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.

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

For example, the code in Listing 1 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. 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 $\mathcal{J}_{\mathsf{S}}$ and $\mathcal{J}_{\mathsf{R}}$, 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 $\mathcal{J}_{\mathsf{S}}$, $\mathcal{J}_{\mathsf{R}}$ and $\mathcal{J}_{\mathsf{B}}$, the semantics for speculation over branch instructions from [21], creating all the possible compositions ($\mathcal{J}_{\mathsf{B+S}}$, $\mathcal{J}_{\mathsf{S+R}}$, $\mathcal{J}_{\mathsf{B+R}}$, and $\mathcal{J}_{\mathsf{B+S+R}}$) and proving their security (Section 5). All these semantics are mechanised in Coq, and we write 🐓 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: μASM, 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 μAsm, a simple assembly-style language, followed by $\mathcal{J}_{\mathsf{B}}$, 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 τ. These events, which we call *observations*, model timing leaks through cache and control flow while abstracting away low-level microarchitectural details.

$$\begin{aligned} Obs &::= \texttt{load}\ n \mid \texttt{store}\ n \mid \texttt{pc}\ n \mid \texttt{call}\ f \mid \texttt{ret}\ n & \tau &::= \varepsilon \mid Obs \\ &\mid \texttt{start}_x\ n \mid \texttt{rlb}_x\ n & \overline{\tau} &::= \emptyset \mid \overline{\tau} \cdot \tau \end{aligned}$$

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

The *non-speculative projection* $\upharpoonright_{ns}$ [21] of a trace $\overline{\tau}$ deletes all speculative observations by removing all sub-traces enclosed between $\texttt{start}_x\ n$ and $\texttt{rlb}_x\ n$. 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 ϕ, 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 ϕ consists of a list of public registers and public memory locations. Two configurations σ^1, σ^2 are called *low-equivalent* for a policy ϕ, written $\sigma^1 \backsim_\phi \sigma^2$, if they agree on all register and memory locations in ϕ. The *speculative semantics* x defines how (speculative) traces describing the program behaviour are generated. We indicate that program p generates trace $\overline{\tau}$ from state σ with semantic x as $Beh_x^{\mathcal{A}}(p, \sigma) = \overline{\tau}$. We formalise multiple speculative semantics in later sections, each one instantiating $Beh_x^{\mathcal{A}}(p, \sigma)$.

A program p satisfies SNI (Definition 1) for a speculative semantics x if any pair of low-equivalent initial configurations σ^1 and σ^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 $p \vdash_x$ SNI) if for all σ^1, σ^2, if $\sigma^1 \backsim_\phi \sigma^2$ and $Beh_x^{\mathcal{A}}(p, \sigma^1)\upharpoonright_{ns} = Beh_x^{\mathcal{A}}(p, \sigma^2)\upharpoonright_{ns}$ then $Beh_x^{\mathcal{A}}(p, \sigma^1) = Beh_x^{\mathcal{A}}(p, \sigma^2)$.*

2.2 μAsm

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

μAsm is an assembly-like language whose syntax is presented above. Programs p in μAsm 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 *Regs*, containing register identifiers and designated registers **pc** and **sp** modelling the program counter and stack pointer respectively, while values come from the set *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, **beqz** is the set of all branch instructions, i.e., $\textbf{beqz} = \{\textbf{beqz}\ x, l \mid x \in \mathit{Regs} \wedge l \in \mathit{Vals}\}$.

2.3 Non-speculative Semantics of μAsm

μAsm has a small-step operational non-speculative semantics $\rightarrow$ that describes how programs execute without speculative execution. 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 $\langle p, \sigma\rangle$ consist of the program p and the configuration σ. The program p is used to look up the current instruction, whereas the configuration $\sigma = \langle m, a\rangle$ 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 $x \in \mathit{Regs} \cup \mathbb{N}$ 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 $a(x)$.

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

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

$$\text{(Call)}\quad \frac{\begin{array}{cc} p(\sigma(\mathbf{pc})) = \textbf{call}\ f & \mathcal{F}(f) = n \\ a' = a[\mathbf{pc} \mapsto n, sp \mapsto a(sp) - 8] & m' = [a'(sp) \mapsto a(\mathbf{pc}) + 1] \end{array}}{\langle p, \langle m, a\rangle\rangle \xrightarrow{\texttt{call}\ f} \langle p, \langle m', a'\rangle\rangle}$$

$$\text{(Return)}\quad \frac{\begin{array}{c} p(\sigma(\mathbf{pc})) = \textbf{ret} \qquad l = m(a(sp)) \\ a' = a[pc \mapsto l, sp \mapsto a(sp) + 8] \end{array}}{\langle p, \langle m, a\rangle\rangle \xrightarrow{\texttt{ret}\ l} \langle p, \langle m, a'\rangle\rangle}$$

Branch instructions emit observations recording the outcome of the branch (Rule 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).

The *non-speculative behaviour* $\mathit{Beh}_{NS}(p)$ 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.

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

2.4 👻$_{\mathrm{B}}$: 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 Spectector 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, 👻$_{\mathrm{B}}$ indicates both the three speculative semantics, and the AM one alone (since it is the most relevant one).

