Automatic Detection of Speculative Execution Combinations

Anonymous Author(s)

ABSTRACT

Modern processors employ different prediction mechanisms to speculate over different kinds of instructions. Attackers can exploit these prediction mechanisms *simultaneously* in order to trigger leaks about speculatively-accessed data. Thus, sound reasoning about such speculative leaks requires accounting for *all* potential mechanisms of speculation. Unfortunately, existing formal models only support reasoning about fixed, hard-coded mechanisms of speculation, 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 branches. Our framework yields speculative semantics for speculating over any combination of those 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 existing codebases that are vulnerable to Spectre v1, Spectre v4, and Spectre v5 vulnerabilities as well as new snippets that are only vulnerable to their compositions.

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 [21], attackers can exploit these side effects to leak information about speculatively accessed data.

Modern general-purpose processors have different prediction mechanisms (branch predictors, memory disambiguators, etc.) that are used to speculate over different kinds of instructions: conditional branching [21], indirect jumps [21], store and load operations [20], and return instructions [22]. While well-known attacks target only a single speculation mechanism (e.g., Spectre-PHT [21] targets branch predictors), some speculative leaks only arise due to the interaction of multiple speculation mechanisms.

Listing 1: Speculative leak arising from speculation over branch and store instructions combined.

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

For example, the code in Listing 1 can speculatively leak the value of &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 speculation over the branch predictor and the memory disambiguator. 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 sources of speculative execution. However, existing formal models (also called *speculative semantics*) support multiple sources of speculation poorly. Some of them support only fixed speculation mechanisms: branch predictors [17, 18, 30–32] and (in addition) memory disambiguators [9, 14, 26]. Furthermore, the different speculation mechanisms are *hard-coded* into the formal semantics [9, 14, 17]. 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 simply 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 which can be used to reason about leaks involving the different kinds of speculation that compose it (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 can be derived automatically from the security of its sub-parts, maximising *proof reuse*. Finally, the composed semantics can be easily implemented in Spectector, which can then be used to *verify the absence of speculative leaks* such as those of Listing 1.

Concretely, this paper makes the following contributions:

- it introduces $\lightning_{\mathsf{S}}$ and $\lightning_{\mathsf{R}}$, 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 $\lightning_{\mathsf{S}}$, $\lightning_{\mathsf{R}}$ and $\lightning_{\mathsf{B}}$, the semantics for speculation over branch instructions [17], creating all the possible compositions ($\lightning_{\mathsf{B+S}}$, $\lightning_{\mathsf{S+R}}$, $\lightning_{\mathsf{B+R}}$, and $\lightning_{\mathsf{B+S+R}}$) and proving their security (Section 5).
  All the presented semantics are formalised in the Coq Proof assistant, and we write 🐓 to indicate that the semantics of some specific snippet is 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 7) and conclusions (Section 8).

For space constraints, formal details of the semantics, auxiliary lemmas and proofs can be found in the companion technical report. Our code extension to Spectector will be open sourced.

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 $\mathcal{J}_B$, the semantics for speculation over branch instructions (Section 2.4). Most of the notions that we overview next are taken from Guarnieri et al. [17].

2.1 Attacker Model and Security Definition

We adopt a commonly-used attacker model [3, 9, 14, 16–18, 25, 30]: 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.

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

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, $\texttt{pc}$, $\texttt{call}\ f$, 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* [17] (with identifier n) produced by the speculative semantics x (we often use x to range over any of 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* $\upharpoonright_{ns}$ [17] 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.

With this trace model we can define the security property we use in this paper: *Speculative Non-Interference* (SNI) [17]. 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 and in the speculative semantics x it uses. The policy ϕ describes the public/low information of the program. We use the same policy ϕ as described by Guarnieri et al. [17]: a list describing 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 the (speculative) traces describing the program behaviour are generated. We indicate that program p generates trace $\overline{\tau}$ from state σ with semantic x as $Beh_x^{\mathcal{A}}(p,\sigma)$). We fix the maximal speculation window, i.e., the maximum number of speculative instructions, to a global constant ω. We formalise several speculative semantics in later sections.

A program p satisfies SNI (Definition 1) for a speculative semantics x iff any pair of low-equivalent initial configurations σ^1 and σ^2 for program p that generate the same observations without speculation events, then the two configurations generate the same observations considering speculation events too.

Definition 1 (SNI). *$p \vdash_x SNI \stackrel{\text{def}}{=} \forall\sigma^1,\sigma^2$ if $\sigma^1\backsim_\phi\sigma^2$ and $Beh_x^{\mathcal{A}}(p,\sigma^1)$) and $Beh_x^{\mathcal{A}}(p,\sigma^2)$) and $\overline{\tau}^1\upharpoonright_{ns}=\overline{\tau}^2\upharpoonright_{ns}$ then $\overline{\tau}^1=\overline{\tau}^2$.*

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. μAsm programs p are sequences of mappings from natural numbers n (i.e., the instruction address) to instructions i or $\bot$. Instructions include skipping, register assignments, loads, store, indirect jumps, branching, conditional assignments, speculation barriers, calls, and returns. Instructions can contain expressions and values. The former come from the set *Regs*, containing register identifiers and designated registers **pc** and **sp** modelling the program counter and stack pointer respectively, while the latter come from the set *Vals*, which includes natural numbers and $\bot$.

2.3 Non-speculative Semantics of μAsm

μAsm has a small-step operational non-speculative semantics that describes how its programs execute. This semantics judgement is $(p,\sigma)\xrightarrow{\tau}(p,\sigma')$ 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 σ is used to read from/write to the register file a and the memory m. Memories map addresses (which are natural numbers) to values while registers map registers id to values.

Most of the rules of the semantics are standard and thus omitted, we present selected rules below. The rules rely on the evaluation of expressions (indicated as $[\![e]\!](a) = v$) where expression e is evaluated to value n under register file a. In the rules, $a[x\mapsto y]$, where $x\in Regs\cup\mathbb{N}$ and $y\in Vals$, denotes the update of a map (memory or registers), whereas $a(x)$ denotes reading from a map. Finally, $\sigma(x)$, where $x\in Regs$ and $\sigma=\langle m,a\rangle$, denotes $a(x)$.

