Exorcising Spectres with Secure Compilers

***Abstract*—Attackers can access sensitive information of programs by exploiting the side-effects of speculatively-executed instructions using Spectre attacks. To mitigate theses attacks, popular compilers deployed a wide range of countermeasures. The security of these countermeasures, however, has not been ascertained: while some of them are *believed* to be secure, others are *known* to be insecure and result in vulnerable programs.**

To reason about the security guarantees of these compiler-level countermeasures, this paper presents a framework comprising several secure compilation criteria characterizing when compilers produce code resistant against Spectre attacks. With this framework, we perform a comprehensive security analysis of compiler-level countermeasures against Spectre attacks implemented in major compilers.

This work provides sound foundations to formally reason about the security of compiler-level countermeasures against Spectre attacks as well as the first proofs of security and insecurity of said countermeasures.

To better present notions, this paper uses colours in a way that both colourblind and black&white readers can benefit from [49]. For a better experience, please print or view this in colour.

I. Introduction

By predicting the outcome of branching (and other) instructions, CPUs can trigger speculative execution and speed up computation by executing code based on such predictions. When predictions are incorrect, CPUs roll back the effects of speculatively-executed instructions on the architectural state (i.e., memory, flags, and registers). However, they do *not* roll back effects on microarchitectural components like caches.

Attackers can exploit microarchitectural leaks caused by speculative execution using Spectre attacks [35, 37, 38, 41, 57]. To mitigate these attacks, compilers deployed a number of compiler-level countermeasures. For instance, the insertion of `lfence` speculation barriers [31] and speculative load hardening [16] *can* mitigate leaks introduced by speculation over branch instructions (i.e., the Spectre v1 attack [37]).

Existing countermeasures, however, are often developed in an unprincipled way, that is, they are not *proven* to be secure, and some of them fail in blocking speculative leaks, i.e., those resulting from speculatively executed instructions. For instance, the Microsoft Visual C++ compiler misplaces speculation barriers, thereby producing programs that are still vulnerable to Spectre attacks [27, 36].

In this paper, we present a framework for reasoning about compiler-level countermeasures against speculative execution attacks. Using this framework, we precisely characterize the security guarantees provided by Spectre countermeasures in major C compilers. Thus, we make these contributions:

▶ We present a secure compilation framework tailored towards reasoning about speculative execution attacks (Section II). The distinguishing feature of our framework is that compilers translate programs from a source language L, which has a standard imperative semantics, into a target language **T** that is equipped with a speculative semantics (inspired by the always mispredict semantics from [27]) capturing the effects of speculatively-executed instructions.[1] This matches a programmer's mental model: programmers do not think about speculative execution when writing source code (and they should not!) since speculation only exists in processors (captured by **T**'s speculative semantics). It is the duty of a (secure) compiler to ensure **T**'s features cannot be exploited.[2] Through minor changes to the languages' semantics, our framework encompasses two different security models for speculative execution: (1) *(Strong) speculative non-interference* [27] (SNI), which considers all leaks derived from speculatively-executed instructions as harmful, and (2) *Weak speculative non-interference* [28], which instead focuses only on (speculative) leaks of speculatively-accessed data.

▶ We introduce *speculative safety* (SS, Section III), a novel safety property that implies the absence of classes of speculative leaks. The key features of SS are that (1) it is parametric in a taint-tracking mechanism, which we leverage to reason about security by focusing on single traces, and (2) it is formulated to simplify proving that a compiler preserves it. We instantiate SS using two different taint-tracking mechanisms obtaining *strong SS* and *weak SS*. We characterize the security guarantees of SS by showing that strong (resp. weak) SS over-approximates strong (resp. weak) speculative non-interference.

▶ We define two novel secure compilation criteria: *Robust Speculative Safety Preservation* ($RSSP$) and *Robust Speculative Non-Interference Preservation* ($RSNIP$, Section IV). These criteria respectively ensure that compilers preserve (strong or weak) SS and SNI *robustly*, i.e, even when linked against arbitrary (potentially malicious) code. Satisfying these criteria implies that compilers correctly place countermeasures to prevent speculative leaks. However, $RSSP$ requires preserving a safety property (SS) and it is much simpler to prove than $RSNIP$, which requires preserving a hyperproperty [20]. To the best of our knowledge, these are the first criteria that concretely instantiate a recent theory that phrases security of compilers as the preservation of (hyper)properties [3, 4, 52] (here: absence of speculative leaks).

▶ Using our framework, we perform a comprehensive security analysis of compiler-level countermeasures against Spectre v1 implemented in major compilers (Section V). Specifically, we focus on (1) automated insertion of `lfences`

[1]In this paper we use a blue, sans-serif font for elements of the source language, an **orange**, **bold** font for elements of the **target** language. Elements of the meta-language or common to all languages are typeset in a $black$, $italic$ font (to avoid repeating similar definitions twice).

[2] Secure countermeasures can be seen as preventing speculative leaks.

(implemented in the Microsoft Visual C++ and the Intel ICC compilers [33, 48]), and (2) speculative load hardening (SLH, implemented in Clang [16]). Our analysis proves that:

- The Microsoft Visual C++ implementation of (1) violates weak *RSNIP* and is thus insecure.
- The Intel ICC implementation of (1) provides strong *RSNIP*, so compiled programs have *no* speculative leaks.
- SLH provides weak *RSNIP*, so compiled programs do not leak speculatively-accessed data, which is sufficient to prevent Spectre-style attacks. However, compiled programs might still contain speculative leaks.
- The non-interprocedural variant of SLH violates weak *RSNIP* and is thus insecure.
- We propose a variant of SLH, called strong SLH, that provides strong *RSNIP* and blocks all speculative leaks.

All our security proofs follow a common methodology (see Section IV-C) whose key insight is that, by exploiting that SS over-approximates SNI, proving a countermeasure to be *RSSP* is sufficient to ensure its security. This allows us to directly leverage SS to simplify our security proofs.

After presenting these results, we discuss how to extend our methodology to countermeasures against other Spectre variants (Section VI). Then we discuss related work (Section VII) and conclude (Section VIII).

For simplicity, most formalisation is elided or simplified (but we discuss all key aspects); auxiliary lemmas and proofs are omitted. Full details are in the supplementary material.

II. Modelling Speculative Execution

To illustrate our speculative execution model, we first introduce Spectre v1 (Section II-A). Using that, we define the threat model that we consider (Section II-B). Then, we present the syntax of our languages (Section II-C) and the trace model (Section II-D). This is followed by the non-speculative semantics of our languages (Section II-E). Next, we present the source trace semantics (Section II-F), the target speculative semantics (Section II-G), and the target trace semantics (Section II-H). This formalisation focuses on the strong variants of SS and SNI, so we conclude by defining the changes necessary for the weak variants (Section II-I).

A. Spectre v1: Illustrative example

```
1 void get (int y)
2   if (y < size) then
3     temp = B[A[y]*512]
```

Listing 1. The classic Spectre v1 snippet.

Consider the standard Spectre v1 example [37] in Listing 1. Function `get` checks whether the index stored in variable `y` is less than the size of array `A`, stored in the global variable `size`. If so, the program retrieves `A[y]`, multiplies it by the cache line size (here: 512), and uses the result to access array `B`.

If `size` is not cached, modern processors predict the guard's outcome and speculatively continue the execution. Thus, line 3 might be executed even if `y` $\geq$ `size`. When `size` becomes available, the processor checks whether the prediction was correct. If not, it rolls back all changes to the architectural state and executes the correct branch. However, the speculatively-executed memory accesses leave a footprint in the cache, which enables an adversary to retrieve `A[y]` even for `y` $\geq$ `size`.

B. Threat Model

As mentioned, we study compiler countermeasures that translate source programs into (hardened) target ones.

In our setting, an attacker is an arbitrary program at target level that is linked against a (compiled) partial program of interest. The partial program (or, *component*) stores sensitive information in a private heap that is not accessible to the attacker. For example, in the snippet of Listing 1, the array `A` would be stored in the private heap and the attacker is code that runs before and after function `get`.

While attackers cannot directly access the private heap, they can mount confused deputy attacks [29, 54] to trick components into leaking sensitive information. We focus on preventing *only* speculative leaks, i.e., those caused by speculatively executed instructions. For this, our attacker can observe the program counter and the locations of memory accesses during program execution. This attacker model is commonly used to formalise timing side-channel free code [8, 45] without requiring microarchitectural models. Following [27], we capture this model in our semantics through traces that record the address of all memory accesses (e.g., the address of `B[A[y]*512]` in Listing 1) and the outcome of all control-flow instructions.

To model the effects of speculative execution, our target language mispredicts the outcome of all branch instructions in the component. This is the worst-case scenario in terms of leakage regardless of how attackers poison the branch predictor [27].

C. Languages L and T

Technically, we have a pair of source and target languages (L and T) for studying strong security definitions and a pair of source and target languages (L^- and T^-) for studying the weak ones. Strong (L-T) and weak (L^--T^-) languages have the same syntax but slightly different semantics. We focus this section and the following ones on the strong languages L-T; we introduce the weak languages L^--T^- in Section II-I.

The source (L) and target (T) languages are single-threaded while languages with a heap, a stack to lookup local variables, and a notion of components (our unit of compilation). We focus on such a setting, instead of an assembly-style language like [17, 27], to reason about speculative leaks without getting bogged down in complications like unstructured control flow.

Both L and T have a taint-tracking mechanism, where values can be tainted as "safe" (denoted by S) or "unsafe" (denoted by U). Taint-tracking is at the foundation of our speculative safety definition and it enables reasoning about security on single traces. We consider two taint-tracking mechanisms, a strong and a weak one, that lead to different security guarantees (see Section III). Each mechanism is adopted in the related pair of languages: strong (resp. weak) languages use the strong (resp. weak) taint-tracking. Our semantics are parametric in the taint-tracking, so that there is minimal notation overhead and duplication of rules between languages.

The common syntax of L and T is presented below; we indicate sequences of elements $e_1, \cdots, e_n$ as $\overline{e}$ and $\overline{e} \cdot e$ denotes a stack with top element e and rest of the stack $\overline{e}$.

$$\begin{aligned}
&\mathit{Programs}\ W, P ::= H, \overline{F}, \overline{I} \qquad \mathit{Codebase}\ C ::= \overline{F}, \overline{I}\\
&\mathit{Functions}\ F ::= f(x) \mapsto s; \mathit{return}; \qquad \mathit{Imports}\ I ::= f\\
&\mathit{Attackers}\ A ::= H, \overline{F}\,[\cdot] \qquad \mathit{Taint}\ \sigma ::= S \mid U\\
&\mathit{Heaps}\ H ::= \varnothing \mid H; n \mapsto v : \sigma \quad \text{where } n \in \mathbb{Z}\\
&\mathit{Value\ Heaps}\ H_v ::= \varnothing \mid H_v; n \mapsto v \quad \text{where } n \in \mathbb{Z}\\
&\mathit{Taint\ Heaps}\ H_t ::= \varnothing \mid H_t; n \mapsto \sigma \quad \text{where } n \in \mathbb{Z}\\
&\mathit{Expressions}\ e ::= x \mid v \mid e \oplus e \qquad \mathit{Values}\ v ::= n \in \mathbb{N}\\
&\mathit{Statements}\ s ::= \mathit{skip} \mid s; s \mid \mathit{let}\ x = e\ \mathit{in}\ s \mid \mathit{call}\ f\ e\\
&\qquad \mid \mathit{ifz}\ e\ \mathit{then}\ s\ \mathit{else}\ s \mid e := e \mid e :=_p e\\
&\qquad \mid \mathit{let}\ x = \mathit{rd}\ e\ \mathit{in}\ s \mid \mathit{let}\ x = \mathit{rd}_p\ e\ \mathit{in}\ s\\
&\qquad \mid \mathit{lfence} \mid \mathit{let}\ x = e\ (\mathit{if}\ e)\ \mathit{in}\ s
\end{aligned}$$

We model *components*, i.e., partial programs (P), and *attackers* (A). A (partial) program P defines its heap H, a list of functions $\overline{F}$, and a list of imports $\overline{I}$, which are all the functions an attacker can define. An attacker A just defines its heap and its functions. We indicate the code base of a program (its functions and imports) as C.

Functions are untyped, and their bodies are sequences of statements s that include standard instructions: skipping, sequencing, let-bindings, conditional branching, writing the public and the private heap, reading the public and private heap, speculation barriers, and conditional assignments. Statements can contain expressions e, which include program variables x, natural numbers n, arithmetic and comparison operators $\oplus$.

Heaps H map memory addresses $n \in \mathbb{Z}$ to tainted values $v : \sigma$. Heaps H can be split in their value-only part H_v (used for the language semantics) and their taint-only part H_t (used for taint-tracking). We denote this split as $H \equiv H_v + H_t$. All heaps are partitioned in a public part (when the domain $n \geq 0$) and a private part (if $n < 0$). An attacker A can only define and access the public heap. A program P defines a private heap and it can access both private and public heaps.

