we add a bypass $n$ observation denoting that the store instruction at program counter $n$ was speculatively bypassed.

$$\text{Obs} : \text{Obs} \mid \text{start}ts \ n \mid \text{rlb} \ n \mid \text{bypass} \ n$$

### 3.1.2 Speculative Semantics

The overall structure of the $\mathcal{Q}_5$ semantics is similar to that of $\mathcal{Q}_B$; speculative execution is modeled 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 $\mathcal{Q}_5$ and $\mathcal{Q}_B$ is the differing source of speculation: beqz instructions for $\mathcal{Q}_B$ and store instructions for $\mathcal{Q}_5$.

The states used in $\mathcal{Q}_5$ are similar to those of $\mathcal{Q}_B$:

- **Spec. States** $\Sigma_5 : \Phi_5$
- **Spec. Instance** $\Phi_5 : \langle \ p, ctr, \sigma, n \rangle$

Judgement $\Sigma_5 \vdash \Sigma_5'$ describes how $\Sigma_5$ steps to $\Sigma_5'$ emitting observation $r$. As in $\mathcal{Q}_B$, reductions only happen on top of the stack.

**Example 2** (Return Speculation Vulnerability). Consider the example in Listing 3 and recall that register sp is used to find return addresses saved on the stack.

### 3.1.3 Oracle Semantics

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

### 3.1.4 Symbolic Semantics

Similarly to $\mathcal{Q}_B$, the symbolic speculative semantics $\mathcal{Q}_S^S$ requires two changes w.r.t. the always-mispredict one: concrete configurations $\sigma$ and the non-speculative semantics are replaced by symbolic configurations $\sigma^S$ and the symbolic non-speculative semantics respectively. The behaviour $\text{Beh}_S^S(p)$ of a program $p$ is the set of all its symbolic traces.

### 3.2 $\mathcal{Q}_R$: Speculation on Return Instructions

The return-stack-buffer (RSB) is a small stack used by the CPU 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.

### Listing 3: A program exploiting RSB speculation.

```
Manip_Stack:
  sp ← sp + 8
  ret
Speckle:
call Manip_Stack
load eax, secret
load edx, eax
ret
Main:
call Speculate
skip
```

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 3.2.1), the oracle semantics (Section 3.2.2), and the symbolic semantics (Section 3.2.3). Then, it discusses formalising different implementations of the RSB in the CPU (Section 3.2.4). This semantics is a secure speculative semantics (Theorem 3).

**Theorem 3** $\mathcal{Q}_R$ is SSS. $\vdash \mathcal{Q}_R$ SSS
### Speculative Semantics

Unlike before, the state of $\mathcal{L}_R$ contains a model of the RSB which is used to retrieve return addresses instead of relying on the stack. Thus, speculative instances of $\mathcal{L}_R$ are extended with an additional entry $R$ for tracking the RSB, whose size is limited by a global constant $\text{Rsize}$ denoting the maximal RSB size. A speculative instance $\Phi_R$ now consists of the program $p$, the counter $ctr$, the configuration $\sigma$, the speculation window $\omega$, and the RSB $R$. As before, a state $\Sigma_R$ is a stack of speculative instances $\Phi_R$.

**Spec. States** $\Sigma_R := \Phi_R$

**Spec. Instance** $\Phi_R := (p, \text{ctr}, \sigma, R, n)$

As before, in $\Sigma_R \vdash \tau \Sigma_R$, reductions happen on the top of the stack.

\[
p(\sigma(pc)) = \text{ret} \quad \sigma = (m, a) \quad (p, \sigma) \xrightarrow{\tau} (p, \sigma') \quad \tau(\text{start}) \quad \text{ctr} \quad \text{ret} \ l
\]

\[
(p, \text{ctr}, \sigma, R, n + 1) \xrightarrow{\tau} (p, \text{ctr}, \sigma', R', n) \cdot (p, \text{ctr} + 1, \sigma'', R', j)
\]

During call instructions (Rule $\text{R:AM-Call}$), 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 $\text{R:AM-Ret-Spec}$). Then, the rule creates a new speculative instance that uses the return address from the RSB $R$. Note that speculation only happens when the return address from the RSB differs from the one on the stack (stored in $m(a(sp))$).

Here, we overview how our semantics behaves with empty and full RSB; full formalisation is available in the technical report [18]. 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.

The behaviour $\text{Beh}_R^\ddag(p)$ is the set of all traces generated from an initial state until termination using $\mathcal{L}_R$.

### 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 [9]. The only thing the oracle decides here is the size of the speculation window $\omega$.

### Symbolic Semantics

Just as before, the symbolic speculative semantics $\mathcal{L}_R^\ddag$ replaces concrete configurations and the non-speculative semantics with the symbolic non-speculative semantics respectively. We remark that the program counter $pc$ is always concrete in the symbolic non-speculative semantics [21]. As a result, the RSB only contains concrete values (and return addresses). The behaviour $\text{Beh}_R^\ddag(p)$ of a program $p$ is the set of all traces starting from an initial state until termination using the reflexive-transitive closure of the symbolic semantics.

