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. However, the security of these countermeasures has not been ascertained: while some of them are *believed* to be secure, others are *known* to be insecure and result in vulnerable programs.**

In this paper, we formally prove the security (or insecurity) of compiler-level countermeasures for Spectre. To prove security, we introduce a novel, general methodology built upon recent secure compilation theory. We use this theory to derive secure compilation criteria formalising when a compiler produces secure code against Spectre attacks. With these criteria, we formally prove that some countermeasures, such as speculative load hardening (SLH), are vulnerable to Spectre attacks and others (like speculation-barriers-insertion and variations of SLH) are secure.

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 [48]. 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 [34, 36, 37, 40, 56]. To mitigate these attacks, compilers deployed a number of compiler-level countermeasures. For instance, the insertion of `lfence` speculation barriers [30] and speculative load hardening [16] *can* be used to mitigate leaks introduced speculation over branch instructions (i.e., the Spectre v1 attack [36]).

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 [26, 35].

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 [26]) 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* (SNI) [26] considers all leaks resulting from speculatively-executed instructions as harmful. (2) *Weak speculative non-interference* [27] 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*. Roughly speaking such compilers produce code that is safe 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, therefore, 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, 51] (here: absence of speculative leaks).

► Using our framework, we perform a comprehensive security analysis of compiler-level countermeasures against

[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.

Spectre v1 implemented in major compilers (Section V). Specifically, we focus on (1) automated insertion of `lfences` (implemented in the Microsoft Visual C++ and the Intel ICC compilers [32, 47]), 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.
- We propose a variant of SLH, called strong SLH, that provides strong *RSNIP* and blocks all speculative leaks.
- The non-interprocedural variant of SLH violates weak *RSNIP* and is thus insecure.

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 *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-B). 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 the classical Spectre v1 snippet (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 its trace model (Section II-D). This is followed by the operational semantics and the taint-tracking mechanism of our languages (without speculative execution, 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 is the one required to express the strong variants of SS and SNI, so we conclude by defining the changes necessary to express the weak variants too (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 [36] 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 context, an attacker is an arbitrary program at target level that is linked against a (compiled) partial program of interest in the source language. 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 [28, 53] 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, 44] without requiring microarchitectural models. Following [26], 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.

For modeling speculative execution's effects, our target language mispredicts the outcome of all component's branch instructions. This is the worst-case scenario in terms of leakage regardless of how attackers poison the branch predictor [26].

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, 26], 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 $\overline{e}$.

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

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 (i.e., 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, 49, 51]), or a *μarch. action* δ capturing what a microarchitectural attacker can observe.

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

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

The $\texttt{read}(n)$ and $\texttt{write}(n)$ actions denote respectively read and write accesses to the heap location n, and they model leaks through the data cache. In contrast, the $\texttt{if}(v)$ action denotes the outcome of branch instructions and the $\texttt{rlb}$ 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, 39, 59], 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 describes 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 A and P's functions and heaps. 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 of P's interfaces.

First we introduce program states $C, H, \overline{B} \triangleright (s)_{\overline{f}}$ which consist of a component 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. The function names $\overline{f}$, which we often omit for simplicity, are used to infer if we are executing code from the attacker or component, which determines the produced labels.

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

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 (lub, $\sqcup$) and greatest-lower-bound (glb, $\sqcap$) 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:

$$\frac{B \triangleright e \downarrow n : \sigma \quad B \triangleright e' \downarrow \_ : \sigma'' \quad H'_t = H_t \cup -|n| \mapsto \sigma''}{\sigma_{pc}; C, H_t, \overline{B} \cdot B \triangleright e :=_p e' \xrightarrow{\sigma \sqcap \sigma_{pc}} C, H'_t, \overline{B} \cdot B \triangleright \mathit{skip}} \text{(T-write-prv)}$$

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

Writing to the private heap (Rule T-write-prv) taints the location ($-|n|$) with the written expression's taint (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* since this information is not used by the taint-tracking (as explained in 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.

$$\frac{B_v + B_t \equiv B \quad B_v \triangleright e \downarrow v \quad B_t \triangleright e \downarrow \sigma}{B \triangleright e \downarrow v : \sigma} \text{(Combine-B)}$$

Here ends the part of the semantics that is common to both L and 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.

$$\frac{\Omega_v + \Omega_t = \Omega \quad \Omega'_v + \Omega'_t = \Omega' \quad \Omega_v \xrightarrow{\lambda} \Omega'_v \quad \mathsf{S}; \Omega_t \xrightarrow{\sigma} \Omega'_t}{\Omega \xrightarrow{\lambda^\sigma} \Omega'} \text{(Combine-s-L)}$$

$$\frac{H_v + H_t \equiv H \quad \overline{B'_v} + \overline{B_t} \equiv \overline{B} \quad \overline{B_v} + \overline{B_t} \equiv \overline{B'}}{C; H_v; \overline{B_v} \triangleright s + C; H_t; \overline{B} \triangleright s' \equiv C; H; \overline{B'} \triangleright s} \text{(Merge-}\Omega\text{)}$$

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 ($\mathsf{S}; \Omega_t \xrightarrow{\sigma} \Omega'_t$). Note that the pc taint is always safe since there is no speculation in L. Observe also that merging states $\Omega_v + \Omega_t$ results in ignoring the value information accumulated in Ω_t since we want to rely on the computation performed by the operational semantics (Rule Merge-Ω).

Next, we can define L's big-step semantics $\Rightarrow$ that 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:

$$\frac{\Omega \equiv \overline{\mathsf{F}}, \bar{\mathsf{I}}, \mathsf{H}, \mathsf{B} \triangleright (\mathsf{s})_{\bar{\mathsf{f}}.\mathsf{f}} \qquad \Omega' \equiv \overline{\mathsf{F}}, \bar{\mathsf{I}}, \mathsf{H}', \mathsf{B}' \triangleright (\mathsf{s}')_{\overline{\mathsf{f}'}.\mathsf{f}'} \quad \Omega \xrightarrow{\alpha^\sigma} \Omega' \quad \text{if } \mathsf{f} == \mathsf{f}' \text{ and } \mathsf{f} \in \mathsf{I} \text{ then } \overline{\lambda^\sigma} = \epsilon \text{ else } \overline{\lambda^\sigma} = \alpha^\sigma}{\Omega \xRightarrow{\overline{\lambda^\sigma}} \Omega'} \text{(E-L-single)}$$

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 $\bar{\mathsf{I}}$ are defined by the attacker).

Finally, the behaviour 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 (indicated as Ω_0(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 T

Our semantics for T is inspired by the "always mispredict" semantics of Guarnieri *et al.* [26], which captures the worst-case scenario (from an information theoretic perspective) independently on the branch prediction's outcomes. Whenever the semantics executes a branch instruction, it first mis-speculates by executing the wrong branch for a fixed number w of steps (called *speculation window*). After speculating for 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 S and it is raised to U when speculation happens.

As for the non-speculative semantics, we decouple the operational aspects from the taint-tracking ones. Speculative program states (Σ) are defined as stacks of speculation instances

[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 [38].

(Φ), which in turn are split in their operational (Φ_v) and taint (Φ_t) sub-parts. A speculation instance (Ω, w, σ) records the 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 $\bot$ when no speculation is happening; its maximum length is a global constant ω that depends on physical characteristics of the CPU like the reorder buffer's size.

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

The execution of program W starts in state $(\Omega_0(W), \bot, S)$, 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)
$$\frac{\Omega_v \xrightarrow{\epsilon} \Omega'_v \quad \Omega_v \equiv C, H_v, \overline{B_v} \triangleright s; s' \quad s \equiv \texttt{lfence}}{\overline{\Phi_v} \cdot (\Omega_v, n+1) \overset{\epsilon}{\rightsquigarrow} \overline{\Phi_v} \cdot (\Omega'_v, 0)}$$

(E-T-speculate-action)
$$\frac{\Omega_v \xrightarrow{\lambda} \Omega'_v \quad \Omega_v \equiv C, H_v, \overline{B_v} \triangleright s; s' \quad s \not\equiv \texttt{ifz \_ then \_ else \_} \text{ and } s \not\equiv \texttt{lfence}}{\overline{\Phi_v} \cdot (\Omega_v, n+1) \overset{\lambda}{\rightsquigarrow} \overline{\Phi_v} \cdot (\Omega'_v, n)}$$

(E-T-speculate-if)
$$\frac{\begin{array}{c}\Omega_v \equiv C, H_v, \overline{B_v} \cdot B_v \triangleright (s; s')_{\overline{f} \cdot f} \quad s \equiv \texttt{if e then } s'' \texttt{ else } s''' \\ \Omega_v \xrightarrow{\alpha} \Omega'_v \quad C \equiv \overline{F}; \overline{I} \quad f \notin \overline{I} \quad j = \texttt{min}(\omega, n) \\ \text{if } B_v \triangleright e \downarrow 0 \text{ then } \Omega''_v \equiv C, H_v, \overline{B_v} \cdot B_v \triangleright s'''; s' \\ \text{if } B_v \triangleright e \downarrow n \text{ and } n > 0 \text{ then } \Omega''_v \equiv C, H_v, \overline{B_v} \cdot B_v \triangleright s''; s'\end{array}}{\overline{\Phi_v} \cdot (\Omega_v, n+1) \overset{\alpha}{\rightsquigarrow} \overline{\Phi_v} \cdot (\Omega'_v, n) \cdot (\Omega''_v, j)}$$

