

Exorcising Spectres with Secure Compilers

Marco Patrignani*

CISPA Helmholtz Center for Information Security
Germany
mp@cs.stanford.edu

Marco Guarnieri

IMDEA Software Institute
Spain
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].

CCS CONCEPTS

• **Security and privacy** → **Formal security models**; *Systems security*; • **Software and its engineering** → **Compilers**.

KEYWORDS

Spectre, Speculative Execution, Secure Compilation, Robust Safety

ACM Reference Format:

Marco Patrignani and Marco Guarnieri. 2021. Exorcising Spectres with Secure Compilers. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21), November 15–19, 2021, Virtual Event, Republic of Korea*. ACM, New York, NY, USA, 80 pages. <https://doi.org/10.1145/3460120.3484534>

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:

► 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*,

*Also with Stanford University.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CCS '21, November 15–19, 2021, Virtual Event, Republic of Korea

© 2021 Association for Computing Machinery.

ACM ISBN 978-1-4503-8454-4/21/11.

<https://doi.org/10.1145/3460120.3484534>

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 \downarrow fences (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 6). 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 6. 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 $y \geq \text{size}$. 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 attacker to retrieve `A[y]` even for $y \geq \text{size}$.

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 the attacker cannot access. For this, we assume that attacker and component run on separate processes and OS-level memory protection restricts access to the private heap. For example, in Listing 6, 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.¹ 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 Guarnieri et al. [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 6) 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 \mathbf{L} and \mathbf{T}

Technically, we have a pair of source and target languages (\mathbf{L} and \mathbf{T}) for studying security in the strong SNI model and a pair of source and target languages (\mathbf{L}^\downarrow and \mathbf{T}^\downarrow) for studying weak SNI. Strong (\mathbf{L} - \mathbf{T}) and weak (\mathbf{L}^\downarrow - \mathbf{T}^\downarrow) 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 \mathbf{L} - \mathbf{T} ; we introduce the small changes for the weak languages \mathbf{L}^\downarrow - \mathbf{T}^\downarrow in Section 2.7.

The source (\mathbf{L}) and target (\mathbf{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.

¹Adopting such an active attacker model turns our security definitions into ‘robust’ ones, as we discuss in Appendix C.

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

Programs $W, P ::= H, \bar{F}, \bar{I}$ Codebase $C ::= \bar{F}, \bar{I}$ Imports $I ::= f$
 Functions $F ::= f(x) \mapsto s; \text{return};$ Attackers $A ::= H, \bar{F}[\cdot]$
 Heaps $H ::= \emptyset \mid H; n \mapsto v$ where $n \in \mathbb{Z}$
 Expressions $e ::= x \mid v \mid e \oplus e$ Values $v ::= n \in \mathbb{N}$
 Statements $s ::= \text{skip} \mid s; s \mid \text{let } x = e \text{ in } s \mid \text{call } f \ e \mid e := e$
 $\mid e :=_{pr} e \mid \text{let } x = rd \ e \text{ in } s \mid \text{let } x = rd_{pr} \ e \text{ in } s$
 $\mid \text{ifz } e \text{ then } s \text{ else } s \mid \text{let } x = e \text{ (if } e) \text{ in } s \mid \text{lfence}$

We model *components*, i.e., partial programs (P), and *attackers* (A). A (partial) program P defines its heap H , a list of functions \bar{F} , and a list of imports \bar{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 *μ arch. action* δ capturing what a microarchitectural attacker can observe.

μ arch. Acts. $\delta ::= \text{read}(n) \mid \text{write}(n) \mid \text{read}(n \mapsto v)$
 $\mid \text{write}(n \mapsto v) \mid \text{if}(v) \mid \text{rlb}$
Actions $\alpha ::= \text{call } f \ v \mid \text{ret } v$ *Labels* $\lambda ::= \epsilon \mid \alpha? \mid \alpha! \mid \delta$

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

The $\text{read}(n)$ and $\text{write}(n)$ actions denote respectively read and write accesses to the private heap location n . Dually, the $\text{read}(n \mapsto v)$ and $\text{write}(n \mapsto v)$ 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 $\text{if}(v)$ action denotes the outcome of branch instructions and the 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 $\bar{\lambda}$ are sequences of labels. The semantics only track μ arch. 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 μ arch. 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 $C, H, \bar{B} \triangleright (s)_{\bar{f}}$ that consist of a codebase C , a heap H , a stack of local variables \bar{B} , a statement s , and a stack of function names \bar{f} . C is used to look up function bodies, whereas function names \bar{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 $B ::= \emptyset \mid B; x \mapsto v$ *Prog. States* $\Omega ::= C, H, \bar{B} \triangleright (s)_{\bar{f}}$

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-L-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 .

$$\begin{array}{c}
 \frac{\text{(E-if-true)}}{B \triangleright e \downarrow 0} \\
 \hline
 C, H, \bar{B} \cdot B \triangleright \text{ifz } e \text{ then } s \text{ else } s' \xrightarrow{\text{(if}(0))} C, H, \bar{B} \cdot B \triangleright s \\
 \frac{\text{(E-read-prv)}}{B \triangleright e \downarrow n \quad H(-|n|) = v} \\
 \hline
 C, H, \bar{B} \cdot B \triangleright \text{let } x = rd_{pr} \ e \text{ in } s \xrightarrow{\text{read}(-|n|)} \\
 C, H, \bar{B} \cdot B \cup x \mapsto v \triangleright s \\
 \frac{\text{(E-write-prv)}}{B \triangleright e \downarrow n \quad H = H_1; -|n| \mapsto v'; H_2} \\
 B \triangleright e' \downarrow v \quad H' = H_1; -|n| \mapsto v; H_2 \\
 \hline
 C, H, \bar{B} \cdot B \triangleright e :=_{pr} e' \xrightarrow{\text{write}(-|n|)} C, H', \bar{B} \cdot B \triangleright \text{skip}
 \end{array}$$

The rules of conditionals, read, and write emit the related μ arch. actions (from Section 2.3). Specifically, conditionals produce observations recording the outcome of the condition (Rule E-L-if-true), whereas memory operations produce observations recording the accessed memory address (Rule E-L-read-prv and Rule E-L-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 $\cdot \xRightarrow{\bar{\lambda}} \cdot'$ is read: “state \cdot emits trace $\bar{\lambda}$ and becomes \cdot' ”. The most interesting rule is below. As mentioned in Section 2.3, the trace does not contain μ arch. actions performed by the attacker (see the ‘then’ branch, recall that functions in \bar{I} are defined by the attacker).

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

Finally, the behaviour $\text{Beh}(W)$ of a whole program W is the trace $\bar{\lambda}$ generated from the \Rightarrow semantics starting from the initial state of W (indicated as $\cdot_0(W)$) 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 6). Consider $\text{size}=4$. Trace t_{ns} indicates a valid execution of the code in L (without speculation).

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

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_A^i . \square

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 μ arch. 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 \perp when no speculation is happening; its maximum length is a global constant ω that depends on physical characteristics of the CPU like the size of the reorder buffer.

$$\text{Speculative States } \Sigma ::= \bar{\Phi} \quad \text{Speculation Instance } \Phi ::= (\Omega, w)$$

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

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

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

$$\frac{\Omega \xrightarrow{\lambda} \Omega' \quad \Omega \equiv C, H, \bar{B} \triangleright (s; s')_{\bar{f}, f} \quad (s \neq \text{ifz } \dots \text{ and } s \neq \text{lfence}) \text{ or } (C \equiv \bar{F}; \bar{I} \text{ and } f \in \bar{I})}{\bar{\Phi} \cdot (\Omega, n+1) \xrightarrow{\lambda} \bar{\Phi} \cdot (\Omega', n)} \quad (\text{E-T-speculate-action})$$

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

$$\frac{\Omega \equiv C, H, \bar{B} \triangleright s; s' \quad n = 0 \text{ or } \Omega \text{ is stuck}}{\bar{\Phi} \cdot (\Omega, n) \xrightarrow{\epsilon} \bar{\Phi}} \quad (\text{E-T-speculate-rollback})$$

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-att). Note that speculation does not happen in attacker code (condition $f \notin \bar{I}$, recall that f is the function executing now and \bar{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 $\text{Beh}(W)$ of a whole program W is the trace $\bar{\lambda}$ generated, according to the \Rightarrow semantics, starting from the initial state of W until termination.

Example 2.2 (T Trace for Listing 6). 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_A^i .

$$t_{\text{sp}} = \text{call get } 8? \cdot \text{if}(1) \cdot \text{read}(n_A + 8) \cdot \text{read}(n_B + v_A^8) \cdot \text{r1b} \cdot \text{ret}!$$

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

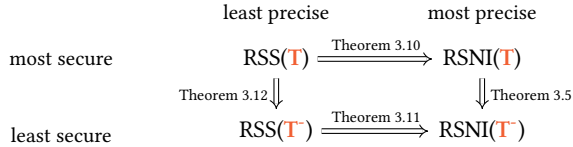
2.7 Weak Languages L^- and T^-

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 $\text{read}(n \mapsto v)$ 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 6, 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 \mathbf{L} , \mathbf{T} , \mathbf{L}^- , and \mathbf{T}^- . Therefore, we write $\text{RSNI}(L)$ and $\text{RSS}(L)$ 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 \mathbf{T} and \mathbf{T}^- since all security definitions trivially hold for the source non-speculative languages \mathbf{L} and \mathbf{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 $W' =_{\mathbf{L}} W$, if they differ only in their private heaps.
- SNI requires comparing the leakage resulting from non-speculative and speculative instructions. The *non-speculative projection* $t \upharpoonright_{nse}$ [27] of a trace t extracts the observations associated with non-speculatively-executed instructions. We obtain $t \upharpoonright_{nse}$ by removing from t all sub-strings enclosed between $\text{if}(\mathbf{v})$ and r1b observations. We illustrate this using an example: $\cdot \upharpoonright_{nse}$ applied to \mathbf{t}_{sp} from Example 2.2 produces $\mathbf{t}_{sp} \upharpoonright_{nse} = \text{call get } 8? \cdot \text{if}(1) \cdot \text{ret}!$.

We now 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)).

$$\vdash W : \text{SNI} \stackrel{\text{def}}{=} \forall W'. \text{ if } W' =_{\mathbf{L}} W$$

$$\begin{aligned} & \text{and } \text{Beh}(\Omega_0(W)) \upharpoonright_{nse} = \text{Beh}(\Omega_0(W')) \upharpoonright_{nse} \\ & \text{then } \text{Beh}(\Omega_0(W)) = \text{Beh}(\Omega_0(W')) \end{aligned}$$

A component P is robustly speculatively non-interferent if it is SNI no matter what valid attacker it is linked to (Definition K.3), where an attacker A is valid ($\vdash A : \text{atk}$) 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)).

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

Example 3.3 (Listing 6 is RSNI in \mathbf{L} and not in \mathbf{T}). Consider the code of Listing 6. As expected, this code is RSNI in \mathbf{L} . Indeed, \mathbf{L} does not support speculative execution and, therefore, for any trace \mathbf{t}_{ns} produced by an \mathbf{L} -program $\mathbf{t}_{ns} \upharpoonright_{nse} = \mathbf{t}_{ns}$.

The same code, however, is not RSNI in \mathbf{T} . Consider the code of Listing 6 (indicated as \mathbf{P}_1) and an attacker \mathbf{A}^8 that calls function get with 8. Since array \mathbf{A} is in the private heap, the low-equivalent program required by Definition K.2 is the same \mathbf{A}^8 linked with some \mathbf{P}_N , which is the same \mathbf{P}_1 with some array \mathbf{N} with contents different from \mathbf{A} in the heap such that $\mathbf{A}[8] \neq \mathbf{N}[8]$. Whole program $\mathbf{A}^8[\mathbf{P}_1]$ generates trace \mathbf{t}_{sp} from Example 2.2 while $\mathbf{A}^8[\mathbf{P}_N]$ generates \mathbf{t}'_{sp} below. We indicate the address of array \mathbf{N} as \mathbf{n}_N and the content of $\mathbf{N}[i]$ as \mathbf{v}_N^i . Low-equivalence yields that addresses are the same ($\mathbf{n}_A + 8 = \mathbf{n}_N + 8$) but contents are not ($\mathbf{v}_A^8 \neq \mathbf{v}_N^8$), and thus \mathbf{B} is accessed at different offsets ($\mathbf{n}_B + \mathbf{v}_A^8 \neq \mathbf{n}_B + \mathbf{v}_N^8$).

$$\mathbf{t}'_{sp} = \text{call get } 8? \cdot \text{if}(1) \cdot \text{read}(\mathbf{n}_N + 8) \cdot \text{read}(\mathbf{n}_B + \mathbf{v}_N^8) \cdot \text{r1b} \cdot \text{ret}!$$

Listing 6 is not RSNI in \mathbf{T} (and neither in \mathbf{T}^-) since the non-speculative projections of \mathbf{t}'_{sp} and of \mathbf{t}_{sp} are the same (see above) while \mathbf{t}'_{sp} and \mathbf{t}_{sp} are *different* ($\text{read}(\mathbf{n}_B + \mathbf{v}_A^8) \neq \text{read}(\mathbf{n}_B + \mathbf{v}_N^8)$). \square

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

THEOREM 3.4 (ALL \mathbf{L} AND \mathbf{L}^- PROGRAMS ARE RSNI).

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

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

$\text{RSNI}(\mathbf{T})$ 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 $\text{RSNI}(\mathbf{T})$.

$\text{RSNI}(\mathbf{T}^-)$, in contrast, corresponds to weak speculative non-interference [28], which allows speculative leaks of information that has been retrieved non-speculatively. Indeed, in \mathbf{T}^- non-speculative reads from the private heap produce actions $\text{read}(\mathbf{n} \mapsto \mathbf{v})$ that additionally disclose the value \mathbf{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 $\text{RSNI}(\mathbf{T})$. That is, $\text{RSNI}(\mathbf{T})$ ensures the absence only of leaks of speculatively-accessed data.

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

THEOREM 3.5 ($\text{RSNI}(\mathbf{T})$ IMPLIES $\text{RSNI}(\mathbf{T}^+)$).

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

As shown in [28], strong and weak speculative non-interference (that is, $\text{RSNI}(\mathbf{T})$ and $\text{RSNI}(\mathbf{T}^+)$) 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 \mathbf{L} , \mathbf{L}^+ , \mathbf{T} , and \mathbf{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 A. These rules simply extend Rule E-L-read-prv with taint, which is highlighted in gray.

$$\begin{array}{c}
 \text{(T-read-prv)} \\
 \frac{B \triangleright e \downarrow n : \sigma' \quad H(-|n|) = v : \sigma'' \quad \sigma = \sigma'' \sqcap \sigma'}{\sigma_{pc} : C, H, \bar{B} \cdot B \triangleright \text{let } x = rd_{pr} \ e \text{ in } s \xrightarrow{\text{read}(-|n|)} \sigma \sqcup \sigma_{pc}} \\
 C, H, \bar{B} \cdot B \cup x \mapsto v : U \triangleright s \\
 \text{(T-read-prv-weak)} \\
 \frac{B \triangleright e \downarrow n : \sigma' \quad H(-|n|) = v : \sigma'' \quad \sigma = \sigma'' \sqcap \sigma'}{\sigma_{pc} : C, H, \bar{B} \cdot B \triangleright \text{let } x = rd_{pr} \ e \text{ in } s \xrightarrow{\text{read}(-|n| \mapsto v)} \sigma \sqcup \sigma_{pc}} \\
 C, H, \bar{B} \cdot B \cup x \mapsto v : \sigma' \sqcup \sigma_{pc} \triangleright s
 \end{array}$$

• 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 (\sqcap) and greatest-lower-bound (\sqcup) operators. For simplicity, we report here the key cases: $S \sqcap U = U$ and $S \sqcup U = S$.

• 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 \mathbf{T} and \mathbf{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 \sqcap , 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 \mathbf{L} and \mathbf{T} taints the variable where the data is stored as unsafe (U) (Rule T-read-prv). In contrast, the weak mechanism of \mathbf{L}^+ and \mathbf{T}^+ , taints the target value with the greatest-lower-bound of the taints of the memory address and of the program counter (Rule T-L-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 $\text{if}(v)$ 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-L-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 \mathbf{L} , \mathbf{L}^+ , \mathbf{T} , and \mathbf{T}^+ extended with the corresponding taint-tracking mechanisms outlined above whenever we talk about speculative safety. That is, for speculative safety, programs in \mathbf{L} , \mathbf{L}^+ , \mathbf{T} , and \mathbf{T}^+ produce traces $\bar{\lambda}^\sigma$ 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)*).

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

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)*).

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

Example 3.8 (Listing 6 is RSS in \mathbf{L} and not in \mathbf{T}). The code of Listing 6 is RSS in \mathbf{L} because σ_{pc} is always S and, therefore, all actions are tainted as S . The code, however, is neither RSS in \mathbf{T}

nor in \mathbf{T} . For this, consider the trace from Example 2.2. The taint-tracking mechanism taints the actions as follows:

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

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 \mathbf{U}) and it depends on data retrieved from the private heap (which \mathbf{T} 's taint-tracking taints as \mathbf{U}). \square

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

THEOREM 3.9 (ALL \mathbf{L} AND \mathbf{L}^- PROGRAMS ARE RSS).

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

In contrast, RSS' guarantees are different for \mathbf{T} and \mathbf{T}^- , which are equipped with distinct taint tracking mechanisms.

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

THEOREM 3.10 ($\text{RSS}(\mathbf{T})$ OVER-APPROXIMATES $\text{RSNI}(\mathbf{T})$).

- 1) $\forall \mathbf{P}. \text{if } \vdash \mathbf{P} : \text{RSS}(\mathbf{T}) \text{ then } \vdash \mathbf{P} : \text{RSNI}(\mathbf{T})$
- 2) $\exists \mathbf{P}. \vdash \mathbf{P} : \text{RSNI}(\mathbf{T}) \text{ and } \not\vdash \mathbf{P} : \text{RSS}(\mathbf{T})$

To understand point 1, observe that $\text{RSS}(\mathbf{T})$ ensures that only safe observations are produced by a program \mathbf{P} . This, in turn, ensures that no information originating from the private heap is leaked through speculatively-executed instructions in \mathbf{P} . Therefore, \mathbf{P} satisfies $\text{RSNI}(\mathbf{T})$ 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 7, which always accesses $B[A[y]]$. This code is $\text{RSNI}(\mathbf{T})$ 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 $\text{RSS}(\mathbf{T})$ because speculative memory accesses will produce \mathbf{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.

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

THEOREM 3.11 ($\text{RSS}(\mathbf{T}^-)$ OVER-APPROXIMATES $\text{RSNI}(\mathbf{T}^-)$).

- 1) $\forall \mathbf{P}. \text{if } \vdash \mathbf{P} : \text{RSS}(\mathbf{T}^-) \text{ then } \vdash \mathbf{P} : \text{RSNI}(\mathbf{T}^-)$
- 2) $\exists \mathbf{P}. \vdash \mathbf{P} : \text{RSNI}(\mathbf{T}^-) \text{ and } \not\vdash \mathbf{P} : \text{RSS}(\mathbf{T}^-)$

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

THEOREM 3.12 ($\text{RSS}(\mathbf{T})$ IMPLIES $\text{RSS}(\mathbf{T}^-)$).

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

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 $\mathbf{L}-\mathbf{T}$ or $\mathbf{L}^--\mathbf{T}^-$. Criteria instantiated with the strong languages (say $\text{RSSP}(\mathbf{L}, \mathbf{T})$) are indicated with a + (that is, RSSP^+). Those instantiated with weak languages (say $\text{RSNIP}(\mathbf{L}^-, \mathbf{T}^-)$) are indicated with a - (that is, RSNIP^-). When we omit the ‘sign’, we refer to both criteria. For simplicity, we only present the strong criteria (for $\mathbf{L}-\mathbf{T}$), weak ones are defined identically (but for $\mathbf{L}^--\mathbf{T}^-$).

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 (RSSP^+).

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

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 M.2) states that a compiler is RSSC if for any target-level attacker \mathbf{A} that generates a trace $\bar{\lambda}^\sigma$, we can build a source-level attacker \mathbf{A} that generates a trace $\bar{\lambda}^\sigma$ that is related to $\bar{\lambda}^\sigma$. A source trace $\bar{\lambda}^\sigma$ and a target trace $\bar{\lambda}^\sigma$ are related (denoted with $\bar{\lambda}^\sigma \approx \bar{\lambda}^\sigma$) if the target trace contains all the actions of the source trace, plus possible interleavings of safe (\mathbf{S}) actions (Rules Trace-Relation-Safe and Trace-Relation-Safe-Heap). All other actions must be the same (i.e., \equiv , Rules Trace-Relation-Same-Act and Trace-Relation-Same-Heap).

$$\begin{array}{c}
\begin{array}{c}
\text{(Trace-Relation-Same)} \\
\frac{\bar{\lambda}^\sigma \approx \bar{\lambda}^\sigma \quad \alpha^\sigma \equiv \alpha^\sigma}{\bar{\lambda}^\sigma \cdot \alpha^\sigma \approx \bar{\lambda}^\sigma \cdot \alpha^\sigma} \\
\text{(Trace-Relation-Safe)} \\
\frac{\bar{\lambda}^\sigma \approx \bar{\lambda}^\sigma}{\bar{\lambda}^\sigma \approx \bar{\lambda}^\sigma \cdot \alpha^S}
\end{array}
\qquad
\begin{array}{c}
\text{(Trace-Relation-Same-Heap)} \\
\frac{\bar{\lambda}^\sigma \approx \bar{\lambda}^\sigma \quad \delta^\sigma \equiv \delta^\sigma}{\bar{\lambda}^\sigma \cdot \delta^\sigma \approx \bar{\lambda}^\sigma \cdot \delta^\sigma} \\
\text{(Trace-Relation-Safe-Heap)} \\
\frac{\bar{\lambda}^\sigma \approx \bar{\lambda}^\sigma}{\bar{\lambda}^\sigma \approx \bar{\lambda}^\sigma \cdot \delta^S}
\end{array}
\end{array}$$

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 (RSSC^+).

$$\vdash [\![\cdot]\!] : \text{RSSC}^+ \stackrel{\text{def}}{=} \forall \mathbf{P} \in \mathbf{L}, \mathbf{A}, \bar{\lambda}^\sigma. \text{if } \text{Beh}(\mathbf{A} [\![\mathbf{P}]\!]]) = \bar{\lambda}^\sigma$$

then $\exists A, \overline{\lambda^\sigma}. \text{Beh}(A [P]) = \overline{\lambda^\sigma}$ and $\overline{\lambda^\sigma} \approx \overline{\lambda^\sigma}$

THEOREM 4.3 (RSSP AND RSSC ARE EQUIVALENT).

$$\begin{aligned} \forall [\cdot]. \vdash [\cdot] : \text{RSSP}^+ &\iff \vdash [\cdot] : \text{RSSC}^+ \\ \forall [\cdot]. \vdash [\cdot] : \text{RSSP}^- &\iff \vdash [\cdot] : \text{RSSC}^- \end{aligned}$$

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 M.6). The reason is that we only directly prove that compilers do *not* attain *RSNIP*. This kind of proof is simple already (Corollary M.7), and we do not need a property-free criterion.

Definition 4.4 (*RSNIP*⁺).

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

COROLLARY 4.5 ($\not\vdash [\cdot] : \text{RSNIP}^+$).

$$\not\vdash [\cdot] : \text{RSNIP}^+ \stackrel{\text{def}}{=} \exists P \in \mathbf{L}. \vdash P : \text{RSNI}(\mathbf{L}) \text{ and } \not\vdash [P] : \text{RSNI}(\mathbf{T})$$

Let us now unfold the corollary in order to understand what must be proven to show that a compiler is not *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 $A [[P']]$ is the same as $A [[P]]$ but with a different private heap.

$$\begin{aligned} \not\vdash [P] : \text{RSNI}(\mathbf{T}) &= \exists A. \vdash A : \text{atk} \text{ and given } A [[P']] =_L A [[P]] \\ &\text{ we have } \text{Beh}(\Omega_0(A [[P]])) \upharpoonright_{\text{nse}} = \text{Beh}(\Omega_0(A [[P']])) \upharpoonright_{\text{nse}} \\ &\text{ and } \text{Beh}(\Omega_0(A [[P]]) \neq \text{Beh}(\Omega_0(A [[P']])) \end{aligned}$$

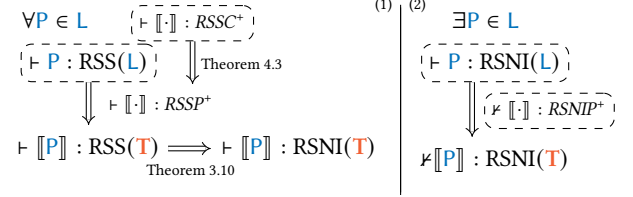
That is, we need to find a program P and an attacker A that violate *RSNI*. Finding these existentially-quantified program (and attacker) may be hard. Fortunately, failed attempts at proving *RSSC* often provide hints for how to do this. \square

We remark that the insecurity part of our methodology is used to show its completeness w.r.t. 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 \mathbf{L} and \mathbf{T} . There are similar chains of implications for \mathbf{L}^- and \mathbf{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 M.7, whereas point (B) holds for any P (Theorem 3.9).

Our security criteria, instantiated for the strong ($\mathbf{L-T}$) and weak ($\mathbf{L-T}^-$) 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 leaks. 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].² However, MSVC fails in inserting some necessary `lfences`, thereby producing insecure code that is not $\text{RSNI}(\mathbf{T})$ and that is vulnerable to Spectre-style attacks.