---

1We follow the way AMD processors handle this kind of speculation [27].
which happens only on store \(\Sigma_{xy}\) (modeled by \(\mathcal{S}_{xy}\)). The combined semantics ensure that the correct rule is applied.

We overload \(\tau_{\mathcal{S}_{xy}}^{\mathcal{S}_{xy}}\) to also work on traces \(\tau\). The projection \(\tau_{\mathcal{S}_{xy}}^{\mathcal{S}_{xy}}\) deletes all speculative transactions (marked by \text{start}_y id and \text{rlb}_y id) not generated by the source semantics \(\mathcal{S}\). The definition of \(\tau_{\mathcal{S}_{xy}}^{\mathcal{S}_{xy}}\) is similar by replacing \(x\) with \(y\):

\[
\tau_{\mathcal{S}_{xy}}^{\mathcal{S}_{xy}}(s) = \epsilon \\
(\text{start}_y id \cdots \text{rlb}_y id \cdot \tau)_{\mathcal{S}_{xy}}^{\mathcal{S}_{xy}} = \tau_{\mathcal{S}_{xy}}^{\mathcal{S}_{xy}}(\tau)
\]

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

### 4.1 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:

\[
\begin{align*}
\Phi_x \downarrow_{\mathcal{S}_x} \Psi_{xy} & \quad \tau_{\mathcal{S}_{xy}}^{\mathcal{S}_{xy}}(\tau, \Psi_{xy}) \\
\Phi_y \downarrow_{\mathcal{S}_y} \Phi_{xy} & \quad \tau_{\mathcal{S}_{xy}}^{\mathcal{S}_{xy}}(\tau, \Phi_{xy}) \\
\Phi_x \downarrow_{\mathcal{S}_x} \Phi_{xy} & \quad \tau_{\mathcal{S}_{xy}}^{\mathcal{S}_{xy}}(\tau, \Phi_{xy}) \\
\Phi_y \downarrow_{\mathcal{S}_y} \Phi_{xy} & \quad \tau_{\mathcal{S}_{xy}}^{\mathcal{S}_{xy}}(\tau, \Phi_{xy})
\end{align*}
\]

The combined semantics does a step by either delegating back to the \(x\) source semantics (Rule \(\text{AM-x-step}\)) or to the \(y\) one (Rule \(\text{AM-y-step}\)). The rules rely on metaparameter \(Z_{xy}\), which is a pair of two metaparameters \(Z_{xy} := (Z_x, Z_y)\) one for \(x\) and one for \(y\). We overload the projections \(\tau_{\mathcal{S}_{xy}}^{\mathcal{S}_{xy}}\) and \(\tau_{\mathcal{S}_{xy}}^{\mathcal{S}_{xy}}\) to extract the corresponding metaparameter from \(Z_{xy}\), e.g., \(Z_x |_{xy} = Z_x\).

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 \(Z = (0, 0)\), consider the execution of the \text{beqz} instruction in Line 4 in Listing 1. The combined semantics \(\mathcal{L}_{\mathcal{S}_{xy}}\) can use Rule \(\text{AM-x-step}\) to delegate back to \(\mathcal{L}_{\mathcal{S}_{xy}}\) for \text{beqz} instructions, creating a new speculative transaction (Rule \(\text{L}:\text{AM-branch}\)). However, \(\mathcal{L}_{\mathcal{S}_{xy}}\) can also use Rule \(\text{AM-y-step}\), because \text{beqz} instructions are also handled by \(\mathcal{S}_{xy}\). Unfortunately, this does not start speculation, which happens only on store instructions (Rule \(\text{S:AM-NoSpec}\)).

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

\[
\begin{align*}
&\mathcal{L}_{\mathcal{S}_{xy}}(p, \sigma) \rightarrow (p, \sigma') \\
&\begin{cases}
\text{beqz} & \text{if } p(\sigma, \rho) \notin \text{store} \cup \text{beqz} \\
\sigma & \text{otherwise}
\end{cases}
\end{align*}
\]

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

#### 4.1.1 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.

#### 4.1.2 Symbolic Combination

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

### 4.2 Properties of Composition

We now illustrate the benefits of our composition framework. For this, we first introduce a notion of well-formed composition (Section 4.2.1), which intuitively tells when a combined semantics "makes sense". Then, we show that for well-formed compositions, if the source semantics are \(\mathcal{S}\), so is the combined semantics (Section 4.2.2). Since we proved this property for any well-formed composition in our framework, all (well-formed) compositions we present in Section 5 are \(\mathcal{S}\) 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 \(\mathcal{S}\) results we instead obtain for free.