D. Labels and Traces

Computation steps in L and T are *labelled* with labels λ, which can be the *empty label* ϵ, an *action* α recording the control-flow between attacker and code (as required for secure compilation proofs [2, 4, 50, 52]), or a *μarch. action* δ capturing what a microarchitectural attacker can observe.

$$\begin{aligned}
&\mathit{Actions}\ \alpha ::= \mathtt{call}\ f\ v? \mid \mathtt{call}\ f\ v! \mid \mathtt{ret!} \mid \mathtt{ret?}\\
&\mu\mathit{arch.\ Acts.}\ \delta ::= \mathtt{read}(n) \mid \mathtt{write}(n) \mid \mathtt{if}(v) \mid \mathtt{rlb}\\
&\mathit{Labels}\ \lambda ::= \epsilon \mid \alpha \mid \delta
\end{aligned}$$

Action $\mathtt{call}\ f\ v?$ represents a call to a function f in the component with value v. Dually, $\mathtt{call}\ f\ v!$ represents a call(back) to the attacker with value v. Action $\mathtt{ret!}$ represents a return to the attacker and $\mathtt{ret?}$ a return(back) to the component.

The $\mathtt{read}(n)$ and $\mathtt{write}(n)$ actions denote respectively read and write accesses to the heap location n, and they model leaks through the data cache. In contrast, the $\mathtt{if}(v)$ action denotes the outcome of branch instructions and the $\mathtt{rlb}$ action indicates the roll-back of speculatively-executed instructions. These actions implicitly expose which instruction we are currently executing, and thus the instruction cache content.

Traces $\overline{\lambda^\sigma}$ are sequences of labels, each tainted with a taint σ. The semantics only track μarch. actions executed inside the component P, whereas those executed in the attacker-controlled context A are ignored (Rule E-L-single later on). The reason is that μarch. actions produced by A can be safely ignored, as done in other robust safety works [23, 25, 40, 60], since A cannot access the private heap. Therefore, traces have this normal form: $\overline{\alpha?^\sigma \overline{\delta^\sigma} \alpha!^\sigma}$, where α^σs are tainted calls/returns, δ^σs are tainted μarch. actions, and the alternation of ? and ! actions is due to well-bracketed control-flow.

E. Non-Speculative Semantics for L and T

Both languages are given a labelled operational semantics that describe how whole programs execute. A component P and an attacker A can be linked to obtain a whole program $W \equiv A\,[P]$ that contains the functions and heaps of A and P. Only whole programs can run, and a program is whole only if it defines all functions that are called and if the attacker defines all the functions in the interfaces of P.

Program states $C, H, \overline{B} \triangleright (s)_{\overline{f}}$ consist of a codebase C, a heap H, a stack of local variables $\overline{B}$, a statement s, and a stack of function names $\overline{f}$. Just like heaps, local variable bindings B are split between a value part B_v and a taint part B_t that can be merged as $B_v + B_t$. C is used to look up function bodies and to determine which functions are the component's and which are the attacker's. Function names $\overline{f}$, which we often omit for simplicity, are used to infer if the code that is executing comes from the attacker or from the component, and this determines the produced labels.

$$\begin{aligned}
&\mathit{Bindings}\ B ::= \varnothing \mid B; x \mapsto v : \sigma\\
&\mathit{Value\ Bindings}\ B_v ::= \varnothing \mid B_v; x \mapsto v\\
&\mathit{Taint\ Bindings}\ B_t ::= \varnothing \mid B_t; x \mapsto \sigma\\
&\mathit{Prog.\ States}\ \Omega ::= C, H, \overline{B} \triangleright (s)_{\overline{f}}\\
&\mathit{Value\ States}\ \Omega_v ::= C, H_v, \overline{B_v} \triangleright (s)_{\overline{f}}\\
&\mathit{Taint\ States}\ \Omega_t ::= C, H_t, \overline{B} \triangleright (s)_{\overline{f}}
\end{aligned}$$

The operational semantics (Section II-E1) relies only on value states Ω_v, that is, states whose heap and stack of bindings only contain values. The taint-tracking semantics (Section II-E2), instead, relies on taint states Ω_t, whose heap only tracks taint but whose stack of bindings contains both values and taints (values are needed to determine how to update taints for memory operations as we explain later).

1) Operational Semantics: Both L and T have a big-step operational semantics for expressions and a small-step, structural operational semantics for statements that generates labels. The former produces judgments $B_v \triangleright e \downarrow v$ meaning: "according to variables B_v, expression e reduces to value v." The latter produces judgments $\Omega_v \xrightarrow{\lambda} \Omega_v'$ meaning: "state Ω_v reduces in one step to Ω_v' emitting label λ." The rules describing these semantics are standard and therefore omitted.

We remark that values are computed as expected (though we use *0* for true in *ifz* statements) and expressions access only local variables in B_v (reading from the heap is treated as a statement). The rules of conditionals, read, and write emit the related μarch. actions (from Section II-D).

2) Taint-tracking semantics: The taint-tracking semantics tracks taints of values (both in heaps and variable bindings) and of the program counter (pc).

Taints form the usual integrity lattice $S \leq U$ and are combined using the least-upper-bound ($\sqcup$) and greatest-lower-bound ($\sqcap$) operators. For simplicity, we report the key cases of the truth tables: $S \sqcup U = U$ and $S \sqcap U = S$.

Taints are calculated using two judgements. Judgement $B_t \triangleright e \downarrow \sigma$ reads as "expression e is tainted as σ according to the variable taints B_t". In contrast, judgement $\sigma; \Omega_t \xrightarrow{\sigma'} \Omega_t'$ reads as "when the pc has taint σ, state Ω_t single-steps to Ω_t' producing a (possibly empty) action with taint σ'". The most representative rules are those interacting with the private heap:

(T-write-prv)

$$\frac{B \triangleright e \downarrow n : \sigma \qquad B \triangleright e' \downarrow \_ : \sigma'' \qquad H_t' = H_t \cup -|n| \mapsto \sigma''}{\sigma_{pc}; C, H_t, \overline{B} \cdot B \triangleright e :=_p e' \xrightarrow{\sigma \sqcap \sigma_{pc}} C, H_t', \overline{B} \cdot B \triangleright \mathit{skip}}$$

(T-read-prv)

$$\frac{B \triangleright e \downarrow n : \sigma' \qquad n_a = -|n| \qquad H_t(n_a) = \sigma'' \qquad \sigma = \sigma'' \sqcup \sigma'}{\begin{array}{r}\sigma_{pc}; C, H_t, \overline{B} \cdot B \triangleright \mathit{let}\ x = \mathit{rd}_p\ e\ \mathit{in}\ s \xrightarrow{\sigma \sqcap \sigma_{pc}} \\ C, H_t, \overline{B} \cdot B \cup x \mapsto \mathit{0} : U \triangleright s\end{array}}$$

Writing to the private heap (Rule T-write-prv) taints the location ($-|n|$) with the taint of the written expression (σ'). In contrast, reading from the private heap (Rule T-read-prv) taints the variable where the content is stored as unsafe (U) and the read value is set to *0* (this information is not used by the taint-tracking, see Section II-F). Both rules taint the action with the least-upper-bound of the pc (σ_{pc}) and data taint (σ). In the rules, we use $|n|$ for the absolute value of n, $H_v \cup n \mapsto \sigma$ to update the binding for n in H_v, and $H_v(n)$ to look up n's taint in H_v.

To correctly taint memory accesses, we need to evaluate expression e to derive the accessed location $|n|$; see, for instance, Rule T-write-prv. This is why taint-tracking states Ω_t contain the full stack of bindings B and not just the taints B_t. The rules above rely on a judgement $B \triangleright e \downarrow n : \sigma$ which is obtained by joining the result of the expression semantics on B's values and of the taint-tracking semantics on B's taints.

(Combine-B)

$$\frac{B_v + B_t \equiv B \qquad B_v \triangleright e \downarrow v \qquad B_t \triangleright e \downarrow \sigma}{B \triangleright e \downarrow v : \sigma}$$

Here ends the part of the semantics that is common to both L and $\mathbf{T}$, we now introduce the bits where they differ.

F. *Trace Semantics for* L

The operational and taint single-steps from Section II-E are combined according to the judgement $\Omega \xrightarrow{\lambda^\sigma} \Omega'$ below.

(Combine-s-L)

$$\frac{\Omega_v + \Omega_t \equiv \Omega \qquad \Omega_v' + \Omega_t' \equiv \Omega' \qquad \Omega_v \xrightarrow{\lambda} \Omega_v' \qquad \mathsf{S}; \Omega_t \xrightarrow{\sigma} \Omega_t'}{\Omega \xrightarrow{\lambda^\sigma} \Omega'}$$

(Merge-Ω)

$$\frac{H_v + H_t \equiv H \qquad \overline{B_v'} + \overline{B_t} \equiv \overline{B} \qquad \overline{B_v} + \overline{B_t} \equiv \overline{B'}}{C; H_v; \overline{B_v} \triangleright s + C; H_t; \overline{B} \triangleright s' \equiv C; H; \overline{B'} \triangleright s}$$

Intuitively, the operational semantics determines how states reduce ($\Omega_v \xrightarrow{\lambda} \Omega_v'$), whereas the taint-tracking semantics determines the action's label and how taints are updated ($\mathsf{S}; \Omega_t \xrightarrow{\sigma} \Omega_t'$). We remark that the pc taint is always safe since there is no speculation in L. Moreover, merging states $\Omega_v + \Omega_t$ results in ignoring the value information accumulated in Ω_t since we rely on the computation performed by the operational semantics for values (Rule Merge-Ω).

Next, we can define the big-step semantics $\Longrightarrow$ of L, which concatenates single steps into multiple ones and single labels into traces. The judgement $\Omega \xRightarrow{\overline{\lambda^\sigma}} \Omega'$ is read: "state Ω emits trace $\overline{\lambda^\sigma}$ and becomes Ω'". The most interesting rule is below:

(E-L-single)

$$\frac{\begin{array}{c}\Omega \equiv \overline{\mathsf{F}}, \overline{\mathsf{I}}, \mathsf{H}, \mathsf{B} \triangleright (\mathsf{s})_{\overline{\mathsf{f}} \cdot \mathsf{f}} \qquad \Omega' \equiv \overline{\mathsf{F}}, \overline{\mathsf{I}}, \mathsf{H}', \mathsf{B}' \triangleright (\mathsf{s}')_{\overline{\mathsf{f}'} \cdot \mathsf{f}'} \\ \Omega \xrightarrow{\alpha^\sigma} \Omega' \qquad \text{if } \mathsf{f} == \mathsf{f}' \text{ and } \mathsf{f} \in \mathsf{I} \text{ then } \overline{\lambda^\sigma} = \epsilon \text{ else } \overline{\lambda^\sigma} = \alpha^\sigma\end{array}}{\Omega \xRightarrow{\overline{\lambda^\sigma}} \Omega'}$$

As mentioned in Section II-D, the trace does not contain μarch. actions performed by the attacker (see the 'then' branch, recall that functions in $\overline{\mathsf{I}}$ are defined by the attacker).

Finally, the behaviour $\mathsf{Beh}(\mathsf{W})$ of a whole program W is the trace $\overline{\lambda^\sigma}$ generated according to the $\Longrightarrow$ semantics starting from the initial state of W (indicated as $\Omega_0(\mathsf{W})$) until it terminates.[3] Intuitively, the initial state of a program is the main function, which is defined by the attacker.

G. *Speculative Semantics for* $\mathbf{T}$

Our semantics for $\mathbf{T}$ is inspired by the "always mispredict" semantics of Guarnieri *et al.* [27], which captures the worst-case scenario (from an information theoretic perspective) independently of the branch prediction outcomes. Whenever the semantics executes a branch instruction, it first mis-speculates by executing the wrong branch for a fixed number $\mathbf{w}$ of steps (called *speculation window*). After speculating for $\mathbf{w}$ steps, the speculative execution is terminated, the changes to the program state are rolled back, and the semantics restarts by executing the correct branch. The μarch. effects of speculatively-executed instructions are recorded on the trace as actions. For taint-tracking, the taint of the program counter starts as $\mathbf{S}$ and it is raised to $\mathbf{U}$ when speculation happens.

As for the non-speculative semantics, we decouple the operational aspects from the taint-tracking ones. Speculative program states ($\mathbf{\Sigma}$) are defined as stacks of speculation instances ($\mathbf{\Phi}$), which in turn are split in their operational ($\mathbf{\Phi_v}$) and taint ($\mathbf{\Phi_t}$) sub-parts. A speculation instance $(\mathbf{\Omega}, \mathbf{w}, \boldsymbol{\sigma})$ records the

[3] In [3, 4], a program behaviour is a set of traces due to non-determinism. Our language is fully deterministic; so the behaviour is a single trace [39].

program state Ω, the remaining speculation window w, and the taint σ of the program counter. The operational part (Φ_v) keeps track of the operational part of the program state (Ω_v) and of the speculation window. The taint part (Φ_t) keeps track of the taint part of the program state (Ω_t) and the taint of the pc (σ). As before, Φ_v and Φ_t can be merged as $\Phi \equiv \Phi_v + \Phi_t$. The speculation window is a natural number n or $\perp$ when no speculation is happening; its maximum length is a global constant ω that depends on physical characteristics of the CPU like the size of the reorder buffer.

$$\begin{aligned}
\textit{Speculative States } \Sigma &::= \overline{\Phi}\\
\textit{Speculation Instance } \Phi &::= (\Omega, w, \sigma)\\
\textit{Speculation Instance Vals. } \Phi_v &::= (\Omega_v, w)\\
\textit{Speculation Instance Taint } \Phi_t &::= (\Omega_t, \sigma)
\end{aligned}$$

The execution of program W starts in state $(\Omega_0(W), \perp, S)$, i.e., in the same initial state that L starts in, with the program counter tainted as S since no speculation has happened yet.

