Exorcising Spectres with Secure Compilers

Marco Patrignani
Stanford University
CISPA Helmholtz Center for Information Security
mp@cs.stanford.edu

Marco Guarnieri
IMDEA Software Institute
marco.guarnieri@imdea.org

ABSTRACT

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

This paper develops formal foundations for reasoning about the security of these defenses. For this, it proposes a framework of secure compilation criteria that characterise when compilers produce code resistant against Spectre v1 attacks. With this framework, this paper performs a comprehensive security analysis of countermeasures against Spectre v1 attacks implemented in major compilers, deriving the first security proofs of said countermeasures.

This paper uses a blue, sans-serif font for elements of the source language and an orange, bold font for elements of the target language. Elements common to all languages are typeset in a black, italic font (to avoid repetitions). For a better experience, please print or view this in colour [48].

1 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.

Exploiting microarchitectural leaks caused by speculative execution leads to Spectre attacks [35, 37, 38, 41, 57]. Compilers support a number of countermeasures, e.g., the insertion of `lfence` speculation barriers [31] and speculative load hardening [16], that *can* mitigate leaks introduced by speculation over branch instructions like those exploited in the Spectre v1 attack [37].

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

In this paper, we propose a novel secure compilation framework for reasoning about speculative execution attacks and we use it to provide the first precise characterization of security for a comprehensive class of compiler countermeasures against Spectre v1 attacks. Let us now discuss our contributions more in detail:

Anonymous Submission to ACM CCS 2021, ,
2021. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

► We present a secure compilation framework tailored towards reasoning about speculative execution attacks (Section 2). The distinguishing feature of our framework is that compilers translate programs from a source language L, with a standard imperative semantics, into a target language T equipped with a speculative semantics capturing the effects of speculatively-executed instructions. 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 the features of T cannot be exploited.
Our framework encompasses two different security models for speculative execution: (1) *(Strong) speculative non-interference* [27] (SNI), which considers *all* leaks derived from speculatively-executed instructions as harmful, and (2) *Weak speculative non-interference* [28], which considers harmful *only* leaks of speculatively-accessed data.

► We introduce *speculative safety* (SS, Section 3), 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 precisely characterize the security guarantees of SS by showing that strong (resp. weak) SS over-approximates strong (resp. weak) SNI.

► We define two novel secure compilation criteria: *Robust Speculative Safety Preservation* (*RSSP*) and *Robust Speculative Non-Interference Preservation* (*RSNIP*, Section 4). These criteria respectively ensure that compilers preserve (strong or weak) SS and SNI *robustly*, i.e, even when linked against arbitrary (potentially malicious) code. Satisfying these criteria implies that compilers correctly place countermeasures to prevent speculative leaks. However, *RSSP* requires preserving a safety property (SS) and it is 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] to reason about a concrete security property, that is, the absence of speculative leaks.

► Using our framework, we perform a comprehensive security analysis of countermeasures against Spectre v1 attacks implemented in major C compilers (Section 5). Specifically, we focus on (1) automated insertion of `lfences` (implemented in the Microsoft Visual C++ and the Intel ICC compilers [33, 47]), and (2) speculative load hardening (SLH, implemented in Clang [16]). We prove 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. This prevents Spectre-style attacks, but compiled programs might still speculatively leak data accessed non-speculatively.
– The non-interprocedural variant of SLH violates weak *RSNIP* and is thus insecure.
– Our novel variant of SLH, called strong SLH, provides strong *RSNIP* and blocks all speculative leaks.

All our security proofs follow a common methodology (see Section 4.3) whose key insight is that proving a countermeasure to be *RSSP* is sufficient to ensure its security since SS over-approximates SNI. This allows us to leverage SS to simplify our proofs.

We conclude by discussing limitations and extensions of our approach (Section 6) and related work (Section 7).

For simplicity, we only discuss key aspects of our formal models. Full details and proofs are in the companion report [52].

2 MODELLING SPECULATIVE EXECUTION

To illustrate our speculative execution model, we first introduce Spectre v1 (Listing 1). Using that, we define the threat model that we consider (Section 2.1). Then, we present the syntax of our languages (Section 2.2) and their trace model (Section 2.3). This is followed by the operational semantics of our languages (Section 2.4). Next, we present the source (non-speculative) trace semantics (Section 2.5) and the target (speculative) trace semantics (Section 2.6). This formalisation focuses on the strong SNI model, so we conclude by defining the changes necessary for weak SNI (Section 2.7).

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

Listing 1: The classic Spectre v1 snippet.

Consider the standard Spectre v1 example [37] in Listing 1. Function `get` checks whether the index stored in variable `y` is less than the size of array `A`, stored in the global variable `size`. If so, the program retrieves `A[y]`, multiplies it by the cache line size (here: 512), and uses the result to access array `B`. If `size` is not cached, modern processors predict the guard's outcome and speculatively continue the execution. Thus, line 3 might be executed even if $y \geq$ `size`. When `size` becomes available, the processor checks whether the prediction was correct. If not, it rolls back all changes to the architectural state and executes the correct branch. However, the speculatively-executed memory accesses leave a footprint in the cache, which enables an adversary to retrieve `A[y]` even for $y \geq$ `size`.

2.1 Threat Model

We study compiler countermeasures that translate source programs into (hardened) target programs. In our setting, an attacker is an arbitrary program at target level that is linked against a (compiled) partial program of interest. The partial program (or, *component*) stores sensitive information in a private heap that is not accessible to the attacker. For this, we assume that attacker and component run on separate processes and OS-level memory protection restrict access to the private heap. For example, in Listing 1, the array `A` would be stored in the private heap and the attacker is code that runs before and after function `get`.

While attackers cannot directly access the private heap, they can mount confused deputy attacks [29, 54] to trick components into leaking sensitive information despite the memory protection. 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 code that has no timing side-channels [8, 44] without requiring microarchitectural models. Following [27], we capture this model in our semantics through traces that record the address of all memory accesses (e.g., the address of `B[A[y]*512]` in Listing 1) and the outcome of all control-flow instructions.

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

2.2 Languages L and T

Technically, we have a pair of source and target languages (L and T) for studying security in the strong SNI model and a pair of source and target languages (L$^-$ and T$^-$) for studying weak SNI. Strong (L-T) and weak (L$^-$-T$^-$) languages have the same syntax and a very similar semantics, which differ *only* in the security-relevant observations produced during the computation. We focus this section and the following ones on the strong languages L-T; we introduce the small changes for the weak languages L$^-$-T$^-$ in Section 2.7.

The source (L) and target (T) languages are single-threaded While languages with a heap, a stack to lookup local variables, and a notion of components (our unit of compilation). We focus on such a setting, instead of an assembly-style language like [17, 27], to reason about speculative leaks without getting bogged down in complications like unstructured control flow. This does not limit the power of attackers: since attackers reside in another process, they would not be able to exploit the additional features of assembly languages (e.g., unstructured control flow) to compromise components.

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 $\overline{e}$.

$$\begin{array}{rr} \textit{Programs } W, P ::= H, \overline{F}, \overline{I} & \textit{Codebase } C ::= \overline{F}, \overline{I} \quad \textit{Imports } I ::= f \\ \textit{Functions } F ::= f(x) \mapsto s; \textit{return}; & \textit{Attackers } A ::= H, \overline{F}\,[\cdot] \\ \textit{Heaps } H ::= \varnothing \mid H; n \mapsto v & \text{where } n \in \mathbb{Z} \\ \textit{Expressions } e ::= x \mid v \mid e \oplus e & \textit{Values } v ::= n \in \mathbb{N} \\ \multicolumn{2}{c}{\textit{Statements } s ::= \textit{skip} \mid s; s \mid \textit{let } x = e \textit{ in } s \mid \textit{call } f\ e \mid e := e} \\ \multicolumn{2}{c}{\mid e :=_{pr} e \mid \textit{let } x = \textit{rd } e \textit{ in } s \mid \textit{let } x = \textit{rd}_{pr}\ e \textit{ in } s} \\ \multicolumn{2}{c}{\mid \textit{ifz } e \textit{ then } s \textit{ else } s \mid \textit{let } x = e\ (\textit{if } e) \textit{ in } s \mid \textit{lfence}} \end{array}$$

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 (its functions and imports) as C.

Functions are untyped, and their bodies are sequences of statements s that include standard instructions: skipping, sequencing, let-bindings, function calls, writing the public and the private heap,

reading the public and private heap, conditional branching, conditional assignments and speculation barriers. 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 values v. 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.

2.3 Labels and Traces

Computation steps in L and T are *labelled* with labels λ, which can be the *empty label* ϵ, an *action* $\alpha?$ or $\alpha!$ 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.

$$\mu arch.\ Acts.\ \delta ::= \texttt{read}(n) \mid \texttt{write}(n) \mid \texttt{read}(n \mapsto v) \mid \texttt{write}(n \mapsto v) \mid \texttt{if}(v) \mid \texttt{rlb}$$

$$Actions\ \alpha ::= \texttt{call}\ f\ v \mid \texttt{ret}\ v \qquad Labels\ \lambda ::= \epsilon \mid \alpha? \mid \alpha! \mid \delta$$

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 private heap location n. Dually, the $\texttt{read}(n \mapsto v)$ and $\texttt{write}(n \mapsto v)$ actions denote respectively read and write accesses to the public heap location n where v is the value read from/written to memory. In these actions, locations n model leaks through the data cache whereas values v, which only appear in operations on the public heap, model that attackers have access to the public heap. 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 $\overline{\lambda}$ are sequences of labels. 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 since A cannot access the private heap (this is analogous to other robust safety works [23, 25, 40, 60]).

2.4 Operational Semantics for L and T

Both languages are given a labelled operational semantics that describes how statements execute. This semantics is defined in terms of program states $C, H, \overline{B} \triangleright (s)_{\overline{f}}$ that consist of a codebase C, a heap H, a stack of local variables $\overline{B}$, a statement s, and a stack of function names $\overline{f}$. C is used to look up function bodies, function names $\overline{f}$, which we often omit for simplicity, are used to infer if the code that is executing comes from the attacker or from the component, and this determines the produced labels.

$$Bindings\ B ::= \varnothing \mid B; x \mapsto v \qquad Prog.\ States\ \Omega ::= C, H, \overline{B} \triangleright (s)_{\overline{f}}$$

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 follows judgements $B \triangleright e \downarrow v$ meaning: "according to variables B, expression e reduces to value v." The latter follows judgements $\Omega \xrightarrow{\lambda} \Omega'$ meaning: "state Ω reduces in one step to Ω' emitting label λ."