#### 4.2.1 Well-formed Compositions

The well-formedness conditions for the composition ensures that the delegation is done properly (Definition 3), they are the minimal set of assumptions that let us derive \(\mathcal{S}\) of the combined semantics for free:

**Definition 3** (Well-formed composition). A composition \(\mathcal{L}_{xy}\) of two speculative semantics \(\mathcal{L}_x\) and \(\mathcal{L}_y\) is well-formed, written \(\mathcal{L}_{xy} : \text{WFC}\), if:

- (Confluence) Whenever \(\Sigma_{xy} \Rightarrow \Sigma_{xy}'\) and \(\Sigma_{xy} \Rightarrow \Sigma_{xy}''\), then \(\Sigma_{xy} = \Sigma_{xy}' = \Sigma_{xy}'\).
- (Projection preservation) For all \(p, \text{Beh}^A_{xy}(p) = \text{Beh}^A_{xy}(p)\) and \(\text{Beh}^A_{xy}(p) = \text{Beh}^A_{xy}(p)\).
- (Relation preservation) If \(\Sigma_{xy} \Rightarrow \Sigma_{xy} X_{xy} \Sigma_{xy}'' \Rightarrow \Sigma_{xy}'' X_{xy}''\), then \(X_{xy} \Rightarrow X_{xy} X_{xy}'' \Rightarrow X_{xy}'' X_{xy}''\).
- (Symbolic preservation) If \(\mu(\Sigma_{xy}) = \mu(\Sigma_{xy}) = \mu(\Sigma_{xy}) = \mu(\Sigma_{xy}) = \mu(\Sigma_{xy})\), then there is \(\Sigma_{xy} s.t. \Sigma_{xy} \Rightarrow \Sigma_{xy} X_{xy} \Sigma_{xy}'' \Rightarrow \Sigma_{xy}'' X_{xy}''\).

Next, we explain the well-formedness conditions:

- Confluence (point 1) ensures that the non-determinism of the combined semantics (non-deterministically delegates back to
its sources) is not harmful. Consider the assignment in Line 1 in Listing 1, $\Delta_{b+s}$ can delegate to either $\Delta_b$ or $\Delta_s$ 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 1, 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 $\text{Obs}_{xy}$ is defined as the union of the source $\text{Obs}$.

- 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 ($\Sigma_{xy}$) 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 ($\Sigma_{xy} \xmapsto{\tau} \Sigma_{xy}^* \approx_{xy} X_{xy}$), 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 ($X_{xy} \xmapsto{\tau_{xy}} X_{xy}^*$). 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 2). Thus, it is not important for the AM and the Oracle semantics to produce the same traces, just that the two state relations follow 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). Note that proving symbolic preservation is almost trivial whenever both source semantics enjoy the same property (like our semantics $\Delta_b, \Delta_s, \Sigma_s$, and $\Sigma_b$).

### 4.2.2 SSS preservation

The key result of our framework is that well-formed compositions whose sources are secure speculative semantics (SSS) are also SSS (Theorem 4). Note that our proof of Theorem 4, available in the companion technical report [18], holds for any well-formed composition in our framework and, therefore, it applies for free to all the compositions in Section 5.

**Theorem 4 ($\Delta_{xy}$ is SSS).** If $\Delta_{x}$ SSS and $\Delta_{y}$ SSS and $\Delta_{xy} : WFC$ then $\Delta_{xy} \approx_{xy} SSS$.

As a corollary of Theorem 4, we obtain that the security of well-formed compositions is related to the security of their components (Theorem 5). 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 1.

**Theorem 5 (Combined SNI Preservation).** If $\Delta_{xy} : WFC$ and $p \not\approx_{xy} SNI$ or $p \not\approx_{xy} SNI$ then $p \not\approx_{xy} SNI$.

If $\Delta_{xy} : WFC$ and $p \not\approx_{xy} SNI$ then $p \not\approx_{xy} SNI$.

These results have an immediate practical impact on Spectector: (1) Spectector’s security analysis relies on the (symbolic) speculative semantics, (2) the source semantics $\Delta_b, \Delta_s, \Sigma_s$, and $\Sigma_b$ are SSS, (3) well-formed compositions are also SSS, and (4) the composition of $\Delta_{b+S}, \Delta_{S+S}$, and $\Delta_{R+S}$ produces sound results, i.e., whenever the tool proves that a program is leak-free then the program satisfies SNI. So, the next section describes all the compositions and proves they are well-formed (this implies that they are SSS thanks to Theorem 4), whereas the section thereafter describes their implementation in Spectector.