(E-T-speculate-rollback)
$$\frac{n = 0 \text{ or } \Omega_v \text{ is stuck}}{\overline{\Phi_v} \cdot (\Omega_v, n) \overset{\texttt{rlb}}{\rightsquigarrow} \overline{\Phi_v}}$$

TODO: do we ever say the attacker cannot speculate? We can make it speculate and just change the taint tracking: only raise the taint to U when the if is done by the code, set to S when the if is done by attacker This is ok because whatever execution the attacker does through speculation, another attacker did without speculation ($\forall A$) so there is no need to consider this execution unsafe

Mis-speculation pushes the mis-speculating state on top of the stack (Rule E-T-speculate-if). 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 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)
$$\frac{\sigma; \Omega_t \xrightarrow{\sigma'} \Omega'_t \quad \Omega_t \equiv C, H_t, \overline{B} \triangleright s; s' \quad s \not\equiv \texttt{ifz \_ then \_ else \_} \text{ and } s \not\equiv \texttt{lfence}}{\overline{\Phi_t} \cdot (\Omega_t, \sigma) \overset{\sigma' \sqcup \sigma}{\rightsquigarrow} \overline{\Phi_t} \cdot (\Omega'_t, \sigma)}$$

(T-T-speculate-if)
$$\frac{\begin{array}{c}\Omega_t \equiv C, H_t, \overline{B} \cdot B \triangleright (s; s')_{\overline{f} \cdot f} \quad s \equiv \texttt{if e then } s'' \texttt{ else } s''' \\ \sigma'; \Omega_t \xrightarrow{\sigma} \Omega'_t \quad C \equiv \overline{F}; \overline{I} \quad f \notin \overline{I} \\ \text{if } B \triangleright e \downarrow 0 : \sigma \text{ then } \Omega''_t \equiv C, H_t, \overline{B} \cdot B \triangleright s'''; s' \\ \text{if } B \triangleright e \downarrow n : \sigma \text{ and } n > 0 \text{ then } \Omega''_t \equiv C, H_t, \overline{B} \cdot B \triangleright s''; s'\end{array}}{\overline{\Phi_t} \cdot (\Omega_t, \sigma') \overset{\sigma \sqcup \sigma'}{\rightsquigarrow} \overline{\Phi_t} \cdot (\Omega'_t, \sigma') \cdot (\Omega''_t, U)}$$

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

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 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 ($\overset{\cdot}{\rightsquigarrow}$) instead of the non-speculative one ($\xrightarrow{\cdot}$).

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 W's initial state 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^i_A.

$$t_{ns} = \texttt{call get } 0?^S \cdot \texttt{if}(0)^S \cdot \texttt{read}(n_A)^S \cdot \texttt{read}(n_B + v^0_A)^S \cdot \texttt{ret!}^S$$
$$t_{sp} = \texttt{call get } 8?^S \cdot \texttt{if}(1)^S \cdot \texttt{read}(n_A + 8)^S \cdot \texttt{read}(n_B + v^8_A)^U \cdot \texttt{rlb}^S \cdot \texttt{ret!}^S$$

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). ⊡

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 [27], non-speculatively reading from the private heap produces an action $\texttt{read}(n \mapsto v)$ that contains the read value v as well as the accessed memory address n. Speculative reads, instead, produce actions $\texttt{read}(n)$ 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 pc's and the read value's taints ($\sigma' \sqcap \sigma_{pc}$) 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$^-$. We remark that these languages have the same syntax and different semantics. This allows us to study the relationships between RSNI and RSS for weak and strong languages (Section III-C). In the following, we write RSNI(L) and RSS(L) to indicate which language L the definitions are referring to.

A. Robust Speculative Non-Interference

Speculative non-interference is a class of security properties [26, 27] 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* $t\restriction_{nse}$ [26] of a trace t extracts the observations associated only with non-speculatively-executed instructions. We obtain $t\restriction_{nse}$ by removing from t all sub-strings enclosed between $\texttt{if}(v)$ and $\texttt{rlb}$ observations. We illustrate this using an example: below is $\cdot\restriction_{nse}$ applied to $\mathbf{t_{sp}}$ from Example 1.

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

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))**.**

$$\vdash W : \text{SNI} \stackrel{\text{def}}{=} \forall W'.\ \text{if}\ W' =_{\mathrm{L}} W \ \text{and}\ Beh(\Omega_0\,(W))\restriction_{nse} = Beh(\Omega_0\,(W'))\restriction_{nse} \ \text{then}\ Beh(\Omega_0\,(W)) = Beh(\Omega_0\,(W'))$$

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))**.**

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

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 $\mathbf{8}$. 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 ($\mathbf{n_B} + \mathbf{v^8_A} \neq \mathbf{n_B} + \mathbf{v^8_N}$).

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

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* ($\texttt{read}(\mathbf{n_B} + \mathbf{v^8_A})^{\mathbf{U}} \neq \texttt{read}(\mathbf{n_B} + \mathbf{v^8_N})^{\mathbf{U}}$). ⊡

B. Robust Speculative Safety

Robust speculative safety is a security property that depends on the taints occurring in traces and its security guarantees depend on the underlying language (as we discuss in Section III-C). Concretely, speculative safety ensures that *whole* program W generates only safe (S) actions in their traces.

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

$$\vdash W : \text{SS} \stackrel{\text{def}}{=} \forall \overline{\lambda^\sigma} \in Beh(W).\, \forall \alpha^\sigma \in \overline{\lambda^\sigma}.\, \sigma \equiv S$$

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

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

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

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$^-$.

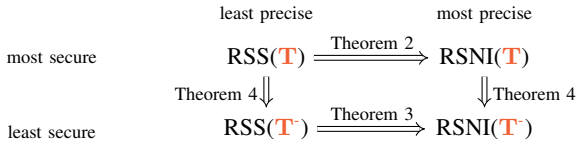
1) Relationships for L and L$^-$: All programs in L and L$^-$ trivially enjoy both speculative non-interference and speculative safety, because L and L$^-$ do not speculatively execute instructions and produce traces with only S actions.

[4] We derive our formalisation from the trace-based characterization in [26, Proposition 1].

Theorem 1 (All $\mathtt{L}$ and $\mathtt{L}^{\text{-}}$ programs are secure)**.**

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

2) Relationships for $\mathtt{T}$ and $\mathtt{T}^{\text{-}}$: The relationships are summarized below in terms of security guarantees and precision.



Characterization of speculative non-interference: Instantiating RSNI with languages $\mathtt{T}$ and $\mathtt{T}^{\text{-}}$ result in different security guarantees. Specifically, RSNI($\mathtt{T}$) corresponds to speculative non-interference [26, 27], which ensures the absence of *all* speculative leaks. In contrast, RSNI($\mathtt{T}^{\text{-}}$) corresponds to weak speculative non-interference [27], which allows speculative leaks of information that has been retrieved non-speculatively. That is, RSNI($\mathtt{T}^{\text{-}}$) ensures the absence only of speculative leaks of speculatively-accessed data.

As shown in [27], strong and weak speculative non-interference (that is, RSNI($\mathtt{T}$) and RSNI($\mathtt{T}^{\text{-}}$)) 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($\mathtt{T}$) semantically characterize the absence of speculative leaks. In contrast, RSS($\mathtt{T}$) is an over-approximation of RSNI($\mathtt{T}$) whose preservation through compilation is easier to prove than RSNI($\mathtt{T}$)-preservation.

Theorem 2 (RSS($\mathtt{T}$) over-approximates RSNI($\mathtt{T}$))**.**

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

To understand point 1, observe that RSS($\mathtt{T}$) ensures that only safe observations are produced by a program $\mathtt{P}$. This, in turn, ensures that no information originating from the private heap is leaked through speculatively-executed instructions in $\mathtt{P}$. Therefore, $\mathtt{P}$ satisfies RSNI($\mathtt{T}$) 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($\mathtt{T}$) because any two configurations 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($\mathtt{T}$) because speculative memory accesses will produce $\mathtt{U}$ actions.

```
1 void get_nc (int y)
2 TODO: < size) then B[A[y] *512] else B[A[y] *512]
```

Listing 2. Code that is RSNI but not RSS.

RSNI($\mathtt{T}^{\text{-}}$) and RSS($\mathtt{T}^{\text{-}}$) enjoy a relationship similar to their RSNI($\mathtt{T}$) and RSS($\mathtt{T}$).

Theorem 3 (RSS($\mathtt{T}^{\text{-}}$) over-approximates RSNI($\mathtt{T}^{\text{-}}$))**.**

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

Strong Variants Imply the Weak Ones: Since RSNI($\mathtt{T}$) ensures the absence of *all* speculative leaks while RSNI($\mathtt{T}^{\text{-}}$) only ensures the absence of *some* of them, any RSNI($\mathtt{T}$) program is also RSNI($\mathtt{T}^{\text{-}}$). Similarly, any RSS($\mathtt{T}$) program is also RSS($\mathtt{T}^{\text{-}}$) since all actions tainted $\mathtt{S}$ by Rule T-read-prv are also tainted $\mathtt{S}$ also by Rule T-read-prv-weak.