We remark that values are computed as expected (though we use *0* for true in *ifz* statements; see Rule E-if-true) and expressions access only local variables in B (reading from the heap is treated as a statement); therefore, we omit the expression semantics. Similarly, many of the rules for the statement semantics are standard and thus omitted; the most illustrative ones are given below. We use $|n|$ for the absolute value of n and $H(n)$ to look up the binding for n in H.

$$\frac{B \triangleright e \downarrow 0}{C, H, \overline{B} \cdot B \triangleright ifz\ e\ then\ s\ else\ s' \xrightarrow{(\texttt{if}(0))} C, H, \overline{B} \cdot B \triangleright s} \text{(E-if-true)}$$

$$\frac{B \triangleright e \downarrow n \qquad H(-|n|) = v}{C, H, \overline{B} \cdot B \triangleright let\ x = rd_{pr}\ e\ in\ s \xrightarrow{\texttt{read}(-|n|)} C, H, \overline{B} \cdot B \cup x \mapsto v \triangleright s} \text{(E-read-prv)}$$

$$\frac{B \triangleright e \downarrow n \qquad H = H_1; -|n| \mapsto v'; H_2 \qquad B \triangleright e' \downarrow v \qquad H' = H_1; -|n| \mapsto v; H_2}{C, H, \overline{B} \cdot B \triangleright e :=_{pr} e' \xrightarrow{\texttt{write}(-|n|)} C, H', \overline{B} \cdot B \triangleright skip} \text{(E-write-prv)}$$

The rules of conditionals, read, and write emit the related μarch. actions (from Section 2.3). Specifically, conditionals produce observations recording the outcome of the condition (Rule E-if-true), whereas memory operations produce observations recording the accessed memory address (Rule E-read-prv and Rule E-write-prv).

2.5 Non-speculative Semantics for L

We now define the non-speculative semantics of L, which describes how (whole) programs behave when executed on a processor without speculative execution. A component P and an attacker A can be linked to obtain a whole program $W \equiv A[P]$ that contains the functions and heaps of A and P. Only whole programs can run, and a program is whole only if it defines all functions that are called and if the attacker defines all the functions in the interfaces of P.

For this, we define the big-step semantics $\Rightarrow$ of L, which concatenates single steps (defined by $\rightarrow$) into multiple ones and single labels into traces. The judgement $\Omega \overset{\overline{\lambda}}{\Longrightarrow} \Omega'$ is read: "state Ω emits trace $\overline{\lambda}$ and becomes Ω'". The most interesting rule is below. As mentioned in Section 2.3, the trace does not contain μarch. actions performed by the attacker (see the 'then' branch, recall that functions in $\overline{\mathsf{I}}$ are defined by the attacker).

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

Finally, the behaviour $\mathsf{Beh}(\mathsf{W})$ of a whole program W is the trace $\overline{\lambda}$ generated from the $\Rightarrow$ semantics starting from the initial state of W (indicated as $\Omega_0(\mathsf{W})$) until termination. Intuitively, a program's initial state is the main function, which is defined by the attacker.

Example 2.1 (L trace for Listing 1). Consider `size=4`. Trace t_{ns} indicates a valid execution of the code in L (without speculation).

$$\mathsf{t}_{ns} = \texttt{call get 0?} \cdot \texttt{if(0)} \cdot \texttt{read}(\mathsf{n}_A) \cdot \texttt{read}(\mathsf{n}_B + \mathsf{v}_A^0) \cdot \texttt{ret!}$$

We indicate the addresses of arrays `A` and `B` in the L heap with n_A and n_B respectively and the value stored at `A[i]` with v^i_A. ⊡

2.6 Speculative Semantics for T

Our semantics for T models the effects of speculatively-executed instructions. This semantics is inspired by the "always mispredict" semantics of Guarnieri et al. [27], which captures the worst-case scenario (from an information theoretic perspective) independently of the branch prediction outcomes. Whenever the semantics executes a branch instruction, it first mis-speculates by executing the wrong branch for a fixed number 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.

Speculative program states (Σ) are defined as stacks of speculation instances ($\Phi = (\Omega, w)$), each one recording the program state Ω and the remaining speculation window w. 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 size of the reorder buffer.

Speculative States $\Sigma ::= \overline{\Phi}$ *Speculation Instance* $\Phi ::= (\Omega, w)$

The execution of program W starts in state $(\Omega_0\,(W)\,, \bot)$, i.e., in the same initial state that L starts in.

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

$$\frac{\Omega \equiv C, H, \overline{B} \cdot B \triangleright (s; s')_{\overline{f}.f} \qquad s \equiv \text{if } e \text{ then } s'' \text{ else } s'''}{\Omega \xrightarrow{\alpha} \Omega' \qquad C \equiv \overline{F}; \overline{I} \qquad f \notin \overline{I} \qquad j = \min(\omega, n)} \text{ (E-T-speculate-if)}$$
$$\text{if } B \triangleright e \downarrow 0 \text{ then } \Omega'' \equiv C, H, \overline{B} \cdot B \triangleright s'''; s' \text{ else } \Omega'' \equiv C, H, \overline{B} \cdot B \triangleright s''; s'$$
$$\overline{\Phi} \cdot (\Omega, n+1) \overset{\alpha}{\rightsquigarrow} \overline{\Phi} \cdot (\Omega', n) \cdot (\Omega'', j)$$

$$\frac{\Omega \xrightarrow{\lambda} \Omega' \qquad \Omega \equiv C, H, \overline{B} \triangleright s; s' \qquad s \not\equiv \text{ifz} \cdots \text{ and } s \not\equiv \text{lfence}}{\overline{\Phi} \cdot (\Omega, n+1) \overset{\lambda}{\rightsquigarrow} \overline{\Phi} \cdot (\Omega', n)} \text{ (E-T-speculate-action)}$$

$$\frac{\Omega \xrightarrow{\epsilon} \Omega' \qquad \Omega \equiv C, H, \overline{B} \triangleright s; s' \qquad s \equiv \text{lfence}}{\overline{\Phi} \cdot (\Omega, n+1) \overset{\epsilon}{\rightsquigarrow} \overline{\Phi} \cdot (\Omega', 0)} \text{ (E-T-speculate-lfence)}$$

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

Executing a statement updates the program state on top of the state and reduces the speculation window by 1 (Rule E-T-speculate-action). Mis-speculation pushes the mis-speculating state on top of the stack (Rule E-T-speculate-if). Note that speculation does not happen in attacker code (condition $f \notin \overline{I}$, recall that f is the function executing now and $\overline{I}$ are all attacker-defined functions). This is without loss of generality since (1) attackers cannot directly access the private heap, and (2) our security definitions (Section 3) will consider any possible attacker, so the speculative behavior of an attacker (i.e., the speculative execution of the 'wrong branch') will be captured by another one who has the same branches but inverted (e.g., the 'then' code of one attacker is the 'else' code of another). When the speculation window is exhausted (or if the speculation reaches 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).

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

Example 2.2 (T Trace for Listing 1). Consider the same setting as Example 2.1. Trace t_{sp} is a valid execution of the code in T, and therefore with speculation. As before, 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_{sp} = call get 8? · if(1) · read($n_A + 8$)·read($n_B + v^8_A$) · rlb · ret!

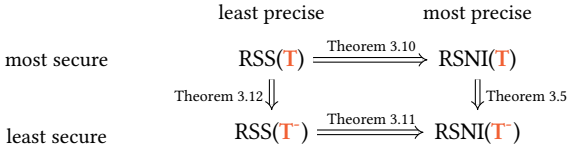
Differently from t_{ns} in Example 2.1, trace t_{sp} contains speculatively executed instructions whose side effects are represented by the actions read($n_A + 8$) and read($n_B + v^8_A$). ⊡

2.7 Weak Languages L⁻ and T⁻

To conclude, we now introduce the weak languages L⁻ and T⁻, which we use to study security in the weak SNI model. Following [28], these languages differ from L and T in a single aspect, that is, in the actions produced by memory reads. Specifically, in L⁻ and T⁻, non-speculatively reading from the private heap produces an action read($n \mapsto v$) that contains the read value v as well as the accessed memory address n. As we show next, this difference allows us to precisely characterize *only* the leaks of transiently loaded data, which are exactly those leaks exploited in speculative disclosure gadgets like Listing 1, rather than all speculative leak.

3 SECURITY DEFINITIONS FOR SECURE SPECULATION

We now present *semantic* security definitions against speculative leaks. We start by presenting (robust) speculative non-interference (RSNI, Section 3.1). Next, we introduce (robust) speculative safety (RSS, Section 3.2). These definitions can be applied to programs in the four languages L, T, L⁻, and T⁻. Therefore, we write RSNI(L) and RSS(L) to indicate which language L the definitions are referring to. Since these languages have the same syntax but different semantics, we also study the relationships between RSNI and RSS for weak and strong languages. We depict these results below (only for T and T⁻ since all security definitions trivially hold for the source non-speculative languages L and L⁻) and discuss them further down.



3.1 Robust Speculative Non-Interference

Speculative non-interference (SNI) is a class of security properties [27, 28] that is based on comparing the information leaked by instructions executed speculatively and non-speculatively. SNI requires that speculatively-executed instructions do not leak more information than what is leaked by executing the program without speculative execution, which is obtained by ignoring observations produced speculatively. Hence, SNI semantically characterize the

information leaks that are introduced by speculative execution, that is, those leaks that are exploited in Spectre-style attacks.

Property. Here, we instantiate robust speculative non-interference in our framework by following SNI's trace-based characterization [27, Proposition 1]. Thus we need to introduce two concepts:

- SNI is parametric in a policy denoting sensitive information. As mentioned in Section 2.1, we assume that only the private heap is sensitive. Hence, whole programs W and W' are *low-equivalent*, written $W' =_{\mathtt{L}} W$, if they differ only in their private heaps.
- SNI requires comparing the leakage resulting from non-speculative and speculative instructions. The *non-speculative projection* $t\restriction_{nse}$ [27] of a trace t extracts the observations associated with non-speculatively-executed instructions. We obtain $t\restriction_{nse}$ by removing from t all sub-strings enclosed between $\mathtt{if}(v)$ and $\mathtt{rlb}$ observations. We illustrate this using an example: $\cdot\restriction_{nse}$ applied to $\mathtt{t_{sp}}$ from Example 2.2 produces $\mathtt{t_{sp}}\restriction_{nse} = \mathtt{call\ get\ 8?} \cdot \mathtt{if(1)} \cdot \mathtt{ret!}$.