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

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

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

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

Conditionals 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* $Beh_{NS}(p)$ 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 [17], 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. As before, we can introduce the *symbolic non-speculative behavior* $Beh^{\alpha}_{NS}(p)$ of a program p, and $\mu(Beh^{\alpha}_{NS}(p))$ is the set of all concrete traces derived from symbolically executing p. As proved by Guarnieri et al. [17], $Beh_{NS}(p) = \mu(Beh^{\alpha}_{NS}(p))$.

2.4 $\lightning_{\mathrm{B}}$: Speculating Over Branch Instructions

To model and reason about the effects of speculation induced by the branch predictor, Guarnieri et al. [17] propose the three 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 one, which depends on a prediction oracle that models the branch predictor. Finally, the symbolic semantics, which is used in Spectector, is the symbolic version of the always-mispredict semantics. We summarize the properties of these three semantics in Section 2.4.4. With a slight abuse of notation, we use $\lightning_{\mathrm{B}}$ to indicate 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 [17].

The state Σ_{B} of the AM semantics is 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).

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

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

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

$$\frac{p(\sigma(\mathbf{pc})) \notin [\mathbf{beqz}\ x, \ell; \boxed{\mathrm{Z}}] \qquad \sigma \xrightarrow{\tau} \sigma' \quad \text{(B:AM-NoSpec)}}{\langle p, ctr, \sigma, n+1\rangle \overset{\tau}{\rightsquigarrow}_{\mathrm{B}} \langle p, ctr, \sigma', n\rangle}$$

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

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 also contains a novel element, the parameter Z (in gray), which indicates other instructions that are skipped. Z is part of our composition framework and we defer to Section 4.1 an in-depth explanation of its role.

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

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 [17] for the full definition. First, speculative instances are extended to track the branching history h. Second, when executing a **beqz** instruction, the oracle is used to predict the branch outcome and to create a new speculative instance that 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. Thus, committing and rolling back speculations happen along the stack. Rolling back deletes all the instances above the rolled back instance, while committing updates the configuration, the counter and the branching history h of the instance below and the committed instance is deleted.

Similarly to before, the behaviour $Beh^{O}_{\mathrm{B}}(p)$ of a program p under the oracle semantics is the set of all traces generated from an initial state until termination.

2.4.3 Symbolic Semantics. The symbolic speculative semantics $\lambda^{\mathcal{S}}_{\mathrm{B}}$ works on symbolic speculative states $\Sigma^{\mathcal{S}}_{\mathrm{B}}$ and is used in the implementation of Spectector [17]. The only two differences w.r.t. the AM semantics are that (1) concrete states Σ_{B} are replaced with symbolic states $\Sigma^{\mathcal{S}}_{\mathrm{B}}$, 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 $Beh^{\mathcal{S}}_{\mathrm{B}}(p)$ of a program p is defined as for the AM semantics.

2.4.4 Properties of λ_{B}. Guarnieri et al. [17] 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 we present in this paper. In the definition we indicate that a program p satisfies SNI w.r.t. the oracle semantics as $p \vdash^{O}_{\mathrm{B}}$ SNI.

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

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

Intuitively, a secure speculative semantics is made of three components, an AM, an oracle and a symbolic semantics. First, the AM semantics must overapproximate the oracle semantics, so it is enough to check a program p for SNI w.r.t. the AM semantics [17, Theorem 1]. Then, since Spectector uses the symbolic semantics in the implementation, the symbolic semantics must be faithful w.r.t. the AM one [17, 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 [17, Propositions 1,3]. So we can execute p only once to get the (non-)speculative behaviour of that program run.

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

Theorem 1 (λ_{B} is *SSS* [17]). $\vdash \lambda_{\mathrm{B}}$ *SSS*

3 SPECULATION ON STORES AND RETURNS

This section defines λ_{S} and λ_{R}, two novel speculative semantics that model the effects of speculative execution over store (Section 3.1) and return instructions (Section 3.2). Similarly to λ_{B}, for each speculation source 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.