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

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

IV. Compiler Criteria for Spectre Security

In this section, we introduce our secure compilation criteria. We start with a criterion that preserves robust speculative safety, which we dub *RSSP*, for *robust speculative safety preservation* (Section IV-A). Next, we introduce a criterion that preserves RSNI, which we dub *RSNIP* for *robust speculative non-interference preservation* (Section IV-B). 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 $\mathtt{L}$-$\mathtt{T}$ or $\mathtt{L}^{\text{-}}$-$\mathtt{T}^{\text{-}}$. Criteria instantiated with the strong languages (e.g., *RSSP*($\mathtt{L}$,$\mathtt{T}$)) are indicated with a + (i.e., *RSSP*$^{+}$) while those instantiated with weak languages (e.g., *RSNIP*($\mathtt{L}^{\text{-}}$,$\mathtt{T}^{\text{-}}$)) are indicated with a - (i.e., *RSNIP*$^{\text{-}}$). When we omit the 'sign', we refer to both criteria For simplicity, we only present the strong criteria, weak ones are defined in exactly the same way (but for different languages).

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*$^{+}$)**.**

$$\vdash [\![\cdot]\!] : \mathit{RSSP}^{+} \stackrel{\text{def}}{=} \forall \mathtt{P} \in \mathtt{L}.\ \text{ if } \vdash \mathtt{P} : \mathrm{RSS}(\mathtt{L})$$
$$\text{then } \vdash [\![\mathtt{P}]\!] : \mathrm{RSS}(\mathtt{T})$$

Definition 5 is a "property-ful" criterion since it explicitly refers to the property that the compiler preserves [3, 4]. 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, 51]. This is often beneficial because "property-free" criteria come in so-called *backtranslation* form, which have established proof techniques [2, 4, 13, 45, 49, 51].

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 $\overline{\lambda^\sigma}$, we can build a source-level attacker A that generates a trace $\overline{\lambda^\sigma}$ that is related to $\overline{\lambda^\sigma}$. A source trace $\overline{\lambda^\sigma}$ and a target trace $\overline{\lambda^\sigma}$ are related (denoted with $\overline{\lambda^\sigma} \approx \overline{\lambda^\sigma}$) 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).

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

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

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 ($RSSC^+$)**.**

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

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

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

TODO: add to TR

Definition 6 requires providing an existentially-quantified source attacker A. The general proof technique for these criteria is called *backtranslation* [4, 50], and it can either be attacker- [13, 21, 45] or trace-based [2, 49, 51]. The distinction tells us what quantified element we can use use to build the source attacker A, either the target attacker $\mathbf{A}$ or the finite trace $\overline{\lambda}$ respectively. Fortunately, our setup is powerful enough that both techniques apply, and, for simplicity, we will adopt the 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-less criterion.

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

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

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

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

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.

$$\text{t.s.} : \exists \mathbf{A}.\ \vdash \mathbf{A} : atk \text{ and given } \mathbf{A}'[\![\mathsf{P}']\!] =_{\mathsf{L}} \mathbf{A}[\![\mathsf{P}]\!]$$
$$\text{we have } \mathbf{Beh}(\boldsymbol{\Omega_0}(\mathbf{A}[\![\mathsf{P}]\!]))\restriction_{nse} = \mathbf{Beh}(\boldsymbol{\Omega_0}(\mathbf{A}'[\![\mathsf{P}']\!]))\restriction_{nse}$$
$$\text{and } \mathbf{Beh}(\boldsymbol{\Omega_0}(\mathbf{A}[\![\mathsf{P}]\!])) \neq \mathbf{Beh}(\boldsymbol{\Omega_0}(\mathbf{A}'[\![\mathsf{P}']\!]))$$

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 Section G-A. □

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 the pair of strong languages L and $\mathbf{T}$. There are similar chains of implications instantiated with the weak languages (L^- and $\mathbf{T}^-$) that use Theorem 3 instead of Theorem 2.

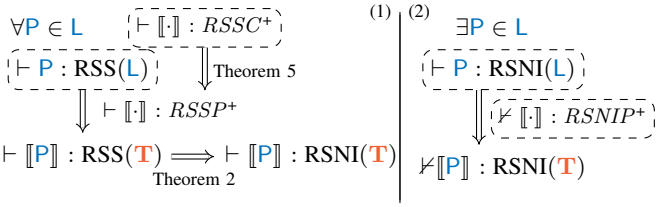


Figure 1. Our methodology to prove security (1) and insecurity (2) for compiler countermeasures against speculative leaks

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 an $RSSP$ compiler (i) so long as any P is RSS in the source (ii). By Theorem 5, for point (i) it is sufficient to show that the compiler is $RSSC$. Point (ii) holds for any P since they cannot speculate (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 need to show that the compiler is not $RSNIP$ (A) given that the source component P was RSNI in the source (B). Point (A) follows from showing what mentioned in Corollary 1. As before, point (B) holds for any source component P since they cannot speculate (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, 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 is vulnerable to any speculative leak.

V. Countermeasures Security and Insecurity

In this section, we precisely characterise the (in)security 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++ [47] (MSVC) compiler implements the first countermeasure in an insecure way, i.e., MSVC violates $RSNIP^-$ and produces programs that are insecure (Section V-A). We also prove that the Intel C++ compiler [32] (ICC) implements the same countermeasure securely, that is, ICC is $RSSP^+$; thereby preventing 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 speculative leak data retrieved non-speculatively, which might result in breaking properties like constant-time (see [27]). 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 $RSNIP^-$ (though for space constraints we report this only in the appendix). For space constraints, proofs are omitted though we provide a high-level proof overview (Section V-D).

A. MSVC is insecure

Inserting speculation barriers—the **lfence** x86 instruction—after branch instructions is a simple countermeasure against Spectre v1 [30, 32, 47]. 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. In our framework, this means that MSVC produce 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 with MSVC is not RSNI($\mathbf{T}^-$). The program we consider, which trivially satisfies RSNI(L^-) (Theorem 1) is given in Listing 3, and it speculatively leaks whether `A[y]` is `0` through the branch statement in line 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 [35]).

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

Its compiled counterpart, however, does not contain **lfence**s. Thus the compiled program still contains the speculative leak, and it violates RSNI($\mathbf{T}^-$). We refer to [26, 35] for additional examples illustrating MSVC's insecurity.

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 $\mathbf{T}$. While most of the compiler is straightforward (and thus omitted), its key feature is inserting an **lfence** at the beginning of every **then** and **else** branch of compiled code.

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

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)**.** $\vdash [\![\cdot]\!]^f : RSSC^+$

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 **1**) or not (predicate bit set to **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 **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 **0xF..F** if predicate bit is **1** or to **0x0..0** otherwise. During the computation, SLH-compiled code uses **cmov** instructions to conditionally update the mask and preserve the invariant that $mask = \mathbf{0xF..F}$ if code is mis-speculating and $mask = \mathbf{0x0..0}$ 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.

[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.

- All inputs to control-flow and store instructions are hardened by masking their values with $mask$ (i.e., by $\mathbf{or}$-ing their value with $mask$). That is, whenever code is mis-speculating (i.e., $mask = \mathbf{0xF..F}$) the inputs to control-flow and store statements are effectively "F-ed" to $\mathbf{0xF..F}$, otherwise they 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 analyze SLH's security guarantees.

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(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.

When this code is compiled with SLH, the value of A[y] is only hardened using the mask retrieved from the stack pointer. As a result, 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 still leaks the value of A[y], which will be different in traces generated from different, low-equivalent states. Hence, the compiled code violates RSNI(T).

2) SLH is RSNIP⁻: We now show that SLH is $RSNIP^-$, that is, it enforces weak speculative non-interference and 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 -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 $-1 :=_{\mathbf{p}} \cdots$). Since location -1 is reserved for the predicate bit, all memory accesses as well as the global 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 $\mathbf{let\ pr} = \mathbf{rd_p}\ -1\ \mathbf{in}\ \cdots$) 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 these statement's output. This is done by retrieving the predicate bit, storing it in variable $\mathbf{pr}$, performing the memory read, and conditionally masking its result based on $\mathbf{pr}$.

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

$[\![\cdot]\!]^s$ is $RSSC^-$ (Theorem 7) for two reasons. First, location -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 -1 and $\mathbf{pr}$ cannot be tampered by the attacker, the compiler initializes -1 correctly, and the assignments right after the branches correctly update location -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 $\mathbf{0}$, and constants are $\mathbf{S}$ since they leak no information. So, computations happening during speculation either depend on data loaded non-speculatively, which are tainted as $\mathbf{S}$ thanks to $\mathbf{T}^-$'s taint-tracking, 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 obtain stronger security guarantees, that is, the absence of *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 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 $\mathbf{0}$ which are tainted as $\mathbf{S}$. That is, labels during speculation are tainted as $\mathbf{S}$, and RSS(T) holds.