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

Definition 3.1 (Speculative Non-Interference (SNI)).

$$\begin{aligned} \vdash W : \mathrm{SNI} \stackrel{\text{def}}{=} \forall W'.\ & \text{if } W' =_{\mathtt{L}} W \\ & \text{and } \mathit{Beh}(\Omega_0(W))\restriction_{nse} = \mathit{Beh}(\Omega_0(W'))\restriction_{nse} \\ & \text{then } \mathit{Beh}(\Omega_0(W)) = \mathit{Beh}(\Omega_0(W')) \end{aligned}$$

A component P is robustly speculatively non-interferent if it is SNI no matter what valid attacker it is linked to (Definition 3.2), where an attacker A is valid ($\vdash A : \mathit{atk}$) if it does not define a private heap and does not contain instructions to read and write it.

Definition 3.2 (Robust Speculative Non-Interference (RSNI)).

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

Example 3.3 (Listing 1 is RSNI in $\mathtt{L}$ and not in $\mathtt{T}$). Consider the code of Listing 1. As expected, this code is RSNI in $\mathtt{L}$. Indeed, $\mathtt{L}$ does not support speculative execution and, therefore, for any trace $\mathtt{t_{ns}}$ produced by an $\mathtt{L}$-program $\mathtt{t_{ns}}\restriction_{nse} = \mathtt{t_{ns}}$.

The same code, however, is not RSNI in $\mathtt{T}$. Consider the code of Listing 1 (indicated as $\mathtt{P_1}$) and an attacker $\mathtt{A^8}$ that calls function get with 8. Since array A is in the private heap, the low-equivalent program required by Definition 3.1 is the same $\mathtt{A^8}$ linked with some $\mathtt{P_N}$, which is the same $\mathtt{P_1}$ with some array N with contents different from A in the heap such that $\mathtt{A[8]} \neq \mathtt{N[8]}$. Whole program $\mathtt{A^8}[\mathtt{P_1}]$ generates trace $\mathtt{t_{sp}}$ from Example 2.2 while $\mathtt{A^8}[\mathtt{P_N}]$ generates $\mathtt{t'_{sp}}$ below. We indicate the address of array N as $\mathtt{n_N}$ and the content of $\mathtt{N[i]}$ as $\mathtt{v^i_N}$. Low-equivalence yields that addresses are the same ($\mathtt{n_A} + 8 = \mathtt{n_N} + 8$) but contents are not ($\mathtt{v^8_A} \neq \mathtt{v^8_N}$), and thus B is accessed at different offsets ($\mathtt{n_B} + \mathtt{v^8_A} \neq \mathtt{n_B} + \mathtt{v^8_N}$).

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

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

Security Guarantees. Since RSNI is defined in terms of traces, its security guarantees depend on which of the four languages $\mathtt{L}$, $\mathtt{T}$, $\mathtt{L^-}$, and $\mathtt{T^-}$ we consider. As expected, for the source languages $\mathtt{L}$ and $\mathtt{L^-}$, RSNI is trivially satisfied; there is no speculative execution in $\mathtt{L}$ and $\mathtt{L^-}$ and all traces are identical to their non-speculative projections.

Theorem 3.4 (All $\mathtt{L}$ and $\mathtt{L^-}$ programs are RSNI).

$$\forall \mathtt{P}.\ \vdash \mathtt{P} : \mathit{RSNI}(\mathtt{L}) \textit{ and } \vdash \mathtt{P} : \mathit{RSNI}(\mathtt{L^-})$$

For the target languages $\mathtt{T}$ and $\mathtt{T^-}$, which support speculative execution, RSNI provides different security guarantees.

RSNI($\mathtt{T}$) corresponds to speculative non-interference [27, 28], which ensures the absence of *all* speculative leaks. In our setting, the only allowed leaks are those depending either on information from the public heap or information from the private heap that is disclosed through actions produced non-speculatively, e.g., as an address of a non-speculative memory access. Any other speculative leak of information from the private heap is disallowed by RSNI($\mathtt{T}$).

RSNI($\mathtt{T^-}$), in contrast, corresponds to weak speculative non-interference [28], which allows speculative leaks of information that has been retrieved non-speculatively. Indeed, in $\mathtt{T^-}$ non-speculative reads from the private heap produce actions $\mathtt{read}(\mathtt{n} \mapsto \mathtt{v})$ that additionally disclose the value v read from the heap as part of the non-speculative projection. As a result, data retrieved non-speculatively from the private heap can influence speculative actions, which are not part of the non-speculative projection of the trace, without violating RSNI($\mathtt{T^-}$). That is, RSNI($\mathtt{T^-}$) ensures the absence only of leaks of speculatively-accessed data.

Since RSNI($\mathtt{T}$) ensures the absence of *all* speculative leaks while RSNI($\mathtt{T^-}$) only ensures the absence of *some* of them, any RSNI($\mathtt{T}$) program is also RSNI($\mathtt{T^-}$).

Theorem 3.5 (RSNI($\mathtt{T}$) Implies RSNI($\mathtt{T^-}$)).

$$\forall \mathtt{P}.\ \textit{if } \vdash \mathtt{P} : \mathit{RSNI}(\mathtt{T}) \textit{ then } \vdash \mathtt{P} : \mathit{RSNI}(\mathtt{T^-})$$

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

3.2 Robust Speculative Safety

We now introduce *speculative safety* (SS), a safety property that soundly over-approximates SNI. To enable reasoning about security using single traces (rather than pairs of traces as in SNI), we extend our languages with a taint-tracking mechanism that (1) taints values as "safe" (denoted by S) whenever they can be leaked speculatively without violating SNI (e.g., the public heap is "safe") or "unsafe" (denoted by U) otherwise, and (2) propagates taints to labels across computations. Speculatively safe programs produce traces containing only safe labels.

Taint tracking Taint-tracking is at the foundation of our speculative safety definition and it enables reasoning about security on single traces. For this, we extend the semantics of our languages $\mathtt{L}$, $\mathtt{L^-}$, $\mathtt{T}$,

and T⁻ with a taint tracking mechanism. We consider two taint-tracking mechanisms, a strong and a weak one, that lead to different security guarantees, as we show later. Each mechanism is adopted in the related pair of languages: strong (resp. weak) languages use the strong (resp. weak) taint-tracking. Our taint-tracking is rather standard, so we provide an informal overview of its key features below using the rules for reading from the private heap as an example; full details are Appendix B. These rules simply extend Rule E-read-prv with taint, which is highlighted in gray.

$$\frac{B \triangleright e \downarrow n : \sigma' \qquad H(-|n|) = v : \sigma'' \qquad \sigma = \sigma'' \sqcup \sigma'}{\sigma_{pc} ; C, H, \overline{B} \cdot B \triangleright let\ x = rd_{pr}\ e\ in\ s \xrightarrow{\mathtt{read}(-|n|)} C, H, \overline{B} \cdot B \cup x \mapsto v : U \triangleright s} \text{(T-read-prv)}$$

$$\frac{B \triangleright e \downarrow n : \sigma' \qquad H(-|n|) = v : \sigma'' \qquad \sigma = \sigma'' \sqcup \sigma'}{\sigma_{pc} ; C, H, \overline{B} \cdot B \triangleright let\ x = rd_{pr}\ e\ in\ s \xrightarrow{\mathtt{read}(-|n| \mapsto v)} C, H, \overline{B} \cdot B \cup x \mapsto v : \sigma' \sqcap \sigma_{pc} \triangleright s} \text{(T-read-prv-weak)}$$

- All values v are tainted with a taint $\sigma \in \{S, U\}$. Heaps H and variable bindings B are extended to record the taint of values. Taints form the usual *integrity* lattice $S \leq U$ (which is the dual of the lattice used for non-interference) and are combined using the least-upper-bound ($\sqcup$) and greatest-lower-bound ($\sqcap$) operators. For simplicity, we report here the key cases: $S \sqcup U = U$ and $S \sqcap U = S$.
- The public part of the initial heap is tainted as safe, and its private part is tainted as unsafe.
- The taint-tracking mechanism also tracks the taint σ_{pc} associated with the program counter. The program counter taint is S whenever we are not speculating and it is raised to U whenever we are executing instructions speculatively. The latter can happen only in the T and T⁻ languages, where it is represented by the speculative state containing more than one speculation instance. In the source languages, instead, σ_{pc} is always S.
- Taint is propagated in the standard way across computations. For example, expressions combine taints using the least-upper-bound $\sqcup$, i.e., expressions involving unsafe values are tainted U.

The strong and weak taint-tracking mechanisms differ, however, in how they handle memory reads from the private heap. When reading from the private heap, the strong mechanism used in L and T taints the variable where the data is stored as unsafe (U) (Rule T-read-prv). In contrast, the weak mechanism of L⁻ and T⁻, taints the target value with the greatest-lower-bound of the taints of the memory address and of the program counter (Rule T-read-prv-weak). This ensures that information retrieved non-speculatively from the private heap (i.e., the program counter taint is S) is tainted S.

- The taint tracking does not keep track of implicit flows. Since the program counter is part of the actions, any sensitive implicit flow would appear in the trace due to the corresponding $\mathtt{if}(v)$ action.
- The taint of labels is the greatest-lower-bound of the taint of the expressions generating the label and the program counter taint (Rule T-read-prv and Rule T-read-prv-weak). This ensures that non-speculative labels are tainted as safe (S), while speculative labels are tainted as unsafe (U) if they depend on unsafe data and safe otherwise.

With a slight abuse of notation, in the following we refer to the languages L, L⁻, T, and T⁻ extended with the corresponding taint-tracking mechanisms outlined above whenever we talk about speculative safety. That is, for speculative safety, programs in L, L⁻, T, and T⁻ produce traces $\overline{\lambda^{\sigma}}$ of tainted labels λ^{σ}, where taints σ are computed as described above.

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

Definition 3.6 (Speculative Safety (SS)).

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

A component P is RSS if it upholds SS when linked against arbitrary valid attackers (Definition 3.7).

Definition 3.7 (Robust Speculative Safety (RSS)).