3.1 λ_{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.

```
1 p = &secret;
2 p = &public;
3 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 (λ_{S} is *SSS*). $\vdash \lambda_{\mathrm{S}}$ *SSS*

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

$$Obs_{\mathrm{S}} ::= Obs \mid \mathtt{start}_{\mathrm{S}}\ n \mid \mathtt{rlb}_{\mathrm{S}}\ n \mid \mathtt{skip}\ n$$

3.1.2 Speculative Semantics. The overall structure of the λ_{S} semantics is similar to that of λ_{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 λ_{S} and λ_{B} is the differing source of speculation: branches for λ_{B} and stores for λ_{S}.

The states used for the speculative semantics of λ_{S} are similar to the states of λ_{B}:

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

Judgement $\Sigma_{\mathrm{S}} \overset{\tau}{\dashrightarrow}_{\lambda_{\mathrm{S}}} \Sigma'_{\mathrm{S}}$ describes how program state Σ_{S} steps to Σ'_{S} emitting observation τ. Similar to λ_{B}, reductions only happen on top of the stack.

(S:AM-Store)

$$\frac{p(\sigma(\mathbf{pc})) = \mathbf{store}\ x, e \qquad (p,\sigma) \xrightarrow{\tau} (p,\sigma') \qquad j = min(\omega, n) \\ \sigma'' = \sigma[\mathbf{pc} \mapsto \sigma(\mathbf{pc}) + 1] \qquad \tau' = \tau \cdot \mathtt{skip}\ \sigma(\mathbf{pc}) \cdot \mathtt{start}\ ctr}{\langle p, ctr, \sigma, n+1\rangle \overset{\tau'}{\rightsquigarrow}_{\mathsf{S}} \langle p, ctr, \sigma', n\rangle \cdot \langle p, ctr+1, \sigma'', j\rangle}$$

To model the effect of bypassing a **store** instruction, Rule S:AM-Store skips 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 skipped **store** instruction loads a stale value.

Similarly to $\rightsquigarrow_{\mathsf{B}}$, all instructions that are not **store** instructions are handled by delegating back to the non-speculative semantics and when the speculation window reaches 0, a roll back occurs that pops the topmost speculative instance from the stack.

The behaviour $Beh_{\mathsf{S}}^{\mathcal{A}}(p)$ is the set of all traces that are generated from an initial state until termination using the reflexive-transitive closure of $\overset{\tau}{\rightsquigarrow}_{\mathsf{S}}$.

3.1.3 Oracle Semantics. Instead of skipping every **store** instruction speculatively, the oracle semantics employs an oracle O that decides if the **store** instruction should be skipped or not. Similar to before, the behaviour $Beh_{\mathsf{S}}^{O}(p)$ of a program p is the set of all traces starting from an initial state until termination using the reflexive-transitive closure of the oracle semantics.

3.1.4 Symbolic Semantics. Similarly to $\rightsquigarrow_{\mathsf{B}}^{\mathcal{S}}$, the symbolic speculative semantics $\rightsquigarrow_{\mathsf{S}}^{\mathcal{S}}$ requires two changes w.r.t. the always-mispredict one: concrete configurations σ and the non-speculative semantics are replaced by symbolic configurations $\sigma^{\mathcal{S}}$ and the symbolic non-speculative semantics respectively. The behaviour $Beh_{\mathsf{S}}^{\mathcal{S}}(p)$ of a program p is the set of all traces starting from an initial state until termination using the reflexive-transitive closure of the symbolic semantics.

3.2 $\rightsquigarrow_{\mathsf{R}}$: 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 that 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 lead to speculative execution which 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* in line 2 the **sp** register is incremented. Thus, **sp** points to the previous return address on the stack, and the non-speculative execution continues in *Main* and terminates. However, because the return address of the call in line 5 is line 6 and is on top of the RSB, the CPU speculatively uses this return address and execution jumps to line 6; thus, the secret is leaked.

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 ($\rightsquigarrow_{\mathsf{R}}$ is *SSS*). $\vdash \rightsquigarrow_{\mathsf{R}}$ *SSS*

3.2.1 Speculative Semantics. Unlike before, the state of $\rightsquigarrow_{\mathsf{R}}$ contains a model of the RSB which is used to retrieve return addresses instead of relying on the stack. Thus, speculative instances of $\rightsquigarrow_{\mathsf{R}}$ 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, the small-step operational semantics $\Sigma_{\mathsf{R}} \overset{\tau}{\rightsquigarrow}_{\mathsf{R}} \Sigma_{\mathsf{R}}$, reductions happen at the top of the stack.

(R:AM-Ret-Spec)

$$\frac{p(\sigma(\mathbf{pc})) = \mathbf{ret} \qquad \sigma \xrightarrow{\tau} \sigma' \\ \mathbb{R} = \mathbb{R}' \cdot l \qquad j = min(\omega, n) \qquad l \neq m(a(\mathbf{sp})) \\ \sigma'' = \sigma[\mathbf{pc} \mapsto l, \mathbf{sp} \mapsto a(\mathbf{sp}) + 8] \qquad \overline{\tau} = \tau \cdot \mathtt{start}_{\mathsf{R}}\ ctr \cdot \mathtt{ret}\ l}{\langle p, ctr, \sigma, \mathbb{R}, n+1\rangle \overset{\overline{\tau}}{\rightsquigarrow}_{\mathsf{R}} \langle p, ctr, \sigma', \mathbb{R}', n\rangle \cdot \langle p, ctr+1, \sigma'', \mathbb{R}', j\rangle}$$

(R:AM-Call)

$$\frac{p(\sigma(\mathbf{pc})) = \mathbf{call}\ f \qquad \sigma \xrightarrow{\tau} \sigma' \\ \mathbb{R}' = \mathbb{R} \cdot (a(\mathbf{pc}) + 1) \qquad |\mathbb{R}| < \mathbb{R}_{size}}{\langle p, ctr, \sigma, \mathbb{R}, n+1\rangle \overset{\tau}{\rightsquigarrow}_{\mathsf{R}} \langle p, ctr, \sigma', \mathbb{R}', n\rangle}$$

During **call** instructions (Rule R:AM-Call), the return address is pushed on top of the RSB (if there is space available) and during **ret** instructions, the return address stored on the RSB is used if the entry on top of the RSB is different from the one stored on the stack (Rule R:AM-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 an empty/full RSB; full formalization is available in the technical report. Intuitively, 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 a so-called *acyclic* RSB.[1]

The behaviour $Beh^{\mathcal{A}}_{\mathsf{R}}(p)$ is is the set of all traces generated from an initial state until termination using the reflexive-transitive closure of $\overset{\tau}{\multimap}_{\mathsf{R}}$.

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 $\multimap^{\mathcal{S}}_{\mathsf{R}}$ 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 [17]. As a result, the RSB only contains concrete values (and return addresses). The behaviour $Beh^{\mathcal{S}}_{\mathsf{R}}(p)$ 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

3.2.4 Different Behaviours of Empty and Full RSBs. Modern CPUs use different RSBs implementations that differ in the way they handle under- and overflow, i.e., when the RSB is empty or full [23]. 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).