To show this, we follow Corollary M.7 and provide a program that is $\text{RSNI}(\mathbf{L})$ and its compilation is not $\text{RSNI}(\mathbf{T})$. The program we consider, which is $\text{RSNI}(\mathbf{L})$ (Theorem 3.9), is given in Listing 8.

```
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 8, and it speculatively leaks whether $A[y]$ is 0 through the branch statement in line 3, i.e., it violates $\text{RSNI}(\mathbf{T})$. 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].³

We model this countermeasure with $\llbracket \cdot \rrbracket^f$, a homomorphic compiler that takes a component in \mathbf{L} and translates all of its subparts to \mathbf{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.

$\llbracket \text{ifz } e \text{ then } s \text{ else } s' \rrbracket^f = \text{ifz } \llbracket e \rrbracket^f \text{ then } \{\text{lfence}; \llbracket s \rrbracket^f\} \text{ else } \{\text{lfence}; \llbracket s' \rrbracket^f\}$

It should come at no surprise that $\llbracket \cdot \rrbracket^f$ is RSSC^+ (Theorem O.1). In \mathbf{T} , the only source of speculation are branches (Rule E-T-speculate-if-att) 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 \mathbf{S} , which makes all actions \mathbf{S} . These actions are easy to relate to their source-level counterparts since they are generated according to the non-speculative semantics.

THEOREM 5.1 (ICC IS SECURE FOR L-T). $\vdash \llbracket \cdot \rrbracket^f : \text{RSSC}^+$

²The countermeasure can be activated with the `/Qspectre` flag.

³The countermeasure can be activated with flag: `-mconditional-branch=all-fix`

5.3 Speculative Load Hardening

Clang implements a countermeasure called speculative load hardening [16] (SLH) that works as follows:⁴

- 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 M.7, we do this by providing a program that is $\text{RSNI}(\mathbf{L})$ and that is compiled to a program that is not $\text{RSNI}(\mathbf{T})$. The program in Listing 9 differs from Listing 6 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 $\text{RSNI}(\mathbf{T})$.

⁴SLH can be activated with flag: `-mllvm -x86-speculative-load-hardening`

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 $\llbracket \cdot \rrbracket^s$ 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, $\llbracket \cdot \rrbracket^s$ 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 $-1 :=_{pr} \dots$). 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, $\llbracket \cdot \rrbracket^s$ 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 ...**) 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, $\llbracket \cdot \rrbracket^s$ masks the output of these statement based on the predicate bit stored in **pr**.

As stated in Theorem Q.1, 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 6.

THEOREM 5.2 (SLH IS SECURE FOR L^-T^-). $\vdash \llbracket \cdot \rrbracket^s : RSSC^-$

$\llbracket \cdot \rrbracket^s$ 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 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 $\text{glb}(U)$ of data taint (S) and pc taint (U). Since $S \sqcup U = S$ (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 $\llbracket \cdot \rrbracket^{ss}$ compiler that takes components in L and outputs compiled code in T . $\llbracket \cdot \rrbracket^{ss}$ differs from $\llbracket \cdot \rrbracket^s$ 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 x_f), retrieving the predicate bit and storing it in variable **pr**, conditionally masking the value of x_f , and, finally, performing the memory access using 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 $L-T$). $\vdash \llbracket \cdot \rrbracket^{ss} : RSSC^+$

$\llbracket \cdot \rrbracket^{ss}$ 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.⁵ Consider the program in Listing 14 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 $RSNI(T^-)$.

It is also possible to secure the non-interprocedural variant of SLH. We model NISLH as $\llbracket \cdot \rrbracket_n^s$ 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.

$$\begin{aligned} \llbracket \begin{array}{l} f(x) \mapsto s; \\ \text{return;} \end{array} \rrbracket_n^s &= f(x) \mapsto \left| \begin{array}{l} \text{lfence; let pr=1 in} \\ \llbracket s \rrbracket_n^s; \text{return;} \end{array} \right. \\ \llbracket \begin{array}{l} \text{ifz } e \\ \text{then } s \\ \text{else } s' \end{array} \rrbracket_n^s &= \left| \begin{array}{l} \text{let } x_f = \llbracket e \rrbracket_n^s \text{ in} \\ \text{ifz } x_f \text{ then let pr=pr } \vee \neg x_f \text{ in } \llbracket s \rrbracket_n^s \\ \text{else let pr=pr } \vee x_f \text{ in } \llbracket s' \rrbracket_n^s \end{array} \right. \end{aligned}$$

This compiler is also $RSSC^-$ since (1) it correctly tracks whether we are speculating (this time using local variable **pr** rather than location -1 as in $\llbracket \cdot \rrbracket^s$), (2) speculation across function boundaries is blocked by **lfence** statements, and (3) masking is done as in $\llbracket \cdot \rrbracket^s$.

THEOREM 5.4 (THE NISLH COMPILER IS $RSSC^-$). $\vdash \llbracket \cdot \rrbracket_n^s : RSSC^-$

In a similar way, one can construct a secure, non-interprocedural version of $\llbracket \cdot \rrbracket^{ss}$ that satisfies $RSSC^+$.

⁵Flags: -mllvm -x86-speculative-load-hardening -mllvm -x86-slh-ip=false

$$\begin{aligned}
\llbracket \text{H}; \bar{\text{F}}; \bar{\text{I}} \rrbracket^s &= \llbracket \text{H} \rrbracket^s \cup (-1 \mapsto 1 : \text{S}); \llbracket \bar{\text{F}} \rrbracket^s; \llbracket \bar{\text{I}} \rrbracket^s & \llbracket \text{H}, -n \mapsto v : \text{U} \rrbracket^s &= \llbracket \text{H} \rrbracket^s, -\llbracket n \rrbracket^s - 1 \mapsto \llbracket v \rrbracket^s : \text{U} \\
\llbracket \text{ifz } e \text{ then } s \text{ else } s' \rrbracket^s &= \text{let } x_f = \llbracket e \rrbracket^s \text{ in let } pr = rd_{pr} - 1 \text{ in let } x_f = 0 \text{ (if } pr) \text{ in ifz } x_f \text{ then } -1 :=_{pr} pr \vee -x_f; \llbracket s \rrbracket^s \text{ else } -1 :=_{pr} pr \vee x_f; \llbracket s' \rrbracket^s \\
\llbracket e :=_{pr} e' \rrbracket^s &= \text{let } x_f = \llbracket e \rrbracket^s + 1 \text{ in let } x'_f = \llbracket e' \rrbracket^s \text{ in let } pr = rd_{pr} - 1 \text{ in let } x_f = 0 \text{ (if } pr) \text{ in let } x'_f = 0 \text{ (if } pr) \text{ in } x_f :=_{pr} x'_f \\
\llbracket \text{let } x = rd_{pr} e \text{ in } s \rrbracket^s &= \text{let } x_f = \llbracket e \rrbracket^s + 1 \text{ in let } pr = rd_{pr} - 1 \text{ in let } x = rd_{pr} x_f \text{ in let } x = 0 \text{ (if } pr) \text{ in } \llbracket s \rrbracket^s \\
\llbracket \text{let } x = rd_{pr} e \text{ in } s \rrbracket^{ss} &= \text{let } x_f = \llbracket e \rrbracket^{ss} + 1 \text{ in let } pr = rd_{pr} - 1 \text{ in let } x_f = 0 \text{ (if } pr) \text{ in let } x = rd_{pr} x_f \text{ in } \llbracket s \rrbracket^{ss}
\end{aligned}$$

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

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 ($A = \langle\langle A \rangle\rangle$) 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.

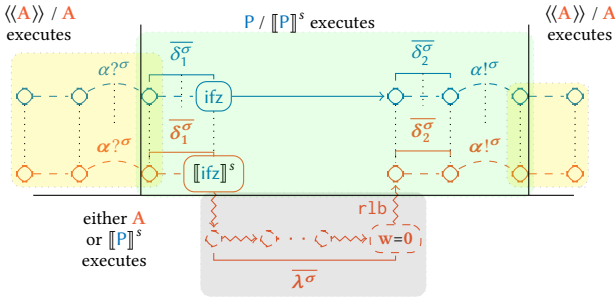


Figure 2: Diagram depicting the proof that $\llbracket \cdot \rrbracket^s$ is RSSC.

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 $\langle\langle A \rangle\rangle$ 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 $\llbracket s \rrbracket^s$ does one or more steps and emits a trace such

that both ending states and traces are related (green areas, related traces are connected by black-dotted lines). This proof is straightforward except for the compilation of `ifz e then s else s'` since it triggers speculation in T (grey area). After $\llbracket \text{ifz } e \text{ then } s \text{ else } s' \rrbracket^s$ is executed, speculation starts and the cross-language state relation is temporarily broken (the stack of target states is not a singleton, so the cross-language state relation cannot hold). Speculative execution continues for w steps in both attacker and compiled code and generating a trace $\bar{\lambda}^\sigma$. We then prove that $\bar{\lambda}^\sigma$ is related to the empty source trace because all actions in $\bar{\lambda}^\sigma$ 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 $\llbracket \cdot \rrbracket^s$ and $\llbracket \cdot \rrbracket^{ss}$, so they are secure.

The compiler $\llbracket \cdot \rrbracket^f$ can be proved secure in a simpler way since speculative reductions immediately trigger an `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, \bullet denotes that *all* compiled programs satisfy the criterion and \circ denotes that *some* compiled programs violates it.

	RSNI (T)	RSNI (T')
<code>lfence(MSVC), SLH-no-interp</code>	\circ	\circ
<code>lfence(ICC)/[·]^f, SSLH/[·]^{ss}</code>	\bullet	\bullet
<code>SLH(Clang)/[·]^s, NISLH/[·]^s</code>	\circ	\bullet

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 $\llbracket \cdot \rrbracket^f$, is secure. The model satisfies $RSSP^+$ (Theorem O.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 $\llbracket \cdot \rrbracket^{ss}$ is secure for $\mathbf{L}^{\mathbf{T}}$. The model satisfies $RSSP^{\mathbf{T}}$ (Theorem Q.1) and, as a result, produces only compiled programs that satisfy weak speculative non-interference, that is, $RSNI(\mathbf{T})$. 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 $\llbracket \cdot \rrbracket^{ss}$ is secure for $\mathbf{L}^{\mathbf{T}}$. The model satisfies $RSSP^+$ (Theorem 5.3) and produces compiled programs that satisfy speculative non-interference, that is, $RSNI(\mathbf{T})$. Thus, compiled programs have *no* speculative leaks.
- Non-interprocedural SLH, denoted SLH-no-interp, is insecure. It violates $RSNIP^{\mathbf{T}}$ and produces programs that violate both $RSNI(\mathbf{T})$ and $RSNI(\mathbf{T}^{\mathbf{T}})$. Hence, compiled programs might still be vulnerable to Spectre attacks.
- Non-interprocedural SLH can be made secure as we show in Section 5.3.4. This variant, denoted NISLH and modelled by $\llbracket \cdot \rrbracket_n^s$, is secure for $\mathbf{L}^{\mathbf{T}}$ and it produces programs that are free of speculatively leaks involving speculatively-accessed data.

Additional security guarantees. In addition to $RSNIP$, the secure compilers $\llbracket \cdot \rrbracket^f$, $\llbracket \cdot \rrbracket^s$, $\llbracket \cdot \rrbracket^{ss}$, and $\llbracket \cdot \rrbracket_n^s$ also preserve the non-speculative behavior of source programs. That is, if two source programs \mathbf{W} and \mathbf{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 **Ifences** 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 $\llbracket \cdot \rrbracket^f$ as an example; the same argument applies to $\llbracket \cdot \rrbracket^s$, $\llbracket \cdot \rrbracket^{ss}$, and $\llbracket \cdot \rrbracket_n^s$. 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 $\mathbf{W} \in \mathbf{L}$ that is NI, we obtain that $\llbracket \mathbf{W} \rrbracket^f$ is NI if we restrict ourselves to the non-speculative projection of traces since $\llbracket \mathbf{W} \rrbracket^f$ preserves the non-speculative behavior of \mathbf{W} . Since $\llbracket \mathbf{W} \rrbracket^f$ is $RSNI(\mathbf{T})$, the full traces do not leak more than their non-speculative projections and thus $\llbracket \mathbf{W} \rrbracket^f$ is also non-interferent.

The security guarantees of NI depend on the underlying language. For strong languages $\mathbf{L}^{\mathbf{T}}$, NI ensures that programs are *constant-time* with respect to the private heap (in \mathbf{L} , we have classical constant-time [8, 44] while in \mathbf{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 $\mathbf{L}^{\mathbf{T}}$, NI ensures a form of *sandboxing* where programs (1) cannot access information from 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 \mathbf{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 \mathbf{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 B). In contrast, reasoning about SSBB would be similar to reasoning about

$\llbracket \cdot \rrbracket^f$ since SSBBs 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 **Ifence** 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.

8 CONCLUSION

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

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 Colloquium 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 Hrițcu, 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 Hrițcu, 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_SoftwareTechniquesforManagingSpeculation_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 Evtvyushkin, 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. Sheshia, 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 Guarneri, 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 Guarneri, 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 Hrițcu, 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 Pietrowski, 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).

A TAIN 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_v (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 $B_e \equiv B_v + B_t$.

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

The taint semantics follows two judgements:

- Judgement $B_t \triangleright e \downarrow \sigma$ 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:

$$\frac{
 \begin{array}{c}
 \text{(T-write-prv)} \\
 B_e \triangleright e \downarrow n : \sigma \quad B_e \triangleright e' \downarrow _ : \sigma'' \quad H'_t = H_t \cup -|n| \mapsto \sigma'' \\
 \hline
 \sigma_{pc}; C, H_t, \overline{B_e} \cdot B_e \triangleright e :=_{pr} e' \xrightarrow{\sigma \sqcup \sigma_{pc}} C, H'_t, \overline{B_e} \cdot B_e \triangleright \text{skip}
 \end{array}
 \quad
 \frac{
 \begin{array}{c}
 \text{(T-read-prv)} \\
 B \triangleright e \downarrow n : \sigma' \quad n_a = -|n| \quad H_t(n_a) = \sigma'' \quad \sigma = \sigma'' \sqcap \sigma' \\
 \hline
 \sigma_{pc}; C, H_t, \overline{B_e} \cdot B_e \triangleright \text{let } x = rd_{pr} e \text{ in } s \xrightarrow{\sigma \sqcup \sigma_{pc}} \\
 C, H_t, \overline{B_e} \cdot B_e \cup x \mapsto 0 : U \triangleright s
 \end{array}
 }{
 }$$

Writing to the private heap (Rule T-L-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 ($\sigma' \sqcup \sigma_{pc}$) instead of U .

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

To correctly taint memory accesses, we need to evaluate expression e to derive the accessed location $|n|$; see, for instance, Rule T-L-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 .

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

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

$$\begin{array}{c}
\text{(Combine-s-L)} \\
\frac{\dot{v} + \dot{t} \equiv \dot{e} \quad \dot{v}' + \dot{t}' \equiv \dot{e}' \quad \dot{v} \xrightarrow{\lambda} \dot{v}' \quad S; \dot{t} \xrightarrow{\sigma} \dot{t}'}{\dot{e} \xrightarrow{\lambda\sigma} \dot{e}'} \\
\text{(Merge-}\Omega\text{)} \\
\frac{H_v + H_t \equiv H_e \quad \overline{B}'_v + \overline{B}'_t \equiv \overline{B}'_e \quad \overline{B}_v + \overline{B}_t \equiv \overline{B}'_e}{C; H_v; \overline{B}_v \triangleright s + C; H_t; \overline{B}_t \triangleright s' \equiv C; H_e; \overline{B}'_e \triangleright s}
\end{array}$$

The operational semantics determines how states reduce ($\dot{v} \xrightarrow{\lambda} \dot{v}'$), whereas the taint-tracking semantics determines the action's label and how taints are updated ($S; \dot{t} \xrightarrow{\sigma} \dot{t}'$). As already mentioned, the pc taint is always safe since there is no speculation in **L**. Moreover, merging states $\dot{v} + \dot{t}$ results in ignoring the value information accumulated in \dot{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 $\Phi_e \equiv \Phi_v + \Phi_t$.

$$\text{Speculative States } \Sigma_e ::= \overline{\Phi}_e$$

$$\text{Extended Speculation Instance } \Phi_e ::= (\Omega_e, w, \sigma)$$

$$\text{Speculation Instance Taint } \Phi_t ::= (\Omega_t, \sigma)$$

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

$$\begin{array}{c}
\text{(T-T-speculate-action)} \\
\frac{\sigma'; \Omega_t \xrightarrow{\sigma} \Omega'_t \quad \Omega_t \equiv C, H_t, \overline{B} \triangleright s; s' \quad s \neq \text{ifz_then_else_and } s \neq \text{lfence}}{\overline{\Phi}_t \cdot (\Omega_t, \sigma) \xrightarrow{\sigma} \overline{\Phi}'_t \cdot (\Omega'_t, \sigma)} \\
\text{(T-T-speculate-if)} \\
\frac{\Omega_t \equiv C, H_t, \overline{B} \cdot B \triangleright (s; s')_{\bar{f}.f} \quad s \equiv \text{ifz } e \text{ then } s'' \text{ else } s''' \quad \sigma'; \Omega_t \xrightarrow{\sigma} \Omega'_t \quad C \equiv \bar{F}; \bar{I} \quad f \notin \bar{I}}{\text{if } B \triangleright e \downarrow 0 : \sigma \text{ then } \Omega'_t \equiv C, H_t, \overline{B} \cdot B \triangleright s''; s' \quad \text{if } B \triangleright e \downarrow n : \sigma \text{ and } n > 0 \text{ then } \Omega'_t \equiv C, H_t, \overline{B} \cdot B \triangleright s''; s'} \\
\overline{\Phi}_t \cdot (\Omega_t, \sigma) \xrightarrow{\sigma} \overline{\Phi}'_t \cdot (\Omega'_t, \sigma) \cdot (\Omega'_t, U)}
\end{array}$$

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-attacker). 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-attacker).

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

$$\frac{\text{(Combine-T)} \quad \overline{\Phi}_v + \overline{\Phi}_t \equiv \Sigma_e \quad \overline{\Phi}'_v + \overline{\Phi}'_t \equiv \Sigma'_e \quad \overline{\Phi}_v \xrightarrow{\lambda} \overline{\Phi}'_v \quad \overline{\Phi}_t \xrightarrow{\sigma} \overline{\Phi}'_t}{\Sigma_e \xrightarrow{\lambda\sigma} \Sigma'_e}$$

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

B 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 return-based *trampoline*. 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 B.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 B.2);
- change the call/return semantics in order to model retpolines, i.e., have the return address explicit (Appendix B.3).

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

B.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.

B.2 Speculative Execution of Jumps

To focus only on speculation over jumps, we would replace Rule E-T-speculate-if-att (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.

B.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.

B.4 The Retpoline Countermeasure

The retpoline countermeasure $\llbracket \cdot \rrbracket^r$ 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 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*).

B.5 Security of $\llbracket \cdot \rrbracket^r$

We believe $\llbracket \cdot \rrbracket^r$ 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 $\llbracket \cdot \rrbracket^r$, 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.

C 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 (among them, applying existing secure compilation theory), 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; see Section 6 for a discussion on this topic.

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.

D L: A SOURCE LANGUAGE WITHOUT SPECULATION

L is a sequential, untyped while language with expressions and statements. A L component (i.e., a partial program) is a collection of function definitions and imports (functions it requires of the programs it links against). We could add more, but this suffices. A component links against an attacker (a context) in order to create a whole program, which is then evaluated starting from its `main` function, which is defined in the attacker. Expressions are given a big step semantics (\Downarrow). Statements are given a labelled structural operational semantics (\rightarrow) that records calls and returns between a component and the attacker. The labels generated by a program are collected in a trace semantics (\Rightarrow), whose actions are tagged as secure or insecure. In L, no speculation is possible, so only safe actions are produced; the metavariable σ is introduced for modularity of rules since it will be expanded in T.

The heap is a map from integers to values. To prevent the attacker from operating directly on the heap of the component – a power that is not given to him normally – the heap consists of two parts. The positive one is shared, while the negative one is private to the component. Instructions to access the private heap cannot be used in the context.

Importantly, the heap is preallocated, so all locations from 0 up to plus and minus infinity are already allocated and initialised to 0.

Call actions only contain the value passed because they model the attack where code is tricked into passing a speculatively-load value directly to the attacker. Return actions are only needed as proof devices. Technically, we need two different write actions: writing on the private heap only leaks the content while writing on the public heap also leaks the value.

D.1 Syntax

$$\begin{aligned}
\text{Whole Programs } \mathbb{W} &::= H, \bar{F}, \bar{I} \\
\text{Programs } \mathbb{P} &::= H, \bar{F}, \bar{I} \\
\text{Components } \mathbb{C} &::= \bar{F}, \bar{I} \\
\text{Contexts } \mathbb{A} &::= H, \bar{F}[\cdot] \\
\text{Imports } \mathbb{I} &::= f \\
\text{Functions } \mathbb{F} &::= f(x) \mapsto s; \text{return;} \\
\text{Operations } \oplus &::= + \mid - \mid \cdot \\
\text{Comparisons } \otimes &::= == \mid < \mid > \\
\text{Values } v &::= n \in \mathbb{N} \\
\text{Expressions } e &::= x \mid v \mid e \oplus e \mid e \otimes e \\
\text{Statements } s &::= \text{skip} \mid s; s \mid \text{let } x = e \text{ in } s \mid \text{if } e \text{ then } s \text{ else } s \\
&\quad \mid \text{call } f \ e \mid e := e \mid \text{let } x = \text{rd } e \text{ in } s \mid \text{let } x = \text{rd}_{\text{pr}} e \text{ in } s \mid e :=_{\text{pr}} e \\
\text{Security Tags } \sigma &::= S \mid U \\
\mathbb{B} &::= \emptyset \mid \mathbb{B}; x \mapsto v : \sigma \\
\mathbb{B}_v &::= \emptyset \mid \mathbb{B}_v; x \mapsto v \\
\mathbb{B}_t &::= \emptyset \mid \mathbb{B}_t; x \mapsto \sigma \\
\mathbb{H} &::= \emptyset \mid \mathbb{H}; n \mapsto v : \sigma \\
\mathbb{H}_v &::= \emptyset \mid \mathbb{H}_v; n \mapsto v \\
\mathbb{H}_t &::= \emptyset \mid \mathbb{H}_t; n \mapsto \sigma \\
\dot{\cdot} &::= \mathbb{C}; \mathbb{H}; \bar{\mathbb{B}} \triangleright s \\
\dot{v} &::= \mathbb{C}; \mathbb{H}_v; \bar{\mathbb{B}}_v \triangleright s \\
\dot{t} &::= \mathbb{C}; \mathbb{H}_t; \bar{\mathbb{B}}_t \triangleright s \\
\text{Labels } \lambda &::= \epsilon \mid \alpha \mid \delta \mid \zeta \\
\text{Actions } \alpha &::= (\text{call } f \ v?) \mid (\text{call } f \ v!) \mid (\text{ret}!) \mid (\text{ret}?) \\
\text{Heap\&Pc Act. } s \delta &::= (\text{read}(n)) \mid (\text{write}(n)) \mid (\text{if}(v)) \mid (\text{write}(n \mapsto v)) \\
\text{Traces } \bar{\lambda}^\sigma &::= \emptyset \mid \bar{\lambda}^\sigma \cdot \alpha^\sigma \mid \bar{\lambda}^\sigma \cdot \delta^\sigma
\end{aligned}$$

Heaps and bindings contain the tags of the values they map.

The additional condition on a trace $\bar{\lambda}$ is that it is list with this shape: $\overline{\alpha? \delta \alpha!}$. We do not filter nor reorder heap actions in traces because they represent cache-visible actions. The attacker is assumed to operate concurrently to our program, so it can effectively observe a difference between `write(0)` and `write(0) · write(0)`.

For simplicity of the trace semantics, reading a location is a statement (despite it being pure, and thus an expression).

In order to model conditional updates, we do not perform substitutions for variables, instead each function has its stack of bindings B where to allocate and lookup variables. Each function can only access its stack frame for simplicity. We concatenate whole stacks of bindings as $B \cdot B'$. We update the bindings for x in a stack B to v by writing $B \cup x \mapsto v$. If an update is made for a binding that is not in the stack, then the update just adds the binding.

The taints are safe S and unsafe U and they are ordered in the usual safety lattice $S \leq U$. The tag of an action-generating expression is the tag of the data involved in that expression. When data is generated (Rules E-L-op and E-L-comparison), it is tagged with the label resulting of the lub (\sqcup) of the label of all its subdatas. A value (natural number, location or boolean) is safe (Rule E-L-val), a variable has the same tag of its content (Rule E-L-var). Reading a value from the heap tags the value (and thus the variable) as unsafe (Rule E-L-read).

D.2 Dynamic Semantics

Rules L-Jump-Internal to L-Jump-OUT dictate the kind of a jump between two functions: if internal to the component/attacker, in(from the attacker to the component) or out(from the component to the attacker). Rule L-Plug tells how to obtain a whole program from a component and an attacker. Rule L-Whole tells when a program is whole. Rule L-Initial State tells the initial state of a whole program.

We change the way the list of imports is used between partial and whole programs. For partial programs, imports are effectively imports, i.e., the functions that contexts define and that the program relies on. So a context can define more functions. In whole programs, we change the imports to be the list of all context defined function (Rule L-Plug) to keep track of what is and what is not context.

Helpers

$$\begin{array}{c}
 \text{(Intfs)} \quad \frac{C = \bar{F}, \bar{I}}{C.\text{intfs} = \bar{I}} \quad \text{(Funs)} \quad \frac{C = \bar{F}, \bar{I}}{C.\text{funs} = \bar{F}} \\
 \\
 \text{(L-Jump-Internal)} \quad \frac{((f' \in \bar{I} \wedge f \in \bar{I}) \vee (f' \notin \bar{I} \wedge f \notin \bar{I}))}{\bar{I} \vdash f, f' : \text{internal}} \quad \text{(L-Jump-IN)} \quad \frac{f \in \bar{I} \wedge f' \notin \bar{I}}{\bar{I} \vdash f, f' : \text{in}} \quad \text{(L-Jump-OUT)} \quad \frac{f \notin \bar{I} \wedge f' \in \bar{I}}{\bar{I} \vdash f, f' : \text{out}} \\
 \\
 \text{(Call Label - call)} \quad \frac{C.\text{intfs} \vdash f', f : \text{in}}{C; f', f; \text{call}; v \vdash \text{call } f \ v?} \quad \text{(Call Label - callback)} \quad \frac{C.\text{intfs} \vdash f', f : \text{out}}{C; f', f; \text{call}; v \vdash \text{call } f \ v!} \quad \text{(Call Label - internal)} \quad \frac{C.\text{intfs} \vdash f', f : \text{internal}}{C; f', f; \text{call}; v \vdash \epsilon} \\
 \\
 \text{(Ret Label - return)} \quad \frac{C.\text{intfs} \vdash f', f : \text{in}}{C; f', f; \text{ret} \vdash \text{ret}!} \quad \text{(Ret Label - returnback)} \quad \frac{C.\text{intfs} \vdash f', f : \text{out}}{C; f', f; \text{ret} \vdash \text{ret}?)} \quad \text{(Ret Label - internal)} \quad \frac{C.\text{intfs} \vdash f', f : \text{internal}}{C; f', f; \text{ret} \vdash \epsilon} \\
 \\
 \text{(L-Plug)} \quad \frac{A \equiv H, \bar{F}[\cdot] \quad P \equiv H', \bar{F}', \bar{I} \quad \vdash \bar{F}'; \bar{F}, \bar{I} : \text{whole} \quad \text{main} \in \text{names}(\bar{F}) \quad \text{dom}(H) \cap \text{dom}(H') = \emptyset \quad \forall n \mapsto v : \sigma \in H', n < 0 \text{ and } \sigma = U}{A[P] = H; H', \bar{F}, \bar{F}', \text{dom}(\bar{F})} \\
 \\
 \text{(L-Whole)} \quad \frac{\text{names}(\bar{F}) \cap \text{names}(\bar{F}') = \emptyset \quad \text{names}(\bar{I}) \subseteq \text{names}(\bar{F}) \cup \text{names}(\bar{F}') \quad \text{fv}(\bar{F}) \cup \text{fv}(\bar{F}') = \emptyset}{\vdash \bar{F}'; \bar{F}, \bar{I} : \text{whole}} \quad \text{(L-Initial State)} \quad \frac{H_0 = H'' \cup H \cup H' \quad H' = \{n \mapsto 0 : S \mid n \in \mathbb{N} \setminus \text{dom}(H)\} \quad H'' = \{-n \mapsto 0 : U \mid n \in \mathbb{N}, -n \notin \text{dom}(H)\}}{\cdot_0((H, \bar{F}, \bar{I})) = \bar{F}, \bar{I}, H_0, \emptyset \cdot x \mapsto 0 \triangleright \text{call main } x} \\
 \\
 \text{(L-Terminal State)} \quad \frac{\nexists', \lambda. \xrightarrow{\lambda} \cdot'}{\vdash \cdot : \perp} \\
 \\
 \text{(Merge-B-base)} \quad \frac{}{\emptyset + \emptyset = \emptyset} \quad \text{(Merge-B-ind)} \quad \frac{B_v + B_t = B}{B_v; x \mapsto v + B_t; x \mapsto \sigma = B; x \mapsto v : \sigma} \quad \text{(Merge-H-base)} \quad \frac{}{\emptyset + \emptyset = \emptyset} \quad \text{(Merge-H-ind)} \quad \frac{H_v + H_t = H}{H_v; n \mapsto v + H_t; n \mapsto \sigma = H; n \mapsto v : \sigma} \\
 \\
 \text{(Merge-s-S)} \quad \frac{H_v + H_t = H \quad \bar{B}'_v + \bar{B}'_t = \bar{B}' \quad \bar{B}_v + \bar{B}_t = \bar{B}'}{C; H_v; \bar{B}'_v \triangleright s + C; H_t; \bar{B}'_t \triangleright s' = C; H; \bar{B}' \triangleright s}
 \end{array}$$

D.2.1 Component Semantics.

Judgements

$$B_v \triangleright e \downarrow v$$

Expression e big-steps to value v .

$$B_t \triangleright e \downarrow \sigma$$

Expression e is tainted σ .

$$B \triangleright e \downarrow v : \sigma$$

Expression e big-steps to value v tagged σ .

$$\cdot_v \xrightarrow{\lambda} \cdot'_v$$

State \cdot_v small-steps to \cdot'_v and emits action λ .

$$\sigma'; \cdot_t \xrightarrow{\sigma} \cdot'_t$$

With pc tainted σ' , the action of state \cdot_v is tainted σ .

$$\cdot \xrightarrow{\lambda^\sigma} \cdot'$$
 aka

$$C, H, \overline{B} \triangleright (s)_{\overline{f}} \xrightarrow{\lambda^\sigma} C, H', \overline{B}' \triangleright (s')_{\overline{f}}$$

Statement s reduces to s' and evolves the rest accordingly, emitting tagged label λ^σ .

$$\cdot \xrightarrow{\overline{\lambda^\sigma}} \cdot'$$

Program state \cdot steps to \cdot'

emitting tagged trace $\overline{\lambda^\sigma}$.

$$P \rightsquigarrow \overline{\lambda^\sigma}$$

Whole program P produces tagged trace $\overline{\lambda^\sigma}$

$$B_v \triangleright e \downarrow v$$

$$\frac{}{B_v \triangleright v \downarrow v} \quad \frac{\text{(E-L-val)}}{B_v(x) = v \quad B_v \triangleright x \downarrow v} \quad \frac{\text{(E-L-op)} \quad B_v \triangleright e \downarrow n \quad B_v \triangleright e' \downarrow n' \quad n'' = [n \oplus n']}{B_v \triangleright e \oplus e' \downarrow n''} \quad \frac{\text{(E-L-comparison)} \quad B_v \triangleright e \downarrow n \quad B_v \triangleright e' \downarrow n' \quad n'' = [n \otimes n']}{B_v \triangleright e \otimes e' \downarrow n''}$$

$$B_t \triangleright e \downarrow \sigma$$

$$\frac{}{B_t \triangleright v \downarrow S} \quad \frac{\text{(T-L-val)}}{B_t(x) = \sigma \quad B_t \triangleright x \downarrow \sigma} \quad \frac{\text{(T-L-op)} \quad B_t \triangleright e \downarrow \sigma \quad B_t \triangleright e' \downarrow \sigma' \quad \sigma'' = \sigma \sqcup \sigma'}{B_t \triangleright e \oplus e' \downarrow \sigma''} \quad \frac{\text{(T-L-comparison)} \quad B_t \triangleright e \downarrow \sigma \quad B_t \triangleright e' \downarrow \sigma' \quad \sigma'' = \sigma \sqcup \sigma'}{B_t \triangleright e \otimes e' \downarrow \sigma''}$$

$$B \triangleright e \downarrow v : \sigma$$

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

The taint propagation for operations is standard, using the lub. A more refined version is possible (i.e., not propagating taint for an op with an identity operand, or propagating taint with glb), but not needed in this case.

$$\cdot_v \xrightarrow{\lambda} \cdot'_v$$

$$\frac{}{C, H_v, \overline{B}_v \triangleright \text{skip}; s \xrightarrow{\epsilon} C, H_v, \overline{B}_v \triangleright s} \quad \frac{\text{(E-L-step)} \quad C, H_v, \overline{B}_v \triangleright s \xrightarrow{\lambda} C, H'_v, \overline{B}'_v \triangleright s'}{C, H_v, \overline{B}_v \triangleright s; s'' \xrightarrow{\lambda} C, H'_v, \overline{B}'_v \triangleright s'; s''} \quad \frac{\text{(E-L-if-true)} \quad B_v \triangleright e \downarrow 0}{C, H_v, \overline{B}_v \cdot B_v \triangleright \text{ifz } e \text{ then } s \text{ else } s' \xrightarrow{(\text{if}(0))} C, H_v, \overline{B}_v \cdot B_v \triangleright s} \quad \frac{\text{(E-L-if-false)} \quad B_v \triangleright e \downarrow n \quad n > 0}{C, H_v, \overline{B}_v \cdot B_v \triangleright \text{ifz } e \text{ then } s \text{ else } s' \xrightarrow{(\text{if}(n))} C, H_v, \overline{B}_v \cdot B_v \triangleright s}$$

$$\begin{array}{c}
\text{(E-L-letin)} \\
\frac{B_V \triangleright e \downarrow v}{C, H_V, \overline{B}_V \cdot B_V \triangleright \text{let } x = e \text{ in } s \xrightarrow{\epsilon} C, H_V, \overline{B}_V \cdot B_V \cup x \mapsto v \triangleright s} \\
\text{(E-L-write)} \\
\frac{B_V \triangleright e \downarrow n \quad H_V = H_{V1}; |n| \mapsto v'; H_{V2} \quad B_V \triangleright e' \downarrow v \quad H'_V = H_{V1}; |n| \mapsto v; H_{V2}}{C, H_V, \overline{B}_V \cdot B_V \triangleright e := e' \xrightarrow{\text{write}(|n| \mapsto v)} C, H'_V, \overline{B}_V \cdot B_V \triangleright \text{skip}} \\
\text{(E-L-read)} \\
\frac{B_V \triangleright e \downarrow n \quad H_V = H_{V1}; |n| \mapsto v; H_{V2}}{C, H_V, \overline{B}_V \cdot B_V \triangleright \text{let } x = \text{rd } e \text{ in } s \xrightarrow{\text{read}(|n|)} C, H_V, \overline{B}_V \cdot B_V \cup x \mapsto v \triangleright s} \\
\text{(E-L-write-prv)} \\
\frac{B_V \triangleright e \downarrow n \quad H_V = H_{V1}; n_a \mapsto v'; H_{V2} \quad B_V \triangleright e' \downarrow v \quad n_a = -|n| \quad H'_V = H_{V1}; n_a \mapsto v; H_{V2}}{C, H_V, \overline{B}_V \cdot B_V \triangleright e :=_{\text{pr}} e' \xrightarrow{\text{write}(n_a)} C, H'_V, \overline{B}_V \cdot B_V \triangleright \text{skip}} \\
\text{(E-L-read-prv)} \\
\frac{B_V \triangleright e \downarrow n \quad n_a = -|n| \quad H_V = H_{V1}; n_a \mapsto v; H_{V2}}{C, H_V, \overline{B}_V \cdot B_V \triangleright \text{let } x = \text{rd}_{\text{pr}} e \text{ in } s \xrightarrow{\text{read}(n_a)} C, H_V, \overline{B}_V \cdot B_V \cup x \mapsto v \triangleright s} \\
\text{(E-L-call)} \\
\frac{\overline{f}' = \overline{f}''; f' \quad f(x) \mapsto s; \text{return}; \in C.\text{funs} \quad C; f', f; \text{call}; v \vdash \lambda \quad B_V \triangleright e \downarrow v}{C, H_V, \overline{B}_V \cdot B_V \triangleright (\text{call } f \ e)_{\overline{f}'} \xrightarrow{\lambda} C, H_V, \overline{B}_V \cdot B_V \cdot x \mapsto v \triangleright (s; \text{return};)_{\overline{f}', f}} \\
\text{(E-L-return)} \\
\frac{\overline{f}' = \overline{f}''; f' \quad C; f', f; \text{ret} \vdash \lambda}{C, H_V, \overline{B}_V \cdot B_V \triangleright (\text{return};)_{\overline{f}', f} \xrightarrow{\lambda} C, H_V, \overline{B}_V \triangleright (\text{skip})_{\overline{f}'}}
\end{array}$$

$$\sigma_{\text{pc}}; t \xrightarrow{\sigma} t'$$

$$\begin{array}{c}
\text{(T-L-sequence)} \\
\frac{\sigma_{\text{pc}}; C, H_t, \overline{B} \triangleright \text{skip}; s \xrightarrow{\epsilon} C, H_t, \overline{B} \triangleright s}{\sigma_{\text{pc}}; C, H_t, \overline{B} \triangleright s \xrightarrow{\sigma} C, H_t, \overline{B} \triangleright s} \\
\text{(T-L-if-true)} \\
\frac{B \triangleright e \downarrow v : \sigma}{\sigma_{\text{pc}}; C, H_t, \overline{B} \cdot B \triangleright \text{ifz } e \text{ then } s \text{ else } s' \xrightarrow{\sigma} C, H_t, \overline{B} \cdot B \triangleright s} \\
\text{(T-L-write)} \\
\frac{B \triangleright e \downarrow n : \sigma \quad B \triangleright e' \downarrow _ : \sigma'' \quad H'_t = H_t \cup |n| \mapsto S}{\sigma_{\text{pc}}; C, H_t, \overline{B} \cdot B \triangleright e := e' \xrightarrow{\sigma \sqcup \sigma''} C, H'_t, \overline{B} \cdot B \triangleright \text{skip}} \\
\text{(T-L-write-prv)} \\
\frac{B \triangleright e \downarrow n : \sigma \quad B \triangleright e' \downarrow _ : \sigma'' \quad n_a = -|n| \quad H'_t = H_t \cup |n| \mapsto \sigma''}{\sigma_{\text{pc}}; C, H_t, \overline{B} \cdot B \triangleright e :=_{\text{pr}} e' \xrightarrow{\sigma} C, H'_t, \overline{B} \cdot B \triangleright \text{skip}} \\
\text{(T-L-step)} \\
\frac{\sigma_{\text{pc}}; C, H_t, \overline{B} \triangleright s \xrightarrow{\sigma} C, H'_t, \overline{B}'_t \triangleright s'}{\sigma_{\text{pc}}; C, H_t, \overline{B} \triangleright s; s'' \xrightarrow{\sigma} C, H'_t, \overline{B}'_t \triangleright s'; s''} \\
\text{(T-L-letin)} \\
\frac{B \triangleright e \downarrow v : \sigma}{\sigma_{\text{pc}}; C, H_t, \overline{B} \cdot B \triangleright \text{let } x = e \text{ in } s \xrightarrow{\epsilon} C, H_t, \overline{B} \cdot B \cup x \mapsto \sigma \triangleright s} \\
\text{(T-L-read)} \\
\frac{B \triangleright e \downarrow n : \sigma \quad H_t(n_a) = \sigma}{\sigma_{\text{pc}}; C, H_t, \overline{B} \cdot B \triangleright \text{let } x = \text{rd } e \text{ in } s \xrightarrow{\sigma} C, H_t, \overline{B} \cdot B \cup x \mapsto 0 : \sigma \triangleright s} \\
\text{(T-L-read-prv)} \\
\frac{B \triangleright e \downarrow n : \sigma' \quad n_a = -|n| \quad H_t(n_a) = \sigma \quad \sigma'' = \sigma \sqcup \sigma'}{\sigma_{\text{pc}}; C, H_t, \overline{B} \cdot B \triangleright \text{let } x = \text{rd}_{\text{pr}} e \text{ in } s \xrightarrow{\sigma''} C, H_t, \overline{B} \cdot B \cup x \mapsto 0 : \sigma \triangleright s} \\
\text{(T-L-call-internal)} \\
\frac{C.\text{intfs} \vdash f, f' : \text{internal} \quad \overline{f}' = \overline{f}''; f' \quad f(x) \mapsto s; \text{return}; \in C.\text{funs} \quad B \triangleright e \downarrow v : \sigma}{\sigma_{\text{pc}}; C, H_t, \overline{B} \cdot B \triangleright (\text{call } f \ e)_{\overline{f}'} \xrightarrow{\epsilon} C, H_t, \overline{B} \cdot B \cdot x \mapsto \sigma \triangleright (s; \text{return};)_{\overline{f}', f}} \\
\text{(T-L-call)} \\
\frac{\overline{f}' = \overline{f}''; f' \quad f(x) \mapsto s; \text{return}; \in C.\text{funs} \quad C.\text{intfs} \not\vdash f', f : \text{internal} \quad B \triangleright e \downarrow v : \sigma}{\sigma_{\text{pc}}; C, H_t, \overline{B} \cdot B \triangleright (\text{call } f \ e)_{\overline{f}'} \xrightarrow{\sigma} C, H_t, \overline{B} \cdot B \cdot x \mapsto S \triangleright (s; \text{return};)_{\overline{f}', f}} \\
\text{(T-L-ret-internal)} \\
\frac{\overline{f}' = \overline{f}''; f' \quad C.\text{intfs} \vdash f, f' : \text{internal}}{\sigma_{\text{pc}}; C, H_t, \overline{B} \cdot B \triangleright (\text{return};)_{\overline{f}', f} \xrightarrow{\epsilon} C, H_t, \overline{B} \triangleright (\text{skip})_{\overline{f}'}} \\
\text{(T-L-return)} \\
\frac{\overline{f}' = \overline{f}''; f' \quad C.\text{intfs} \not\vdash f, f' : \text{internal}}{\sigma_{\text{pc}}; C, H_t, \overline{B} \cdot B \triangleright (\text{return};)_{\overline{f}', f} \xrightarrow{S} C, H_t, \overline{B} \triangleright (\text{skip})_{\overline{f}'}}
\end{array}$$

Note that we perform a trick for the taint tracking of reading values. We do not know the value that is being read, and so we read a default 0, that, however, we taint correctly. When the stack of bindings of the taint tracking is merged, the values stored here are not considered, only the taint, so it is ok to track 0 because it will not affect the semantics at all.

$$\boxed{\cdot \xrightarrow{\lambda^\sigma} \cdot'}$$

$$\begin{array}{c} \text{(Combine-s-S)} \\ \frac{\begin{array}{c} \cdot v + \cdot t = \cdot \quad \cdot v' + \cdot t' = \cdot' \\ \cdot v \xrightarrow{\lambda} \cdot v' \quad S; \cdot t \xrightarrow{\sigma} \cdot t' \end{array}}{\cdot \xrightarrow{\lambda^{\sigma \cap S}} \cdot'} \end{array}$$

Reading from the private heap taints the read value as unsafe, so no matter what you write in the private heap, it'll be always U . Writing to the public heap taints the written value as safe, so no matter what you read from the public heap, it'll be always S . In both cases, the action is labelled with the lub. If i read a private value, that value is U but the read itself may be S . If i write a public value, that value is S but the write itself may be U , if done speculatively (and thus rolled back). Public writes taint their action with the lub of data and address because an attacker sees both effects via that read. A private write taints only with the address because the attacker does not see the data, only the address.

$$\boxed{\cdot \xrightarrow{\bar{\lambda}^\sigma} \cdot'}$$

$$\begin{array}{c} \text{(E-L-single)} \\ \frac{\cdot \xrightarrow{\alpha^\sigma} \cdot' \quad \cdot = \bar{F}, \bar{I}, H, B \triangleright (s)_{\bar{f}.f} \quad \cdot' = \bar{F}, \bar{I}, H', B' \triangleright (s')_{\bar{f}'.f'}}{\text{if } f == f' \text{ and } f \in I \text{ then } \bar{\lambda}^\sigma = \epsilon \text{ else } \bar{\lambda}^\sigma = \alpha^S} \quad \cdot \xrightarrow{\bar{\lambda}^\sigma} \cdot'} \end{array} \quad \begin{array}{c} \text{(E-L-silent)} \\ \frac{\cdot \xrightarrow{\epsilon} \cdot'}{\cdot \Rightarrow \cdot'} \end{array} \quad \begin{array}{c} \text{(E-L-trans)} \\ \frac{\cdot \xrightarrow{\bar{\lambda}_1^{\sigma_1}} \cdot'' \quad \cdot'' \xrightarrow{\bar{\lambda}_2^{\sigma_2}} \cdot'}{\cdot \xrightarrow{\bar{\lambda}_1^{\sigma_1} \cdot \bar{\lambda}_2^{\sigma_2}} \cdot'} \end{array}$$

Rule E-L-single tells that when you have a single action, if that action is done within the context, then no action is shown. Otherwise, if the action is done within the component, or if the action is done between component and context, then the action is shown and it is tagged as S . All generated actions are S despite their label (σ) because the only source of unsafety is speculation, which only exists in T . Technically, we should take the \sqcup of the pc (which here is always S) and of the data, but since this always results ins S , we just write S .

$$\boxed{P \rightsquigarrow \bar{\lambda}^\sigma}$$

$$\begin{array}{c} \text{(E-L-trace)} \\ \frac{\exists \cdot. \vdash \cdot : \perp \quad \cdot_0 (P) \xrightarrow{\bar{\lambda}^\sigma} \cdot}{P \rightsquigarrow \bar{\lambda}^\sigma} \end{array} \quad \begin{array}{c} \text{(E-L-behaviour)} \\ \text{Beh}(P) = \left\{ \bar{\lambda}^\sigma \mid P \rightsquigarrow \bar{\lambda}^\sigma \right\} \end{array}$$

All traces are finite and the set of traces includes all traces that terminate.

Alternative Semantics Definition.

$$\boxed{P \rightsquigarrow \bar{\lambda}^\sigma}$$

$$\begin{array}{c} \text{(E-L-trace)} \\ \frac{\exists \cdot. \cdot_0 (P) \xrightarrow{\bar{\lambda}^\sigma} \cdot}{P \rightsquigarrow \bar{\lambda}^\sigma} \end{array}$$

This definition drops the condition of \cdot being final in Rule E-L-trace. This way, the set of behaviours includes all prefixes of a prefix, i.e., it is subset closed. All the compiler proofs work with this definition too without concerns. Having this definition complicates relating Definition J.3 (Speculative Safety (SS(L))) and Definition K.3 (Robust Speculative Non-Interference) so we do not use this.

E T: ADDING SPECULATION TO L

T extends **L** by adding the ability to speculate as well as the programming constructs that are used as countermeasures against speculation: a conditional move and the lfence. **T** defines a new notion of program states in order to model speculative execution and it adds rules to the semantics of statements to capture speculation (\rightsquigarrow). Thus, the trace alphabet of **T** is richer than the one of **L**.

Notation-wise, **T** includes all that is **L**, i.e., all that is typeset in **blue** also exists in **red**.

E.1 Syntax

All elements from **L** also exist in this language.

$$\begin{aligned}
 \text{Statements } s &::= \dots \mid \text{lfence} \mid \text{let } x = e \text{ (if } e) \text{ in } s \\
 \text{Speculative States } \Sigma &::= n; \bar{\Phi} \\
 \text{Speculation Instance } \Phi &::= (\Omega, w, \sigma) \\
 \text{Speculation Instance Vals. } \Phi_V &::= n; \Omega_V, \omega \\
 \text{Speculation Instance Taint } \Phi_t &::= \Omega_t, \sigma \\
 \text{Actions } \alpha &::= r \mid b \\
 \text{Window } \omega &::= n \mid \perp \\
 \text{Droppable Names } D &::= \emptyset \mid D, x
 \end{aligned}$$

Reading the Notation: A stack of elements e_1, \dots, e_n is indicated with \bar{e} . So, given an \bar{e} , its elements are possibly distinct. If we want to refer to the top of a stack of elements, we will use notation $\bar{e} \cdot e$, where e is the top, \bar{e} is the rest of the stack, and e is possibly different from all elements in \bar{e} .

We define pattern-matching on windows as follows. The form $k + 1$ matches \perp and any number different from 0 so that k is interpreted as \perp if $\omega = \perp$ and it is interpreted as k if $\omega = k + 1$.

Droppable names are a technicality needed for cross-language relations, as explained in Appendix O.1.3.

E.2 Dynamic Semantics

E.2.1 Auxiliary Functions.

$$\frac{\text{(Merge-S)}}{\frac{\Omega_V + \Omega_t = \bar{\Omega}}{n; \Omega_V, \omega + \Omega_t, \sigma = n; \bar{\Omega}, \omega, \sigma}}$$

E.2.2 Component Semantics.

$$\begin{array}{ll}
 \Phi_V \xrightarrow{\lambda} \bar{\Phi}'_V & \text{Speculative state } \Phi_V \text{ evolves into stack } \bar{\Phi}'_V \text{ emitting action } \alpha. \\
 \bar{\Phi}_V \xrightarrow{\lambda} \bar{\Phi}'_V & \text{Stack } \bar{\Phi}_V \text{ evolves into } \bar{\Phi}'_V. \\
 \Phi_t \xrightarrow{\sigma} \bar{\Phi}'_t & \text{Speculative tainted state } \Phi_t \text{ evolves into stack } \bar{\Phi}'_t \text{ emitting taint } \sigma. \\
 \bar{\Phi}_t \xrightarrow{\sigma} \bar{\Phi}'_t & \text{Stack } \bar{\Phi}_t \text{ evolves into } \bar{\Phi}'_t. \\
 \Sigma \xrightarrow{\alpha\sigma} \Sigma' & \text{Speculative state } \Sigma \text{ evolves into } \Sigma' \text{ emitting tainted action } \alpha\sigma.
 \end{array}$$

$$\boxed{\Omega_V \xrightarrow{\lambda} \Omega'_V}$$

(E-T-lfence)

$$C, H_V, \bar{B}_V \triangleright \text{lfence} \xrightarrow{\epsilon} C, H_V, \bar{B}_V \triangleright \text{skip}$$

(E-T-cmove-true)

$$x \in \text{dom}(B_V) \quad B_V \triangleright e' \downarrow 0 \quad B_V \triangleright e \downarrow v$$

$$C, H_V, \bar{B}_V \cdot B_V \triangleright \text{let } x = e \text{ (if } e') \text{ in } s \xrightarrow{\epsilon} C, H_V, \bar{B}_V \cdot B_V \cup x \mapsto v \triangleright s$$

(E-T-cmove-false)

$$x \in \text{dom}(B_V) \quad B_V \triangleright e' \downarrow n \quad n > 0 \quad B_V(x) = v$$

$$C, H_V, \bar{B}_V \cdot B_V \triangleright \text{let } x = e \text{ (if } e') \text{ in } s \xrightarrow{\epsilon} C, H_V, \bar{B}_V \cdot B_V \cup x \mapsto v \triangleright s$$

$$\boxed{\sigma_{pc}; \Omega_t \xrightarrow{\sigma} \Omega'_t}$$

$$\begin{array}{c}
\text{(T-T-lfence)} \\
\hline
\sigma_{pc}; C, H_t, \bar{B} \triangleright \text{lfence} \xrightarrow{\epsilon} C, H_t, \bar{B} \triangleright \text{skip} \\
\text{(T-T-cmove-true)} \\
\hline
x \in \text{dom}(B) \quad B \triangleright e' \downarrow 0 : \sigma' \quad B \triangleright e \downarrow v : \sigma \quad \sigma'' = \sigma \sqcup \sigma' \\
\sigma_{pc}; C, H_t, \bar{B} \cdot B \triangleright \text{let } x = e \text{ (if } e') \text{ in } s \xrightarrow{\epsilon} C, H_t, \bar{B} \cdot B \cup x \mapsto v : \sigma'' \triangleright s \\
\text{(E-T-cmove-false)} \\
\hline
x \in \text{dom}(B) \quad B \triangleright e' \downarrow n : \sigma' \quad n > 0 \quad B(x) = v : \sigma \quad \sigma'' = \sigma \sqcup \sigma' \\
\sigma_{pc}; C, H_t, \bar{B} \cdot B \triangleright \text{let } x = e \text{ (if } e') \text{ in } s \xrightarrow{\epsilon} C, H_t, \bar{B} \cdot B \cup x \mapsto v : \sigma'' \triangleright s
\end{array}$$

In a conditional move we require that x is always bound afterwards.

$$\boxed{\overline{\Phi}_V \rightsquigarrow \overline{\Phi}'_V}$$

$$\begin{array}{c}
\text{(E-T-speculate-stack)} \\
\hline
\overline{\Phi}_V \rightsquigarrow \overline{\Phi}'_V \\
\hline
\overline{\Phi}_V \cdot \Phi_V \rightsquigarrow \overline{\Phi}_V \cdot \overline{\Phi}'_V
\end{array}$$

$$\boxed{\overline{\Phi}_V \rightsquigarrow \overline{\Phi}'_V}$$

$$\begin{array}{c}
\text{(E-T-speculate-epsilon)} \\
\hline
\Omega_V \xrightarrow{\epsilon} \Omega'_V \quad \Omega_V \equiv C, H_V, \bar{B}_V \triangleright s; s' \quad s \neq \text{ifz_then_else_and } s \neq \text{lfence} \\
w, (\Omega_V, n+1) \xrightarrow{\epsilon} w, (\Omega'_V, n) \\
\text{(E-T-speculate-lfence)} \\
\hline
\Omega_V \xrightarrow{\epsilon} \Omega'_V \quad \Omega_V \equiv C, H_V, \bar{B}_V \triangleright s; s' \quad s \equiv \text{lfence} \\
w, (\Omega_V, n+1) \xrightarrow{\epsilon} w, (\Omega'_V, 0) \\
\text{(E-T-speculate-action)} \\
\hline
\Omega_V \xrightarrow{\lambda} \Omega'_V \quad \Omega_V \equiv C, H_V, \bar{B}_V \triangleright s; s' \quad s \neq \text{ifz_then_else_and } s \neq \text{lfence} \\
w, (\Omega_V, n+1) \xrightarrow{\lambda} w, (\Omega'_V, n) \\
\text{(E-T-speculate-if)} \\
\hline
\Omega_V \xrightarrow{\alpha} \Omega'_V \quad \Omega_V \equiv C, H_V, \bar{B}_V \cdot B_V \triangleright (s; s')_{\bar{f}, f} \quad s \equiv \text{if } e \text{ then } s'' \text{ else } s''' \\
C \equiv \bar{F}; \bar{I} \quad f \notin \bar{I} \\
\text{if } B_V \triangleright e \downarrow 0 \text{ then } \Omega''_V \equiv C, H_V, \bar{B}_V \cdot B_V \triangleright s'''; s' \\
\text{if } B_V \triangleright e \downarrow n \text{ and } n > 0 \text{ then } \Omega''_V \equiv C, H_V, \bar{B}_V \cdot B_V \triangleright s''; s' \\
j = \min(w, n) \\
\text{(E-T-speculate-rollback)} \\
\hline
w, (\Omega_V, 0) \rightsquigarrow^{\text{rlb}} w, \emptyset \\
\hline
w, (\Omega_V, n+1) \rightsquigarrow w, (\Omega'_V, n) \cdot w, (\Omega'_V, j) \\
\text{(E-T-speculate-rollback-stuck)} \\
\hline
\vdash \Phi_V : \perp \quad \Phi_V = w, (\Omega_V, \omega) \\
\hline
\Phi_V \rightsquigarrow^{\text{rlb}} w, \emptyset \\
\text{(E-T-speculate-if-att)} \\
\hline
\Omega_V \xrightarrow{\alpha} \Omega'_V \quad \Omega_V \equiv C, H_V, \bar{B}_V \cdot B_V \triangleright (s; s')_{\bar{f}, f} \quad s \equiv \text{if } e \text{ then } s'' \text{ else } s''' \\
C \equiv \bar{F}; \bar{I} \quad f \in \bar{I} \\
\hline
w, (\Omega_V, n+1) \rightsquigarrow w, (\Omega'_V, n)
\end{array}$$

Note that our attacker essentially cannot speculate. This is because speculation would only trigger execution of the ‘other’ branch. But recall that our attackers are universally quantified. It is therefore needless to have an attacker speculate and do something while the same behaviour is anyway considered by another attacker.

If we allowed the attacker to speculate, we would have a problem: the pc taint would be \mathbf{U} and if the attacker called us, our actions would be \mathbf{U} . However, this would be an overapproximation of our taint-tracking: as already said, those actions are perfectly valid since other attackers will make our code do that without speculation. Thus, if we were to allow the attacker to speculate, we should not raise the pc to \mathbf{U} there, but keep it \mathbf{S} .

$$\boxed{\overline{\Phi}_t \rightsquigarrow \overline{\Phi}'_t}$$

$$\frac{\text{(T-T-speculate-stack)} \quad \Phi_t \rightsquigarrow \Phi'_t}{\overline{\Phi_t} \cdot \Phi_t \rightsquigarrow \overline{\Phi_t} \cdot \Phi'_t}$$

$$\boxed{\Phi_t \rightsquigarrow \Phi'_t}$$

$$\frac{\text{(T-T-speculate-epsilon)} \quad \sigma; \Omega_t \xrightarrow{\epsilon} \Omega'_t \quad \Omega_t \equiv C, H_t, \overline{B} \triangleright s; s' \quad s \neq \text{ifz_then_else_}}{\mathbf{w}, (\Omega_t, \sigma) \rightsquigarrow \mathbf{w}, (\Omega'_t, \sigma)}$$

$$\frac{\text{(T-T-speculate-action)} \quad \sigma; \Omega_t \xrightarrow{\sigma'} \Omega'_t \quad \Omega_t \equiv C, H_t, \overline{B} \triangleright s; s' \quad s \neq \text{ifz_then_else_ and } s \neq \text{lfence}}{\mathbf{w}, (\Omega_t, \sigma) \rightsquigarrow \mathbf{w}, (\Omega'_t, \sigma)}$$

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

(T-T-speculate-rollback)

$$\mathbf{w}, (\Omega_t, \sigma) \rightsquigarrow \mathbf{w}, \emptyset$$

$$\frac{\text{(T-T-speculate-if-attacker)} \quad \sigma'; \Omega_t \xrightarrow{\sigma} \Omega'_t \quad \Omega_t \equiv C, H_t, \overline{B} \cdot B \triangleright (s; s')_{\overline{f}, f} \quad s \equiv \text{if } e \text{ then } s'' \text{ else } s'''}{\begin{array}{l} C \equiv \overline{f}; \overline{I} \quad f \in \overline{I} \end{array}}{\mathbf{w}, (\Omega_t, \sigma') \rightsquigarrow \mathbf{w}, (\Omega'_t, \sigma')}$$

$$\boxed{\Sigma \rightsquigarrow \Sigma'}$$

$$\frac{\text{(Combine-T)} \quad \begin{array}{l} \Sigma = \overline{\Phi} \quad \Sigma' = \overline{\Phi'} \\ \overline{\Phi_v} + \overline{\Phi_t} = \overline{\Phi} \quad \overline{\Phi'_v} + \overline{\Phi'_t} = \overline{\Phi'} \\ \overline{\Phi_v} \rightsquigarrow \overline{\Phi'_v} \quad \overline{\Phi_t} \rightsquigarrow \overline{\Phi'_t} \end{array}}{\Sigma \rightsquigarrow \Sigma'}$$

We allow speculation only when it happens inside the component, not in the attacker (Rule E-T-speculate-if-att), for this reason. Suppose a context speculates and call us with a parameter \mathbf{v} that it loaded speculatively. This is not a concern because we quantify over all attackers, so there exist also the attacker that would call us with \mathbf{v} without speculation. This is the reason why the semantics simplifies the situation and does not let the context speculate.

If we allowed speculation in the context, we would also have to allow the context to possibly access the private memory. This would be the case for a 'reverse' spectre attack, suppose the standard vulnerable snippet is in the context and that compiled code calls it with an out-of-bound parameter. The context could effectively access the memory of the component. Now, we do not model this for a simple reason: this attack can not be defended against if we link with things in the same address space. If we allow to link with things in different address spaces, then this attack would not be possible anymore. Since this is not possible, and since this is the only interesting attack that the context can mount, we do not let the context speculate.

We keep track of only those actions that occur inside the component or between component and context, not of those that happen inside the context (Rule E-T-single).

Note that the rule for rollback is nondeterministic: this works because the + of value and taint states is only valid if its sub-stacks have the same cardinality. Since the only operation that eliminates from the stack is a rollback, the last two rules can only be used in conjunction with their corresponding rule on value states.

$$\boxed{\Sigma \rightsquigarrow \Sigma'}$$

$$\begin{array}{c}
\text{(E-T-single)} \\
\frac{\Sigma \xrightarrow{\alpha^\sigma} \Sigma'}{\Sigma = (w, (\Omega, m, \sigma) \cdot (\bar{F}, \bar{I}, H, \bar{B}) \triangleright (s)_{\bar{f}, f}, n, \sigma'))} \\
\Sigma' = (w, (\Omega, m, \sigma) \cdot (\bar{F}, \bar{I}, H', \bar{B}') \triangleright (s')_{\bar{f}, f'}, n', \sigma'')) \\
\text{if } f == f' \text{ and } f \in I \text{ then } \bar{\lambda}^\sigma = \epsilon \text{ else } \bar{\lambda}^\sigma = \alpha^\sigma \\
\text{if } \alpha^\sigma == \text{rlb}^S \text{ then } j = n \text{ else } j = n + 1 \\
\hline
(n, \Sigma) \xrightarrow{\bar{\lambda}^\sigma} (j, \Sigma')
\end{array}
\quad
\begin{array}{c}
\text{(E-T-silent)} \\
\frac{(n, \Sigma) \xrightarrow{\bar{\lambda}_1^\sigma} (n', \Sigma'') \quad \Sigma'' \xrightarrow{\epsilon} \Sigma'}{(n, \Sigma) \xrightarrow{\bar{\lambda}_1^\sigma} (n' + 1, \Sigma')}
\end{array}
\quad
\begin{array}{c}
\text{(E-T-init)} \\
\frac{}{(n, \Sigma) \xrightarrow{\epsilon} (n, \Sigma)}
\end{array}$$

Helpers

$$\begin{array}{c}
\text{(T-Initial State)} \\
H_0 = H'' \cup H \cup H' \\
H' = \{n \mapsto 0 : S \mid n \in \mathbb{N} \setminus \text{dom}(H)\} \\
H'' = \{-n \mapsto 0 : U \mid n \in \mathbb{N}, -n \notin \text{dom}(H)\} \\
\Sigma \equiv w, (\bar{F}; \bar{I}; H_0; \emptyset \cdot x \mapsto 0 \triangleright \text{call main } x; \text{skip}, \perp, S) \\
\hline
\Omega_0((H; \bar{F}; \bar{I})) = (0, \Sigma)
\end{array}
\quad
\begin{array}{c}
\text{(E-T-trace)} \\
\frac{\exists \Sigma. \Phi_v + \Phi_t = \Sigma \text{ and } \vdash \Phi_v : \perp_f \quad \Omega_0(P) \xrightarrow{\bar{\lambda}^\sigma} (_, \Sigma)}{P \rightsquigarrow \bar{\lambda}^\sigma \downarrow}
\end{array}$$

$$\begin{array}{c}
\text{(E-T-behaviour)} \\
\frac{}{\text{Beh}(P) = \left\{ \bar{\lambda}^\sigma \mid P \rightsquigarrow \bar{\lambda}^\sigma \right\}}
\end{array}
\quad
\begin{array}{c}
\text{(T-Terminal State)} \\
\frac{\Phi_v = w, (\Omega_v, \omega) \cdot (\Omega_v, \omega) \quad \nexists \Omega'_v, \lambda. \Omega_v \xrightarrow{\lambda} \Omega'_v}{\vdash \Phi_v : \perp}
\end{array}
\quad
\begin{array}{c}
\text{(T-Terminal Ending State)} \\
\frac{\Phi_v = w, (\Omega_v, \perp) \quad \vdash \Phi_v : \perp}{\vdash \Phi_v : \perp_f}
\end{array}$$

The speculative semantics starts with the pc tag as safe (S, Rule T-Initial State). Any misspeculation sets the pc tag as unsafe (U, Rule E-T-speculate-if-att). When a speculation is rolled back, the pc returns to be the one set previously, so when all speculation is rolled back, the pc will return to be S, otherwise it'll be U. Roll backs (Rule E-T-speculate-rollback) are triggered by the window reaching 0. Rules Rule E-T-speculate-epsilon and Rule E-T-speculate-action decrement the window, whereas **lfence** sets the remaining window to 0 (Rule E-T-speculate-lfence).

When an action is generated (Rule E-T-speculate-action) it is tagged with the label resulting of the glb (\sqcap) of the label of the action and the label of the pc, thus:

- a safe action-generating expression (S) done while not speculating (S) generates a safe action $S \sqcap S = S$.
- an unsafe action-generating expression (U) done while not speculating (S) generates a safe action $U \sqcap S = S$.
- a safe action-generating expression (S) done while speculating (U) generates a safe action $S \sqcap U = S$.
- an unsafe action-generating expression (U) done while speculating (U) generates an unsafe action $U \sqcap U = U$.

E.3 Alternative Modelling of Attacker Speculation

We could have let the attacker always speculate. Then we'd need to change how we keep track of the pc taint. In fact, all actions done during the speculation started by an attacker can also be done by another attacker (whose code simply has the if guards negated). Thus, those actions must not be considered unsafe, and thus the pc taint is left S (else branch on line 2).

$$\begin{array}{c}
\text{(E-T-speculate-if-alt)} \\
\frac{\Omega_v \xrightarrow{\alpha} \Omega'_v \quad \Omega_v \equiv C, H_v, \bar{B}_v \cdot B_v \triangleright (s; s')_{\bar{f}, f} \quad s \equiv \text{if } e \text{ then } s'' \text{ else } s''' \\
\text{if } B_v \triangleright e \downarrow 0 \text{ then } \Omega'_v \equiv C, H_v, \bar{B}_v \cdot B_v \triangleright s'''; s' \\
\text{if } B_v \triangleright e \downarrow n \text{ and } n > 0 \text{ then } \Omega'_v \equiv C, H_v, \bar{B}_v \cdot B_v \triangleright s''; s' \\
j = \min(w, n)}{w, (\Omega_v, n + 1) \xrightarrow{\alpha} w, (\Omega'_v, n) \cdot w, (\Omega'_v, j)}
\end{array}$$

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

F L⁻: A LANGUAGE WITH WEAK TAINT-TRACKING

This language considers a different read label, which is manifested in the read semantics rule, as well as a different form of taint tracking.

$$\begin{array}{c}
 \text{Heap\&Pc Act.s } \delta ::= \dots \mid (\text{read}(n \mapsto v)) \\
 \\
 \frac{
 \begin{array}{c}
 \text{(E-L-read)} \\
 B_v \triangleright e \downarrow n \quad H_v = H_{v_1}; |n| \mapsto v; H_{v_2}
 \end{array}
 }{
 C, H_v, \overline{B}_v \cdot B_v \triangleright \text{let } x = \text{rd e in } s \xrightarrow{\text{read}(|n| \mapsto v)} C, H_v, \overline{B}_v \cdot B_v \cup x \mapsto v \triangleright s
 } \\
 \frac{
 \begin{array}{c}
 \text{(T-L-read-prv-weak)} \\
 B \triangleright e \downarrow n : \sigma' \quad n_a = -|n| \quad H_t(n_a) = \sigma'' \quad \sigma = \sigma'' \sqcup \sigma'
 \end{array}
 }{
 \sigma_{pc}; C, H_t, \overline{B} \cdot B \triangleright \text{let } x = \text{rd}_{pr} e \text{ in } s \xrightarrow{\sigma \sqcap \sigma_{pc}} C, H_t, \overline{B} \cdot B \cup x \mapsto 0 : \sigma' \sqcap \sigma_{pc} \triangleright s
 }
 \end{array}$$

G T⁻: A LANGUAGE WITH WEAK LEAKS

This language is obtained by adopting the changes made for L⁻ in T.

H EXAMPLES

In this section we write the classical spectre-susceptible program in both **L** and **T** and see what kind of semantics it yields.

Example H.1 (The classical Spectre V1 attack). We have this pseudocode:

```

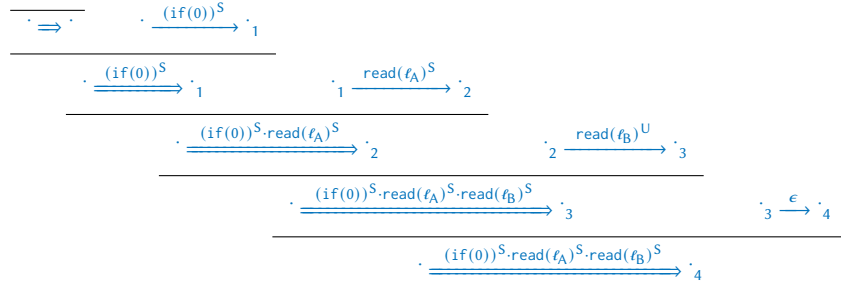
1  if (y < size) {
2  temp = B[A[y] * 512]
3  }

```

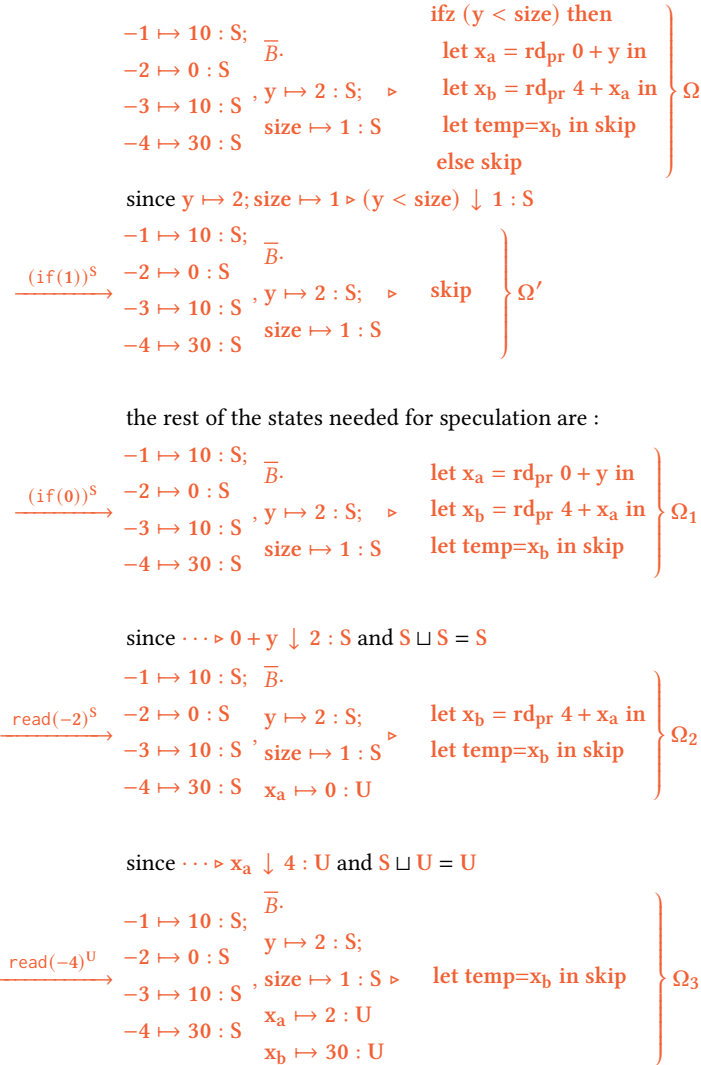
The program checks whether the index stored in the variable y is less than the size of the array A , stored in the variable $size$. If that is the case, the program retrieves $A[y]$, amplifies it with a multiple (here: 512) of the cache line size, and uses the result as an address for accessing the array B . The memory accesses in line 2 may be executed even if $y \geq size$. However, the speculatively executed memory accesses leave a footprint in the microarchitectural state, in particular in the cache, which enables an adversary to retrieve $A[y]$, even for $y \geq size$, by probing the array B .

$$\begin{array}{l}
 \left. \begin{array}{l}
 -1 \mapsto 0 : S; \\
 -2 \mapsto 10 : S \quad \bar{B}. \\
 -3 \mapsto 10 : S, y \mapsto 0 : S; \triangleright \\
 -4 \mapsto 30 : S \quad size \mapsto 3 : S
 \end{array} \right\} \begin{array}{l}
 \text{ifz } (y < size) \text{ then} \\
 \text{let } x_a = rd_{pr} \ 1 + y \text{ in} \\
 \text{let } x_b = rd_{pr} \ 4 + x_a \text{ in} \\
 \text{let temp} = x_b \text{ in skip} \\
 \text{else skip}
 \end{array} \\
 \\
 \text{Rule E-L-if-true} \\
 \text{since } y \mapsto 0; size \mapsto 1 \triangleright (y < size) \downarrow 0 : S \\
 \xrightarrow{(\text{if}(0))^S} \left. \begin{array}{l}
 -1 \mapsto 0 : S; \\
 -2 \mapsto 10 : S \quad \bar{B}. \\
 -3 \mapsto 10 : S, y \mapsto 0 : S; \triangleright \\
 -4 \mapsto 30 : S \quad size \mapsto 3 : S
 \end{array} \right\} \begin{array}{l}
 \text{let } x_a = rd_{pr} \ 1 + y \text{ in} \\
 \text{let } x_b = rd_{pr} \ 4 + x_a \text{ in} \\
 \text{let temp} = x_b \text{ in skip}
 \end{array} \quad \cdot_1 \\
 \\
 \text{Rule E-L-read} \\
 \text{since } \dots \triangleright 1 + y \downarrow 1 : S \text{ and } S \sqcup S = S \\
 \xrightarrow{\text{read}(-1)^S} \left. \begin{array}{l}
 -1 \mapsto 0 : S; \quad \bar{B}. \\
 -2 \mapsto 10 : S \quad y \mapsto 0 : S; \\
 -3 \mapsto 10 : S, \quad size \mapsto 3 : S \triangleright \\
 -4 \mapsto 30 : S \quad x_a \mapsto 0 : U
 \end{array} \right\} \begin{array}{l}
 \text{let } x_b = rd_{pr} \ 4 + x_a \text{ in} \\
 \text{let temp} = x_b \text{ in skip}
 \end{array} \quad \cdot_2 \\
 \\
 \text{Rule E-L-read} \\
 \text{since } \dots \triangleright x_a \downarrow 4 : U \text{ and } S \sqcup U = U \\
 \xrightarrow{\text{read}(-4)^U} \left. \begin{array}{l}
 -1 \mapsto 0 : S; \quad \bar{B}. \\
 -2 \mapsto 10 : S; \quad y \mapsto 0 : S; \\
 -3 \mapsto 10 : S, \quad size \mapsto 3 : S \triangleright \\
 -4 \mapsto 30 : S \quad x_a \mapsto 0 : U \\
 \quad \quad \quad x_b \mapsto 30 : U
 \end{array} \right\} \begin{array}{l}
 \text{let temp} = x_b \text{ in skip}
 \end{array} \quad \cdot_3 \\
 \\
 \text{Rule E-L-letin} \\
 \xrightarrow{\epsilon} \left. \begin{array}{l}
 \bar{B}. \\
 -1 \mapsto 0 : S; \quad y \mapsto 0 : S; \\
 -2 \mapsto 10 : S \quad size \mapsto 3 : S \\
 -3 \mapsto 10 : S, \quad x_a \mapsto 0 : U \\
 -4 \mapsto 30 : S \quad x_b \mapsto 30 : U \\
 \quad \quad \quad temp \mapsto 30 : U
 \end{array} \right\} \triangleright \text{skip} \quad \cdot_4
 \end{array}$$

This is not going to be a problem for **L**, when calculating the trace semantics, all actions are then turned into safe since there is no speculation in **L**. Thus, this program performs the following action:



If we consider the same execution in **T**, however, something different happens when considering the trace semantics. Each individual action is generated as before, but the “if-then-else” will trigger a speculation, which raises the pc tag to **U**. We start from a different state with $y \mapsto 2$.



$$\xrightarrow{\epsilon} \left. \begin{array}{l} \bar{B}. \\ -1 \mapsto 10 : S; \quad y \mapsto 2 : S; \\ -2 \mapsto 0 : S \quad \text{size} \mapsto 1 : S \\ -3 \mapsto 10 : S \quad x_a \mapsto 4 : U \\ -4 \mapsto 30 : S \quad x_b \mapsto 30 : U \\ \quad \text{temp} \mapsto 30 : U \end{array} \right\} \triangleright \text{skip} \quad \Omega_4$$

We now take a look at the speculating semantics for this program. We assume we are not already speculating and that Ω is our starting state. For simplicity, we consider a speculation window of 4 steps.

Given these states, we have the following reductions

$$\begin{aligned} \Sigma &= (4, \emptyset \cdot (\Omega, 4, S)) \\ \Sigma_1 &= (4, \emptyset \cdot (\Omega', 3, S) \cdot (\Omega_1, 3, U)) \\ \Sigma_2 &= (4, \emptyset \cdot (\Omega', 3, S) \cdot (\Omega_2, 2, U)) \\ \Sigma_3 &= (4, \emptyset \cdot (\Omega', 3, S) \cdot (\Omega_3, 1, U)) \\ \Sigma_4 &= (4, \emptyset \cdot (\Omega', 3, S) \cdot (\Omega_4, 0, U)) \\ \Sigma' &= (4, \emptyset \cdot (\Omega', 3, S)) \end{aligned}$$

$$\begin{aligned} \underbrace{(4, \emptyset \cdot (\Omega, 4, S))}_{\Sigma} &\xrightarrow{(\text{if}(0))^S} \underbrace{(4, \emptyset \cdot (\Omega', 3, S) \cdot (\Omega_1, 3, U))}_{\Sigma_1} \text{ by Rule E-T-speculate-if-att} \\ \underbrace{(4, \emptyset \cdot (\Omega', 3, S) \cdot (\Omega_1, 3, U))}_{\Sigma_1} &\xrightarrow{(\text{read}(\ell_A))^S} \underbrace{(4, \emptyset \cdot (\Omega', 3, S) \cdot (\Omega_2, 2, U))}_{\Sigma_2} \text{ by Rule E-T-speculate-action} \\ \underbrace{(4, \emptyset \cdot (\Omega', 3, S) \cdot (\Omega_2, 2, U))}_{\Sigma_2} &\xrightarrow{(\text{read}(\ell_B))^U} \underbrace{(4, \emptyset \cdot (\Omega', 3, S) \cdot (\Omega_3, 1, U))}_{\Sigma_3} \text{ by Rule E-T-speculate-action} \\ \underbrace{(4, \emptyset \cdot (\Omega', 3, S) \cdot (\Omega_3, 1, U))}_{\Sigma_3} &\rightsquigarrow \underbrace{(4, \emptyset \cdot (\Omega', 3, S) \cdot (\Omega_4, 0, U))}_{\Sigma_4} \text{ by Rule E-T-speculate-epsilon} \\ \underbrace{(4, \emptyset \cdot (\Omega', 3, S) \cdot (\Omega_4, 0, U))}_{\Sigma_4} &\xrightarrow{\text{rlb}} \underbrace{(4, \emptyset \cdot (\Omega', 3, S))}_{\Sigma'} \text{ by Rule E-T-speculate-rollback} \end{aligned}$$

The crucial reduction is the third one, where the label is tagged as **U** since the pc tag is **U** and the label itself is **U**. The second reduction is tagged **S** since the action itself is **S** and so it is ok to perform it even under speculation.

For simplicity, we omit the **n** parameter in the $(n, \Sigma) \Rightarrow (n', \Sigma')$ reductions.

$$\begin{aligned} \Sigma &\Rightarrow \Sigma \quad \Sigma \xrightarrow{(\text{if}(0))^S} \Sigma_1 \\ \Sigma &\xrightarrow{(\text{if}(0))^S} \Sigma_1 \quad \Sigma_1 \xrightarrow{(\text{read}(\ell_A))^S} \Sigma_2 \\ \Sigma &\xrightarrow{(\text{if}(0))^S \cdot (\text{read}(\ell_A))^S} \Sigma_2 \quad \Sigma_2 \xrightarrow{(\text{read}(\ell_B))^U} \Sigma_3 \\ \Sigma &\xrightarrow{(\text{if}(0))^S \cdot (\text{read}(\ell_A))^S \cdot (\text{read}(\ell_B))^U} \Sigma_3 \quad \Sigma_3 \rightsquigarrow \Sigma_4 \\ \Sigma &\xrightarrow{(\text{if}(0))^S \cdot (\text{read}(\ell_A))^S \cdot (\text{read}(\ell_B))^U} \Sigma_4 \quad \Sigma_4 \xrightarrow{\text{rlb}} \Sigma' \\ \Sigma &\xrightarrow{(\text{if}(0))^S \cdot (\text{read}(\ell_A))^S \cdot (\text{read}(\ell_B))^U \cdot \text{rlb}} \Sigma' \end{aligned}$$

□

Example H.2 (Spectre V1 with implicit flow). We have this pseudocode:

```

1  y = A[x]
2  if (y < size) {
3    temp = B[y * 512]
4  }

```

This is analogous to the code above, but the leak is implicit.

We could support the thesis that this is not a leak and to do so our semantics would turn the taint of the private heap to U only when speculating. We do not share that idea so we do not do it. □

H.1 Additional Snippets

```

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

```

Listing 6: The classic Spectre v1 snippet.

```

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

```

Listing 7: Code that is RSNI but not RSS.

```

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

```

Listing 8: A variant of the classic Spectre v1 snippet (Example 10 from [36]).

```

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

```

Listing 9: Another variant of the classic Spectre v1 snippet.

I FORMALISATION OF CODE SNIPPETS

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

I.1 Snippet of Listing 6

```
get(y)  $\mapsto$  let size = rd 1 in ifz y < size
  then let x = rdpr nA + y in let temp = rd nB + x in skip
  else skip
```

I.2 Snippet of Listing 7

```
get(y)  $\mapsto$  let size = rd 1 in ifz y < size
  then let x = rdpr nA + y in let temp = rd nB + x in skip
  else let x = rdpr nA + y in let temp = rd nB + x in skip
```

I.3 Snippet of Listing 10

```
get(y)  $\mapsto$  let size = rd 1 in ifz y < size
  then lfence; let x = rdpr nA + y in
    let temp = rd nB + x in skip    else skip
```

I.4 Snippet of Listing 8

```
get(y)  $\mapsto$  let size = rd 1 in ifz y < size
  then let x = rdpr nA + y in ifz x == 0
    then let temp = rd nB + 0 in skip
    else skip    else skip
```

I.5 Snippet of Listing 11

```
get(y)  $\mapsto$ 
  let size = rd 1 in ifz y < size
  then let x = rdpr nA + y in ifz x == 0
    then lfence; let temp = rd nB + 0 in skip
    else skip    else skip
```

I.6 Snippet of Listing 12

Here we are saving the predicate bit in location -1 so source n_a is stored at $n_a - 1$, which is the address that gets calculated in the reads.

```
get(y)  $\mapsto$ 
  let size = rd 1 in let xg=y < size in ifz xg
  then let x = rdpr -1 in -1 :=pr x  $\vee$   $\neg$ xg;
    let x = rdpr na + y + 1 in let pr = rdpr -1 in
    let x = 0 (if pr) in let temp = rd nb + x in skip
  else let x = rdpr -1 in -1 :=pr x  $\vee$  xg; skip
```

I.7 Snippet of Listing 9

```
get(y)  $\mapsto$  let size = rd 1 in let x = rdpr na + y in ifz y < size
  then let temp = rd nB + x in skip    else skip
```

I.8 Snippet of Listing 13

```

get(y)  $\mapsto$ 
  let size = rd 1 in let x = rdpr na + y + 1 in
  let pr = rdpr -1 in let x = 0 (if pr) in ifz y < size
  then let xf = rdpr -1 in -1 :=pr xf  $\vee$   $\neg$ xg;
  let temp = rd nb + x in skip
  else let xf = rdpr -1 in -1 :=pr xf  $\vee$  xg; skip

```

I.9 Snippet of Listing 14

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