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

Example 3.8 (Listing 1 is RSS in L and not in T). The code of Listing 1 is RSS in L because σ_{pc} is always S and, therefore, all actions are tainted as S. The code, however, is neither RSS in T nor in T⁻. For this, consider the trace from Example 2.2. The taint-tracking mechanism taints the actions as follows:

$$\mathtt{t_{sp}} = \mathtt{call\ get\ 8?}^{\mathtt{S}} \cdot \mathtt{if(1)}^{\mathtt{S}} \cdot \mathtt{read(A[8])}^{\mathtt{S}} \cdot \mathtt{read(B[A[8]])}^{\mathtt{U}} \cdot \mathtt{rlb}^{\mathtt{S}} \cdot \mathtt{ret!}^{\mathtt{S}}$$

The trace contains an unsafe action corresponding to the second memory access. This happens because the action has been generated speculatively (that is, σ_{pc} is U) and it depends on data retrieved from the private heap (which T's taint-tracking taints as U). ⊡

Security Guarantees. Similarly to SNI, the security guarantees of SS depend on the underlying language. As expected, RSS trivially holds for L and L⁻ since they only produce labels tainted S.

Theorem 3.9 (All L and L⁻ programs are RSS).

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

In contrast, RSS' guarantees are different for T and T⁻, which are equipped with distinct taint tracking mechanisms.

RSS(T) is a strict over-approximation of RSNI(T) (and, thus, of speculative non-interference in terms of [27, 28]) and its preservation through compilation is easier to prove than RSNI(T)-preservation.

Theorem 3.10 (RSS(T) over-approximates RSNI(T)).

1) $\forall \mathtt{P}.$ *if* $\vdash \mathtt{P} : RSS(\mathtt{T})$ *then* $\vdash \mathtt{P} : RSNI(\mathtt{T})$

2) $\exists \mathtt{P}.\ \vdash \mathtt{P} : RSNI(\mathtt{T})$ *and* $\nvdash \mathtt{P} : RSS(\mathtt{T})$

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

To understand point 2, consider get_nc from Listing 2, which always accesses B[A[y]]. This code is RSNI(T) because states that

can be distinguished by the traces can also be distinguished by their non-speculative projections, i.e., speculatively-executed instructions do not leak additional information. However, it is not RSS(T) because speculative memory accesses will produce U actions.

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

Listing 2: Code that is RSNI but not RSS.

RSS($\mathtt{T}^-$), in contrast, is a strict over-approximation of RSNI($\mathtt{T}^-$) (and, therefore, of weak speculative non-interference in terms of [28]).

Theorem 3.11 (RSS($\mathtt{T}^-$) over-approximates RSNI($\mathtt{T}^-$)).

$$1)\ \forall \mathtt{P}.\ \textit{if} \vdash \mathtt{P} : \textit{RSS}(\mathtt{T}^-)\ \textit{then} \vdash \mathtt{P} : \textit{RSNI}(\mathtt{T}^-)$$

$$2)\ \exists \mathtt{P}.\ \vdash \mathtt{P} : \textit{RSNI}(\mathtt{T}^-)\ \textit{and} \nvdash \mathtt{P} : \textit{RSS}(\mathtt{T}^-)$$

Finally, it is easy to see that any RSS(T) program is also RSS($\mathtt{T}^-$) since all actions tainted S by the taint-tracking of T are tainted S also by the taint-tracking of $\mathtt{T}^-$.

Theorem 3.12 (RSS(T) Implies RSS($\mathtt{T}^-$)).

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

4 COMPILER CRITERIA FOR SECURE SPECULATION

We now introduce our secure compilation criteria: *robust speculative safety preservation* (*RSSP*, Section 4.1), which preserves RSS, and *robust speculative non-interference preservation* (*RSNIP*, Section 4.2), which preserves RSNI. We conclude by discussing how compilers can be proven secure or insecure using these criteria (Section 4.3).

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

4.1 Robust Speculative Safety Preservation

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

Definition 4.1 ($\textit{RSSP}^+$).

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

Definition 4.1 is a "property-ful" criterion since it explicitly refers to the preserved property [3, 4]. Proving a "property-ful" criterion, however, can be fairly complex. Fortunately, it is generally possible to turn a "property-ful" definition into an *equivalent* "property-free" one [3, 4, 51], which come in so-called *backtranslation* form with 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 4.2) states that a compiler is *RSSC* if for any target-level attacker 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 (S) actions (Rules Trace-Relation-Safe and Trace-Relation-Safe-Heap). All other actions must be the same (i.e., ≡, Rules Trace-Relation-Same and Trace-Relation-Same-Heap).

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

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

We are now ready to formalise *RSSC*, which intuitively states that compiled programs produce the same traces as their source counterparts with possibly additional safe actions. Crucially, *RSSC* is equivalent to *RSSP* (Theorem 4.3), 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 4.2 ($\textit{RSSC}^+$).

$$\vdash [\![\cdot]\!] : \textit{RSSC}^+ \stackrel{\text{def}}{=} \forall \mathtt{P} \in \mathtt{L}, \mathtt{A}, \overline{\lambda^\sigma}.\ \text{if}\ \mathrm{Beh}\left(\mathtt{A}\left[[\![\mathtt{P}]\!]\right]\right) = \overline{\lambda^\sigma}$$
$$\text{then}\ \exists \mathtt{A}, \overline{\lambda^\sigma}.\ \mathrm{Beh}(\mathtt{A}\,[\mathtt{P}]) = \overline{\lambda^\sigma}\ \text{and}\ \overline{\lambda^\sigma} \approx \overline{\lambda^\sigma}$$

Theorem 4.3 (*RSSP* and *RSSC* are equivalent).

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

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

Definition 4.2 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-based [13, 21, 45] or trace-based [2, 49, 51]. The distinction tells us what quantified element one can use to build the source attacker A, either the target attacker A or the trace $\overline{\lambda^\sigma}$ respectively. In our proofs, we will use an attacker-based backtranslation.

4.2 Robust Speculative Non-Interference Preservation

Here, we only present a property-ful criterion for the preservation of RSNI (Definition 4.4). The reason is that we only directly prove that compilers do *not* attain *RSNIP*. This kind of proof is simple already (Corollary 4.5), and we do not need a property-free criterion.

Definition 4.4 ($\textit{RSNIP}^+$).

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

Corollary 4.5 ($\nvdash [\![\cdot]\!] : \textit{RSNIP}^+$).

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

Let us now unfold the corollary in order to understand what must be proven to show that a compiler is not $\textit{RSNIP}^+$. The crux is the second clause of the corollary, which gets unfolded to the following. Recall that low-equivalent programs simply differ in their private heap, so $\mathtt{A}\left[[\![\mathtt{P}']\!]\right]$ *is the same as* $\mathtt{A}\left[[\![\mathtt{P}]\!]\right]$ *but with a different private heap.*

$$\nvdash [\![\mathtt{P}]\!] : \textit{RSNI}(\mathtt{T}) = \exists \mathtt{A}.\ \vdash \mathtt{A} : \textit{atk and given}\ \mathtt{A}\left[[\![\mathtt{P}']\!]\right] =_L \mathtt{A}\left[[\![\mathtt{P}]\!]\right]$$
$$\textit{we have}\ \mathrm{Beh}\left(\Omega_0\left(\mathtt{A}\left[[\![\mathtt{P}]\!]\right]\right)\right)\restriction_{nse} = \mathrm{Beh}\left(\Omega_0\left(\mathtt{A}\left[[\![\mathtt{P}']\!]\right]\right)\right)\restriction_{nse}$$

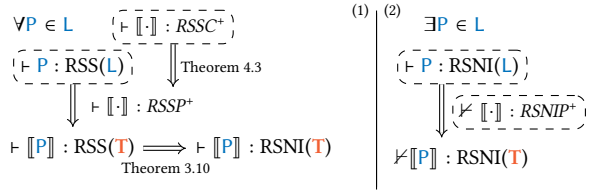
$$\textit{and}\ \ \mathbf{Beh}\big(\Omega_0\ \big(\mathtt{A}\ \big[\![\mathtt{P}]\!]\big)\big) \neq \mathbf{Beh}\big(\Omega_0\ \big(\mathtt{A}\ \big[\![\mathtt{P}']\!]\big)\big)$$

That is, we need to find a program P *and an attacker* A *that violate RSNI. 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 for this in the appendix.* □

We remark that the insecurity part of our methodology is used to show its completeness wrt vulnerability to Spectre v1 attacks. Unfortunately, one still has to manually come up with the insecure counterexample and verify that it is not RSNI.

4.3 A Methodology for Provably-(In)Secure Countermeasures

To prevent speculative leaks, secure compilers should produce target programs that satisfy RSNI (cf. Section 3.1) whereas insecure compilers will produce some programs that fail to achieve RSNI. In this section, we show how to combine the results from the previous sections to derive exactly these facts about compilers; we depict this with the two chains of implications below. The first one (1) lists the assumptions (black dashed lines) and logical steps (theorem-annotated implications) to conclude compiler security while the second one (2) lists assumptions and logical steps for compiler insecurity. For simplicity, the diagram focuses on security definitions and compiler criteria for L and T. There are similar chains of implications for $\mathtt{L}^-$ and $\mathtt{T}^-$ that use Theorem 3.11 instead of Theorem 3.10.



To show security (1), we need to prove that any compiled component is RSNI in the target language. Rather than directly reasoning about RSNI, we rely on RSS, which over-approximates RSNI (cf. Theorem 3.10). This significantly simplifies our security proofs since it allows us to reason about single traces rather than pairs of traces. Thus, it suffices to show that any compiled component is RSS in the target. This can be obtained by (i) an $RSSP^+$ compiler so long as (ii) any P is RSS in the source. By Theorem 4.3, for point (i) it is sufficient to show that the compiler is $RSSC^+$. Point (ii) holds for any P (Theorem 3.9). This direction highlights how RSS really is a working security definition that simplifies proving the more precise, semantic security definition which is RSNI.

To show insecurity (2), we need to prove that there exists a compiled component that is *not* RSNI in the target language. For this, we show (A) that the compiler is not $RSNIP^+$ given that (B) the source component P was RSNI in the source. To show (A), we follow Corollary 4.5, whereas point (B) holds for any P (Theorem 3.9).

Our security criteria, instantiated for the strong (L-T) and weak ($\mathtt{L}^-$-$\mathtt{T}^-$) languages, provide a way of characterizing the security guarantees of any countermeasure $[\![\cdot]\!]$, which is what 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 Spectre attacks. Finally, showing that $[\![\cdot]\!]$ is not $RSNIP^-$ implies that compiled code leaks speculatively accessed data, like in Spectre attacks.

Preservation or Enforcement? *RSNIP* and *RSSP* focus on *preserving* the related security property. Since their premise is always satisfied, we could also state them in terms of *enforcing* RSNI and RSS over compiled programs. We choose against this to be able to reuse established compiler theory [39], and since it is unclear how to prove Theorem 4.3 with enforcement statements.