D. How to Prove RSSC

We now illustrate the proof technique used to prove SLH-related countermeasures secure. Recall from Section IV that 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 ($\langle\!\langle \cdot \rangle\!\rangle$) homomorphically translates target heaps, functions, statements etc. into source ones. A benefit of our setup is that we use the

$$[\![\overline{\mathsf{H}};\overline{\mathsf{F}};\overline{\mathsf{I}}]\!]^s = [\![\mathsf{H}]\!]^s \cup (-1 \mapsto 1 : \mathbf{S});\ [\![\overline{\mathsf{F}}]\!]^s;\ [\![\overline{\mathsf{I}}]\!]^s \qquad [\![\mathsf{e} :=_{\mathsf{p}} \mathsf{e}']\!]^s = \left|\begin{array}{l}\mathbf{let}\ \mathbf{x_f}{=}[\![\mathsf{e}]\!]^s{+}\mathbf{1}\ \mathbf{in}\ \mathbf{let}\ \mathbf{x'_f}{=}[\![\mathsf{e}']\!]^s\ \mathbf{in}\ \mathbf{let}\ \mathbf{pr} = \mathbf{rd_p}\ {-}\mathbf{1}\ \mathbf{in}\\ \quad \mathbf{let}\ \mathbf{x_f} = \mathbf{0}\ (\mathbf{if}\ \mathbf{pr})\ \mathbf{in}\ \mathbf{let}\ \mathbf{x'_f} = \mathbf{0}\ (\mathbf{if}\ \mathbf{pr})\ \mathbf{in}\ \mathbf{x_f} :=_{\mathbf{p}} \mathbf{x'_f}\end{array}\right.$$

$$[\![\mathsf{H}, -\mathsf{n} \mapsto \mathsf{v} : \mathsf{U}]\!]^s = [\![\mathsf{H}]\!]^s, -[\![\mathsf{n}]\!]^s - 1 \mapsto [\![\mathsf{v}]\!]^s : \mathbf{U}$$

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

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

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

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

same backtranslation for all proofs since the backtranslation operates on attacker code where no countermeasure is applied.

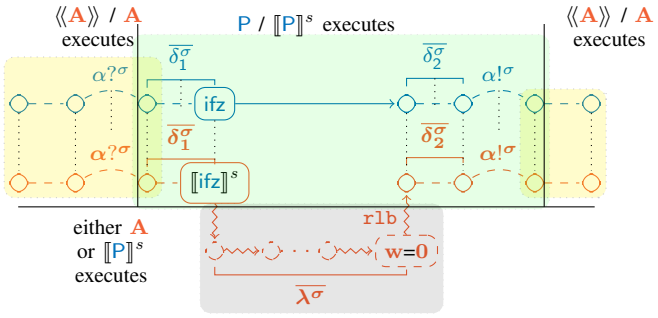


Figure 3. A diagram depicting the proof that countermeasure $[\![\cdot]\!]^s$ is *RSSC*.

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 A does a step, then ⟨⟨A⟩⟩ 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, 38]. That is, if s steps and emits a trace, then $[\![\mathsf{s}]\!]^s$ 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 T (grey area). After $[\![\mathsf{ifz}]\!]^s$ 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 w steps in both attacker and compiled code and generating a trace $\overline{\lambda^\sigma}$. We then prove that $\overline{\lambda^\sigma}$ is related to the empty source trace because all actions in $\overline{\lambda^\sigma}$ are tainted S, and so they do not leak. This fact follows from proving that while speculating, bindings always contain S values and therefore any generated action is S. In turn, this follows from proving that pr correctly captures if speculation is ongoing or not and that the mask is S. As mentioned, both of these hold for $[\![\cdot]\!]^s$ and $[\![\cdot]\!]^{ss}$ so they are secure.

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

VI. Discussion

A. Scope of the model

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

Our attacker can observe location of memory accesses and the outcome of control-flow statements during the execution. This attacker model offers a good tradeoff between precision and simplicity [8, 44], and it has proven to capture interesting microarchitectural leaks, such as those resulting from data and instruction caches or port contention.

Our target languages have a speculative semantics which is adequate to reason about security against Spectre v1-style attacks. The semantics, however, ignore other microarchitectural effects like out-of-order execution as well asother sources of speculative execution, like speculation over indirect jumps, that are exploited by other Spectre variants, as we discuss next.

TODO: (1) discuss private heap, (2) discuss what kind of leaks we capture/we do not capture, and (3) discuss attacker that is active on code but passive on observations

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 [36] exploits speculation over indirect jump instructions. The *retpoline* compiler-level countermeasure [31] 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 [41], in contrast, exploits speculation over return addresses (through `ret` instructions). To prevent it, Intel deployed a microcode update [31] that renders *retpoline* a valid countermeasure also against Spectre-RSB [15].

• Spectre-STL [29] exploits speculation over data dependencies between in-flight store and load operations (used for memory disambiguation). To mitigate it, ARM introduced a dedicated `SSBB` speculation barrier to prevent store bypasses that could potentially be injected by compilers.

To reason about these Spectre variants and their countermeasures, we first have to extend **T**'s speculative semantics. This can be done similarly to other semantics [9, 17, 43, 62].

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 H). 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 [34, 36] have been discovered; they differ in the exploited speculation sources [29, 37, 40], the covert channels [56, 58, 61], 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, 27, 43, 62] explicitly model microarchitectural details like, for instance, multiple pipeline stages, reorder buffers, caches, and branch predictors. These semantics are significantly more complex (and brittle) than ours (which is inspired by [26]), and they would lead to much harder proofs.

Security definition against Spectre attacks: SNI [26] has been used as security definition against speculative leaks also by [9, 27, 62]. 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.* [27] 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, 30] and SLH [16, 46], few countermeasures for Spectre v1 exist. Replacing branch instructions with branchless computations (using `cmov` and bit masking) is effective [52] but not generally applicable. oo7 [63] is a tool that automatically patches speculative leaks by injective speculation barriers. Similarly to the Microsoft MSVC compiler, however, oo7 misses some speculative leaks [26], fails in injecting all necessary speculation barriers, and ultimately violates $RSNIP^-$.

Blade [62] 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, 51]. Like [3, 51], we related 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, 33, 49, 50, 54, 57]. FAC compilers must preserve (and reflect) observational equivalence of source programs in their compiled counterparts [1, 50]. 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. If all execution is eventually rolled back, two programs that would be inequivalent due to speculation become equivalent simply because their speculative execution cannot affect their final result.

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 whether a program satisfies a 2-hypersafety property [20] (like RSNI) is notoriously challenging. Existing approaches for this include taint-tracking [6, 55] (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, 60] (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

This paper presented the first formal proofs of security and insecurity for existing compiler countermeasures against Spectre v1. For this, it took strong [26] and weak [27] SNI, two precise hyperproperties that tell when code is vulnerable to all or some speculation attacks (or not). Then it defined strong and weak SS, over-approximations of strong and weak SNI that only tell when code is secure against those attacks. Then, it formalised secure compilation criteria that state how to preserve those properties robustly through compilation and it defined a general methodology for proving security or insecurity of countermeasures by using those criteria.

APPENDIX A
MSVC DETAILS

Unlike ICC, MSVC tries to reduce the number of lfences by selectively determining which branches to patch. Example 3 below illustrates how MSVC works on the standard Spectre v1 snippet while Example 4 (and as pointed out in [26, 35]) illustrates that MSVC sometimes omits necessary lfences.

Example 3 (MSVC in action)**.** Listing 5 presents the (simplified) assembly produced by MSVC on the code of Listing 1.

```
1   mov rax, size // load size
2   cmp rcx, rax // compare y (in rcx) and size
3   jae END // jump if y is out−of−bound
4   lfence // halt speculative execution
5   movzx eax, A[rcx] // load A[y]
6   movzx eax, B[rax] // load B[A[y]]
7   mov temp, al // assignment to temp
8 END:
9   ret 0
```

Listing 5. Listing 1 compiled with MSVC with `/Qspectre` flag enabled.

In this case, MSVC correctly inserts the lfence (line 4) just after the branch instruction that checks whether x (stored in register rcx) is in-bound. The lfence stops the mis-speculated execution of the two memory accesses and it effectively prevents speculative leaks. ⊡

Example 4 (MSVC is not *RSNIP*)**.** Consider now the code in Listing 3, which is adapted from from [35, Example 10]. In contrast to Listing 1, this example speculatively leaks whether `A[y]` is `0` through the branch statement in line 3. When compiling this code with the lfence-countermeasure enabled, MSVC produces the snippet shown in Listing 6.

```
1     mov rax, size // load size
2     cmp rcx, rax // compare y (in rcx) and size
3     jae END // jump if out−of−bound
4     cmp [A+ rcx], 0 // compare A[y] and 0
5     jne END // jump if A[y] is not 0
6     movzx eax, B // load B[0]
7     mov temp, al // assignment to temp
8   END:
9     ret 0
```

Listing 6. Listing 3 compiled with MVCC with `/Qspectre` flag enabled.

In this case, the compiler does not insert an lfence after the first branch instruction on line 3. Therefore, the compiled program still contains a speculative leak.