### 5 INSTANTIATING OUR FRAMEWORK

This section describes all combinations of the speculative semantics $\Delta_{b+S}, \Delta_{S+S}$, and $\Delta_{R+S}$ (Section 5.1), $\Delta_{b+S}$ (Section 5.2), $\Delta_{S+S}$ (Section 5.3), and $\Delta_{b+S+R}$ (Section 5.4). For each one, we overview the combined AM semantics using examples (whose traces we computed using our Coq executable composed semantics) and we prove that the combined semantics is well-formed, i.e., it satisfies Definition 3. In the following, we describe in detail how the $\Delta_{S+S}$ semantics can be instantiated as part of our framework; the other combinations can be instantiated similarly and we only provide a higher-level description. Full details and well-formedness proofs are available in the companion technical report [18].

#### 5.1 $\Delta_{S+S}$ Composition

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

**Spec. States** $\Sigma_{S+S} ::= \Sigma_s + \Sigma_b$  
**Spec. Instance** $\Phi_{S+S} ::= (p, \sigma, \alpha, \omega, n)$

The union $\text{Obs}_{S+S}$ of the trace models $\text{Obs}_s$ and $\text{Obs}_b$ is defined as:

$\text{Obs}_{S+S} ::= \text{start}_s n | \text{start}_b n | \text{rlb}_s n | \text{rlb}_b n | \text{bypass} n | ...$

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

**Theorem 6 ($\Delta_{S+S}$ is well-formed).** $\vdash \Delta_{S+S} : WFC$.

Listing 4 presents a program that contains a leak that can be detected only by $\Delta_{S+S}$ but not by its components $\Delta_b$ and $\Delta_s$.

**Listing 4: $\Delta_{S+S}$ example**

```plaintext
Manip_Stack:
1  sp ← sp + 8
```
\section{\( \mathcal{B}+R \) Composition}

In this combination, the instructions that influence speculative execution are \texttt{call} and \texttt{ret} \( (\mathcal{B}_R) \) and \texttt{beqz} \( (\mathcal{R}_R) \). Thus, we set \( \mathcal{B}+R = \texttt{(call \cup ret, beqz)} \) to account for this and to allow speculation from both sources.

Theorem 7 states that \( \mathcal{B}+R \) is well-formed. This allows us to derive “for free” that \( \mathcal{B}+R \) is SSS by applying Theorem 4.

\begin{theorem} [\( \mathcal{B}+R \) is well-formed], \( \vdash \mathcal{B}+R : \text{WFC} \)
\end{theorem}

Listing 5: \( \mathcal{B}+R \) example

```plaintext
Manip_Stack:
sp <- sp + 8
ret

Speculate:
call Manip_Stack
x <- 0
beq x, L2
load eax, secret
L2:
ret

Main:
call Speculate
skip
```

Listing 5 presents a leak that can be detected only under \( \mathcal{B}+R \). The execution proceeds similarly to Listing 4 until the \texttt{ret} instruction in Line 3 is reached. Under \( \mathcal{B}_R \), mispredicting the return address leads to function \texttt{Manip_Stack} returning to Line 6. However, the \texttt{beqz} instructions in Line 7 jumps to Line 10 (since \( x = 0 \)) and speculation ends without leaking. Under \( \mathcal{R}_R \), the \texttt{beqz} instruction in Line 7 is never executed and the function \texttt{Manip_Stack} returns to Line 13 without leaking. Hence, Listing 5 is secure (i.e., it satisfies SNI) when considering \( \mathcal{B}_R \) and \( \mathcal{R}_R \) in isolation.

Under the combined semantics \( \mathcal{B}+R \), function \texttt{Manip_Stack} returns to Line 6 and the \texttt{beqz} instruction is then mispredicted. This leads to executing the \texttt{load} instructions in Line 8, which leaks secret information. The resulting traces \( (\mathcal{B}+R) \) are given below, where we highlight the secret-dependent observations. Given the length of the trace, we present only the most relevant parts, i.e., that both kinds of speculations need to have \texttt{started} for the leak to appear.

\begin{align*}
\tau_{\mathcal{B}+R}^2 & \neq \tau_{\mathcal{B}+R}^1 \overset{\text{def}}{=} \texttt{call Speculate} \cdots \texttt{start}_B \ 0 \cdots \texttt{start}_R \ 1 \cdots \texttt{start}_S \ 2 \ \texttt{bypass} \ 7 \ \texttt{load} \ p \ \texttt{load secret} \ \cdots
\end{align*}

The relation between the source semantics and their composition is visualised in Figure 1, which shows the insecure programs (with respect to SNI) detected under the different semantics. The combined semantics encompasses all vulnerable programs of \( \mathcal{B}_S \) and \( \mathcal{R}_R \) and additional programs like Listing 4. These additional programs are the reason why the semantics \( \mathcal{B}+R \) is "stronger than the sum of its parts" \( \mathcal{B}_S \) and \( \mathcal{R}_R \).