1) Operational Semantics: In the small-step operational semantics $\overline{\Phi_v} \overset{\lambda}{\rightsquigarrow} \overline{\Phi'_v}$, reductions happen at the top of the stack:

(E-T-speculate-lfence)

$$\frac{\Omega_v \xrightarrow{\epsilon} \Omega'_v \qquad \Omega_v \equiv C, H_v, \overline{B_v} \triangleright s; s' \qquad s \equiv \mathtt{lfence}}{\overline{\Phi_v} \cdot (\Omega_v, n+1) \overset{\epsilon}{\rightsquigarrow} \overline{\Phi_v} \cdot (\Omega'_v, 0)}$$

(E-T-speculate-action)

$$\frac{\begin{gathered}\Omega_v \xrightarrow{\lambda} \Omega'_v \qquad \Omega_v \equiv C, H_v, \overline{B_v} \triangleright s; s'\\ s \not\equiv \mathtt{ifz}\ \_\ \mathtt{then}\ \_\ \mathtt{else}\ \_ \text{ and } s \not\equiv \mathtt{lfence}\end{gathered}}{\overline{\Phi_v} \cdot (\Omega_v, n+1) \overset{\lambda}{\rightsquigarrow} \overline{\Phi_v} \cdot (\Omega'_v, n)}$$

(E-T-speculate-if)

$$\frac{\begin{gathered}\Omega_v \equiv C, H_v, \overline{B_v} \cdot B_v \triangleright (s; s')_{\overline{f} \cdot f} \qquad s \equiv \mathtt{if}\ e\ \mathtt{then}\ s''\ \mathtt{else}\ s'''\\ \Omega_v \xrightarrow{\alpha} \Omega'_v \qquad C \equiv \overline{F}; \overline{I} \qquad f \notin \overline{I} \qquad j = \mathtt{min}(\omega, n)\\ \text{if } B_v \triangleright e \downarrow 0 \text{ then } \Omega''_v \equiv C, H_v, \overline{B_v} \cdot B_v \triangleright s'''; s'\\ \text{if } B_v \triangleright e \downarrow n \text{ and } n > 0 \text{ then } \Omega''_v \equiv C, H_v, \overline{B_v} \cdot B_v \triangleright s''; s'\end{gathered}}{\overline{\Phi_v} \cdot (\Omega_v, n+1) \overset{\alpha}{\rightsquigarrow} \overline{\Phi_v} \cdot (\Omega'_v, n) \cdot (\Omega''_v, j)}$$

(E-T-speculate-rollback)

$$\frac{n = 0 \text{ or } \Omega_v \text{ is stuck}}{\overline{\Phi_v} \cdot (\Omega_v, n) \overset{\mathtt{rlb}}{\rightsquigarrow} \overline{\Phi_v}}$$

Mis-speculation pushes the mis-speculating state on top of the stack (Rule E-T-speculate-if). Note that speculation does not happen in attacker code (condition $f \notin \overline{I}$, recall that f is the function executing now and $\overline{I}$ are all attacker-defined functions). This is without loss of generality since (1) attackers cannot directly access the private heap, and (2) our security definitions (Section III) will consider any possible attacker, so the speculative behavior of an attacker (i.e., the speculative execution of the 'wrong branch') will be captured by another one who has the same branches but inverted (e.g., the 'then' code of one attacker is the 'else' code of another). When the speculation window is exhausted (or if the speculation reached a stuck state), speculation ends and the top of the stack is popped (Rule E-T-speculate-rollback). The role of the $\mathtt{lfence}$ instruction is setting to zero the speculation window, so that rollbacks are triggered (Rule E-T-speculate-lfence).

2) Taint-tracking semantics: Similarly to the operational semantics, reductions happen at the top of the stack also for the taint-tracking semantics $\overline{\Phi_t} \overset{\sigma}{\rightsquigarrow} \overline{\Phi'_t}$. Selected rules are below:

(T-T-speculate-action)

$$\frac{\begin{gathered}\sigma; \Omega_t \xrightarrow{\sigma'} \Omega'_t \qquad \Omega_t \equiv C, H_t, \overline{B} \triangleright s; s'\\ s \not\equiv \mathtt{ifz}\ \_\ \mathtt{then}\ \_\ \mathtt{else}\ \_ \text{ and } s \not\equiv \mathtt{lfence}\end{gathered}}{\overline{\Phi_t} \cdot (\Omega_t, \sigma) \overset{\sigma' \sqcap \sigma}{\rightsquigarrow} \overline{\Phi_t} \cdot (\Omega'_t, \sigma)}$$

(T-T-speculate-if)

$$\frac{\begin{gathered}\Omega_t \equiv C, H_t, \overline{B} \cdot B \triangleright (s; s')_{\overline{f} \cdot f} \qquad s \equiv \mathtt{if}\ e\ \mathtt{then}\ s''\ \mathtt{else}\ s'''\\ \sigma'; \Omega_t \xrightarrow{\sigma} \Omega'_t \qquad C \equiv \overline{F}; \overline{I} \qquad f \notin \overline{I}\\ \text{if } B \triangleright e \downarrow 0 : \sigma \text{ then } \Omega''_t \equiv C, H_t, \overline{B} \cdot B \triangleright s'''; s'\\ \text{if } B \triangleright e \downarrow n : \sigma \text{ and } n > 0 \text{ then } \Omega''_t \equiv C, H_t, \overline{B} \cdot B \triangleright s''; s'\end{gathered}}{\overline{\Phi_t} \cdot (\Omega_t, \sigma') \overset{\sigma \sqcap \sigma'}{\rightsquigarrow} \overline{\Phi_t} \cdot (\Omega'_t, \sigma') \cdot (\Omega''_t, U)}$$

In these rules, σ is the program counter taint which is combined with the action taint σ' (Rules T-T-speculate-action and T-T-speculate-if). Mis-speculation pushes a new state on top of the stack whose program counter is tainted U denoting the beginning of speculation (Rule T-T-speculate-if).

H. Trace Semantics for T

The two operational and taint-tracking single steps from Section II-G are combined in a single reduction as follows:

(Combine-T)

$$\frac{\overline{\Phi_v} + \overline{\Phi_t} \equiv \Sigma \qquad \overline{\Phi'_v} + \overline{\Phi'_t} \equiv \Sigma' \qquad \overline{\Phi_v} \overset{\lambda}{\rightsquigarrow} \overline{\Phi'_v} \qquad \overline{\Phi_t} \overset{\sigma}{\rightsquigarrow} \overline{\Phi'_t}}{\Sigma \overset{\lambda^\sigma}{\rightsquigarrow} \Sigma'}$$

This reduction is used by the big-step semantics $\Sigma \overset{\overline{\lambda^\sigma}}{\Longrightarrow} \Sigma'$ that concatenates single labels into traces, which, as before, do not contain microarchitectural actions generated by the attacker. Rules for traces of T are analogous to those of L (e.g., Rule E-L-single) except that they rely on single steps made by the speculative semantics ($\rightsquigarrow$) instead of the non-speculative one ($\longrightarrow$).

As before, the behaviour $\mathrm{Beh}(W)$ of a whole program W is the trace $\overline{\lambda^\sigma}$ generated, according to the $\Rightarrow$ semantics, starting from the initial state of W until termination.

We now show how to apply the trace semantics to Listing 1.

Example 1 (L and T Traces for Listing 1)**.** Consider array `A` being U and `size=4`. Trace t_{ns} below indicates a valid execution of the code in L, and thus without speculation. On the other hand, trace t_{sp} is a valid execution of the code in T, and therefore with speculation. We indicate the addresses of arrays `A` and `B` in the source and target heaps with n_A and n_B respectively and the value stored at `A[i]` with v_A^i.

$$t_{ns} = \mathtt{call\ get}\ 0?^{S} \cdot \mathtt{if}(0)^{S} \cdot \mathtt{read}(n_A)^{S} \cdot \mathtt{read}(n_B + v_A^0)^{S} \cdot \mathtt{ret!}^{S}$$

$$\begin{aligned}t_{sp} = {} & \mathtt{call\ get}\ 8?^{S} \cdot \mathtt{if}(1)^{S} \cdot \mathtt{read}(n_A + 8)^{S} \cdot\\ & \mathtt{read}(n_B + v_A^8)^{U} \cdot \mathtt{rlb}^{S} \cdot \mathtt{ret!}^{S}\end{aligned}$$

In the two traces, the function is called with different parameters. Specifically, the parameter is out-of-bound in t_{sp} thereby resulting in speculatively-executed instructions. The key difference between the traces is that while all actions in t_{ns} are S, there is a U action in t_{sp} (which speculatively leaks the unsafe value $A[8]$ from the private heap). $\boxdot$

I. Weak Languages L^- and T^-

We are now ready to introduce the weak languages L^- and T^-, which we use to study weak security definitions. These languages differ from L and T in two aspects:

1) Following [28], non-speculatively reading from the private heap produces an action $\mathtt{read}(n \mapsto v)$ that contains the read value v as well as the accessed memory address n. Speculative reads, instead, produce actions $\mathtt{read}(n)$ as before.

2) For taint-tracking, we replace Rule T-read-prv with the one below that taints the read variable with the glb of the taints of pc and read value ($\sigma' \sqcap \sigma_{pc}$) instead of U.

$$\frac{B \triangleright e \downarrow n : \sigma' \quad n_a = -|n| \quad H_t(n_a) = \sigma'' \quad \sigma = \sigma'' \sqcup \sigma'}{\sigma_{pc}; C, H_t, \overline{B} \cdot B \triangleright let\ x = rd_p\ e\ in\ s \xrightarrow{\sigma \sqcap \sigma_{pc}} C, H_t, \overline{B} \cdot B \cup x \mapsto 0 : \sigma' \sqcap \sigma_{pc} \triangleright s}\ \text{(T-read-prv-weak)}$$

III. Security Definition for Secure Speculation

We now present *semantic* security definitions against speculative leaks. We start by presenting (robust) speculative non-interference (RSNI, Section III-A). Next, we introduce (robust) speculative safety (RSS, Section III-B). These definitions can be applied to programs in the four languages L, T, L⁻, and T⁻. Thus, in the following, we write RSNI(L) and RSS(L) to indicate which language L the definitions are referring to. Since these languages have the same syntax and different semantics, we can study the relationships between RSNI and RSS for weak and strong languages (Section III-C).

A. Robust Speculative Non-Interference

Speculative non-interference is a class of security properties [27, 28] characterizing speculative leaks. Here, we instantiate robust speculative non-interference in our framework.[4] For this, we need to introduce two concepts:

- SNI is parametric in a policy denoting sensitive information. As mentioned in Section II-B, we assume that only the private heap is sensitive. Hence, whole programs W and W' are *low-equivalent*, written $W' =_{\mathrm{L}} W$, if they differ only in their private heaps.
- SNI requires comparing the leakage resulting from non-speculative and speculative instructions. The *non-speculative projection* $t\!\upharpoonright_{nse}$ [27] of a trace t extracts the observations associated only with non-speculatively-executed instructions. We obtain $t\!\upharpoonright_{nse}$ by removing from t all sub-strings enclosed between $\mathtt{if}(v)$ and $\mathtt{rlb}$ observations. We illustrate this using an example: below is $\cdot\!\upharpoonright_{nse}$ applied to $\mathbf{t_{sp}}$ from Example 1.

$$\mathbf{t_{sp}}\!\upharpoonright_{nse} = \mathtt{call\ get\ 8?}^{\mathbf{S}} \cdot \mathtt{if(1)}^{\mathbf{S}} \cdot \mathtt{ret!}^{\mathbf{S}}$$

We are now ready to formalise SNI. A whole program W is SNI if its traces do not leak more than their non-speculative projections. That is, whenever an attacker can distinguish the traces produced by W and a low-equivalent program W', the distinguishing observation must be generated by an instruction that does not result from mis-speculation.

Definition 1 (Speculative Non-Interference (SNI))**.**

$$\vdash W : \mathrm{SNI} \overset{\mathsf{def}}{=} \forall W'.\ \text{ if } W' =_{\mathrm{L}} W \text{ and } Beh(\Omega_0(W))\!\upharpoonright_{nse} = Beh(\Omega_0(W'))\!\upharpoonright_{nse} \text{ then } Beh(\Omega_0(W)) = Beh(\Omega_0(W'))$$

A component P is robustly speculatively non-interferent if it is SNI no matter what valid attacker it is linked to (Definition 2), where an attacker is valid (indicated as $\vdash A : atk$) if it does not define a private heap and if it does not contain instructions to read and write the private heap.