2.4.1 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 Σ_{B} of the AM semantics is a stack of speculative instances Φ_{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 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 $\perp$ 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 ω.

$$\mathit{Spec.\ States}\ \Sigma_{\mathrm{B}} ::= \overline{\Phi}_{\mathrm{B}} \qquad \mathit{Spec.\ Instances}\ \Phi_{\mathrm{B}} ::= \langle p, \mathit{ctr}, \sigma, n\rangle$$

This judgement for the AM semantics is: $\Sigma_{\mathrm{B}} \overset{\tau}{=}\!\!👻_{\mathrm{B}}\ \Sigma'_{\mathrm{B}}$.

$$\text{(B:AM-branch)}\quad \frac{\begin{array}{ccc} p(\sigma(\mathbf{pc})) = \textbf{beqz}\ x, \ell & \langle p, \sigma\rangle \xrightarrow{\tau} \langle p, \sigma'\rangle & j = \mathit{min}(\omega, n) \\ \sigma'' = \sigma[\mathbf{pc} \mapsto l'] & & \overline{\tau} = \tau \cdot \texttt{start}_{\mathrm{B}}\ \mathit{ctr} \cdot \texttt{pc}\ l \\ \multicolumn{3}{c}{l' = \begin{cases} \sigma(\mathbf{pc}) + 1 & \text{if } \sigma'(\mathbf{pc}) = l \\ l & \text{if } \sigma'(\mathbf{pc}) \neq l \end{cases}} \end{array}}{\langle p, \mathit{ctr}, \sigma, n+1\rangle \overset{\overline{\tau}}{=}\!\!👻_{\mathrm{B}}\ \langle p, \mathit{ctr}, \sigma', n\rangle \cdot \langle p, \mathit{ctr}+1, \sigma'', j\rangle}$$

$$\text{(B:AM-NoSpec)}\quad \frac{p(\sigma(\mathbf{pc})) \notin \textbf{beqz} \cup Z_{\mathrm{B}} \qquad \langle p, \sigma\rangle \xrightarrow{\tau} \langle p, \sigma'\rangle}{\langle p, \mathit{ctr}, \sigma, n+1\rangle \overset{\tau}{=}\!\!👻_{\mathrm{B}}\ \langle p, \mathit{ctr}, \sigma', n\rangle}$$

$$\frac{n' = 0 \text{ or p is stuck}}{\langle p, ctr, \sigma, n\rangle \cdot \langle p, ctr', \sigma', n'\rangle \xRightarrow{\texttt{rlb}_{\mathsf{B}}\ ctr} \Omega_{\mathsf{B}} \langle p, ctr', \sigma, n\rangle}\ (\mathsf{B}\text{:AM-Rollback})$$

As mentioned, Rule B:AM-branch 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 [21]: it applies to instructions that are not branch instructions (as in [21]) 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 4.1). Instantiating Z_{B} allows us to restrict when to apply non-speculative steps in composed semantics. When we consider Ω_{B} in isolation, Z_{B} is the empty set (so, Rule B:AM-NoSpec applies to everything except branch instructions as in [21]). However, we will instantiate Z_{B} in different manners when building the composed semantics. In the following, we write $\Omega_{\mathsf{B}}^{Z_{\mathsf{B}}}$ to stress the value of Z_{B} when needed but we often omit Z_{B} for simplicity.

The always-mispredict behaviour $Beh_{\mathsf{B}}^{\mathcal{A}}(p)$ of a program p is the set of all traces generated from an initial state until termination using the reflexive-transitive closure of $\Rightarrow\Omega_{\mathsf{B}}$.

2.4.2 Oracle Semantics. The oracle semantics explicitly models the branch predictor using an oracle O_{B} that relies on the branching history h of the program p to predict branch outcomes.

Here, we quickly summarize the key differences with the AM semantics; see [21] for the full definition. 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. Finally, whenever the speculation window of an instance anywhere on the stack reaches 0, the execution needs to be rolled back or committed. 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.

As before, the behaviour $Beh_{\mathsf{B}}^{O}(p)$ of a program p under the oracle semantics is the set of all its traces until termination.

2.4.3 Symbolic Semantics. The symbolic speculative semantics $\Omega_{\mathsf{B}}^{\mathcal{S}}$ works on symbolic speculative states $\Sigma_{\mathsf{B}}^{\mathcal{S}}$, and it is used in Spectector [21]. The only two differences w.r.t. the AM semantics are that (1) concrete states Σ_{B} are replaced with symbolic states $\Sigma_{\mathsf{B}}^{\mathcal{S}}$, which store symbolic configurations $\sigma^{\mathcal{S}}$ instead of concrete configurations σ, and (2) the semantics uses the symbolic non-speculative semantic instead of the concrete one. The rules of the symbolic semantics look like those of the AM one, and the behaviour $Beh_{\mathsf{B}}^{\mathcal{S}}(p)$ of a program p is defined as for the AM semantics.