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure1}
\caption{Relating \( \mathcal{B}_S, \mathcal{R}_R \) and \( \mathcal{B}+R \) wrt SNI.}
\end{figure}

\section{\( \mathcal{B}+S \) Composition}

In this combination, speculation happens on \texttt{beqz} instructions \( (\mathcal{R}_S) \) and on \texttt{store} instructions \( (\mathcal{S}_S) \). Thus, we set \( \mathcal{B}+S = \texttt{(store, beqz)} \). Therefore, in the combined semantics \( \mathcal{B}+S \), \texttt{beqz} instructions are only executed by delegating back to \( \mathcal{B}_B \) and \texttt{store} instructions are only executed by delegating back to \( \mathcal{S}_S \). This semantics is also a well-formed composition (Theorem 8) and SSS.

\begin{theorem} [\( \mathcal{B}+S \) is well-formed], \( \vdash \mathcal{B}+S : \text{WFC} \)
\end{theorem}
Listing 1 from Section 1 contains a leak that can only be detected by $\mathcal{L}_{\text{B+S}}$ but not by its components. The traces associated with the code (\(\tau\)) are given below, where secret-dependent observations are highlighted in gray:

\[
\tau_{\text{B+S}}^{2} \neq \tau_{\text{B+S}}^{1} \quad \text{def} = \cdots \text{start}_5 \cdot \text{bypass} \cdot \cdots \text{start}_2 \cdot \text{pc} \cdot 5 \cdot \text{load} \cdot p \cdot \text{load} \cdot A + \text{secret} \cdot \text{rlb}_9 \cdot 2 \cdot \text{rlb}_{15} \cdot 1 \cdots
\]

The program is not secure under $\mathcal{L}_{\text{B+S}}$, i.e., Listing 1 $\not\models_{\text{B+S}}$ SNI.

### 5.4 $\mathcal{L}_{\text{B+S}+\text{R}}$ Composition

We conclude this section by combining all three semantics $\mathcal{L}_{\text{B}}$, $\mathcal{L}_{\text{S}}$, and $\mathcal{L}_{\text{R}}$. Our framework (Section 4) allows only combining a pair of source semantics into a combined one. For simplicity, we present $\mathcal{L}_{\text{B+S}+\text{R}}$ as a direct combination of the three source semantics (technically, we obtain $\mathcal{L}_{\text{B+S}+\text{R}}$ by combining $\mathcal{L}_{\text{B+S}}$ with $\mathcal{L}_{\text{R}}$). The metaparameter $Z_{\text{B+S}+\text{R}}$ (which we represent as a triple of values) is $(\text{call} \cup \text{ret} \cup \text{store}, \text{call} \cup \text{ret} \cup \text{beq} \cup \text{beq} \cup \text{store})$.

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

As before, $\mathcal{L}_{\text{B+S}+\text{R}}$ is a well-formed composition (Theorem 9) and we get that $\mathcal{L}_{\text{B+S}+\text{R}}$ is SSS by applying Theorem 4.

**Theorem 9 ($\mathcal{L}_{\text{B+S}+\text{R}}$ is well-formed).**\(\vdash \mathcal{L}_{\text{B+S}+\text{R}} \models \text{WFC}\)

### Listing 6: $\mathcal{L}_{\text{B+S}+\text{R}}$ example

| Line | Code                                      |
|------|-------------------------------------------|
| 1    | Manip\_Stack:                            |
| 2    | sp <- sp + 8                             |
| 3    | ret                                       |
| 4    | Speculate:                               |
| 5    | call Manip\_Stack                        |
| 6    | x <- 0                                   |
| 7    | beqz x, L2                               |
| 8    | load eax, p                              |
| 9    | load edi, eax                            |
| 10   | L2:                                       |
| 11   | ret                                       |
| 12   | Main:                                     |
| 13   | store secret, p                          |
| 14   | store pub, p                             |
| 15   | call Speculate                           |

Listing 6 depicts a leaky program that can be detected only under $\mathcal{L}_{\text{B+S}+\text{R}}$, since the program satisfies SNI under $\mathcal{L}_{\text{B}}$, $\mathcal{L}_{\text{S}}$, and $\mathcal{L}_{\text{R}}$. Under $\mathcal{L}_{\text{B+S}+\text{R}}$, the store instruction in Line 14 is bypassed. Therefore, when returning from $\text{Manip\_Stack}$, the program mispredicts the return address and speculatively returns to Line 6. Here, the beqz instruction in Line 7 is mispredicted and the load instructions are executed, which now leaks the secret value.

The resulting traces (\(\tau\)) are given below:

\[
\tau_{\text{B+S+R}}^{2} \neq \tau_{\text{B+S+R}}^{1} \quad \text{def} = \cdots \text{starts} \cdot \text{bypass} \cdot 14 \cdot \text{call Speculate} \cdots \cdot \text{start}_2 \cdot \text{ret} \cdot 6 \cdot \text{start}_3 \cdot \text{pc} \cdot 8 \cdot \text{load} \cdot p \cdot \text{load} \cdot \text{secret} \cdot \text{rlb}_3 \cdot \text{rlb}_3 \cdot 2 \cdot \text{rlb}_5 \cdot 1 \cdots
\]

Thus, the program is not secure, i.e., Listing 6 $\not\models_{\text{B+S+R}}$ SNI.