Definition 2 (Robust Speculative Non-Interference (RSNI))**.**

$$\vdash P : \mathrm{RSNI} \overset{\mathsf{def}}{=} \forall A.\ \text{if } \vdash A : atk \text{ then } \vdash A[P] : \mathrm{SNI}$$

Example 2 (Listing 1 is not RSNI in T)**.** Consider the code of Listing 1 (indicated as $\mathbf{P_1}$) and an attacker $\mathbf{A^8}$ that calls function get with 8. Since array A is in the private heap, the low-equivalent program required by Definition 1 is the same $\mathbf{A^8}$ linked with some $\mathbf{P_N}$, which is the same $\mathbf{P_1}$ with some array N with contents different from A in the heap such that A[8]≠N[8]. Whole program $\mathbf{A^8}[\mathbf{P_1}]$ generates trace $\mathbf{t_{sp}}$ from Example 1 while $\mathbf{A^8}[\mathbf{P_N}]$ generates $\mathbf{t'_{sp}}$ below. We indicate the address of array N as $\mathbf{n_N}$ and the content of N[i] as $\mathbf{v^i_N}$. Low-equivalence yields that addresses are the same ($\mathbf{n_A} + \mathbf{8} = \mathbf{n_N} + \mathbf{8}$) but contents are not ($\mathbf{v^8_A} \neq \mathbf{v^8_N}$), and thus B is accessed at different offsets ($\mathbf{n_B} + \mathbf{v^8_A} \neq \mathbf{n_B} + \mathbf{v^8_N}$).

$$\mathbf{t'_{sp}} = \mathtt{call\ get\ 8?}^{\mathbf{S}} \cdot \mathtt{if(1)}^{\mathbf{S}} \cdot \mathtt{read}(\mathbf{n_N}+\mathbf{8})^{\mathbf{S}} \cdot \mathtt{read}(\mathbf{n_B}+\mathbf{v^8_N})^{\mathbf{U}} \cdot \mathtt{rlb}^{\mathbf{S}} \cdot \mathtt{ret!}^{\mathbf{S}}$$

Listing 1 is not RSNI in T since the non-speculative projections of $\mathbf{t'_{sp}}$ and of $\mathbf{t_{sp}}$ are the same (see above) while $\mathbf{t'_{sp}}$ and $\mathbf{t_{sp}}$ are *different* ($\mathtt{read}(\mathbf{n_B}+\mathbf{v^8_A})^{\mathbf{U}} \neq \mathtt{read}(\mathbf{n_B}+\mathbf{v^8_N})^{\mathbf{U}}$). ⊡

B. Robust Speculative Safety

Speculative safety ensures that *whole* programs W generate only safe (S) actions in their traces. As we show in Section III-C, its security guarantees depend on the underlying language (and on its taint-tracking mechanism).

Definition 3 (Speculative Safety (SS))**.**

$$\vdash W : \mathrm{SS} \overset{\mathsf{def}}{=} \forall \overline{\lambda^{\sigma}} \in Beh(W). \forall \alpha^{\sigma} \in \overline{\lambda^{\sigma}}. \sigma \equiv S$$

A component P is RSS if it upholds SS when linked against arbitrary valid attackers (Definition 4).

Definition 4 (Robust Speculative Safety (RSS))**.**

$$\vdash P : \mathrm{RSS} \overset{\mathsf{def}}{=} \forall A.\ \text{if } \vdash A : atk \text{ then } \vdash A[P] : \mathrm{SS}$$

The snippet of Listing 1 is not RSS in T because the attacker that calls get with argument 8 generates trace $\mathbf{t_{sp}}$, which has an unsafe action (Example 1). The same code in L is RSS because it never generates actions tainted as U.

C. Relationships Between Security Definitions

We now illustrate the relationships between the security definitions instantiated for our languages L, T, L⁻, and T⁻.

1) Relationships for L *and* L⁻*:* All programs in L and L⁻ trivially enjoy both speculative non-interference and speculative safety, because L and L⁻ do not speculatively execute instructions and produce traces with only S actions.

[4] We follow SNI's trace-based characterization from [27, Proposition 1].

Theorem 1 (All L and L⁻ programs are secure)**.**

$$\forall \mathsf{P}.\ \vdash \mathsf{P} : \text{RSS}(\mathsf{L}) \text{ and } \vdash \mathsf{P} : \text{RSS}(\mathsf{L}^-)$$
$$\text{and } \vdash \mathsf{P} : \text{RSNI}(\mathsf{L}) \text{ and } \vdash \mathsf{P} : \text{RSNI}(\mathsf{L}^-)$$

2) Relationships for T *and* T⁻*:* The relationships are summarized below in terms of security guarantees and precision.

$$\begin{tikzcd} & \text{least precise} & \text{most precise} \\ \text{most secure} & \text{RSS}(\mathbf{T}) \arrow[r, Rightarrow, "\text{Theorem 2}"] \arrow[d, Rightarrow, "\text{Theorem 4}"'] & \text{RSNI}(\mathbf{T}) \arrow[d, Rightarrow, "\text{Theorem 4}"] \\ \text{least secure} & \text{RSS}(\mathbf{T}^-) \arrow[r, Rightarrow, "\text{Theorem 3}"] & \text{RSNI}(\mathbf{T}^-) \end{tikzcd}$$

Characterization of speculative non-interference: Instantiating RSNI with languages T and T⁻ result in different security guarantees. Specifically, RSNI(T) corresponds to speculative non-interference [27, 28], which ensures the absence of *all* speculative leaks. In contrast, RSNI(T⁻) corresponds to weak speculative non-interference [28], which allows speculative leaks of information that has been retrieved non-speculatively. That is, RSNI(T⁻) ensures the absence only of speculative leaks of speculatively-accessed data.

As shown in [28], strong and weak speculative non-interference (that is, RSNI(T) and RSNI(T⁻)) have different implications for secure programming. In particular, programs that are traditionally constant-time (i.e., constant-time under the non-speculative semantics) and satisfy strong speculative non-interference are also constant-time w.r.t. the speculative semantics. Similarly, programs that are traditionally sandboxed (i.e., do not access out-of-the-sandbox data non-speculatively) and satisfy weak speculative non-interference are also sandboxed w.r.t. the speculative semantics.

Speculative non-interference and speculative safety: As mentioned before, RSNI(T) semantically characterize the absence of speculative leaks. In contrast, RSS(T) is an over-approximation of RSNI(T) whose preservation through compilation is easier to prove than RSNI(T)-preservation.

Theorem 2 (RSS(T) over-approximates RSNI(T))**.**

1) $\forall \mathbf{P}.$ if $\vdash \mathbf{P} : \text{RSS}(\mathbf{T})$ then $\vdash \mathbf{P} : \text{RSNI}(\mathbf{T})$
2) $\exists \mathbf{P}. \vdash \mathbf{P} : \text{RSNI}(\mathbf{T})$ and $\nvdash \mathbf{P} : \text{RSS}(\mathbf{T})$

To understand point 1, observe that RSS(T) ensures that only safe observations are produced by a program P. This, in turn, ensures that no information originating from the private heap is leaked through speculatively-executed instructions in P. Therefore, P satisfies RSNI(T) because everything except the private heap is visible to the attacker, i.e., there are no additional leaks due to speculatively-executed instructions.

To understand point 2, consider function get_nc from Listing 2, which always accesses B[A[y]]. This code is RSNI(T) because any two states that can be distinguished by looking at the traces would also be distinguished by looking at their non-speculative projections, i.e., speculatively-executed instructions do not leak additional information. However, it is not RSS(T) because speculative memory accesses will produce U actions.

```
void get_nc (int y)
  if (y < size) then B[A[y]] *512] else B[A[y]] *512]
```

Listing 2. Code that is RSNI but not RSS.

RSNI(T⁻) and RSS(T⁻) enjoy a relationship similar to RSNI(T) and RSS(T).

Theorem 3 (RSS(T⁻) over-approximates RSNI(T⁻))**.**

1) $\forall \mathbf{P}.$ if $\vdash \mathbf{P} : \text{RSS}(\mathbf{T}^-)$ then $\vdash \mathbf{P} : \text{RSNI}(\mathbf{T}^-)$
2) $\exists \mathbf{P}. \vdash \mathbf{P} : \text{RSNI}(\mathbf{T}^-)$ and $\nvdash \mathbf{P} : \text{RSS}(\mathbf{T}^-)$

Strong variants imply the weak ones: Since RSNI(T) ensures the absence of *all* speculative leaks while RSNI(T⁻) only ensures the absence of *some* of them, any RSNI(T) program is also RSNI(T⁻). Similarly, any RSS(T) program is also RSS(T⁻) since all actions tainted S by T's taint-tracking are tainted S also by T⁻'s taint-tracking.

Theorem 4 (Strong Variants Imply Weak Ones)**.**

$$\forall \mathbf{P}.\ \text{ if } \vdash \mathbf{P} : \text{RSNI}(\mathbf{T}) \text{ then } \vdash \mathbf{P} : \text{RSNI}(\mathbf{T}^-)$$
$$\forall \mathbf{P}.\ \text{ if } \vdash \mathbf{P} : \text{RSS}(\mathbf{T}) \text{ then } \vdash \mathbf{P} : \text{RSS}(\mathbf{T}^-)$$

IV. Compiler Criteria for Spectre Security

In this section, we introduce our secure compilation criteria: *robust speculative safety preservation* (*RSSP*, Section IV-A), which preserves RSS, and *robust speculative non-interference preservation* (*RSNIP*, Section IV-B), which preserves RSNI. We conclude by discussing how compilers can be proven secure or insecure using these criteria (Section IV-C).

As before, criteria can be instantiated using pairs of languages L-T or L⁻-T⁻. Criteria instantiated with the strong languages (say *RSSP*(L,T)) are indicated with a + (that is, $RSSP^+$). Those instantiated with weak languages (say *RSNIP*(L⁻,T⁻)) are indicated with a - (that is, $RSNIP^-$). When we omit the 'sign', we refer to both criteria. For simplicity, we only present the strong criteria (for L-T), weak ones are defined identically (but for L⁻-T⁻).

A. Robust Speculative Safety Preservation

The first criterion (Definition 5) is clear: a compiler preserves RSS if, given a source component that is RSS, the compiled counterpart is also RSS.

Definition 5 ($RSSP^+$)**.**

$$\vdash [\![\cdot]\!] : RSSP^+ \overset{\text{def}}{=} \forall \mathsf{P} \in \mathsf{L}.\ \text{ if } \vdash \mathsf{P} : \text{RSS}(\mathsf{L})$$
$$\text{then } \vdash [\![\mathsf{P}]\!] : \text{RSS}(\mathbf{T})$$

Definition 5 is a "property-ful" criterion since it explicitly refers to the property that the compiler preserves [3, 4] and it clearly states the security implications of a compiler upholding it. Unfortunately, proving a "property-ful" criterion can be fairly complex at times, but fortunately, it is generally possible to turn a "property-ful" definition into an *equivalent* "property-free" one [3, 4, 52]. This is often beneficial because "property-free" criteria come in so-called *backtranslation* form, which have established proof techniques [2, 4, 13, 46, 50, 52].

To state the equivalence of these criteria, we introduce a cross-language relation between traces of the two languages, which specifies when two possibly different traces have the same "meaning". Our property-free security criterion ($RSSC$, Definition 6) states that a compiler is $RSSC$ if for any target-level attacker $\mathbf{A}$ that generates a trace $\overline{\boldsymbol{\lambda}^{\sigma}}$, we can build a source-level attacker A that generates a trace $\overline{\lambda^{\sigma}}$ that is related to $\overline{\boldsymbol{\lambda}^{\sigma}}$. A source trace $\overline{\lambda^{\sigma}}$ and a target trace $\overline{\boldsymbol{\lambda}^{\sigma}}$ are related (denoted with $\overline{\lambda^{\sigma}} \approx \overline{\boldsymbol{\lambda}^{\sigma}}$) if the target trace contains all the actions of the source trace, plus possible interleavings of safe ($\mathbf{S}$) actions (Rules Trace-Relation-Safe and Trace-Relation-Safe-Heap). All other actions must be the same (i.e., $\equiv$, Rules Trace-Relation-Same and Trace-Relation-Same-Heap).

$$\frac{\overline{\lambda^{\sigma}} \approx \overline{\boldsymbol{\lambda}^{\sigma}} \qquad \alpha^{\sigma} \equiv \boldsymbol{\alpha}^{\sigma}}{\overline{\lambda^{\sigma}} \cdot \alpha^{\sigma} \approx \overline{\boldsymbol{\lambda}^{\sigma}} \cdot \boldsymbol{\alpha}^{\sigma}}\ \text{(Trace-Relation-Same)} \qquad \frac{\overline{\lambda^{\sigma}} \approx \overline{\boldsymbol{\lambda}^{\sigma}} \qquad \delta^{\sigma} \equiv \boldsymbol{\delta}^{\sigma}}{\overline{\lambda^{\sigma}} \cdot \delta^{\sigma} \approx \overline{\boldsymbol{\lambda}^{\sigma}} \cdot \boldsymbol{\delta}^{\sigma}}\ \text{(Trace-Relation-Same-Heap)}$$

$$\frac{\overline{\lambda^{\sigma}} \approx \overline{\boldsymbol{\lambda}^{\sigma}}}{\overline{\lambda^{\sigma}} \approx \overline{\boldsymbol{\lambda}^{\sigma}} \cdot \boldsymbol{\alpha}^{\mathbf{S}}}\ \text{(Trace-Relation-Safe)} \qquad \frac{\overline{\lambda^{\sigma}} \approx \overline{\boldsymbol{\lambda}^{\sigma}}}{\overline{\lambda^{\sigma}} \approx \overline{\boldsymbol{\lambda}^{\sigma}} \cdot \boldsymbol{\delta}^{\mathbf{S}}}\ \text{(Trace-Relation-Safe-Heap)}$$

We are now ready to formalise $RSSC$, which is equivalent to $RSSP$ (Theorem 5). Importantly, this result implies that our choice for the trace relation is correct; a relation that is too strong or too weak would not let us prove this equivalence.