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

We remark that extending $\multimap_{\mathsf{R}}$ 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:

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

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

The program in Listing 1 seems secure since there is no secret value leaked in the speculative transaction; thus the program satisfies SNI for $\multimap_{\mathsf{S}}$ and $\multimap_{\mathsf{R}}$ in isolation. As we show later, the program in Listing 1 is not secure, but we need our combined semantics to point out this vulnerability, as we show in Section 5.3.

[1] We follow the way AMD processors handle this kind of speculation [23].

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 the CPU uses all the predictors described here (and many others as well) at the same time. Thus, we should not only focus on these different attacks in *isolation* but we need to look at their *combinations* as well. That is, we need is a way to compose the different semantics into new semantics that can detect these combined vulnerabilities.

This section presents a novel, general framework for composing two speculative semantics x and y to allow for speculation from both sources x and y. The speculative semantics x and y are also called the *source* semantics of the composition. Thus, this section first introduces the new composed semantics, which consists of the always-mispredict, the oracle and the symbolic semantics (Section 4.1). Finally, the section define what it means to be a *proper* composition by identifying key correctness properties (Section 4.2). From this definition, we can derive corollaries like that SNI of the combination implies SNI of both of its parts.

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

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

For example, the $\Phi_{\mathsf{S+R}}$ states resulting of 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 the common elements (the program p, the counter ctr, the state σ, and the speculation count n), plus the return stack buffer $\mathbb{R}$ that belongs to Φ_{R} only. Taking the $\cdot\restriction^{\mathsf{S}}_{\mathsf{S+R}}$ of a $\Phi_{\mathsf{S+R}}$ state returns the Φ_{S} subpart (i.e., all but the return stack buffer).

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

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

We indicate source semantics for x and y as $\multimap_x$ and $\multimap_y$ respectively and use $\multimap_{xy}$ to indicate their composed semantics.

4.1 Combined Speculative Semantics

The idea behind the combined semantics is to delegate back to the source semantics of x and y to allow for speculation on both sources. This, in turn, yields proof reuse and is encapsulated in the two core rules below:

$$\text{(AM-x-step)}\quad \frac{\Phi_{xy}\restriction^{x}_{xy} \xrightarrow{\tau}_{x} Z_{xy}\restriction^{x}_{xy}\ \overline{\Phi}'_{xy}\restriction^{x}_{xy}}{\Phi_{xy} \xrightarrow{\tau}_{xy} Z_{xy}\ \overline{\Phi}'_{xy}}$$

$$\text{(AM-y-step)}\quad \frac{\Phi_{xy}\restriction^{y}_{xy} \xrightarrow{\tau}_{y} Z_{xy}\restriction^{y}_{xy}\ \overline{\Phi}'_{xy}\restriction^{y}_{xy}}{\Phi_{xy} \xrightarrow{\tau}_{xy} Z_{xy}\ \overline{\Phi}'_{xy}}$$

The 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). Each rule relies on metaparameter Z_{xy}, which is a pair of two metaparameters $Z_{xy} := (Z_x, Z_y)$ — one for x and one for y. We overload the projections $\restriction^{x}_{xy}$ and $\restriction^{y}_{xy}$ to extract he corresponding metaparameter from Z_{xy}.

The role of Z is central to making the composed semantics work as expected, so now we explain Z in detail. If we were to remove metaparameter Z and combine $\mathcal{A}_{\mathsf{B}}$ and $\mathcal{A}_{\mathsf{S}}$ into $\mathcal{A}_{\mathsf{B+S}}$, consider the execution of the **beqz** instruction in Line 4 in Listing 1. Without Z, $\mathcal{A}_{\mathsf{B+S}}$ can use Rule AM-x-step and delegate back to $\mathcal{A}_{\mathsf{B}}$ for **beqz** instructions, creating a new speculative transaction. However, $\mathcal{A}_{\mathsf{B+S}}$ can also use Rule AM-y-step, because **beqz** instructions are also handled by $\mathcal{A}_{\mathsf{S}}$. Unfortunately, this would lead to no speculative transaction because speculation happens only on **store** instructions. Intuitively, $\mathcal{A}_{\mathsf{B+S}}$ should delegate back to $\mathcal{A}_{\mathsf{B}}$, so Rule AM-y-step should not be applicable.

With metaparameter Z, we can instantiate it for $\mathcal{A}_{\mathsf{B+S}}$ as $Z_{\mathsf{B+S}} = ([\textbf{store}], [\textbf{beqz}])$, so that its projections are $Z_{\mathsf{B}} = [\textbf{store}]$ and $Z_{\mathsf{S}} = [\textbf{beqz}]$. Now, $\mathcal{A}_{\mathsf{B+S}}$ can only apply Rule AM-x-step on the **beqz** of Line 4, because Z_{S} ensures that $\mathcal{A}_{\mathsf{S}}$ can not execute **beqz** instructions, as depicted in the full rule for $\mathcal{A}_{\mathsf{S}}$ below (where we indicate the instructions derived from Z in blue).

$$\text{(S:AM-NoSpec)}\quad \frac{p(\sigma(\mathbf{pc})) \notin [\textbf{store}\ x, e; \textbf{beqz}\ x, \ell] \qquad \sigma \xrightarrow{\tau} \sigma'}{\langle p, ctr, \sigma, n+1\rangle \xrightarrow{\tau}_{\mathsf{S}} \langle p, ctr, \sigma', n\rangle}$$

Having clarified the intuition behind the semantics, we can define the behaviour $Beh^{\mathcal{A}}_{xy}$ as the set of all traces generated from an initial state until termination using the reflexive-transitive closure of $\xrightarrow{\tau}_{xy}$.

4.1.1 Oracle Combination. Instead of using one oracle, the combination uses a pair of oracles, one from each source. 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 $\mathcal{A}^{\mathcal{S}}_{xy}$ 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 from an initial state until termination using the reflexive-transitive closure of 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*, then the combined semantics is also *SSS* (Section 4.2.2). Note that since we prove these properties in the framework, any well-formed composition is also *SSS for free*. This proof reuse and extensibility is the key advantage of our framework over having ad-hoc semantics combining multiple speculation sources, 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. For this, the well-formedness conditions attest that (1) the composed semantics is deterministic, and (2) the composed semantics does *not* hide observations produced by its components.