2.4.4 Properties of Ω_{B}. Guarnieri et al. [21] prove several properties relating the three semantics we presented above, which were instrumental in proving Spectector's security. We recap these properties in a single definition (Definition 2), which we will prove for all semantics in this paper. In the definition we indicate that a program p satisfies SNI w.r.t. the oracle semantics as $p \vdash_{\mathsf{B}}^{O}$ SNI.

Definition 2 (Secure Speculative Semantics). *A speculative semantics Ω_x is secure (denoted $\vdash \Omega_x$ SSS) if:*

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

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, 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, 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, Propositions 1,3]. So we can execute p only once to get the (non-)speculative behaviour of that program run.

Theorem 1 states that Ω_{B} is a secure speculative semantics.

Theorem 1 (Ω_{B} is *SSS* [21]). $\vdash \Omega_{\mathsf{B}}$ *SSS*

3 SPECULATION ON STORES AND RETURNS

This section defines Ω_{S} and Ω_{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 Ω_{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 Ω_{S}: Speculation on Store Instructions

Modern processors write **store**s 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.

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

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 ($\mathcal{J}_{S}$ IS *SSS*). $\vdash \mathcal{J}_{S}$ *SSS*

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

$$Obs_{S} ::= Obs \mid \texttt{start}_{S}\ n \mid \texttt{rlb}_{S}\ n \mid \texttt{bypass}\ n$$

3.1.2 Speculative Semantics. The overall structure of the $\mathcal{J}_{S}$ semantics is similar to that of $\mathcal{J}_{B}$: 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 $\mathcal{J}_{S}$ and $\mathcal{J}_{B}$ is the differing source of speculation: **beqz** instructions for $\mathcal{J}_{B}$ and **store** instructions for $\mathcal{J}_{S}$.

The states used in $\mathcal{J}_{S}$ are similar to those of $\mathcal{J}_{B}$:

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

Judgement $\Sigma_{S} \overset{\tau}{=\mathcal{J}}_{S} \Sigma'_{S}$ describes how Σ_{S} steps to Σ'_{S} emitting observation τ. As in $\mathcal{J}_{B}$, reductions only happen on top of the stack.

$$\dfrac{\begin{matrix} p(\sigma(\mathbf{pc})) = \mathbf{store}\ x, e & \langle p, \sigma\rangle \xrightarrow{\tau} \langle p, \sigma'\rangle & j = min(\omega, n) \\ \sigma'' = \sigma[\mathbf{pc} \mapsto \sigma(\mathbf{pc}) + 1] & \tau' = \tau \cdot \texttt{bypass}\ \sigma(\mathbf{pc}) \cdot \texttt{start}\ ctr & \end{matrix}}{\langle p, ctr, \sigma, n+1\rangle \overset{\tau'}{=\mathcal{J}}_{S} \langle p, ctr, \sigma', n\rangle \cdot \langle p, ctr+1, \sigma'', j\rangle}\ (\text{S:AM-Store})$$

$$\dfrac{p(\sigma(\mathbf{pc})) \notin \mathbf{store} \cup Z_{S} \qquad \langle p, \sigma\rangle \xrightarrow{\tau} \langle p, \sigma'\rangle}{\langle p, ctr, \sigma, n+1\rangle \overset{\tau}{=\mathcal{J}}_{S} \langle p, ctr, \sigma', n\rangle}\ (\text{S:AM-NoSpec})$$

To model the effect of bypassing a **store** instruction, Rule S:AM-Store 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.

Similarly to $\mathcal{J}_{B}$, all instructions that are not **store** instructions (and are not in Z_{S}) are handled by delegating back to the non-speculative semantics (Rule S:AM-NoSpec) and when the speculation window reaches 0, a roll back occurs that pops the topmost speculative instance from the stack.

The set $Beh^{\mathcal{A}}_{S}(p)$ contains all traces generated from an initial state until termination using the reflexive-transitive closure of $=\mathcal{J}_{S}$.

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 $Beh^{O}_{S}(p)$ 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.

3.1.4 Symbolic Semantics. Similarly to $\mathcal{J}^{S}_{B}$, the symbolic speculative semantics $\mathcal{J}^{S}_{S}$ requires two changes w.r.t. the always-mispredict one: concrete configurations σ and the non-speculative semantics are replaced by symbolic configurations σ^{S} and the symbolic non-speculative semantics respectively. The behaviour $Beh^{S}_{S}(p)$ of a program *p* is the set of all its symbolic traces.

3.2 $\mathcal{J}_{R}$: 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.

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.