Definition 6 ($RSSC^{+}$)**.**

$$\vdash [\![\cdot]\!] : RSSC^{+} \overset{\text{def}}{=} \forall \mathsf{P} \in \mathsf{L}, \mathbf{A}, \overline{\boldsymbol{\lambda}^{\sigma}}.\ \text{if } \mathbf{Beh}(\mathbf{A}\,[[\![\mathsf{P}]\!]]) = \overline{\boldsymbol{\lambda}^{\sigma}}$$
$$\text{then } \exists \mathsf{A}, \overline{\lambda^{\sigma}}. \mathsf{Beh}(\mathsf{A}\,[\mathsf{P}]) = \overline{\lambda^{\sigma}} \text{ and } \overline{\lambda^{\sigma}} \approx \overline{\boldsymbol{\lambda}^{\sigma}}$$

Theorem 5 ($RSSP$ and $RSSC$ are equivalent)**.**

$$\forall [\![\cdot]\!]. \vdash [\![\cdot]\!] : RSSP^{+} \iff \vdash [\![\cdot]\!] : RSSC^{+}$$
$$\forall [\![\cdot]\!]. \vdash [\![\cdot]\!] : RSSP^{-} \iff \vdash [\![\cdot]\!] : RSSC^{-}$$

Definition 6 requires providing an existentially-quantified source attacker A. The general proof technique for these criteria is called *backtranslation* [4, 51], and it can either be attacker-based [13, 21, 46] or trace-based [2, 50, 52]. The distinction tells us what quantified element we can use to build the source attacker A, either the target attacker $\mathbf{A}$ or the trace $\overline{\boldsymbol{\lambda}^{\sigma}}$ respectively. In our proofs, we will use an attacker-based backtranslation.

B. Robust Speculative Non-Interference Preservation

Here, we only present a property-ful criterion for the preservation of RSNI (Definition 7). The reason is that we only directly prove that compilers do *not* attain $RSNIP$. This kind of proof is simple already (Corollary 1), and we do not need a property-free criterion.

Definition 7 ($RSNIP^{+}$)**.**

$$\vdash [\![\cdot]\!] : RSNIP^{+} \overset{\text{def}}{=} \forall \mathsf{P} \in \mathsf{L}.\ \text{if } \vdash \mathsf{P} : \mathrm{RSNI}(\mathsf{L})$$
$$\text{then } \vdash [\![\mathsf{P}]\!] : \mathrm{RSNI}(\mathbf{T})$$

Corollary 1 ($\nvdash [\![\cdot]\!] : RSNIP^{+}$)**.**

$$\nvdash [\![\cdot]\!] : RSNIP^{+} \overset{\text{def}}{=} \exists \mathsf{P} \in \mathsf{L}.\ \vdash \mathsf{P} : \mathrm{RSNI}(\mathsf{L})$$
$$\text{and } \nvdash [\![\mathsf{P}]\!] : \mathrm{RSNI}(\mathbf{T})$$

Here, the second clause gets unfolded to the following; recall that low-equivalent programs simply differ in their private heap, so $\mathbf{A}'\,[[\![\mathsf{P}']\!]]$ is the same attacker $\mathbf{A}$ linked with the same program with a different private heap.

$$\text{t.s.} : \exists \mathbf{A}. \vdash \mathbf{A} : atk \text{ and given } \mathbf{A}'\,[[\![\mathsf{P}']\!]] =_{\mathrm{L}} \mathbf{A}\,[[\![\mathsf{P}]\!]]$$
$$\text{we have } \mathbf{Beh}(\boldsymbol{\Omega}_0\,(\mathbf{A}\,[[\![\mathsf{P}]\!]]))\restriction_{nse} = \mathbf{Beh}(\boldsymbol{\Omega}_0\,(\mathbf{A}'\,[[\![\mathsf{P}']\!]]))\restriction_{nse}$$
$$\text{and } \mathbf{Beh}(\boldsymbol{\Omega}_0\,(\mathbf{A}\,[[\![\mathsf{P}]\!]])) \neq \mathbf{Beh}(\boldsymbol{\Omega}_0\,(\mathbf{A}'\,[[\![\mathsf{P}']\!]]))$$

Note that finding the existentially-quantified program (and attacker) that demonstrate insecurity of a countermeasure may be hard. Fortunately, some failed attempts at proving $RSSC$ can provide hints for how to do this; we provide more insights after discussing proof techniques in Appendix B. □

C. A Methodology for Provably-(In)Secure Countermeasures

Recall from Section III-A that RSNI in the target language is the property we should have for components that are compiled with secure countermeasures. Conversely, components that are compiled with insecure countermeasures *cannot attain* RSNI in the target. These intuitive ideas are represented as two chains of implications in Figure 1. The first one lists the assumptions (black dashed lines) and logical steps (theorem-annotated implications) to conclude compiler security while the second one lists assumptions and logical steps for compiler insecurity. For simplicity, the figure focuses on security definitions and compiler criteria for L and $\mathbf{T}$. There are similar chains of implications instantiated with languages L^{-} and $\mathbf{T}^{-}$ that use Theorem 3 instead of Theorem 2.

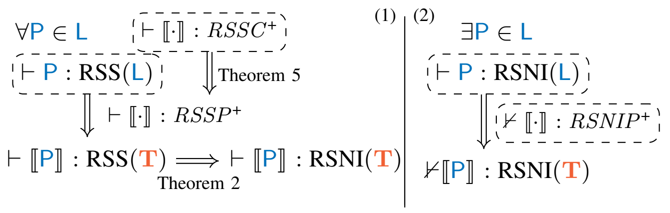

Figure 1. Our methodology to prove security (1) and insecurity (2) for compiler countermeasures against speculative leaks

To show security (1), we need to prove that any compiled component is RSNI in the target language. By Theorem 2, it suffices to show that any compiled component is RSS in the target. This can be obtained by (i) an $RSSP^{+}$ compiler so long as (ii) any P is RSS in the source. By Theorem 5, for point (i) it is sufficient to show that the compiler is $RSSC^{+}$. Point (ii) holds for any P; see Theorem 1.

To show insecurity (2), we need to prove that there exists a compiled component that is *not* RSNI in the target language. For this, we show (A) that the compiler is not $RSNIP^{+}$ given that (B) the source component P was RSNI in the source. To show (A), we follow Corollary 1, whereas point (B) holds for any source component P (Theorem 1).

Our security criteria, instantiated for the strong (L-$\mathbf{T}$) and weak (L^{-}-$\mathbf{T}^{-}$) languages, provide a way of characterizing the security guarantees of any countermeasure $[\![\cdot]\!]$, which we do next. In particular, showing that $[\![\cdot]\!]$ is $RSSC^{+}$ ensures that compiled code has no speculative leak. Similarly, showing that $[\![\cdot]\!]$ is RSSC^{-} (*and not* RSNIP^{+}) ensures that compiled code does not leak information about speculatively-accessed data, i.e., it would prevent leaks like the Spectre v1 attack. Finally, showing that $[\![\cdot]\!]$ is not RSNIP^{-} implies that compiled code leaks speculatively accessed data, like in Spectre attacks.

V. Countermeasures Security and Insecurity

In this section, we precisely characterise the security guarantees of the two main Spectre v1 countermeasures implemented by compiler vendors: insertion of speculation barriers and speculative load hardening. We show that the Microsoft Visual C++ [48] (MSVC) compiler implements the first countermeasure in an insecure way, i.e., MSVC violates RSNIP^{-} and produces programs that are vulnerable to Spectre attacks (Section V-A). We also prove that the Intel C++ compiler [33] (ICC) implements the same countermeasure securely, that is, ICC is RSSP^{+} and prevents all speculative leaks (Section V-B). Finally, we study the security of SLH (Section V-C). We prove that SLH prevents only leaks involving speculatively-accessed data, i.e., SLH is RSSP^{-} but it violates RSNIP^{+}. While this is sufficient for preventing Spectre-style attacks, compiled programs may still speculatively leak data retrieved non-speculatively, which might result in breaking properties like constant-time (see [28]). Additionally, we provide a modification to SLH that prevents all speculative leaks, i.e., it is RSSP^{+}. SLH also has a non-interprocedural variant but we prove that it is completely insecure, i.e., it violates RSNIP^{-}. We provide a high-level proof overview in Section V-D; full proofs are in supplementary material.

A. MSVC is insecure

Inserting speculation barriers—the **lfence** x86 instruction—after branch instructions is a simple countermeasure against Spectre v1 [31, 33, 48]. This instruction stops speculative execution at the price of significant performance overhead.

MSVC implements a countermeasure that tries to minimize the number of **lfence**s by selectively determining which branches to patch.[5] However, MSVC fails in inserting some necessary **lfence**s, thereby producing insecure code that is not RSNI($\mathbf{T}^{-}$). To show this, we follow Corollary 1 and provide a program that is RSNI(L^{-}) and its compilation is not RSNI($\mathbf{T}^{-}$). The program we consider, which is RSNI(L^{-}) (Theorem 1), is given in Listing 3.

```
1 void get (int y)
2   if (y < size) then
3     if (A[y] == 0) then
4       temp = B[0];
```

Listing 3. A variant of the classic Spectre v1 snippet (Example 10 from [36]).

Its compiled counterpart does not contain **lfence**s and it speculatively leaks whether A[y] is 0 through the branch statement in line 3, i.e., it violates RSNI($\mathbf{T}^{-}$). We refer to [27, 36] for additional examples of MSVC's insecurity.

[5]The countermeasure can be activated with the `/Qspectre` flag.

B. ICC is secure

The Intel C++ compiler also implements a countermeasure that inserts **lfence**s after each branch instruction.[6] We model this countermeasure with $[\![\cdot]\!]^{f}$, a homomorphic compiler that takes a component in L and translates all of its subparts to $\mathbf{T}$. Its key feature is inserting an **lfence** at the beginning of every **then** and **else** branch of compiled code.

$$[\![\mathsf{ifz\ e\ then\ s\ else\ s'}]\!]^{f} = \mathbf{ifz}\ [\![\mathsf{e}]\!]^{f}\ \mathbf{then}\ \{\mathbf{lfence};\ [\![\mathsf{s}]\!]^{f}\}$$
$$\mathbf{else}\ \{\mathbf{lfence};\ [\![\mathsf{s'}]\!]^{f}\}$$

It should come at no surprise that $[\![\cdot]\!]^{f}$ is RSSC^{+} (Theorem 6). In $\mathbf{T}$, the only source of speculation are branches (Rule E-$\mathbf{T}$-speculate-if) but any branch, whether it evaluates to true or false, will execute an **lfence** (Rule E-$\mathbf{T}$-speculate-lfence), triggering a rollback (Rule E-$\mathbf{T}$-speculate-rollback). So, compiled code performs no action during speculation. It can only perform actions when the pc is tainted as $\mathbf{S}$, which makes all actions $\mathbf{S}$. These actions are easy to relate to their source-level counterparts since they are generated according to the non-speculative semantics.

Theorem 6 (ICC is secure)**.** $\vdash [\![\cdot]\!]^{f} : \mathit{RSSC}^{+}$

C. Speculative Load Hardening

Clang implements a countermeasure called speculative load hardening [16] (SLH) that works as follows:[7]

- Compiled code keeps track of a *predicate bit* that records whether the processor is mis-speculating (predicate bit set to $\mathbf{1}$) or not (predicate bit set to $\mathbf{0}$). This is done by replicating the behaviour of all branch instructions using branch-less **cmov** instructions, which do not trigger speculation. SLH-compiled code tracks the predicate bit inter-procedurally by storing it on the most-significant bits of the stack pointer register, which are always unused. We remark that whenever all speculative transactions have been rolled back, the predicate bit is reset to $\mathbf{0}$ by the rollback capabilities of the processor.
- Compiled code uses the predicate bit to initialise a mask whose usage is detailed below. At the beginning of a function, SLH-compiled code retrieves the predicate bit from the stack and uses it to initialize a mask either to $\mathbf{0xF..F}$ if predicate bit is $\mathbf{1}$ or to $\mathbf{0x0..0}$ otherwise. During the computation, SLH-compiled code uses **cmov** instructions to conditionally update the mask and preserve the invariant that $\mathit{mask} = \mathbf{0xF..F}$ if code is mis-speculating and $\mathit{mask} = \mathbf{0x0..0}$ otherwise. Before returning from a function, SLH-compiled code pushes the most-significant bit of the current mask to the stack; thereby preserving the predicate bit.
- All inputs to control-flow and store instructions are hardened by masking their values with mask (i.e., by **or**-ing their value with mask). That is, whenever code is mis-speculating (i.e., $\mathit{mask} = \mathbf{0xF..F}$) the inputs to control-flow and store statements are effectively "F-ed" to $\mathbf{0xF..F}$, otherwise they

[6]The countermeasure can be activated with the `-mconditional-branch=all-fix` flag.