In our framework, the source program from Listing 3 trivially satisfies RSNI, because the source language L does not allow speculative execution. Its compilation in Listing 6, however, violates RSNI. To show this, consider two low-equivalent initial states Ω^0, Ω^1 where y is out-of-bound, A[y] is 0 in Ω^0 and 1 in Ω^1, and the value of y is 42 in both. The corresponding traces are:

$$\mathtt{t}_{\Omega^0} = \mathtt{call}\ \mathtt{get}\ 42?^{\mathtt{S}} \cdot \mathtt{if}(\mathbf{0})^{\mathtt{S}} \cdot \mathtt{read}(\mathbf{n_A}+42)^{\mathtt{S}} \cdot \mathtt{if}(\mathbf{0})^{\mathtt{U}} \cdot \mathtt{rlb}^{\mathtt{S}} \cdot \mathtt{read}(\mathbf{n_B}+0)^{\mathtt{S}} \cdot \mathtt{rlb}^{\mathtt{S}}$$

$$\mathtt{t}_{\Omega^1} = \mathtt{call}\ \mathtt{get}\ 42?^{\mathtt{S}} \cdot \mathtt{if}(\mathbf{0})^{\mathtt{S}} \cdot \mathtt{read}(\mathbf{n_A}+42)^{\mathtt{S}} \cdot \mathtt{if}(\mathbf{1})^{\mathtt{U}} \cdot \mathtt{read}(\mathbf{n_B}+0)^{\mathtt{S}} \cdot \mathtt{rlb}^{\mathtt{S}} \cdot \mathtt{rlb}^{\mathtt{S}}$$

These two traces have the same non-speculative projection $\mathtt{call}\ \mathtt{get}\ 42?^{\mathtt{S}} \cdot \mathtt{if}(\mathbf{0})^{\mathtt{S}}$ but they differ in the observation associated with the branch instruction from line 5 (which is $\mathtt{if}(\mathbf{0})^{\mathtt{U}}$ in $\mathtt{t}_{\Omega^0}$ and $\mathtt{if}(\mathbf{1})^{\mathtt{U}}$ in $\mathtt{t}_{\Omega^1}$). Therefore, they are a counterexample to RSNI. As a result, MVCC violates *RSNIP* since it does not preserve RSNI. ⊡

APPENDIX B
SLH DETAILS

Example 5 (SLH in action)**.** Consider again the Spectre v1 snippet from Listing 1. Clang with SLH enabled compiles the program into the (simplified) assembly in Listing 7.

```
1   mov rax, rsp // load predicate bit from stack pointer
2   sar rax, 63 // initialize mask (0xF...F if left−most bit of rax is 1)
3   mov edx, size // load size
4   cmp rdx, rdi // compare size and y
5   jbe ELSE // jump if out−of−bound
6 THEN:
7   cmovbe rax, rcx // set mask to −1 if out−of−bound
8   movzx ecx, [A + rdi] // load A[y]
9   or rcx, rax // mask A[y]
10  mov cl, [B + rcx] // load B[mask(A[y])]
11  or cl, al // mask B[mask(A[y])]
12  mov temp, cl // assignment to temp
13  jmp END
14 ELSE:
15  cmova rax, −1 // set mask to −1 if in bound
16 END:
17  shl rax, 47
18  or rsp, rax // store predicate bit on stack pointer
19  ret
```

Listing 7. Compiled version of Listing 1 produced by Clang with `-x86-speculative-load-hardening` flag enabled.

The masking introduced by SLH is sufficient to avoid speculative leaks. Indeed, if the processor speculates over the branch instruction in line 5 and speculatively executes the first memory access on line 7, the loaded value is masked immediately afterwards (line 8) and it is set to $\mathbf{0xF..F}$. Thus, the second memory access (line 9) will not depend on sensitive information; thereby preventing the leak. ⊡

Example 6 (SLH is not *RSNIP*)**.** Consider the variant of Spectre v1 illustrated in Listing 4. The main difference with the standard Spectre v1 example (Listing 1) is that the first memory access is performed non-speculatively (line 2). Its value, however, is still leaked through the speculatively-executed memory access in line 4. Clang with SLH compiles this code into the snippet of Listing 8.

```
1   mov rax, rsp  // load predicate bit from stack pointer
2   sar rax, 63 // initialize mask (0xF...F if left−most bit of rax is 1)
3   movzx edx, [A + rdi]  // load A[y]
4   or edx, eax  // mask A[y]
5   mov x, edx  // assignment to x
6   mov esi, size  // load size
7   cmp rsi, rdi  // compare size and y
8   jbe ELSE  // jump if out−of−bound
9 THEN:
10  cmovbe rax, −1  // set mask to −1 if out−of−bound
11  mov cl, [B + rdx]  // load B[x]
12  or cl, al  // mask B[x]
13  mov temp, cl  // assignment to temp
14  jmp END
15 ELSE:
16  cmova rax, −1  // set mask to −1 if in−bound
17 END:
18  shl rax, 47
19  or rsp, rax  // store predicate bit on stack pointer
20  ret
```

Listing 8. Compiled version of Listing 4 produced by Clang with `-x86-speculative-load-hardening` flag enabled.

In the compiled code, the value of A[y] is hardened using the mask retrieved from the stack pointer (line 4). As a result, if the get function is invoked non-speculatively, then the mask is set to $0x0..0$ and the value of A[y] is not protected. Therefore, speculatively executing the load in line 11 may still leak the value of A[y] speculatively, which will be different in traces generated from different, low-equivalent states. ⊡

Example 7 (Non-interprocedural SLH is not *RSNIP*)**.** The program of Listing 9 splits the memory accesses of A and B of the classical 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 9. Inter-procedural variant of the Spectre v1 snippet [42].

Intuitively, 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 0 upon calling the function and therefore the memory access corresponding to B[x] is not masked and it leaks the value of x (which is equivalent to A[y]). ⊡

Appendix C
Formalisation of Code Snippets

Generally, n_A, n_B, n_A and n_B indicate the addresses of arrays A and B in the source and target heaps respectively. Assume variable size is passed through location *1*.

1) Snippet of Listing 1:

$$\begin{array}{l}\mathsf{get(y)} \mapsto \mathsf{let\ size = rd\ 1\ in\ ifz\ y < size}\\ \quad \mathsf{then\ let\ x = rd_p\ n_A + y\ in\ let\ temp = rd\ n_B + x\ in\ skip}\\ \quad \mathsf{else\ skip}\end{array}$$

2) Snippet of Listing 2:

$$\begin{array}{l}\mathsf{get(y)} \mapsto \mathsf{let\ size = rd\ 1\ in\ ifz\ y < size}\\ \quad \mathsf{then\ let\ x = rd_p\ n_A + y\ in\ let\ temp = rd\ n_B + x\ in\ skip}\\ \quad \mathsf{else\ let\ x = rd_p\ n_A + y\ in\ let\ temp = rd\ n_B + x\ in\ skip}\end{array}$$

3) Snippet of Listing 5:

$$\begin{array}{l}\mathbf{get(y)} \mapsto \mathbf{let\ size = rd\ 1\ in\ ifz\ y < size}\\ \quad \mathbf{then\ lfence;\ let\ x = rd_p\ n_A + y\ in}\\ \quad\quad \mathbf{let\ temp = rd\ n_B + x\ in\ skip \quad else\ skip}\end{array}$$

4) Snippet of Listing 3:

$$\begin{array}{l}\mathsf{get(y)} \mapsto \mathsf{let\ size = rd\ 1\ in\ ifz\ y < size}\\ \quad \mathsf{then\ let\ x = rd_p\ n_A + y\ in\ ifz\ x == 0}\\ \quad\quad \mathsf{then\ let\ temp = rd\ n_B + 0\ in\ skip}\\ \quad\quad \mathsf{else\ skip \quad else\ skip}\end{array}$$

5) Snippet of Listing 6:

$$\begin{array}{l}\mathbf{get(y)} \mapsto\\ \quad \mathbf{let\ size = rd\ 1\ in\ ifz\ y < size}\\ \quad\quad \mathbf{then\ let\ x = rd_p\ n_A + y\ in\ ifz\ x == 0}\\ \quad\quad\quad \mathbf{then\ lfence;\ let\ temp = rd\ n_B + 0\ in\ skip}\\ \quad\quad\quad \mathbf{else\ skip \quad else\ skip}\end{array}$$

6) Snippet of Listing 7: Here we are saving the predicate bit in location -1 so source n_a is stored at $n_a - 1$, which is the address that gets calculated in the reads.

$$\begin{array}{l}\mathbf{get(y)} \mapsto\\ \quad \mathbf{let\ size = rd\ 1\ in\ let\ x_g{=}y < size\ in\ ifz\ x_g}\\ \quad\quad \mathbf{then\ let\ x = rd_p\ -1\ in\ -1 :=_p x \vee \neg x_g;}\\ \quad\quad\quad \mathbf{let\ x = rd_p\ n_a + y + 1\ in\ let\ pr = rd_p\ -1\ in}\\ \quad\quad\quad \mathbf{let\ x = 0\ (if\ pr)\ in\ let\ temp = rd\ n_b + x\ in\ skip}\\ \quad\quad \mathbf{else\ let\ x = rd_p\ -1\ in\ -1 :=_p x \vee x_g;\ skip}\end{array}$$

7) Snippet of Listing 4:

$$\begin{array}{l}\mathsf{get(y)} \mapsto \mathsf{let\ size = rd\ 1\ in\ let\ x = rd_p\ n_a + y\ in\ ifz\ y < size}\\ \quad \mathsf{then\ let\ temp = rd\ n_B + x\ in\ skip \quad else\ skip}\end{array}$$