```

get(y)  $\mapsto$  let size = rd 1 in let x = rdpr na + y in ifz y < size
  then call get2 x else skip
get2(x)  $\mapsto$  let temp = rd nB + x in skip
get(y)  $\mapsto$ 
  let pr=1 in let size = rd 1 in let x = rdpr na + y in
  let xg=y < size in ifz xg
  then let pr=pr  $\vee$   $\neg$ xg in call get2 x
  else let pr=pr  $\vee$  xg in skip
get2(x)  $\mapsto$  let pr=1 in let temp = rd n + b + x in skip

```

J SPECULATIVE SAFETY

We need to define some helpers first.

A heap is valid if and only if its private part contains unsafe values. We do not enforce this on the public part too because a program may write a private value in the public heap.

Definition J.1 (Valid Heap).

$$\vdash H : \text{vld} \stackrel{\text{def}}{=} \forall n \mapsto v : \sigma \in H, \text{ if } n < 0 \text{ then } \sigma = U$$

An attacker is one that has no private heap nor instructions to manipulate it directly.

Definition J.2 (Attacker).

$$\begin{aligned} \vdash A : \text{atk} \stackrel{\text{def}}{=} A \equiv H; \bar{F} \text{ and } \forall n \mapsto v : \sigma \in H, n \geq 0 \\ \text{and } \forall f(x) \mapsto s; \text{return}; \in \bar{F}, \text{ let } x = \text{rd}_{pr} \ e \text{ in } s', e :=_{pr} \ e \notin s \end{aligned}$$

A whole program of some language L is speculatively safe (SS(L)) if all its actions are safe.

Definition J.3 (Speculative Safety (SS(L))).

$$\vdash P : \text{SS}(L) \stackrel{\text{def}}{=} \forall \lambda \bar{\sigma} \in \text{Beh}(P). \forall \alpha^\sigma \in \bar{\lambda}^\sigma. \sigma \equiv S$$

A component is robustly speculatively safe (RSS(L)) if it is safe no matter what attacker it is linked against.

Definition J.4 (Robust Speculative Safety (RSS(L))).

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

If we consider the code of Example H.1, we see that the **L** execution only produces **S** actions. This is intuitive, since there is no speculation in **L**.

Still in Example H.1, the **T** execution can produce **U** actions, precisely when a memory access is done speculatively and based on other memory-accessed data.

K REVIEW: SPECULATIVE NON-INTERFERENCE

We first define what it means for heaps and programs to be low equivalent.

Definition K.1 (Low-equivalence for heaps and programs).

$$\begin{aligned}
H =_{\perp} H' &\stackrel{\text{def}}{=} \vdash H : \text{vld} \wedge \vdash H' : \text{vld} \wedge \text{dom}(H) = \text{dom}(H') \wedge \\
&\quad \forall n \mapsto v : \sigma \in H, \text{ if } n \geq 0 \text{ then } n \mapsto v : \sigma \in H' \\
&\quad \text{if } n < 0 \text{ then } \exists v'. n \mapsto v' : \sigma \in H' \\
P =_{\perp} P' &\stackrel{\text{def}}{=} P \equiv H; \bar{F}; \bar{I} \text{ and } P' \equiv H'; \bar{F}; \bar{I} \text{ and } H =_{\perp} H'
\end{aligned}$$

We can define the non-speculative projection of a trace:

$$\begin{aligned}
\bar{\lambda} \upharpoonright_{nse} &= (\bar{\lambda}, 0) \upharpoonright_{nse} \\
(\emptyset, n) \upharpoonright_{nse} &= \emptyset \\
(\alpha^\sigma \cdot \bar{\lambda}, 0) \upharpoonright_{nse} &= \alpha^\sigma \cdot (\bar{\lambda}, 0) \upharpoonright_{nse} \\
(\delta^\sigma \cdot \bar{\lambda}, 0) \upharpoonright_{nse} &= \delta^\sigma \cdot (\bar{\lambda}, 0) \upharpoonright_{nse} && \text{where } \delta^\sigma \neq (\text{if}(v))^\sigma \\
((\text{if}(v))^\sigma \cdot \bar{\lambda}, 0) \upharpoonright_{nse} &= (\text{if}(v))^\sigma \cdot (\bar{\lambda}, 1) \upharpoonright_{nse} \\
(\alpha^\sigma \cdot \bar{\lambda}, n+1) \upharpoonright_{nse} &= (\bar{\lambda}, 0) \upharpoonright_{nse} \\
(\delta^\sigma \cdot \bar{\lambda}, n+1) \upharpoonright_{nse} &= (\bar{\lambda}, n+1) \upharpoonright_{nse} && \text{where } \delta^\sigma \neq (\text{if}(v))^\sigma \\
((\text{if}(v))^\sigma \cdot \bar{\lambda}, n+1) \upharpoonright_{nse} &= (\bar{\lambda}, n+2) \upharpoonright_{nse} \\
((\text{r1b})^\sigma \cdot \bar{\lambda}, n+1) \upharpoonright_{nse} &= (\bar{\lambda}, n) \upharpoonright_{nse}
\end{aligned}$$

A program of a language L is SNI if, taking any other program that is low-equivalent to itself, if their non-speculative traces are non-interferent, then their speculative traces must also be non-interferent.

Definition K.2 (Speculative Non-interference).

$$\begin{aligned}
\vdash P : \text{SNI}(L) &\stackrel{\text{def}}{=} \forall P' =_{\perp} P, \forall \bar{\lambda}_1 \in \text{Beh}(\Omega_0(P)), \bar{\lambda}_2 \in \text{Beh}(\Omega_0(P')) \\
&\quad \text{if } \bar{\lambda}_1 \upharpoonright_{nse} = \bar{\lambda}_2 \upharpoonright_{nse} \text{ then } \bar{\lambda}_1 = \bar{\lambda}_2
\end{aligned}$$

A component is robustly SNI if it is SNI for any attacker it links against.

Definition K.3 (Robust Speculative Non-Interference).

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

L SECURITY DEFINITIONS AND THEIR IMPLICATIONS

Here, we show the relationship between speculative safety and speculative non-interference. We prove these relationships only for programs in **T**, since both definitions are trivially satisfied in **L** (this immediately follows from **L** not having any speculative behavior; see Theorem L.1).

THEOREM L.1 (SS AND SNI HOLD FOR ALL SOURCE PROGRAMS).

$$\begin{aligned} \forall P \in \mathbf{L}. \vdash P : SS(\mathbf{L}) \text{ and } \vdash P : SNI(\mathbf{L}) \\ \forall P \in \mathbf{L}^-. \vdash P : SS(\mathbf{L}^-) \text{ and } \vdash P : SNI(\mathbf{L}^-) \end{aligned}$$

PROOF. This trivially holds from (1) programs in **L** and **L**⁻ produce only actions labelled with **S** (for SS(**L**) and SS(**L**⁻)), and (2) traces are identical to their non-speculative projection for programs **L** and **L**⁻ (for SNI(**L**) and SNI(**L**⁻)). \square

L.1 SS implies SNI

L.1.1 Strong Variants.

THEOREM L.2 (SS IMPLIES SNI (STRONG)).

$$\forall P \in \mathbf{T}. \text{ if } \vdash P : SS(\mathbf{T}), \text{ then } \vdash P : SNI(\mathbf{T})$$

PROOF. Let **P** be an arbitrary program in **T** such that $\vdash P : SS(\mathbf{T})$. Assume, for contradiction's sake, that $\vdash P : SNI(\mathbf{T})$ does not hold. That is, there is another program **P'** and traces $\bar{\lambda}_1 \in \text{Beh}(\Omega_0(\mathbf{P}))$, $\bar{\lambda}_2 \in \text{Beh}(\Omega_0(\mathbf{P}'))$ such that $\mathbf{P} =_{\mathbf{L}} \mathbf{P}'$, $\bar{\lambda}_1 \upharpoonright_{nse} = \bar{\lambda}_2 \upharpoonright_{nse}$, and $\bar{\lambda}_1 \neq \bar{\lambda}_2$. By unrolling the **T**-Terminal State rule, we have that $\mathbf{P} \rightsquigarrow \bar{\lambda}_1$ and $\mathbf{P}' \rightsquigarrow \bar{\lambda}_2$. By unrolling the rule **E**-T-trace, we have that there are $\Sigma, \Sigma', \mathbf{n}, \mathbf{n}'$ such that $\vdash \Sigma : \perp, \vdash \Sigma' : \perp$, $(0, \Omega_0(\mathbf{P})) \xrightarrow{\bar{\lambda}_1} (\mathbf{n}, \Sigma)$, and $(0, \Omega_0(\mathbf{P}')) \xrightarrow{\bar{\lambda}_2} (\mathbf{n}', \Sigma')$. From $\mathbf{P} =_{\mathbf{L}} \mathbf{P}'$ and lemma L.3, we get $(0, \Omega_0(\mathbf{P})) \approx (0, \Omega_0(\mathbf{P}'))$. There are two cases:

n = n': Then, we have $\vdash \Sigma_0 : \text{safe}$ and from $\vdash P : SS(\mathbf{T})$, we get that $\vdash \bar{\lambda}_1 : \text{safe}$. From $(0, \Omega_0(\mathbf{P})) \approx (0, \Omega_0(\mathbf{P}'))$, $(0, \Omega_0(\mathbf{P})) \xrightarrow{\bar{\lambda}_1} (\mathbf{n}, \Sigma)$,

$(0, \Omega_0(\mathbf{P}')) \xrightarrow{\bar{\lambda}_2} (\mathbf{n}', \Sigma')$, and lemma L.6, we get that $\bar{\lambda}_1 \upharpoonright_{nse} <> \bar{\lambda}_2 \upharpoonright_{nse} \vee \bar{\lambda}_1 = \bar{\lambda}_2$. From this and $\bar{\lambda}_1 \upharpoonright_{nse} = \bar{\lambda}_2 \upharpoonright_{nse}$, we get $\bar{\lambda}_1 = \bar{\lambda}_2$ leading to a contradiction.

n ≠ n': Let **m** be the maximum value such that $(0, \Omega_0(\mathbf{P})) \xrightarrow{\lambda} (\mathbf{m}, \Sigma_1)$ and $(0, \Omega_0(\mathbf{P}')) \xrightarrow{\lambda} (\mathbf{m}, \Sigma'_1)$. Observe that (1) such an **m** always exists, and (2) λ is a prefix to both $\vdash \bar{\lambda}_1 : \text{safe}$ and $\vdash \bar{\lambda}_2 : \text{safe}$. From $\vdash P : SS(\mathbf{T})$ and λ being a prefix of $\vdash \bar{\lambda}_1 : \text{safe}$, we have $\vdash \lambda : \text{safe}$. From $(0, \Omega_0(\mathbf{P})) \approx (0, \Omega_0(\mathbf{P}'))$, $(0, \Omega_0(\mathbf{P})) \xrightarrow{\lambda} (\mathbf{m}, \Sigma_1)$, $(0, \Omega_0(\mathbf{P}')) \xrightarrow{\lambda} (\mathbf{m}, \Sigma'_1)$, $\vdash \lambda : \text{safe}$, and lemma L.6, we have that $\Sigma_1 \approx \Sigma'_1$. There are three cases:

$\neg \exists \alpha, \sigma, \Sigma_2. (0, \Omega_0(\mathbf{P})) \xrightarrow{\lambda \cdot \alpha^\sigma} (\mathbf{m} + 1, \Sigma_2)$: From this, we get that $\Sigma_1 = \Sigma$ and therefore $\vdash \Sigma : \perp$. From this and $\Sigma_1 \approx \Sigma'_1$, we get that $\vdash \Sigma' : \perp$ holds as well. From this, we get that $\mathbf{m} = \mathbf{n} = \mathbf{n}'$, leading to a contradiction.

$\neg \exists \alpha', \sigma', \Sigma'_2. (0, \Omega_0(\mathbf{P}')) \xrightarrow{\lambda \cdot \alpha'^{\sigma'}} (\mathbf{m} + 1, \Sigma'_2)$: The proof of this case is similar to the case $\neg \exists \alpha, \sigma, \Sigma_2. (0, \Omega_0(\mathbf{P})) \xrightarrow{\lambda \cdot \alpha^\sigma} (\mathbf{m} + 1, \Sigma_2)$.

$\exists \alpha, \sigma, \Sigma_2, \alpha', \sigma', \Sigma'_2. (0, \Omega_0(\mathbf{P})) \xrightarrow{\lambda \cdot \alpha^\sigma} (\mathbf{m} + 1, \Sigma_2) \wedge (0, \Omega_0(\mathbf{P}')) \xrightarrow{\lambda \cdot \alpha'^{\sigma'}} (\mathbf{m} + 1, \Sigma'_2) \wedge \alpha^\sigma \neq \alpha'^{\sigma'}$: Observe that $\lambda \cdot \alpha^\sigma$ is a prefix of $\bar{\lambda}_1$. Therefore, $\vdash \lambda \cdot \alpha^\sigma : \text{safe}$ follows from $\vdash P : SS$. From $(0, \Omega_0(\mathbf{P})) \approx (0, \Omega_0(\mathbf{P}'))$, $(0, \Omega_0(\mathbf{P})) \xrightarrow{\lambda \cdot \alpha^\sigma} (\mathbf{m} + 1, \Sigma_2)$, $(0, \Omega_0(\mathbf{P}')) \xrightarrow{\lambda \cdot \alpha'^{\sigma'}} (\mathbf{m} + 1, \Sigma'_2)$, $\vdash \lambda \cdot \alpha^\sigma : \text{safe}$, and lemma L.6, we have $(\lambda \cdot \alpha^\sigma) \upharpoonright_{nse} <> (\lambda \cdot \alpha'^{\sigma'}) \upharpoonright_{nse} \vee (\Sigma_2 \approx \Sigma'_2 \wedge \lambda \cdot \alpha^\sigma = \lambda \cdot \alpha'^{\sigma'})$. There are two cases:

$(\lambda \cdot \alpha^\sigma) \upharpoonright_{nse} <> (\lambda \cdot \alpha'^{\sigma'}) \upharpoonright_{nse}$: This contradicts $\bar{\lambda}_1 \upharpoonright_{nse} = \bar{\lambda}_2 \upharpoonright_{nse}$ given that $\lambda \cdot \alpha^\sigma$ is a prefix of $\bar{\lambda}_1$ and $\lambda \cdot \alpha'^{\sigma'}$ is a prefix of $\bar{\lambda}_2$.

$(\Sigma_2 \approx \Sigma'_2 \wedge \lambda \cdot \alpha^\sigma = \lambda \cdot \alpha'^{\sigma'})$: This leads to a contradiction with $\alpha^\sigma \neq \alpha'^{\sigma'}$.

Since all cases lead to a contradiction, this completes the proof of our theorem. \square

L.1.2 Low-equivalence and initial states.

LEMMA L.3 (LOW-EQUIVALENT PROGRAMS HAVE LOW-EQUIVALENT INITIAL STATES).

$$\forall P, P'. \text{ if } P =_{\mathbf{L}} P' \text{ then } \Omega_0(P) \approx \Omega_0(P')$$

PROOF. Let $\mathbf{P} = (\mathbf{H}; \bar{\mathbf{F}}; \bar{\mathbf{I}})$ and $\mathbf{P}' = (\mathbf{H}'; \bar{\mathbf{F}}'; \bar{\mathbf{I}}')$ be two arbitrary programs such that $\mathbf{P} =_{\mathbf{L}} \mathbf{P}'$. Then, $\Omega_0(\mathbf{P})$ and $\Omega_0(\mathbf{P}')$ are as follows:

$$\begin{aligned} \mathbf{H}_0 &= \mathbf{H}_1 \cup \mathbf{H} \cup \mathbf{H}_2 \\ \mathbf{H}_1 &= \{\mathbf{n} \mapsto \mathbf{0} : \mathbf{S} \mid \mathbf{n} \in \mathbb{N} \setminus \text{dom}(\mathbf{H})\} \end{aligned}$$

$$\begin{aligned}
H_2 &= \{-n \mapsto 0 : U \mid n \in \mathbb{N}, -n \notin \text{dom}(H)\} \\
\Omega_0(P) &\stackrel{\text{def}}{=} w, (\bar{F}; \bar{I}; H_0; \emptyset \cdot x \mapsto 0 \triangleright \text{call main } x; \text{skip}, \perp, S) \\
H'_0 &= H'_1 \cup H' \cup H'_2 \\
H'_1 &= \{n \mapsto 0 : S \mid n \in \mathbb{N} \setminus \text{dom}(H')\} \\
H'_2 &= \{-n \mapsto 0 : U \mid n \in \mathbb{N}, -n \notin \text{dom}(H')\} \\
\Omega_0(P') &\stackrel{\text{def}}{=} w, (\bar{F}; \bar{I}; H'_0; \emptyset \cdot x \mapsto 0 \triangleright \text{call main } x; \text{skip}, \perp, S)
\end{aligned}$$

Moreover, from $P \equiv_L P'$, we get $H \approx H'$. Therefore, $\Omega_0(P) \approx \Omega_0(P')$ immediately follows. \square

Before continuing with our proof, we define what it means for program states to be safe-equivalent.

Definition L.4 (Safe-equivalence for heaps and program states).

$$\begin{aligned}
&\vdash w, ss_0 \cdot (\Omega, m, \sigma) : \text{unsafe} \stackrel{\text{def}}{=} \sigma = U \\
&\vdash w, (\Omega, m, \sigma) : \text{safe} \stackrel{\text{def}}{=} \sigma = S \\
&\vdash \epsilon : \text{safe} \stackrel{\text{def}}{=} \text{true} \\
&\vdash \alpha^\sigma : \text{safe} \stackrel{\text{def}}{=} \sigma = S \\
&\vdash \bar{\lambda}^\sigma \cdot \alpha^\sigma : \text{safe} \stackrel{\text{def}}{=} \vdash \bar{\lambda}^\sigma : \text{safe} \text{ and } \vdash \alpha^\sigma : \text{safe} \\
&\vdash H(n) : \text{def} \stackrel{\text{def}}{=} \exists v, \sigma. H(n) = v : \sigma \\
&\vdash B(x) : \text{def} \stackrel{\text{def}}{=} \exists v, \sigma. B(x) = v : \sigma \\
&v : \sigma \approx v' : \sigma' \stackrel{\text{def}}{=} \sigma = \sigma' \text{ and if } \sigma = S \text{ then } v = v' \\
&H \approx H' \stackrel{\text{def}}{=} \forall n. \vdash H(n) : \text{def} \text{ iff } \vdash H'(n) : \text{def} \text{ and} \\
&\quad \text{if } \vdash H(n) : \text{def} \text{ then } H(n) \approx H'(n) \\
&B \approx B' \stackrel{\text{def}}{=} \forall x. \vdash B(x) : \text{def} \text{ iff } \vdash B'(x) : \text{def} \text{ and} \\
&\quad \text{if } \vdash B(x) : \text{def} \text{ then } B(x) \approx B'(x) \\
&\bar{B} \cdot B \approx \bar{B}' \cdot B' \stackrel{\text{def}}{=} \bar{B} \approx \bar{B}' \text{ and } B \approx B' \\
&\Omega \approx \Omega' \stackrel{\text{def}}{=} \Omega \equiv C, H, \bar{B} \triangleright s \text{ and } \Omega' \equiv C, H', \bar{B}' \triangleright s \text{ and } H \approx H' \text{ and } \bar{B} \approx \bar{B}' \\
&\emptyset \approx \emptyset \\
&ss \approx ss' \stackrel{\text{def}}{=} ss \equiv ss_0 \cdot (\Omega, m, \sigma) \text{ and } ss' \equiv ss'_0 \cdot (\Omega', m, \sigma) \text{ and } ss_0 \approx ss'_0 \text{ and } \Omega \approx \Omega' \\
&\Sigma \approx \Sigma' \stackrel{\text{def}}{=} \Sigma \equiv (w, ss) \text{ and } \Sigma' \equiv (w, ss') \text{ and } ss \approx ss'
\end{aligned}$$

L.1.3 Transitive-closure semantics \implies .

Definition L.5.

$$\begin{aligned}
&\alpha^\sigma \langle \rangle \alpha'^{\sigma'} \text{ if } \alpha \neq \alpha' \vee \sigma \neq \sigma' \\
&\alpha^\sigma \cdot \lambda \langle \rangle \alpha'^{\sigma'} \cdot \lambda' \text{ if } \alpha^\sigma \langle \rangle \alpha'^{\sigma'} \vee \lambda \langle \rangle \lambda'
\end{aligned}$$

LEMMA L.6 (STEPS OF \implies PRESERVE LOW-EQUIVALENCE OR PRODUCE DISTINCT NON-SPECULATIVE PROJECTIONS).

$$\begin{aligned}
&\forall P, P', \Sigma_0, \Sigma'_0, \Sigma_1, \Sigma'_1, \lambda, \lambda', n. \text{ if } \Sigma_0 = \Omega_0(P), \Sigma'_0 = \Omega_0(P'), \\
&\quad (0, \Sigma_0) \xrightarrow{\lambda} (n, \Sigma_1), (0, \Sigma'_0) \xrightarrow{\lambda'} (n, \Sigma'_1), \\
&\quad \Sigma_0 \approx \Sigma'_0, \vdash \Sigma_0 : \text{safe}, \vdash \lambda : \text{safe}, \\
&\quad \text{then } \lambda \upharpoonright_{nse} \langle \rangle \lambda' \upharpoonright_{nse} \vee (\Sigma_1 \approx \Sigma'_1 \wedge \lambda = \lambda')
\end{aligned}$$

PROOF. Let $P, P', \Sigma_0, \Sigma'_0, \Sigma_1, \Sigma'_1, \lambda, \lambda', n$ be such that $\Sigma_0 = \Omega_0(P), \Sigma'_0 = \Omega_0(P'), (0, \Sigma_0) \xrightarrow{\lambda} (n, \Sigma_1), (0, \Sigma'_0) \xrightarrow{\lambda'} (n, \Sigma'_1), \Sigma_0 \approx \Sigma'_0, \vdash \Sigma_0 : \text{safe}$, and $\vdash \lambda : \text{safe}$. We prove the lemma by induction on n :

Base case: For the base case, we consider $n = 0$. Then, both $(0, \Sigma_0) \xRightarrow{\lambda} (n, \Sigma_1)$ and $(0, \Sigma'_0) \xRightarrow{\lambda'} (n, \Sigma'_1)$ have been derived using the E-T-init rule. Therefore, $\lambda = \epsilon$, $\Sigma_1 = \Sigma_0$, $\lambda' = \epsilon$, and $\Sigma'_1 = \Sigma'_0$. As a result, $\lambda = \lambda'$ follows from $\lambda = \epsilon$ and $\lambda' = \epsilon$, whereas $\Sigma_1 \approx \Sigma'_1$ follows from $\Sigma_1 = \Sigma_0$, $\Sigma'_1 = \Sigma'_0$, and $\Sigma_0 \approx \Sigma'_0$. Therefore, $\lambda \upharpoonright_{nse} \langle \lambda' \upharpoonright_{nse} \vee (\Sigma_1 \approx \Sigma'_1 \wedge \lambda = \lambda') \rangle$ holds for the base case.

Induction step: For the induction step, we assume that the claim holds for all $n' < n$ and we show that it holds for n as well. We proceed by case distinction on the rule used to derive $(0, \Sigma_0) \xRightarrow{\lambda} (n, \Sigma_1)$:

E-T-silent: Then, $(0, \Sigma_0) \xRightarrow{\lambda} (n, \Sigma_1)$ has been derived using the E-T-silent rule. Therefore, we have $(0, \Sigma_0) \xRightarrow{\bar{\lambda}_1^\sigma} (n-1, \Sigma_2)$, $\Sigma_2 \xrightarrow{\epsilon} \Sigma_1$, and $\lambda = \bar{\lambda}_1^\sigma \cdot \epsilon$.

Since $n > 1$, $(0, \Sigma'_0) \xRightarrow{\lambda'} (n, \Sigma'_1)$ has been derived using the E-T-silent or E-T-single rules. Therefore, we have $(0, \Sigma'_0) \xRightarrow{\bar{\lambda}'_1^{\sigma'}}$

and $\Sigma'_2 \xrightarrow{\lambda'} \Sigma'_1$. From the induction hypothesis, there are two cases:

$\bar{\lambda}_1^\sigma \upharpoonright_{nse} \langle \bar{\lambda}'_1^{\sigma'} \upharpoonright_{nse} \rangle$: Observe that $\lambda \upharpoonright_{nse} = \bar{\lambda}_1^\sigma \upharpoonright_{nse}$ and $\lambda' \upharpoonright_{nse}$ is either $\bar{\lambda}'_1^{\sigma'} \upharpoonright_{nse}$ or $\bar{\lambda}'_1^{\sigma'} \upharpoonright_{nse} \cdot \lambda'_1$. From this and $\bar{\lambda}_1^\sigma \upharpoonright_{nse} \langle \bar{\lambda}'_1^{\sigma'} \upharpoonright_{nse} \rangle$, we get $\lambda \upharpoonright_{nse} \langle \lambda' \upharpoonright_{nse} \rangle$.

$\Sigma_2 \approx \Sigma'_2 \wedge \bar{\lambda}_1^\sigma = \bar{\lambda}'_1^{\sigma'}$: From $\Sigma_2 \approx \Sigma'_2$, $\Sigma_2 \xrightarrow{\epsilon} \Sigma_1$, lemma L.8, we get $\Sigma'_2 \xrightarrow{\epsilon} \Sigma''_1$ and $\Sigma_1 \approx \Sigma''_1$. From this and the determinism of \rightsquigarrow , we get that $\lambda'_1 = \epsilon$ and $\Sigma'_1 = \Sigma''_1$. Hence, $\lambda = \lambda'$ follows from $\lambda = \bar{\lambda}_1^\sigma$, $\lambda' = \bar{\lambda}'_1^{\sigma'} \cdot \lambda'_1$, $\bar{\lambda}_1^\sigma = \bar{\lambda}'_1^{\sigma'}$, and $\lambda'_1 = \epsilon$. Moreover, $\Sigma_1 \approx \Sigma'_1$ follows from $\Sigma_1 \approx \Sigma''_1$ and $\Sigma'_1 = \Sigma''_1$.

Therefore, $\lambda \upharpoonright_{nse} \langle \lambda' \upharpoonright_{nse} \vee (\Sigma_1 \approx \Sigma'_1 \wedge \lambda = \lambda') \rangle$ holds for this case.

E-T-single: Then, $(0, \Sigma_0) \xRightarrow{\lambda} (n, \Sigma_1)$ has been derived using the E-T-single rule. Therefore, we have $(0, \Sigma_0) \xRightarrow{\bar{\lambda}_1^{\sigma_1}} (n-1, \Sigma_2)$, $\Sigma_2 \xrightarrow{\alpha^\sigma} \Sigma_1$, $\Sigma_2 = (w, \overline{(\Omega, m, \sigma)} \cdot (\bar{F}, \bar{I}, H_2, B_2 \triangleright (s)_{\bar{F}, F}, n, \sigma'))$, $\Sigma_1 = (w, \overline{(\Omega, m, \sigma)} \cdot (\bar{F}, \bar{I}, H_1, B_1 \triangleright (s')_{\bar{F}, F}, n', \sigma'))$, and $\lambda = \bar{\lambda}_1^\sigma$ if $f == f' \wedge f \in I$ and $\lambda = \bar{\lambda}_1^\sigma \cdot \alpha^\sigma$ otherwise.

Since $n > 1$, $(0, \Sigma'_0) \xRightarrow{\lambda'} (n, \Sigma'_1)$ has been derived using the E-T-silent or E-T-single rules. Therefore, we have $(0, \Sigma'_0) \xRightarrow{\bar{\lambda}'_1^{\sigma'_1}}$

and $\Sigma'_2 \xrightarrow{\lambda'} \Sigma'_1$. From the induction hypothesis, there are two cases:

$\bar{\lambda}_1^{\sigma_1} \upharpoonright_{nse} \langle \bar{\lambda}'_1^{\sigma'_1} \upharpoonright_{nse} \rangle$: Observe that $\lambda \upharpoonright_{nse}$ is either $\bar{\lambda}_1^{\sigma_1} \upharpoonright_{nse}$ or $\bar{\lambda}_1^{\sigma_1} \upharpoonright_{nse} \cdot \alpha^\sigma$ and $\lambda' \upharpoonright_{nse}$ is either $\bar{\lambda}'_1^{\sigma'_1} \upharpoonright_{nse}$ or $\bar{\lambda}'_1^{\sigma'_1} \upharpoonright_{nse} \cdot \lambda'_1$. From this and $\bar{\lambda}_1^{\sigma_1} \upharpoonright_{nse} \langle \bar{\lambda}'_1^{\sigma'_1} \upharpoonright_{nse} \rangle$, we get $\lambda \upharpoonright_{nse} \langle \lambda' \upharpoonright_{nse} \rangle$.

$\Sigma_2 \approx \Sigma'_2 \wedge \bar{\lambda}_1^{\sigma_1} = \bar{\lambda}'_1^{\sigma'_1}$: From $\Sigma_2 \approx \Sigma'_2$ and the determinism of \rightsquigarrow , then $(0, \Sigma'_0) \xRightarrow{\lambda'} (n, \Sigma'_1)$ must have been derived using the E-T-single rule (since $\Sigma_2 \xrightarrow{\alpha^\sigma} \Sigma_1$ has produced an observation that is not ϵ). From this, $\Sigma_2 \approx \Sigma'_2$, and the determinism of \rightsquigarrow , we get that $\Sigma'_2 \xrightarrow{\alpha^{\sigma'}} \Sigma'_1$, $\Sigma'_2 = (w, \overline{(\Omega', m, \sigma')} \cdot (\bar{F}, \bar{I}, H'_2, B'_2 \triangleright (s)_{\bar{F}, F}, n, \sigma'))$, $\Sigma'_1 = (w, \overline{(\Omega', m, \sigma')} \cdot (\bar{F}, \bar{I}, H'_1, B'_1 \triangleright (s')_{\bar{F}, F}, n', \sigma'))$, and $\lambda' = \bar{\lambda}'_1^{\sigma'_1}$ if $f == f' \wedge f \in I$ and $\lambda' = \bar{\lambda}'_1^{\sigma'_1} \cdot \alpha^{\sigma'}$ otherwise. There are two cases:

$f == f' \wedge f \in I$: Hence, $\lambda = \bar{\lambda}_1^{\sigma_1}$ and $\lambda' = \bar{\lambda}'_1^{\sigma'_1}$. Therefore, $\lambda = \lambda'$ follows from $\lambda = \bar{\lambda}_1^{\sigma_1}$, $\lambda' = \bar{\lambda}'_1^{\sigma'_1}$, and $\bar{\lambda}_1^{\sigma_1} = \bar{\lambda}'_1^{\sigma'_1}$.

From $f \in I$, $(0, \Sigma_0) \xRightarrow{\bar{\lambda}_1^{\sigma_1}} (n-1, \Sigma_2)$, $\Sigma_2 = (w, \overline{(\Omega, m, \sigma)} \cdot (\bar{F}, \bar{I}, H_2, B_2 \triangleright (s)_{\bar{F}, F}, n, \sigma'))$, and lemma L.17, we get that for all x, v , σ , if $B_2(x) = v : \sigma$, then $\sigma = S$. From this, $\Sigma'_2 = (w, \overline{(\Omega', m, \sigma')} \cdot (\bar{F}, \bar{I}, H'_2, B'_2 \triangleright (s)_{\bar{F}, F}, n, \sigma'))$, and $\Sigma_2 \approx \Sigma'_2$, we get that $B_2 = B'_2$.

From this, $\Sigma_2 \xrightarrow{\alpha^\sigma} \Sigma_1$, and the determinism of \rightsquigarrow , we get that $\Sigma'_2 \xrightarrow{\alpha^{\sigma'}} \Sigma'_1$. Then, $\Sigma_1 \approx \Sigma'_1$ follows from $\Sigma_2 \xrightarrow{\alpha^\sigma} \Sigma_1$, $\Sigma'_2 \xrightarrow{\alpha^{\sigma'}} \Sigma'_1$, $\Sigma_2 \approx \Sigma'_2$, and lemma L.9.

$\neg(f == f' \wedge f \in I)$: Hence, $\lambda = \bar{\lambda}_1^{\sigma_1} \cdot \alpha^\sigma$ and $\lambda' = \bar{\lambda}'_1^{\sigma'_1} \cdot \alpha^{\sigma'}$. There are two cases:

$\vdash \Sigma_2 : unsafe$: Then, from $\vdash \lambda : safe$, we have $\vdash \alpha^\sigma : safe$. From $\vdash \Sigma_2 : unsafe$, $\Sigma_2 \xrightarrow{\alpha^\sigma} \Sigma_1$, $\vdash \alpha^\sigma : safe$, $\Sigma_2 \approx \Sigma'_2$, $\Sigma'_2 \xrightarrow{\alpha^{\sigma'}} \Sigma'_1$, the determinism of \rightsquigarrow , and lemma L.7, we get $\Sigma_1 \approx \Sigma'_1$, $\alpha^\sigma = \alpha^{\sigma'}$, and $\lambda = \lambda'$ (from $\alpha^\sigma = \alpha^{\sigma'}$ and $\bar{\lambda}_1^{\sigma_1} = \bar{\lambda}'_1^{\sigma'_1}$).

$\vdash \Sigma_2 : safe$: From this and $\Sigma_2 \approx \Sigma'_2$, we have $\vdash \Sigma'_2 : safe$. From this, $\Sigma_0 = \Omega_0(P)$, $\Sigma'_0 = \Omega_0(P')$, and lemma L.16, we get $\lambda \upharpoonright_{nse} = \bar{\lambda}_1^{\sigma_1} \upharpoonright_{nse} \cdot \alpha^\sigma$ and $\lambda' \upharpoonright_{nse} = \bar{\lambda}'_1^{\sigma'_1} \upharpoonright_{nse} \cdot \alpha^{\sigma'}$. There are two cases:

$\alpha^\sigma = \alpha^{\sigma'}$: Then, $\lambda = \lambda'$ follows from $\alpha^\sigma = \alpha^{\sigma'}$ and $\bar{\lambda}_1^{\sigma_1} = \bar{\lambda}'_1^{\sigma'_1}$. Moreover, $\Sigma_1 \approx \Sigma'_1$ follows from $\Sigma_2 \approx \Sigma'_2$, $\Sigma_2 \xrightarrow{\alpha^\sigma} \Sigma_1$, $\Sigma'_2 \xrightarrow{\alpha^{\sigma'}} \Sigma'_1$, $\alpha^\sigma = \alpha^{\sigma'}$, and lemma L.9.

$\alpha^\sigma \neq \alpha^{\sigma'}$: From this, $\lambda \upharpoonright_{nse} = \bar{\lambda}_1^{\sigma_1} \upharpoonright_{nse} \cdot \alpha^\sigma$, $\lambda' \upharpoonright_{nse} = \bar{\lambda}'_1^{\sigma'_1} \upharpoonright_{nse} \cdot \alpha^{\sigma'}$, and $\bar{\lambda}_1^{\sigma_1} \upharpoonright_{nse} = \bar{\lambda}'_1^{\sigma'_1} \upharpoonright_{nse}$, we get that $\lambda \upharpoonright_{nse} \langle \lambda' \upharpoonright_{nse} \rangle$.

Therefore, $\lambda \upharpoonright_{nse} \langle \lambda' \upharpoonright_{nse} \vee (\Sigma_1 \approx \Sigma'_1 \wedge \lambda = \lambda') \rangle$ holds for this case.

Therefore, $\lambda \upharpoonright_{nse} \langle \lambda' \upharpoonright_{nse} \vee (\Sigma_1 \approx \Sigma'_1 \wedge \lambda = \lambda') \rangle$ holds for the induction step.

This concludes the proof. \square

L.1.4 Speculative semantics \rightsquigarrow .

LEMMA L.7 (STEPS OF \rightsquigarrow WITH SAFE OBSERVATIONS PRESERVE LOW-EQUIVALENCE FOR UNSAFE CONFIGURATIONS).

$$\forall \Sigma_0, \Sigma'_0, \Sigma_1, \lambda. \text{ if } \Sigma_0 \rightsquigarrow \Sigma_1, \Sigma_0 \approx \Sigma'_0, \vdash \lambda : \text{safe}, \vdash \Sigma_0 : \text{unsafe} \\ \text{ then } \exists \Sigma'_1, \lambda'. \Sigma'_0 \rightsquigarrow \Sigma'_1, \Sigma_1 \approx \Sigma'_1, \lambda = \lambda'$$

PROOF. Let $\Sigma_0, \Sigma'_0, \Sigma_1$, and λ be such that $\Sigma_0 \rightsquigarrow \Sigma_1, \Sigma_0 \approx \Sigma'_0$, and $\vdash \lambda : \text{safe}$. We proceed by case distinction on the rule used to derive $\Sigma_0 \rightsquigarrow \Sigma_1$:

E-T-speculate-epsilon: Then, $\Sigma_0 \stackrel{\text{def}}{=} \mathbf{w}, \overline{(\Omega, \omega, \sigma)} \cdot (\Omega_0, \mathbf{n} + 1, \sigma), \Omega_0 \xrightarrow{\epsilon} \Omega_1, \Omega_0 \equiv \mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \triangleright \mathbf{s}; \mathbf{s}' \neq \mathbf{s}''; \mathbf{s}'''$ and $\mathbf{s} \neq \text{lfence}$, and $\lambda = \epsilon$, and $\Sigma_1 \stackrel{\text{def}}{=} \mathbf{w}, \overline{(\Omega, \omega, \sigma)} \cdot (\Omega_1, \mathbf{n}, \sigma)$. From $\Sigma_0 \approx \Sigma'_0$, we get that $\Sigma'_0 \stackrel{\text{def}}{=} \mathbf{w}, \overline{(\Omega', \omega, \sigma)} \cdot (\Omega'_0, \mathbf{n} + 1, \sigma)$ and $\Omega_0 \approx \Omega'_0$. From $\Omega_0 \xrightarrow{\epsilon} \Omega_1, \Omega_0 \approx \Omega'_0$, $\vdash \epsilon : \text{safe}$, and lemma L.11, we get that $\Omega'_0 \xrightarrow{\epsilon} \Omega'_1$ and $\Omega_1 \approx \Omega'_1$. We can therefore apply the E-T-speculate-epsilon rule to derive $\Sigma'_0 \rightsquigarrow \Sigma'_1$ where $\lambda' = \epsilon$ and $\Sigma'_1 \stackrel{\text{def}}{=} \mathbf{w}, \overline{(\Omega', \omega, \sigma)} \cdot (\Omega'_1, \mathbf{n}, \sigma)$. Hence, $\lambda = \lambda'$ follows from $\lambda' = \epsilon$ and $\lambda' = \epsilon$, whereas $\Sigma_1 \approx \Sigma'_1$ follows from $\Sigma_0 \approx \Sigma'_0$ and $\Omega_1 \approx \Omega'_1$.

E-T-speculate-lfence: The proof of this case is similar to that of E-T-speculate-epsilon.

E-T-speculate-action: Then, $\Sigma_0 \stackrel{\text{def}}{=} \mathbf{w}, \overline{(\Omega, \omega, \sigma)} \cdot (\Omega_0, \mathbf{n} + 1, \sigma), \Omega_0 \xrightarrow{\lambda^{\sigma_0}} \Omega_1, \Omega_0 \equiv \mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \triangleright \mathbf{s}; \mathbf{s}' \neq \mathbf{s}''; \mathbf{s}'''$ and $\mathbf{s} \neq \text{lfence}$, and $\lambda = \lambda_0^{\sigma_0 \sqcap \sigma}$, and $\Sigma_1 \stackrel{\text{def}}{=} \mathbf{w}, \overline{(\Omega, \omega, \sigma)} \cdot (\Omega_1, \mathbf{n}, \sigma)$. From $\Sigma_0 \approx \Sigma'_0$, we get that $\Sigma'_0 \stackrel{\text{def}}{=} \mathbf{w}, \overline{(\Omega', \omega, \sigma)} \cdot (\Omega'_0, \mathbf{n} + 1, \sigma)$ and $\Omega_0 \approx \Omega'_0$. From $\vdash \Sigma_0 : \text{unsafe}$ and $\vdash \lambda : \text{safe}$, we get that $\vdash \lambda^{\sigma_0} : \text{safe}$. From this, $\Omega_0 \approx \Omega'_0$, and lemma L.11, we get that $\Omega'_0 \xrightarrow{\lambda^{\sigma_0}} \Omega'_1$ and $\Omega_1 \approx \Omega'_1$. We can therefore apply the E-T-speculate-action rule to derive $\Sigma'_0 \rightsquigarrow \Sigma'_1$ where $\lambda' = \lambda_0^{\sigma_0 \sqcap \sigma}$ and $\Sigma'_1 \stackrel{\text{def}}{=} \mathbf{w}, \overline{(\Omega', \omega, \sigma)} \cdot (\Omega'_1, \mathbf{n}, \sigma)$. Hence, $\lambda = \lambda'$ follows from $\lambda' = \lambda_0^{\sigma_0 \sqcap \sigma}$ and $\lambda' = \lambda_0^{\sigma_0 \sqcap \sigma}$, whereas $\Sigma_1 \approx \Sigma'_1$ follows from $\Sigma_0 \approx \Sigma'_0$ and $\Omega_1 \approx \Omega'_1$.

E-T-speculate-if: Then, $\Sigma_0 \stackrel{\text{def}}{=} \mathbf{w}, \overline{(\Omega, \omega, \sigma)} \cdot (\Omega_0, \mathbf{n} + 1, \sigma), \Omega_0 \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \cdot \mathbf{B} \triangleright (\text{if } \mathbf{e} \text{ then } \mathbf{s}'' \text{ else } \mathbf{s}'''; \mathbf{s}')_{\overline{\mathbf{f}}, \mathbf{f}}, \mathbf{C} \equiv \overline{\mathbf{F}}; \overline{\mathbf{I}}, \mathbf{f} \notin \overline{\mathbf{I}}, \Omega_0 \xrightarrow{\alpha^{\sigma_0}} \Omega_1$, $\lambda = \alpha^{\sigma_0 \sqcap \sigma}$, and $\Sigma_1 \stackrel{\text{def}}{=} \mathbf{w}, \overline{(\Omega, \omega, \sigma)} \cdot (\Omega_1, \mathbf{n}, \sigma) \cdot (\Omega_2, \min(\mathbf{w}, \mathbf{n}), \mathbf{U})$ where $\Omega_2 = \mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \cdot \mathbf{B} \triangleright \mathbf{s}''; \mathbf{s}'$ if $\mathbf{B} \triangleright \mathbf{e} \downarrow \mathbf{0} : \sigma_0$ and $\Omega_2 = \mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \cdot \mathbf{B} \triangleright \mathbf{s}''; \mathbf{s}'$ if $\mathbf{B} \triangleright \mathbf{e} \downarrow \mathbf{n} : \sigma_0 \wedge \mathbf{n} > \mathbf{0}$. From $\Sigma_0 \approx \Sigma'_0$, we get that $\Sigma'_0 \stackrel{\text{def}}{=} \mathbf{w}, \overline{(\Omega', \omega, \sigma)} \cdot (\Omega'_0, \mathbf{n} + 1, \sigma)$ where $\overline{(\Omega', \omega, \sigma)} \approx \overline{(\Omega, \omega, \sigma)}$ and $\Omega_0 \approx \Omega'_0$. Thus, $\Omega'_0 \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}', \overline{\mathbf{B}'} \triangleright (\text{if } \mathbf{e} \text{ then } \mathbf{s}'' \text{ else } \mathbf{s}'''; \mathbf{s}')_{\overline{\mathbf{f}}, \mathbf{f}}$ where $\mathbf{H} \approx \mathbf{H}'$ and $\overline{\mathbf{B}} \approx \overline{\mathbf{B}'}$.

From $\vdash \Sigma_0 : \text{unsafe}$ and $\vdash \lambda : \text{safe}$, we get $\vdash \alpha^{\sigma_0} : \text{safe}$ and $\sigma_0 = \mathbf{S}$. From this, $\Sigma_0 \approx \Sigma'_0$, and lemma L.11, we get that $\Omega'_0 \xrightarrow{\alpha^{\sigma_0}} \Omega'_1$ and $\Omega_1 \approx \Omega'_1$.

From $\Omega_0 \approx \Omega'_0$ and $\Omega_0 \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \cdot \mathbf{B} \triangleright (\text{if } \mathbf{e} \text{ then } \mathbf{s}'' \text{ else } \mathbf{s}'''; \mathbf{s}')_{\overline{\mathbf{f}}, \mathbf{f}}$, we have $\Omega_0 \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}, \overline{\mathbf{B}'} \cdot \mathbf{B}' \triangleright (\text{if } \mathbf{e} \text{ then } \mathbf{s}'' \text{ else } \mathbf{s}'''; \mathbf{s}')_{\overline{\mathbf{f}}, \mathbf{f}}$ where $\mathbf{H} \approx \mathbf{H}'$, $\mathbf{B} \approx \mathbf{B}'$, and $\overline{\mathbf{B}} \approx \overline{\mathbf{B}'}$. From $\mathbf{B} \approx \mathbf{B}'$, $\mathbf{B} \triangleright \mathbf{e} \downarrow \mathbf{v} : \sigma_0$, $\sigma_0 = \mathbf{S}$, and lemma L.15, we have that $\mathbf{B}' \triangleright \mathbf{e} \downarrow \mathbf{v} : \sigma_0$.

We can therefore apply the E-T-speculate-if rule to derive $\Sigma'_0 \rightsquigarrow \Sigma'_1$ where $\lambda' = \alpha^{\sigma_0 \sqcap \sigma'}$ and $\Sigma'_1 \stackrel{\text{def}}{=} \mathbf{w}, \overline{(\Omega', \omega, \sigma)} \cdot (\Omega'_1, \mathbf{n}, \sigma') \cdot (\Omega'_2, \mathbf{j}, \mathbf{U})$ where $\Omega'_2 = \mathbf{C}, \mathbf{H}', \overline{\mathbf{B}''} \cdot \mathbf{B}'' \triangleright \mathbf{s}''; \mathbf{s}'$ if $\mathbf{B}' \triangleright \mathbf{e} \downarrow \mathbf{0} : \sigma_0$ and $\Omega'_2 = \mathbf{C}, \mathbf{H}', \overline{\mathbf{B}''} \cdot \mathbf{B}'' \triangleright \mathbf{s}''; \mathbf{s}'$ if $\mathbf{B}' \triangleright \mathbf{e} \downarrow \mathbf{n} : \sigma_0 \wedge \mathbf{n} > \mathbf{0}$. Hence, $\lambda = \lambda'$ follows from $\lambda' = \alpha^{\sigma_0 \sqcap \sigma'}$ and $\lambda' = \alpha^{\sigma_0 \sqcap \sigma'}$, whereas $\Sigma_1 \approx \Sigma'_1$ follows from $\Sigma_0 \approx \Sigma'_0$, $\Omega_1 \approx \Omega'_1$, and $\Omega_2 \approx \Omega'_2$ (which follows from $\mathbf{B} \triangleright \mathbf{e} \downarrow \mathbf{v} : \sigma_0$ and $\mathbf{B}' \triangleright \mathbf{e} \downarrow \mathbf{v} : \sigma_0$).

E-T-speculate-rollback: Then, $\Sigma_0 \stackrel{\text{def}}{=} \mathbf{w}, \overline{(\Omega, \omega, \sigma)} \cdot (\Omega, \mathbf{0}, \sigma), \lambda = \text{rlb}$, and $\Sigma_1 \stackrel{\text{def}}{=} \mathbf{w}, \overline{(\Omega, \omega, \sigma)}$. From $\Sigma_0 \approx \Sigma'_0$, we get that $\Sigma'_0 \stackrel{\text{def}}{=} \mathbf{w}, \overline{(\Omega', \omega, \sigma)} \cdot (\Omega'_0, \mathbf{0}, \sigma)$ where $\overline{(\Omega', \omega, \sigma)} \approx \overline{(\Omega, \omega, \sigma)}$ and $\Omega_0 \approx \Omega'_0$. Hence, we can apply the E-T-speculate-rollback rule to derive $\Sigma'_0 \rightsquigarrow \Sigma'_1$ where $\lambda' = \text{rlb}$ and $\Sigma'_1 \stackrel{\text{def}}{=} \mathbf{w}, \overline{(\Omega', \omega, \sigma)}$. Hence, $\lambda = \lambda'$ follows from $\lambda' = \text{rlb}$ and $\lambda' = \text{rlb}$, whereas $\Sigma_1 \approx \Sigma'_1$ follows from $\Sigma_0 \approx \Sigma'_0$.

E-T-speculate-rollback-stuck: Then, $\Sigma_0 \stackrel{\text{def}}{=} \mathbf{w}, \overline{(\Omega, \omega, \sigma)} \cdot (\Omega_0, \mathbf{W}, \sigma), \vdash \Sigma_0 : \perp, \lambda = \text{rlb}$, and $\Sigma_1 = (\mathbf{w}, \overline{(\Omega, \omega, \sigma)})$. From $\Sigma_0 \approx \Sigma'_0$, we get that $\Sigma'_0 \stackrel{\text{def}}{=} \mathbf{w}, \overline{(\Omega', \omega, \sigma)} \cdot (\Omega'_0, \mathbf{0}, \sigma)$ where $\overline{(\Omega', \omega, \sigma)} \approx \overline{(\Omega, \omega, \sigma)}$ and $\Omega_0 \approx \Omega'_0$. From $\Sigma_0 \approx \Sigma'_0$ and $\vdash \Sigma_0 : \perp$, we get $\vdash \Sigma'_0 : \perp$. Therefore, we can apply the E-T-speculate-rollback-stuck to derive $\Sigma'_0 \rightsquigarrow \Sigma'_1$ where $\lambda' = \text{rlb}$ and $\Sigma'_1 \stackrel{\text{def}}{=} \mathbf{w}, \overline{(\Omega', \omega, \sigma)}$. Hence, $\lambda = \lambda'$ follows from $\lambda' = \text{rlb}$ and $\lambda' = \text{rlb}$, whereas $\Sigma_1 \approx \Sigma'_1$ follows from $\Sigma_0 \approx \Sigma'_0$.

This concludes our proof. \square

LEMMA L.8 (STEPS OF \rightsquigarrow WITHOUT OBSERVATIONS PRESERVE LOW-EQUIVALENCE).

$$\forall \Sigma_0, \Sigma'_0, \Sigma_1, \lambda. \text{ if } \Sigma_0 \rightsquigarrow \Sigma_1, \Sigma_0 \approx \Sigma'_0 \\ \text{ then } \exists \Sigma'_1. \Sigma'_0 \rightsquigarrow \Sigma'_1, \Sigma_1 \approx \Sigma'_1$$

PROOF. Proof is similar to lemma L.7 (only rules E-T-speculate-epsilon and E-T-speculate-lfence) but using lemma L.12 instead of lemma L.11. \square

LEMMA L.9 (STEPS OF \rightsquigarrow WITH SAME OBSERVATIONS PRESERVE LOW-EQUIVALENCE).

$$\forall \Sigma_0, \Sigma'_0, \Sigma_1, \Sigma'_1, \lambda, \lambda'. \text{ if } \Sigma_0 \rightsquigarrow \Sigma_1, \Sigma'_0 \rightsquigarrow \Sigma'_1, \Sigma_0 \approx \Sigma'_0, \lambda = \lambda' \\ \text{ then } \Sigma_1 \approx \Sigma'_1$$

PROOF. Let $\Sigma_0, \Sigma'_0, \Sigma_1, \Sigma'_1, \lambda, \lambda'$ be such that $\Sigma_0 \rightsquigarrow \Sigma_1, \Sigma'_0 \rightsquigarrow \Sigma'_1, \Sigma_0 \approx \Sigma'_0$, and $\lambda = \lambda'$. We proceed by case distinction on the rule used to derive $\Sigma_0 \rightsquigarrow \Sigma_1$:

E-T-speculate-epsilon: Then, $\Sigma_0 \stackrel{\text{def}}{=} \mathbf{w}, (\overline{\Omega, \omega, \sigma}) \cdot (\Omega_0, \mathbf{n} + 1, \sigma), \Omega_0 \xrightarrow{\epsilon} \Omega_1, \Omega_0 \equiv \mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \triangleright \mathbf{s}; \mathbf{s}', \mathbf{s} \neq \mathbf{s}''; \mathbf{s}'''$ and $\mathbf{s} \neq \mathbf{lfence}$, and $\lambda = \epsilon$, and $\Sigma_1 \stackrel{\text{def}}{=} \mathbf{w}, (\overline{\Omega, \omega, \sigma}) \cdot (\Omega_1, \mathbf{n}, \sigma)$. From $\Sigma_0 \approx \Sigma'_0$, we get that $\Sigma'_0 \stackrel{\text{def}}{=} \mathbf{w}, (\overline{\Omega', \omega, \sigma}) \cdot (\Omega'_0, \mathbf{n} + 1, \sigma)$ and $\Omega_0 \approx \Omega'_0$. From $\Omega_0 \approx \Omega'_0$ and $\Omega_0 \equiv \mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \triangleright \mathbf{s}; \mathbf{s}'$, we get that $\Omega'_0 \equiv \mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \triangleright \mathbf{s}; \mathbf{s}'$. Since $\lambda = \lambda'$ and $\lambda = \epsilon$, $\Sigma'_0 \rightsquigarrow \Sigma'_1$ has also been derived using the E-T-speculate-epsilon rule. Thus, $\Sigma'_1 \stackrel{\text{def}}{=} \mathbf{w}, (\overline{\Omega', \omega, \sigma}) \cdot (\Omega'_1, \mathbf{n}, \sigma)$ and $\Omega'_0 \xrightarrow{\epsilon} \Omega'_1$. From $\Omega_0 \approx \Omega'_0, \Omega_0 \xrightarrow{\epsilon} \Omega_1, \Omega'_0 \xrightarrow{\epsilon} \Omega'_1$, and lemma L.10, we get $\Omega_1 \approx \Omega'_1$. Hence, $\Sigma_1 \approx \Sigma'_1$ follows from $\Sigma_0 \approx \Sigma'_0$ and $\Omega_1 \approx \Omega'_1$.

E-T-speculate-lfence: The proof of this case is similar to that of E-T-speculate-epsilon.

E-T-speculate-action: Then, $\Sigma_0 \stackrel{\text{def}}{=} \mathbf{w}, (\overline{\Omega, \omega, \sigma}) \cdot (\Omega_0, \mathbf{n} + 1, \sigma), \Omega_0 \xrightarrow{\lambda_0 \sigma_0} \Omega_1, \Omega \equiv \mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \triangleright \mathbf{s}; \mathbf{s}', \mathbf{s} \neq \mathbf{s}''; \mathbf{s}'''$ and $\mathbf{s} \neq \mathbf{lfence}$, and $\lambda = \lambda_0 \sigma_0 \square \sigma$, and $\Sigma_1 \stackrel{\text{def}}{=} \mathbf{w}, (\overline{\Omega, \omega, \sigma}) \cdot (\Omega_1, \mathbf{n}, \sigma)$. From $\Sigma_0 \approx \Sigma'_0$, we get that $\Sigma'_0 \stackrel{\text{def}}{=} \mathbf{w}, (\overline{\Omega', \omega, \sigma}) \cdot (\Omega'_0, \mathbf{n} + 1, \sigma)$ and $\Omega_0 \approx \Omega'_0$. From $\Omega_0 \approx \Omega'_0$ and $\Omega_0 \equiv \mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \triangleright \mathbf{s}; \mathbf{s}'$, we get that $\Omega'_0 \equiv \mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \triangleright \mathbf{s}; \mathbf{s}'$. Since $\lambda = \lambda'$ and $\lambda = \lambda_0 \sigma_0 \square \sigma$, $\Sigma'_0 \rightsquigarrow \Sigma'_1$ has also been derived using the E-T-speculate-action rule. Thus, $\Sigma'_1 \stackrel{\text{def}}{=} \mathbf{w}, (\overline{\Omega', \omega, \sigma}) \cdot (\Omega'_1, \mathbf{n}, \sigma)$ and $\Omega'_0 \xrightarrow{\lambda'_0 \sigma'_0} \Omega'_1$. From $\lambda = \lambda_0 \sigma_0 \square \sigma, \lambda' = \lambda'_0 \sigma'_0 \square \sigma$, and $\lambda = \lambda'$, we get $\lambda_0 = \lambda'_0$. From $\Omega_0 \approx \Omega'_0, \Omega_0 \xrightarrow{\lambda_0 \sigma_0} \Omega_1, \Omega'_0 \xrightarrow{\lambda'_0 \sigma'_0} \Omega'_1$, and lemma L.13, we get $\sigma_0 = \sigma'_0$. From $\Omega_0 \approx \Omega'_0, \Omega_0 \xrightarrow{\lambda_0 \sigma_0} \Omega_1, \Omega'_0 \xrightarrow{\lambda'_0 \sigma'_0} \Omega'_1$, and lemma L.10, we get $\Omega_1 \approx \Omega'_1$. Hence, $\Sigma_1 \approx \Sigma'_1$ follows from $\Sigma_0 \approx \Sigma'_0$ and $\Omega_1 \approx \Omega'_1$.

E-T-speculate-if: Then, $\Sigma_0 \stackrel{\text{def}}{=} \mathbf{w}, (\overline{\Omega, \omega, \sigma}) \cdot (\Omega_0, \mathbf{n} + 1, \sigma), \Omega_0 \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \cdot \mathbf{B} \triangleright$ (if e then \mathbf{s}'' else \mathbf{s}''' ; \mathbf{s}') $_{\bar{\mathbf{f}}, \mathbf{f}}, \mathbf{C} \equiv \bar{\mathbf{f}}; \bar{\mathbf{I}}, \mathbf{f} \notin \bar{\mathbf{I}}, \Omega_0 \xrightarrow{\alpha \sigma_0} \Omega_1,$ $\lambda = \alpha \sigma_0 \square \sigma$, and $\Sigma_1 \stackrel{\text{def}}{=} \mathbf{w}, (\overline{\Omega, \omega, \sigma}) \cdot (\Omega_1, \mathbf{n}, \sigma) \cdot (\Omega_2, \min(\mathbf{w}, \mathbf{n}), \mathbf{U})$ where $\Omega_2 = \mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \cdot \mathbf{B} \triangleright \mathbf{s}''; \mathbf{s}'$ if $\mathbf{B} \triangleright \mathbf{e} \downarrow \mathbf{0} : \sigma_0$ and $\Omega_2 = \mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \cdot \mathbf{B} \triangleright \mathbf{s}''; \mathbf{s}'$ if $\mathbf{B} \triangleright \mathbf{e} \downarrow \mathbf{n} : \sigma_0 \wedge \mathbf{n} > \mathbf{0}$. From $\Sigma_0 \approx \Sigma'_0$, we get that $\Sigma'_0 \stackrel{\text{def}}{=} \mathbf{w}, (\overline{\Omega', \omega, \sigma}) \cdot (\Omega'_0, \mathbf{n} + 1, \sigma)$ where $(\overline{\Omega', \omega, \sigma}) \approx (\overline{\Omega, \omega, \sigma})$ and $\Omega_0 \approx \Omega'_0$. Thus, $\Omega'_0 \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}', \overline{\mathbf{B}'} \triangleright$ (if e then \mathbf{s}'' else \mathbf{s}''' ; \mathbf{s}') $_{\bar{\mathbf{f}}, \mathbf{f}}$ where $\mathbf{H} \approx \mathbf{H}'$ and $\overline{\mathbf{B}} \approx \overline{\mathbf{B}'}$. Hence, $\Sigma'_0 \rightsquigarrow \Sigma'_1$ has also been derived using the E-T-speculate-action rule. Thus, $\Sigma'_1 \stackrel{\text{def}}{=} \mathbf{w}, (\overline{\Omega', \omega, \sigma}) \cdot (\Omega'_1, \mathbf{n}, \sigma) \cdot (\Omega'_2, \min(\mathbf{w}, \mathbf{n}), \mathbf{U})$ and $\Omega'_0 \xrightarrow{\alpha' \sigma'_0} \Omega'_1$. From $\lambda = \alpha \sigma_0 \square \sigma, \lambda' = \alpha' \sigma'_0 \square \sigma$, and $\lambda = \lambda'$, we get $\alpha = \alpha'$. From $\Omega_0 \approx \Omega'_0, \Omega_0 \xrightarrow{\alpha \sigma_0} \Omega_1, \Omega'_0 \xrightarrow{\alpha' \sigma'_0} \Omega'_1$, and lemma L.13, we get $\sigma_0 = \sigma'_0$. From $\alpha \sigma_0 = \alpha' \sigma'_0, \Sigma_0 \approx \Sigma'_0, \Omega_0 \xrightarrow{\alpha \sigma_0} \Omega_1, \Omega'_0 \xrightarrow{\alpha' \sigma'_0} \Omega'_1$, and lemma L.10, we get $\Omega_1 \approx \Omega'_1$. Observe that from $\Omega_0 \approx \Omega'_0$ and $\Omega'_0 \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}', \overline{\mathbf{B}'} \triangleright$ (if e then \mathbf{s}'' else \mathbf{s}''' ; \mathbf{s}') $_{\bar{\mathbf{f}}, \mathbf{f}}$, we get that $\alpha \sigma_0 = (\text{if}(\mathbf{n})^{\sigma_0} \leftrightarrow \mathbf{B} \triangleright \mathbf{e} \downarrow \mathbf{n} : \sigma_0$ and $\alpha' \sigma'_0 = (\text{if}(\mathbf{n}')^{\sigma'_0} \leftrightarrow \mathbf{B}' \triangleright \mathbf{e} \downarrow \mathbf{n}' : \sigma'_0$. From this and $\alpha \sigma_0 = \alpha' \sigma'_0$, we get $\mathbf{n} : \sigma_0 = \mathbf{n}' : \sigma'_0$. Therefore, $\Omega_2 \approx \Omega'_2$. Hence, $\Sigma_1 \approx \Sigma'_1$ follows from $\Sigma_0 \approx \Sigma'_0, \Omega_1 \approx \Omega'_1$, and $\Omega_2 \approx \Omega'_2$.

E-T-speculate-rollback: Then, $\Sigma_0 \stackrel{\text{def}}{=} \mathbf{w}, (\overline{\Omega, \omega, \sigma}) \cdot (\Omega, \mathbf{0}, \sigma), \lambda = \mathbf{r1b}$, and $\Sigma_1 \stackrel{\text{def}}{=} \mathbf{w}, (\overline{\Omega, \omega, \sigma})$. From $\Sigma_0 \approx \Sigma'_0$, we get that $\Sigma'_0 \stackrel{\text{def}}{=} \mathbf{w}, (\overline{\Omega', \omega, \sigma}) \cdot (\Omega', \mathbf{0}, \sigma)$ where $(\overline{\Omega', \omega, \sigma}) \approx (\overline{\Omega, \omega, \sigma})$ and $\Omega_0 \approx \Omega'_0$. Hence, $\Sigma'_0 \rightsquigarrow \Sigma'_1$ has also been derived using the E-T-speculate-rollback rule. Thus, $\Sigma'_1 \stackrel{\text{def}}{=} \mathbf{w}, (\overline{\Omega', \omega, \sigma})$. Hence, $\Sigma_1 \approx \Sigma'_1$ follows from $(\overline{\Omega', \omega, \sigma}) \approx (\overline{\Omega, \omega, \sigma})$.

E-T-speculate-rollback-stuck: Then, $\Sigma_0 \stackrel{\text{def}}{=} \mathbf{w}, (\overline{\Omega, \omega, \sigma}) \cdot (\Omega_0, \mathbf{W}, \sigma), \vdash \Sigma_0 : \perp, \lambda = \mathbf{r1b}$, and $\Sigma_1 = (\mathbf{w}, (\overline{\Omega, \omega, \sigma}))$. From $\Sigma_0 \approx \Sigma'_0$, we get that $\Sigma'_0 \stackrel{\text{def}}{=} \mathbf{w}, (\overline{\Omega', \omega, \sigma}) \cdot (\Omega'_0, \mathbf{0}, \sigma)$ where $(\overline{\Omega', \omega, \sigma}) \approx (\overline{\Omega, \omega, \sigma})$ and $\Omega_0 \approx \Omega'_0$. From $\Sigma_0 \approx \Sigma'_0$ and $\vdash \Sigma_0 : \perp$, we get $\vdash \Sigma'_0 : \perp$. Hence, $\Sigma'_0 \rightsquigarrow \Sigma'_1$ has also been derived using the E-T-speculate-rollback-stuck rule. Thus, $\Sigma'_1 \stackrel{\text{def}}{=} \mathbf{w}, (\overline{\Omega', \omega, \sigma})$. Hence, $\Sigma_1 \approx \Sigma'_1$ follows from $(\overline{\Omega', \omega, \sigma}) \approx (\overline{\Omega, \omega, \sigma})$.

This concludes our proof. \square

L.1.5 Non-speculative semantics \rightarrow .

LEMMA L.10 (STEPS OF \rightarrow WITH SAME OBSERVATIONS PRESERVE LOW-EQUIVALENCE).

$$\forall \Omega_0, \Omega'_0, \lambda, \lambda', \Omega_1, \Omega'_1. \text{ if } \Omega_0 \xrightarrow{\lambda} \Omega_1, \Omega'_0 \xrightarrow{\lambda'} \Omega'_1, \Omega_0 \approx \Omega'_0, \lambda = \lambda' \\ \text{ then } \Omega_1 \approx \Omega'_1$$

PROOF. Let $\Omega_0, \Omega'_0, \lambda, \lambda', \Omega_1, \Omega'_1$ be such that $\Omega_0 \xrightarrow{\lambda} \Omega_1, \Omega'_0 \xrightarrow{\lambda'} \Omega'_1, \Omega_0 \approx \Omega'_0$, and $\lambda = \lambda'$. We proceed by structural induction on the rule used to derive $\Omega_0 \xrightarrow{\lambda} \Omega_1$.

Base case: There are several cases depending on the rule used to derive $\Omega_0 \xrightarrow{\lambda} \Omega_1$:

E-T-sequence: Then, $\Omega_0 \stackrel{\text{def}}{=} C, H, \bar{B} \triangleright \text{skip}; s, \lambda = \epsilon$, and $\Omega_1 \stackrel{\text{def}}{=} C, H, \bar{B} \triangleright s$. From $\Omega_0 \approx \Omega'_0$, we have that $\Omega'_0 \stackrel{\text{def}}{=} C, H', \bar{B}' \triangleright \text{skip}; s$ where $H \approx H'$ and $\bar{B} \approx \bar{B}'$. Therefore, $\Omega'_0 \xrightarrow{\lambda'} \Omega'_1$ has also been derived using the E-T-sequence where $\Omega'_1 \stackrel{\text{def}}{=} C, H', \bar{B}' \triangleright s$ and $\lambda' = \epsilon$. Hence, $\Omega_1 \approx \Omega'_1$ follows from $H \approx H', \bar{B} \approx \bar{B}'$, and $B \approx B'$.