Listing 3: 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
```

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 ($\mathcal{J}_{R}$ IS *SSS*). $\vdash \mathcal{J}_{R}$ *SSS*

3.2.1 Speculative Semantics. Unlike before, the state of Ω_{R} contains a model of the RSB which is used to retrieve return addresses instead of relying on the stack. Thus, speculative instances of Ω_{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 $\overline{\Phi}_{\mathsf{R}}$.

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

As before, in $\Sigma_{\mathsf{R}} \overset{\tau}{\Rightarrow}\Omega_{\mathsf{R}}\ \Sigma_{\mathsf{R}}$ reductions happen on the top of the stack.

$$\frac{\begin{matrix} p(\sigma(\mathbf{pc})) = \mathbf{ret} & \sigma = \langle m, a\rangle & \langle p, \sigma\rangle \xrightarrow{\tau} \langle p, \sigma'\rangle \\ \mathbb{R} = \mathbb{R}' \cdot l & j = min(\omega, n) & l \neq m(a(\mathbf{sp})) \\ \sigma'' = \sigma[\mathbf{pc} \mapsto l, \mathbf{sp} \mapsto a(\mathbf{sp}) + 8] & & \overline{\tau} = \tau \cdot \mathtt{start}_{\mathsf{R}}\ ctr \cdot \mathtt{ret}\ l \end{matrix}}{\langle p, ctr, \sigma, \mathbb{R}, n+1\rangle \overset{\overline{\tau}}{\Rightarrow}\Omega_{\mathsf{R}} \langle p, ctr, \sigma', \mathbb{R}', n\rangle \cdot \langle p, ctr+1, \sigma'', \mathbb{R}', j\rangle}\ (\mathsf{R}\text{:AM-Ret-Spec})$$

$$\frac{\begin{matrix} p(\sigma(\mathbf{pc})) = \mathbf{call}\ f & \langle p, \sigma\rangle \xrightarrow{\tau} \langle p, \sigma'\rangle \\ \mathbb{R}' = \mathbb{R} \cdot \langle a(\mathbf{pc}) + 1\rangle & |\mathbb{R}| < \mathbb{R}_{size} \end{matrix}}{\langle p, ctr, \sigma, \mathbb{R}, n+1\rangle \overset{\tau}{\Rightarrow}\Omega_{\mathsf{R}} \langle p, ctr, \sigma', \mathbb{R}', n\rangle}\ (\mathsf{R}\text{:AM-Call})$$

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-Ret-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; 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.[1]

The behaviour $Beh^{\mathcal{A}}_{\mathsf{R}}(p)$ is the set of all traces generated from an initial state until termination using $\Rightarrow\Omega_{\mathsf{R}}$.

3.2.2 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 [9]. The only thing the oracle decides here is the size of the speculation window ω.

3.2.3 Symbolic Semantics. Just as before, the symbolic speculative semantics Ω^{S}_{R} replaces concrete configurations and the non-speculative semantics with symbolic configurations and 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 $Beh^{S}_{\mathsf{R}}(p)$ 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.

[1]We follow the way AMD processors handle this kind of speculation [27].

3.2.4 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 [27]. 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).

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

We remark that extending Ω_{R} to support different RSBs implementations can be done with minimal effort.

4 A FRAMEWORK FOR COMPOSING SPECULATIVE SEMANTICS

The presented speculative semantics allow us to verify programs for violations of SNI but they do not capture the vulnerability in Listing 1, as the traces of Example 3 show.

Example 3 (SNI for Listing 1, **).** The traces generated are:

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

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

The program in Listing 1 seems secure since there is no secret value leaked in the speculative transaction; thus the program satisfies SNI for Ω_{B} and Ω_{S} in isolation. However, this program speculatively leaks when considering speculation over **beqz** and **store** instructions, but we need our combined semantics to detect this vulnerability; see Section 5.3.

The vulnerability only appears when the branch predictor (Section 2.4.1) and the memory disambiguator (Section 3.1.2) 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 different 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.

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 4.1). Then, we present the notion of *well-formed* composition which we use to study the properties of composed semantics (Section 4.2).

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 $\upharpoonright_{xy}$ and two projections $\upharpoonright^{x}_{xy}$ and

$\upharpoonright^y_{xy}$ that return the first and second projection of the pair from $\upharpoonright_{xy}$. These functions are lifted to states by applying them pointwise:

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

For example, the $\Phi_{\mathsf{S+R}}$ state resulting from the union of Φ_{S} and Φ_{R} states (from Section 3.1.2 and Section 3.2.1 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 $\cdot\upharpoonright^{\mathsf{S}}_{\mathsf{S+R}}$ of a $\Phi_{\mathsf{S+R}}$ state returns the Φ_{S} subpart, i.e., all but the return stack buffer.

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

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

We indicate source semantics for x and y as $\mathcal{J}_x$ and $\mathcal{J}_y$ respectively and use $\mathcal{J}_{xy}$ 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:

$$\text{(AM-x-step)}\quad \frac{\Phi_{xy}\upharpoonright^x_{xy} \overset{\tau}{\Longrightarrow}\mathcal{J}_x^{Z_{xy}\upharpoonright^x_{xy}}\ \overline{\Phi}'_{xy}\upharpoonright^x_{xy}}{\Phi_{xy} \overset{\tau}{\Longrightarrow}\mathcal{J}_{xy}^{Z_{xy}}\ \overline{\Phi}'_{xy}}$$

$$\text{(AM-y-step)}\quad \frac{\Phi_{xy}\upharpoonright^y_{xy} \overset{\tau}{\Longrightarrow}\mathcal{J}_y^{Z_{xy}\upharpoonright^y_{xy}}\ \overline{\Phi}'_{xy}\upharpoonright^y_{xy}}{\Phi_{xy} \overset{\tau}{\Longrightarrow}\mathcal{J}_{xy}^{Z_{xy}}\ \overline{\Phi}'_{xy}}$$

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). 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 $\upharpoonright^x_{xy}$ and $\upharpoonright^y_{xy}$ to extract the corresponding metaparameter from Z_{xy}, e.g., $Z_{xy}\upharpoonright^x_{xy} = Z_x$.

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

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

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

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

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 $\mathcal{J}^{\mathcal{S}}_{xy}$ uses the symbolic source semantics for delegation. Furthermore, the new notation (union, projections) is lifted to the symbolic combination to create the symbolic states $\Sigma^{\mathcal{S}}_{xy}$. The behaviour $Beh^{\mathcal{S}}_{xy}(p)$ 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 *SSS*, 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 *SSS 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 *SSS* 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 *SSS* of the combined semantics for free:

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

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

(2) (Projection preservation) *For all p, $Beh^{\mathcal{A}}_x(p) = Beh^{\mathcal{A}}_{xy}(p)\upharpoonright^x_{xy}$ and $Beh^{\mathcal{A}}_y(p) = Beh^{\mathcal{A}}_{xy}(p)\upharpoonright^y_{xy}$.*

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

(4) (Symbolic preservation) *If $\Sigma^{\mathcal{S}}_{xy} \overset{\tau_{\mathcal{S}}}{\Longrightarrow}\mathcal{J}^{\mathcal{S}}_{xy}\ \Sigma^{\mathcal{S}'}_{xy}$ and $\mu(\Sigma^{\mathcal{S}}_{xy}) = \Sigma_{xy}$, then there is Σ'_{xy} s.t. $\Sigma_{xy} \overset{\mu(\tau_{\mathcal{S}})}{\Longrightarrow}\mathcal{J}_{xy}\ \Sigma'_{xy}$ 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 1. $\text{👻}_{\mathsf{B+S}}$ can delegate to either 👻_{B} or 👻_{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 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 ($\Sigma_{xy} \overset{\overline{\tau}}{=}\text{👻}^{*}_{xy}\ \Sigma'_{xy}$), 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 ($X_{xy} \overset{\overline{\tau}'}{\rightsquigarrow}{}^{O_{xy}}_{xy}{}^{*}\ X'_{xy}$). 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 2). 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[2]). Note that proving symbolic preservation is almost trivial whenever both source semantics enjoy the same property (like our semantics 👻_{B}, 👻_{S}, and 👻_{R}).

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 (👻_{xy} is *SSS*). *If* $\vdash \text{👻}_{x}$ *SSS and* $\vdash \text{👻}_{y}$ *SSS and* $\vdash \text{👻}_{xy}$: *WFC then* $\vdash \text{👻}_{xy}$ *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* $\vdash \text{👻}_{xy}$: *WFC and* $p \nvdash_{x}$ *SNI or* $p \nvdash_{y}$ *SNI, then* $p \nvdash_{xy}$ *SNI.*
If $\vdash \text{👻}_{xy}$: *WFC and* $p \vdash_{xy}$ *SNI, then* $p \vdash_{x}$ *SNI and* $p \vdash_{y}$ *SNI.*

[2]For space reasons, Definition 3 only reports one direction (with a simplified notation).

These results have an immediate practical impact on Spectector: (1) Spectector's security analysis relies on the (symbolic) speculative semantics, (2) the source semantics 👻_{B}, 👻_{S}, and 👻_{R} are *SSS*, (3) well-formed compositions are also *SSS*, and (4) the composition of 👻_{B}, 👻_{S}, and 👻_{R} are well-formed. So, the Spectector security analysis equipped with any combination of the 👻_{B}, 👻_{S}, and 👻_{R} 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 👻_{B}, 👻_{S}, and 👻_{R}: $\text{👻}_{\mathsf{S+R}}$ (Section 5.1), $\text{👻}_{\mathsf{B+R}}$ (Section 5.2), $\text{👻}_{\mathsf{B+S}}$ (Section 5.3), and $\text{👻}_{\mathsf{B+S+R}}$ (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 $\text{👻}_{\mathsf{S+R}}$ 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 $\text{👻}_{\mathsf{S+R}}$ Composition

To combine semantics using our framework, we need to define the states, observations, and metaparameter $Z_{\mathsf{S+R}}$ for the composed semantics $\text{👻}_{\mathsf{S+R}}$. 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.

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

The union $Obs_{\mathsf{S+R}}$ of the trace models Obs_{S} and Obs_{R} is defined as:

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

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

Theorem 6 states the combination of 👻_{S} and 👻_{R} described above is well-formed. Given that 👻_{S} and 👻_{R} are *SSS* (Theorem 2 and Theorem 3), we can derive "for free" that $\text{👻}_{\mathsf{S+R}}$ is *SSS* (Theorem 4).

Theorem 6 ($\text{👻}_{\mathsf{S+R}}$ is well-formed). $\vdash \text{👻}_{\mathsf{S+R}}$: *WFC*

Listing 4 presents a program that contains a leak that can be detected only by $\text{👻}_{\mathsf{S+R}}$ but not by its components 👻_{S} and 👻_{R}.

Listing 4: $\text{👻}_{\mathsf{S+R}}$ example