8) Snippet of Listing 8:

$$\begin{array}{l}\mathbf{get(y)} \mapsto\\ \quad \mathbf{let\ size = rd\ 1\ in\ let\ x = rd_p\ n_a + y + 1\ in}\\ \quad \mathbf{let\ pr = rd_p\ -1\ in\ let\ x = 0\ (if\ pr)\ in\ ifz\ y < size}\\ \quad\quad \mathbf{then\ let\ x_f = rd_p\ -1\ in\ -1 :=_p x_f \vee \neg x_g;}\\ \quad\quad\quad \mathbf{let\ temp = rd\ n_b + x\ in\ skip}\\ \quad\quad \mathbf{else\ let\ x_f = rd_p\ -1\ in\ -1 :=_p x_f \vee x_g;\ skip}\end{array}$$

9) Snippet of Listing 9: In this case the predicate bit is stored in each function in a local variable pr, so heap accesses are not shifted by 1 in the target. Below is the source code and its compiled counterpart.

$$\begin{array}{l}\mathsf{get(y)} \mapsto \mathsf{let\ size = rd\ 1\ in\ let\ x = rd_p\ n_a + y\ in\ ifz\ y < size}\\ \quad \mathsf{then\ call\ get2\ x \quad else\ skip}\\ \mathsf{get2(x)} \mapsto \mathsf{let\ temp = rd\ n_B + x\ in\ skip}\end{array}$$

$$\begin{array}{l}\mathbf{get(y)} \mapsto\\ \quad \mathbf{let\ pr{=}1\ in\ let\ size = rd\ 1\ in\ let\ x = rd_p\ n_a + y\ in}\\ \quad \mathbf{let\ x_g{=}y < size\ in\ ifz\ x_g}\\ \quad\quad \mathbf{then\ let\ pr{=}pr \vee \neg x_g\ in\ call\ get2\ x}\\ \quad\quad \mathbf{else\ let\ pr{=}pr \vee x_g\ in\ skip}\\ \mathbf{get2(x)} \mapsto \mathbf{let\ pr{=}1\ in\ let\ temp = rd\ n + b + x\ in\ skip}\end{array}$$

Appendix D
Formal Details that SLH is not *RSNIP*

In the following, assume that the compilation of A (i.e., n_a or $n_a - 1$) contains a value v_a and its low-equivalent counterpart contains v'_a. As before, assume size is 4 and y is 8. We indicate the two traces for the two low-equivalent states as t and t' respectively and highlight in yellow where they differ. Note that these traces contain more heap actions, specifically those required to read and write the predicate bit when it is stored on the heap (location -1).

The attacker is the same:

$$A^8 \stackrel{\text{def}}{=} \mathbf{main(x) \mapsto call\ get\ 8; return;}$$

1) SLH is not RSNIP: Below are the two different target traces for the code of Section C-8.

$$\begin{aligned} t' = {} & \mathtt{call\ get\ 8?}^{S} \cdot \mathtt{read}(1)^{S} \cdot \mathtt{read}(-(n_a - 1 + 8 + 1))^{S} \cdot \\ & \mathtt{read}(-1)^{S} \cdot \mathtt{if}(1)^{S} \cdot \mathtt{read}(-1)^{S} \cdot \mathtt{write}(-1)^{S} \cdot \end{aligned}$$

$$\mathtt{read}(\mathbf{n_b}+\mathbf{v_a})^{\mathbf{U}} \cdot \mathtt{rlb}^{\mathbf{S}} \cdot \mathtt{read}(-1)^{\mathbf{S}} \cdot \mathtt{write}(-1)^{\mathbf{S}} \cdot \mathtt{ret!}^{\mathbf{S}}$$

$$\mathbf{t}' = \mathtt{call\ get\ 8?}^{\mathbf{S}} \cdot \mathtt{read}(1)^{\mathbf{S}} \cdot \mathtt{read}(-(\mathbf{n_a}-1+8+1))^{\mathbf{S}} \cdot \mathtt{read}(-1)^{\mathbf{S}} \cdot \mathtt{if}(1)^{\mathbf{S}} \cdot \mathtt{read}(-1)^{\mathbf{S}} \cdot \mathtt{write}(-1)^{\mathbf{S}} \cdot \mathtt{read}(\mathbf{n_b}+\mathbf{v'_a})^{\mathbf{U}} \cdot \mathtt{rlb}^{\mathbf{S}} \cdot \mathtt{read}(-1)^{\mathbf{S}} \cdot \mathtt{write}(-1)^{\mathbf{S}} \cdot \mathtt{ret!}^{\mathbf{S}}$$

$$\mathbf{t}\restriction_{nse} = \mathbf{t}'\restriction_{nse} = \mathtt{call\ get\ 8?}^{\mathbf{S}} \cdot \mathtt{read}(1)^{\mathbf{S}} \cdot \mathtt{read}(-(\mathbf{n_a}-1+8+1))^{\mathbf{S}} \cdot \mathtt{if}(1)^{\mathbf{S}} \cdot \mathtt{read}(-1)^{\mathbf{S}} \cdot \mathtt{write}(-1)^{\mathbf{S}} \cdot \mathtt{ret!}^{\mathbf{S}}$$

2) Inter-procedural SLH is not RSNIP: Below are the two different target traces for the code of Section C-9.

$$\mathbf{t}' = \mathtt{call\ get\ 8?}^{\mathbf{S}} \cdot \mathtt{read}(1)^{\mathbf{S}} \cdot \mathtt{read}(-(\mathbf{n_a}+8))^{\mathbf{S}} \cdot \mathtt{if}(1)^{\mathbf{S}} \cdot \mathtt{read}(\mathbf{n_b}+\mathbf{v_a})^{\mathbf{U}} \cdot \mathtt{rlb}^{\mathbf{S}} \cdot \mathtt{ret!}^{\mathbf{S}}$$

$$\mathbf{t}' = \mathtt{call\ get\ 8?}^{\mathbf{S}} \cdot \mathtt{read}(1)^{\mathbf{S}} \cdot \mathtt{read}(-(\mathbf{n_a}+8))^{\mathbf{S}} \cdot \mathtt{if}(1)^{\mathbf{S}} \cdot \mathtt{read}(\mathbf{n_b}+\mathbf{v'_a})^{\mathbf{U}} \cdot \mathtt{rlb}^{\mathbf{S}} \cdot \mathtt{ret!}^{\mathbf{S}}$$

$$\mathbf{t}\restriction_{nse} = \mathbf{t}'\restriction_{nse} = \mathtt{call\ get\ 8?}^{\mathbf{S}} \cdot \mathtt{read}(1)^{\mathbf{S}} \cdot \mathtt{read}(-(\mathbf{n_a}+8))^{\mathbf{S}} \cdot \mathtt{if}(1)^{\mathbf{S}} \cdot \mathtt{ret!}^{\mathbf{S}}$$

Appendix E
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 1 (false) in a local variable pr. Compiled code updates pr as before after every branching instruction. To ensure that pr correctly captures whether we are misspeculating, we place an lfence as the first instruction of every compiled function, otherwise the same vulnerability of Example 7 would arise.

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

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

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

Theorem 9 (The NISLH compiler is *RSSC*)**.** $\vdash [\![\cdot]\!]^s_n : \mathit{RSSC}$

Appendix F
Stronger Results for lfence Insertion

Beyond Safety: We believe $[\![\cdot]\!]^f$ preserves more than just a safety property (RSS) since compiled code has the exact same behaviour of its source counterpart. Abate *et al.* [3, 4] provide statements for compilers that preserve more than safety, e.g., arbitrary hyperproperties [20] and we could prove that for $[\![\cdot]\!]^f$. However, for the purpose of showing absence of speculative leaks what we prove is sufficient.

Beyond Cache Leaks: We also study whether this compiler allows information to leak through side channels other than the cache (which is the one we model).

For this, consider language **E**, an extension of **T** that leaks additional information when speculating. The speculative semantics of **E** models other leakage η through side channels when statements other than lfence are executed (Rule E-speculate-eta). Intuitively, η could be the time required by certain operations (e.g., costly multiplications) or the prefetching of instructions, but we leave it abstract [7, 10, 11].

(E-speculate-eta)

$$\frac{\mathbf{\Omega_v} \xrightarrow{\epsilon} \mathbf{\Omega'_v} \qquad \mathbf{\Omega_v} \equiv \mathbf{C}, \mathbf{H_v}, \overline{\mathbf{B_v}} \triangleright \mathbf{s}; \mathbf{s'} \qquad \mathbf{s} \not\equiv \mathbf{ifz} \cdots \text{ nor } \mathbf{lfence}}{\overline{\mathbf{\Phi_v}} \cdot (\mathbf{\Omega_v}, \mathbf{n}+1) \overset{\eta}{\rightsquigarrow} \overline{\mathbf{\Phi_v}} \cdot (\mathbf{\Omega'_v}, \mathbf{n})}$$

Denote with $[\![\cdot]\!]^f_{\mathbf{E}}$ the lfence-inserting compiler to **E** (which is exactly like $[\![\cdot]\!]^f$). In code compiled with $[\![\cdot]\!]^f_{\mathbf{E}}$, *no statement* is executed speculatively, so compiled code has no leak since not even η is emitted. Formally, we also prove that $[\![\cdot]\!]^f_{\mathbf{E}}$ is *RSSC* with the same trace relation. Since this relation does not mention η, this proof is only possible if compiled code never emits η. This is not true for the case of SLH below (and for analogous countermeasures), since there some compiled statements are executed speculatively. This is, after all, expected, since SLH focusses on protecting the cache side channel and not, for example, timing side-channels.