E-T-if-true: Then, $\Omega_0 \stackrel{\text{def}}{=} C, H, \bar{B} \cdot B \triangleright \text{ifz } e \text{ then } s \text{ else } s', B \triangleright e \downarrow 0 : \sigma, \lambda = (\text{if}(0))^\sigma$, and $\Omega_1 \stackrel{\text{def}}{=} C, H, \bar{B} \cdot B \triangleright s$. From $\Omega_0 \approx \Omega'_0$, we have that $\Omega'_0 \stackrel{\text{def}}{=} C, H', \bar{B}' \cdot B' \triangleright \text{ifz } e \text{ then } s \text{ else } s'$ where $H \approx H', \bar{B} \approx \bar{B}'$, and $B \approx B'$. From this, $\lambda = (\text{if}(0))^\sigma$, and $\lambda = \lambda', \Omega'_0 \xrightarrow{\lambda'} \Omega'_1$ has also been derived using the E-T-if-true rule. Therefore, $\Omega_1 \stackrel{\text{def}}{=} C, H', \bar{B}' \cdot B' \triangleright s$. Hence, $\Omega_1 \approx \Omega'_1$ follows from $H \approx H', \bar{B} \approx \bar{B}'$, and $B \approx B'$.

E-T-if-false: The proof of this case is similar to that of the E-T-if-true case.

E-T-letin: Then, $\Omega_0 \stackrel{\text{def}}{=} C, H, \bar{B} \cdot B \triangleright \text{let } x = e \text{ in } s, \lambda = \epsilon, B \triangleright e \downarrow v : \sigma$, and $\Omega_1 \stackrel{\text{def}}{=} C, H, \bar{B} \cdot B \cup x \mapsto v : \sigma \triangleright s$. From $\Omega_0 \approx \Omega'_0$, we have that $\Omega'_0 \stackrel{\text{def}}{=} C, H', \bar{B}' \cdot B' \triangleright \text{let } x = e \text{ in } s$ where $H \approx H', \bar{B} \approx \bar{B}'$, and $B \approx B'$. From this, $\Omega'_0 \xrightarrow{\lambda'} \Omega'_1$ has also been derived using the E-T-letin rule. Hence, $\Omega'_1 \stackrel{\text{def}}{=} C, H', \bar{B}' \cdot B' \cup x \mapsto v' : \sigma' \triangleright s$ where $B' \triangleright e \downarrow v' : \sigma'$. From lemma L.14, $B \approx B', B \triangleright e \downarrow v : \sigma$, and $B' \triangleright e \downarrow v' : \sigma'$, we have $v : \sigma \approx v' : \sigma'$. Hence, $\Omega_1 \approx \Omega'_1$ follows from $H \approx H', \bar{B} \approx \bar{B}', B \approx B'$, and $v : \sigma \approx v' : \sigma'$.

E-T-write: Then, $\Omega_0 \stackrel{\text{def}}{=} C, H_0, \bar{B} \cdot B \triangleright e_1 := e_2, \lambda = \text{write}(|n| \mapsto v)^{\sigma_1 \sqcup \sigma_2}, B \triangleright e_1 \downarrow n : \sigma_1, B \triangleright e_2 \downarrow v : \sigma_2, H_0 = H_2; |n| \mapsto v_0 : \sigma_0; H_3, H_1 = H_2; |n| \mapsto v : S; H_3$, and $\Omega_1 \stackrel{\text{def}}{=} C, H_1, \bar{B} \cdot B \triangleright \text{skip}$. From $\Omega_0 \approx \Omega'_0, \Omega'_0 \stackrel{\text{def}}{=} C, H'_0, \bar{B}' \cdot B \triangleright e_1 := e_2$ where $H_0 \approx H'_0, \bar{B} \approx \bar{B}'$, and $B \approx B'$. From $H_0 \approx H'_0$ and $H_0 = H_2; |n| \mapsto v_0 : \sigma_0; H_3$, it follows that there are H'_2, H'_3 such that $H'_0 = H'_2; |n| \mapsto v'_0 : \sigma'_0; H'_3, H_2 \approx H'_2$, and $H_3 \approx H'_3$. Therefore, $\Omega'_0 \xrightarrow{\lambda'} \Omega'_1$ has also been derived using the E-T-write rule. From this, we get $\Omega'_1 \stackrel{\text{def}}{=} C, H'_2; |n'| \mapsto v' : S; H'_3, \bar{B}' \cdot B' \triangleright \text{skip}$ where $B' \triangleright e_1 \downarrow n' : \sigma'_1$ and $B' \triangleright e_2 \downarrow v' : \sigma'_2$. From $\lambda = \text{write}(|n| \mapsto v)^{\sigma_1 \sqcup \sigma_2}, \lambda' = \text{write}(|n'| \mapsto v')^{\sigma'_1 \sqcup \sigma'_2}$, and $\lambda = \lambda'$, we get that $|n| = |n'|$ and $v = v'$. Hence, $\Omega_1 \approx \Omega'_1$ follows from $H_2 \approx H'_2, H_3 \approx H'_3, \bar{B} \approx \bar{B}', B \approx B', |n| = |n'|$, and $v = v'$ (the latter is needed since the label of the written value is S).

E-T-read: Then, $\Omega_0 \stackrel{\text{def}}{=} C, H, \bar{B} \cdot B \triangleright \text{let } x = \text{rd } e \text{ in } s, B \triangleright e \downarrow n : \sigma_1, H = H_1; |n| \mapsto v : \sigma_0; H_2, \lambda = \text{read}(|n|)^{\sigma_1}$, and $\Omega_1 \stackrel{\text{def}}{=} C, H, \bar{B} \cdot B \cup x \mapsto v : \sigma_0 \triangleright s$. From $\Omega_0 \approx \Omega'_0$, we get that $\Omega'_0 \stackrel{\text{def}}{=} C, H', \bar{B}' \cdot B' \triangleright \text{let } x = \text{rd } e \text{ in } s, H \approx H', \bar{B} \approx \bar{B}'$, and $B \approx B'$. Therefore, $\Omega'_0 \xrightarrow{\lambda'} \Omega'_1$ has also been derived using the E-T-read rule. Thus, $\Omega'_1 \stackrel{\text{def}}{=} C, H', \bar{B}' \cdot B' \cup x \mapsto v' : \sigma'_0 \triangleright s$ where $B' \triangleright e \downarrow n' : \sigma'_1, H'(|n'|) = v' : \sigma'_0$, and $\lambda' = \text{read}(|n'|)^{\sigma'_1}$. From $\lambda = \text{read}(|n|)^{\sigma_1}, \lambda' = \text{read}(|n'|)^{\sigma'_1}$, and $\lambda = \lambda'$, we get $|n| = |n'|$. From $|n| = |n'|, H \approx H', H(|n|) = v : \sigma_0$, and $H'(|n'|) = v' : \sigma'_0$, we get $v : \sigma_0 \approx v' : \sigma'_0$. Hence, $\Omega_1 \approx \Omega'_1$ follows from $H \approx H', \bar{B} \approx \bar{B}', B \approx B', |n| = |n'|$, and $v : \sigma_0 \approx v' : \sigma'_0$.

E-T-write-prv: Then, we have that $\Omega_0 \stackrel{\text{def}}{=} C, H, \bar{B} \cdot B \triangleright e :=_{\text{pr}} e', B \triangleright e \downarrow n : \sigma_0, B \triangleright e' \downarrow v : \sigma_1, \lambda = \text{write}(-|n|)^{\sigma_0}, H = H_2; -|n| \mapsto v_0 : \sigma_2; H_3, H_1 = H_2; -|n| \mapsto v : \sigma_1; H_3$, and $\Omega_1 \stackrel{\text{def}}{=} C, H_1, \bar{B} \cdot B \triangleright \text{skip}$. From $\Omega_0 \approx \Omega'_0$, we get that $\Omega'_0 \stackrel{\text{def}}{=} C, H', \bar{B}' \cdot B' \triangleright e :=_{\text{pr}} e', H \approx H', \bar{B} \approx \bar{B}'$, and $B \approx B'$. Hence, $\Omega'_0 \xrightarrow{\lambda'} \Omega'_1$ has also been derived using the E-T-write-prv rule. Thus, $\Omega'_1 \stackrel{\text{def}}{=} C, H'_1, \bar{B}' \cdot B' \triangleright \text{skip}$ where $B' \triangleright e \downarrow n' : \sigma'_0, B' \triangleright e' \downarrow v' : \sigma'_1, H'_1 = H'_2; -|n'| \mapsto v' : \sigma'_1; H_3$, and $\lambda' = \text{write}(-|n'|)^{\sigma'_0}$. From $\lambda = \lambda', \lambda' = \text{write}(-|n'|)^{\sigma'_0}$, and $\lambda = \text{write}(-|n|)^{\sigma_0}$, we get $|n| : \sigma_0 = |n'| : \sigma'_0$. From $B \approx B', B' \triangleright e \downarrow v' : \sigma'_1, B \triangleright e' \downarrow v : \sigma_1$, and lemma L.14, we get $v : \sigma_1 \approx v' : \sigma'_1$. Hence, $\Omega_1 \approx \Omega'_1$ follows from $H \approx H', \bar{B} \approx \bar{B}', B \approx B', |n| : \sigma_0 = |n'| : \sigma'_0$, and $v : \sigma_1 \approx v' : \sigma'_1$.

E-T-read-prv: Then, $\Omega_0 \stackrel{\text{def}}{=} C, H, \bar{B} \cdot B \triangleright \text{let } x = \text{rd}_{\text{pr}} e \text{ in } s, B \triangleright e \downarrow n : \sigma_0, H = H_1; -|n| \mapsto v : \sigma_1; H_2, \lambda = \text{read}(-|n|)^{\sigma_0}$, and $\Omega_1 \stackrel{\text{def}}{=} C, H, \bar{B} \cdot B \cup x \mapsto v : U \triangleright s$. From $\Omega_0 \approx \Omega'_0$, we get that $\Omega'_0 \stackrel{\text{def}}{=} C, H', \bar{B}' \cdot B' \triangleright \text{let } x = \text{rd}_{\text{pr}} e \text{ in } s$ where $H \approx H', \bar{B} \approx \bar{B}'$, and $B \approx B'$. Hence, $\Omega'_0 \xrightarrow{\lambda'} \Omega'_1$ has also been derived using the E-T-read-prv rule. Thus, $\Omega'_1 \stackrel{\text{def}}{=} C, H', \bar{B}' \cdot B' \cup x \mapsto v' : U \triangleright s$ where $B' \triangleright e \downarrow n' : \sigma'_0, \lambda' = \text{read}(-|n'|)^{\sigma'_0}$, and $H'(-|n'|) = v' : \sigma'_1$. From $\lambda = \lambda', \lambda = \text{read}(-|n|)^{\sigma_0}$, and $\lambda' = \text{read}(-|n'|)^{\sigma'_0}$, we have $|n| : \sigma_0 = |n'| : \sigma'_0$. From this and $H \approx H'$, we have $H(-|n|) \approx H'(-|n'|)$. Hence, $\Omega_1 \approx \Omega'_1$ follows from $H \approx H', \bar{B} \approx \bar{B}', B \approx B', |n| : \sigma_0 = |n'| : \sigma'_0$, and $v : \sigma_1 \approx v' : \sigma'_1$ (observe that the latter is enough since x is tagged U).

E-T-call-internal: Then, $\Omega_0 \stackrel{\text{def}}{=} C, H, \bar{B} \cdot B \triangleright (\text{call } f \text{ e})_{\bar{F}}, \lambda = \epsilon, C.\text{intfs} \vdash f, f' : \text{internal}, \bar{F}' = \bar{F}''; f', f(x) \mapsto s; \text{return}; \in C.\text{funs}, B \triangleright e \downarrow v : \sigma$, and $\Omega_1 \stackrel{\text{def}}{=} C, H, \bar{B} \cdot B \cdot x \mapsto v : \sigma \triangleright (s; \text{return};)_{\bar{F}; \bar{F}'}$. From $\Omega_0 \approx \Omega'_0$, we get that $\Omega'_0 \stackrel{\text{def}}{=} C, H', \bar{B}' \cdot B' \triangleright (\text{call } f \text{ e})_{\bar{F}'}$ where $H \approx H', \bar{B} \approx \bar{B}'$, and $B \approx B'$. Hence, $\Omega'_0 \xrightarrow{\lambda'} \Omega'_1$ has also been derived using the E-T-call-internal rule. Thus, $\Omega'_1 \stackrel{\text{def}}{=} C, H', \bar{B}' \cdot B' \cdot x \mapsto v' : \sigma' \triangleright (s; \text{return};)_{\bar{F}; \bar{F}'}$ and $B' \triangleright e \downarrow v' : \sigma'$. From $B \approx B', B \triangleright e \downarrow v : \sigma, B' \triangleright e \downarrow v' : \sigma'$, and lemma L.14, we get $v : \sigma \approx v' : \sigma'$. Hence, $\Omega_1 \approx \Omega'_1$ follows from $H \approx H', \bar{B} \approx \bar{B}', B \approx B'$, and $v : \sigma \approx v' : \sigma'$.

E-T-callback: Then, $\Omega_0 \stackrel{\text{def}}{=} C, H, \bar{B} \cdot B \triangleright (\text{call } f \text{ e})_{\bar{F}}, \lambda = \text{call } f \text{ v!}^\sigma, \bar{F}' = \bar{F}''; f', f(x) \mapsto s; \text{return}; \in C.\text{funs}, C.\text{intfs} \vdash f', f : \text{out}, B \triangleright e \downarrow v : \sigma$, and $\Omega_1 \stackrel{\text{def}}{=} C, H, \bar{B} \cdot B \cdot x \mapsto v : \sigma \triangleright (s; \text{return};)_{\bar{F}; \bar{F}'}$. From $\Omega_0 \approx \Omega'_0$, we get that $\Omega'_0 \stackrel{\text{def}}{=} C, H', \bar{B}' \cdot B' \triangleright (\text{call } f \text{ e})_{\bar{F}'}$ where $H \approx H', \bar{B} \approx \bar{B}'$, and $B \approx B'$. Hence, $\Omega'_0 \xrightarrow{\lambda'} \Omega'_1$ has also been derived using the E-T-callback rule. Thus, $\Omega'_1 \stackrel{\text{def}}{=} C, H', \bar{B}' \cdot B' \cdot x \mapsto v' : \sigma'$

$\sigma' \triangleright (s; \text{return}); \overline{f}, f', B' \triangleright e \downarrow v' : \sigma'$, and $\lambda' = \text{call } f \ v!^{\sigma'}$. From $\lambda = \lambda'$, $\lambda = \text{call } f \ v!^{\sigma}$, and $\lambda' = \text{call } f \ v!^{\sigma'}$, we get $v : \sigma = v' : \sigma'$.

Hence, $\Omega_1 \approx \Omega'_1$ follows from $H \approx H'$, $\overline{B} \approx \overline{B}'$, $B \approx B'$, and $v : \sigma = v' : \sigma'$.

E-T-call: The proof of this case is similar to that of the case E-T-callback.

E-T-ret-internal: Then, $\Omega_0 \stackrel{\text{def}}{=} C, H, \overline{B} \triangleright B \triangleright (\text{return}); \overline{f}, f', C.\text{intfs} \vdash f, f' : \text{internal}$, $\lambda = \epsilon$, and $\Omega_1 = C, H, \overline{B} \triangleright (\text{skip})_{\overline{f}}$.

From $\Omega_0 \approx \Omega'_0$, we get that $\Omega'_0 \stackrel{\text{def}}{=} C, H', \overline{B}' \triangleright B' \triangleright (\text{return}); \overline{f}, f'$ where $H \approx H'$, $\overline{B} \approx \overline{B}'$, and $B \approx B'$. Hence, $\Omega'_0 \xrightarrow{\lambda'} \Omega'_1$ has also been derived using the E-T-ret-internal rule. Thus, $\Omega_1 = C, H', \overline{B}' \triangleright (\text{skip})_{\overline{f}}$. Hence, $\Omega_1 \approx \Omega'_1$ follows from $H \approx H'$ and $\overline{B} \approx \overline{B}'$.

E-T-retback: The proof of this case is similar to that of E-T-retback.

E-T-return: The proof of this case is similar to that of E-T-retback.

E-T-lfence: The proof of this case is similar to that of E-T-skip.

E-T-cmove-true: Then, $\Omega_0 \stackrel{\text{def}}{=} C, H, \overline{B} \triangleright B \triangleright \text{let } x = e_0 \text{ (if } e_1) \text{ in } s, x \in \text{dom}(B), \lambda = \epsilon, \Omega_1 \stackrel{\text{def}}{=} C, H, \overline{B} \triangleright B \cup x \mapsto v_0 : \sigma \triangleright s, B \triangleright e_0 \downarrow v_0 : \sigma_0, B \triangleright e_1 \downarrow 0 : \sigma_1$, and $\sigma = \sigma_0 \sqcup \sigma_1$. From $\Omega_0 \approx \Omega'_0$, we have that $\Omega'_0 \stackrel{\text{def}}{=} C, H', \overline{B}' \triangleright B' \triangleright \text{let } x = e_0 \text{ (if } e_1) \text{ in } s$ where $H \approx H'$, $\overline{B} \triangleright B \approx \overline{B}' \triangleright B'$, and $B \approx B'$. From $B \approx B'$, $B \triangleright e_1 \downarrow 0 : \sigma_1$, and lemma L.15, we get that $B' \triangleright e_1 \downarrow n' : \sigma'_1$ and $0 : \sigma_1 \approx n' : \sigma'_1$. There are two cases:

$\sigma_1 = S$: Then, $\sigma'_1 = S$ and $v'_1 = 0$ follows from $0 : \sigma_1 \approx n' : \sigma'_1$. Hence, $\Omega'_0 \xrightarrow{\lambda'} \Omega'_1$ has also been derived using the E-T-cmove-true rule. Thus, $\Omega_1 \stackrel{\text{def}}{=} C, H', \overline{B}' \triangleright B' \cup x \mapsto v'_0 : \sigma' \triangleright s$ where $B' \triangleright e_0 \downarrow v'_0 : \sigma'_0$ and $\sigma' = \sigma'_0 \sqcup \sigma'_1$. From $B \approx B'$, $B \triangleright e_0 \downarrow v_0 : \sigma_0$, $B' \triangleright e_0 \downarrow v'_0 : \sigma'_0$, and lemma L.14, we get $v_0 : \sigma_0 \approx v'_0 : \sigma'_0$. Hence, $\Omega_1 \approx \Omega'_1$ follows from $H \approx H'$, $\overline{B} \approx \overline{B}'$, $B \approx B'$, and $v_0 : \sigma_0 \approx v'_0 : \sigma'_0$.

$\sigma_1 = U$: Then, $\sigma'_1 = U$ holds as well. There are two cases:

$v'_1 = 0$: Hence, $\Omega'_0 \xrightarrow{\lambda'} \Omega'_1$ has also been derived using the E-T-cmove-true rule. Thus, $\Omega_1 \stackrel{\text{def}}{=} C, H', \overline{B}' \triangleright B' \cup x \mapsto v'_0 : \sigma' \triangleright s$ where $B' \triangleright e_0 \downarrow v'_0 : \sigma'_0$ and $\sigma' = \sigma'_0 \sqcup \sigma'_1$. Hence, $\Omega_1 \approx \Omega'_1$ follows from $H \approx H'$, $\overline{B} \approx \overline{B}'$, $B \approx B'$, and $\sigma = \sigma' = U$.

$v'_1 > 0$: Hence, $\Omega'_0 \xrightarrow{\lambda'} \Omega'_1$ has been derived using the E-T-cmove-false rule. Thus, $\Omega'_1 \stackrel{\text{def}}{=} C, H', \overline{B}' \triangleright B' \cup x \mapsto v_0 : \sigma' \triangleright s$ where $B(x) = v'_0 : \sigma'_0$ and $\sigma' = \sigma'_0 \sqcup \sigma'_1$. Hence, $\Omega_1 \approx \Omega'_1$ follows from $H \approx H'$, $\overline{B} \approx \overline{B}'$, $B \approx B'$, and $\sigma = \sigma' = U$.

E-T-cmove-false: The proof of this case is similar to that of the E-T-cmove-true case.

Induction step: Then, $\Omega_0 \xrightarrow{\lambda} \Omega_1$ has been derived using the E-T-step rule. Then, $\Omega_0 \stackrel{\text{def}}{=} C, H, \overline{B} \triangleright s; s'', C, H, \overline{B} \triangleright s \xrightarrow{\lambda} C, H_1, \overline{B}_1 \triangleright s_1$, and $\Omega_1 = C, H_1, \overline{B}_1 \triangleright s_1; s''$. From $\Omega_0 \approx \Omega'_0$, we get that $\Omega_0 \stackrel{\text{def}}{=} C, H', \overline{B}' \triangleright s; s''$ where $H \approx H'$ and $\overline{B} \approx \overline{B}'$. Hence, $\Omega'_0 \xrightarrow{\lambda'} \Omega'_1$ has also been derived using the E-T-step rule. Therefore, we get $\Omega'_0 \stackrel{\text{def}}{=} C, H', \overline{B}' \triangleright s; s'', C, H', \overline{B}' \triangleright s \xrightarrow{\lambda'} C, H'_1, \overline{B}'_1 \triangleright s_1$, and $\Omega'_1 = C, H'_1, \overline{B}'_1 \triangleright s_1; s''$. From $H \approx H'$, $\overline{B} \approx \overline{B}'$, $C, H, \overline{B} \triangleright s \xrightarrow{\lambda} C, H_1, \overline{B}_1 \triangleright s_1$, $C, H', \overline{B}' \triangleright s \xrightarrow{\lambda'} C, H'_1, \overline{B}'_1 \triangleright s_1$, $\lambda = \lambda'$, and the induction hypothesis, we get that $H_1 \approx H'_1$ and $\overline{B}_1 \approx \overline{B}'_1$. Hence, $\Omega_1 \approx \Omega'_1$ follows from $H_1 \approx H'_1$ and $\overline{B}_1 \approx \overline{B}'_1$. \square

LEMMA L.11 (STEPS OF $\xrightarrow{\lambda}$ WITH SAFE OBSERVATIONS PRESERVE LOW-EQUIVALENCE).

$$\begin{aligned} \forall \Omega_0, \Omega'_0, \lambda, \Omega_1. \text{ if } \Omega_0 \xrightarrow{\lambda} \Omega_1, \Omega_0 \approx \Omega'_0, \vdash \lambda : \text{safe} \\ \text{ then } \exists \lambda', \Omega'_1. \Omega'_0 \xrightarrow{\lambda'} \Omega'_1, \Omega_1 \approx \Omega'_1, \lambda = \lambda' \end{aligned}$$

PROOF. Let $\Omega_0, \Omega'_0, \lambda$, and Ω_1 be such that $\Omega_0 \approx \Omega'_0$, $\Omega_0 \xrightarrow{\lambda} \Omega_1$, and $\vdash \lambda : \text{safe}$. We proceed by structural induction on the rules used to derive $\Omega_0 \xrightarrow{\lambda} \Omega_1$:

Base case: There are several cases depending on the rule used to derive $\Omega_0 \xrightarrow{\lambda} \Omega_1$:

E-T-sequence: Then, $\Omega_0 \stackrel{\text{def}}{=} C, H, \overline{B} \triangleright \text{skip}; s, \lambda = \epsilon$, and $\Omega_1 \stackrel{\text{def}}{=} C, H, \overline{B} \triangleright s$. From $\Omega_0 \approx \Omega'_0$, we have that $\Omega'_0 \stackrel{\text{def}}{=} C, H', \overline{B}' \triangleright \text{skip}; s$ where $H \approx H'$ and $\overline{B} \approx \overline{B}'$. We can apply the E-T-sequence rule to derive $\Omega'_0 \xrightarrow{\lambda'} \Omega'_1$ where $\Omega'_1 \stackrel{\text{def}}{=} C, H', \overline{B}' \triangleright s$ and $\lambda' = \epsilon$. Hence, $\Omega_1 \approx \Omega'_1$ follows from $\Omega_0 \approx \Omega'_0$ and $\lambda = \lambda'$ follows from $\lambda = \epsilon$ and $\lambda' = \epsilon$.

E-T-if-true: Then, $\Omega_0 \stackrel{\text{def}}{=} C, H, \overline{B} \triangleright B \triangleright \text{ifz } e \text{ then } s \text{ else } s', B \triangleright e \downarrow 0 : \sigma, \lambda = (\text{if}(0))^\sigma$, and $\Omega_1 \stackrel{\text{def}}{=} C, H, \overline{B} \triangleright B \triangleright s$. From $\Omega_0 \approx \Omega'_0$, we have that $\Omega'_0 \stackrel{\text{def}}{=} C, H', \overline{B}' \triangleright B' \triangleright \text{ifz } e \text{ then } s \text{ else } s'$ where $H \approx H'$, $\overline{B} \approx \overline{B}'$, and $B \approx B'$. From $B \triangleright e \downarrow 0 : \sigma$, $B \approx B'$, and lemma L.15, we get that $B' \triangleright e \downarrow v' : \sigma'$ and $0 : \sigma \approx v' : \sigma'$. From $\vdash \lambda : \text{safe}$ and $\lambda = (\text{if}(0))^\sigma$, it follows that $\sigma = S$ and $\lambda = (\text{if}(0))^S$. From this and $0 : \sigma \approx v' : \sigma'$, it follows that $v' = 0$. Hence, $B' \triangleright e \downarrow 0 : S'$ holds. Therefore, we can apply the E-T-if-true rule to derive $\Omega'_0 \xrightarrow{\lambda'} \Omega'_1$ where $\Omega'_1 \stackrel{\text{def}}{=} C, H', \overline{B}' \triangleright B' \triangleright s$ and $\lambda' = (\text{if}(0))^S$. Hence, $\Omega_1 \approx \Omega'_1$ follows from $\Omega_0 \approx \Omega'_0$ and $\lambda = \lambda'$ follows from $\lambda = (\text{if}(0))^S$ and $\lambda' = (\text{if}(0))^S$.

E-T-if-false: The proof of this case is similar to that of the E-T-if-true case.

- E-T-letin:** Then, $\Omega_0 \stackrel{\text{def}}{=} C, H, \bar{B} \cdot B \triangleright \text{let } x = e \text{ in } s, \lambda = \epsilon, B \triangleright e \downarrow v : \sigma$, and $\Omega_1 \stackrel{\text{def}}{=} C, H, \bar{B} \cdot B \cup x \mapsto v : \sigma \triangleright s$. From $\Omega_0 \approx \Omega'_0$, we have that $\Omega'_0 \stackrel{\text{def}}{=} C, H', \bar{B}' \cdot B' \triangleright \text{let } x = e \text{ in } s$ where $H \approx H', \bar{B} \approx \bar{B}'$, and $B \approx B'$. From $B \approx B', B \triangleright e \downarrow v : \sigma$, and lemma L.15, we have $B' \triangleright e \downarrow v' : \sigma'$ and $v : \sigma \approx v' : \sigma'$. Therefore, we can apply the rule E-T-letin to derive $\Omega'_0 \xrightarrow{\lambda'} \Omega'_1$ where $\lambda' = \epsilon$ and $\Omega_1 \stackrel{\text{def}}{=} C, H', \bar{B}' \cdot B' \cup x \mapsto v' : \sigma' \triangleright s$. Hence, $\Omega_1 \approx \Omega'_1$ follows from $\Omega_0 \approx \Omega'_0$ and $v : \sigma \approx v' : \sigma'$ whereas $\lambda = \lambda'$ follows from $\lambda = \epsilon$ and $\lambda' = \epsilon$.
- E-T-write:** Then, $\Omega_0 \stackrel{\text{def}}{=} C, H_0, \bar{B} \cdot B \triangleright e_1 := e_2, \lambda = \text{write}(|n| \mapsto v)^{\sigma_1 \sqcup \sigma_2}, B \triangleright e_1 \downarrow n : \sigma_1, B \triangleright e_2 \downarrow v : \sigma_2, H_0 = H_2; |n| \mapsto v_0 : \sigma_0; H_3, H_1 = H_2; |n| \mapsto v : S; H_3$, and $\Omega_1 \stackrel{\text{def}}{=} C, H_1, \bar{B} \cdot B \triangleright \text{skip}$.
 From $\Omega_0 \approx \Omega'_0, \Omega'_0 \stackrel{\text{def}}{=} C, H'_0, \bar{B}' \cdot B' \triangleright e_1 := e_2$ where $H_0 \approx H'_0, \bar{B} \approx \bar{B}'$, and $B \approx B'$. From $H_0 \approx H'_0$ and $H_0 = H_2; |n| \mapsto v_0 : \sigma_0; H_3$, it follows that there are H'_2, H'_3 such that $H'_0 = H'_2; |n| \mapsto v'_0 : \sigma_0; H'_3$. From $B \approx B', B \triangleright e_1 \downarrow n : \sigma_1, B \triangleright e_2 \downarrow v : \sigma_2$, and lemma L.15, we get $B' \triangleright e_1 \downarrow n' : \sigma'_1$ and $B' \triangleright e_2 \downarrow v' : \sigma'_2$ such that $n : \sigma_1 \approx n' : \sigma'_1$ and $v : \sigma_2 \approx v' : \sigma'_2$. From $\vdash \lambda : \text{safe}$, we get $\sigma_1 \sqcup \sigma_2 = S$ and therefore $\sigma_1 = S$ and $\sigma_2 = S$. From $\sigma_1 = S, \sigma_2 = S, n : \sigma_1 \approx n' : \sigma'_1$, and $v : \sigma_2 \approx v' : \sigma'_2$, we therefore get that $n : \sigma_1 = n' : \sigma'_1$, and $v : \sigma_2 = v' : \sigma'_2$. Therefore, we can apply the rule E-T-write to derive $\Omega'_0 \xrightarrow{\lambda'} \Omega'_1$ where $\lambda' = \text{write}(|n'| \mapsto v')^{\sigma'_1 \sqcup \sigma'_2}, H'_1 = H'_2; |n| \mapsto v' : S; H'_3, \Omega_1 \stackrel{\text{def}}{=} C, H'_1, \bar{B}' \cdot B' \triangleright \text{skip}$.
 From $n : \sigma_1 = n' : \sigma'_1, v : \sigma_2 = v' : \sigma'_2, \lambda' = \text{write}(|n'| \mapsto v')^{\sigma'_1 \sqcup \sigma'_2}$, and $\lambda = \text{write}(|n| \mapsto v)^{\sigma_1 \sqcup \sigma_2}$, we immediately get that $\lambda = \lambda'$.
 Finally, $\Omega_1 \approx \Omega'_1$ follows from $\Omega_0 \approx \Omega'_0, \Omega_1 \stackrel{\text{def}}{=} C, H'_1, \bar{B}' \cdot B' \triangleright \text{skip}, \Omega'_1 \stackrel{\text{def}}{=} C, H'_1, \bar{B}' \cdot B' \triangleright \text{skip}, H_1 = H_2; |n| \mapsto v : S; H_3, H'_1 = H'_2; |n'| \mapsto v' : S; H'_3, n : \sigma_1 = n' : \sigma'_1, v : \sigma_2 = v' : \sigma'_2, \bar{B} \approx \bar{B}'$, and $B \approx B'$.
- E-T-read:** Then, $\Omega_0 \stackrel{\text{def}}{=} C, H, \bar{B} \cdot B \triangleright \text{let } x = \text{rd } e \text{ in } s, B \triangleright e \downarrow n : \sigma_1, H = H_1; |n| \mapsto v : \sigma_0; H_2, \lambda = \text{read}(|n|)^{\sigma_1}$, and $\Omega_1 \stackrel{\text{def}}{=} C, H, \bar{B} \cdot B \cup x \mapsto v : \sigma_0 \triangleright s$.
 From $B \approx B', B \triangleright e \downarrow n : \sigma_1$, and lemma L.15, we get $B' \triangleright e \downarrow n' : \sigma'_1$ and $n : \sigma_1 \approx n' : \sigma'_1$. From $\vdash \lambda : \text{safe}$, we have that $\sigma_1 = S$.
 From $\sigma_1 = S$ and $n : \sigma_1 \approx n' : \sigma'_1$, we get $n : \sigma_1 = n' : \sigma'_1$.
 From $\Omega_0 \approx \Omega'_0$ and $n : \sigma_1 = n' : \sigma'_1$, we get that $\Omega'_0 \stackrel{\text{def}}{=} C, H', \bar{B}' \cdot B' \triangleright \text{let } x = \text{rd } e \text{ in } s$ where $H' = H'_1; |n| \mapsto v' : \sigma'_0; H'_2, H \approx H', \bar{B} \approx \bar{B}', B \approx B'$, and $v : \sigma_0 \approx v' : \sigma'_0$.
 Therefore, we can apply the rule E-T-read to derive $\Omega'_0 \xrightarrow{\lambda'} \Omega'_1$ where $\lambda' = \text{read}(|n'|)^{\sigma'_1}$, and $\Omega_1 \stackrel{\text{def}}{=} C, H', \bar{B}' \cdot B' \cup x \mapsto v' : \sigma'_0 \triangleright \text{skip}$.
 Hence, $\Omega_1 \approx \Omega'_1$ and $\lambda = \lambda'$ follow from $H \approx H', \bar{B} \approx \bar{B}', B \approx B', v : \sigma_0 \approx v' : \sigma'_0$, and $n : \sigma_1 = n' : \sigma'_1$.
- E-T-write-prv:** Then, we have that $\Omega_0 \stackrel{\text{def}}{=} C, H, \bar{B} \cdot B \triangleright e :=_{\text{pr}} e', B \triangleright e \downarrow n : \sigma_0, B \triangleright e' \downarrow v : \sigma_1, \lambda = \text{write}(-|n|)^{\sigma_0}, H = H_2; -|n| \mapsto v_0 : \sigma_2; H_3, H_1 = H_2; -|n| \mapsto v : \sigma_1; H_3$, and $\Omega_1 \stackrel{\text{def}}{=} C, H_1, \bar{B} \cdot B \triangleright \text{skip}$.
 From $\Omega_0 \approx \Omega'_0$, we get that $\Omega'_0 \stackrel{\text{def}}{=} C, H', \bar{B}' \cdot B' \triangleright e :=_{\text{pr}} e', H' = H'_2; -|n| \mapsto v'_0 : \sigma'_2; H'_3, H \approx H', \bar{B} \approx \bar{B}',$ and $B \approx B'$.
 From $\vdash \lambda : \text{safe}$, we get that $\sigma_0 = S$. From $\Omega_0 \approx \Omega'_0, B \triangleright e \downarrow n : \sigma_0$, and lemma L.15, we get $B \triangleright e \downarrow n' : \sigma'_0$ and $n : \sigma_0 \approx n' : \sigma'_0$. From this and $\sigma_0 = S$, we get $n : \sigma_0 = n' : \sigma'_0$.
 From $\Omega_0 \approx \Omega'_0, B \triangleright e' \downarrow v : \sigma_1$, and lemma L.15, we get $B \triangleright e' \downarrow v' : \sigma'_1$ and $v : \sigma_1 \approx v' : \sigma'_1$.
 Hence, we can apply the rule E-T-write-prv to derive $\Omega'_0 \xrightarrow{\lambda'} \Omega'_1$ where $\lambda = \text{write}(-|n'|)^{\sigma'_0}$ and $\Omega_1 \stackrel{\text{def}}{=} C, H', \bar{B}' \cdot B' \triangleright \text{skip}$ where $H'_1 = H'_2; -|n'| \mapsto v' : \sigma'_1; H_3$. Therefore, $\lambda = \lambda'$ follows from $n : \sigma_0 = n' : \sigma'_0$ and $\Omega_1 \approx \Omega'_1$ follows from $\Omega_0 \approx \Omega'_0, n : \sigma_0 = n' : \sigma'_0$, and $v : \sigma_1 \approx v' : \sigma'_1$.
- E-T-read-prv:** Then, $\Omega_0 \stackrel{\text{def}}{=} C, H, \bar{B} \cdot B \triangleright \text{let } x = \text{rd}_{\text{pr}} e \text{ in } s, B \triangleright e \downarrow n : \sigma_0, H = H_1; -|n| \mapsto v : \sigma_1; H_2, \lambda = \text{read}(-|n|)^{\sigma_0}$, and $\Omega_1 \stackrel{\text{def}}{=} C, H, \bar{B} \cdot B \cup x \mapsto v : U \triangleright s$.
 From $\Omega_0 \approx \Omega'_0$, we get that $\Omega'_0 \stackrel{\text{def}}{=} C, H', \bar{B}' \cdot B' \triangleright \text{let } x = \text{rd}_{\text{pr}} e \text{ in } s$ where $H \approx H', \bar{B} \approx \bar{B}'$, and $B \approx B'$. From $H \approx H'$ and $H = H_1; -|n| \mapsto v : \sigma_1; H_2$, we get that $H = H'_1; -|n| \mapsto v' : \sigma'_1; H'_2$ and $v : \sigma_1 \approx v' : \sigma'_1$. Moreover, from $B \triangleright e \downarrow n : \sigma_0, B \approx B'$, and lemma L.15, we get that $B' \triangleright e \downarrow n' : \sigma'_0$ and $n : \sigma_0 \approx n' : \sigma'_0$.
 From $\vdash \lambda : \text{safe}$, we get that $\sigma_0 = S$. From this and $n : \sigma_0 \approx n' : \sigma'_0$, we get $n : \sigma_0 = n' : \sigma'_0$.
 We can apply the rule E-T-read-prv to derive $\Omega'_0 \xrightarrow{\lambda'} \Omega'_1$ where $\lambda = \text{read}(-|n'|)^{\sigma'_0}$, and $\Omega_1 \stackrel{\text{def}}{=} C, H', \bar{B}' \cdot B' \cup x \mapsto v' : U \triangleright \text{skip}$.
 Therefore, $\lambda = \lambda'$ follows from $n : \sigma_0 = n' : \sigma'_0$, whereas $\Omega_1 \approx \Omega'_1$ follows from $\Omega_0 \approx \Omega'_0$ and the fact that the values assigned to x has tag U .
- E-T-call-internal:** Then, $\Omega_0 \stackrel{\text{def}}{=} C, H, \bar{B} \cdot B \triangleright (\text{call } f \text{ e})_{\bar{f}}, \lambda = \epsilon, C.\text{intfs} \vdash f, f' : \text{internal}, \bar{f} = \bar{f}'; f', f(x) \mapsto s; \text{return}; \in C.\text{funs}, B \triangleright e \downarrow v : \sigma$, and $\Omega_1 \stackrel{\text{def}}{=} C, H, \bar{B} \cdot B \cdot x \mapsto v : \sigma \triangleright (s; \text{return});_{\bar{f}, f}$. From $\Omega_0 \approx \Omega'_0$, we get that $\Omega'_0 \stackrel{\text{def}}{=} C, H', \bar{B}' \cdot B' \triangleright (\text{call } f \text{ e})_{\bar{f}}$ where $H \approx H', \bar{B} \approx \bar{B}'$, and $B \approx B'$. From $B \approx B', B \triangleright e \downarrow v : \sigma$, and lemma L.15, we get that $B \triangleright e \downarrow v' : \sigma'$ and $v : \sigma \approx v' : \sigma'$. Therefore, we can apply the rule E-T-call-internal to derive $\Omega'_0 \xrightarrow{\lambda'} \Omega'_1$ where $\lambda' = \epsilon$, and $\Omega_1 \stackrel{\text{def}}{=} C, H', \bar{B}' \cdot B' \cdot x \mapsto v' : \sigma' \triangleright (s; \text{return});_{\bar{f}, f}$. Hence, $\lambda = \lambda'$ (which follows from $\lambda = \epsilon$ and $\lambda' = \epsilon$), and $\Omega_1 \approx \Omega'_1$ (which follows from $\Omega_0 \approx \Omega'_0$ and $v : \sigma \approx v' : \sigma'$).
- E-T-callback:** Then, $\Omega_0 \stackrel{\text{def}}{=} C, H, \bar{B} \cdot B \triangleright (\text{call } f \text{ e})_{\bar{f}}, \lambda = \text{call } f \text{ v! } \bar{f}, \bar{f} = \bar{f}'; f', f(x) \mapsto s; \text{return}; \in C.\text{funs}, C.\text{intfs} \vdash f', f : \text{out}, B \triangleright e \downarrow v : \sigma$, and $\Omega_1 = C, H, \bar{B} \cdot B \cdot x \mapsto v : \sigma \triangleright (s; \text{return});_{\bar{f}, f}$. From $\Omega_0 \approx \Omega'_0$, we get that $\Omega'_0 \stackrel{\text{def}}{=} C, H', \bar{B}' \cdot B' \triangleright (\text{call } f \text{ e})_{\bar{f}}$ where

$H \approx H', \bar{B} \approx \bar{B}'$, and $B \approx B'$. From $B \approx B'$, $B \triangleright e \downarrow v : \sigma$, and lemma L.15, we get that $B \triangleright e \downarrow v' : \sigma'$ and $v : \sigma \approx v' : \sigma'$. Therefore, we can apply the rule E-T-callback to derive $\Omega'_0 \xrightarrow{\lambda'} \Omega'_1$ where $\lambda' = \text{call } f \ v!^{\sigma'}$ and $\Omega'_1 \stackrel{\text{def}}{=} C, H', \bar{B}' \cdot B' \cdot x \mapsto v' : \sigma' \triangleright (s; \text{return}); \bar{f}'$. From $\vdash \lambda : \text{safe}$, we get that $\sigma = S$. From this and $v : \sigma \approx v' : \sigma'$, we get that $v : \sigma = v' : \sigma'$. Hence, $\lambda = \lambda'$ (which follows from $\lambda = \text{call } f \ v!^{\sigma}$, $\lambda' = \text{call } f \ v!^{\sigma'}$, and $v : \sigma = v' : \sigma'$), and $\Omega_1 \approx \Omega'_1$ (which follows from $\Omega_0 \approx \Omega'_0$ and $v : \sigma \approx v' : \sigma'$).

E-T-call: The proof of this case is similar to that of the case E-T-callback.

E-T-ret-internal: Then, $\Omega_0 \stackrel{\text{def}}{=} C, H, \bar{B} \cdot B \triangleright (\text{return}); \bar{f}'$, $\bar{f}' = \bar{f}''; f'$, $C.\text{intfs} \vdash f, f' : \text{internal}$, $\lambda = \epsilon$, and $\Omega_1 = C, H, \bar{B} \triangleright (\text{skip}); \bar{f}'$.

From $\Omega_0 \approx \Omega'_0$, we get that $\Omega'_0 \stackrel{\text{def}}{=} C, H', \bar{B}' \cdot B' \triangleright (\text{return}); \bar{f}'$ where $H \approx H'$, $\bar{B} \approx \bar{B}'$, and $B \approx B'$. Therefore, we can apply the rule

E-T-ret-internal to derive $\Omega'_0 \xrightarrow{\lambda'} \Omega'_1$ where $\lambda' = \epsilon$ and $\Omega'_1 \stackrel{\text{def}}{=} C, H', \bar{B}' \triangleright (\text{skip}); \bar{f}'$. Hence, $\lambda = \lambda'$ (which follows from $\lambda = \epsilon$ and $\lambda' = \epsilon$) and $\Omega_1 \approx \Omega'_1$ (which follows from $H \approx H'$ and $\bar{B} \approx \bar{B}'$).

E-T-retback: Then, $\Omega_0 \stackrel{\text{def}}{=} C, H, \bar{B} \cdot B \triangleright (\text{return}); \bar{f}'$, $\bar{f}' = \bar{f}''; f'$, $C.\text{intfs} \vdash f, f' : \text{internal}$, $\lambda = \text{ret}^?S$, and $\Omega_1 = C, H, \bar{B} \triangleright (\text{skip}); \bar{f}'$.

From $\Omega_0 \approx \Omega'_0$, we get that $\Omega'_0 \stackrel{\text{def}}{=} C, H', \bar{B}' \cdot B' \triangleright (\text{return}); \bar{f}'$ where $H \approx H'$, $\bar{B} \approx \bar{B}'$, and $B \approx B'$. Therefore, we can apply the rule

E-T-ret-retback to derive $\Omega'_0 \xrightarrow{\lambda'} \Omega'_1$ where $\lambda' = \text{ret}^?S$ and $\Omega'_1 \stackrel{\text{def}}{=} C, H', \bar{B}' \triangleright (\text{skip}); \bar{f}'$. Hence, $\lambda = \lambda'$ (which follows from $\lambda = \text{ret}^?S$ and $\lambda' = \text{ret}^?S$) and $\Omega_1 \approx \Omega'_1$ (which follows from $H \approx H'$ and $\bar{B} \approx \bar{B}'$).

E-T-return: The proof of this case is similar to that of E-T-retback.

E-T-lfence: Then, $\Omega_0 \stackrel{\text{def}}{=} C, H, \bar{B} \triangleright \text{lfence}$, $\lambda = \epsilon$, and $\Omega_1 \stackrel{\text{def}}{=} C, H, \bar{B} \triangleright \text{skip}$. From $\Omega_0 \approx \Omega'_0$, we get that $\Omega'_0 \stackrel{\text{def}}{=} C, H', \bar{B}' \triangleright \text{lfence}$ where

$H' \approx H$ and $\bar{B} \approx \bar{B}'$. Hence, we can apply the E-T-lfence rule to derive $\Omega'_0 \xrightarrow{\lambda'} \Omega'_1$ where $\lambda' = \epsilon$ and $\Omega'_1 \stackrel{\text{def}}{=} C, H', \bar{B}' \triangleright \text{skip}$. Hence, $\Omega_1 \approx \Omega'_1$ and $\lambda = \lambda'$ hold.

E-T-cmove-true: Then, $\Omega_0 \stackrel{\text{def}}{=} C, H, \bar{B} \cdot B \triangleright \text{let } x = e_0 \text{ (if } e_1 \text{) in } s, x \in \text{dom}(B), \lambda = \epsilon, \Omega_1 \stackrel{\text{def}}{=} C, H, \bar{B} \cdot B \cup x \mapsto v_0 : \sigma \triangleright s, B \triangleright e_0 \downarrow v_0 : \sigma_0,$

$B \triangleright e_1 \downarrow 0 : \sigma_1$, and $\sigma = \sigma_0 \sqcup \sigma_1$. From $\Omega_0 \approx \Omega'_0$, we have that $\Omega'_0 \stackrel{\text{def}}{=} C, H', \bar{B}' \cdot B' \triangleright \text{let } x = e_0 \text{ (if } e_1 \text{) in } s$ where $H \approx H'$, $\bar{B} \cdot B \approx \bar{B}' \cdot B'$, and $B \approx B'$. From $B \approx B'$, $B \triangleright e_0 \downarrow v_0 : \sigma_0$, $B \triangleright e_1 \downarrow 0 : \sigma_1$, and lemma L.15, we get that $B' \triangleright e_0 \downarrow v'_0 : \sigma'_0$, $B' \triangleright e_1 \downarrow v'_1 : \sigma'_1$ where $v_0 : \sigma_0 \approx v'_0 : \sigma'_0$ and $0 : \sigma_1 \approx v'_1 : \sigma'_1$. There are two cases:

$\sigma_1 = S$: Then, $\sigma'_1 = S$ and $v'_1 = 0$. Hence, we can apply the E-T-cmove-true rule to derive $\Omega'_0 \xrightarrow{\lambda'} \Omega'_1$ where $\Omega_1 \stackrel{\text{def}}{=} C, H', \bar{B}' \cdot B' \cup x \mapsto v'_0 : \sigma' \triangleright s$, and $\sigma' = \sigma'_0 \sqcup \sigma'_1$. Observe that $\Omega_1 \approx \Omega'_1$ immediately follows from $\Omega_0 \approx \Omega'_0$ and $v_0 : \sigma_0 \approx v'_0 : \sigma'_0$. Hence, $\Omega_1 \approx \Omega'_1$ and $\lambda = \lambda'$ hold.

$\sigma_1 = U$: Then, $\sigma'_1 = U$ holds as well. There are two cases:

$v'_1 = 0$: Then, we can apply the E-T-cmove-true rule to derive $\Omega'_0 \xrightarrow{\lambda'} \Omega'_1$ where $\Omega_1 \stackrel{\text{def}}{=} C, H', \bar{B}' \cdot B' \cup x \mapsto v'_0 : \sigma' \triangleright s$, and $\sigma' = \sigma'_0 \sqcup \sigma'_1$. Observe that $\Omega_1 \approx \Omega'_1$ immediately follows from $\Omega_0 \approx \Omega'_0$ and $\sigma = \sigma' = U$. Hence, $\Omega_1 \approx \Omega'_1$ and $\lambda = \lambda'$ hold.

$v'_1 > 0$: Then, we can apply the E-T-cmove-false rule to derive $\Omega'_0 \xrightarrow{\lambda'} \Omega'_1$ where $\Omega_1 \stackrel{\text{def}}{=} C, H', \bar{B}' \cdot B' \cup x \mapsto v' : \sigma' \triangleright s, B(x) = v' : \sigma'_0$, and $\sigma' = \sigma_0 \sqcup \sigma'_1$. Observe that $\Omega_1 \approx \Omega'_1$ immediately follows from $\Omega_0 \approx \Omega'_0$ and $\sigma = \sigma' = U$. Hence, $\Omega_1 \approx \Omega'_1$ and $\lambda = \lambda'$ hold.

E-T-cmove-false: The proof of this case is similar to that of the E-T-cmove-true case.

Induction step: Then, $\Omega_0 \xrightarrow{\lambda} \Omega_1$ using the E-T-step rule. Then, $\Omega_0 \stackrel{\text{def}}{=} C, H, \bar{B} \triangleright s; s'', C, H, \bar{B} \triangleright s \xrightarrow{\lambda} C_1, H_1, \bar{B}_1 \triangleright s_1$, and $\Omega_1 = C, H_1, \bar{B}_1 \triangleright s_1; s''$.

From $\Omega_0 \approx \Omega'_0$, we get that $\Omega_0 \stackrel{\text{def}}{=} C, H', \bar{B}' \triangleright s; s''$ where $H \approx H'$ and $\bar{B} \approx \bar{B}'$. From the induction hypothesis, $\vdash \lambda : \text{safe}$, and $\Omega_0 \approx \Omega'_0$,

we get that $C, H', \bar{B}' \triangleright s \xrightarrow{\lambda'} C, H'_1, \bar{B}'_1 \triangleright s_1$ such that $H_1 \approx H'_1$, $\bar{B}_1 \approx \bar{B}'_1$, and $\lambda = \lambda'$. Therefore, we can apply the E-L-step rule to

derive $\Omega'_0 \xrightarrow{\lambda'} \Omega'_1$ where $\Omega_1 \stackrel{\text{def}}{=} C, H'_1, \bar{B}'_1 \triangleright s_1; s''$. Observe that $\Omega_1 \approx \Omega'_1$ and $\lambda = \lambda'$ hold.

This concludes the proof of our lemma. \square

LEMMA L.12 (STEPS OF \rightarrow WITHOUT OBSERVATIONS PRESERVE LOW-EQUIVALENCE).

$$\begin{aligned} & \forall \Omega_0, \Omega'_0, \lambda, \Omega_1. \text{ if } \Omega_0 \xrightarrow{\epsilon} \Omega_1, \Omega_0 \approx \Omega'_0 \\ & \text{ then } \exists \Omega'_1. \Omega'_0 \xrightarrow{\epsilon'} \Omega'_1, \Omega_1 \approx \Omega'_1 \end{aligned}$$

PROOF. Special case of lemma L.11 since $\vdash \epsilon : \text{safe}$ is always satisfied. \square

LEMMA L.13 (STEPS OF \rightarrow THAT AGREE ON OBSERVATION AND LOW-EQUIVALENCE CANNOT DISAGREE ON LABEL).

$$\begin{aligned} & \forall \Omega_0, \Omega'_0, \lambda, \sigma, \sigma', \Omega_1, \Omega'_1. \text{ if } \Omega_0 \xrightarrow{\lambda^\sigma} \Omega_1, \Omega'_0 \xrightarrow{\lambda^{\sigma'}} \Omega'_1, \Omega_0 \approx \Omega'_0 \\ & \text{ then } \sigma = \sigma' \end{aligned}$$

PROOF. By structural induction on the rules defining \rightarrow . It follows from (1) there are no two rules producing the same observation, (2) labels are always derived by computation over bindings and heaps which are low-equivalent from $\Omega_0 \approx \Omega'_0$, and (3) computation over low-equivalent bindings produce the same labels (lemma L.14). \square

L.1.6 Bindings.

LEMMA L.14 (LOW-EQUIVALENT BINDINGS PRODUCE LOW-EQUIVALENT RESULTS - 1).

$$\begin{aligned} & \forall \mathbf{B}, \mathbf{B}', \mathbf{e}, \mathbf{v}, \mathbf{v}', \sigma, \sigma'. \\ & \text{if } \mathbf{B} \approx \mathbf{B}', \mathbf{B} \triangleright \mathbf{e} \downarrow \mathbf{v} : \sigma, \mathbf{B}' \triangleright \mathbf{e} \downarrow \mathbf{v}' : \sigma' \\ & \text{then } \mathbf{v} : \sigma \approx \mathbf{v}' : \sigma' \end{aligned}$$

PROOF. Let $\mathbf{B}, \mathbf{B}', \mathbf{e}, \mathbf{v}, \mathbf{v}', \sigma$, and σ' be such that $\mathbf{B} \approx \mathbf{B}'$, $\mathbf{B} \triangleright \mathbf{e} \downarrow \mathbf{v} : \sigma$, $\mathbf{B}' \triangleright \mathbf{e} \downarrow \mathbf{v}' : \sigma'$. From $\mathbf{B} \approx \mathbf{B}'$, $\mathbf{B} \triangleright \mathbf{e} \downarrow \mathbf{v} : \sigma$, and lemma L.15, there are \mathbf{v}'' , σ'' such that $\mathbf{B}' \triangleright \mathbf{e} \downarrow \mathbf{v}'' : \sigma''$ and $\mathbf{v} : \sigma \approx \mathbf{v}'' : \sigma''$. Since \downarrow is deterministic, we immediately get that $\mathbf{v}'' : \sigma'' = \mathbf{v}' : \sigma'$. Hence, $\mathbf{v} : \sigma \approx \mathbf{v}' : \sigma'$. \square

LEMMA L.15 (LOW-EQUIVALENT BINDINGS PRODUCE LOW-EQUIVALENT RESULTS - 2).

$$\begin{aligned} & \forall \mathbf{B}, \mathbf{B}', \mathbf{e}, \mathbf{v}, \sigma. \\ & \text{if } \mathbf{B} \approx \mathbf{B}', \mathbf{B} \triangleright \mathbf{e} \downarrow \mathbf{v} : \sigma, \\ & \text{then } \exists \mathbf{v}', \sigma'. \mathbf{B}' \triangleright \mathbf{e} \downarrow \mathbf{v}' : \sigma', \mathbf{v} : \sigma \approx \mathbf{v}' : \sigma' \end{aligned}$$

PROOF. Let $\mathbf{B}, \mathbf{B}', \mathbf{e}, \mathbf{v}, \sigma$ be such that $\mathbf{B} \approx \mathbf{B}'$ and $\mathbf{B} \triangleright \mathbf{e} \downarrow \mathbf{v} : \sigma$. We show that the lemma holds by structural induction on the rule used to derive $\mathbf{B} \triangleright \mathbf{e} \downarrow \mathbf{v} : \sigma$:

Base case: There are two cases based on the rule used to derive $\mathbf{B} \triangleright \mathbf{e} \downarrow \mathbf{v} : \sigma$:

E-T-val: Then, $\mathbf{e} = \mathbf{v}$ and $\sigma = \mathbf{S}$. Hence, we can derive $\mathbf{B}' \triangleright \mathbf{e} \downarrow \mathbf{v}' : \sigma'$ using the E-T-val rule, by picking $\mathbf{v}' = \mathbf{v}$ and $\sigma' = \mathbf{S}$. Therefore, $\mathbf{v} : \sigma \approx \mathbf{v}' : \sigma'$.

E-T-var: Then, $\mathbf{e} = \mathbf{x}$ and $\mathbf{B}(\mathbf{x}) = \mathbf{v} : \sigma$. From $\mathbf{B}(\mathbf{x}) = \mathbf{v} : \sigma$, we get that $\vdash \mathbf{B}(\mathbf{x}) : \text{def}$. From $\mathbf{B} \approx \mathbf{B}'$, $\vdash \mathbf{B}(\mathbf{x}) : \text{def}$, and $\mathbf{B}(\mathbf{x}) = \mathbf{v} : \sigma$, we get $\mathbf{B}'(\mathbf{x}) = \mathbf{v}' : \sigma'$ and $\mathbf{v} : \sigma \approx \mathbf{v}' : \sigma'$. Hence, we can derive $\mathbf{B}' \triangleright \mathbf{e} \downarrow \mathbf{v}' : \sigma'$ using the E-T-var rule and $\mathbf{v} : \sigma \approx \mathbf{v}' : \sigma'$.

Induction step: There are two cases based on the rule used to derive $\mathbf{B} \triangleright \mathbf{e} \downarrow \mathbf{v} : \sigma$:

E-T-op: Then, $\mathbf{e} = \mathbf{e}_1 \oplus \mathbf{e}_2$, $\mathbf{B} \triangleright \mathbf{e}_1 \downarrow \mathbf{n}_1 : \sigma_1$, $\mathbf{B} \triangleright \mathbf{e}_2 \downarrow \mathbf{n}_2 : \sigma_2$, $\mathbf{v} = [\mathbf{n}_1 \oplus \mathbf{n}_2]$, and $\sigma = \sigma_1 \sqcup \sigma_2$. From $\mathbf{B} \approx \mathbf{B}'$, $\mathbf{B} \triangleright \mathbf{e}_1 \downarrow \mathbf{n}_1 : \sigma_1$, $\mathbf{B} \triangleright \mathbf{e}_2 \downarrow \mathbf{n}_2 : \sigma_2$, and the induction hypothesis, we get that there are $\mathbf{n}'_1, \mathbf{n}'_2, \sigma'_1, \sigma'_2$ such that $\mathbf{B}' \triangleright \mathbf{e}_1 \downarrow \mathbf{n}'_1 : \sigma'_1$, $\mathbf{B}' \triangleright \mathbf{e}_2 \downarrow \mathbf{n}'_2 : \sigma'_2$, $\mathbf{n}'_1 : \sigma'_1 \approx \mathbf{n}_1 : \sigma_1$ and $\mathbf{n}_2 : \sigma_2 \approx \mathbf{n}'_2 : \sigma'_2$. Hence, we can apply the E-T-rule to derive $\mathbf{B}' \triangleright \mathbf{e} \downarrow \mathbf{v}' : \sigma'$, where $\mathbf{v}' = [\mathbf{n}'_1 \oplus \mathbf{n}'_2]$, and $\sigma' = \sigma'_1 \sqcup \sigma'_2$. There are two cases:

$\sigma = \mathbf{S}$: Then, $\sigma_1 = \mathbf{S}$ and $\sigma_2 = \mathbf{S}$. From this, $\mathbf{n}'_1 : \sigma'_1 \approx \mathbf{n}_1 : \sigma_1$, and $\mathbf{n}_2 : \sigma_2 \approx \mathbf{n}'_2 : \sigma'_2$, we get $\mathbf{v}_1 = \mathbf{v}'_1$, $\mathbf{v}_2 = \mathbf{v}'_2$, $\sigma'_1 = \mathbf{S}$, and $\sigma'_2 = \mathbf{S}$. Hence, $[\mathbf{n}_1 \oplus \mathbf{n}_2] = [\mathbf{n}'_1 \oplus \mathbf{n}'_2]$ and $\sigma' = \mathbf{S}$. Thus, $\mathbf{v} : \sigma = \mathbf{v}' : \sigma'$ and thus $\mathbf{v} : \sigma \approx \mathbf{v}' : \sigma'$.

$\sigma = \mathbf{U}$: Then, $\sigma_1 = \mathbf{U} \vee \sigma_2 = \mathbf{U}$. From this, $\mathbf{n}'_1 : \sigma'_1 \approx \mathbf{n}_1 : \sigma_1$, and $\mathbf{n}_2 : \sigma_2 \approx \mathbf{n}'_2 : \sigma'_2$, we also get that $\sigma'_1 = \mathbf{U} \vee \sigma'_2 = \mathbf{U}$. Hence, $\sigma' = \mathbf{U}$ as well. Therefore, $\mathbf{v} : \sigma \approx \mathbf{v}' : \sigma'$ holds.

E-T-comparison: The proof of this case is similar to the case for the E-T-op rule.

This completes the proof. \square

L.1.7 Non-speculative projection $\cdot \uparrow_{nse}$.

LEMMA L.16 (PROPERTIES OF NON-SPECULATIVE PROJECTION $\cdot \uparrow_{nse}$).

$$\begin{aligned} & \forall \mathbf{P}, \Sigma, \Sigma', \lambda, \lambda', \mathbf{n}. \text{if } (\mathbf{0}, \Omega_0(\mathbf{P})) \xrightarrow{\lambda} (\mathbf{n}, \Sigma), \Sigma \xrightarrow{\lambda'} \Sigma', \vdash \Sigma' : \text{safe} \\ & \text{then } (\lambda \cdot \lambda') \uparrow_{nse} = \lambda \uparrow_{nse} \cdot \lambda' \\ & \text{if } (\mathbf{0}, \Omega_0(\mathbf{P})) \xrightarrow{\lambda} (\mathbf{n}, \Sigma), \Sigma \xrightarrow{\lambda'} \Sigma', \vdash \Sigma' : \text{unsafe} \\ & \text{then } (\lambda \cdot \lambda') \uparrow_{nse} = \lambda \uparrow_{nse} \end{aligned}$$

PROOF. The lemma follows by inspection of the semantics and of the non-speculative projection. \square

L.1.8 Properties of components.

LEMMA L.17 (ONLY SAFE VALUES IN COMPONENTS).

$$\begin{aligned} & \forall \mathbf{P}, \Sigma, \lambda, \mathbf{n}. \text{if } (\mathbf{0}, \Omega_0(\mathbf{P})) \xrightarrow{\lambda} (\mathbf{n}, \Sigma), \Sigma \stackrel{\text{def}}{=} (\mathbf{w}, (\overline{\Omega, \mathbf{m}, \sigma}) \cdot (\overline{\mathbf{F}, \bar{\mathbf{I}}, \mathbf{H}, \bar{\mathbf{B}}} \triangleright (\mathbf{s})_{\bar{\mathbf{F}}, \mathbf{f}}, \mathbf{n}, \sigma')), \mathbf{f} \in \bar{\mathbf{I}}, \\ & \text{then } \forall \mathbf{x}, \mathbf{v}, \sigma. \mathbf{B}(\mathbf{x}) = \mathbf{v} : \sigma \rightarrow \sigma = \mathbf{S} \end{aligned}$$

PROOF. By induction on n combined with (1) contexts can only write and read information from the public heap which is always labelled S , and (2) E-T-call and E-T-callback rules tag the variable x with S . \square

L.1.9 Weak Variants.

THEOREM L.18 (SS IMPLIES SNI (WEAK)).

$$\forall P \in T^-. \text{ if } \vdash P : SS(T^-), \text{ then } \vdash P : SNI(T^-)$$

PROOF. The proof of this result is similar to the one of Theorem L.2 (SS implies SNI (strong)). The key differences is that all data loaded non-speculatively is tagged as S (rather than U). Therefore, one has to show that executing a non-speculative memory load preserves indistinguishability or results in different non-speculative observations. This immediately follows from the fact that actions generated by non-speculative memory loads disclose both the memory address and the loaded value (so either we have different non-speculative observations or we load the same value from memory and therefore we preserve indistinguishability). \square

L.2 RSS implies RSNI

COROLLARY L.19 (RSS IMPLIES RSNI).

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

PROOF. Let P be an arbitrary program in T such that $\vdash P : RSS$ holds. Let A be an arbitrary context. Then, there are two cases:
 $\vdash A : atk$: From $\vdash P : RSS$, it follows that $\vdash A[P] : SS(T)$. By applying theorem L.2, we have $\vdash A[P] : SNI(T)$. Hence, $\vdash A : atk \Rightarrow \vdash A[P] : SNI(T)$ holds.
 $\not\vdash A : atk$: Then, $\vdash A : atk \Rightarrow \vdash A[P] : SNI(T)$ trivially holds.

Since $\vdash A : atk \Rightarrow \vdash A[P] : SNI(T)$ holds for an arbitrary context A and program P , we have that $\forall A. \vdash A : atk \Rightarrow \vdash A[P] : SNI(T)$ holds as well. Hence, $\vdash P : RSNI(T)$ holds. This completes the proof of our corollary. \square

COROLLARY L.20 (RSS IMPLIES RSNI).

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

PROOF. The proof is similar to the one of corollary L.19 except that we use theorem L.18 instead of theorem L.2. \square

L.3 SNI does not imply SS

THEOREM L.21 (SNI NOT IMPLY SS).

$$\begin{aligned} \exists P \in T. \vdash P : SNI(T) \wedge \not\vdash P : SS(T) \\ \exists P \in T^-. \vdash P : SNI(T^-) \wedge \not\vdash P : SS(T^-) \end{aligned}$$

PROOF. Consider the following program P :