```
Manip_Stack:
    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
```

In Listing 4, 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 👻$_{\mathsf{R}}$, 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 👻$_{\mathsf{S}}$, speculation over store bypasses has no effect in Listing 4 because the **store** instruction in Line 6 is never reached and function *Manip_Stack* returns to Line 13. Therefore, the leak is missed under 👻$_{\mathsf{S}}$ and 👻$_{\mathsf{R}}$, i.e., Listing 4 $\vdash_{\mathsf{S}}$ SNI and Listing 4 $\vdash_{\mathsf{R}}$ SNI.

However, under the combined semantics 👻$_{\mathsf{S+R}}$, 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 4 $\nvdash_{\mathsf{S+R}}$ SNI.

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

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 👻$_{\mathsf{S}}$ and 👻$_{\mathsf{R}}$ *and* additional programs like Listing 4. These additional programs are the reason why the semantics 👻$_{\mathsf{S+R}}$ is "stronger than the sum of its parts" 👻$_{\mathsf{S}}$ and 👻$_{\mathsf{R}}$.

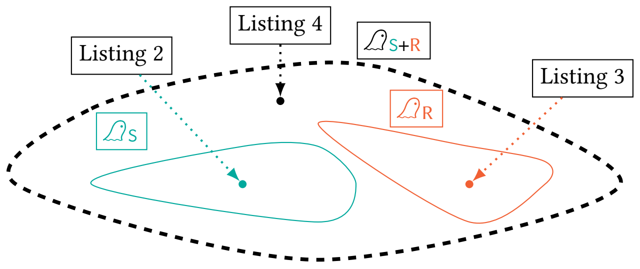


Figure 1: Relating 👻$_{\mathsf{S}}$, 👻$_{\mathsf{R}}$ and 👻$_{\mathsf{S+R}}$ wrt SNI.

5.2 👻$_{\mathsf{B+R}}$ Composition

In this combination, the instructions that influence speculative execution are **call** and **ret** (👻$_{\mathsf{R}}$) and **beqz** (👻$_{\mathsf{B}}$). Thus, we set $Z_{\mathsf{B+R}} =$ (**call** ∪ **ret**, **beqz**) to account for this and to allow speculation from both sources.

Theorem 7 states that 👻$_{\mathsf{B+R}}$ is well-formed. This allows us to derive "for free" that 👻$_{\mathsf{B+R}}$ is *SSS* by applying Theorem 4.

Theorem 7 (👻$_{\mathsf{B+R}}$ is well-formed). $\vdash$ 👻$_{\mathsf{B+R}}$: *WFC*

Listing 5: 👻$_{\mathsf{B+R}}$ example