[7] The countermeasure has been available from Clang v7.0.0 and it can be activated using the `-mllvm -x86-speculative-load-hardening` flag.

are left unchanged. This effectively prevents speculative leaks through control-flow and store statements.

- The outputs of memory loads instructions are hardened by $\mathbf{or}$-ing their value with $mask$. So, when code is mis-speculating, the result of load instructions is "F-ed" to $\mathbf{0xF..F}$. This prevents leaks of speculatively-accessed memory locations. Inputs to load instructions, however, are *not* masked.

In the following, we analyse the security guarantees of SLH.

1) SLH is not $RSNIP^{+}$: We start by showing that SLH is not $RSNIP^{+}$, that is, it does not preserve (strong) speculative non-interference. Following Corollary 1, we do this by providing a program that is RSNI(L) and that is compiled to a program that is not RSNI($\mathbf{T}$). Consider the program in Listing 4, which differs from Listing 1 in that the first memory access is performed non-speculatively (line 2).

```
1 void get (int y)
2   x = A[y];
3   if (y < size) then
4     temp = B[x];
```

Listing 4. Another variant of the classic Spectre v1 snippet.

SLH hardens the value of A[y] using the mask retrieved from the stack pointer. When the get function is invoked non-speculatively, the mask is set to $\mathbf{0x0..0}$ and A[y] is not masked. Thus, speculatively executing the load in (the compiled counterpart of) line 4 leaks the value of A[y], which might differ for low-equivalent states, and violates RSNI($\mathbf{T}$).

2) SLH is $RSSC^{-}$: We now show that SLH is $RSSC^{-}$, that is, it prevents leaks of speculatively-accessed data.

We formalise SLH using the $[\![\cdot]\!]^{s}$ compiler, whose most interesting cases are given in the top of Figure 2. The compiler takes components in L^{-} and outputs compiled code in $\mathbf{T}^{-}$. The compiler keeps track of the predicate bit in a cross-procedural way, masks inputs to control-flow and store instructions, and masks outputs of load instructions as described before.

Since the stack pointer is not accessible from an attacker residing in another process, $[\![\cdot]\!]^{s}$ tracks the predicate bit in the first location of the private heap. So location $\mathbf{-1}$ is initialised to $\mathbf{1}$ (false) and updated to $\mathbf{0}$ whenever we are speculating. Compiled code must update the predicate bit right after the $\mathbf{then}$ and $\mathbf{else}$ branches (statements $\mathbf{-1} :=_{\mathbf{p}} \cdots$). Since location $\mathbf{-1}$ is reserved for the predicate bit, all private memory accesses as well as the private heap are shifted by 1.

Several statements may leak information to the attacker: calling attacker functions, reading and writing the public and private heap, and branching. For function calls, memory writes, and branch instructions, $[\![\cdot]\!]^{s}$ masks these statement's input. That is, we evaluate the sub-expressions used in those statements and store them in auxiliary variables (called $\mathbf{x_f}$). Then, we look up the predicate bit (via statement $\mathbf{let\ pr = rd_p\ {-1}\ in}\ \cdots$) and store it in variable $\mathbf{pr}$. Finally, using the conditional assignment, we set the result of those expressions to $\mathbf{0}$ if the predicate bit is $\mathbf{0}$ (true). In contrast, for memory reads, $[\![\cdot]\!]^{s}$ masks the output of these statement based on the predicate bit stored in $\mathbf{pr}$.

Theorem 7 (SLH is weakly-secure)**.** $\vdash [\![\cdot]\!]^{s} : RSSC^{-}$

$[\![\cdot]\!]^{s}$ is $RSSC^{-}$ for two reasons. First, location $\mathbf{-1}$ (and thus variable $\mathbf{pr}$ where its contents are loaded) always correctly tracks whether speculation is ongoing or not. This is true since location $\mathbf{-1}$ and $\mathbf{pr}$ cannot be tampered by the attacker, the compiler initializes $\mathbf{-1}$ correctly, and the assignments right after the branches correctly update location $\mathbf{-1}$ (via the negation of the guard $\mathbf{x_f}$). Second, whenever speculation is happening, the result of load operations is set to a constant $\mathbf{0}$ whose taint is $\mathbf{S}$. So, computations happening during speculation either depend on data loaded non-speculatively, which are tainted as $\mathbf{S}$ thanks to the taint-tracking of $\mathbf{T}^{-}$, or on masked values, which are also tainted $\mathbf{S}$. Therefore, labels generated when speculating will be tainted $\mathbf{S}$, thereby satisfying RSS($\mathbf{T}^{-}$).

3) Making SLH More Secure: We now show how to modify SLH to prevent *all* speculative leaks. We do so by introducing *strong SLH* (SSLH for short) that differs from standard SLH in that it masks the input (rather than the output) of memory read operations. We model SSLH using the $[\![\cdot]\!]^{ss}$ compiler that takes components in L and outputs compiled code in $\mathbf{T}$. $[\![\cdot]\!]^{ss}$ differs from $[\![\cdot]\!]^{s}$ in how memory reads are compiled, as shown in Figure 2. The compiler masks the input of memory loads by evaluating the sub-expressions and storing them in auxiliary variables (called $\mathbf{x_f}$), retrieving the predicate bit and storing it in variable $\mathbf{pr}$, conditionally masking the value of $\mathbf{x_f}$, and, finally, performing the memory access using $\mathbf{x_f}$ as address.

Theorem 8 (SSLH is secure)**.** $\vdash [\![\cdot]\!]^{ss} : RSSC^{+}$

$[\![\cdot]\!]^{ss}$ satisfies $RSSC^{+}$ (Theorem 8) for two reasons. First, as discussed above, the compiler correctly tracks whether speculation is ongoing. Second, whenever speculation is happening, the result of any possibly-leaking expression is set to a constant $\mathbf{0}$ whose taint is $\mathbf{S}$. That is, labels during speculation are tainted as $\mathbf{S}$, and RSS($\mathbf{T}$) holds.

4) Non-interprocedural SLH is insecure: We conclude by showing that the non-interprocedural variant of SLH, where the predicate bit is set to $\mathbf{0}$ at the beginning of each function, is insecure.[8] Consider the program in Listing 5 that splits the memory accesses of A and B of the classical Spectre v1 snippet across functions get and get_2.

```
1 void get (int y)
2   x = A[y];
3   if (y < size) then get_2 (x);
4
5 void get_2 (int x) temp = B[x];
```

Listing 5. Inter-procedural variant of the Spectre v1 snippet [43].

Once compiled, get starts the speculative execution (line 3), then the compiled code corresponding to get_2 is executed speculatively. However, the predicate bit of get_2 is set to $\mathbf{0}$ upon calling the function. Hence, the memory access corresponding to B[x] is not masked leading to the leak of x (which contains A[y]). Hence, the target program violates RSNI($\mathbf{T}^{-}$).

D. How to Prove RSSC

We now illustrate the proof technique used to prove SLH-related countermeasures secure. Recall from Section IV that

[8]Non-interprocedural SLH can be activated with the `-mllvm -x86-speculative-load-hardening -mllvm -x86-slh-ip=false` flag.

$$\llbracket \mathsf{H}; \overline{\mathsf{F}}; \overline{\mathsf{I}} \rrbracket^{s} = \llbracket \mathsf{H} \rrbracket^{s} \cup (-1 \mapsto 1 : \mathbf{S}); \llbracket \overline{\mathsf{F}} \rrbracket^{s}; \llbracket \overline{\mathsf{I}} \rrbracket^{s}$$

$$\llbracket \mathsf{H}, -\mathsf{n} \mapsto \mathsf{v} : \mathsf{U} \rrbracket^{s} = \llbracket \mathsf{H} \rrbracket^{s}, -\llbracket \mathsf{n} \rrbracket^{s} - 1 \mapsto \llbracket \mathsf{v} \rrbracket^{s} : \mathbf{U}$$

$$\llbracket \mathsf{e} :=_{\mathsf{p}} \mathsf{e}' \rrbracket^{s} = \left| \begin{array}{l} \mathbf{let}\ \mathbf{x_f} = \llbracket \mathsf{e} \rrbracket^{s} + \mathbf{1}\ \mathbf{in}\ \mathbf{let}\ \mathbf{x_f'} = \llbracket \mathsf{e}' \rrbracket^{s}\ \mathbf{in}\ \mathbf{let}\ \mathbf{pr} = \mathbf{rd_p}\ \mathbf{-1}\ \mathbf{in} \\ \quad \mathbf{let}\ \mathbf{x_f} = \mathbf{0}\ (\mathbf{if}\ \mathbf{pr})\ \mathbf{in}\ \mathbf{let}\ \mathbf{x_f'} = \mathbf{0}\ (\mathbf{if}\ \mathbf{pr})\ \mathbf{in}\ \mathbf{x_f} :=_{\mathbf{p}} \mathbf{x_f'} \end{array} \right.$$

$$\llbracket \mathsf{ifz}\ \mathsf{e}\ \mathsf{then}\ \mathsf{s}\ \mathsf{else}\ \mathsf{s}' \rrbracket^{s} = \left| \begin{array}{l} \mathbf{let}\ \mathbf{x_f} = \llbracket \mathsf{e} \rrbracket^{s}\ \mathbf{in}\ \mathbf{let}\ \mathbf{pr} = \mathbf{rd_p}\ \mathbf{-1}\ \mathbf{in}\ \mathbf{let}\ \mathbf{x_f} = \mathbf{0}\ (\mathbf{if}\ \mathbf{pr})\ \mathbf{in} \\ \quad \mathbf{ifz}\ \mathbf{x_f}\ \mathbf{then}\ \mathbf{-1} :=_{\mathbf{p}} \mathbf{pr} \vee \neg \mathbf{x_f}; \llbracket \mathsf{s} \rrbracket^{s}\ \mathbf{else}\ \mathbf{-1} :=_{\mathbf{p}} \mathbf{pr} \vee \mathbf{x_f}; \llbracket \mathsf{s}' \rrbracket^{s} \end{array} \right.$$

$$\llbracket \mathsf{let}\ \mathsf{x} = \mathsf{rd_p}\ \mathsf{e}\ \mathsf{in}\ \mathsf{s} \rrbracket^{s} = \left| \mathbf{let}\ \mathbf{x_f} = \llbracket \mathsf{e} \rrbracket^{s} + \mathbf{1}\ \mathbf{in}\ \mathbf{let}\ \mathbf{pr} = \mathbf{rd_p}\ \mathbf{-1}\ \mathbf{in}\ \mathbf{let}\ \mathbf{x} = \mathbf{rd_p}\ \mathbf{x_f}\ \mathbf{in}\ \mathbf{let}\ \mathbf{x} = \mathbf{0}\ (\mathbf{if}\ \mathbf{pr})\ \mathbf{in}\ \llbracket \mathsf{s} \rrbracket^{s} \right.$$

$$\llbracket \mathsf{let}\ \mathsf{x} = \mathsf{rd_p}\ \mathsf{e}\ \mathsf{in}\ \mathsf{s} \rrbracket^{ss} = \left| \mathbf{let}\ \mathbf{x_f} = \llbracket \mathsf{e} \rrbracket^{ss} + \mathbf{1}\ \mathbf{in}\ \mathbf{let}\ \mathbf{pr} = \mathbf{rd_p}\ \mathbf{-1}\ \mathbf{in}\ \mathbf{let}\ \mathbf{x_f} = \mathbf{0}\ (\mathbf{if}\ \mathbf{pr})\ \mathbf{in}\ \mathbf{let}\ \mathbf{x} = \mathbf{rd_p}\ \mathbf{x_f}\ \mathbf{in}\ \llbracket \mathsf{s} \rrbracket^{ss} \right.$$

Figure 2. Key bits of the SLH compiler $\llbracket \cdot \rrbracket^{s}$ (above). The SSLH compiler (below) differs from $\llbracket \cdot \rrbracket^{ss}$ in the compilation of memory reads

to prove that a compiler is *RSSC* we *backtranslate* a target attacker ($\mathbf{A}$) to create a source attacker ($\mathsf{A} = \langle\langle \mathbf{A} \rangle\rangle$) so that the two behave the same (i.e., they produce traces related by the relation of Section IV). Our backtranslation function ($\langle\langle \cdot \rangle\rangle$), which is the same for all proofs, homomorphically translates target heaps, functions, statements etc. into source ones.

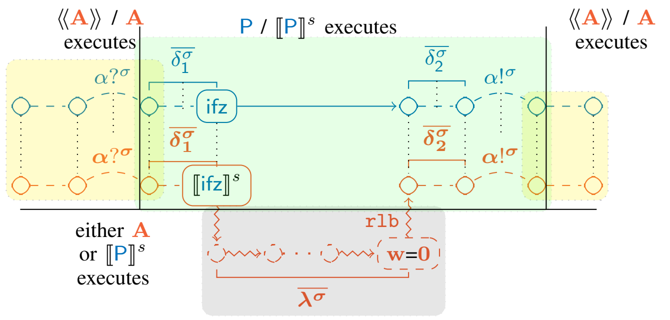

Figure 3. A diagram depicting the proof that countermeasure $\llbracket \cdot \rrbracket^{s}$ is *RSSC*-.