### 6 IMPLEMENTATION AND EVALUATION

This section describes how our combined semantics can be used to detect leaks introduced by the interaction of multiple speculation mechanisms. For this, we extended Spectector, a symbolic analysis tool for speculative leaks against $\mathcal{L}_{\text{R}}$, with the semantics for $\mathcal{L}_{\text{B}}$ and $\mathcal{L}_{\text{S}}$, and for all the combinations from Section 5 (Section 6.1).

Using Spectector, we analyze a corpus of 49 microbenchmarks containing speculative leaks generated by different speculation mechanisms (Section 6.2). With these experiments, we aim to show that (1) our $\mathcal{L}_{\text{S}}$ and $\mathcal{L}_{\text{R}}$ speculative semantics can correctly identify speculative leaks associated with speculation over store-bypasses and return instructions, and (2) our combined semantics can detect novel leaks that are otherwise undetectable when considering single speculation mechanisms in isolation.

#### 6.1 Implementation

We implemented all our semantics (the symbolic versions of $\mathcal{L}_{\text{B}}$ and $\mathcal{L}_{\text{R}}$ plus all compositions from Section 5) as an extension of Spectector [21]. The implementation of compositions closely follows the structure of our framework. As in Section 5, selecting one of the composed semantics in Spectector sets the metaparameter $Z$, which is used to delegate back to the correct individual semantics. Spectector then uses symbolic execution together with self-composition [6] and an SMT solver to check for SNI against $\mathcal{L}_{\text{S}}$. Due to this setup, we inherit all limitations of Spectector’s speculative analysis, e.g., path explosion due to symbolic execution and limitations in the translation from x86 to $\mu$Assm. We refer to [21] for an in-depth discussion of such limitations.

#### 6.2 Experiments

**Benchmarks:** We analyze 49 snippets of code containing leaks resulting from speculation over branch, $\text{store}/\text{load}$, and $\text{ret}$ instructions (and their combinations):

- **Spectre-STL:** 13 programs are variants of the Spectre-STL vulnerability. They exploit speculation over memory disambiguation, and they have been used as benchmarks in prior work [14, 32]. For each program, we also analyze a patched version where a manually inserted lfence instruction stops speculation over store-bypasses and prevents the leak.

- **Spectre-RSB:** 5 programs are variants of the Spectre-RSB vulnerability. They exploit speculation over return instructions, and they are obtained from the safeside [1] and transientfail [8] projects. For each program, we also analyze manually patched versions obtained by (1) inserting lfences after call instructions (i.e., at the instruction address where $\text{ret}$ speculatively returns), and (2) using the modified $\text{ret}$ test defense proposed in [27, Section 6.1].

- **Spectre-Comb:** 4 programs contain leaks that arise from combining speculation mechanisms. These are the programs depicted in listing 1, listing 5, listing 4, and listing 6 and discussed in Section 5. For each program, we also analyze a manually patched version where $\text{lfence}$ instructions prevent the speculative leaks.

\[^{3}\text{Out of the three Spectre-RSB examples from safeside [1], we analyze the only one that works against an x86 RSB like the one supported by Spectector. Programs ca_ip, ca_eop, and sa_ip from transientfail [8] rely on concurrent execution. Since Spectector does not support concurrency, we hardcode the worst-case interleaving in terms of speculative leakage in our benchmark.}\]
Experimental setup: The benchmarks for Spectre-STL and Spectre-RSB are implemented in C and compiled with Gcc 11.1.0 and we manually inserted `fence/retpoline` countermeasures in the patched versions. The benchmarks for Spectre-Comb are directly formalised in μAsm. We run all our experiments on a laptop with a Dual Core Intel Core i5-7200U CPU and 8GB of RAM.

Spectre-STL: Figure 2a reports the results of analysing the programs in the Spectre-STL benchmark\(^4\). Using the \(\mathcal{L}_2\) semantics, Spectrector 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 [14, 32]). Observe that Binsec/Haunted [14] flags program 13 as secure since the program can only speculatively leak initial values from the stack, which Binsec/Haunted treats as public by default [2]. Since we assume initial memory values to be secret (like Ponce de León and Kinder [32]), Spectrector correctly detected the leak in program 13. Spectrector also successfully proved that all patched programs (where an `fence` is added between `store` instructions) satisfy SNI and are free of speculative leaks.