```
ifz (y < size) then
  let xa = rdpr 0 + y in
  let xb = rdpr 4 + xa in
  let temp=xb in skip
else
  let xa = rdpr 0 + y in
  let xb = rdpr 4 + xa in
  let temp=xb in skip
```

The above program clearly satisfies speculative non-interference for, e.g., $w = 10$ since the program leaks the same information under the speculative and non-speculative semantics. However, the program violates speculative safety (see the step-by-step example in appendix H). Additionally, observe that the program satisfy $SNI(T^-)$ but still violates $SS(T^-)$. \square

L.4 RSNI does not imply RSS

COROLLARY L.22 (RSNI NOT IMPLY RSS).

$$\begin{aligned} \exists P \in T. \vdash P : RSNI(T) \wedge \not\vdash P : RSS(T) \\ \exists P \in T^-. \vdash P : RSNI(T^-) \wedge \not\vdash P : RSS(T^-) \end{aligned}$$

PROOF. The counter-example provided in theorem L.21 can be directly ported to the robust setting (by moving the code in the component and having a context simply calling the component). \square

M COMPILER CRITERIA AND THEIR IMPLICATIONS

M.1 Strong Criteria for Secure Compilers

Definition M.1 (Robust Speculative Safety-Preserving Compiler (RSSP)).

$$\vdash \llbracket \cdot \rrbracket : \text{RSSP}^+ \stackrel{\text{def}}{=} \forall P. \text{ if } \vdash P : \text{RSS}(\mathbf{L}) \text{ then } \vdash \llbracket P \rrbracket : \text{RSS}(\mathbf{T})$$

This gets expanded to:

$$\begin{aligned} & \forall P. \text{ if } \forall A. \forall \bar{\lambda}^\sigma \in \text{Beh}(A \llbracket P \rrbracket). \forall \alpha^\sigma \in \bar{\lambda}^\sigma. \sigma \equiv S \\ & \text{ then } \forall A. \forall \bar{\lambda}^\sigma \in \text{Beh}(A \llbracket \llbracket P \rrbracket \rrbracket). \forall \alpha^\sigma \in \bar{\lambda}^\sigma. \sigma \equiv S \end{aligned}$$

We say that a source and a target trace are related (\approx) if the latter contains the source trace plus interleavings of only safe actions. The trace relation relies on a relation on actions which in turn relies on a relation on values and heaps.

The last two are compiler-dependent, so they are presented later, for lfence in Appendix O.1.3 and for slh in Appendix Q.2.1.

Trace relation \approx				
(Trace-Relation) $\emptyset \approx \emptyset$	(Trace-Relation-Same-Act) $\frac{\bar{\lambda}^\sigma \approx \bar{\lambda}^\sigma \quad \alpha^\sigma \stackrel{A}{\approx} \alpha^\sigma}{\bar{\lambda}^\sigma \cdot \alpha^\sigma \approx \bar{\lambda}^\sigma \cdot \alpha^\sigma}$	(Trace-Relation-Same-Heap) $\frac{\bar{\lambda}^\sigma \approx \bar{\lambda}^\sigma \quad \delta^\sigma \stackrel{A}{\approx} \delta^\sigma}{\bar{\lambda}^\sigma \cdot \delta^\sigma \approx \bar{\lambda}^\sigma \cdot \delta^\sigma}$	(Trace-Relation-Safe-Act) $\frac{\bar{\lambda}^\sigma \approx \bar{\lambda}^\sigma \quad \epsilon \stackrel{A}{\approx} \alpha^S}{\bar{\lambda}^\sigma \approx \bar{\lambda}^\sigma \cdot \alpha^S}$	(Trace-Relation-Safe-Heap) $\frac{\bar{\lambda}^\sigma \approx \bar{\lambda}^\sigma \quad \epsilon \stackrel{A}{\approx} \delta^S}{\bar{\lambda}^\sigma \approx \bar{\lambda}^\sigma \cdot \delta^S}$
(Trace-Relation-Rollback) $\frac{\bar{\lambda}^\sigma \approx \bar{\lambda}^\sigma \quad \epsilon \stackrel{A}{\approx} \text{rlb}^\sigma}{\bar{\lambda}^\sigma \approx \bar{\lambda}^\sigma \cdot \text{rlb}^\sigma}$				
Action relation $\stackrel{A}{\approx}$				
(Action Relation - call) $\frac{f \equiv f \quad v \stackrel{V}{\approx} v \quad \sigma \equiv \sigma}{\text{call } f \ v?^\sigma \stackrel{A}{\approx} \text{call } f \ v?^\sigma}$		(Action Relation - return) $\frac{}{\text{ret}!^S \stackrel{A}{\approx} \text{ret}!^S}$		(Action Relation - callback) $\frac{f \equiv f \quad v \stackrel{V}{\approx} v \quad \sigma \equiv \sigma}{\text{call } f \ v!^\sigma \stackrel{A}{\approx} \text{call } f \ v!^\sigma}$
(Action Relation - returnback) $\frac{}{\text{ret}?^S \stackrel{A}{\approx} \text{ret}?^S}$	(Action Relation - read) $\frac{n \stackrel{V}{\approx} n \quad \sigma \equiv \sigma}{\text{read}(n)^\sigma \stackrel{A}{\approx} \text{read}(n)^\sigma}$		(Action Relation - write) $\frac{n \stackrel{V}{\approx} n \quad \sigma \equiv \sigma}{\text{write}(n)^\sigma \stackrel{A}{\approx} \text{write}(n)^\sigma}$	
(Action Relation - write 2) $\frac{n \stackrel{V}{\approx} n \quad v \stackrel{V}{\approx} v \quad \sigma \equiv \sigma}{\text{write}(n \mapsto v)^\sigma \stackrel{A}{\approx} \text{write}(n \mapsto v)^\sigma}$		(Action Relation - if) $\frac{n \stackrel{V}{\approx} n \quad \sigma \equiv \sigma}{\text{if}(n)^\sigma \stackrel{A}{\approx} \text{if}(n)^\sigma}$		
(Action Relation - epsi alpha) $\frac{\sigma \equiv S}{\epsilon \stackrel{A}{\approx} \alpha^\sigma}$		(Action Relation - epsi heap) $\frac{\sigma \equiv S}{\epsilon \stackrel{A}{\approx} \delta^\sigma}$	(Action Relation - rlb) $\frac{\sigma \equiv S}{\epsilon \stackrel{A}{\approx} \text{rlb}^\sigma}$	

Definition M.2 (Robust Speculative Safety Compilation (RSSC)).

$$\begin{aligned} \vdash \llbracket \cdot \rrbracket : \text{RSSC}^+ \stackrel{\text{def}}{=} & \forall P, A, \bar{\lambda}^\sigma, \exists A, \bar{\lambda}^\sigma. \\ & \text{ if } A \llbracket \llbracket P \rrbracket \rrbracket \rightsquigarrow \bar{\lambda}^\sigma \text{ then } A \llbracket P \rrbracket \rightsquigarrow \bar{\lambda}^\sigma \text{ and } \bar{\lambda}^\sigma \approx \bar{\lambda}^\sigma \end{aligned}$$

THEOREM M.3 (RSSC IMPLIES RSSP). (Proof 1) With the relation \approx :

$$\forall \llbracket \cdot \rrbracket \text{ if } \vdash \llbracket \cdot \rrbracket : \text{RSSC}^+ \text{ then } \vdash \llbracket \cdot \rrbracket : \text{RSSP}^+$$

THEOREM M.4 (RSSP IMPLIES RSSC). (Proof 2) With the relation \approx :

$$\forall \llbracket \cdot \rrbracket \text{ if } \vdash \llbracket \cdot \rrbracket : \text{RSSP}^+ \text{ then } \vdash \llbracket \cdot \rrbracket : \text{RSSC}^+$$

THEOREM M.5 (RSSC AND RSSP ARE EQUIVALENT). (Proof 3) With the relation \approx :

$$\forall \llbracket \cdot \rrbracket \vdash \llbracket \cdot \rrbracket : \text{RSSC}^+ \iff \vdash \llbracket \cdot \rrbracket : \text{RSSP}^+$$

M.2 Strong Criteria for Insecure Compilers

Definition M.6 (RSNIP).

$$\vdash \llbracket \cdot \rrbracket : \text{RSNIP} \stackrel{\text{def}}{=} \forall P. \text{ if } \vdash P : \text{RSNI}(\mathbf{L}) \text{ then } \vdash \llbracket P \rrbracket : \text{RSNI}(\mathbf{T})$$

COROLLARY M.7 ($\nexists \llbracket \cdot \rrbracket : \text{RSNIP}$).

$$\nexists \llbracket \cdot \rrbracket : \text{RSNIP} \stackrel{\text{def}}{=} \exists P. \vdash P : \text{RSNI}(\mathbf{L}) \text{ and } \nexists \llbracket P \rrbracket : \text{RSNI}(\mathbf{T})$$

M.3 Weak Criteria for Secure Compilers

Definition M.8 (Robust Speculative Safety-Preserving Compiler (RSSP⁻)).

$$\vdash \llbracket \cdot \rrbracket : \text{RSSP}^- \stackrel{\text{def}}{=} \forall P. \text{ if } \vdash P : \text{RSS}(\text{L}^-) \text{ then } \vdash \llbracket P \rrbracket : \text{RSS}(\text{T}^-)$$

Definition M.9 (Robust Speculative Safety Compilation (RSSC⁻)).

$$\begin{aligned} \vdash \llbracket \cdot \rrbracket : \text{RSSC}^- &\stackrel{\text{def}}{=} \forall P, A, \overline{\lambda^\sigma}, \exists A, \overline{\lambda^\sigma}. \\ &\text{ if } A \llbracket \llbracket P \rrbracket \rrbracket \rightsquigarrow \overline{\lambda^\sigma} \text{ then } A \llbracket P \rrbracket \rightsquigarrow \overline{\lambda^\sigma} \text{ and } \overline{\lambda^\sigma} \approx \overline{\lambda^\sigma} \end{aligned}$$

THEOREM M.10 (RSSC⁻ IMPLIES RSSP⁻). (*Proof 4*) *With the relation \approx :*

$$\forall \llbracket \cdot \rrbracket \text{ if } \vdash \llbracket \cdot \rrbracket : \text{RSSC}^- \text{ then } \vdash \llbracket \cdot \rrbracket : \text{RSSP}^-$$

THEOREM M.11 (RSSP⁻ IMPLIES RSSC⁻). (*Proof 5*) *With the relation \approx :*

$$\forall \llbracket \cdot \rrbracket \text{ if } \vdash \llbracket \cdot \rrbracket : \text{RSSP}^- \text{ then } \vdash \llbracket \cdot \rrbracket : \text{RSSC}^-$$

THEOREM M.12 (RSSC⁻ AND RSSP⁻ ARE EQUIVALENT). (*Proof 6*) *With the relation \approx :*

$$\forall \llbracket \cdot \rrbracket \vdash \llbracket \cdot \rrbracket : \text{RSSC}^- \iff \vdash \llbracket \cdot \rrbracket : \text{RSSP}^-$$

M.4 Weak Criteria for Insecure Compilers

Definition M.13 (RSNIP⁻).

$$\vdash \llbracket \cdot \rrbracket : \text{RSNIP}^- \stackrel{\text{def}}{=} \forall P. \text{ if } \vdash P : \text{RSNI}(\text{L}^-) \text{ then } \vdash \llbracket P \rrbracket : \text{RSNI}(\text{T}^-)$$

COROLLARY M.14 ($\not\vdash \llbracket \cdot \rrbracket : \text{RSNIP}^-$).

$$\not\vdash \llbracket \cdot \rrbracket : \text{RSNIP}^- \stackrel{\text{def}}{=} \exists P. \vdash P : \text{RSNI}(\text{L}^-) \text{ and } \not\vdash \llbracket P \rrbracket : \text{RSNI}(\text{T}^-)$$

N COMPILER INSECURITY RESULTS

N.1 Unsafe SLH

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

THEOREM N.1 (THIS SLH COMPILER IS NOT *RSNIP*⁺).

$$\not\vdash \llbracket \cdot \rrbracket^s : \text{RSNIP}^+$$

PROOF. The attacker is the same:

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

Below are the two different target traces for the code of Appendix I.8.

$$\begin{aligned} t' &= \text{call get } 8?^S \cdot \text{read}(1)^S \cdot \text{read}(-(n_a - 1 + 8 + 1))^S \cdot \\ &\quad \text{read}(-1)^S \cdot \text{if}(1)^S \cdot \text{read}(-1)^S \cdot \text{write}(-1)^S \cdot \\ &\quad \text{read}(n_b + v_a)^U \cdot \text{rlb}^S \cdot \text{read}(-1)^S \cdot \text{write}(-1)^S \cdot \text{ret!}^S \\ t' &= \text{call get } 8?^S \cdot \text{read}(1)^S \cdot \text{read}(-(n_a - 1 + 8 + 1))^S \cdot \\ &\quad \text{read}(-1)^S \cdot \text{if}(1)^S \cdot \text{read}(-1)^S \cdot \text{write}(-1)^S \cdot \\ &\quad \text{read}(n_b + v'_a)^U \cdot \text{rlb}^S \cdot \text{read}(-1)^S \cdot \text{write}(-1)^S \cdot \text{ret!}^S \\ t \upharpoonright_{nse} = t' \upharpoonright_{nse} &= \text{call get } 8?^S \cdot \text{read}(1)^S \cdot \text{read}(-(n_a - 1 + 8 + 1))^S \cdot \\ &\quad \text{if}(1)^S \cdot \text{read}(-1)^S \cdot \text{write}(-1)^S \cdot \text{ret!}^S \end{aligned}$$

□

N.2 Unsafe Inter-procedural SLH

This SLH compiler does not pass the *pr* state across procedures and stores it in a local variable.

$$\begin{aligned} \llbracket f(x) \mapsto s; \text{return}; \rrbracket_n^s &= f(x) \mapsto \text{let } x_{pr} = \text{false} \text{ in } \llbracket s \rrbracket_n^s; \text{return}; \\ \llbracket \text{ifz } e \text{ then } s \text{ else } s' \rrbracket_n^s &= \text{let } x_g = \llbracket e \rrbracket^s \text{ in} \\ &\quad \text{ifz } x_g \text{ then let } x_{pr} = x_{pr} \vee \neg x_g \text{ in } \llbracket s \rrbracket_n^s \\ &\quad \text{else let } x_{pr} = x_{pr} \vee x_g \text{ in } \llbracket s' \rrbracket_n^s \\ \llbracket \text{let } x = \text{rd}_{pr} \ e \text{ in } s \rrbracket_n^s &= \text{let } x = \text{rd}_{pr} \ e \text{ in let } x = 0 \text{ (if } x_{pr}) \text{ in } \llbracket s \rrbracket_n^s \end{aligned}$$

In order to prove *RSSC* for this compiler, we need a strong relation between states that instead of asserting that $H(-1)$ keeps a bool of the speculation, each state has the first binding for a variable which captures speculation.

When proving Lemma Q.11 (Speculation Lasts at Most Omega), in the case of a call from a context to compiled component, we are not able to instate this invariant. So, there, we need to add **lfence**, so that we stop speculation altogether when jumping into compiled code. This is noted in Proof 30.

Crucially, this compiler is not *RSNIP*.

THEOREM N.2 ($\llbracket \cdot \rrbracket_n^s$ IS NOT *RSNIP*).

$$\not\vdash \llbracket \cdot \rrbracket_n^s : \text{RSNIP}$$

PROOF.

$$\begin{aligned} \text{get}(y) &\mapsto \text{let size} = \text{rd } 1 \text{ in let } x = \text{rd}_{pr} \ n_a + y \text{ in ifz } y < \text{size} \\ &\quad \text{then call get2 } x \quad \text{else skip} \\ \text{get2}(x) &\mapsto \text{let temp} = \text{rd } n_b + x \text{ in skip} \\ \text{get}(y) &\mapsto \\ &\quad \text{let pr} = 1 \text{ in let size} = \text{rd } 1 \text{ in let } x = \text{rd}_{pr} \ n_a + y \text{ in} \\ &\quad \text{let } x_g = y < \text{size} \text{ in ifz } x_g \end{aligned}$$