We depict our proof approach in Figure 3. There, circles and contoured statements represent source and target states. A black dotted connection between source and target states indicates that they are related; dashed target states are not related to any source state. In our setup, execution happens either on the attacker side or on the component side, coloured connections between same-colour states represent reductions.

To prove that source and target traces are related, we set up a cross-language relation between source and target states and prove that reductions both preserve this relation and generate related traces. The state relation we use is strong: a source state is related to a target one if the latter is a singleton stack and all the sub-part of the state are identical, i.e., heaps bind the same locations to the same values and bindings bind the same variables to the same values. To reason about attacker reductions, we use a lock-step simulation: we show that starting from related states, if $\mathbf{A}$ does a step, then $\langle\langle \mathbf{A} \rangle\rangle$ does the same step and ends up in related states (yellow areas). To reason about component reductions, we adapt a reasoning from compiler correctness results [10, 39]. That is, if s steps and emits a trace, then $\llbracket \mathsf{s} \rrbracket^{s}$ does one or more steps and emits a trace such that both ending states and traces are related (green areas, related traces are connected by black-dotted lines). This proof is straightforward except for the compilation of ifz since it triggers speculation in $\mathbf{T}$ (grey area). After $\llbracket \mathsf{ifz} \rrbracket^{s}$ is executed, speculation starts and the cross-language state relation is temporarily broken (the stack of target states is not a singleton, so the cross-language state relation cannot hold). Speculative execution continues for $\mathbf{w}$ steps in both attacker and compiled code and generating a trace $\overline{\boldsymbol{\lambda}^{\sigma}}$. We then prove that $\overline{\boldsymbol{\lambda}^{\sigma}}$ is related to the empty source trace because all actions in $\overline{\boldsymbol{\lambda}^{\sigma}}$ are tainted $\mathbf{S}$, and so they do not leak. This fact follows from proving that while speculating, bindings always contain $\mathbf{S}$ values and therefore any generated action is $\mathbf{S}$. In turn, this follows from proving that $\mathbf{pr}$ correctly captures if speculation is ongoing or not and that the mask is $\mathbf{S}$. As mentioned, both of these hold for $\llbracket \cdot \rrbracket^{s}$ and $\llbracket \cdot \rrbracket^{ss}$, so they are secure.

The compiler $\llbracket \cdot \rrbracket^{f}$ can be proved secure in a simpler way since speculative reductions immediately trigger an $\mathbf{lfence}$, which rolls the speculation back (the speculation window $\mathbf{w}$ is $\mathbf{0}$) reinstating the cross-language state relation right away.

VI. Discussion

A. *Scope of the model*

Lifting our security analysis to actual microarchitectures is only valid to the extent that our attacker model and speculative semantics capture the target system.

Our attacker observes the location of memory accesses and the outcome of control-flow statements during execution. This attacker model offers a good tradeoff between precision and simplicity [8, 45], and it has proven to capture interesting microarchitectural leaks, like those resulting from caches and port contention. Other classes of microarchitectural leaks, like those resulting from internal buffers [63] or hardware prefetchers [26], might not be captured by our model.

We also assume that attackers cannot access the private heap. This can be achieved, for instance, by running attacker and component in separate processes and leveraging OS-level memory protection.

Finally, our target languages are adequate to reason about Spectre v1-style attacks. The language semantics, however, ignores out-of-order execution as well as other sources of speculative execution, like speculation over indirect jumps, that are exploited by other Spectre variants, as we discuss next.

B. *Beyond Spectre v1*

Spectre v1 (also called Spectre-PHT) is just one of the (many) variants of Spectre attacks. After a brief recount of the variants and of their existing compiler countermeasures, we discuss how the proof techniques applied for v1 can be applied for countermeasures against other Spectre variants.

- Spectre BTB [37] exploits speculation over indirect jump instructions. The *retpoline* compiler-level countermeasure [32] replaces indirect jumps with return-based trampoline that leads to effectively dead code. As a result, the speculated jump executes no code and thus cannot leak anything.
- Spectre-RSB [42], in contrast, exploits speculation over return addresses (through `ret` instructions). To prevent it, Intel deployed a microcode update [32] that renders *retpoline* a valid countermeasure also against Spectre-RSB [15].
- Spectre-STL [30] exploits speculation over data dependencies between in-flight store and load operations. To mitigate it, ARM introduced a dedicated `SSBB` speculation barrier to prevent store bypasses that could be injected by compilers.

To reason about these Spectre variants and their countermeasures, we first have to extend the speculative semantics of **T**. This can be done similarly to other semantics [9, 17, 44, 64].

We believe that the new countermeasures are $RSSP$ and their proofs should follow the overview in Section V-D. Specifically, proofs for *retpoline* would follow the approach of Figure 3 since speculative execution gets diverted to code that does not produce observations (we provide an in-depth discussion on *retpoline* in Appendix C). In contrast, reasoning about `SSBB` would be similar to reasoning about $[\![\cdot]\!]^f$ since `SSBB`s instructions act as speculation barriers.

VII. Related Work

Speculative execution attacks: Many attacks analogous to Spectre [35, 37] have been discovered; they differ in the exploited speculation sources [30, 38, 41], the covert channels [57, 59, 62], or the target platforms [19]. We refer the reader to [15] for a survey of attacks and countermeasures.

Speculative semantics: These semantics model the effects of speculatively-executed instructions. Several semantics [9, 17, 28, 44, 64] explicitly model microarchitectural details like multiple pipeline stages, reorder buffers, caches, and predictors. These semantics are significantly more complex than ours (which is inspired by [27]), and they would lead to much harder proofs.

Security definition against Spectre attacks: SNI [27] has been used as security definition against speculative leaks also by [9, 28, 64]. Cheang *et al.* [18] propose *trace property-dependent observational determinism*, a property similar to SNI. Cauligi *et al.* [17] present speculative constant-time (SCT), i.e., constant-time w.r.t. the speculative semantics. Differently from SNI, SCT captures leaks under the non-speculative *and* the speculative semantics, and it is inadequate for reasoning about compiler-level countermeasures that only modify a program's speculative behaviour. More generally, Guarnieri *et al.* [28] presents a secure programming framework that subsumes both SNI and SCT.

Compiler-level countermeasures for Spectre v1: Apart from the insertion of speculation barriers [5, 31] and SLH [16, 47], few countermeasures for Spectre v1 exist. Replacing branch instructions with branchless computations (using `cmov` and bit masking) is effective [53] but not generally applicable. oo7 [65] is a tool that automatically patches speculative leaks by injecting speculation barriers. However, oo7 misses some speculative leaks [27] and violates $RSNIP^{-}$.

Blade [64] is a compiler-level countermeasure that aims at optimising compiled code performance. It finds the minimal set of variables that need to be masked in order to eliminate paths between sources (i.e., speculative memory reads) and sinks (i.e., operations resulting in microarchitectural side-effects). Since Blade employs an SLH-style mechanism for preventing leaks, we believe Blade to satisfy $RSSC$. Its security proof should follow the same insights of Figure 3.

Secure compilation: $RSSC$ and $RSSP$ are instantiations of robustly-safe compilation [2, 3, 4, 52]. Like [3, 52], we relate source and target traces using a cross-language relation; however, our target language has a speculative semantics.

Fully abstract compilation (FAC) is a widely used secure compilation criterion [24, 34, 50, 51, 55, 58]. FAC compilers must preserve (and reflect) observational equivalence of source programs in their compiled counterparts [1, 51]. While FAC has been used to reason about microarchitectural side-effects [14], it is unclear whether FAC is well-suited for reasoning about speculative leaks as it would require explicitly modelling microarchitectural components that are modified speculatively (like caches).

Constant-time-preserving compilation ($CTPC$) has been used to show that compilers preserve constant-time [7, 10, 11]. Similarly to $RSNIP$, proving $CTPC$ requires proving the preservation of a hypersafety property, which is more challenging than preserving safety properties like RSS. Additionally, $CTPC$ has been devised for whole programs only (like SNI), and it cannot be used to reason about countermeasures like SLH that do not preserve constant-time.

Verifying Hypersafety as Safety Properties: Verifying if a program satisfies a 2-hypersafety property [20] (like RSNI) is notoriously challenging. Approaches for this include taint-tracking [6, 56] (which over-approximates the 2-hypersafety property with a safety property), secure multi-execution [22] (which runs the code twice in parallel) and self-composition [12, 61] (which runs the code twice sequentially). Our secure compilation criterion leverages taint-tracking (RSS); we leave investigating criteria based on the other approaches as future work.

VIII. Conclusion

The paper presented a comprehensive and precise characterization of the security guarantees of compiler countermeasures against Spectre v1, as well as the first proofs of security for such countermeasures. For this, it introduced SS, a safety property implying the absence of (classes of) speculative leaks. SS provides precise security guarantees in that it can be instantiated to over-approximate both strong [27] and weak [28] SNI, and it is tailored towards simplifying secure compilation proofs. The paper also formalised secure compilation criteria that state how to preserve SS and SNI properties robustly through compilation, which are at the basis of security proofs.

References

[1] Martín Abadi. Protection in programming-language translations. In *ICALP'98*, pages 868–883, 1998.

[2] Carmine Abate, Arthur Azevedo de Amorim, Roberto Blanco, Ana Nora Evans, Guglielmo Fachini, Catalin Hritcu, Théo Laurent, Benjamin C. Pierce, Marco Stronati, and Andrew Tolmach. When good components go bad: Formally secure compilation despite dynamic compromise. CCS '18, 2018.

[3] Carmine Abate, Roberto Blanco, Stefan Ciobaca, Alexandre Durier, Deepak Garg, Cătălin Hriţcu, Marco Patrignani, , Eric Tanter, and Jérémy Thibault. Trace-relating compiler correctness and secure compilation. In *ESOP 2020*, 2020.

[4] Carmine Abate, Roberto Blanco, Deepak Garg, Cătălin Hriţcu, Marco Patrignani, and Jérémy Thibault. Journey beyond full abstraction: Exploring robust property preservation for secure compilation. In *CSF 2019*, 2019.

[5] Advanced Micro Devices, Inc. Software techniques for managing speculation on amd processors. https://developer.amd.com/wp-content/resources/90343-B_SotwareTechniquesforManagingSpeculation_WP_7-18Update_FNL.pdf, 2018.

[6] Peter Aldous and Matthew Might. Static analysis of non-interference in expressive low-level languages. In *Static Analysis*, pages 1–17, 2015.

[7] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. Jasmin: High-assurance and high-speed cryptography. In *CCS '17*, pages 1807–1823, 2017.

[8] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 53–70, 2016.

[9] Musard Balliu, Mads Dam, and Roberto Guanciale. Inspectre: Breaking and fixing microarchitectural vulnerabilities by formal analysis. *CoRR*, abs/1911.00868, 2019.

[10] G. Barthe, B. Gregoire, and V. Laporte. Secure compilation of side-channel countermeasures: The case of cryptographic constant-time. In *CSF 2018*, pages 328–343, 2018.

[11] Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. Formal verification of a constant-time preserving c compiler. *Proc. ACM Program. Lang.*, 4(POPL), 2019.

[12] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. *Math. Struct. Comput. Sci.*, 21(6):1207–1252, 2011.

[13] William J. Bowman and Amal Ahmed. Noninterference for free. In *ICFP*. ACM, 2015.

[14] Matteo Busi, Job Noorman, Jo Van Bulck, Letterio Galletta, Pierpaolo Degano, Jan Tobias Mühlberg, and Frank Piessens. Provably secure isolation for interruptible enclaved execution on small microprocessors: Extended version. *CoRR*, abs/2001.10881, 2020.

[15] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *USENIX Security '19*, 2019.

[16] Chandler Carruth. Speculative load hardening. https://llvm.org/docs/SpeculativeLoadHardening.html, 2018.

[17] Sunjay Cauligi, Craig Disselkoen, Klaus v Gleissenthall, Deian Stefan, Tamara Rezk, and Gilles Barthe. Towards constant-time foundations for the new spectre era. *arXiv preprint arXiv:1910.01755*, 2019.

[18] Kevin Cheang, Cameron Rasmussen, Sanjit A. Seshia, and Pramod Subramanyan. A formal approach to secure speculation. In *CSF '19*, 2019.

[19] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. Stealing intel secrets from SGX enclaves via speculative execution. In *EuroS&P '19*, 2019.

[20] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.

[21] Dominique Devriese, Marco Patrignani, and Frank Piessens. Fully-abstract compilation by approximate back-translation. In *POPL '16*, 2016.

[22] Dominique Devriese and Frank Piessens. Noninterference through secure multi-execution. In *S&P 2010*, pages 109–124, 2010.

[23] Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. A type discipline for authorization policies. *ACM Trans. Program. Lang. Syst.*, 29(5), August 2007.

[24] Cedric Fournet, Nikhil Swamy, Juan Chen, Pierre-Evariste Dagand, Pierre-Yves Strub, and Benjamin Livshits. Fully abstract compilation to JavaScript. In *Proceedings POPL '13*, pages 371–384, 2013.

[25] Andrew D. Gordon and Alan Jeffrey. Authenticity by typing for security protocols. *J. Comput. Secur.*, 11(4):451–519, July 2003.

[26] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch side-channel attacks: Bypassing smap and kernel aslr. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16. ACM, 2016.