Spectre-RSB: Figure 2b reports the analysis results on the Spectre-RSB programs. Using \(\mathcal{L}_R\), Spectrector successfully detected leaks in all unpatched programs. Moreover, Spectrector successfully proved that the patched programs where a `fence` instruction is added after every `call` satisfy SNI, i.e., they are free of speculative leaks. Spectrector also successfully proved secure the programs patched using the modified `retpoline` defense proposed by Maisuradze and Rossow [27], which replaces return instructions with a construct that traps the speculation in an infinite loop.

4We had to slightly modify programs 02, 05, and 06 due to limitations of Spectrector’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 Spectrector’s limitations in symbolic execution, when analyzing program 09, which contains a loop.

Spectre-Comb: Figure 3a reports the results of our analysis on the Spectre-Comb programs, which involve leaks arising from a combination of multiple speculation mechanisms. Spectrector equipped with the single semantics \(\mathcal{L}_R\), \(\mathcal{L}_S\), and \(\mathcal{L}_R\) is not able to detect the speculative leaks in any of the 4 programs and, therefore, proves them secure. This is expected since the programs contain leaks that arise from a combination of semantics. Spectrector can successfully identify leaks in listing 1, listing 5, listing 4 when using, respectively, the semantics \(\mathcal{L}_{R+S}, \mathcal{L}_{S+R}, \) and \(\mathcal{L}_{R+S+R}\). Each semantics, however, fail in detecting leaks in the other programs, and all of them fail in detecting a leak in listing 6 as expected. Finally, Spectrector is able to successfully detect leaks in all programs when using the \(\mathcal{L}_{R+S+R}\) semantics that combines all speculation mechanisms studied in this paper.

We also analyzed programs manually patched with `fence` statements (“listing 1 Fence”, “listing 5 Fence”, “listing 4 Fence”, and “listing 6 Fence” in Figure 3a). As before, Spectrector successfully proved the security of patched programs. Even for leaks that arise from multiple speculation mechanisms, it is often sufficient to insert a single `fence` to secure the entire program, e.g., an `fence` after the `beqz` instruction in Listing 5 is enough to make the program SNI with respect to \(\mathcal{L}_{R+S+R}\).

Figure 3b reports the average execution time (for 1000 executions) of Spectrector’s analysis for the Spectre-Comb programs under the different semantics. We highlight the following findings:

- For the programs patched with `fence`, Spectrector’s execution time under a combined semantics is larger than Spectrector’s execution times under the corresponding source semantics. This follows from the combined semantics exploring (a) everything explored by the source semantics as well as (b) additional statements resulting from extra interactions between the source semantics. Note that the placement of `fences` influence execution time. For instance, the execution time for “listing 5 Fence” under \(\mathcal{L}_R\) and
| Test case       | $\mathcal{A}$ | $\mathcal{B}$ | $\mathcal{R}$ | $\mathcal{A} + \mathcal{B}$ | $\mathcal{B} + \mathcal{R}$ | $\mathcal{A} + \mathcal{B} + \mathcal{R}$ |
|----------------|--------------|--------------|--------------|-----------------------------|-----------------------------|----------------------------------|
| listing 1      | (I)          | ●            | ●            | ○                           | ●                           | ○                                |
| listing 5      | (I)          | ●            | ●            | ○                           | ●                           | ○                                |
| listing 4      | (I)          | ●            | ●            | ○                           | ●                           | ○                                |
| listing 6      | (I)          | ●            | ●            | ○                           | ●                           | ○                                |
| listing 1 Fence| (S)          | ●            | ●            | ○                           | ●                           | ○                                |
| listing 5 Fence| (S)          | ●            | ●            | ○                           | ●                           | ○                                |
| listing 4 Fence| (S)          | ●            | ●            | ○                           | ●                           | ○                                |
| listing 6 Fence| (S)          | ●            | ●            | ○                           | ●                           | ○                                |

(a) 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. Next to each program, we report if it is Secure or Insecure w.r.t. $\mathcal{A} + \mathcal{B} + \mathcal{R}$.

(b) Average execution time for Spectector's analysis for the code snippets in the Spectre-Comb benchmark (over 1000 samples) for the relevant individual and composed speculative semantics. The white bar ("non-spec") represents the analysis time w.r.t. μAsm non-speculative semantics.

Figure 3: Results of the Spectre-Comb benchmark, where "listing x Fence" is the patched version (using ifence) of "listing x".

$\mathcal{A} + \mathcal{B} + \mathcal{R}$ is similar because the ifence is placed just after the branch instruction of Line 7, thereby stopping $\mathcal{A}$-speculation.

- For most of the unpatched programs, execution time under a combined semantics is again larger than the execution times under the source semantics. This is, however, not always the case. For instance, Spectector’s execution time for listing 6 and $\mathcal{A} + \mathcal{B} + \mathcal{R}$ is larger than its execution time for $\mathcal{A} + \mathcal{B} + \mathcal{R}$. This is due to Spectector’s terminating early after finding a violation of SNI, which happens under $\mathcal{A} + \mathcal{B} + \mathcal{R}$ but not under $\mathcal{A} + \mathcal{B} + \mathcal{R}$ (see also Figure 3a).