TODO: confusing ?

Appendix G
A Complete Insight on the Proofs

This section describes how to prove that compiler countermeasures are secure (i.e, *RSSC*), starting with $[\![\cdot]\!]^f$.

As mentioned in Section IV, to prove that a compiler is *RSSC* we need to backtranslate target attackers **A** into source ones A. Our setup has very similar source and target languages to enable this kind of backtranslation; the alternative would have been to rely on the target trace in order to build A. In this case, the backtranslation function ($\langle\!\langle\cdot\rangle\!\rangle$) homomorphically translates target heaps, functions, statements etc. into source ones as below:

$$\langle\!\langle \mathbf{H}; \overline{\mathbf{F}} \rangle\!\rangle^f_c = \langle\!\langle \mathbf{H} \rangle\!\rangle^f_c; \langle\!\langle \overline{\mathbf{F}} \rangle\!\rangle^f_c$$

$$\langle\!\langle \varnothing \rangle\!\rangle^f_c = \varnothing$$
$$\langle\!\langle \mathbf{H}; \mathbf{n} \mapsto \mathbf{v} : \boldsymbol{\sigma} \rangle\!\rangle^f_c = \langle\!\langle \mathbf{H} \rangle\!\rangle^f_c; \langle\!\langle \mathbf{n} \rangle\!\rangle^f_c \mapsto \langle\!\langle \mathbf{v} \rangle\!\rangle^f_c : \langle\!\langle \boldsymbol{\sigma} \rangle\!\rangle^f_c$$

$$\langle\!\langle \mathbf{f(x)} \mapsto \mathbf{s; return;} \rangle\!\rangle^f_c = \mathtt{f(x)} \mapsto \langle\!\langle \mathbf{s} \rangle\!\rangle^f_c\mathtt{; return;}$$

$$\langle\!\langle \boldsymbol{\sigma} \rangle\!\rangle^f_c = \sigma$$

$$\langle\!\langle \mathbf{n} \rangle\!\rangle^f_c = \mathtt{n}$$
$$\langle\!\langle \mathbf{e} \oplus \mathbf{e'} \rangle\!\rangle^f_c = \langle\!\langle \mathbf{e} \rangle\!\rangle^f_c \oplus \langle\!\langle \mathbf{e'} \rangle\!\rangle^f_c$$
$$\langle\!\langle \mathbf{e} \otimes \mathbf{e'} \rangle\!\rangle^f_c = \langle\!\langle \mathbf{e} \rangle\!\rangle^f_c \otimes \langle\!\langle \mathbf{e'} \rangle\!\rangle^f_c$$

$$\langle\!\langle \mathbf{skip} \rangle\!\rangle^f_c = \mathtt{skip}$$

$$
\begin{aligned}
\langle\!\langle \mathbf{s}; \mathbf{s}' \rangle\!\rangle_c^f &= \langle\!\langle \mathbf{s} \rangle\!\rangle_c^f; \langle\!\langle \mathbf{s}' \rangle\!\rangle_c^f \\
\langle\!\langle \mathbf{let\ x = e\ in\ s} \rangle\!\rangle_c^f &= \mathsf{let\ x =} \langle\!\langle \mathbf{e} \rangle\!\rangle_c^f \mathsf{\ in\ } \langle\!\langle \mathbf{s} \rangle\!\rangle_c^f \\
\langle\!\langle \mathbf{ifz\ e\ then\ s\ else\ s}' \rangle\!\rangle_c^f &= \mathsf{ifz\ } \langle\!\langle \mathbf{e} \rangle\!\rangle_c^f \mathsf{\ then\ } \langle\!\langle \mathbf{s} \rangle\!\rangle_c^f \mathsf{\ else\ } \langle\!\langle \mathbf{s}' \rangle\!\rangle_c^f \\
\langle\!\langle \mathbf{call\ f\ e} \rangle\!\rangle_c^f &= \mathsf{call\ f\ } \langle\!\langle \mathbf{e} \rangle\!\rangle_c^f \\
\langle\!\langle \mathbf{e := e}' \rangle\!\rangle_c^f &= \langle\!\langle \mathbf{e} \rangle\!\rangle_c^f := \langle\!\langle \mathbf{e}' \rangle\!\rangle_c^f \\
\langle\!\langle \mathbf{let\ x = rd\ e\ in\ s} \rangle\!\rangle_c^f &= \mathsf{let\ x = rd\ } \langle\!\langle \mathbf{e} \rangle\!\rangle_c^f \mathsf{\ in\ } \langle\!\langle \mathbf{s} \rangle\!\rangle_c^f \\
\langle\!\langle \mathbf{lfence} \rangle\!\rangle_c^f &= \mathsf{skip} \\
\langle\!\langle \mathbf{let\ x = e\ (if\ e')\ in\ s} \rangle\!\rangle_c^f &= \begin{array}{l} \mathsf{ifz\ } \langle\!\langle \mathbf{e}' \rangle\!\rangle_c^f \\ \quad \mathsf{then\ let\ x=} \langle\!\langle \mathbf{e} \rangle\!\rangle_c^f \mathsf{\ in\ skip} \\ \quad \mathsf{else\ skip} \\ \mathsf{;} \langle\!\langle \mathbf{s} \rangle\!\rangle_c^f \end{array}
\end{aligned}
$$

Since our compilers are devised for essentially the same languages (or at least, languages with the same notions of attacker), we can define a single backtranslation to use for the security of all compilers we define.

To prove the compiler is $RSSC$ we need to show that given a trace produced by the execution of attacker and compiled code, the backtranslated attacker and the source code produce a related trace (according to the trace relation of Section IV). This can be broken down in a sequence of canonical steps:

- we first set up a cross-language relation between source and target states;
- we prove that initial states are related;
- we prove that reductions preserve this relation and generate related traces.
  We reason about reductions depending on whether the pc that is triggering the reduction is in attacker or in program code.

The state relation we use is strong: a source state is related to a target state if the latter is a singleton stack and all the subpart of the state are identical. That is, the target state must not be speculating (starting speculation adds a state to the stack, which is not a singleton in that case), the heaps bind the same locations to the same values, the bindings bind the same variables to the same values.

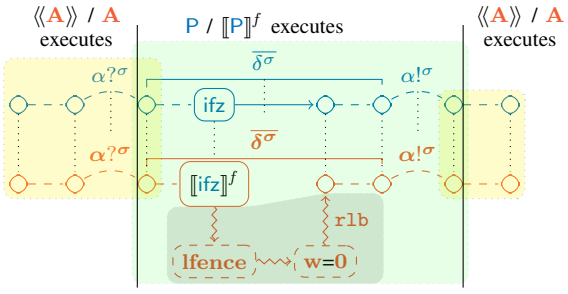


Figure 4. A diagram depicting the proof that $[\![\cdot]\!]^f$ is $RSSC$.

We depict our proof approach for $[\![\cdot]\!]^f$ in Figure 4. The top half of the picture represents the source reductions: source states (circles) perform multiple steps (dashed lines) and reduce producing traces (annotations on reductions). We highlight the single reduction caused by the execution of an ifz statement since that has relevance in the target language.

The bottom half of the picture represents the target states and their reductions, here we have an additional kind of program states: dashed ones. Intuitively, these states are not related to any source state, while other states are.

This is symbolised by black dotted connections between source and target states. The same kind of connection between source and target actions indicates that these actions are related.

In our setup, execution either happens with the pc in attacker code or in component code. We now describe how to reason about these reductions.

To reason about attacker code, we use a lock-step simulation: we show that starting from related states, if A does a step, then its backtranslation $\langle\!\langle \mathbf{A} \rangle\!\rangle$ does the same step and ends up in related states. This is what happens in the yellow areas in the picture.

To reason about component code, we adapt a reasoning commonly used in compiler correctness results [10, 38]. That is, if s steps and emits a trace, then $[\![\mathsf{s}]\!]^f$ does one or more steps and emits a trace such that

- the ending states are related;
- the emitted source and target traces are related;

This is what happens in the green area, recall that related traces are connected by black-dotted lines.

The only place where this proof is not straightforward is the case for compilation of ifz i.e., the statement that triggers speculation in T. This is what happens in the grey area. When observing target-level executions for $[\![\mathsf{ifz}]\!]^f$, we see that the cross-language state relation is temporarily broken. After the $[\![\mathsf{ifz}]\!]^f$ is executed, speculation starts, so the stack of target states is not a singleton and therefore the cross-language state relation cannot hold. However, if we unfold the reductions, we see that compiled code immediately triggers an lfence, which rolls the speculation back (the speculation window w is 0) reinstating the cross-language state relation. Thus, for the case of ifz to go through, we see that the target effectively does more steps than the source (it starts speculation, it executes the lfence and then it rolls speculation back) before ending up in a state related to the source one.