Point (3) is fairly technical, and to present it, we need to mention a technical detail: the state relation (denoted $\approx_{xy}$ and defined in our technical report) between the AM states (Σ_{xy}) and the Oracle ones (X_{xy}). Intuitively, two states are related if they are the same or if one is waiting on a speculation of the other to end. Then, point (3) ensures that whenever we start from related states ($\Sigma_{xy} \approx_{xy} X_{xy}$) and we do one or more steps of the AM composed semantics ($\Sigma_{xy} \xrightarrow{\overline{\tau}}{}^{*}_{xy} \Sigma'_{xy}$), then we can *always* find a related state ($\Sigma'_{xy} \approx_{xy} X'_{xy}$) that is reachable by performing one or more steps of the composed oracle semantics ($X_{xy} \xrightarrow{\overline{\tau}'}{}^{O_{xy}}_{xy}{}^{*} X'_{xy}$). This fact is used when proving that SNI of a program under the composed AM semantics implies SNI under the composed oracle semantics (point 1 of Definition 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.

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

1. *(Confluence) Whenever $\Sigma_{xy} \xrightarrow{\tau}_{xy} \Sigma'_{xy}$ and $\Sigma_{xy} \xrightarrow{\tau}_{xy} \Sigma''_{xy}$, then $\Sigma'_{xy} = \Sigma''_{xy}$.*
2. *(Projection preservation) For all programs p, $Beh^{\mathcal{A}}_{x}(p) = Beh^{\mathcal{A}}_{xy}(p)\restriction^{x}_{xy}$ and $Beh^{\mathcal{A}}_{y}(p) = Beh^{\mathcal{A}}_{xy}(p)\restriction^{y}_{xy}$.*
3. *(Relation preservation) If $\Sigma_{xy} \approx_{xy} X_{xy}$ and $\Sigma_{xy} \xrightarrow{\overline{\tau}}{}^{*}_{xy} \Sigma'_{xy}$ then $X_{xy} \xrightarrow{\overline{\tau}'}{}^{O_{xy}}_{xy}{}^{*} X'_{xy}$ and $\Sigma'_{xy} \approx_{xy} X'_{xy}$.*

Since the combined semantics delegates back to its sources, it becomes non-deterministic. Confluence is needed to ensure the non-determinism is not harmful. Consider the assignment in Line 1 in Listing 1. $\mathcal{A}_{\mathsf{B+S}}$ can delegate to either $\mathcal{A}_{\mathsf{B}}$ or $\mathcal{A}_{\mathsf{S}}$ to reduce the assignment. If the combined semantics is *Confluent* then it does not matter which source rule executes the assignment in Line 1 in Listing 1, the semantics reaches the same state either way.

The projection preservation condition, instead, 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*.

4.2.2 Free Theorems. The key result of our framework is that well-formed compositions whose source are secure speculative semantics (*SSS*) are also *SSS* (Theorem 4).

Theorem 4 ($\mathcal{L}_{xy}$ is *SSS*). *if* $\vdash \mathcal{L}_x$ *SSS and* $\vdash \mathcal{L}_y$ *SSS and* $\vdash \mathcal{L}_{xy}$: *WFC then* $\vdash \mathcal{L}_{xy}$ *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* $\vdash \mathcal{L}_{xy}$: *WFC and* $p \nvdash_x$ *SNI or* $p \nvdash_y$ *SNI, then* $p \nvdash_{xy}$ *SNI.*
If $\vdash \mathcal{L}_{xy}$: *WFC and* $p \vdash_{xy}$ *SNI, then* $p \vdash_x$ *SNI and* $p \vdash_y$ *SNI.*

Our free theorems have an immediate practical impact on Spectector. Note in fact that (1) Spectector's security analysis relies on the symbolic semantics, (2) the source symbolic semantics $\mathcal{L}_B$, $\mathcal{L}_S$, and $\mathcal{L}_R$ are *SSS*, (3) well-formed compositions are also *SSS*, and (4) the composition of $\mathcal{L}_B$, $\mathcal{L}_S$, and $\mathcal{L}_R$ are well-formed. So, the Spectector security analysis equipped with any combination of the $\mathcal{L}_B$, $\mathcal{L}_S$, and $\mathcal{L}_R$ 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, while the section thereafter describes their implementation in Spectector.

5 INSTANTIATING OUR FRAMEWORK

This section describes all possible combinations of the speculative semantics $\mathcal{L}_B$, $\mathcal{L}_S$, and $\mathcal{L}_R$: $\mathcal{L}_{S+R}$ (Section 5.1), $\mathcal{L}_{B+R}$ (Section 5.2), $\mathcal{L}_{B+S}$ (Section 5.3), and $\mathcal{L}_{B+S+R}$ (Section 5.4). For each of them, we overview how the combined always-mispredict semantics behave using concrete examples 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 $\mathcal{L}_{S+R}$ semantics can be instantiated as part of our framework; the other combinations can be instantiated similarly and we only provide a higher-level description. We remark that the traces associated with the code snippets in this section have been computed using our Coq executable composed semantics. Due to the length of these traces, here we elide uninteresting observations.

5.1 $\mathcal{L}_{S+R}$ Composition

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

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

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

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

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

Theorem 6 states that combining $\mathcal{L}_S$ and $\mathcal{L}_R$ in the way described above results in a well-formed composition. Given that $\mathcal{L}_S$ and $\mathcal{L}_R$ are both *SSS* (Theorem 2 and Theorem 3), we can derive "for free" that $\mathcal{L}_{S+R}$ is also *SSS* by applying Theorem 4.

Theorem 6 (Well-formed Composition $\mathcal{L}_{S+R}$). $\vdash \mathcal{L}_{S+R}$: *WFC*

Let us now informally argue why Theorem 6 holds. Intuitively, Confluence follows from the fact that both $\mathcal{L}_S$ and $\mathcal{L}_R$ delegate back to the non-speculative semantics for instructions not related to speculation. Since instructions related to speculation are already restricted by the metaparameter Z_{S+R}, we only need to show *Confluence* for those instructions not related to speculation. Because the non-speculative semantics is deterministic, we can derive *Confluence* for instructions not related to speculation.

For the Projection preservation, the composed semantics allows nesting of *different* speculative transactions i.e., the nesting of speculative transactions for **store** and **ret** instructions. Despite the nesting, these transactions are still fully explored — similar to what the corresponding source semantics would do.

Listing 4 presents a program that contains a leak that can be detected only by $\mathcal{L}_{S+R}$ but not by its components $\mathcal{L}_S$ and $\mathcal{L}_R$.

Listing 4: $\mathcal{L}_{S+R}$ example

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

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 $\mathcal{A}_R$, mispredicting the return address of *Manip_Stack* using the RSB leads 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 $\mathcal{A}_S$, 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 in missed under $\mathcal{A}_S$ and $\mathcal{A}_R$, i.e., Listing 4 $\vdash_S$ SNI and Listing 4 $\vdash_R$ SNI.