7 DISCUSSION

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).

Other speculation mechanisms: There are many speculation mechanisms beyond those modeled in $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{B} + \mathcal{R}$:

- Speculation over indirect jumps [24] can be modeled as an always-mispredict semantics (similarly to $\mathcal{A}$) where mispredicted paths can start from any other statement. This, however, makes automated reasoning challenging due to the large number of speculative paths. Mechanisms like Intel Control-Flow-Integrity [34] can improve the situation by restricting potential jump targets.
- 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 [27] or RSBs that fall back to indirect branch prediction [41]). Some ARM processors, moreover, use straight-line speculation that allows CPUs to speculatively bypass a ret instruction and execute the instructions following it. Both kinds of speculation can be modeled by modifying the Rule $\mathcal{R}$:AM-Ref-Spec rule in $\mathcal{A} + \mathcal{B} + \mathcal{R}$.
- Many proposals for value prediction over different kinds of instructions exist [26, 29, 35]. 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.

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, $L_{\text{S+R}}$ cannot speculate over the implicit 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 $L_R$ with a semantics modeling straight-line speculation. 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.

We leave addressing both limitations as future work.

8 RELATED WORK

Speculative execution attacks: After Spectre [24] has been disclosed to the public in 2018, researchers have identified many other speculative execution attacks [4, 7, 25, 27, 42]. These attacks differ in the exploited speculation sources [23, 25, 27], the covert channels [33, 36, 37] used, or the target platforms [12]. We refer the reader to Canella et al. [8] for a survey of existing attacks.

Security conditions for speculative leaks: Researchers have proposed many program-level properties for security against speculative leaks, which can be classified in three main groups [10]:

1. Non-interference definitions ensure the security of speculative and non-speculative instructions. For instance, speculated constant-time [9] (used also in [3, 14, 38]) extends the constant-time security condition to account also for transient instructions.

2. Relative non-interference definitions [11, 19, 21, 22] ensure that transient instructions do not leak more information than what is leaked by non-transient instructions. For instance, speculative non-interference [21], which we build on, (used also in [20, 31]) restricts the information leaked by speculatively executed instructions (without constraining what can be leaked non-speculatively).

3. Definitions that formalise security as a safety property [31, 32], which may over-approximate definitions from the groups above.

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 [10] for a comprehensive survey on the topic. In the following, we discuss the approaches that are more relevant to our paper.

Our speculative semantics $L_S$ and $L_R$ capture the effects of transient instructions at a rather high level, and they are inspired by the always-mispredict $L_R$ semantics from [21]. Our $L_S$ semantics is also similar to the CT-BPAS speculation contract used by the Revizor testing tool [30]. In contrast, other approaches, which we overview next, explicitly model microarchitectural components like multiple pipeline stages, caches, and branch predictors.

For instance, KleE Spectre [39] and SpecuSym [22] consider a semantics that explicitly model the cache, which enable reasoning about the cache content. McIroy et al. [28] 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. [9]’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.

Binsec/Haunted [14] detect violations of speculative constant-time due to speculation over store-bypasses and branch instructions. For this, they explicitly model the store buffer, which $L_S$ abstracts away. Barthe et al. [5] extend the Jasmin [3] cryptographic verification framework to reason about speculative constant-time and supports speculation over store-bypasses and 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 [9] 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 [13] and Disselkoen et al. [15] capture the effects of branch speculation but both lack program analyses.

Ponce de León and Kinder [32] 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 theirs semantics [32, §IV.F], which should allow them to detect vulnerabilities like Listing 1 (which we detect under $L_{\text{S+R}}$). 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 4). 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.

9 CONCLUSION AND FUTURE WORK

This paper presented new speculative semantics for speculation on store and return instructions. It also defined a general framework to reason about the composition of different speculative semantics and instantiated the framework with our new speculative semantics $L_S$ and $L_R$ and the semantics by Guarnieri et al. [21]. Our framework yields security of the composed semantics (almost) for free, given the security of its parts. All the new semantics have been implemented in the SPECTECTOR program analysis tool, which correctly detects all vulnerabilities in existing and novel benchmarks.

Acknowledgments: This work was partially supported by the Madrid regional government under the project S2018/TCS-4339 BLOQUES-CM, by the Spanish Ministry of Science, Innovation, and Universities under the project RTI2018-102043-B-I00 SCUM, by the Italian Ministry of Education under the Rita Levi Montalcini grant (2019 call), by the German Ministry for Education and Research under the project CISPA-Stanford Center for Cybersecurity (funding number: 16KIS0761), and by a gift from Intel Corporation.