```

then let pr=pr  $\vee$   $\neg$ xg in call get2 x
else let pr=pr  $\vee$  xg in skip
get2(x)  $\mapsto$  let pr=1 in let temp = rd n + b + x in skip

```

$$t' = \text{call get } 8?^S \cdot \text{read}(1)^S \cdot \text{read}(-(n_a + 42))^S \cdot$$

$$\text{if}(1)^S \cdot \text{read}(n_b + v_a)^U \cdot \text{r1b}^S \cdot \text{ret}!^S$$

$$t' = \text{call get } 8?^S \cdot \text{read}(1)^S \cdot \text{read}(-(n_a + 42))^S \cdot$$

$$\text{if}(1)^S \cdot \text{read}(n_b + v_a')^U \cdot \text{r1b}^S \cdot \text{ret}!^S$$

$$t \upharpoonright_{nse} = t' \upharpoonright_{nse} = \text{call get } 8?^S \cdot \text{read}(1)^S \cdot \text{read}(-(n_a + 42))^S \cdot$$

$$\text{if}(1)^S \cdot \text{ret}!^S$$

□

N.3 MSVC Details

Unlike ICC, MSVC tries to reduce the number of **lfences** by selectively determining which branches to patch. Example N.3 below illustrates how MSVC works on the standard Spectre v1 snippet while Example N.4 (and as pointed out in [27, 36]) illustrates that MSVC sometimes omits necessary **lfences**.

Example N.3 (MSVC in action). Listing 10 presents the (simplified) assembly produced by MSVC on the code of Listing 6.

```

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

```

Listing 10: Listing 6 compiled with MSVC with /Qspectre flag enabled.

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

Example N.4 (MSVC is not RSNI). Consider now the code in Listing 8, which is adapted from from [36, Example 10]. In contrast to Listing 6, this example speculatively leaks whether $A[y]$ is 0 through the branch statement in line 3.

When compiling this code with the `lfence`-countermeasure enabled, MSVC produces the snippet shown in Listing 11.

```

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

```

Listing 11: Listing 8 compiled with MVCC with /Qspectre flag enabled.

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

In our framework, the source program from Listing 8 trivially satisfies RSNI, because the source language L does not allow speculative execution. Its compilation in Listing 11, however, violates RSNI. To show this, consider two low-equivalent initial states Ω^0, Ω^1 where y is out-of-bound, $A[y]$ is 0 in Ω^0 and 1 in Ω^1 , and the value of y is 42 in both. The corresponding traces are:

$$t_{\Omega^0} = \text{call get } 42?^S \cdot \text{if}(0)^S \cdot \text{read}(n_A + 42)^S \cdot$$

$$\text{if}(0)^U \cdot \text{r1b}^S \cdot \text{read}(n_B + 0)^S \cdot \text{r1b}^S$$

$$t_{\Omega^1} = \text{call get } 42?^S \cdot \text{if}(0)^S \cdot \text{read}(n_A + 42)^S \cdot$$

$$\text{if}(1)^U \cdot \text{read}(n_B + 0)^S \cdot \text{r1b}^S \cdot \text{r1b}^S$$

These two traces have the same non-speculative projection

`call get 42?`^S · `if(0)`^S but they differ in the observation associated with the branch instruction from line 5 (which is `if(0)`^U in t_{Q^0} and `if(1)`^U in t_{Q^1}). Therefore, they are a counterexample to RSNI. As a result, MVCC violates *RSNIP* since it does not preserve RSNI. \square

N.4 SLH Details

Example N.5 (SLH in action). Consider again the Spectre v1 snippet from Listing 6. Clang with SLH enabled compiles the program into the (simplified) assembly in Listing 12.

```

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

```

Listing 12: Compiled version of Listing 6 produced by Clang with `-x86-speculative-load-hardening` flag enabled.

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

Example N.6 (SLH is not RSNI). Consider the variant of Spectre v1 illustrated in Listing 9. The main difference with the standard Spectre v1 example (Listing 6) is that the first memory access is performed non-speculatively (line 2). Its value, however, is still leaked through the speculatively-executed memory access in line 4. Clang with SLH compiles this code into the snippet of Listing 13.

```

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

```

Listing 13: Compiled version of Listing 9 produced by Clang with `-x86-speculative-load-hardening` flag enabled.

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

Example N.7 (Non-interprocedural SLH is not RSNI). The program of Listing 14 splits the memory accesses of `A` and `B` of the classical snippet across functions `get` and `get_2`.