However, following the combined semantics $\mathcal{A}_{S+R}$, the **store** instruction on Line 7 is now skipped and when returning from function *Manip_Stack* 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, we can find two low-equivalent configurations σ^1, σ^2 that differ in the value of *secret*. This means we can find two traces ($\lightning$) that differs in the observation load *secret* (highlighted in gray). Thus, the program is not secure under the combined semantics i.e., Listing 4 $\not\vdash_{S+R}$ SNI.

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

The relation between the source semantics and their composition is visualised in Figure 1, which shows the insecure programs (w.r.t SNI) detected under the different semantics. The combined semantics encompasses all vulnerable programs of $\mathcal{A}_S$ and $\mathcal{A}_R$ *and* additional programs like Listing 4. These additional programs are the reason why the semantics $\mathcal{A}_{S+R}$ is 'stronger than the sum of its parts' $\mathcal{A}_S$ and $\mathcal{A}_R$.

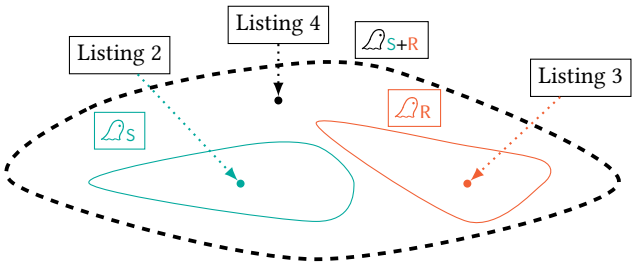

Figure 1: Relating $\mathcal{A}_S$, $\mathcal{A}_R$ and $\mathcal{A}_{S+R}$ wrt SNI.

5.2 $\mathcal{A}_{B+R}$ Composition

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

Theorem 7 states that $\mathcal{A}_{B+R}$ is a well-formed composition. As before, this allows us to derive "for free" that $\mathcal{A}_{B+R}$ is *SSS* by applying Theorem 4.

Theorem 7 (Well-formed Composition $\mathcal{A}_{B+R}$). $\vdash \mathcal{A}_{B+R}$: *WFC*

Listing 5: $\mathcal{A}_{B+R}$ 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 $\mathcal{A}_{B+R}$. The execution proceeds similarly to Listing 4 until the **ret** instruction in Line 3 is reached. Under $\mathcal{A}_R$, 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 $\mathcal{A}_B$, the branch 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 $\mathcal{A}_R$ and $\mathcal{A}_B$ in isolation.

Under the combined semantics $\mathcal{A}_{B+R}$, 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 ($\lightning$) are given below, where we highlight the secret-dependent observations. Given the length of the trace, we carve out only the most relevant parts, i.e., that both kinds of speculations need to have started in order for the leak to appear.

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

Again, the two traces that differ in the observation in the grey box and we have Listing 5 $\not\vdash_{B+R}$ SNI.

5.3 $\mathcal{A}_{B+S}$ Composition

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

Theorem 8 (Well-formed Composition $\mathcal{A}_{B+S}$). $\vdash \mathcal{A}_{B+S}$: *WFC*

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

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

Thus, the program is not secure under $\mathcal{A}_{B+S}$, i.e., Listing 1 $\not\vdash_{B+S}$ SNI.

5.4 λ_{B+S+R} Composition

We conclude this section by combining all three semantics λ_B, λ_S, and λ_R. Our framework (Section 4) allows only combining a pair of source semantics into a combined one. For simplicity, we present λ_{B+S+R} as a direct combination of the three source semantics (technically, we obtain λ_{B+S+R} by combining λ_{B+S} with λ_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 λ_{B+S+R} can only delegate to the corresponding speculative semantics for the appropriate speculation sources.

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

Theorem 9 (Well-formed Composition λ_{B+S+R}). $\vdash \lambda_{B+S+R} : WFC$

Listing 6: λ_{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 λ_{B+S+R}, since the program satisfies SNI under λ_B, λ_S and λ_R. Under λ_{B+S+R}, the **store** instruction in Line 14 is bypassed Therefore, 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 ($\lightning$) are given below:

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

Thus, the program is not secure i.e., Listing 6 $\not\sim_{B+S+R}$ 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, with the semantics for λ_S, λ_R and for all the combinations presented in 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 λ_S and λ_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 λ_S and λ_R plus all the compositions from Section 5) as an extension of Spectector [17]. Spectector uses symbolic execution together with self-composition [6] and a SMT solver to check for SNI w.r.t. λ_B. Due to this setup, we inherit limitations of Spectector as well, i.e., path explosion because of symbolic execution and limitations in the translation from x86 to μAsm.

6.2 Experiments

Benchmarks: We analyze 49 snippets of code containing leaks resulting from speculation over branch, store/load, and return instructions (and their combinations):

- **Spectre-STL:** 13 programs are variants of the Spectre-STL vulnerability. They exploit speculation over memory disambiguation, and they have been used as benchmarks in prior work [14, 26]. 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][2] and `transientfail` [8][3] projects. For each program, we also analyze manually patched versions obtained by (1) inserting lfences after each call instruction (i.e., at the instruction address where **ret** speculatively returns), and (2) using the `retpoline` defense [23].
- **Spectre-Comb:** 4 programs contain leaks that arise from combining speculation mechanisms. These are the programs depicted in listing 1, listing 5, listing 4, and listing 6 and discussed in Section 5. For each program, we also analyze a manually patched version where `lfence` instructions prevent the speculative leaks.

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 formalized 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: Table 1a reports the results of analysing the programs in the **Spectre-STL** benchmark[4]. Using the λ_S semantics, Spectector successfully detected leaks (i.e., violations of SNI) in

[2] Out of the three Spectre-RSB examples from [1], we analyze the only one that works against an acyclic RSB like the one supported by λ_R.
[3] Programs *ca_ip*, *ca_oop*, and *sa_ip* from `transientfail` rely on concurrent execution. Since Spectector does not support concurrency, we hardcode the worst-case interleaving in terms of speculative leakage in our benchmark.
[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.

Test case		λ_S None	λ_S Fence
case01	(-)	○	●
case02	(-)	○	●
case03	(+)	●	●
case04	(-)	○	●
case05	(-)	○	●
case06	(-)	○	●
case07	(-)	○	●
case08	(-)	○	●
case09	(+)	●	●
case10	(-)	○	●
case11	(-)	○	●
case12	(+)	●	●
case13	(-)	○	●

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

Test case		λ_R None	λ_R Fence	λ_R Retpoline
ret2spec_c_d	(-)	○	●	●
ca_ip	(-)	○	●	●
ca_oop	(-)	○	●	●
sa_ip	(-)	○	●	●
sa_oop	(-)	○	●	●