5 COUNTERMEASURES ANALYSIS

In this section, we characterise the security guarantees of the main Spectre v1 countermeasures implemented by compiler vendors: insertion of speculation barriers (`lfence`) and speculative load hardening (`slh`). For this, we develop formal models that capture the key aspects of these countermeasures as implemented by the Microsoft Visual C++ compiler [47] (MSVC, Section 5.1), the Intel C++ compiler [33] (ICC, Section 5.2), and the Clang compiler (Section 5.3), and we analyze their guarantees using our secure compilation criteria. We continue the section with an overview of our proofs (Section 5.4). We conclude by discussing our analysis' results (Section 5.5). For space constraints, compiled snippets, their formalisation, and full security proofs can be found in [52].

5.1 MSVC is Insecure

Inserting speculation barriers—the lfence x86 instruction—after branch instructions is a simple countermeasure against Spectre v1 [31, 33, 47]. This instruction stops speculative execution at the price of significant performance overhead.

MSVC implements a countermeasure that tries to minimize the number of lfences by selectively determining which branches to patch [47]. However, MSVC fails in inserting some necessary lfence s, thereby producing insecure code that is not RSNI($\mathtt{T}^-$) and that is vulnerable to Spectre-style attacks.

To show this, we follow Corollary 4.5 and provide a program that is RSNI($\mathtt{L}^-$) and its compilation is not RSNI($\mathtt{T}^-$). The program we consider, which is RSNI($\mathtt{L}^-$) (Theorem 3.9), is given in Listing 3.

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

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

As shown in [27, 36], MSVC fails in injecting an lfence after the first branch instruction. As a result, the compiled target program is identical to Listing 3, and it speculatively leaks whether A[y] is 0 through the branch statement in line 3, i.e., it violates RSNI($\mathtt{T}^-$). We refer to [27, 36] for additional examples of MSVC's insecurity.

5.2 ICC is Secure

The Intel C++ compiler also implements a countermeasure that inserts lfences after each branch instruction [33].

We model this countermeasure with $[\![\cdot]\!]^f$, a homomorphic compiler that takes a component in L and translates all of its subparts

to T. Its key feature is inserting an lfence statement at the beginning of every then and else branch of compiled code. All other statements are left unmodified by the compiler.

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

It should come at no surprise that $[\![\cdot]\!]^f$ is *RSSC*$^+$ (Theorem 5.1). In T, the only source of speculation are branches (Rule E-T-speculate-if) but any branch, whether it evaluates to true or false, will execute an lfence (Rule E-T-speculate-lfence), triggering a rollback (Rule E-T-speculate-rollback). Since compiled code performs no action during speculation, it can only perform actions when the program counter is tainted as S, which makes all actions S. These actions are easy to relate to their source-level counterparts since they are generated according to the non-speculative semantics.

Theorem 5.1 (ICC is secure for L-T). $\vdash [\![\cdot]\!]^f : \mathit{RSSC}^+$

5.3 Speculative Load Hardening

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

- 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 into the most-significant bits of the stack pointer register, which are always unused. Note that when 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* = 0xF..F if code is mis-speculating and *mask* = 0x0..0 otherwise. Before returning from a function, SLH-compiled code pushes the most-significant bit of the current mask to the stack; thereby preserving the predicate bit.
- All inputs to control-flow and store instructions are hardened by masking their values with *mask* (i.e., by or-ing their value with *mask*). That is, whenever code is mis-speculating (i.e., *mask* = 0xF..F) the inputs to these statements are "F-ed" to 0xF..F, otherwise they are left unchanged. This prevents speculative leaks through control-flow and store statements.
- The outputs of memory loads instructions are hardened by or-ing their value with *mask*. So, when code is mis-speculating, the result of load instructions is "F-ed" to 0xF..F. This prevents leaks of speculatively-accessed memory locations. Inputs to load instructions, however, are *not* masked.

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

5.3.1 SLH is not RSNIP$^+$. We show that SLH is not *RSNIP*$^+$, i.e., it does not preserve (strong) speculative non-interference and thus it allows speculative leaks of data retrieved non-speculatively.

Following Corollary 4.5, we do this by providing a program that is RSNI(L) and that is compiled to a program that is not RSNI(T). The program in Listing 4 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.

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

5.3.2 SLH is RSSC$^-$. We now show that SLH is *RSSC*$^-$, that is, it prevents leaks of speculatively-accessed data.

We formalise SLH using the $[\![\cdot]\!]^s$ compiler, whose most interesting cases are given in the top of Figure 1. The compiler takes components in L$^-$ and outputs compiled code in 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 which attackers cannot access. So location -1 is initialised to 1 (false) and updated to 0 whenever we are speculating. Compiled code must update the predicate bit right after the then and else branches (statements $-1 :=_{pr} \cdots$). Since location -1 is reserved for the predicate bit, all private memory accesses and the private heap are shifted by 1.

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

As stated in Theorem 5.2, programs compiled with SLH are RSS(T$^-$) and, therefore, RSNI(T$^-$) (Theorem 3.10). Hence, they are free of leaks of speculatively-accessed data, which is sufficient to stop Spectre-style leaks like those in Listing 1.

Theorem 5.2 (SLH is secure for L$^-$-T$^-$). $\vdash [\![\cdot]\!]^s : \mathit{RSSC}^-$

$[\![\cdot]\!]^s$ is *RSSC*$^-$ for two reasons. First, location -1 (and thus variable pr where its contents are loaded) always correctly tracks whether speculation is ongoing or not. This holds because location -1 and 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 x_f). Second, whenever speculation is happening, the result of load operations is set to a constant 0 whose taint is S. So, computations happening during

$$\left[\!\left[\mathsf{H};\overline{\mathsf{F}};\overline{\mathsf{I}}\right]\!\right]^{s} = [\![\mathsf{H}]\!]^{s} \cup (-1 \mapsto 1 : \mathsf{S});\ \left[\!\left[\overline{\mathsf{F}}\right]\!\right]^{s};\ \left[\!\left[\overline{\mathsf{I}}\right]\!\right]^{s} \qquad [\![\mathsf{H}, -\mathsf{n} \mapsto \mathsf{v} : \mathsf{U}]\!]^{s} = [\![\mathsf{H}]\!]^{s}, -[\![\mathsf{n}]\!]^{s} - 1 \mapsto [\![\mathsf{v}]\!]^{s} : \mathsf{U}$$

$$[\![\mathsf{if}\ \mathsf{e}\ \mathsf{then}\ \mathsf{s}\ \mathsf{else}\ \mathsf{s}']\!]^{s} = \mathsf{let}\ \mathsf{x_f}{=}[\![\mathsf{e}]\!]^{s}\ \mathsf{in}\ \mathsf{let}\ \mathsf{pr} = \mathsf{rd_{pr}}\ {-}1\ \mathsf{in}\ \mathsf{let}\ \mathsf{x_f} = 0\ (\mathsf{if}\ \mathsf{pr})\ \mathsf{in}\ \mathsf{ifz}\ \mathsf{x_f}\ \mathsf{then}\ {-}1 :=_{\mathsf{pr}} \mathsf{pr} \vee \neg \mathsf{x_f}; [\![\mathsf{s}]\!]^{s}\ \mathsf{else}\ {-}1 :=_{\mathsf{pr}} \mathsf{pr} \vee \mathsf{x_f}; [\![\mathsf{s}']\!]^{s}$$

$$\left[\!\left[\mathsf{e} :=_{\mathsf{pr}} \mathsf{e}'\right]\!\right]^{s} = \mathsf{let}\ \mathsf{x_f}{=}[\![\mathsf{e}]\!]^{s}{+}1\ \mathsf{in}\ \mathsf{let}\ \mathsf{x_f'}{=}[\![\mathsf{e}']\!]^{s}\ \mathsf{in}\ \mathsf{let}\ \mathsf{pr} = \mathsf{rd_{pr}}\ {-}1\ \mathsf{in}\ \mathsf{let}\ \mathsf{x_f} = 0\ (\mathsf{if}\ \mathsf{pr})\ \mathsf{in}\ \mathsf{let}\ \mathsf{x_f'} = 0\ (\mathsf{if}\ \mathsf{pr})\ \mathsf{in}\ \mathsf{x_f} :=_{\mathsf{pr}} \mathsf{x_f'}$$

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

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

Figure 1: Key bits of the SLH compiler $[\![\cdot]\!]^{s}$ (above). The SSLH compiler $[\![\cdot]\!]^{ss}$ (below) differs in the compilation of memory reads.

speculation either depend on data loaded non-speculatively, which are tainted as S by the taint-tracking of T^{-}, or on masked values, which are also tainted S. Speculative actions are tainted with glb ($\sqcap$) of data taint (S) and pc taint (U). Since $S \sqcap U = S$ (see Section 3.2), speculative actions are tainted S, satisfying RSS(T^{-}).

5.3.3 Making SLH More Secure. We now show how to modify SLH to prevent *all* speculative leaks. We do so by introducing *strong SLH* (SSLH for short) that differs from standard SLH in that it masks the input (rather than the output) of memory read operations (as such, we expect an implementation of SSLH to have a small overhead caused by the newly introduced data dependencies that might delay some masked loads). 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 (Figure 1). The compiler masks the input of memory loads by evaluating the sub-expressions and storing them in auxiliary variables (called $\mathsf{x_f}$), retrieving the predicate bit and storing it in variable pr, conditionally masking the value of $\mathsf{x_f}$, and, finally, performing the memory access using $\mathsf{x_f}$ as address.

As stated in Theorem 5.3, programs compiled using SSLH are RSS(T) and, thanks to Theorem 3.10, RSNI(T). Therefore, they are free of all speculative leaks.

Theorem 5.3 (SSLH is secure for L-T). $\vdash [\![\cdot]\!]^{ss} : \mathit{RSSC}^{+}$

$[\![\cdot]\!]^{ss}$ satisfies RSSC^{+} for two reasons. First, the compiler correctly tracks whether speculation is ongoing (cf. §5.3.2). Second, whenever speculation is happening, the result of any possibly-leaking expression is set to a constant 0 whose taint is S. That is, labels during speculation are tainted as S, and RSS(T) holds.

5.3.4 Non-interprocedural SLH is insecure. We conclude by showing that the non-interprocedural variant of SLH, where the predicate bit is set to 0 at the beginning of each function, is insecure and does not prevent all speculative leaks. Consider the program in Listing 5 that splits the memory accesses of A and B of the classical Spectre v1 snippet across functions get and get_2.