```

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

```

Listing 14: Inter-procedural variant of the Spectre v1 snippet [42].

Intuitively, once compiled, `get` starts the speculative execution (line 3), then the compiled code corresponding to `get_2` is executed speculatively. However, the predicate bit of `get_2` is set to `0` upon calling the function and therefore the memory access corresponding to `B[x]` is not masked and it leaks the value of `x` (which is equivalent to `A[y]`). □

O THE LFENCE COMPILER $\llbracket \cdot \rrbracket^f$

The lfence compiler (as implemented in Intel ICC).

The main feature is that the ‘then’ and ‘else’ branches of the conditionals start with an **lfence**, so no speculation is possible in the branches. We do not add a speculation barrier at function boundaries for the same reason why we do not let the context speculate (see Appendix E.2.2). Since the context speculates and since the only source of speculation is branching, we do not need to add **lfence** at function boundaries. We would need to do so were we to model speculation on return addresses too.

$$\llbracket \mathbf{H}; \bar{\mathbf{F}}; \bar{\mathbf{i}} \rrbracket^f = \llbracket \mathbf{H} \rrbracket^f; \llbracket \bar{\mathbf{F}} \rrbracket^f; \llbracket \bar{\mathbf{i}} \rrbracket^f$$

$$\llbracket \emptyset \rrbracket^f = \emptyset$$

$$\llbracket \bar{\mathbf{i}} \cdot \mathbf{f} \rrbracket^f = \llbracket \bar{\mathbf{i}} \rrbracket^f \cdot \mathbf{f}$$

$$\llbracket \mathbf{H}; -\mathbf{n} \mapsto \mathbf{v} : \mathbf{U} \rrbracket^f = \llbracket \mathbf{H} \rrbracket^f; -\llbracket \mathbf{n} \rrbracket^f \mapsto \llbracket \mathbf{v} \rrbracket^f : \mathbf{U}$$

$$\llbracket \mathbf{f}(\mathbf{x}) \mapsto \mathbf{s}; \mathbf{return}; \rrbracket^f = \mathbf{f}(\mathbf{x}) \mapsto \llbracket \mathbf{s} \rrbracket^f; \mathbf{return};$$

$$\llbracket \mathbf{s}; \mathbf{s}' \rrbracket^f = \llbracket \mathbf{s} \rrbracket^f; \llbracket \mathbf{s}' \rrbracket^f$$

$$\llbracket \mathbf{skip} \rrbracket^f = \mathbf{skip}$$

$$\llbracket \mathbf{let} \mathbf{x} = \mathbf{e} \mathbf{in} \mathbf{s} \rrbracket^f = \mathbf{let} \mathbf{x} = \llbracket \mathbf{e} \rrbracket^f \mathbf{in} \llbracket \mathbf{s} \rrbracket^f$$

$$\llbracket \mathbf{ifz} \mathbf{e} \mathbf{then} \mathbf{s} \mathbf{else} \mathbf{s}' \rrbracket^f = \mathbf{ifz} \llbracket \mathbf{e} \rrbracket^f \mathbf{then} \{\mathbf{lfence}; \llbracket \mathbf{s} \rrbracket^f\} \mathbf{else} \{\mathbf{lfence}; \llbracket \mathbf{s}' \rrbracket^f\}$$

$$\llbracket \mathbf{call} \mathbf{f} \mathbf{e} \rrbracket^f = \mathbf{call} \mathbf{f} \llbracket \mathbf{e} \rrbracket^f$$

$$\llbracket \mathbf{e} := \mathbf{e}' \rrbracket^f = \llbracket \mathbf{e} \rrbracket^f := \llbracket \mathbf{e}' \rrbracket^f$$

$$\llbracket \mathbf{let} \mathbf{x} = \mathbf{rd} \mathbf{e} \mathbf{in} \mathbf{s} \rrbracket^f = \mathbf{let} \mathbf{x} = \mathbf{rd} \llbracket \mathbf{e} \rrbracket^f \mathbf{in} \llbracket \mathbf{s} \rrbracket^f$$

$$\llbracket \mathbf{e} :=_{\mathbf{pr}} \mathbf{e}' \rrbracket^f = \llbracket \mathbf{e} \rrbracket^f :=_{\mathbf{pr}} \llbracket \mathbf{e}' \rrbracket^f$$

$$\llbracket \mathbf{let} \mathbf{x} = \mathbf{rd}_{\mathbf{pr}} \mathbf{e} \mathbf{in} \mathbf{s} \rrbracket^f = \mathbf{let} \mathbf{x} = \mathbf{rd}_{\mathbf{pr}} \llbracket \mathbf{e} \rrbracket^f \mathbf{in} \llbracket \mathbf{s} \rrbracket^f$$

$$\llbracket \mathbf{n} \rrbracket^f = \mathbf{n}$$

$$\llbracket \mathbf{e} \oplus \mathbf{e}' \rrbracket^f = \llbracket \mathbf{e} \rrbracket^f \oplus \llbracket \mathbf{e}' \rrbracket^f$$

$$\llbracket \mathbf{e} \otimes \mathbf{e}' \rrbracket^f = \llbracket \mathbf{e} \rrbracket^f \otimes \llbracket \mathbf{e}' \rrbracket^f$$

THEOREM O.1 (THE LFENCE COMPILER IS $RSSC^+$). (Proof 7)

$$\vdash \llbracket \cdot \rrbracket^f : RSSC^+$$

THEOREM O.2 (ALL LFENCE-COMPILED PROGRAMS ARE $RSS(L)$). (Proof 8)

$$\forall P. \vdash \llbracket P \rrbracket^f : RSS(L)$$

O.1 Backtranslation

We need a backtranslation for the proof. In this case, given that the languages are so close, we build both a context-based backtranslation (Appendix O.1.1) and a trace-based backtranslation (analogous to the one of the SLH compiler).

O.1.1 Context-based Backtranslation.

$$\langle\langle \mathbf{H}; \bar{\mathbf{F}} \rangle\rangle_c^f = \langle\langle \mathbf{H} \rangle\rangle_c^f; \langle\langle \bar{\mathbf{F}} \rangle\rangle_c^f$$

$$\langle\langle \emptyset \rangle\rangle_c^f = \emptyset$$

$$\langle\langle \mathbf{H}; \mathbf{n} \mapsto \mathbf{v} : \sigma \rangle\rangle_c^f = \langle\langle \mathbf{H} \rangle\rangle_c^f; \langle\langle \mathbf{n} \rangle\rangle_c^f \mapsto \langle\langle \mathbf{v} \rangle\rangle_c^f : \langle\langle \sigma \rangle\rangle_c^f$$

$$\begin{aligned}
\llbracket f(x) \mapsto s; \text{return}; \rrbracket_c^f &= f(x) \mapsto \llbracket s \rrbracket_c^f; \text{return}; \\
\llbracket \sigma \rrbracket_c^f &= \sigma \\
\llbracket n \rrbracket_c^f &= n \\
\llbracket e \oplus e' \rrbracket_c^f &= \llbracket e \rrbracket_c^f \oplus \llbracket e' \rrbracket_c^f \\
\llbracket e \otimes e' \rrbracket_c^f &= \llbracket e \rrbracket_c^f \otimes \llbracket e' \rrbracket_c^f \\
\llbracket \text{skip} \rrbracket_c^f &= \text{skip} \\
\llbracket s; s' \rrbracket_c^f &= \llbracket s \rrbracket_c^f; \llbracket s' \rrbracket_c^f \\
\llbracket \text{let } x = e \text{ in } s \rrbracket_c^f &= \text{let } x = \llbracket e \rrbracket_c^f \text{ in } \llbracket s \rrbracket_c^f \\
\llbracket \text{ifz } e \text{ then } s \text{ else } s' \rrbracket_c^f &= \text{ifz } \llbracket e \rrbracket_c^f \text{ then } \llbracket s \rrbracket_c^f \text{ else } \llbracket s' \rrbracket_c^f \\
\llbracket \text{call } f \ e \rrbracket_c^f &= \text{call } f \ \llbracket e \rrbracket_c^f \\
\llbracket e := e' \rrbracket_c^f &= \llbracket e \rrbracket_c^f := \llbracket e' \rrbracket_c^f \\
\llbracket \text{let } x = \text{rd } e \text{ in } s \rrbracket_c^f &= \text{let } x = \text{rd } \llbracket e \rrbracket_c^f \text{ in } \llbracket s \rrbracket_c^f \\
\llbracket \text{lfence} \rrbracket_c^f &= \text{skip} \\
\llbracket \text{let } x = e \text{ (if } e') \text{ in } s \rrbracket_c^f &= \text{ifz } \llbracket e' \rrbracket_c^f \text{ then let } x = \llbracket e \rrbracket_c^f \text{ in skip else skip}; \llbracket s \rrbracket_c^f
\end{aligned}$$

Note that we define the backtranslation of heaps because attackers define them. We do not define compilation of heaps because components do not define them, though adding them would be simple.

We can use this backtranslation to prove *RSSC*.

O.1.2 Properties of the Context-based Backtranslation. We want the backtranslation to be correct, so given a compiled program and a context, the backtranslation generates a source context that with the program generates a trace that is related to the target one.

THEOREM O.3 (CORRECTNESS OF THE BACKTRANSLATION FOR LFENCE). (*Proof 9*)

$$\begin{aligned}
&\text{if } A \left[\llbracket P \rrbracket^f \right] \rightsquigarrow \bar{\lambda} \bar{\sigma} \\
&\text{then } \llbracket A \rrbracket_c^f [P] \rightsquigarrow \bar{\lambda} \bar{\sigma} \\
&\text{and } \bar{\lambda} \bar{\sigma} \approx \bar{\lambda} \bar{\sigma}
\end{aligned}$$

THEOREM O.4 (GENERALISED BACKWARD SIMULATION FOR LFENCE). (*Proof 10*)

$$\begin{aligned}
&\text{if } f \in \bar{f}' \text{ then } s = \llbracket s \rrbracket_c^f \text{ else } s = \llbracket s \rrbracket_c^f \\
&\text{and } f' \in \bar{f}'' \text{ then } s' = \llbracket s' \rrbracket_c^f \text{ else } s' = \llbracket s' \rrbracket_c^f \\
&\text{and } \Sigma = \mathbf{w}(C, H, \bar{B} \triangleright (s; s'')_{\bar{f}, f}, \perp, S) \\
&\text{and } \Sigma' = \mathbf{w}(C, H', \bar{B}' \triangleright (s'; s'')_{\bar{f}', f'}, \perp, S) \\
&\text{and } (n, \Sigma) \xrightarrow{\bar{\lambda} \bar{\sigma}} (n', \Sigma') \\
&\text{and } \cdot \stackrel{s}{\approx}_{\bar{f}'} \Sigma \\
&\text{then } \cdot = C, H, \bar{B} \triangleright (s; s'')_{\bar{f}, f} \xrightarrow{\bar{\lambda} \bar{\sigma}} C, H', \bar{B}' \triangleright (s'; s'')_{\bar{f}', f'} = \cdot' \\
&\text{and } \bar{\lambda} \bar{\sigma} \approx \bar{\lambda} \bar{\sigma} \\
&\text{and } \cdot' \stackrel{s}{\approx}_{\bar{f}'} \Sigma'
\end{aligned}$$

O.1.3 Simulation and Relation for Compiled Code. We need the usual backward simulation result which we derive from forward simulation plus determinism of the semantics.

Values are only nats, so values are related if they are the same nat. Heaps are related if they map related nats (the same address) to related values.

$$\begin{array}{c}
 \boxed{\text{Heap relation } \overset{H}{\approx} \text{ Value relation } \overset{V}{\approx}} \\
 \hline
 \begin{array}{ccc}
 \text{(Heap - base)} & \text{(Heap - ind)} & \text{(Heap - start)} \\
 \frac{\emptyset \overset{H}{\approx} \emptyset}{\emptyset \overset{H}{\approx} \emptyset} & \frac{\begin{array}{c} H \overset{H}{\approx} H \\ v \overset{V}{\approx} v \quad \sigma \equiv \sigma \end{array}}{H; z \mapsto v : \sigma \overset{H}{\approx} H; z \mapsto v : \sigma} & \frac{\begin{array}{c} H \overset{H}{\approx} H \quad H' \overset{H}{\approx} H' \\ v \overset{V}{\approx} v \quad \sigma \equiv \sigma \end{array}}{H; 0 \mapsto v : \sigma; H' \overset{H}{\approx} H; 0 \mapsto v : \sigma; H'} \\
 \end{array}
 \end{array}
 \quad
 \begin{array}{c}
 \text{(Value - num)} \\
 \frac{z \equiv z \quad z \in \mathbb{Z}}{z \overset{V}{\approx} z}
 \end{array}
 \end{array}$$

Bindings are related if they map same-named variables to related values. Components are related according to a list of function names \bar{f} which identify compiled code. All functions in the list have a compiled counterpart in the target component while all functions not in the list have a backtranslated counterpart in the source component. States are related if the target is not speculating and the Ω sub-component of the target state is related to the source state. This relation is the key one for this compiler, the SLH compiler will need a different one.

$$\begin{array}{c}
 \boxed{\text{Binding relation } \overset{B}{\approx} \text{ Component relation } \overset{C}{\approx} \text{ State relation } \overset{S}{\approx}} \\
 \hline
 \begin{array}{ccc}
 \text{(Binding - base)} & \text{(Binding - ind)} & \text{(Bindings)} \\
 \frac{\emptyset \overset{B}{\approx} \emptyset}{\emptyset \overset{B}{\approx} \emptyset} & \frac{\begin{array}{c} B \overset{B}{\approx} B \quad v \overset{V}{\approx} v \quad \sigma \equiv \sigma \\ B \cdot x \mapsto v : \sigma \overset{B}{\approx} B \cdot x \mapsto v : \sigma \end{array}}{\begin{array}{c} B \overset{B}{\approx} B \quad v \overset{V}{\approx} v \quad \sigma \equiv \sigma \\ B \cdot x \mapsto v : \sigma \overset{B}{\approx} B \cdot x \mapsto v : \sigma \end{array}} & \frac{\begin{array}{c} \bar{B} \overset{B}{\approx} \bar{B} \quad B \overset{B}{\approx} B \\ \bar{B} \cdot B \overset{B}{\approx} \bar{B} \cdot B \end{array}}{\bar{B} \cdot B \overset{B}{\approx} \bar{B} \cdot B} \\
 \end{array}
 \end{array}
 \quad
 \begin{array}{c}
 \text{(Components)} \\
 \frac{\begin{array}{l} \forall f \in \bar{f}. \text{ if } f(x) \mapsto s \in \bar{F} \text{ then } f(x) \mapsto \llbracket s \rrbracket^f \in \bar{F} \\ \forall f(x) \mapsto s \in \bar{F} \text{ if } f \notin \bar{f} \text{ then } f(x) \mapsto \langle\langle s \rangle\rangle_c^f \in \bar{F} \\ \bar{I} \equiv \bar{I} \end{array}}{\bar{F}; \bar{I} \overset{C}{\approx} \bar{F}; \bar{I}}
 \end{array}$$

$$\begin{array}{c}
 \text{(States)} \\
 \frac{\begin{array}{c} \bar{B} \overset{B}{\approx} \bar{B} \quad H \overset{H}{\approx} H \quad \bar{f} \equiv \bar{f} \quad C \overset{C}{\approx} C \\ C, H, \bar{B} \triangleright (s)_{\bar{f}} \overset{S}{\approx} w, (C, H, \bar{B} \triangleright (s)_{\bar{f}}, \perp, S) \end{array}}{C, H, \bar{B} \triangleright (s)_{\bar{f}} \overset{S}{\approx} w, (C, H, \bar{B} \triangleright (s)_{\bar{f}}, \perp, S)}
 \end{array}$$

The state relation is very powerful because the target cannot be in a speculation state. We can break this invariant temporarily (at the beginning of a compiled if) but we need to reinstate it (by executing the lfence).

Starting with a related stack frame, if a source expression with a substitution star-reduces to a value, then the compiled expression with the compiled substitution star-reduces to the compiled value.

LEMMA O.5 (FORWARD SIMULATION FOR EXPRESSIONS IN LFENCE). (*Proof 12*)

$$\begin{array}{l}
 \text{if } B \triangleright e \downarrow v : \sigma \\
 \text{and } B \overset{B}{\approx} B \\
 \text{and } \sigma \equiv \sigma \\
 \text{then } B \triangleright \llbracket e \rrbracket^f \downarrow \llbracket v \rrbracket^f : \sigma
 \end{array}$$

Starting with related components, heaps and stack frames, if a source statement takes a step emitting a label, then the compiled statement can take several steps and emit a trace that is related to the label. Effectively, the target also only emits a single label, but we need to account for multiple steps (for the compilation of the if). We keep track of arbitrary source and target continuations s'' and s''' that are not touched by the reductions in order to use this result in a general setting.

LEMMA O.6 (FORWARD SIMULATION FOR COMPILED STATEMENTS IN LFENCE). (*Proof 13*)

$$\begin{array}{l}
 \text{if } \cdot = C, H, \bar{B} \triangleright (s; s'')_{\bar{f}} \xrightarrow{\lambda^\sigma} C, H', \bar{B}' \triangleright (s'; s''')_{\bar{f}} = \cdot' \\
 \text{and } \cdot \overset{S}{\approx} \bar{f} \Sigma \\
 \text{and } \Sigma = w, (C, H, \bar{B} \triangleright (\llbracket s \rrbracket^f; s'')_{\bar{f}}, \perp, S) \\
 \text{and } \Sigma' = w(C, H', \bar{B}' \triangleright (\llbracket s' \rrbracket^f; s''')_{\bar{f}}, \perp, S) \\
 \text{then } (n, \Sigma) \xrightarrow{\bar{\lambda}^\sigma} (n', \Sigma') \\
 \text{and } \lambda^\sigma \approx \bar{\lambda}^\sigma \quad (\text{using the trace relation!}) \\
 \text{and } \cdot' \overset{S}{\approx} \bar{f} \Sigma'
 \end{array}$$

In the general theorem we need backward simulation, which we derive from forward simulation in a standard way. Note that by ending up in a state with a compiled statement, we rule out cross-boundary calls and returns. These cases pop up in the proofs using this theorem and we deal with them there.

THEOREM O.7 (BACKWARD SIMULATION FOR COMPILED STEPS IN LFENCE). *(Proof 11)*

$$\begin{aligned}
& \text{if } \Sigma = \mathbf{w}, (\mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \triangleright (\llbracket \mathbf{s} \rrbracket^f; \mathbf{s}'')_{\overline{\mathbf{F}}}, \perp, \mathbf{S}) \\
& \text{and } \Sigma' = \mathbf{w}(\mathbf{C}, \mathbf{H}', \overline{\mathbf{B}'} \triangleright (\llbracket \mathbf{s}' \rrbracket^f; \mathbf{s}'')_{\overline{\mathbf{F}'}, \perp, \mathbf{S}}) \\
& \text{and } (\mathbf{n}, \Sigma) \xrightarrow{\overline{\lambda}^\sigma} (\mathbf{n}', \Sigma') \\
& \text{and } \cdot \overset{\mathcal{S}}{\approx}_{\overline{\mathbf{F}'}} \Sigma \\
& \text{then } \cdot = \mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \triangleright (\mathbf{s}; \mathbf{s}'')_{\overline{\mathbf{F}}} \xrightarrow{\lambda^\sigma} \mathbf{C}, \mathbf{H}', \overline{\mathbf{B}'} \triangleright (\mathbf{s}'; \mathbf{s}'')_{\overline{\mathbf{F}'}} = \cdot' \\
& \text{and } \lambda^\sigma \approx \overline{\lambda}^\sigma \quad (\text{using the trace relation!}) \\
& \text{and } \cdot' \overset{\mathcal{S}}{\approx}_{\overline{\mathbf{F}'}} \Sigma'
\end{aligned}$$

O.1.4 *Simulation and Relation for Backtranslated Code.* We need backward simulation for backtranslated code. Since we are doing a context-based backtranslation, and since backtranslated values are related to source values using $\overset{\mathcal{V}}{\approx}$ as for compiled values, we do not need extra relations.

As before, we need two lemmas on the backward simulation for backtranslated expressions and on the backward simulation for statements.

LEMMA O.8 (BACKWARD SIMULATION FOR BACKTRANSLATED EXPRESSIONS IN LFENCE). *(Proof 14)*

$$\begin{aligned}
& \text{if } \mathbf{B} \triangleright \mathbf{e} \downarrow \mathbf{v} : \sigma \\
& \text{and } \mathbf{B} \overset{\mathcal{B}}{\approx} \mathbf{B} \\
& \text{and } \sigma \equiv \sigma \\
& \text{then } \mathbf{B} \triangleright \langle\langle \mathbf{e} \rangle\rangle_c^f \downarrow \langle\langle \mathbf{v} \rangle\rangle_c^f : \sigma
\end{aligned}$$

LEMMA O.9 (BACKWARD SIMULATION FOR BACKTRANSLATED STATEMENTS). *(Proof 15)*

$$\begin{aligned}
& \text{if } \Sigma = \mathbf{w}(\mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \triangleright (\mathbf{s}; \mathbf{s}'')_{\overline{\mathbf{F}}}, \perp, \mathbf{S}) \\
& \text{and } \Sigma' = \mathbf{w}(\mathbf{C}, \mathbf{H}', \overline{\mathbf{B}'} \triangleright (\mathbf{s}'; \mathbf{s}'')_{\overline{\mathbf{F}'}, \perp, \mathbf{S}}) \\
& \text{and } \Sigma \overset{\lambda^\sigma}{\rightsquigarrow} \Sigma' \\
& \text{and } \cdot \overset{\mathcal{S}}{\approx}_{\overline{\mathbf{F}'}} \Sigma \\
& \text{then } \cdot = \mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \triangleright (\langle\langle \mathbf{s} \rangle\rangle_c^f; \mathbf{s}'')_{\overline{\mathbf{F}}} \xrightarrow{\lambda^\sigma} \mathbf{C}, \mathbf{H}', \overline{\mathbf{B}'} \triangleright (\langle\langle \mathbf{s}' \rangle\rangle_c^f; \mathbf{s}'')_{\overline{\mathbf{F}'}} = \cdot' \\
& \text{and } \lambda^\sigma \overset{\mathcal{A}}{\approx} \overline{\lambda}^\sigma \quad (\text{using the action relation!}) \\
& \text{and } \cdot' \overset{\mathcal{S}}{\approx}_{\overline{\mathbf{F}'}} \Sigma'
\end{aligned}$$

Note that by ending up in a state with a backtranslated statement, we rule out cross-boundary calls and returns, they are dealt with in Theorem O.4.

The final result we will need for the general theorem is that initial states made of a compiled program and a backtranslated context are related. This relies on heap and value cross-language relation holding for compiled and backtranslated heaps and values.

LEMMA O.10 (INITIAL STATES ARE RELATED). *(Proof 16)*

$$\begin{aligned}
& \forall \mathbf{P}, \forall \overline{\mathbf{f}} = \text{dom}(\mathbf{P}, \mathbf{F}), \forall \mathbf{A} \\
& \cdot_0 \left(\langle\langle \mathbf{A} \rangle\rangle_c^f [\mathbf{P}] \right) \overset{\mathcal{S}}{\approx}_{\overline{\mathbf{f}}} \Omega_0 \left(\mathbf{A} \llbracket \mathbf{P} \rrbracket^f \right)
\end{aligned}$$

LEMMA O.11 (A VALUE IS RELATED TO ITS COMPILATION FOR LFENCE). *(Proof 17)*

$$\mathbf{v} \overset{\mathcal{V}}{\approx} \llbracket \mathbf{v} \rrbracket^f$$

LEMMA O.12 (A HEAP IS RELATED TO ITS COMPILATION FOR LFENCE). *(Proof 18)*

$$\mathbf{H} \overset{\mathcal{H}}{\approx} \llbracket \mathbf{H} \rrbracket^f$$

LEMMA O.13 (A VALUE IS RELATED TO ITS BACKTRANSLATION). *(Proof 19)*

$$\langle\langle \mathbf{v} \rangle\rangle_c^f \overset{\mathcal{V}}{\approx} \mathbf{v}$$

LEMMA O.14 (A TAINT IS RELATED TO ITS BACKTRANSLATION). (*Proof 20*)

$$\langle\langle \sigma \rangle\rangle_c^f \equiv \sigma$$

LEMMA O.15 (A HEAP IS RELATED TO ITS BACKTRANSLATION). (*Proof 21*)

$$\langle\langle \mathbf{H} \rangle\rangle_c^f \stackrel{H}{\approx} \mathbf{H}$$

O.2 A Stronger Property for $\llbracket \cdot \rrbracket^f$

THEOREM O.16 (THE LFENCE COMPILER IS *RSSC* WITH MORE LEAKS). (*Proof 22*)

$$\vdash \llbracket \cdot \rrbracket^f : \text{RSSC}$$

Let's add a reduction that generates an η action:

$$\frac{\text{(E-T-speculate-eta)} \quad \Omega \xrightarrow{\epsilon} \Omega' \quad \Omega \equiv \mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \triangleright \mathbf{s}; \mathbf{s}' \quad \mathbf{s} \neq \text{lfence}}{\mathbf{w}, \overline{(\Omega, \omega, \sigma)} \cdot (\Omega, \mathbf{n} + 1, \sigma) \xrightarrow{\eta^\sigma} \mathbf{w}, \overline{(\Omega, \omega, \sigma)} \cdot (\Omega', \mathbf{n}, \sigma)}$$

Any silent step that is not lfence now generates an η action whose tag is the same as the pc's. Thus, eta actions are safe when not speculating and they are unsafe when speculating.

P THE CURRENT SLH COMPILER $\llbracket \cdot \rrbracket^s$

$$\begin{aligned}
\llbracket \text{call } f \ e \rrbracket^s &= \text{call } f \ \llbracket e \rrbracket^s \\
\llbracket e \ := \ e' \rrbracket^s &= \llbracket e \rrbracket^s \ := \ \llbracket e' \rrbracket^s \\
\llbracket \text{let } x = \text{rd } e \ \text{in } s \rrbracket^s &= \text{let } x = \text{rd} \ \llbracket e \rrbracket^s \ \text{in} \ \llbracket s \rrbracket^s \\
\llbracket e \ :=_{\text{pr}} \ e' \rrbracket^s &= \llbracket e \rrbracket^s \ + \ 1 \ :=_{\text{pr}} \ \llbracket e' \rrbracket^s \\
\llbracket \text{ifz } e \ \text{then } s \ \text{else } s' \rrbracket^s &= \text{let } x_g = \llbracket e \rrbracket^s \ \text{in} \\
&\quad \text{ifz } x_g \ \text{then let } x = \text{rd}_{\text{pr}} \ -1 \ \text{in } -1 \ :=_{\text{pr}} \ x \ \vee \ \neg x_g; \ \llbracket s \rrbracket^s \\
&\quad \text{else let } x = \text{rd}_{\text{pr}} \ -1 \ \text{in } -1 \ :=_{\text{pr}} \ x \ \vee \ x_g; \ \llbracket s' \rrbracket^s \\
\llbracket \text{let } x = \text{rd}_{\text{pr}} \ e \ \text{in } s \rrbracket^s &= \text{let } x = \text{rd}_{\text{pr}} \ \llbracket e \rrbracket^s \ + \ 1 \ \text{in let } \text{pr} = \text{rd}_{\text{pr}} \ -1 \ \text{in let } x = 0 \ (\text{if } \text{pr}) \ \text{in} \ \llbracket s \rrbracket^s
\end{aligned}$$

Omitted cases are as in Appendix Q.

THEOREM P.1 (THIS SLH COMPILER IS RSSC^-). (*Proof 41*)

$$\vdash \llbracket \cdot \rrbracket^s : \text{RSSC}^-$$

Q THE STRONG SLH COMPILER $\llbracket \cdot \rrbracket^{SS}$

$$\llbracket H; \bar{F}; \bar{I} \rrbracket^{SS} = \llbracket H \rrbracket^{SS} \cup (-1 \mapsto \text{false} : S); \llbracket \bar{F} \rrbracket^{SS}; \llbracket \bar{I} \rrbracket^{SS}$$

$$\llbracket \emptyset \rrbracket^{SS} = \emptyset$$

$$\llbracket \bar{i} \cdot f \rrbracket^{SS} = \llbracket \bar{i} \rrbracket^{SS} \cdot f$$

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

$$\llbracket f(x) \mapsto s; \text{return}; \rrbracket^{SS} = f(x) \mapsto \llbracket s \rrbracket^{SS}; \text{return};$$

$$\llbracket n \rrbracket^{SS} = n$$

$$\llbracket e \oplus e' \rrbracket^{SS} = \llbracket e \rrbracket^{SS} \oplus \llbracket e' \rrbracket^{SS}$$

$$\llbracket e \otimes e' \rrbracket^{SS} = \llbracket e \rrbracket^{SS} \otimes \llbracket e' \rrbracket^{SS}$$

$$\llbracket s; s' \rrbracket^{SS} = \llbracket s \rrbracket^{SS}; \llbracket s' \rrbracket^{SS}$$

$$\llbracket \text{skip} \rrbracket^{SS} = \text{skip}$$

$$\llbracket \text{let } x = e \text{ in } s \rrbracket^{SS} = \text{let } x = \llbracket e \rrbracket^{SS} \text{ in } \llbracket s \rrbracket^{SS}$$

$$\begin{aligned} \llbracket \text{call } f \ e \rrbracket^{SS} &= \text{let } x_f = \llbracket e \rrbracket^{SS} \text{ in} \\ &\quad \text{let } pr = rd_{pr} - 1 \text{ in} \\ &\quad \text{let } x_f = 0 \text{ (if } pr) \text{ in call } f \ x_f \end{aligned}$$

$$\begin{aligned} \llbracket e := e' \rrbracket^{SS} &= \text{let } x_f = \llbracket e \rrbracket^{SS} \text{ in} \\ &\quad \text{let } x'_f = \llbracket e' \rrbracket^{SS} \text{ in} \\ &\quad \text{let } pr = rd_{pr} - 1 \text{ in} \\ &\quad \text{let } x_f = 0 \text{ (if } pr) \text{ in} \\ &\quad \text{let } x'_f = 0 \text{ (if } pr) \text{ in} \\ &\quad x_f := x'_f \end{aligned}$$

$$\begin{aligned} \llbracket \text{let } x = rd \ e \text{ in } s \rrbracket^{SS} &= \text{let } x = rd \ \llbracket e \rrbracket^{SS} \text{ in} \\ &\quad \text{let } pr = rd_{pr} - 1 \text{ in} \\ &\quad \text{let } x = 0 \text{ (if } pr) \text{ in} \\ &\quad \llbracket s \rrbracket^{SS} \end{aligned}$$

$$\begin{aligned} \llbracket e :=_{pr} e' \rrbracket^{SS} &= \text{let } x_f = \llbracket e \rrbracket^{SS} + 1 \text{ in} \\ &\quad \text{let } x'_f = \llbracket e' \rrbracket^{SS} \text{ in} \\ &\quad \text{let } pr = rd_{pr} - 1 \text{ in} \\ &\quad \text{let } x_f = 0 \text{ (if } pr) \text{ in} \\ &\quad \text{let } x'_f = 0 \text{ (if } pr) \text{ in} \\ &\quad x_f :=_{pr} x'_f \end{aligned}$$

$$\begin{aligned} \llbracket \text{ifz } e \text{ then } s \text{ else } s' \rrbracket^{SS} &= \text{let } x_g = \llbracket e \rrbracket^{SS} \text{ in} \\ &\quad \text{let } pr = rd_{pr} - 1 \text{ in} \\ &\quad \text{let } x_g = 0 \text{ (if } pr) \text{ in} \\ &\quad \text{ifz } x_g \\ &\quad \quad \text{then let } x = rd_{pr} - 1 \text{ in } -1 :=_{pr} x \vee \neg x_g; \llbracket s \rrbracket^{SS} \\ &\quad \quad \text{else let } x = rd_{pr} - 1 \text{ in } -1 :=_{pr} x \vee x_g; \llbracket s' \rrbracket^{SS} \end{aligned}$$

$$\begin{aligned} \llbracket \text{let } x = \text{rd}_{\text{pr}} \text{ e in } s \rrbracket^{\text{SS}} &= \text{let } x = \llbracket e \rrbracket^{\text{SS}} + 1 \text{ in} \\ &\quad \text{let } \text{pr} = \text{rd}_{\text{pr}} - 1 \text{ in} \\ &\quad \text{let } x = 0 \text{ (if } \text{pr} \text{) in} \\ &\quad \text{let } x = \text{rd}_{\text{pr}} \text{ x in } \llbracket s \rrbracket^{\text{SS}} \end{aligned}$$

We compile all private reads and write to operate on an address that is greater than the expected one by 1. This ensures that location -1 is untouched by the compiled code, so it can be used to keep information, namely the **pr** state.

THEOREM Q.1 (OUR SLH COMPILER IS $RSSC^+$). (Proof 38)

$$\vdash \llbracket \cdot \rrbracket^{\text{SS}} : RSSC^+$$

Q.1 Inter-procedural SLH

This SLH compiler does not pass the **pr** state across procedures and it needs the **lfence** to still be $RSSC^+$.

$$\begin{aligned} \llbracket f(x) \mapsto s; \text{return}; \rrbracket_n^s &= f(x) \mapsto \text{lfence}; \\ &\quad \text{let } x_{\text{pr}} = \text{false in } \llbracket s \rrbracket_n^s; \text{return}; \\ \llbracket \text{ifz } e \text{ then } s \text{ else } s' \rrbracket_n^s &= \text{let } x_g = \llbracket e \rrbracket^{\text{SS}} \text{ in} \\ &\quad \text{ifz } x_g \text{ then let } x_{\text{pr}} = x_{\text{pr}} \vee \neg x_g \text{ in } \llbracket s \rrbracket_n^s \\ &\quad \text{else let } x_{\text{pr}} = x_{\text{pr}} \vee x_g \text{ in } \llbracket s' \rrbracket_n^s \\ \llbracket \text{let } x = \text{rd}_{\text{pr}} \text{ e in } s \rrbracket_n^s &= \text{let } x = \text{rd}_{\text{pr}} \text{ e in let } x = 0 \text{ (if } x_{\text{pr}} \text{) in } \llbracket s \rrbracket_n^s \end{aligned}$$

In order to prove $RSSC$ for this compiler, we need a strong relation between states that instead of asserting that $H(-1)$ keeps a bool of the speculation, each state has the first binding for a variable which captures speculation.

When proving Lemma Q.11 (Speculation Lasts at Most Omega), in the case of a call from a context to compiled component, we need to add **lfence**, so that we stop speculation altogether to instate this invariant. This is noted in Proof 30.

THEOREM Q.2 (OUR INTER-PROCEDURAL SLH COMPILER IS $RSSC^+$). (Proof 39)

$$\vdash \llbracket \cdot \rrbracket_n^s : RSSC^+$$

Q.2 Backtranslation

We can use the same context-based backtranslation of Appendix O.1.1, what changes are the cross-language relations.

Q.2.1 Relations for the SLH Compiler. The state relation extends the previous one because now we can relate a source state to a target state that is speculating. Given a target state Σ that is a stack of operational states $\bar{\Omega}$, we require that the first element of the stack is in the strong state relation ($\overset{\approx}{\approx}$) with a source state \cdot . All other states need to be related at a weaker state relation ($\overset{\approx}{\approx}$), which only enforces that all bindings map variables to **S** values. That is, a target and a source list of bindings are related if all that the target binds is **S**, though the target can bind possibly more variables and the target stack of bindings can be. While speculating, target state do not mimick source reductions anymore but we need to enforce this taint on variables in order to ensure that any target action will be **S**.

Since our target language uses possibly more variables than the source (e.g., **pr**), we need to sometimes forget them. So the binding relation is parametrised by a stack of possibly growing list of name variables \bar{D} that tell us when to not care for a binding, i.e., when we find a variable in **B** that is in the current **D**. We adapt the state relation to forward that (later).

Binding relation $\overset{B}{\approx}$

$$\begin{array}{c} \text{(Binding - base)} \\ \frac{\emptyset \overset{B}{\approx} \emptyset}{\emptyset \overset{B}{\approx} \emptyset} \end{array} \quad \begin{array}{c} \text{(Binding - ind)} \\ \frac{B \overset{B}{\approx} D \ B \quad v \overset{V}{\approx} v \quad \sigma \equiv \sigma}{B \cdot x \mapsto v : \sigma \overset{B}{\approx} D \ B \cdot x \mapsto v : \sigma} \end{array} \quad \begin{array}{c} \text{(Binding - ind)} \\ \frac{B \overset{B}{\approx} D \ B \quad x \in D}{B \overset{B}{\approx} D \ B \cdot x \mapsto v : \sigma} \end{array} \quad \begin{array}{c} \text{(Bindings)} \\ \frac{\bar{B} \overset{B}{\approx} \bar{D} \ \bar{B} \quad B \overset{B}{\approx} D \ B}{\bar{B} \cdot B \overset{B}{\approx} \bar{D} \cdot D \ \bar{B} \cdot B} \end{array}$$

We need to change the heap relation to account for the fact that private heaps are related when the addresses are -1 in the target. So we define the relation $\overset{H}{\approx}$ for private heaps (whose domain is negative integers) to account for this.

Heap relation $\overset{H}{\approx}$

$$\begin{array}{c} \text{(Heap - base)} \\ \frac{\emptyset \overset{H}{\approx} \emptyset}{\emptyset \overset{H}{\approx} \emptyset} \end{array} \quad \begin{array}{c} \text{(Heap - negative)} \\ \frac{z - 1 \overset{V}{\approx} z \quad v \overset{V}{\approx} v \quad \sigma \equiv \sigma}{H; z \mapsto v : \sigma \overset{H}{\approx} H; z \mapsto v : \sigma} \end{array} \quad \begin{array}{c} \text{(Heap - start)} \\ \frac{H \overset{H}{\approx} H \quad H' \overset{H}{\approx} H' \quad v \overset{V}{\approx} v \quad \sigma \equiv \sigma}{H; 0 \mapsto v : \sigma; H' \overset{H}{\approx} H; -1 \mapsto \text{false} : S; 0 \mapsto v : \sigma; H'} \end{array}$$

Binding relation $\overset{B}{\approx}$ State relation $\overset{S}{\approx}$

$$\begin{array}{c}
 \text{(States relation)} \qquad \qquad \qquad \text{(Base States)} \\
 \frac{\cdot \overset{S}{\approx}_{\bar{f}}^{\bar{D}} (w, (\Omega, \perp, S)) \quad \forall \Omega_s \in \bar{\Omega}, \cdot \overset{S}{\approx}_{\bar{f}}^{\bar{D}} \Omega_s}{\cdot \overset{S}{\approx}_{\bar{f}}^{\bar{D}} (w, (\Omega, \perp, S)) \cdot (\bar{\Omega}, W, U)} \quad \frac{\bar{B} \overset{B}{\approx} \bar{B} \quad H \overset{H}{\approx} H \quad \bar{f} \equiv \bar{f} \quad C \overset{C}{\approx}_{\bar{f}'} C}{C, H, \bar{B} \triangleright (s)_{\bar{f}} \overset{S}{\approx}_{\bar{f}'}^{\bar{D}} w, (C, H, \bar{B} \triangleright (s)_{\bar{f}}, \perp, S)} \\
 \text{(Single state relation - ctx)} \\
 \frac{(\text{if } f \notin \bar{f}' \text{ then } \vdash B : \overset{B}{\approx}) \quad C \overset{C}{\approx}_{\bar{f}'} C \quad \vdash H : \overset{H}{\approx}}{C, H, \bar{B} \triangleright (s)_{\bar{f}} \overset{S}{\approx}_{\bar{f}'} C, H, \bar{B} \cdot B \triangleright (s)_{\bar{f}, f}} \\
 \text{(Heap relation same)} \\
 \frac{\forall n \mapsto v : \sigma \in H \text{ if } n \geq 0 \text{ then } \sigma = S \quad H(-1) = \text{true} : S}{\vdash H : \overset{H}{\approx}} \\
 \text{(Target Bindings ok base)} \qquad \qquad \text{(Target Bindings ok sing)} \\
 \frac{}{\vdash \emptyset : \overset{B}{\approx}} \qquad \qquad \frac{}{\vdash B \cdot x \mapsto v : S : \overset{B}{\approx}}
 \end{array}$$

When speculating, any heap is related: the target may write extra things speculatively.

Q.2.2 Relation for Inter-procedural SLH.

Binding relation $\overset{B}{\approx}_{\alpha}$ State relation $\overset{S}{\approx}_{\alpha}$

$$\begin{array}{c}
 \text{(States relation)} \qquad \qquad \qquad \text{(Single state relation - ctx)} \\
 \frac{\cdot \overset{S}{\approx}_{\alpha}^{\bar{D}} (w, (\Omega, \perp, S)) \quad \forall \Omega_s \in \bar{\Omega}, \cdot \overset{S}{\approx}_{\alpha}^{\bar{D}} \Omega_s}{\cdot \overset{S}{\approx}_{\alpha}^{\bar{D}} (w, (\Omega, \perp, S)) \cdot (\bar{\Omega}, W, U)} \quad \frac{(\text{if } f \notin \bar{f}' \text{ then } \vdash B : \overset{B}{\approx}_{\alpha}) \quad C \overset{C}{\approx}_{\bar{f}'} C \quad \vdash H : \overset{H}{\approx}_{\alpha}}{C, H, \bar{B} \triangleright (s)_{\bar{f}} \overset{S}{\approx}_{\alpha}^{\bar{D}} w, (C, H, \bar{B} \triangleright (s)_{\bar{f}}, \perp, S)} \\
 \text{(Heap relation same)} \qquad \qquad \text{(Target Bindings ok base)} \qquad \qquad \text{(Target Bindings ok sing)} \\
 \frac{\forall n \mapsto v : \sigma \in H \text{ if } n \geq 0 \text{ then } \sigma = S}{\vdash H : \overset{H}{\approx}_{\alpha}} \quad \frac{}{\vdash \text{pr} \mapsto 0 : S : \overset{B}{\approx}_{\alpha}} \quad \frac{}{\vdash B \cdot x \mapsto v : S : \overset{B}{\approx}_{\alpha}}
 \end{array}$$

Speculating states are related $\overset{S}{\approx}_{\alpha}$ when, in case the executing function is not attacker (Rule Single state relation - ctx), the heap is whatever but the bindings contain the predicate bit set to true.

LEMMA Q.3 (INITIAL STATES ARE RELATED FOR SLH). (Proof 23)

$$\begin{array}{l}
 \forall P, \forall \bar{f} = \text{dom}(P.F), \forall A \\
 \cdot \left(\langle \langle A \rangle \rangle_c^f [P] \right) \overset{S}{\approx}_{\bar{f}} \Omega_0 (A \llbracket P \rrbracket^{SS})
 \end{array}$$

LEMMA Q.4 (A VALUE IS RELATED TO ITS COMPILE FOR SLH). (Proof 24)

$$v \overset{V}{\approx} \llbracket v \rrbracket^{SS}$$

LEMMA Q.5 (A HEAP IS RELATED TO ITS COMPILE FOR SLH). (Proof 25)

$$H \overset{H}{\approx} \llbracket H \rrbracket^{SS}$$

LEMMA Q.6 (FORWARD SIMULATION FOR EXPRESSIONS IN SLH). (Proof 26)

$$\begin{array}{l}
 \text{if } B \triangleright e \downarrow v : \sigma \\
 \text{and } B \overset{B}{\approx} \bar{B} \\
 \text{and } \sigma \equiv \sigma \\
 \text{then } B \triangleright \llbracket e \rrbracket^{SS} \downarrow \llbracket v \rrbracket^{SS} : \sigma
 \end{array}$$

LEMMA Q.7 (FORWARD SIMULATION FOR COMPILED STATEMENTS IN SLH). (Proof 27)

$$\begin{array}{l}
 \text{if } \cdot = C, H, \bar{B} \triangleright (s; s'')_{\bar{f}} \xrightarrow{\lambda \sigma} C, H', \bar{B}' \triangleright (s'; s'')_{\bar{f}'} = \cdot' \\
 \text{and } \cdot \overset{S}{\approx}_{\bar{f}'}^{\bar{D}} \Sigma \\
 \text{and } \Sigma = w(C, H, \bar{B} \triangleright (\llbracket s \rrbracket^{SS}; s'')_{\bar{f}}, \perp, S)
 \end{array}$$

$$\begin{aligned}
& \text{and } \Sigma' = \mathbf{w}(C, H', \overline{B'} \triangleright (\llbracket s' \rrbracket^{ss}; s'')_{\overline{f}}, \perp, S) \\
& \text{then } (n, \Sigma) \xrightarrow{\overline{\lambda}^\sigma} (n', \Sigma') \\
& \text{and } \lambda^\sigma \approx \overline{\lambda}^\sigma \quad (\text{using the trace relation!}) \\
& \text{and } \exists \overline{D'} \supseteq \overline{D}. \cdot' \stackrel{s}{\approx} \frac{\overline{D'}}{\overline{f'}} \Sigma'
\end{aligned}$$

THEOREM Q.8 (BACKWARD SIMULATION FOR COMPILED STATEMENTS IN SLH). (Proof 28)

$$\begin{aligned}
& \text{if } (n, \Sigma) \xrightarrow{\overline{\lambda}^\sigma} (n', \Sigma') \\
& \text{and } \cdot \stackrel{s}{\approx} \frac{\overline{D}}{\overline{f}} \Sigma \\
& \text{and } \Sigma = \mathbf{w}(C, H, \overline{B} \triangleright (\llbracket s \rrbracket^{ss}; s'')_{\overline{f}}, \perp, S) \\
& \text{and } \Sigma' = \mathbf{w}(C, H', \overline{B'} \triangleright (\llbracket s' \rrbracket^{ss}; s'')_{\overline{f'}}, \perp, S) \\
& \text{then } \cdot = C, H, \overline{B} \triangleright (s; s'')_{\overline{f}} \xrightarrow{\lambda^\sigma} C, H', \overline{B'} \triangleright (s'; s'')_{\overline{f'}} = \cdot' \\
& \text{and } \lambda^\sigma \approx \overline{\lambda}^\sigma \quad (\text{using the trace relation!}) \\
& \text{and } \exists \overline{D'} \supseteq \overline{D}. \cdot' \stackrel{s}{\approx} \frac{\overline{D'}}{\overline{f'}} \Sigma'
\end{aligned}$$

LEMMA Q.9 (EXPRESSION REDUCTIONS WITH SAFE BINDINGS ARE SAFE). (Proof 29)

$$\begin{aligned}
& \text{if } \vdash B : \frac{B}{S} \\
& \text{then } B \triangleright e \downarrow v : S
\end{aligned}$$

LEMMA Q.10 (ANY SPECULATION FROM RELATED STATES IS SAFE). (Proof 35)

$$\begin{aligned}
& \text{if } \cdot \stackrel{s}{\approx} \frac{\overline{D}}{\overline{f}_c} \Sigma \\
& \text{and } \Sigma = \mathbf{w}(C, H_b, \overline{B}_b \triangleright (s_b)_{\overline{f}_b}, \perp, S) \cdot \overline{(C, H', \overline{B}' \triangleright (s')_{\overline{f}'}, \omega', U)} \cdot (C, H, \overline{B} \triangleright (s)_{\overline{f}_f}, \omega, U) \\
& \text{and } \Sigma' = \mathbf{w}(C, H_b, \overline{B}_b \triangleright (s_b)_{\overline{f}_b}, \perp, S) \\
& \text{and } n' \leq n + \mathbf{w} \\
& \text{and let } \overline{\omega}' = \omega'_1, \dots, \omega'_k, \omega'_1 + \dots + \omega'_k + \omega \leq \mathbf{w} \\
& \text{then } (n, \Sigma) \xrightarrow{\overline{\lambda}^\sigma} (n', \Sigma') \\
& \text{and } \emptyset \approx \overline{\lambda}^\sigma \\
& \text{and } \cdot \stackrel{s}{\approx} \frac{\overline{D}}{\overline{f}_c} \Sigma'
\end{aligned}$$

LEMMA Q.11 (SPECULATION LASTS AT MOST OMEGA). (Proof 30)

$$\begin{aligned}
& \text{if } \cdot \stackrel{s}{\approx} \frac{\overline{D}}{\overline{f}_c} \Sigma \\
& \text{and } \Sigma = \mathbf{w}(C, H_b, \overline{B}_b \triangleright (s_b)_{\overline{f}_b}, \perp, S) \cdot \overline{(C, H', \overline{B}' \triangleright (s')_{\overline{f}'}, \omega', U)} \cdot (C, H, \overline{B} \triangleright (s)_{\overline{f}_f}, \omega, U) \\
& \text{and } \Sigma' = \mathbf{w}(C, H_b, \overline{B}_b \triangleright (s_b)_{\overline{f}_b}, \perp, S) \cdot \overline{(C, H', \overline{B}' \triangleright (s')_{\overline{f}'}, \omega', U)} \cdot (C, H'', \overline{B}'' \triangleright (s'')_{\overline{f}''}, \omega'', 0, U) \\
& \text{and } n' \leq n + \omega \\
& \text{then } (n, \Sigma) \xrightarrow{\overline{\lambda}^\sigma} (n', \Sigma') \\
& \text{and } \emptyset \approx \overline{\lambda}^\sigma \\
& \text{and } \cdot \stackrel{s}{\approx} \frac{\overline{D}}{\overline{f}_c} \Sigma'
\end{aligned}$$

LEMMA Q.12 (CONTEXT SPECULATION LASTS AT MOST OMEGA). (Proof 31)

$$\text{if } \cdot \stackrel{s}{\approx} \frac{\overline{D}}{\overline{f}_c} \Sigma$$

$$\begin{aligned}
& \text{and } \Sigma = \mathbf{w}(C, H_b, \overline{B}_b \triangleright (s_b)_{\overline{f}_b}, \perp, S) \cdot \overline{\mathbf{w}(C, H', \overline{B}' \triangleright (s')_{\overline{f}'}, \omega', U)} \cdot (C, H, \overline{B} \triangleright (s)_{\overline{f}, f}, \omega, U) \\
& \text{and } \Sigma' = \mathbf{w}(C, H_b, \overline{B}_b \triangleright (s_b)_{\overline{f}_b}, \perp, S) \cdot \overline{\mathbf{w}(C, H', \overline{B}' \triangleright (s')_{\overline{f}'}, \omega', U)} \cdot (C, H'', \overline{B}'' \triangleright (s'')_{\overline{f}'', f''}, 0, U) \\
& \text{and } \mathbf{n}' \leq \mathbf{n} + \omega \\
& \text{and } \mathbf{f}, \mathbf{f}'' \notin \overline{f}_c \\
& \text{then } (\mathbf{n}, \Sigma) \xrightarrow{\overline{\lambda}^\sigma} (\mathbf{n}', \Sigma') \\
& \text{and } \emptyset \approx \overline{\lambda}^\sigma \\
& \text{and } \cdot \underset{\approx}{\overset{s}{\overline{f}_c}} \overline{D} \Sigma'
\end{aligned}$$

LEMMA Q.13 (SINGLE CONTEXT SPECULATION IS SAFE). (*Proof 32*)

$$\begin{aligned}
& \text{if } \cdot \underset{\approx}{\overset{s}{\overline{f}_c}} \overline{D} \Sigma \\
& \text{and } \Sigma = \mathbf{w}(C, H_b, \overline{B}_b \triangleright (s_b)_{\overline{f}_b}, \perp, S) \cdot \overline{\mathbf{w}(C, H', \overline{B}' \triangleright (s')_{\overline{f}'}, \omega', U)} \cdot (C, H, \overline{B} \triangleright (s)_{\overline{f}, f}, \omega, U) \\
& \text{and } \Sigma' = \mathbf{w}(C, H_b, \overline{B}_b \triangleright (s_b)_{\overline{f}_b}, \perp, S) \cdot \overline{\mathbf{w}(C, H', \overline{B}' \triangleright (s')_{\overline{f}'}, \omega', U)} \cdot (C, H'', \overline{B}'' \triangleright (s'')_{\overline{f}'', f''}, \omega - 1, U) \\
& \text{and } \mathbf{f}, \mathbf{f}'' \notin \overline{f}_c \\
& \text{then } (\Sigma) \xrightarrow{\alpha^\sigma} (\Sigma') \\
& \text{and } \emptyset \approx \alpha^\sigma \\
& \text{and } \cdot \underset{\approx}{\overset{s}{\overline{f}_c}} \overline{D} \Sigma'
\end{aligned}$$

LEMMA Q.14 (COMPILED SPECULATION LASTS AT MOST OMEGA). (*Proof 33*)

$$\begin{aligned}
& \text{if } \cdot \underset{\approx}{\overset{s}{\overline{f}_c}} \overline{D} \Sigma \\
& \text{and } \Sigma = \mathbf{w}(C, H_b, \overline{B}_b \triangleright (s_b)_{\overline{f}_b}, \perp, S) \cdot \overline{\mathbf{w}(C, H', \overline{B}' \triangleright (s')_{\overline{f}'}, \omega', U)} \cdot (C, H, \overline{B} \triangleright (\llbracket s \rrbracket^{ss})_{\overline{f}, f}, \omega, U) \\
& \text{and } \Sigma' = \mathbf{w}(C, H_b, \overline{B}_b \triangleright (s_b)_{\overline{f}_b}, \perp, S) \cdot \overline{\mathbf{w}(C, H', \overline{B}' \triangleright (s')_{\overline{f}'}, \omega', U)} \cdot (C, H'', \overline{B}'' \triangleright (\llbracket s'' \rrbracket^{ss})_{\overline{f}'', f''}, 0, U) \\
& \text{and } \mathbf{n}' \leq \mathbf{n} + \omega \\
& \text{and } \mathbf{f}, \mathbf{f}'' \in \overline{f}_c \\
& \text{then } (\mathbf{n}, \Sigma) \xrightarrow{\overline{\lambda}^\sigma} (\mathbf{n}', \Sigma') \\
& \text{and } \emptyset \approx \overline{\lambda}^\sigma \\
& \text{and } \cdot \underset{\approx}{\overset{s}{\overline{f}_c}} \overline{D} \Sigma'
\end{aligned}$$

LEMMA Q.15 (COMPILED SPECULATION IS SAFE). (*Proof 34*)

$$\begin{aligned}
& \text{if } \cdot \underset{\approx}{\overset{s}{\overline{f}_c}} \overline{D} \Sigma \\
& \text{and } \Sigma = \mathbf{w}(C, H_b, \overline{B}_b \triangleright (s_b)_{\overline{f}_b}, \perp, S) \cdot \overline{\mathbf{w}(C, H', \overline{B}' \triangleright (s')_{\overline{f}'}, \omega', U)} \cdot (C, H, \overline{B} \triangleright (\llbracket s \rrbracket^{ss})_{\overline{f}, f}, \omega, U) \\
& \text{and } \Sigma' = \mathbf{w}(C, H_b, \overline{B}_b \triangleright (s_b)_{\overline{f}_b}, \perp, S) \cdot \overline{\mathbf{w}(C, H', \overline{B}' \triangleright (s')_{\overline{f}'}, \omega', U)} \cdot (C, H'', \overline{B}'' \triangleright (\llbracket s'' \rrbracket^{ss})_{\overline{f}'', f''}, \omega'', U) \\
& \text{and } \mathbf{f}, \mathbf{f}'' \in \overline{f}_c \\
& \text{then } (\mathbf{n}, \Sigma) \xrightarrow{\overline{\lambda}^\sigma} (\mathbf{n}', \Sigma') \\
& \text{and } \emptyset \approx \overline{\lambda}^\sigma \\
& \text{and either } \cdot \underset{\approx}{\overset{s}{\overline{f}_c}} \overline{D} \Sigma' \\
& \text{or } \omega'' = 0
\end{aligned}$$

THEOREM Q.16 (CORRECTNESS OF THE BACKTRANSLATION FOR SLH). (*Proof 37*)

$$\text{if } A \llbracket \llbracket P \rrbracket^{ss} \rrbracket \rightsquigarrow \overline{\lambda}^\sigma$$

then $\langle\langle A \rangle\rangle_c^f [P] \rightsquigarrow \bar{\lambda}^\sigma$
 and $\bar{\lambda}^\sigma \approx \bar{\lambda}^\sigma$

THEOREM Q.17 (GENERALISED BACKWARD SIMULATION FOR SLH). (*Proof 36*)

if $f \in \bar{f}''$ then $s = \llbracket s \rrbracket^{ss}$ else $s = \langle\langle s \rangle\rangle_c^f$
 and if $f' \in \bar{f}''$ then $s' = \llbracket s' \rrbracket^{ss}$ else $s' = \langle\langle s' \rangle\rangle_c^f$
 and $\Sigma = w(C, H, \bar{B} \triangleright (s; s'')_{\bar{f}, f}, \perp, S)$
 and $\Sigma' = w(C, H', \bar{B}' \triangleright (s'; s'')_{\bar{f}', f'}, \perp, S)$
 and $(n, \Sigma) \xrightarrow{\bar{\lambda}^\sigma} (n', \Sigma')$
 and $\cdot \stackrel{s}{\approx}_{\bar{f}''} \bar{D} \Sigma$
 then $\cdot = C, H, \bar{B} \triangleright (s; s'')_{\bar{f}, f} \xrightarrow{\bar{\lambda}^\sigma} C, H', \bar{B}' \triangleright (s'; s'')_{\bar{f}', f'} = \cdot'$
 and $\bar{\lambda}^\sigma \approx \bar{\lambda}^\sigma$
 and $\exists \bar{D}' \supseteq \bar{D}. \cdot' \stackrel{s}{\approx}_{\bar{f}''} \bar{D}' \Sigma'$

LEMMA Q.18 (COMPILED SPECULATION IS SAFE INTER-PROCEDURALLY). (*Proof 40*)

if $\cdot \stackrel{s}{\approx}_{\bar{f}_c} \bar{D} \Sigma$
 and $\Sigma = w(C, H_b, \bar{B}_b \triangleright (s_b)_{\bar{f}_b}, \perp, S) \cdot \overline{(C, H', \bar{B}' \triangleright (s')_{\bar{f}'}, \omega', U)} \cdot (C, H, \bar{B} \triangleright (\llbracket s \rrbracket_n^s)_{\bar{f}, f}, \omega, U)$
 and $\Sigma' = w(C, H_b, \bar{B}_b \triangleright (s_b)_{\bar{f}_b}, \perp, S) \cdot \overline{(C, H', \bar{B}' \triangleright (s')_{\bar{f}'}, \omega', U)} \cdot (C, H'', \bar{B}'' \triangleright (\llbracket s'' \rrbracket_n^s)_{\bar{f}'', f''}, \omega'', U)$
 and $f, f' \in \bar{f}_c$
 then $(n, \Sigma) \xrightarrow{\bar{\lambda}^\sigma} (n', \Sigma')$
 and $\emptyset \approx \bar{\lambda}^\sigma$
 and either $\cdot \stackrel{s}{\approx}_{\bar{f}_c} \bar{D} \Sigma'$
 or $\omega'' = \mathbf{0}$

R A COMPLETE INSIGHT ON THE PROOFS

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

To prove that a compiler is *RSSC* we need to backtranslate target attackers **A** into source ones **A**. Our setup has very similar source and target languages to enable this kind of backtranslation; the alternative would have been to rely on the target trace in order to build **A**. In this case, the backtranslation function ($\llbracket \cdot \rrbracket$) homomorphically translates target heaps, functions, statements etc. into source ones (see Appendix O.1). Since our compilers are devised for essentially the same languages (or at least, languages with the same notions of attacker), we can define a single backtranslation to use for the security of all compilers we define.

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

- we first set up a cross-language relation between source and target states;
- we prove that initial states are related;
- we prove that reductions preserve this relation and generate related traces.

We reason about reductions depending on whether the pc that is triggering the reduction is in attacker or in program code.

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

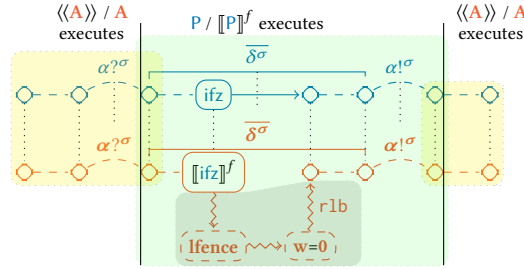


Figure 3: A diagram depicting the proof that $\llbracket \cdot \rrbracket^f$ is *RSSC*.

We depict our proof approach for $\llbracket \cdot \rrbracket^f$ in Figure 3. The top half of the picture represents the source reductions: source states (circles) perform multiple steps (dashed lines) and reduce producing traces (annotations on reductions). We highlight the single reduction caused by the execution of an `ifz` statement since that has relevance in the target language.

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

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

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

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

To reason about component code, we adapt a reasoning commonly used in compiler correctness results [12, 39]. That is, if **s** steps and emits a trace, then $\llbracket s \rrbracket^f$ does one or more steps and emits a trace such that

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

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

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

This is the part where the proofs of the SLH-related countermeasures gets more complicated (Figure 4), though the general structure remains unchanged. In compiled code speculation is not rolled back immediately after the `then` or `else` branch start executing. Instead,

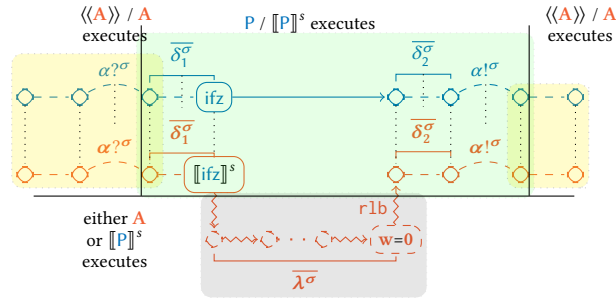


Figure 4: A diagram depicting the proof that $[·]^S$, $[·]^{SS}$ and $[·]^S_n$ are RSSC.

execution can continue for ω steps, spanning both attacker and compiled code and generating a trace $\overline{\lambda^\sigma}$. Our proof here relies on an auxiliary lemma stating the following:

- in the target, speculation lasts at most ω steps and then it will be rolled back;
- after the rollback, the strong state relation we need is reinstated;
- during this speculation any trace produced in the target is related to the empty source trace.

This is needed because such a relation is only possible when all actions in the target trace $\overline{\lambda^\sigma}$ are tainted **S**: i.e., they do not leak.

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

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

As already shown, both conditions hold for $[·]^S$, $[·]^{SS}$ and $[·]^S_n$, so we can conclude that they are RSSC.

R.1 Failing RSSC Proofs

When a countermeasure is not RSSC we can use the insights of its failed proof to understand whether it is also not RSNIP. In fact, while MSVC was already known to be insecure, this was not true for SLH. When we modelled vanilla SLH and started proving RSSC, the proof broke in the “gray area”. While this does not directly mean that SLH is insecure, the way the proof broke provided insights on the insecurity of SLH. Concretely, we were not able to show that the property on speculating target states holds when speculating reductions are done and this led to Examples N.6 and N.7. We believe the insights of this proof technique can guide proofs of (in)security of other countermeasures too.

S PROOFS FOR COUNTERMEASURES AND CRITERIA

PROOF OF THEOREM M.3 (RSSC IMPLIES RSSP).

We have (HPSSC)

$$\vdash \llbracket \cdot \rrbracket : \text{RSSC} \stackrel{\text{def}}{=} \forall P, A, \bar{\lambda}^\sigma, \exists A, \bar{\lambda}^\sigma. \\ \text{if } A \llbracket \llbracket P \rrbracket \rrbracket \rightsquigarrow \bar{\lambda}^\sigma \text{ then } A [P] \rightsquigarrow \bar{\lambda}^\sigma \text{ and } \bar{\lambda}^\sigma \approx \bar{\lambda}^\sigma$$

We need to prove:

$$\forall P. \text{if } \forall A. \forall \bar{\lambda}^\sigma \in \text{Beh}(A [P]). \forall \alpha^\sigma \in \bar{\lambda}^\sigma. \sigma \equiv S \\ \text{then } \forall A. \forall \bar{\lambda}^\sigma \in \text{Beh}(A \llbracket \llbracket P \rrbracket \rrbracket). \forall \alpha^\sigma \in \bar{\lambda}^\sigma. \sigma \equiv S$$

We proceed by contradiction:

$$\forall A. \forall \bar{\lambda}^\sigma \in \text{Beh}(A [P]). \forall \alpha^\sigma \in \bar{\lambda}^\sigma. \sigma \equiv S \text{ (HPS)} \\ \text{and } \exists A. \exists \bar{\lambda}^\sigma \in \text{Beh}(A \llbracket \llbracket P \rrbracket \rrbracket). \exists \alpha^\sigma \in \bar{\lambda}^\sigma \text{ (HPT)} \\ \text{and } \sigma \equiv U \text{ HPU}$$

We instantiate HPSSC with HPS and HPT so we get that $\bar{\lambda}^\sigma \approx \bar{\lambda}^\sigma$.

By definition of \approx we have two main cases:

- source and target actions are the same Rules Trace-Relation-Same-Act and Trace-Relation-Same-Heap.
By Rules Action Relation - call to Action Relation - write and by Theorem 3.9, we conclude that all target taint is **S**, which contradicts (HPU);
- there is no source action and a single target one Rules Trace-Relation-Safe-Act to Trace-Relation-Rollback
By Rules Action Relation - epsi alpha to Action Relation - rlb, the target taint is **S**, which contradicts (HPU)

Having found a contradiction in all cases, this theorem holds. \square

PROOF OF THEOREM M.4 (RSSP IMPLIES RSSC).

We have (RSSP)

$$\forall P. \text{if } \forall A. \forall \bar{\lambda}^\sigma \in \text{Beh}(A [P]). \forall \alpha^\sigma \in \bar{\lambda}^\sigma. \sigma \equiv S \\ \text{then } \forall A. \forall \bar{\lambda}^\sigma \in \text{Beh}(A \llbracket \llbracket P \rrbracket \rrbracket). \forall \alpha^\sigma \in \bar{\lambda}^\sigma. \sigma \equiv S$$

We need to prove:

$$\forall P, A, \bar{\lambda}^\sigma, \exists A, \bar{\lambda}^\sigma. \\ \text{if } A \llbracket \llbracket P \rrbracket \rrbracket \rightsquigarrow \bar{\lambda}^\sigma \text{ then } A [P] \rightsquigarrow \bar{\lambda}^\sigma \text{ and } \bar{\lambda}^\sigma \approx \bar{\lambda}^\sigma$$

We proceed by contradiction, assuming that the target reduction is not related to the source one:

$$\exists A, \bar{\lambda}^\sigma, \forall A, \bar{\lambda}^\sigma. \\ A \llbracket \llbracket P \rrbracket \rrbracket \rightsquigarrow \bar{\lambda}^\sigma \text{ and } A [P] \rightsquigarrow \bar{\lambda}^\sigma \text{ (HPSR) and } \bar{\lambda}^\sigma \not\approx \bar{\lambda}^\sigma \text{ (HPA)}$$

We analyse HPA.

By \approx we determine when the two traces are not related.

By the universal quantification over **A** rules out all cases when there are trivial mismatches: a source call/ret and a target ret/call, calls to two different functions, a source write/read and a target read/write.

So we are left with these cases:

- a target call matched by a source call with unrelated argument
This contradicts Rules Action Relation - call and Action Relation - callback.
- a target call matched by a source call with related arguments but with different taint.
In this case, since all source taints are **S**, we conclude that we have a target action that is tagged as **U**.
- a target return matched by a source return with unrelated heaps
This contradicts Rules Action Relation - return and Action Relation - returnback.
- a target read/write matched by no action in the source
This contradicts Rules Action Relation - epsi alpha and Action Relation - epsi heap.
- a target read/write matched by a source read/write to a different location.
This contradicts Rules Action Relation - read and Action Relation - write.

- a target read/write matched by a source read/write to the same location but with different taint.

In this case, since all source taints are S , we conclude that we have a target action that is tagged as U .

We can therefore conclude that $\exists A. \exists \bar{\lambda}^\sigma \in \text{Beh}(A \llbracket P \rrbracket). \exists \alpha^\sigma \in \bar{\lambda}^\sigma. \sigma \equiv U$ (HPU).

We instantiate RSSP with HPSR and conclude

$$\forall A. \forall \bar{\lambda}^\sigma \in \text{Beh}(A \llbracket P \rrbracket).$$

$$\forall \alpha^\sigma \in \bar{\lambda}^\sigma. \sigma \equiv S \text{ (HPS)}$$

We obtain the contradiction between HPU and HPS. □

PROOF OF THEOREM M.5 (RSSC AND RSSP ARE EQUIVALENT).

By Theorem M.3 (RSSC implies RSSP) and Theorem M.4 (RSSP implies RSSC). □

PROOF OF THEOREM M.10 (RSSC⁻ IMPLIES RSSP⁻).

Trivial adaptation of Theorem M.3 (RSSC implies RSSP). □

PROOF OF THEOREM M.11 (RSSP⁻ IMPLIES RSSC⁻).

Trivial adaptation of Theorem M.4 (RSSP implies RSSC). □

PROOF OF THEOREM M.12 (RSSC⁻ AND RSSP⁻ ARE EQUIVALENT).

By Theorem M.10 (RSSC⁻ implies RSSP⁻) and Theorem M.11 (RSSP⁻ implies RSSC⁻). □

PROOF OF THEOREM O.1 (THE LFENCE COMPILER IS RSSC⁺).

Instantiate A with $\langle\langle A \rangle\rangle_c^f$.

This holds by Theorem O.3 (Correctness of the Backtranslation for lfence). □

PROOF OF THEOREM O.2 (ALL LFENCE-COMPILED PROGRAMS ARE RSS (L)).

By Theorem 3.9 we have HPS: $\forall P. \vdash P : \text{RSS}$.

By Theorem O.1 we have HPC: $\vdash \llbracket \cdot \rrbracket^f : \text{RSSC}$.

By Theorem M.3 with HPC we have HPP: $\vdash \llbracket \cdot \rrbracket^f : \text{RSSP}$.

By Definition M.1 (Robust Speculative Safety-Preserving Compiler (RSSP)) of HPP with HPS we conclude that $\forall P. \vdash \llbracket P \rrbracket^f : \text{RSS}$. □

PROOF OF THEOREM O.3 (CORRECTNESS OF THE BACKTRANSLATION FOR LFENCE).

This holds by Lemma O.10 (Initial States are Related) and by Theorem O.4 (Generalised Backward Simulation for lfence). □

PROOF OF THEOREM O.4 (GENERALISED BACKWARD SIMULATION FOR LFENCE).

We proceed by induction on the reduction $\xrightarrow{\bar{\lambda}^\sigma}$

Base no reductions, this case is trivial

Inductive we have n target steps to states $\Sigma_i = C, H_i, \bar{B}_i \triangleright (s_i; s'_i)_{\bar{f}_i, f_i}$ and $\dot{\Sigma}_i = C, H_i, \bar{B}_i \triangleright (s_i; s'_i)_{\bar{f}_i, f_i}$

where $\dot{\Sigma}_i \stackrel{s}{\approx} \Sigma_i$ and the traces produced so far are related via \approx .

We proceed by case analysis on f_i

in $\overline{f''}$ (in the compiled component) By case analysis on f'

in $\overline{f''}$ (in the compiled component) This holds by Theorem O.7 (Backward Simulation for Compiled Steps in Ifence);

not in $\overline{f''}$ (in the context) This happens by the execution of two statements:

call This is a call from a compiled function to a context function.

In this case we have that $\Sigma_i = \mathbf{C}, \mathbf{H}_i, \overline{\mathbf{B}}_i \cdot \mathbf{B}_i \triangleright (\llbracket \text{call } f' \text{ e} \rrbracket^f; s_i'')_{\overline{f}_i, f_i}$

so by Rule E-T-speculate-action with Rule E-L-call(in the target ofc) we know

$$\Sigma_i \xrightarrow{(\text{call } f' \llbracket v \rrbracket^f?)^S} \Sigma' = \mathbf{w}, (\mathbf{C}, \mathbf{H}', \overline{\mathbf{B}}' \cdot \mathbf{x} \mapsto \llbracket v \rrbracket^f \triangleright (s_f; s_i'')_{\overline{f}', f'}, \perp, \mathbf{S})$$

and we know $\mathbf{B}_i \triangleright \llbracket e \rrbracket^f \downarrow \llbracket v \rrbracket^f$ (HPE)

where $f(\mathbf{x}) \mapsto s_f \in \mathbf{C}$ and $\overline{\mathbf{B}}' = \overline{\mathbf{B}}_i \cdot \mathbf{B}_i$ and $\overline{f}' = \overline{f}_i \cdot f_i$ and $\mathbf{H}' = \mathbf{H}_i$

and the taint is \mathbf{S} by definition of \sqcap since by \approx the pc taint is \mathbf{S} .

By Lemma O.8 (Backward Simulation for Backtranslated Expressions in Ifence), HPE yields $\mathbf{B}_i \triangleright e \downarrow v$ (HPSE)

We have that $\cdot_i = \mathbf{C}, \mathbf{H}_i, \overline{\mathbf{B}}_i \cdot \mathbf{B}_i \triangleright (\text{call } f \text{ e}; s_i'')_{\overline{f}_i, f_i}$

by definition of $\langle\langle \cdot \rangle\rangle_c^f$ we know that $f(\mathbf{x}) \mapsto \langle\langle s_f \rangle\rangle_c^f \in \mathbf{C}$

we take $\overline{\mathbf{B}}' = \overline{\mathbf{B}}_i \cdot \mathbf{B}_i$ so that by hypothesis we have that $\overline{\mathbf{B}}' \stackrel{\beta}{\approx} \overline{\mathbf{B}}'$ (HPB)

we take $\overline{f}' = \overline{f}_i \cdot f_i$ so that by hypothesis we have that $\overline{f}' \equiv \overline{f}'$ (HPF)

we take $\mathbf{H}' = \mathbf{H}_i$ so that by hypothesis we have that $\mathbf{H}' \stackrel{H}{\approx} \mathbf{H}'$ (HPH)

By Rule E-L-call(in the source) with HPSE and the hypotheses above, we have that

$$\cdot_i \xrightarrow{(\text{call } f' v?)^S} \cdot' = \mathbf{C}, \mathbf{H}', \overline{\mathbf{B}}' \cdot \mathbf{x} \mapsto v \triangleright (\langle\langle s_f \rangle\rangle_c^f; s_i'')_{\overline{f}', f'}$$

We need to prove that:

- $\cdot' \stackrel{S}{\approx} \Sigma'$, which by Rule States means proving that:
 - $\overline{\mathbf{B}}' \cdot \mathbf{x} \mapsto v \stackrel{\beta}{\approx} \overline{\mathbf{B}}' \cdot \mathbf{x} \mapsto \llbracket v \rrbracket^f$, which holds by (HPB) and by Rule Binding - ind with Rule Value - num with Lemma O.11 (A Value is Related to its Compilation for Ifence);
 - $\overline{f}' \equiv \overline{f}'$, which holds by (HPF);
 - $\mathbf{C} \stackrel{c}{\approx} \overline{\mathbf{C}}$, which holds by \approx of the initial states since components do not change;
 - $\mathbf{H}' \stackrel{H}{\approx} \mathbf{H}'$, which holds by (HPH)
- $(\text{call } f' v?)^S \stackrel{A}{\approx} (\text{call } f' \llbracket v \rrbracket^f?)^S$, which by Rule Action Relation - call means proving that:
 - $f' \equiv f'$, which holds;
 - $v \stackrel{v}{\approx} \llbracket v \rrbracket^f$, which holds by Lemma O.11;
 - $\mathbf{S} \equiv \mathbf{S}$

so this case holds.

ret This is a return from a compiled function to a context function.

This is the dual of the case below for return from context to code.

not in $\overline{f''}$ (in the context) By case analysis on f'

in $\overline{f''}$ (in the compiled component) This happens by the execution of two statements:

call This is a call from a context function to a compiled function.

This is the dual of the case for call above.

ret This is a return from a context function to a compiled function.

In this case we have that $\Sigma_i = \mathbf{C}, \mathbf{H}_i, \overline{\mathbf{B}}_i \cdot \mathbf{B}_i \triangleright (\text{return}; \llbracket s_f \rrbracket^f; s_i'')_{\overline{f}_i, f_i}$

so by Rule E-T-speculate-action with Rule E-L-return(in the target ofc) we know

$$\Sigma_i \xrightarrow{(\text{ret})^S} \Sigma' = \mathbf{w}, (\mathbf{C}, \mathbf{H}_i, \overline{\mathbf{B}}_i \triangleright (\llbracket s_f \rrbracket^f; s_i'')_{\overline{f}_i, f'}, \perp, \mathbf{S})$$

We have that $\cdot_i = \mathbf{C}, \mathbf{H}_i, \overline{\mathbf{B}}_i \cdot \mathbf{B}_i \triangleright (\text{return}; s_f; s_i'')_{\overline{f}_i, f_i}$

By Rule E-L-call(in the source), we have that

$$\cdot_i \xrightarrow{(\text{ret})^S} \cdot' = \mathbf{C}, \mathbf{H}_i, \overline{\mathbf{B}}_i \triangleright (s_f; s_i'')_{\overline{f}_i}$$

We need to prove that:

- $\cdot' \stackrel{S}{\approx} \Sigma'$, which by Rule States means proving that:
 - $\overline{\mathbf{B}}_i \stackrel{\beta}{\approx} \overline{\mathbf{B}}_i$, which holds by hypothesis;
 - $\overline{f}_i \equiv \overline{f}_i$, which holds by hypothesis;
 - $\mathbf{C} \stackrel{c}{\approx} \overline{\mathbf{C}}$, which holds by \approx of the initial states since components do not change;
 - $\mathbf{H}_i \stackrel{H}{\approx} \mathbf{H}_i$, which holds by hypothesis
 - $(\text{ret})^S \stackrel{A}{\approx} (\text{ret})^S$, which by Rule Action Relation - return is trivially true.
- so this case holds.

not in $\overline{f'}$ (in the context) This holds by Lemma O.9 (Backward Simulation for Backtranslated Statements);

□

♠

PROOF OF THEOREM O.7 (BACKWARD SIMULATION FOR COMPILED STEPS IN LFENCE).

By contradiction, assume the source ends up in a different state Σ'' such that $\Sigma'' \neq \Sigma'$.

By Lemma O.6 (Forward Simulation for Compiled Statements in lfence) we have that the target also ends up in state Σ'' such that $\Sigma'' \stackrel{s}{\approx_{\overline{f'}}} \Sigma'$ and such that $\Sigma'' \neq \Sigma'$ (HPC).

So we have that Σ reaches both Σ' and Σ'' .

Since the semantics is deterministic, it must be that $\Sigma'' = \Sigma'$ (HPE).

We now have a contradiction between HPC and HPE.

□

♠

PROOF OF LEMMA O.5 (FORWARD SIMULATION FOR EXPRESSIONS IN LFENCE).

Trivial induction on e .

□

♠

PROOF OF LEMMA O.6 (FORWARD SIMULATION FOR COMPILED STATEMENTS IN LFENCE).

The proof proceeds by structural induction on s .

Base skip trivial;

call Two cases arise:

(1) f is defined by the component.

This holds by Lemma O.5 (Forward Simulation for Expressions in lfence) and by relatedness of heaps;

(2) f is defined by the context.

This cannot arise as in the target we don't step to a compiled statement $\llbracket s' \rrbracket^f$.

return Two cases arise:

(1) f is defined by the component.

This holds by relatedness of heaps;

(2) f is defined by the context.

This cannot arise as in the target we don't step to a compiled statement $\llbracket s' \rrbracket^f$.

write by Lemma O.5 (Forward Simulation for Expressions in lfence) and Rule Action Relation - write 2;

private write by Lemma O.5 (Forward Simulation for Expressions in lfence) and Rule Action Relation - write.

Inductive sequencing by IH;

letin by IH and Lemma O.5 (Forward Simulation for Expressions in lfence);

if zero by IH and Lemma O.5 (Forward Simulation for Expressions in lfence).

By definition of $\llbracket \cdot \rrbracket^f$, this is the only case where we need to account for multiple steps in the target, since there is an **lfence**.

By Rule E-T-lfence, a rollback is triggered via Rule E-T-speculate-rollback.

Relatedness of states is therefore ensured, while relatedness of traces is ensured by: Rule Trace-Relation-Rollback and Rule Action Relation - rlb.

let read by IH and Lemma O.5 (Forward Simulation for Expressions in lfence) and Rule Action Relation - read;

let private read by IH and Lemma O.5 (Forward Simulation for Expressions in lfence) and Rule Action Relation - read;

conditional letin by IH and Lemma O.5 (Forward Simulation for Expressions in lfence).

□

♠

PROOF OF LEMMA O.8 (BACKWARD SIMULATION FOR BACKTRANSLATED EXPRESSIONS IN LFENCE).

The proof proceeds by structural induction on e .

Base number trivial;

variable this follows from the relatedness of stack frames;

Inductive ops by IH;

bops by IH.

□



PROOF OF LEMMA O.9 (BACKWARD SIMULATION FOR BACKTRANSLATED STATEMENTS).

The proof proceeds by structural induction on s .

Base skip trivial;

call Two cases arise:

(1) f is defined by the component.

This holds by Lemma O.8 (Backward Simulation for Backtranslated Expressions in lfence) and relatedness of heaps.

(2) f is defined by the context.

This cannot arise as in the source we don't step to a backtranslated statement $\langle\langle s' \rangle\rangle_c^f$.

return Two cases arise:

(1) f is defined by the component.

This holds by relatedness of heaps.

(2) f is defined by the context.

This cannot arise as in the source we don't step to a backtranslated statement $\langle\langle s' \rangle\rangle_c^f$.

write by Lemma O.8 (Backward Simulation for Backtranslated Expressions in lfence).

One complexity is showing that the action taint $\sigma = S$, but this follows from \lesssim which tells that the pc taint is S .

private write this cannot arise by Definition J.2;

lfence trivial.

Inductive sequencing by IH;

letin by IH and Lemma O.8 (Backward Simulation for Backtranslated Expressions in lfence);

if zero by IH and Lemma O.8 (Backward Simulation for Backtranslated Expressions in lfence).

In this case, Rule E-T-speculate-if-att is not applicable, so we cannot speculate.

let read by IH and Lemma O.8 (Backward Simulation for Backtranslated Expressions in lfence).

One complexity is showing that the action taint $\sigma = S$, but this follows from \lesssim which tells that the pc taint is S .

let private read this cannot arise by Definition J.2;

conditional letin by IH and Lemma O.8 (Backward Simulation for Backtranslated Expressions in lfence).

□



PROOF OF LEMMA O.10 (INITIAL STATES ARE RELATED).

By definition of \lesssim , we need to prove that:

- bindings are related via $\overset{B}{\approx}$ (by Rule Bindings, then Rule Binding - ind where $0 \overset{V}{\approx} 0$ by Rule Value - num);
- heaps are related via $\overset{H}{\approx}$:
 - for the program heap, Rule Target Bindings ok sing tells us that its domain is negative numbers, and the negative heap relatedness holds by Lemma O.12 (A Heap is Related to its Compilation for lfence);
 - for the context heap, Definition J.2 tells us that its domain is natural numbers, so this holds by Lemma O.15;
- components are related via $\overset{C}{\approx}$ by simple inspection of $\llbracket \cdot \rrbracket^f$ and $\langle\langle \cdot \rangle\rangle_c^f$;
- the target taint is S : this holds by Rule T-Initial State;
- the target window is \perp : this holds by Rule T-Initial State;

□



PROOF OF LEMMA O.11 (A VALUE IS RELATED TO ITS COMPILATION FOR LFENCE).

Trivial analysis of the compiler.

□



PROOF OF LEMMA O.12 (A HEAP IS RELATED TO ITS COMPILATION FOR LFENCE).

Trivial analysis of the compiler.

□



PROOF OF LEMMA O.13 (A VALUE IS RELATED TO ITS BACKTRANSLATION).

Trivial analysis of the backtranslation.

□

♣

PROOF OF LEMMA O.14 (A TAINT IS RELATED TO ITS BACKTRANSLATION). Trivial analysis of the backtranslation.	□
--	---

♣

PROOF OF LEMMA O.15 (A HEAP IS RELATED TO ITS BACKTRANSLATION). Trivial analysis of the backtranslation with Lemmas O.13 and O.14.	□
---	---

♣

PROOF OF THEOREM O.16 (THE LFENCE COMPILER IS <i>RSSC</i> WITH MORE LEAKS). This holds by an adaptation of Theorem O.3 (Correctness of the Backtranslation for lfence) to the extra reduction, which in turn holds by Lemma O.10 (Initial States are Related) and by an adaptation of Theorem O.4 (Generalised Backward Simulation for lfence) to the extra reduction, which in turn holds by Lemma O.8 (Backward Simulation for Backtranslated Expressions in lfence) and by an adaptation of Theorem O.7 (Backward Simulation for Compiled Steps in lfence) to the extra reduction, where the additional reduction must be considered. Since that reduction is not triggered, these adaptations trivially hold.	Instantiate A with $\langle\langle A \rangle\rangle_c^f$. □
--	---

♣

PROOF OF LEMMA Q.3 (INITIAL STATES ARE RELATED FOR SLH). Analogous to the proof of Lemma O.10 (Initial States are Related) but with Lemma Q.4 (A Value is Related to its Compilation for SLH) and Lemma Q.5 (A Heap is Related to its Compilation for SLH).	□
--	---

♣

PROOF OF LEMMA Q.4 (A VALUE IS RELATED TO ITS COMPILATION FOR SLH). Trivial analysis of $\llbracket \cdot \rrbracket^{ss}$.	□
---	---

♣

PROOF OF LEMMA Q.5 (A HEAP IS RELATED TO ITS COMPILATION FOR SLH). Trivial analysis of $\llbracket \cdot \rrbracket^{ss}$ given that -1 is allocated by the compiler and that all addresses are shifted by 1, so they account for Rule Heap-negative.	□
--	---

♣

PROOF OF LEMMA Q.6 (FORWARD SIMULATION FOR EXPRESSIONS IN SLH).	Trivial induction on e . □
---	---------------------------------

♣

PROOF OF LEMMA Q.7 (FORWARD SIMULATION FOR COMPILED STATEMENTS IN SLH). The proof proceeds by induction on s .	
---	--

Base skip Trivial.

call f We have two cases:

- f is component-defined.
This follows from Lemma Q.6 (Forward Simulation for Expressions in SLH).
- f is context-defined.
This is a contradiction because the ending statement is not a compiled one.

assign This follows from Lemma Q.6 (Forward Simulation for Expressions in SLH).

private assign This follows from Lemma Q.6 (Forward Simulation for Expressions in SLH).

return Trivial.

Inductive sequencing By IH.

let-in By IH and Lemma Q.6 (Forward Simulation for Expressions in SLH).

if then else In this case we have this source reduction, wlog assume HE $B \triangleright e \downarrow \text{true} : \sigma$

$$C, H, \bar{B} \cdot B \triangleright \text{ifz } e \text{ then } s \text{ else } s'; s'' \xrightarrow{(\text{if}(0))^S} C, H, \bar{B} \cdot B \triangleright s; s''$$

By Lemma Q.6 (Forward Simulation for Expressions in SLH) with HE we get HET : $B \triangleright \llbracket e \rrbracket^{SS} \downarrow \llbracket \text{true} \rrbracket^{SS} : \sigma$.
In the target we have these reductions (by HET):

$$\begin{aligned} \Sigma = & \\ & \mathbf{w}(C, H, \bar{B} \cdot B, \perp, S) \triangleright \llbracket \text{ifz } e \text{ then } s \text{ else } s' \rrbracket^{SS}; s'' \\ \equiv & \mathbf{w}(C, H, \bar{B} \cdot B, \perp, S) \triangleright \text{let } x_g = \llbracket e \rrbracket^{SS} \text{ in} & ; s'' \\ & \quad \text{let } \text{pr} = \text{rd}_{\text{pr}} - 1 \text{ in} \\ & \quad \text{let } x_g = 0 \text{ (if } \text{pr} \text{) in} \\ & \quad \text{ifz } x_g \text{ then let } x = \text{rd}_{\text{pr}} - 1 \text{ in } -1 :=_{\text{pr}} x \vee \neg x_g; \llbracket s \rrbracket^{SS} \\ & \quad \text{else let } x = \text{rd}_{\text{pr}} - 1 \text{ in } -1 :=_{\text{pr}} x \vee x_g; \llbracket s' \rrbracket^{SS} \\ & \quad \text{let } B' = B \cdot x_g \mapsto \text{true} : \sigma \\ \rightsquigarrow & \mathbf{w}(C, H, \bar{B} \cdot B', \perp, S) \triangleright \text{let } \text{pr} = \text{rd}_{\text{pr}} - 1 \text{ in} & ; s'' \\ & \quad \text{let } x_g = 0 \text{ (if } \text{pr} \text{) in} \\ & \quad \text{ifz } x_g \text{ then let } x = \text{rd}_{\text{pr}} - 1 \text{ in } -1 :=_{\text{pr}} x \vee \neg x_g; \llbracket s \rrbracket^{SS} \\ & \quad \text{else let } x = \text{rd}_{\text{pr}} - 1 \text{ in } -1 :=_{\text{pr}} x \vee x_g; \llbracket s' \rrbracket^{SS} \\ & \quad \text{since } H(-1) \mapsto \text{false} : S \\ & \quad \text{let } B'' = B' \cup \text{pr} \mapsto \text{false} : S \\ \rightsquigarrow & \mathbf{w}(C, H, \bar{B} \cdot B'', \perp, S) \triangleright \text{let } x_g = 0 \text{ (if } \text{pr} \text{) in} & ; s'' \\ & \quad \text{ifz } x_g \text{ then let } x = \text{rd}_{\text{pr}} - 1 \text{ in } -1 :=_{\text{pr}} x \vee \neg x_g; \llbracket s \rrbracket^{SS} \\ & \quad \text{else let } x = \text{rd}_{\text{pr}} - 1 \text{ in } -1 :=_{\text{pr}} x \vee x_g; \llbracket s' \rrbracket^{SS} \\ & \quad \text{since } \text{pr} \mapsto \text{false} : S \\ \rightsquigarrow & \mathbf{w}(C, H, \bar{B} \cdot B'', \perp, S) \triangleright \text{ifz } x_g \text{ then let } x = \text{rd}_{\text{pr}} - 1 \text{ in } -1 :=_{\text{pr}} x \vee \neg x_g; \llbracket s \rrbracket^{SS}; s'' \\ & \quad \text{else let } x = \text{rd}_{\text{pr}} - 1 \text{ in } -1 :=_{\text{pr}} x \vee x_g; \llbracket s' \rrbracket^{SS} \end{aligned}$$

By Rule E-T-speculate-if-att

$$\begin{aligned} \rightsquigarrow & \mathbf{w}(C, H, \bar{B} \cdot B'', \perp, S) \triangleright \text{let } x = \text{rd}_{\text{pr}} - 1 \text{ in } -1 :=_{\text{pr}} x \vee \neg x_g; \llbracket s \rrbracket^{SS}; s'' \\ & \quad \cdot (C, H, \bar{B} \cdot B'', w, U) \triangleright \text{let } x = \text{rd}_{\text{pr}} - 1 \text{ in } -1 :=_{\text{pr}} x \vee x_g; \llbracket s' \rrbracket^{SS}; s'' \\ & \quad \text{since } H(-1) \mapsto \text{false} : S \\ & \quad \text{let } B''' = B'' \cdot x \mapsto \text{false} : S \\ \rightsquigarrow & \mathbf{w}(C, H, \bar{B} \cdot B'', \perp, S) \triangleright \text{let } x = \text{rd}_{\text{pr}} - 1 \text{ in } -1 :=_{\text{pr}} x \vee \neg x_g; \llbracket s \rrbracket^{SS}; s'' \\ & \quad \cdot (C, H, \bar{B} \cdot B''', w, U) \triangleright -1 :=_{\text{pr}} x \vee x_g; \llbracket s' \rrbracket^{SS}; s'' \\ & \quad \text{since } B''' \triangleright x \vee x_g \downarrow \text{true} : S \text{ let } H' = H \cup -1 \mapsto \text{true} : S \\ \rightsquigarrow & \mathbf{w}(C, H, \bar{B} \cdot B'', \perp, S) \triangleright \text{let } x = \text{rd}_{\text{pr}} - 1 \text{ in } -1 :=_{\text{pr}} x \vee \neg x_g; \llbracket s \rrbracket^{SS}; s'' \\ & \quad \cdot (C, H', \bar{B} \cdot B''', w, U) \triangleright \llbracket s' \rrbracket^{SS}; s'' \end{aligned}$$

Call this last state Σ_i .

Let $\bar{D} = \bar{D} \cdot D$, consider $\bar{D}' = \bar{D} \cdot D, x_g$.

We can easily prove that $\bar{D}' \stackrel{s}{\approx}_{\bar{c}} \Sigma_i$ since by HP the first states are related at \bar{c} and the speculating states in Σ_i only have safe bindings.

Note: this is where the proof for $\llbracket \cdot \rrbracket_3^S$ breaks, because there we need a stronger invariant, namely that all bindings are safe when speculating, and we can't prove that here because we inherit **B** with all it can have, including **U** bindings. If we do not require the bindings to all be safe when speculating, the speculation lemmas break (Lemma Q.15 (Compiled Speculation is Safe)): without this we cannot show that actions are safe.

By Lemma Q.10 (Any Speculation from Related States is Safe) we know that:

$$(n'', \Sigma_i) \xrightarrow{\bar{\lambda}^\sigma} (n', \Sigma'')$$

Where

$$\Sigma'' = w(C, H, \bar{B} \cdot B'', \perp, S \triangleright \text{let } x = \text{rd}_{\text{pr}} -1 \text{ in } -1 :=_{\text{pr}} x \vee \neg \text{true}; \llbracket s \rrbracket^{\text{SS}}; s'')$$

and (HL): $\emptyset \approx \bar{\lambda}^\sigma$

and (HS): $\cdot \xrightarrow[\approx_{f_c}]{s \bar{D}'} \Sigma''$

and that the first element in the stack of states in Σ'' is the same as the first element in the stack of states of Σ' , which is still \approx with \cdot .

The reductions proceed as follows:

$$\begin{aligned} & \Sigma'' \\ & \xrightarrow{\text{(read(-1))^S}} (C, H, \bar{B} \cdot B'', \perp, S \triangleright -1 :=_{\text{pr}} \text{false} \vee \neg \text{true}; \llbracket s \rrbracket^{\text{SS}}; s'') \\ & \quad \text{since } _ \triangleright \text{false} \vee \neg \text{true} \downarrow \text{false} : S \\ & \xrightarrow{\text{(write(-1))^S}} (C, H, \bar{B} \cdot B'', \perp, S \triangleright \llbracket s \rrbracket^{\text{SS}}; s'') \end{aligned}$$

At this point, we have this target trace:

$$\begin{aligned} (n, \Sigma) & \xrightarrow{\text{(read(-1))^S} \cdot \text{(if(0))^S} \cdot \text{(read(-1))^S} \cdot \text{(write(-1))^S}} (n'', \Sigma_i) \xrightarrow{\bar{\lambda}^\sigma} (n', \Sigma'') \\ & \xrightarrow{\text{(read(-1))^S} \cdot \text{(write(-1))^S}} (n', \Sigma') \\ \text{i.e.,} \\ (n, \Sigma) & \xrightarrow{\text{(read(-1))^S} \cdot \text{(if(0))^S} \cdot \text{(read(-1))^S} \cdot \text{(write(-1))^S} \cdot \bar{\lambda}^\sigma \cdot \text{(read(-1))^S} \cdot \text{(write(-1))^S}} (n', \Sigma') \end{aligned}$$

We need to show that this trace is \approx to the source trace $(\text{if}(0))^S$.

This holds because

- the first read action can be dropped by Rule Trace-Relation-Safe-Heap and Rule Action Relation - epsi heap;
- the if actions are related by Rule Trace-Relation-Same-Heap and Rule Action Relation - if;
- the second read action can be dropped by Rule Trace-Relation-Safe-Heap and Rule Action Relation - epsi heap;
- the first write action can be dropped by Rule Trace-Relation-Safe-Heap and Rule Action Relation - epsi heap;
- the trace $\bar{\lambda}^\sigma$ can be dropped by HL;
- the third read action can be dropped by Rule Trace-Relation-Safe-Heap and Rule Action Relation - epsi heap;
- the second write action can be dropped by Rule Trace-Relation-Safe-Heap and Rule Action Relation - epsi heap.

So this case holds.

read By IH and Lemma Q.6 (Forward Simulation for Expressions in SLH).

private read By IH and Lemma Q.6 (Forward Simulation for Expressions in SLH). □

♣

PROOF OF THEOREM Q.8 (BACKWARD SIMULATION FOR COMPILED STATEMENTS IN SLH).

Analogous to Theorem O.7 (Backward Simulation for Compiled Steps in Ifence) with Lemma Q.7 (Forward Simulation for Compiled Statements in SLH). □

♣

PROOF OF LEMMA Q.9 (EXPRESSION REDUCTIONS WITH SAFE BINDINGS ARE SAFE).

Trivial induction on e , the only nontrivial case is when $e = x$ but this follows from the safety of bindings. □



PROOF OF LEMMA Q.11 (SPECULATION LASTS AT MOST OMEGA).

This proceeds by cases on f and f'' :

Both in \overline{f}_c These are compiled reductions, this holds by Lemma Q.14 (Compiled Speculation Lasts at Most Omega);

Both not in \overline{f}_c These are context reductions, this holds by Lemma Q.12 (Context Speculation Lasts at Most Omega);

$f \in \overline{f}_c$ and $f'' \notin \overline{f}_c$ This is a reduction going from compiled code to context.

We proceed by induction on ω .

The base case is trivial by Rule E-T-init, the inductive case has two cases:

call By analysing the case of $\llbracket \cdot \rrbracket^{ss}$ for call, we have:

let $x_f = \llbracket e \rrbracket^{ss}$ **in let** $x_f = 0$ (if pr) **in call** $f x_f$

We have two cases: $B \triangleright \llbracket e \rrbracket^{ss} \downarrow v : \sigma$ or $B \triangleright \llbracket e \rrbracket^{ss} \downarrow e' : \sigma$, i.e., the execution of $\llbracket e \rrbracket^{ss}$ gets stuck.

The latter case is trivially true by Rule E-T-speculate-rollback-stuck: when speculation gets stuck it gets rolled back.

The former case proceeds as follows.

We have two cases, either there speculation window is long enough ($\omega > 3$) or not.

In the latter case, some of the reductions below happen and then a Rule E-T-speculate-rollback is triggered, so this holds.

Otherwise, if the window is long enough, we have the following.

By HP we have that $H(-1) \mapsto \mathbf{true} : S$ so the code above will step as follows:

(for simplicity we only keep track of the top of the stack of execution states)

$$\begin{aligned}
 & (C, H, \overline{B} \cdot B \triangleright \mathbf{let} \ x_f = \llbracket e \rrbracket^{ss} \ \mathbf{in} \ \mathbf{let} \ x_f = 0 \ \mathbf{(if \ pr)} \ \mathbf{in} \ \mathbf{call} \ f \ x_f), \omega, U \\
 & \text{assuming } B \triangleright \llbracket e \rrbracket^{ss} \downarrow v : \sigma \\
 & \rightarrow (C, H, \overline{B} \cdot B \cdot x_f \mapsto v : \sigma \triangleright \mathbf{let} \ x_f = 0 \ \mathbf{(if \ pr)} \ \mathbf{in} \ \mathbf{call} \ f \ x_f), \omega - 1, U \\
 & \text{since } H(-1) \mapsto \mathbf{true} : S \\
 & \rightarrow (C, H, \overline{B} \cdot B \cdot x_f \mapsto 0 : S \triangleright \mathbf{call} \ f \ x_f), \omega - 2, U \\
 & \text{where } \overline{B}' \text{ is the current stack and the body of } f \text{ is } s \\
 & \xrightarrow{\mathbf{call} \ f \ 0!^S} (C, H, \overline{B}' \cdot x \mapsto 0 : S \triangleright s), \omega - 3, U
 \end{aligned}$$

We need to prove that the new binding is safe (which is true) and that the action is droppable (which holds by Rule Action Relation - epsi alpha), so this case holds.

Note: this is where the proof for $\llbracket \cdot \rrbracket_n^s$ without **lfence** breaks, because there we need a stronger invariant, namely that all bindings start with a variable capturing speculation, and here we cannot set that up correctly.

$\llbracket \cdot \rrbracket_n^s$ with **lfence** goes through because a rollback is triggered, and the stack of bindings goes back to what was related.

ret This is analogous to the point above.

Additionally, we need to prove that the bindings we go back to are all safe.

This trivially holds because a context cannot create unsafe bindings (Lemma Q.13 (Single Context Speculation is Safe)) and the binding created in a call is safe (Rule E-L-call).

$f'' \in \overline{f}_c$ and $f \notin \overline{f}_c$ This is a reduction going from context to compiled code.

We proceed by induction on ω .

The base case is trivial by Rule E-T-init, the inductive case has two cases:

call

ret

Both are trivially true since nothing extra needs to be enforced. □



PROOF OF LEMMA Q.12 (CONTEXT SPECULATION LASTS AT MOST OMEGA).

By induction on the reduction and with Lemma Q.13 (Single Context Speculation is Safe). □



PROOF OF LEMMA Q.13 (SINGLE CONTEXT SPECULATION IS SAFE).

By induction on s :

Base **skip**
assignment
lfence
Inductive call
sequence
letin
if
read
cmove

All cases are trivial, all expressions evaluate to S due to Lemma Q.9 (Expression Reductions with Safe Bindings are Safe) and no reduction can load U values, so the conditions are met.

All actions are tagged S so they can be related to \emptyset . □

♠

PROOF OF LEMMA Q.14 (COMPILED SPECULATION LASTS AT MOST OMEGA).

By induction on the reduction with Lemma Q.15 (Compiled Speculation is Safe) □

♠

PROOF OF LEMMA Q.15 (COMPILED SPECULATION IS SAFE).

By induction on s :

Base **skip** Trivial.

assign This is analogous to the call case, save that there are two case analyses for both expressions.

private assign This is analogous to the call case, save that there are two case analyses for both expressions.

Inductive call If $\omega = 0$ then this trivially holds by Rule E-T-init.

If $\omega = n + 1$ then we have two cases:

- f is compiled code.

We have two cases:

- (1) $B \triangleright \llbracket e \rrbracket^{SS} \downarrow v : \sigma$

We have two cases here:

- (a) $\omega > 3$

By HP we have that $H(-1) \mapsto \text{true} : S$ so we have:

(for simplicity we only keep track of the top of the stack of execution states)

$$\begin{aligned}
 & (C, H, \overline{B} \cdot B \triangleright \text{let } x_f = \llbracket e \rrbracket^{SS} \text{ in let } x_f = 0 \text{ (if pr) in call } f \ x_f), \omega, U \\
 & \text{assuming } B \triangleright \llbracket e \rrbracket^{SS} \downarrow v : \sigma \\
 & \rightarrow (C, H, \overline{B} \cdot B \cdot x_f \mapsto v : \sigma \triangleright \text{let } x_f = 0 \text{ (if pr) in call } f \ x_f), \omega - 1, U \\
 & \text{since } H(-1) \mapsto \text{true} : S \\
 & \rightarrow (C, H, \overline{B} \cdot B \cdot x_f \mapsto 0 : S \triangleright \text{call } f \ x_f), \omega - 2, U \\
 & \text{where } \overline{B}' \text{ is the current stack and the body of } f \text{ is } \llbracket s \rrbracket^{SS} \\
 & \xrightarrow{\text{call } f \ 0!^S} (C, H, \overline{B}' \cdot x \mapsto 0 : S \triangleright \llbracket s \rrbracket^{SS}), \omega - 3, U
 \end{aligned}$$

So this case holds by IH and by Rule Action Relation - epsi alpha since the action is safe.

- (b) $\omega \leq 3$

In this case the execution will run out of steps and the case holds by Rule E-T-speculate-rollback.

- (2) $B \triangleright \llbracket e \rrbracket^{SS} \downarrow e : \sigma$

If $\llbracket e \rrbracket^{SS}$ gets stuck, this holds by Rule E-T-speculate-rollback-stuck.

- f is context.

This is a contradiction.

seq This is analogous to the if case save for the considerations on the expressions.

letin This is analogous to the if case.

if We have the same cases as in the call case, so we take a look at the most interesting one, namely when all reductions go through:
We have these reductions:

$$\begin{aligned}
& C, H, \bar{B} \cdot B \triangleright \\
& \text{let } x_g = \llbracket e \rrbracket^{SS} \text{ in} \\
& \text{let } pr = rd_{pr} - 1 \text{ in} \\
& \text{let } x_g = 0 \text{ (if } pr) \text{ in} \\
& \text{ifz } x_g \text{ then let } x = rd_{pr} - 1 \text{ in } -1 :=_{pr} x \vee \neg x_g; \llbracket S \rrbracket^{SS} \\
& \quad \text{else let } x = rd_{pr} - 1 \text{ in } -1 :=_{pr} x \vee x_g; \llbracket S' \rrbracket^{SS} \\
\rightarrow & C, H, \bar{B} \cdot B \cdot x_g \mapsto v : \sigma \triangleright \\
& \text{let } pr = rd_{pr} - 1 \text{ in} \\
& \text{let } x_g = 0 \text{ (if } pr) \text{ in} \\
& \text{ifz } x_g \text{ then let } x = rd_{pr} - 1 \text{ in } -1 :=_{pr} x \vee \neg x_g; \llbracket S \rrbracket^{SS} \\
& \quad \text{else let } x = rd_{pr} - 1 \text{ in } -1 :=_{pr} x \vee x_g; \llbracket S' \rrbracket^{SS} \\
\rightarrow & C, H, \bar{B} \cdot B \cdot x_g \mapsto v : \sigma \cdot pr \mapsto \text{true} : S \triangleright \\
& \text{let } x_g = 0 \text{ (if } pr) \text{ in} \\
& \text{ifz } x_g \text{ then let } x = rd_{pr} - 1 \text{ in } -1 :=_{pr} x \vee \neg x_g; \llbracket S \rrbracket^{SS} \\
& \quad \text{else let } x = rd_{pr} - 1 \text{ in } -1 :=_{pr} x \vee x_g; \llbracket S' \rrbracket^{SS} \\
& \text{this step is key: note that } x_g \text{ becomes } S \\
\frac{\text{if}(0)^S}{\rightarrow} & C, H, \bar{B} \cdot B \cdot x_g \mapsto 0 : S \cdot pr \mapsto \text{true} : S \triangleright \\
& \text{ifz } x_g \text{ then let } x = rd_{pr} - 1 \text{ in } -1 :=_{pr} x \vee \neg x_g; \llbracket S \rrbracket^{SS} \\
& \quad \text{else let } x = rd_{pr} - 1 \text{ in } -1 :=_{pr} x \vee x_g; \llbracket S' \rrbracket^{SS} \\
\frac{\text{read}(-1)^S}{\rightarrow} & C, H, \bar{B} \cdot B \cdot x_g \mapsto 0 : S \cdot pr \mapsto \text{true} : S \triangleright \\
& \text{let } x = rd_{pr} - 1 \text{ in } -1 :=_{pr} x \vee \neg x_g; \llbracket S \rrbracket^{SS} \\
\frac{\text{read}(-1)^S}{\rightarrow} & C, H, \bar{B} \cdot B \cdot x_g \mapsto 0 : S \cdot pr \mapsto \text{true} : S \cdot x \mapsto \text{true} : S \triangleright \\
& -1 :=_{pr} x \vee \neg x_g; \llbracket S \rrbracket^{SS} \\
\frac{\text{write}(-1)^S}{\rightarrow} & C, H \cup -1 \mapsto \text{true} : S, \bar{B} \cdot B \cdot x_g \mapsto 0 : S \cdot pr \mapsto \text{true} : S \cdot x \mapsto \text{true} : S \triangleright \\
& \llbracket S \rrbracket^{SS}
\end{aligned}$$

The rest holds by IH so long as the states are related by $\stackrel{S}{\sim}$, which is trivially true and if the trace is related to \emptyset by \approx .
We have this trace

$$\underline{\underline{\text{read}(-1)^S \cdot \text{if}(0)^S \cdot \text{read}(-1)^S \cdot \text{write}(-1)^S}}$$

and each action is related to \emptyset by Rules Action Relation - epsi alpha and Action Relation - epsi heap, so the whole trace is related to \emptyset .

Thus this case holds.

read This is analogous to the if case.

private read This is analogous to the if case.

□



PROOF OF LEMMA Q.10 (ANY SPECULATION FROM RELATED STATES IS SAFE).

This proof proceeds by induction on the stack of configurations.

Base Empty stack:

By Lemma Q.11 (Speculation Lasts at Most Omega) we have the first reductions and $\varnothing \approx \overline{\lambda^\sigma}$:

$$\begin{aligned} & w(C, H_b, \overline{B}_b \triangleright (s_b)_{\overline{f}_b}, \perp, S) \cdot (C, H, \overline{B} \triangleright (s)_{\overline{f}}, \omega, U) \\ \xrightarrow{\overline{\lambda^\sigma}} & w(C, H_b, \overline{B}_b \triangleright (s_b)_{\overline{f}_b}, \perp, S) \cdot (C, H', \overline{B}' \triangleright (s')_{\overline{f}'}, 0, U) \\ \xrightarrow{r1b} & w(C, H_b, \overline{B}_b \triangleright (s_b)_{\overline{f}_b}, \perp, S) \end{aligned}$$

Given the rollback reduction and Rule Action Relation - rlb this case holds.

Inductive This holds by IH plus the same reasoning as in the base case. □

♣

PROOF OF THEOREM Q.17 (GENERALISED BACKWARD SIMULATION FOR SLH).

We have these kinds of reductions:

- compiled-to-compiled code: this holds by Theorem Q.8 (Backward Simulation for Compiled Statements in SLH);
- backtranslated-to-backtranslated code: this holds by Lemma O.9 (Backward Simulation for Backtranslated Statements);
- compiled-to-backtranslated or backtranslated-to-compiled code: this is analogous to the cases discussed in Appendix O.1.4 (Simulation and Relation for Backtranslated Code) since these do not trigger any speculation. □

♣

PROOF OF THEOREM Q.16 (CORRECTNESS OF THE BACKTRANSLATION FOR SLH).

By Theorem Q.17 (Generalised Backward Simulation for SLH) with Lemma Q.3 (Initial States are Related for SLH). □

♣

PROOF OF THEOREM Q.1 (OUR SLH COMPILER IS RSSC⁺).

Instantiate A with $\langle\langle A \rangle\rangle_c^f$.

This holds by Theorem Q.16 (Correctness of the Backtranslation for SLH). □

♣

PROOF OF THEOREM Q.2 (OUR INTER-PROCEDURAL SLH COMPILER IS RSSC⁺).

Instantiate A with $\langle\langle A \rangle\rangle_c^f$.

This holds by a variation of Theorem Q.16 (Correctness of the Backtranslation for SLH) to account for the different state relation ($\overset{S}{\approx}$) which in turns holds by a variation of both Theorem Q.17 (Generalised Backward Simulation for SLH) and Lemma Q.3 (Initial States are Related for SLH).

The latter is a trivial variation of the same theorem to account for the different trace relation.

The former holds by a variation of three results : Theorem Q.8 (Backward Simulation for Compiled Statements in SLH) and Lemma O.9 (Backward Simulation for Backtranslated Statements) and Appendix O.1.4 (Simulation and Relation for Backtranslated Code) with a variation to account for the different state relation.

Of these three, only the first is effectively affected by the change of state relation, so the former holds by a variation of Lemma Q.7 (Forward Simulation for Compiled Statements in SLH), which relies on Lemma Q.10 (Any Speculation from Related States is Safe) and then on Lemma Q.11 (Speculation Lasts at Most Omega) and then on an adaptation of Lemma Q.15 (Compiled Speculation is Safe), where the new trace relation plays a role.

We provide only the proof of the last theorem (in Lemma Q.18 (Compiled Speculation is Safe Inter-procedurally)) since it is the only one with any change. □

♣

PROOF OF LEMMA Q.18 (COMPILED SPECULATION IS SAFE INTER-PROCEDURALLY).

By induction on s :

Base skip Trivial.

assign This is analogous to the call case, save that there are two case analyses for both expressions.

private assign This is analogous to the call case, save that there are two case analyses for both expressions.

Inductive call If $\omega = 0$ then this trivially holds by Rule E-T-init.

If $\omega = n + 1$ then we have two cases:

- f is compiled code.

We have two cases:

- (1) $B \triangleright \llbracket e \rrbracket^{ss} \downarrow v : \sigma$

We have two cases here:

- (a) $\omega > 1$

Compiled code starts with an **lfence** so the execution is immediately rolled back and this case holds by Rule E-T-speculate-rollback.

- (b) $\omega \leq 1$

In this case the execution will run out of steps and the case holds by Rule E-T-speculate-rollback.

- (2) $B \triangleright \llbracket e \rrbracket^{ss} \downarrow e : \sigma$

If $\llbracket e \rrbracket^{ss}$ gets stuck, this holds by Rule E-T-speculate-rollback-stuck.

- f is context.

This is a contradiction.

seq This is analogous to the if case save for the considerations on the expressions.

letin This is analogous to the if case.

if This is analogous to Proof 34 of Lemma Q.15 (Compiled Speculation is Safe).

read This is analogous to the if case.

private read This is analogous to the if case.

□



PROOF OF THEOREM P.1 (THIS SLH COMPILER IS $RSSC^-$).

This is analogous to the proof of Theorem Q.1 (Our SLH compiler is $RSSC^+$).

The only changes are in the compilation of calls, public reads, private reads, public writes and private writes in two theorems:

The former is the adaptation of Lemma Q.7 (Forward Simulation for Compiled Statements in SLH) from $\llbracket \cdot \rrbracket^{ss}$ to $\llbracket \cdot \rrbracket^s$.

The second is the adaptation of Lemma Q.14 (Compiled Speculation Lasts at Most Omega) from $\llbracket \cdot \rrbracket^{ss}$ to $\llbracket \cdot \rrbracket^s$.

□