(b) Results for the Spectre-RSB programs under the λ_R semantics against unpatched programs (column "None"), programs patched with `lfence` (column "Fence"), and programs patched with retpoline (column "Retpoline")

Test case		λ_B	λ_S	λ_R	λ_{B+S}	λ_{S+R}	λ_{B+R}	λ_{B+S+R}
listing 1	(-)	●	●	●	○	●	●	○
listing 5	(-)	●	●	●	●	○	●	○
listing 4	(-)	●	●	●	●	●	○	○
listing 6	(-)	●	●	●	●	●	●	○
listing 1 Fence	(+)	●	●	●	●	●	●	●
listing 5 Fence	(+)	●	●	●	●	●	●	●
listing 4 Fence	(+)	●	●	●	●	●	●	●
listing 6 Fence	(+)	●	●	●	●	●	●	●

(c) Results for the Spectre-Comb programs where "listing x Fence" denotes the patched version (using `lfence`) of "listing x"

Table 1: Result of the analysis of our benchmarks. For each program, ○ denotes that our tool finds a violation of SNI w.r.t to the corresponding semantics, whereas ● denotes that our tool proves the program secure under the semantics. Next to each program, we report whether the program contains a speculative leak (+) or not (-) in its unpatched version.

all unpatched programs, except programs 03, 09, and 12 which do not contain speculative leaks (consistently with other analysis results [14, 26]). 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 [26]), 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: Table 1b reports the analysis results on the **Spectre-RSB** programs. Using λ_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 [23], which replaces return instructions with a construct that traps the speculation in an infinite loop until speculation ends.

Spectre-Comb: Table 1c 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 λ_B, λ_S, and λ_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 λ_{B+S}, λ_{S+R}, and λ_{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 λ_{B+S+R} semantics that combines all speculation mechanisms studied in this paper.

Similarly to **Spectre-STL** and **Spectre-RSB**, we also analyzed programs that have been manually patched by inserting `lfence` statements ("listing 1 Fence", "listing 5 Fence", "listing 4 Fence", and "listing 6 Fence" in Table 1c). As before, Spectector successfully prove the security of patched programs. We remark that, even for leaks that arise from multiple speculation sources, it is often sufficient to insert a single `lfence` to secure the entire program. For instance, it is sufficient to introduce a `lfence` after the **beqz** instruction in Listing 5 to ensure that the program satisfies SNI with respect to λ_{B+S+R}.

7 RELATED WORK

Speculative execution attacks: After Spectre [21] has been disclosed to the in 2018, researchers have identified many other speculative execution attacks [4, 7, 22, 23, 33]. These attacks differ in the exploited speculation sources [20, 22, 23], the covert channels [27–29] 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 that capture different flavors of security against speculative leaks. These properties 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, 30]) extends the constant-time security condition to account also for transient instructions.

(2) Relative non-interference definitions [11, 16, 17, 19] ensure that transient instructions do not leak more information than what is leaked by non-transient instructions. For instance, speculative non-interference [17], which we build on, (used also in [18, 25]) restricts the amount of information that can be leaked by speculatively executed instructions (without constraining what can be leaked non-speculatively).

(3) Definitions that formalise security as a safety property [25, 26], which often over-approximate the non-interference definitions from the categories 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 $\lightning_S$ and $\lightning_R$ capture the effects of transient instructions at a rather high-level, and they are inspired by the always-mispredict $\lightning_B$ semantics from [17]. In contrast, other approaches, which we overview next, consider more complex models that explicitly model microarchitectural components like multiple pipeline stages, caches, and branch predictors.

For instance, KLEESpectre [31] and SpecuSym [19] consider a semantics that explicitly model the cache, which enable reasoning about the cache content. McIlroy et al. [24] 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 high-level semantics, their 3-stage pipeline semantics explicitly models several microarchitectural components like a reorder buffer and an RSB. Their tool uses symbolic execution to detects violations of speculative constant-time induced by speculation over branch instructions and store-bypasses.

Binsec/Haunted [14] can also detect violations of speculative constant-time induced by speculation over store-bypasses and branch instructions. For this, their semantics explicitly model the store buffer, which $\lightning_S$ abstracts away. Barthe et al. [5] extend the Jasmin [3] cryptographic verification framework to reason about speculative constant-time and, again, it supports speculation over store-bypasses and branch instructions.

While several of the models describe above support multiple speculation mechanisms, these mechanisms are *hard-coded* and none of the aforementioned approaches provide a composition framework like ours (or extensible ways of extending the main theoretical results to new mechanisms "for free"). Moreover, while we could have used other operational semantics, like the one in [9], as a basis for our composition 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 formalize 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 speculation over branch instructions but they both lack program analyses.

Ponce de León and Kinder [26] illustrate how one can model leaks resulting from speculation over branch instructions and load-/store instructions 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 [26, IV E.], which in theory should allow them to detect vulnerabilities like Listing 1 (which we detect under $\lightning_{B+S}$). Differently from our framework, 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 a main advantage of axiomatic models, our framework (and tool implementation) clearly show that composability can be done with operational semantics as well.

8 CONCLUSION AND FUTURE WORK

This paper presented new speculative semantics for speculation on store instructions and on return instructions. Furthermore, it defined a general framework to reason about the composition of different speculative semantics and instantiated the framework with our new speculative semantics $\lightning_S$ and $\lightning_R$ and the semantics by Guarnieri et al. [17]. Our framework yields security of the composed semantics (almost) for free, given the security of its parts. All the new semantics are implemented as extension in the tool Spectector and the tool correctly detects all vulnerabilities in existing as well as in novel benchmarks.