```
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, secret
9  L2:
10     ret
11 Main:
12     call Speculate
13     skip
```

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

Under the combined semantics 👻$_{\mathsf{B+R}}$, function *Manip_Stack* returns to Line 6 and the **beqz** instruction is then mispredicted. This leads to executing the **load** instructions in Line 8, which leaks secret information. The resulting traces (🐓) 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 $\texttt{start}$ed for the leak to appear.

$$\tau^2_{\mathsf{B+R}} \neq \tau^1_{\mathsf{B+R}} \overset{\text{def}}{=} \texttt{call}\ \mathit{Speculate} \cdots \texttt{start}_{\mathsf{R}}\ 0 \cdots \texttt{start}_{\mathsf{B}}\ 1 \cdot \texttt{pc}\ 8 \cdot \boxed{\texttt{load}\ \mathit{secret}} \cdot \texttt{rlb}_{\mathsf{B}}\ 1 \cdot \texttt{rlb}_{\mathsf{R}}\ 0$$

Again, the two traces differ in the observation in the grey box and we have Listing 5 $\nvdash_{\mathsf{B+R}}$ SNI.

5.3 👻$_{\mathsf{B+S}}$ Composition

In this combination,speculation happens on **beqz** instructions (👻$_{\mathsf{B}}$) and on **store** instructions (👻$_{\mathsf{S}}$). Thus, we set $Z_{\mathsf{B+S}} =$ (**store**, **beqz**). Therefore, in the combined semantics 👻$_{\mathsf{B+S}}$, **beqz** instructions are only executed by delegating back to 👻$^{\textbf{store}}_{\mathsf{B}}$ and **store** instructions are only executed by delegating back to 👻$^{\textbf{beqz}}_{\mathsf{S}}$. This semantics is also a well-formed composition (Theorem 8) and *SSS*.

Theorem 8 (👻$_{\mathsf{B+S}}$ is well-formed). $\vdash$ 👻$_{\mathsf{B+S}}$: *WFC*

Listing 1 from Section 1 contains a leak that can only be detected by $\mathcal{S}_{B+S}$ but not by its components. The traces associated with the code (🐦) are given below, where secret-dependent observations are highlighted in gray:

$$\overline{\tau}^2_{B+S} \neq \overline{\tau}^1_{B+S} \overset{\text{def}}{=} \cdots \texttt{start}_S\ 1 \cdot \texttt{bypass}\ 1 \cdot \cdots \cdot \texttt{start}_B\ 2 \cdot \texttt{pc}\ 5$$
$$\cdot\ \texttt{load}\ p \cdot \boxed{\texttt{load}\ A + secret} \cdot \texttt{rlb}_B\ 2 \cdot \texttt{rlb}_S\ 1 \cdots$$

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

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

We conclude this section by combining all three semantics $\mathcal{S}_B$, $\mathcal{S}_S$, and $\mathcal{S}_R$. Our framework (Section 4) allows only combining a pair of source semantics into a combined one. For simplicity, we present $\mathcal{S}_{B+S+R}$ as a direct combination of the three source semantics (technically, we obtain $\mathcal{S}_{B+S+R}$ by combining $\mathcal{S}_{B+S}$ with $\mathcal{S}_R$). The metaparameter Z_{B+S+R} (which we represent as a triple of values) is (**call** ∪ **ret** ∪ **store**, **call** ∪ **ret** ∪ **beqz**, **beqz** ∪ **store**). As a result, the combined semantics $\mathcal{S}_{B+S+R}$ can only delegate to the corresponding speculative semantics for the appropriate speculation sources.

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

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

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