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

Listing 5: Inter-procedural variant of Spectre v1 snippet [42]

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. Hence, the memory access corresponding to B[x] is not masked leading to the leak of x (which contains A[y]), so the target program violates RSNI(T^{-}).

In Appendix C, we show how to secure this SLH variant.

5.4 How to Prove *RSSC*

We now illustrate the backtranslation proof technique used to prove SLH-related countermeasures secure. Our backtranslation is a simple adaptation of the general backtranslation proof technique [51]. To prove that a compiler is *RSSC*, we *backtranslate* a target attacker (A) to create a source attacker ($\mathsf{A} = \langle\!\langle \mathsf{A} \rangle\!\rangle$) so that they produce traces related by the relation of Section 4. Our backtranslation function ($\langle\!\langle \cdot \rangle\!\rangle$), which is the same for all proofs, homomorphically translates target heaps, functions, statements etc. into source ones.

We depict our proof approach in Figure 2. 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.

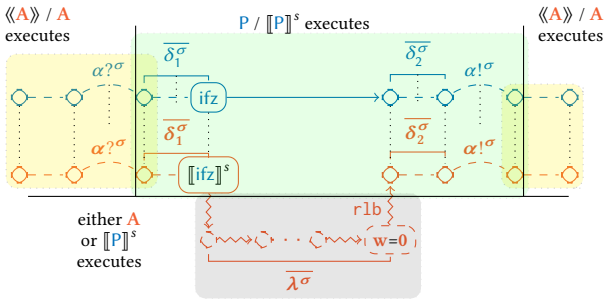


Figure 2: Diagram depicting the proof that $[\![\cdot]\!]^{s}$ is RSSC^{-}.

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 $\langle\!\langle \mathsf{A} \rangle\!\rangle$ does the same step and ends up in related states (yellow areas). To reason about component reductions, we adapt a

reasoning from compiler correctness results [12, 39]. That is, if s steps and emits a trace, then $[\![s]\!]^s$ 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 $[\![\text{ifz}]\!]^s$ 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 speculative reductions immediately trigger an lfence, which rolls the speculation back (the speculation window w is 0) reinstating the cross-language state relation right away.

5.5 Summary

Our security analysis is the first rigorous characterization of the security guarantees provided by Spectre v1 compiler countermeasures, and it complements existing results that focus on selected code snippets [27, 36]. The table below depicts the results of our analysis in terms of the security properties satisfied by compiled programs. There, ● denotes that *all* compiled programs satisfy the criterion and ○ denotes that some compiled programs violates it.

	RSNI (T)	RSNI (T^-)
`lfence`(MSVC), `SLH-no-interp`	○	○
`lfence`(ICC)/$[\![\cdot]\!]^f$, `SSLH`/$[\![\cdot]\!]^{ss}$	●	●
`SLH`(Clang)/$[\![\cdot]\!]^s$	○	●

The main findings of our security analysis are summarized below:

- The `lfence` countermeasure implemented in MSVC, denoted `lfence`(MSVC), is insecure. It violates *RSNIP*$^-$ and produces programs that are not speculatively non-interference, i.e., that violate both RSNI (T) and RSNI (T^-). Hence, compiled programs still contain speculative leaks and might be vulnerable to Spectre attacks.
- The `lfence` countermeasure implemented in ICC, denoted `lfence`(ICC) and modelled by $[\![\cdot]\!]^f$, is secure. The model satisfies *RSSP*$^+$ (Theorem 5.1) and, as a result, produces only compiled programs that satisfy speculative non-interference, that is, RSNI(T). Hence, compiled programs are free of speculative leaks.
- The speculative load hardening countermeasure implemented in Clang, denoted `SLH`(Clang) and modelled by $[\![\cdot]\!]^{ss}$ is secure for L^--T^-. The model satisfies *RSSP*$^-$ (Theorem 5.2) and, as a result, produces only compiled programs that satisfy weak speculative non-interference, that is, RSNI(T^-). Hence, compiled programs are free of speculatively leaks that involve speculatively-accessed data. While this is sufficient for preventing Spectre-style attacks, compiled programs may still speculatively leak data retrieved non-speculatively, which might result in breaking properties like constant-time (see [28]).
- The strong variant of `SLH`, denoted `SSLH` and modelled by $[\![\cdot]\!]^{ss}$ is secure for L-T. The model satisfies *RSSP*$^+$ (Theorem 5.3) and produces compiled programs that satisfy speculative non-interference, that is, RSNI(T). Thus, compiled programs have *no* speculative leaks.
- Non-interprocedural `SLH`, denoted `SLH-no-interp`, is insecure. It violates *RSNIP*$^-$ and produces programs that violate both RSNI (T) and RSNI (T^-). Hence, compiled programs might still be vulnerable to Spectre attacks.

Additional security guarantees. In addition to *RSNIP*, the secure compilers $[\![\cdot]\!]^f$, $[\![\cdot]\!]^s$, and $[\![\cdot]\!]^{ss}$ also preserve the non-speculative behavior of source programs. That is, if two source programs W and W′ produce the same traces, then their compiled counterparts produce traces with the same non-speculative projection. This directly follows from the compilers only modifying the speculative behavior of programs, either through lfences or conditional masking.

By combining *RSNIP* with the preservation of non-speculative behaviors, we can derive an additional security guarantee for our compilers: preservation of non-interference. For simplicity, we only focus on whole programs and we use $[\![\cdot]\!]^f$ as an example; the same argument applies to $[\![\cdot]\!]^s$ and $[\![\cdot]\!]^{ss}$. We say that a program W *is non-interferent* (NI) if all programs W' that differ from W only in the private heap (i.e., they are low-equivalent) produce the same traces as W. Given a source program $W \in L$ that is NI, we obtain that $[\![W]\!]^f$ is NI if we restrict ourselves to the non-speculative projection of traces since $[\![W]\!]^f$ preserves the non-speculative behavior of W. Since $[\![W]\!]^f$ is RSNI(T), the full traces do not leak more than their non-speculative projections and thus $[\![W]\!]^f$ is also non-interferent.

The security guarantees of NI depend on the underlying language. For strong languages L-T, NI ensures that programs are *constant-time* with respect to the private heap (in L, we have classical constant-time [8, 44] while in T we have speculative constant-time [17]). Indeed, information from the private heap cannot influence the traces where `read`(n), `write`(n), and `if`(v) actions correspond to the standard constant-time observer. For the weak languages L^--T^-, NI ensures a form of *sandboxing* where programs (1) cannot access information form the private heap non-speculatively (because reading values from the private heap violates NI through actions `read`($n \mapsto v$)), and (2) cannot speculatively leak information about the private heap. We leave exploring these additional security results as future work.

6 SCOPE AND LIMITATIONS OF THE MODEL

Lifting our analysis to real CPUs is only valid to the extent that our attacker model and speculative semantics capture the target system.

Our attacker observes the location of memory accesses and the outcome of control-flow statements. This attacker model offers a good trade-off between precision and simplicity [8, 44], and it has proven to capture interesting microarchitectural leaks, like those resulting from caches and port contention. Other classes of microarchitectural leaks, like those resulting from internal buffers [63] or hardware prefetchers [26], might not be captured by our model.

We also assume that attackers cannot access the private heap since there can be no protection against same-process attackers. This can be achieved by running attacker and component in separate processes and leveraging OS-level memory protection.

Finally, the semantics of our target languages are adequate to reason only about Spectre v1-style attacks. These semantics ignore the effects of out-of-order execution. As a result, they cannot be used to reason about countermeasures that rely only on data dependencies to restrict speculatively executed instructions [46]. For a similar reason, our analysis of SLH might be too pessimistic in that the data dependencies resulting from the injected masking operations might effectively limit the scope of speculative execution. Our semantics also ignore other sources of speculation (e.g., indirect jumps) that are exploited by other Spectre variants, as we discuss next.

Beyond Spectre v1. Spectre v1 (also called Spectre-PHT) is just one of the (many) Spectre variants, we recount other variants below and discuss how to extend this work to reason about them.

- Spectre BTB [37] exploits speculation over indirect jump instructions. The *retpoline* compiler countermeasure [32] replaces indirect jumps with a return-based trampoline that leads to code that perform busy waiting. 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 [32] that renders *retpoline* a valid countermeasure also against Spectre-RSB [15].
- Spectre-STL [30] exploits speculation over data dependencies between in-flight store and load operations. To mitigate it, ARM introduced a dedicated `SSBB` speculation barrier to prevent store bypasses that could be injected by compilers.

To reason about these Spectre variants, we need to extend the speculative semantics of T to capture the new kinds of speculative execution; this is analogous to other semantics [9, 17, 43, 64]. Crucially, the traces must capture events that are meaningful for the related variant (e.g., reads and writes for Spectre-STL, returns for Spectre-RSB). These actions are already present in traces of T, so the new semantics can reuse the trace model presented here. This, in turn, ensures that we can use the secure compilation criteria and trace relation from Section 4 to reason about whether compiler-inserted countermeasures for these variants are secure or not. Any proof that countermeasures for these variants are *RSSP* should follow the overview in Section 5.4. Specifically, proofs for *retpoline* would follow the approach of Figure 2 since speculative execution gets diverted to code that does not produce observations (we provide an in-depth discussion on *retpoline* in Appendix D). In contrast, reasoning about `SSBB` would be similar to reasoning about $[\![\cdot]\!]^f$ since `SSBB`s instructions act as speculation barriers. We leave investigating these topics in detail for future work.

7 RELATED WORK

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

Speculative semantics These semantics model the effects of speculatively-executed instructions. Several semantics [9, 17, 28, 43, 64] explicitly model microarchitectural details like multiple pipeline stages, reorder buffers, caches, and predictors. These semantics are significantly more complex than ours (which is inspired by [27]), and they would lead to much harder proofs.

Security definition against Spectre attacks SNI [27] has been used as security definition against speculative leaks also by [9, 28]. 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 countermeasures that *only* modify a program's speculative behaviour. More generally, Guarnieri *et al.* [28] presents a secure programming framework that subsumes both SNI and SCT.

Compiler countermeasures for Spectre v1 Apart from the insertion of speculation barriers [5, 31] and SLH [16, 46], few countermeasures for Spectre v1 exist. Replacing branch instructions with branchless computations (using `cmov` and bit masking) is effective [53] but not generally applicable. oo7 [65] is a tool that automatically patches speculative leaks by injecting speculation barriers. However, oo7 misses some speculative leaks [27] and violates *RSNIP*$^-$.