There are multiple directions for future works. Our composition is restricted in the sense that the different speculation sources do not influence each other. Koruyeh et al. [22] argued that during a transient execution caused by branch misprediction, one can add entries to the RSB by transiently executing a call instruction. During the roll back, the RSB is not rolled back, which could lead to more speculation down the line. However, this composition would not be $\vdash : \mathit{WFC}$ because this composition would not be related to its projection (for a rather technical reason one can see in the technical report). In the future, we would like to allow these combinations as well without losing all the 'free' theorems presented here.

REFERENCES

[1] 2019. SafeSide. https://github.com/google/safeside (2022).

[2] 2021. Expected 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 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 1807–1823. https://doi.org/10.1145/3133956.3134078

[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 *USENIX Security*, Vol. 11.

[5] G. Barthe, S. Cauligi, B. Gregoire, A. Koutsos, K. Liao, T. Oliveira, S. Priya, T. Rezk, and P. Schwabe. 2021. High-Assurance Cryptography in the Spectre Era. In *2021 2021 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 788–805. https://doi.org/10.1109/SP40001.2021.00046

[6] G. Barthe, P.R. D'Argenio, and T. Rezk. 2004. Secure information flow by self-composition. In *Proceedings. 17th IEEE Computer Security Foundations Workshop, 2004.* 100–114. https://doi.org/10.1109/CSFW.2004.1310735

[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 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 785–800. https://doi.org/10.1145/3319535.3363194

[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 *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 249–266. https://www.usenix.org/conference/usenixsecurity19/presentation/canella

[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* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 913–926. https://doi.org/10.1145/3385412.3385970

[10] S. Cauligi, C. Disselkoen, D. Moghimi, G. Barthe, and D. Stefan. 2022. SoK: Practical Foundations for Software Spectre Defenses. In *2022 2022 IEEE Symposium on Security and Privacy (SP) (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 1517–1517. https://doi.org/10.1109/SP46214.2022.00088

[11] K. Cheang, C. Rasmussen, S. Seshia, and P. Subramanyan. 2019. A Formal Approach to Secure Speculation. In *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*. 288–28815. https://doi.org/10.1109/CSF.2019.00027

[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 *Formal Methods. FM 2019 International Workshops: Porto, Portugal, October 7–11, 2019, Revised Selected Papers, Part II* (Porto, Portugal). Springer-Verlag, Berlin, Heidelberg, 323–341. https://doi.org/10.1007/978-3-030-54997-8_21

[14] L.-A. Daniel, S. Bardin, and T. Rezk. 2021. Hunting the Haunter — Efficient relational symbolic execution for Spectre with Haunted RelSE. In *NDSS*.

[15] Craig Disselkoen, Radha Jagadeesan, Alan Jeffrey, and James Riely. 2019. The Code That Never Ran: Modeling Attacks on Speculative Evaluation. In *2019 IEEE Symposium on Security and Privacy (SP)*. 1238–1255. https://doi.org/10.1109/SP.2019.00047

[16] Roberto Guanciale, Musard Balliu, and Mads Dam. 2020. InSpectre: Breaking and Fixing Microarchitectural Vulnerabilities by Formal Analysis. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, USA) *(CCS '20)*. Association for Computing Machinery, New York, NY, USA, 1853–1869. https://doi.org/10.1145/3372297.3417246

[17] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez. 2020. Spectector: Principled Detection of Speculative Information Flows. In *2020 IEEE Symposium on Security and Privacy (SP)*. 1–19. https://doi.org/10.1109/SP40000.2020.00011

[18] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. 2021. Hardware-Software Contracts for Secure Speculation. In *2021 IEEE Symposium on Security and Privacy (SP)*. 1868–1883. https://doi.org/10.1109/SP40001.2021.00036

[19] 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 ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1235–1247. https://doi.org/10.1145/3377811.3380428

[20] 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.

[21] 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 *2019 IEEE Symposium on Security and Privacy (SP)*. 1–19. https://doi.org/10.1109/SP.2019.00002

[22] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2018. Spectre Returns! Speculation Attacks Using the Return Stack Buffer *(WOOT'18)*. USENIX Association, USA, 3.

[23] Giorgi Maisuradze and Christian Rossow. 2018. Ret2spec: Speculative Execution Using Return Stack Buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 2109–2122. https://doi.org/10.1145/3243734.3243761

[24] 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. *CoRR* abs/1902.05178 (2019). arXiv:1902.05178 http://arxiv.org/abs/1902.05178

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

[26] H. Ponce de León and J. Kinder. 2022. Cats vs. Spectre: An Axiomatic Approach to Modeling Speculative Execution Attacks. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 1415–1428. https://doi.org/10.1109/SP46214.2022.00082

[27] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. 2019. NetSpectre: Read Arbitrary Memory over Network. In *Computer Security – ESORICS 2019*, Kazue Sako, Steve Schneider, and Peter Y. A. Ryan (Eds.). Springer International Publishing, Cham, 279–299.

[28] Julian Stecklina and Thomas Prescher. 2018. LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels. *CoRR* abs/1806.07480 (2018). arXiv:1806.07480 http://arxiv.org/abs/1806.07480

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

[30] 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. *Proc. ACM Program. Lang.* 5, POPL, Article 49 (Jan. 2021), 30 pages. https://doi.org/10.1145/3434330

[31] 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 Trans. Softw. Eng. Methodol.* 29, 3, Article 14 (June 2020), 31 pages. https://doi.org/10.1145/3385897

[32] 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), 2504–2519. https://doi.org/10.1109/TSE.2019.2953709

[33] Tao Zhang, Kenneth Koltermann, and Dmitry Evtyushkin. 2020. Exploring Branch Predictors for Constructing Transient Execution Trojans. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 667–682. https://doi.org/10.1145/3373376.3378526