```
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 $\mathcal{S}_{B+S+R}$, since the program satisfies SNI under $\mathcal{S}_B$, $\mathcal{S}_S$ and $\mathcal{S}_R$. Under $\mathcal{S}_{B+S+R}$, the **store** instruction in Line 14 is bypassed There-fore, when returning from *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 (🐦) are given below:

$$\tau^2_{B+S+R} \neq \tau^1_{B+S+R} \overset{\text{def}}{=} \cdots \texttt{start}_S\ 1 \cdot \texttt{bypass}\ 14 \cdot \texttt{call}\ Speculate \cdots$$
$$\cdot\ \texttt{start}_R\ 2 \cdot \texttt{ret}\ 6 \cdot \texttt{start}_B\ 3 \cdot \texttt{pc}\ 8 \cdot \texttt{load}\ p$$
$$\cdot\ \boxed{\texttt{load}\ secret} \cdot \texttt{rlb}_B\ 3 \cdot \texttt{rlb}_R\ 2 \cdot \cdot \texttt{rlb}_S\ 1 \cdots$$

Thus, the program is not secure, i.e., Listing 6 $\nvdash_{B+S+R}$ 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 $\mathcal{S}_B$, with the semantics for $\mathcal{S}_S$ and $\mathcal{S}_R$ 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 $\mathcal{S}_S$ and $\mathcal{S}_R$ 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 $\mathcal{S}_S$ and $\mathcal{S}_R$ plus all compositions from Section 5) as an extension of Spectector [21]. The implementation of compositions closely fol-lows the structure of our framework. As in Section 5, selecting one of the composed semantics in Spectector sets the metapa-rameter 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 $\mathcal{S}_x$. 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 μAsm. 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, **store**/**load**, and **ret** instruc-tions (and their combinations):

- **Spectre-STL:** 13 programs are variants of the Spectre-STL vul-nerability. 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[3]. For each program, we also analyze manually patched versions obtained by (1) inserting lfences after call instructions (i.e., at the instruction address where **ret** speculatively returns), and (2) using the modified `retpoline` defense proposed in [27, Section 6.1].
- **Spectre-Comb:** 4 programs contain leaks that arise from com-bining 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 `lfence` instructions prevent the speculative leaks.

[3]Out of the three Spectre-RSB examples from `safeside` [1], we analyze the only one that works against an acyclic RSB like the one supported by $\mathcal{S}_R$. Programs *ca_ip*, *ca_oop*, 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.

Test case		👻$_{S}$	
		None	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 👻$_{S}$ semantics against unpatched programs (column "None") and programs patched with `lfence` (column "Fence")

Test case		👻$_{R}$		
		None	Fence	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 👻$_{R}$ semantics against unpatched programs (column "None"), programs patched with `lfence` (column "Fence"), and programs patched with the modified `retpoline` defense proposed in [27, §6.1] (column "Retpoline")

Figure 2: Result of the analysis of our benchmarks for 👻$_{S}$ and 👻$_{R}$. 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.

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 `lfence`/`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 👻$_{S}$ 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 [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]), 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.

Spectre-RSB: Figure 2b reports the analysis results on the **Spectre-RSB** programs. Using 👻$_{R}$, 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 [27], which replaces return instructions with a construct that traps the speculation in an infinite loop.

[4]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.

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. Spectector equipped with the single semantics 👻$_{B}$, 👻$_{S}$, and 👻$_{R}$ 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. Spectector can successfully identify leaks in listing 1, listing 5, listing 4 when using, respectively, the semantics 👻$_{B+S}$, 👻$_{S+R}$, and 👻$_{B+R}$. 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, Spectector is able to successfully detect leaks in all programs when using the 👻$_{B+S+R}$ semantics that combines all speculation mechanisms studied in this paper.