[27] Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. SPECTECTOR: principled detection of speculative information flows. In *S&P '20*. IEEE, 2020.

[28] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. Harware/software contracts for secure speculation. In *S&P '21*. IEEE, 2021.

[29] Norm Hardy. The confused deputy: (or why capabilities might have been invented). *SIGOPS Oper. Syst. Rev.*, 22(4):36–38, October 1988.

[30] Jann Horn. Google project zero - issue 1528: speculative execution, variant 4: speculative store bypass. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528.

[31] Intel. Intel Analysis of Speculative Execution Side Channels. https://software.intel.com/sites/default/files/managed/b9/f9/336983-Intel-Analysis-of-Speculative-Execution-Side-Channels-White-Paper.pdf, 2018.

[32] Intel. Retpoline: A branch target injection mitigation. https://software.intel.com/security-software-guidance/api-app/sites/default/files/Retpoline-A-Branch-Target-Injection-Mitigation.pdf, 2018.

[33] Intel. Using Intel Compilers to Mitigate Speculative Execution Side-Channel Issues. https://software.intel.com/en-us/articles/using-intel-compilers-to-mitigate-speculative-execution-side-channel-issues, 2018.

[34] Yannis Juglaret, Cătălin Hriţcu, Arthur Azevedo de Amorim, Boris Eng, and Benjamin C. Pierce. Beyond good and evil: Formalizing the security guarantees of compartmentalizing compilation. In *CSF '16*, pages 45–60, 2016.

[35] Vladimir Kiriansky and Carl Waldspurger. Speculative Buffer Overflows: Attacks and Defenses. *CoRR*, abs/1807.03757, 2018.

[36] Paul Kocher. Spectre mitigations in Microsoft's C/C++ compiler. https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html, 2018.

[37] 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. Spectre Attacks: Exploiting Speculative Execution. In *S&P '19*. IEEE, 2019.

[38] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *WOOT '18*, 2018.

[39] Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[40] Sergio Maffeis, Martín Abadi, Cédric Fournet, and Andrew D. Gordon. Code-carrying authorization. In *ESORICS 2008*, pages 563–579, 2008.

[41] Giorgi Maisuradze and Christian Rossow. Ret2Spec: Speculative Execution Using Return Stack Buffers. In *CCS '18*, 2018.

[42] Giorgi Maisuradze and Christian Rossow. Ret2Spec: Speculative Execution Using Return Stack Buffers. In *Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security*, CCS '18. ACM, 2018.

[43] Andrea Mambretti, Matthias Neugschwandtner, Alessandro Sorniotti, Engin Kirda, William Robertson, and Anil Kurmus. Let's Not Speculate: Discovering and Analyzing Speculative Execution Attacks. In IBM Technical Report RZ3933, 2018.

[44] Ross Mcilroy, Jaroslav Sevcik, Tobias Tebbi, Ben L. Titzer, and Toon Verwaest. Spectre is here to stay: An analysis of side-channels and speculative execution. *CoRR*, abs/1902.05178, 2019.

[45] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *International Conference on Information Security and Cryptology*, pages 156–168. Springer, 2005.

[46] Max S. New, William J. Bowman, and Amal Ahmed. Fully abstract compilation via universal embedding. In *ICFP '16*, pages 103–116, 2016.

[47] Oleksii Oleksenko, Bohdan Trach, Tobias Reiher, Mark Silberstein, and Christof Fetzer. You shall not bypass: Employing data dependencies to prevent bounds check bypass. *CoRR*, abs/1805.08506, 2018.

[48] Andrew Pardoe. Spectre mitigations in MSVC. https://blogs.msdn.microsoft.com/vcblog/2018/01/15/spectre-mitigations-in-msvc/, 2018.

[49] Marco Patrignani. Why should anyone use colours? or, syntax highlighting beyond code snippets. CoRR abs/2001.11334, 2020.

[50] Marco Patrignani, Pieter Agten, Raoul Strackx, Bart Jacobs, Dave Clarke, and Frank Piessens. Secure compilation to protected module architectures. *TOPLAS*, 2015.

[51] Marco Patrignani, Amal Ahmed, and Dave Clarke. Formal approaches to secure compilation a survey of fully abstract compilation and related work. *ACM Comput. Surv.*, 51(6):125:1–125:36, January 2019.

[52] Marco Patrignani and Deepak Garg. Robustly Safe Compilation. In *ESOP'19*, 2019.

[53] Filip Pizlo. What Spectre and Meltdown mean for WebKit. https://webkit.org/blog/8048/what-spectre-and-meltdown-mean-for-webkit/, 2018.

[54] V. Rajani, D. Garg, and T. Rezk. On access control, capabilities, their equivalence, and confused deputy attacks. In *CSF '16*, pages 150–163, 2016.

[55] Gabriel Scherer, Max New, Nick Rioux, and Amal Ahmed. Fabous interoperability for ml and a linear language. In *FOSSACS '18*, pages 146–162, 2018.

[56] D. Schoepe, M. Balliu, B. C. Pierce, and A. Sabelfeld. Explicit secrecy: A policy for taint tracking. In *Euro S&P '16*, pages 15–30, 2016.

[57] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. Netspectre: Read arbitrary memory over network. In *European Symposium on Research in Computer Security*, pages 279–299, 2019.

[58] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. Reasoning about a machine with local capabilities: Provably safe stack and return pointer management. *ACM Trans. Program. Lang. Syst.*, 42(1):5:1–5:53, 2020.

[59] Julian Stecklina and Thomas Prescher. LazyFP: Leaking FPU register state using microarchitectural side-channels. *CoRR*, abs/1806.07480, 2018.

[60] David Swasey, Deepak Garg, and Derek Dreyer. Robust and compositional verification of object capability patterns. In *OOPSLA '17*, 2017.

[61] Tachio Terauchi and Alex Aiken. Secure information

flow as a safety problem. In *SAS '05*, 2005.

[62] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. MeltdownPrime and SpectrePrime: Automatically-synthesized attacks exploiting invalidation-based coherence protocols. *CoRR*, abs/1802.03802, 2018.

[63] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue in-flight data load. In *Proceedings of the 40th IEEE Symposium on Security and Privacy*, S&P '19. IEEE, 2019.

[64] Marco Vassena, Klaus Gleissenthall, Rami Kici, Deian Stefan, and Ranjit Jhala. Automatically eliminating speculative leaks with BLADE. *CoRR*, to appear, 2020.

[65] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. oo7: Low-overhead defense against spectre attacks via binary analysis. *CoRR*, abs/1807.05843, 2018.

Appendix A
NISLH: a Secure Non-Interprocedural SLH

It is also possible to secure the variant of SLH that does not carry the predicate bit across procedures. We model NISLH as $[\![\cdot]\!]^{s}_{n}$ by having the predicate bit initialized at the beginning of each function to $\mathbf{1}$ (false) in a local variable $\mathbf{pr}$. As before, compiled code updates $\mathbf{pr}$ after every branching instruction. To ensure that $\mathbf{pr}$ correctly captures whether we are mis-speculating, we place an $\mathbf{lfence}$ as the first instruction of every compiled function.

$$\left[\!\!\left[\begin{array}{l}\mathsf{f(x)} \mapsto \mathsf{s;}\\ \quad \mathsf{return;}\end{array}\right]\!\!\right]^{s}_{n} = \mathbf{f(x)} \mapsto \left|\begin{array}{l}\mathbf{lfence;\ let\ pr{=}false\ in}\\ \ [\![\mathsf{s}]\!]^{s}_{n}\mathbf{;\ return;}\end{array}\right.$$

$$\left[\!\!\left[\begin{array}{l}\mathsf{ifz\ e}\\ \quad \mathsf{then\ s}\\ \quad \mathsf{else\ s'}\end{array}\right]\!\!\right]^{s}_{n} = \left|\begin{array}{l}\mathbf{let\ x_f{=}}[\![\mathsf{e}]\!]^{s}_{n}\ \mathbf{in}\\ \ \mathbf{ifz\ x_f\ then\ let\ pr{=}pr \vee \neg x_f\ in\ } [\![\mathsf{s}]\!]^{s}_{n}\\ \quad \mathbf{else\ let\ pr{=}pr \vee x_f\ in\ } [\![\mathsf{s'}]\!]^{s}_{n}\end{array}\right.$$

This compiler is also $RSSC^{-}$ for the same reason as before. Instead of having location $\mathbf{-1}$ that correctly tracks speculation, local variable $\mathbf{pr}$ does (masking is done as in $[\![\cdot]\!]^{s}$ before).

Theorem 9 (The NISLH compiler is $RSSC^{-}$). $\vdash [\![\cdot]\!]^{s}_{n} : RSSC^{-}$

In a similar way, one can construct a secure, non-interprocedura version of $[\![\cdot]\!]^{ss}$ that satisfies $RSSC^{+}$.

Appendix B
Failing *RSSC* Proofs

When a countermeasure is not $RSSC$ we can use the insights of its failed proof to understand whether it is also not $RSNIP$. In fact, while MSVC was already known to be insecure, this was not true for SLH. When we modelled vanilla SLH and started proving $RSSC^{+}$, the proof broke in the "gray area". While this does not directly mean that SLH is insecure, the way the proof broke provided insights on the speculative leaks not blocked by SLH. Concretely, we were not able to show that the property on speculating target states holds when speculating reductions are done and this led to the example Listing 4. We believe the insights of this proof technique can guide proofs of (in)security of other countermeasures too.

Appendix C
The Spectre v2 Case

This section describes how to apply our methodology to reason about countermeasures against the Spectre v2 attack. The Spectre v2 attack relies on speculation over the outcome of indirect jumps, rather than branch instructions. When an indirect jump is encountered, if the location where to jump is not present in the cache, heuristics are used in order to understand where to jump to. As for the speculation over branches, these heuristics can be wrong, and when this is detected, execution is rolled back. An attacker can therefore exploit this kind of speculative execution in order to make benign code execute malicious one. The main countermeasure against this kind of attack is the use of a *retpoline*, i.e., a *ret*urn-based tram*poline*. Intuitively, the retpoline replaces indirect jumps with a return to dead code, where the program will effectively sleep until the speculation window is over.

In order to prove security of the retpoline countermeasure, we therefore need the following:

- add indirect jumps to our languages and give them a regular semantics (Section C-A);
- give a speculative reduction to jump in $\mathbf{T}$ such that the location where to jump is nondeterministically chosen; this will be the start of speculation (Section C-B);
- change the call/return semantics in order to model retpolines, i.e., have the return address explicit (Section C-C).

With these changes, we can formalise a compiler that introduces the retpoline countermeasure (Section C-D) and reason about whether it is secure (Section C-E).

A. Indirect Jumps

The simplest way to add indirect jumps to our while languages is to treat function names f as natural numbers and add a statement $goto\ \ e$ that jumps to function f where $B \triangleright e\ \downarrow\ f$. Additionally, we need to add the way for a component to specify private functions, i.e., functions that are not callable from the attacker. This is still generic enough that one can model the assembly-level kind of attacks without having to add a pc to all instructions or labels to the language.

B. Speculative Execution of Jumps

To focus only on speculation over jumps, we would replace Rule E-$\mathbf{T}$-speculate-if (handling the speculation over branch instructions) with a rule that checks that the statement being executed is a $goto\ \ e$ where e evaluates to f. In that case, the right state (jumping to f) is pushed on the stack of states, but on top of that we push another state with a jump to function $f' \neq f$, for a non-deterministically chosen f' that is valid.

C. Explicit Call and Return Semantics

We need to add a return address, keep track of the return address in a stack of return addresses as well as a register where the return address can be read from. The reason is that the retpoline countermeasure relies on another kind of speculation, the one on return addresses. Normally, architectures push the return address on the stack and in a specific register `rsp`. When it is time to return, if the value on top of the stack differs from that on `rsp`, speculation starts, and a return to the top of the stack is made. When speculation ends, it is rolled back (as before, with the usual microarchitectural leaks) and a return to the value of `rsp` is done.

D. *The Retpoline Countermeasure*

The retpoline countermeasure $[\![\cdot]\!]^r$ is a homomorphic compiler with a single salient case: the compilation of goto e, where we encode the implementation of retpolines from Compiling a goto will not rely on target-level **goto**, since they would trigger the goto-speculation and result in vulnerable code. Instead, the compilation of goto will be turned into a call to an auxiliary function **aux**. Function **aux** will change the contents of register **rsp** to the function where the source goto wanted to jump. Then, function **aux** will contain code that sleeps. This way, when the compiled goto is executed, function **aux** is called and the address where to the goto should have jumped to to is pushed on the stack. This function speculatively returns to the code that sleeps and then, when speculation ends, execution resumes from the address popped from the stack (the target of the goto).

E. *Security of* $[\![\cdot]\!]^r$

We believe $[\![\cdot]\!]^r$ is $RSSC^{+}$ and we can argue that using the same proof technique described in Section V-D. As before, the key part of these proofs is reasoning when speculation happens, i.e., in the gray area of Figure 3. In the case of $[\![\cdot]\!]^r$, we see that the only code executed during speculation is sleeping code. Additionally, once the speculation window runs out, we need to prove that the state we end up in is the same as the source state that executed the goto. However, this last step only amounts to proving that the retpoline is correct, i.e., that it jumps where it is supposed to.