Blade [64] is a compiler 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). Similarly to our framework, Blade consider a source language *without* speculation and a target language *with* speculation and it preserves constant-time from source to target [64, Corollary 1]. This is different from the compilers we study, which block (classes of) speculative leaks regardless of whether the source program is constant-time. Blade's design relies on fine-grained barriers whose scope are single instructions. Since these barriers are not available in current CPUs, Blade's prototype realises them via both **lfence**s and masking. We believe that our framework can be applied to reason about both Blade's design and prototype, but we leave this for future work. The challenges are extending the target languages with fine-grained barriers and formalising the optimal placement of those barriers.

Recent work [27, 36] studied the security of compiler countermeasures by inspecting specific compiled code snippets and detected insecurities in MSVC. Our work extends and complements these results by providing the first rigorous characterization of these countermeasures' security guarantees. In particular, we prove the security of countermeasures for all source programs, rather than simply detecting insecurities on specific examples.

Secure compilation *RSSC* and *RSSP* are instantiations of robustly-safe compilation [2–4, 51]. Like [3, 51], we relate source and target traces using a cross-language relation; however, our target language has a speculative semantics. While program behaviors are sets of traces due to non-determinism in [3, 4], behaviors are single traces for our (deterministic) languages [39].

Fully abstract compilation (*FAC*) is a widely used secure compilation criterion [24, 34, 49, 50, 55, 58]. *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 speculative leaks as it would require explicitly modelling microarchitectural components that are modified speculatively (like caches).

Constant-time-preserving compilation (*CTPC*) has been used to show that compilers preserve constant-time [7, 10, 12]. 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 Verifying if a program satisfies a 2-hypersafety property [20] (like RSNI) is notoriously challenging. Approaches for this include taint-tracking [6, 56] (which over-approximates the 2-hypersafety property with a safety property), secure multi-execution [22] (which runs the code twice in parallel) and self-composition [11, 61] (which runs the code twice sequentially). Our criteria leverage taint-tracking (RSS); we leave investigating criteria based on the other approaches as future work.

Acknowledgements. This work was partially supported by the German Federal Ministry of Education and Research (BMBF) through funding for the CISPA-Stanford Center for Cybersecurity (FKZ: 13N1S0762), by the Community of Madrid under the project S2018/TCS-4339 BLOQUES and the Atracción de Talento Investigador grant 2018-T2/TIC-11732A, by the Spanish Ministry of Science, Innovation, and University under the project RTI2018-102043-B-I00 SCUM and the Juan de la Cierva-Formación grant FJC2018-036513-I, and by a gift from Intel Corporation.

REFERENCES

[1] Martín Abadi. 1998. Protection in Programming-Language Translations. In *Proceedings of the 25th International Colloqium on Automata, Languages and Programming (ICALP)*. Springer.

[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. 2018. When Good Components Go Bad: Formally Secure Compilation Despite Dynamic Compromise. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM.

[3] Carmine Abate, Roberto Blanco, Stefan Ciobaca, Alexandre Durier, Deepak Garg, Cătălin Hriţcu, Marco Patrignani, , Eric Tanter, and Jérémy Thibault. 2020. Trace-Relating Compiler Correctness and Secure Compilation. In *Proceedings of the 29th European Symposium on Programming (ESOP)*. Springer.

[4] Carmine Abate, Roberto Blanco, Deepak Garg, Cătălin Hriţcu, Marco Patrignani, and Jérémy Thibault. 2019. Journey Beyond Full Abstraction: Exploring Robust Property Preservation for Secure Compilation. In *Proceedings of the 32nd IEEE Computer Security Foundations Symposium (CSF)*. IEEE.

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

[6] Peter Aldous and Matthew Might. 2015. Static Analysis of Non-interference in Expressive Low-Level Languages. In *Proceedings of the 22nd International Symposium on Static Analysis (SAS)*. Springer.

[7] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. 2017. Jasmin: High-Assurance and High-Speed Cryptography. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM.

[8] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016. Verifying constant-time implementations. In *Proceedings of the 25th USENIX Security Symposium (USENIX Security)*. USENIX Association.

[9] Musard Balliu, Mads Dam, and Roberto Guanciale. 2020. InSpectre: Breaking and Fixing Microarchitectural Vulnerabilities by Formal Analysis. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM.

[10] Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. 2020. Formal Verification of a Constant-Time Preserving C Compiler. *Proceedings of the ACM on Programming Languages* 4, POPL (2020).

[11] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. 2011. Secure information flow by self-composition. *Mathematical Structures in Computer Science* 21, 6 (2011).

[12] Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. 2018. Secure Compilation of Side-Channel Countermeasures: The Case of Cryptographic Constant-Time. In *Proceedings of the 31st IEEE Computer Security Foundations Symposium (CSF)*. IEEE.

[13] William J. Bowman and Amal Ahmed. 2015. Noninterference for free. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. ACM.

[14] Matteo Busi, Job Noorman, Jo Van Bulck, Letterio Galletta, Pierpaolo Degano, Jan Tobias Mühlberg, and Frank Piessens. 2020. Provably Secure Isolation for Interruptible Enclaved Execution on Small Microprocessors. In *Proceedings of the 33rd IEEE Computer Security Foundations Symposium (CSF)*. IEEE.

[15] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security)*. USENIX Association.

[16] Chandler Carruth. 2018. Speculative Load Hardening. https://llvm.org/docs/SpeculativeLoadHardening.html.

[17] Sunjay Cauligi, Craig Disselkoen, Klaus v Gleissenthall, Dean M. Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. 2020. Constant-Time Foundations for the New Spectre Era. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. ACM.

[18] Kevin Cheang, Cameron Rasmussen, Sanjit A. Seshia, and Pramod Subramanyan. 2019. A Formal Approach to Secure Speculation. In *Proceedings of the 32nd IEEE Computer Security Foundations Symposium (CSF)*. IEEE.

[19] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. 2019. Stealing Intel Secrets from SGX Enclaves via Speculative Execution. In *Proceedings of the 4th IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE.

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

[21] Dominique Devriese, Marco Patrignani, and Frank Piessens. 2016. Fully-Abstract Compilation by Approximate Back-Translation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*

(POPL). ACM.

[22] Dominique Devriese and Frank Piessens. 2010. Noninterference through Secure Multi-execution. In *Proceedings of the 31st IEEE Symposium on Security and Privacy (S&P)*. IEEE.

[23] Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. 2007. A Type Discipline for Authorization Policies. *ACM Transactions on Programming Languages and Systems* 29, 5 (2007).

[24] Cedric Fournet, Nikhil Swamy, Juan Chen, Pierre-Evariste Dagand, Pierre-Yves Strub, and Benjamin Livshits. 2013. Fully Abstract Compilation to JavaScript. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM.

[25] Andrew D. Gordon and Alan Jeffrey. 2003. Authenticity by Typing for Security Protocols. *Journal of Computer Security* 11, 4 (2003).

[26] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. 2016. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM.

[27] Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. 2020. Spectector: Principled detection of speculative information flows. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P)*. IEEE.

[28] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. 2021. Hardware/software contracts for secure speculation. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (S&P)*. IEEE.

[29] Norm Hardy. 1988. The Confused Deputy: (Or Why Capabilities Might Have Been Invented). *SIGOPS Operating Systems Review* 22, 4 (1988).

[30] Jann Horn. 2019. Google Project zero - Issue 1528: speculative execution, variant 4: speculative store bypass. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528.

[31] Intel. 2018. 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.

[32] Intel. 2018. 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.

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

[34] Yannis Juglaret, Cătălin Hriţcu, Arthur Azevedo de Amorim, Boris Eng, and Benjamin C. Pierce. 2016. Beyond Good and Evil: Formalizing the Security Guarantees of Compartmentalizing Compilation. In *Proceedings of the 29th IEEE Computer Security Foundations Symposium (CSF)*. IEEE.

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

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

[37] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P)*. IEEE.

[38] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2018. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *Proceedings of the 12th USENIX Workshop on Offensive Technologies (WOOT)*. USENIX Association.

[39] Xavier Leroy. 2009. A Formally Verified Compiler Back-end. *Journal of Automated Reasoning* 43, 4 (2009), 363–446. http://dx.doi.org/10.1007/s10817-009-9155-4

[40] Sergio Maffeis, Martín Abadi, Cédric Fournet, and Andrew D. Gordon. 2008. Code-Carrying Authorization. In *Proceedings of the 13th European Symposium on Research in Computer Security (ESORICS)*. Springer.

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

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

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

[44] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. 2005. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *Proceedings of the 8th International Conference on Information Security and Cryptology (ICISC)*. Springer.

[45] Max S. New, William J. Bowman, and Amal Ahmed. 2016. Fully Abstract Compilation Via Universal Embedding. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP)*. ACM.

[46] Oleksii Oleksenko, Bohdan Trach, Tobias Reiher, Mark Silberstein, and Christof Fetzer. 2018. You Shall Not Bypass: Employing data dependencies to prevent Bounds Check Bypass. *CoRR* abs/1805.08506 (2018).

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

[48] Marco Patrignani. 2020. Why Should Anyone use Colours? or, Syntax Highlighting Beyond Code Snippets. *CoRR* (2020).

[49] Marco Patrignani, Pieter Agten, Raoul Strackx, Bart Jacobs, Dave Clarke, and Frank Piessens. 2015. Secure compilation to protected module architectures. *ACM Transactions on Programming Languages and Systems* 37, 2 (2015).

[50] Marco Patrignani, Amal Ahmed, and Dave Clarke. 2019. Formal Approaches to Secure Compilation A Survey of Fully Abstract Compilation and Related Work. *Comput. Surveys* 51, 6 (2019).

[51] Marco Patrignani and Deepak Garg. 2021. Robustly Safe Compilation, an Efficient Form of Secure Compilation. *ACM Transactions on Programming Languages and Systems* 43, 1 (2021).

[52] Marco Patrignani and Marco Guarnieri. 2020. Exorcising Spectres with Secure Compilers. *CoRR* abs/1910.08607 (2020).

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

[54] Vineet Rajani, Deepak Garg, and Tamara Rezk. 2016. On Access Control, Capabilities, Their Equivalence, and Confused Deputy Attacks. In *Proceedings of the 29th IEEE Computer Security Foundations Symposium (CSF)*. IEEE.

[55] Gabriel Scherer, Max New, Nick Rioux, and Amal Ahmed. 2018. Fabous Interoperability for ML and a Linear Language. In *FOSSACS '18*. 146–162.

[56] Daniel Schoepe, Musard Balliu, Benjamin Pierce, and Andrei Sabelfeld. 2016. Explicit Secrecy: A Policy for Taint Tracking. In *Proceedings of the 1st IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE.