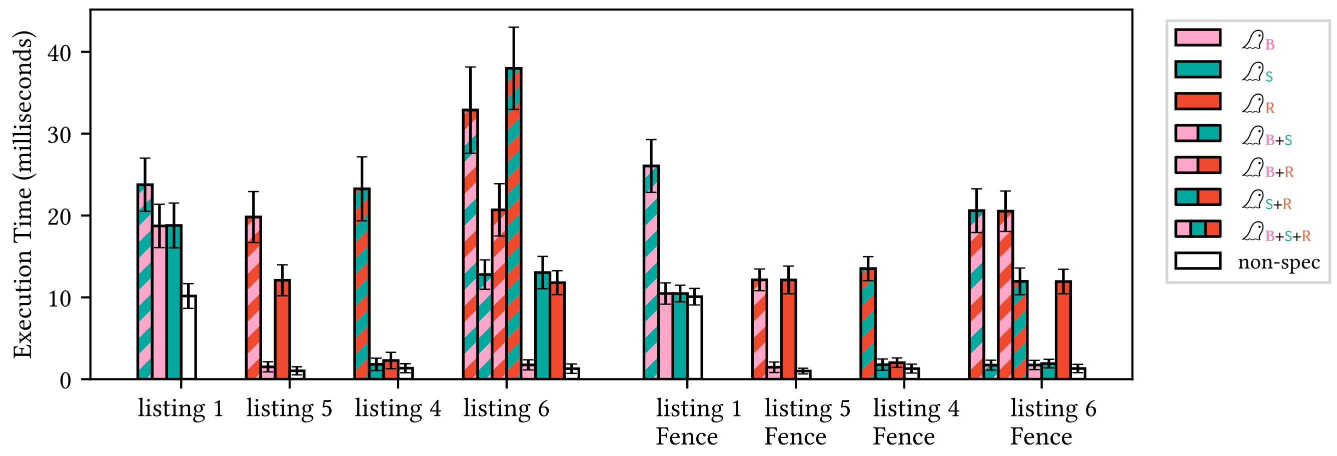
We also analyzed programs manually patched with `lfence` statements ("listing 1 Fence", "listing 5 Fence", "listing 4 Fence", and "listing 6 Fence" in Figure 3a). As before, Spectector successfully prove the security of patched programs. Even for leaks that arise from multiple speculation mechanisms,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

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

- For the programs patched with `lfence`, Spectector's execution time under a combined semantics is larger than Spectector'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 `lfence`s influence execution time. For instance, the execution time for "listing 5 Fence" under 👻$_{R}$ and

Test case		👻$_{\text{B}}$	👻$_{\text{S}}$	👻$_{\text{R}}$	👻$_{\text{B+S}}$	👻$_{\text{S+R}}$	👻$_{\text{B+R}}$	👻$_{\text{B+S+R}}$
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. 👻$_{\text{B+S+R}}$.



(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 `lfence`) of "listing x".

👻$_{\text{B+R}}$ is similar because the `lfence` is placed just after the branch instruction of Line 7, thereby stopping 👻$_{\text{B}}$-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 👻$_{\text{S+R}}$ is larger than its execution time for 👻$_{\text{B+S+R}}$. This is due to Spectector's terminating early after finding a violation of SNI, which happens under 👻$_{\text{B+S+R}}$ but not under 👻$_{\text{S+R}}$ (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 👻$_{\text{B}}$, 👻$_{\text{S}}$, and 👻$_{\text{R}}$:

- Speculation over indirect jumps [24] can be modeled as an always-mispredict semantics (similarly to 👻$_{\text{B}}$) 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 R:AM-Ret-Spec rule in 👻$_{\text{R}}$.
- 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, $\mathcal{J}_{S+R}$ 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 perform speculation-relevant changes in both source semantics. For instance, consider a combination of $\mathcal{J}_{R}$ 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, speculative 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 $\mathcal{J}_{S}$ and $\mathcal{J}_{R}$ capture the effects of transient instructions at a rather high-level, and they are inspired by the always-mispredict $\mathcal{J}_{B}$ semantics from [21]. Our $\mathcal{J}_{S}$ 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, KLEESpectre [39] and SpecuSym [22] consider a semantics that explicitly model the cache, which enable reasoning about the cache content. McIlroy 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 $\mathcal{J}_{S}$ 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 $\mathcal{J}_{B+S}$). 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 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 $\mathcal{J}_{S}$ and $\mathcal{J}_{R}$ 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 University 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.

REFERENCES

[1] 2019. SafeSide. https://github.com/google/safeside

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

[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] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] Xaver Fabian, Marco Guarnieri, and Marco Patrignani. 2022. https://github.com/XFabian/Spectector-Combined

[17] Xaver Fabian, Marco Guarnieri, and Marco Patrignani. 2022. https://github.com/XFabian/Spectecoq

[18] Xaver Fabian, Marco Guarnieri, and Marco Patrignani. 2022. Automatic Detection of Speculative Execution Combinations. (2022). arXiv:2209.0117

[19] 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.

[20] 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.

[21] 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)*.

[22] 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.

[23] 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.

[24] 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)*.

[25] 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.

[26] 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.

[27] 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.

[28] 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

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

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

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

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

[38] 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).

[39] 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).

[40] 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).

[41] 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.

[42] 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.