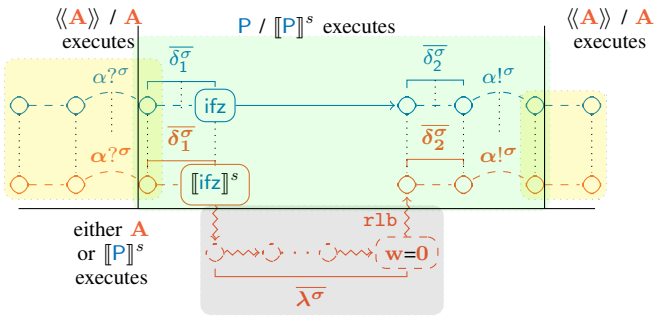


Figure 5. A diagram depicting the proof that $[\![\cdot]\!]^s$, $[\![\cdot]\!]^{ss}$ and, $[\![\cdot]\!]^s_n$ are $RSSC$.

This is the part where the proofs of the SLH-related countermeasures gets more complicated (Figure 5), though the general structure remains unchanged. In compiled code speculation is not rolled back immediately after the then or else branch start executing. Instead, execution can continue for ω steps, spanning both attacker and compiled code and generating a trace $\overline{\lambda^\sigma}$. Our proof here relies on an auxiliary lemma stating the following:

- in the target, speculation lasts at most ω steps and then it will be rolled back;
- after the rollback, the strong state relation we need is reinstated;
- during this speculation any trace produced in the target is related to the empty source trace.
  This is needed because such a relation is only possible when all actions in the target trace $\overline{\lambda^\sigma}$ are tainted S: i.e., they do not leak.

Ensuring that all target actions are S is achieved through declaring a property on target (speculating) states and prove that any speculating transition *preserves* that property. Specifically, the property is that the bindings always contain S values. From this property we can easily see that any generated action is S. To prove that this property holds right after speculation, we need

- pr correctly captures whether speculation is ongoing or not;
- the mask used by the compiler taints the variable it is applied to as S.

As already shown, both conditions hold for $[\![\cdot]\!]^s$, $[\![\cdot]\!]^{ss}$ and, $[\![\cdot]\!]^s_n$, so we can conclude that they are $RSSC$.

A. 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 insecurity of 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 Examples 6 and 7. We believe the insights of this proof technique can guide proofs of (in)security of other countermeasures too.

Appendix H
The Spectre v2 Case

This section describes how to apply our methodology to reason about countermeasures to the Spectre v2 attack. The Spectre v2 attack relies on a different kind of speculative execution than what presented in this paper, namely it relies on speculation of indirect jumps. 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.

TODO: how does the rollback work ?

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 H-A);
- give a speculative reduction to jump in T such that the location where to jump is nondeterministically chosen; this will be the start of speculation (Section H-B);
- change the call/return semantics in order to model retpolines, i.e., have the return address explicit (Section H-C).

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

We would replace Rule E-T-speculate-if 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 nondeterministically 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 G-A. 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.

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] 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.

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

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

[29] 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.

[30] 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.

[31] 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.

[32] 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 2018.

[33] 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.

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

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

[36] 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.

[37] 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.

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

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

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

[41] 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.

[42] 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.

[43] 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.

[44] 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 Informa-*

tion Security and Cryptology, pages 156–168. Springer, 2005.

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

[46] 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.

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

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

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

[50] 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.

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

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

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

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

[55] 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.

[56] 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.

[57] 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.

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

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

[60] Tachio Terauchi and Alex Aiken. Secure information flow as a safety problem. In *SAS '05*, 2005.

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

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

[63] 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 I
ON SPECTRE V2 - TO DISCUSS

TODO: Outline dettagliata di come supportare Spectre v2 – l'unico caso veramente interessante from a secure compilation perspective

A. Language

We need to add to the language some sort of `goto` expression. The easiest way is to add *numeric labels* to all instructions. Then, a `goto` e statement evaluates the expression e, uses its value to determine which label to invoke, jumps to it and continues the execution.

TODO: We might need this assumption: "(for simplicity, we assume that one can jump only at beginning of function without parameters)". In general, we no longer have a well-bracketed control flow. Does this matters for proofs?

In L, the `goto` e statement behaves as expected. In T, we perform the following changes:

1) We extend the speculative semantics with a return stack buffer, i.e., a (bounded) stack of return addresses. Whenever we execute a call or icall instruction, we push $pc+1$ to the stack, where pc is the address of current instruction. Whenever we return from a function, we pop an element from the buffer.
   TODO: alternatives:
   - goto x: goes to statement number x in the current function - goto x and label x: goes to label called x anywhere in the codebase
   TODO: Mh, I'll have to think about it. I guess this would keep well-bracketed control flow wrt call/ret, no?
2) We add a designated variable identifier `rsp` storing the return address. Whenever we execute a call statement, `rsp` is set to $pc+1$.
   Assumption: component code cannot directly modify the `rsp` register.
3) Call instructions do not trigger speculative execution.
4) For `goto` e instructions, if e is not a constant value, we non-deterministically pick one of the labels, start a speculative transaction, and continue the mispredicted execution for a fixed number of steps. Afterwards, we roll back and continue the execution using the value of e as next program counter.
   TODO: is this really needed? isn't the call speculation enough?
   TODO: We're not speculative over calls, only returns
5) Return instructions trigger a speculative execution as follows: they speculate by returning to the top of RSB rather than to `rsp`. Concretely, whenever the top of RSB is different from `rsp`, we speculate by first following the address at the head of RSB for a bounded number of steps, rolling back and then continuing with `rsp`.
6) We add a **pause** statement to T that loops indefinitely. That is, once a program gets to a **pause** statement, it is stuck.

B. Criteria

No changes needed here (except that a program's behaviour in T is now a *set* of traces rather than a singleton).

C. Countermeasure

Spectre v2 (also called Spectre-BTB) [36] exploits speculation over indirect jump instructions.[8]

The *retpoline* compiler-level countermeasure [31] replaces indirect jumps with return-based trampoline (thus the name retpoline) that leads to effectively dead code. As a result, the speculated jump executes no code and thus cannot leak anything.

Here we illustrate how the retpoline countermeasure works on an indirect jump.

Consider an indirect jump `jmp *%rax`. A CPU can speculate over the value of `*%rax` when executing the jump. Concretely, the predicted value depends on the content of several prediction data structures (e.g., the branch target buffer) which can be poisoned by an attacker. Hence, potentially the prediction of `*%rax` can point to an arbitrary instruction address.

An indirect jump `jmp *%rax` is compiled as follows (with retpoline):

```
1       call l2
2  l1: pause
3      lfence
4      jmp l1
5  l2: mov %rax, (%rsp)
6      ret
```

The countermeasure works as follows:

- At line 1, we push the address of `l1` (line 2) on the stack and in the return stack buffer (RSB). Then, we jump to `l2` (line 5).
- At line 5, we overwrite the return addressed stored in the stack (`(%rsp)`) with the target of the indirect jump (`*%rax`). Now, the RSB points to `l2` while `rsp` points to `*%rax`.
- At line 6, if the CPU speculates over the `ret` instruction, then the CPU uses the latest RSB entry (i.e., `l1` created at line 1) to continue the execution and jumps at line 2. When the CPU realizes that the speculation was wrong (or if the `ret` does not trigger any speculation), the execution correctly continues at `*%rax`.
- At line 2, the three instructions `pause; lfence; jmp l1` effectively lock the execution in a non-terminating loop (which the processor cannot leave even speculatively). We remark that instructions at lines 2–4 are executed only in case of mis-speculation. Hence, these three instructions effectively block the speculative execution of arbitrary code.

[8] In assembly, indirect jumps results from branch instructions whose target is read from a register or memory, e.g., such as a `jmp eax` instruction. In compiled languages, using function pointers and calls to object functions often result in indirect jumps.

D. Modeling retpoline

We should be able to add a retpoline-style mitigation to the LFENCE/SLH compiler as follows.

$$[\![\textbf{goto e}\ ]\!] = \textbf{call aux(e); pause}$$

where $\textbf{aux(x)} \mapsto \{\textbf{rsp} = \textbf{x}; \textbf{ret}\}$.

TODO: it feels more natural that:
the compiler defines a new function "div(x) = call div(0)" that diverges.
goto e is compiled to call aux ; call div
non-issue: there's an action 'call div' but that is S so it's droppable.

TODO: Fine with replacing pause with div(0). Still, we'd have to add a rule anyway, no? I don't think we currently support divergence on div(0)

The intuition behind the compiler security as follows. When we execute $\textbf{call aux(e)}$, we evaluate the target address e, we push the address of the **pause** statement on the RSB, and we set **rsp** to the address of **pause**. Then, we execute **aux** we set **rsp** to the value of e, i.e., the intended target of **goto** e. When executing the **ret** instruction we speculate by returning to the address at the head of RSB, that is, we execute the **pause** statement and we are stuck until the speculative window reaches 0. As a result, during speculative execution we produce only observations labelled as S (because the speculative ret only outputs the address of $\textbf{call aux(e);} + 1$ which is S). Afterwards, we jump to the address in **rsp**, i.e., the intended target of **goto** e, and we continue the execution. Further observations are labelled as S since we are not speculating.

TODO: need private function