[57] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. 2019. Netspectre: Read arbitrary memory over network. In *Proceedings of the 24th European Symposium on Research in Computer Security (ESORICS)*. Springer.

[58] Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. 2020. Reasoning about a Machine with Local Capabilities: Provably Safe Stack and Return Pointer Management. *ACM Transactions on Programming Languages and Systems* 42, 1 (2020).

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

[60] David Swasey, Deepak Garg, and Derek Dreyer. 2017. Robust and Compositional Verification of Object Capability Patterns. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017).

[61] Tachio Terauchi and Alex Aiken. 2005. Secure Information Flow as a Safety Problem. In *Proceedings of the 12th International Symposium on Static Analysis (SAS)*. Springer.

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

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

[64] Marco Vassena, Craig Disselkoen, Klaus von Gleissenthall, Sunjay Cauligi, Rami Gökhan Kıcı, Ranjit Jhala, Dean Tullsen, and Deian Stefan. 2021. Automatically Eliminating Speculative Leaks from Cryptographic Code with Blade. *Proceedings of the ACM on Programming Languages* 5, POPL (2021).

[65] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. 2018. oo7: Low-overhead Defense against Spectre Attacks via Binary Analysis. *CoRR* abs/1807.05843 (2018).

APPENDIX INDEX

This appendix contains additional information that the paper could not contain for space constraints. The companion report [52] contains all the language formalisation, theorems and lemmas; it also contains the code snippets of existing compilers that are used to prove Corollary 4.5 and the formalisation of all code snippets in our languages.Appendix A contains details of how to activate each v1 countermeasure. Appendix B contains details of how we formalise the taint tracking rules and how we merge them with the language semantics. Appendix C describes how to secure the non-interprocedural SLH countermeasure. Appendix D provides details of how to apply our methodology to reason about the countermeasures for Spectre v2. Appendix E argues in favour of robustness, i.e., why the way we model attackers is sound.

A ACTIVATING SPECTRE V1 COUNTERMEASURES

In MSVC, the countermeasure can be activated with flag: `/Qspectre`.

In ICC, the countermeasure can be activated with flag: `-mconditional- branch=all-fix`.

In GCC, SLH can be activated with flag: `-mllvm -x86-speculative-load-hardening`.

The non-interprocedural SLH can be activated with flags: `-mllvm -x86-speculative-load-hardening -mllvm -x86-slh-ip=false`.

B TAINT TRACKING OVERVIEW

The language semantics we devise contains two kinds of semantics that operate in parallel: the operational semantics, presented in the paper, and the taint tracking semantics, presented here. Thus, technically, the top-level semantics is parametric in the taint tracking semantics. The semantics of strong languages L and T uses the strong form of taint tracking while the semantics of weak languages L$^-$ and T$^-$ uses the weak form of taint tracking. We now give an in-depth overview of our taint-tracking semantics; see [52] for the full models.

To add taint-tracking to our semantics, we enrich the program state with taint information and devise a taint-tracking semantics that determines how taint is propagated. The top-level semantic judgement is then expressed in terms of the extended program states. An extended state steps if its operational part steps according to the semantics of Section 2.4 and if its taint part steps according to the rules of the taint semantics.

We now define all the elements needed to define the extended program states: extended heaps and extended bindings. In this appendix we indicate the heap, state, and bindings used by the operational semantics with a v suffix, so the H, Ω and B from Section 2.4 are denoted as H_v, Ω_v and B_v respectively. Formally, we indicate taint as $\sigma ::= S \mid U$. Extended heaps H_e extend heaps with the taint of each location, whereas taint heaps H_t only track the taint. Extended heaps H_e can be split/merged in their value-only part H (used for the language semantics) and their taint-only part H_t (used for taint-tracking). We denote this split as $H_e \equiv H_v + H_t$. Just like heaps, extended variable bindings B_e extend the binding with the taint of the variable, whereas taint bindings B_t only track the taint. Still like heaps, bindings can be split/merged as $B_e \equiv B_v + B_t$.

$$\begin{aligned}
\textit{Extended Heaps } H_e &::= \varnothing \mid H_e; n \mapsto v : \sigma \quad \text{where } n \in \mathbb{Z} \\
\textit{Taint Heaps } H_t &::= \varnothing \mid H_t; n \mapsto \sigma \quad \text{where } n \in \mathbb{Z} \\
\textit{Extended Bindings } B_e &::= \varnothing \mid B_e; x \mapsto v : \sigma \\
\textit{Taint Bindings } B_t &::= \varnothing \mid B_t; x \mapsto \sigma \\
\textit{Exended Prog. States } \Omega_e &::= C, H_e, \overline{B_e} \triangleright (s)_{\overline{f}} \\
\textit{Taint States } \Omega_t &::= C, H_t, \overline{B_v} \triangleright (s)_{\overline{f}}
\end{aligned}$$

The taint semantics follows two judgements:

- Judgment $B_t \triangleright e \downarrow \sigma$ reads as "expression e is tainted as σ according to the variable taints B_t".
- 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'".

Below are the most representative rules for the taint tracking used by strong languages:

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

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

Writing to the private heap (Rule T-write-prv) taints the location $(-|n|)$ with the taint of the written expression (σ''). In contrast, reading from the private heap (Rule T-read-prv) taints the variable where the content is stored as unsafe (U) and the read value is set to 0 (this information is not used by the taint-tracking).

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

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

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_v and not just the taints B_t. The rules above rely on a judgement $B_e \triangleright e \downarrow n : \sigma$ which is obtained by joining the result of the expression semantics on the values of B_e and of the taint-tracking semantics on the taints of B_e.

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

The operational and taint single-steps from Section 2.4 are combined according to the judgement $\Omega_e \xrightarrow{\lambda^\sigma} \Omega_e'$ below.

$$\frac{\Omega_v + \Omega_t \equiv \Omega_e \quad \Omega_v' + \Omega_t' \equiv \Omega_e' \quad \Omega_v \xrightarrow{\lambda} \Omega_v' \quad S; \Omega_t \xrightarrow{\sigma} \Omega_t'}{\Omega_e \xrightarrow{\lambda^\sigma} \Omega_e'} \text{(Combine-s-L)}$$

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

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 ($S; \Omega_t \xrightarrow{\sigma} \Omega'_t$). As already mentioned, the pc taint is always safe since there is no speculation in L. Moreover, merging states $\Omega_v + \Omega_t$ results in ignoring the value information accumulated in Ω_t since we rely on the computation performed by the operational semantics for values (Rule Merge-Ω).

In the speculative semantics, as for the non-speculative one, we decouple the operational aspects from the taint-tracking ones. At the top level, speculative program states (Σ_e) are defined as stacks of extended speculation instances (Φ_e), which can be merged/split in their operational (Φ_v) and taint (Φ_t) sub-parts. The operational part (Φ_v) was presented in Section 2. 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 split/merged as $\Phi_e \equiv \Phi_v + \Phi_t$.

$$\textit{Speculative States } \Sigma_e ::= \overline{\Phi_e}$$
$$\textit{Extended Speculation Instance } \Phi_e ::= (\Omega_e, w, \sigma)$$
$$\textit{Speculation Instance Taint } \Phi_t ::= (\Omega_t, \sigma)$$

In the taint tracking used by the speculative semantics, similarly to the operational one, reductions happen at the top of the stack: $\overline{\Phi_t} \overset{\sigma}{\rightsquigarrow} \overline{\Phi'_t}$. Selected rules are below:

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

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

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).

The two operational and taint-tracking single steps from Section 2.6 are combined in a single reduction as follows:

$$\frac{\overline{\Phi_v} + \overline{\Phi_t} \equiv \Sigma_e \qquad \overline{\Phi'_v} + \overline{\Phi'_t} \equiv \Sigma'_e \qquad \overline{\Phi_v} \overset{\lambda}{\rightsquigarrow} \overline{\Phi'_v} \qquad \overline{\Phi_t} \overset{\sigma}{\rightsquigarrow} \overline{\Phi'_t}}{\Sigma_e \overset{\lambda^\sigma}{\rightsquigarrow} \Sigma'_e} \text{(Combine-T)}$$

This reduction is used by the big-step semantics $\Sigma_e \overset{\overline{\lambda^\sigma}}{\Longrightarrow} \Sigma'_e$ that concatenates single labels into traces, which, as before, do not contain microarchitectural actions generated by the attacker.

C 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. As before, compiled code updates pr after every branching instruction. To ensure that pr correctly captures whether we are mis-speculating, we place an lfence as the first instruction of every compiled function.

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

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

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 C.1 (The NISLH compiler is *RSSC*$^-$). $\vdash [\![\cdot]\!]^s_n : RSSC^-$

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

D THE SPECTRE V2 CASE

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

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

- add indirect jumps to our languages and give them a regular semantics (Appendix D.1);
- 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 (Appendix D.2);
- change the call/return semantics in order to model retpolines, i.e., have the return address explicit (Appendix D.3).

With these changes, we can formalise a compiler that introduces the retpoline countermeasure (Appendix D.4) and reason about whether it is secure (Appendix D.5).

D.1 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.

D.2 Speculative Execution of Jumps

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

D.3 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.4 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).

D.5 Security of $[\![\cdot]\!]^r$

We believe $[\![\cdot]\!]^r$ is $RSSC^+$ and we can argue that using the same proof technique described in the paper. As before, the key part of these proofs is reasoning when speculation happens, i.e., in the gray area of Figure 2. 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.

E ROBUSTNESS AND ATTACKERS

Typically, works that deal with Spectre attacks do not consider an active attacker, like us, but a passive one. If we were to adopt the same view, we would have to elide the whole 'robustness' aspect in our paper. We believe that dealing with robustness and with an explicit representation of attackers has its merits, and this is why we opted in favour of it. As already mentioned, these attackers can mount confused deputy attacks [29, 54], unlike passive ones. Then, by using robustness we give a precise characterisation of the attacker and of its power. It is by having this characterisation that we can tell precisely that with a single memory shared between code and attacker, no defence mechanism is possible. Thus we need two memories (in the model), which gets justified in practice by saying that the attacker needs to reside in another process. Deriving this conclusion seems harder –if at all possible– without a concrete notion of attacker. Conversely, our precise definition of the attacker power also limits the scope of the attackers we can meaningfully reason about. Thus, we need to ensure that the model faithfully comprises all attack vectors that practical attackers mounting Spectre attack rely on – which is what we believe the model does. Finally, this approach lets us apply existing secure compilation theory.