Exorcising Spectres with Secure Compilers

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

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

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

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

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

To better present notions, this paper uses colours in a way that both colourblind and black&white readers can benefit from [46]. For a better experience, please print or view this in colours.

I. Introduction

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

Attackers can exploit microarchitectural leaks caused by speculative execution using Spectre attacks [33, 35, 36, 39, 56]. To mitigate these attacks, major compilers deployed a number of compiler-level countermeasures. For instance, the insertion of `lfence` speculation barriers [29] (implemented in the the Microsoft Visual C++ and the Intel ICC compilers [31, 45]) and speculative load hardening [16] (implemented in Clang) *can* be used to mitigate speculative leaks introduced by branch instructions (i.e., the Spectre v1 attack [35]).

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 by a gift from Intel Corporation.

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. For instance, the Microsoft Visual C++ compiler misplaces speculation barriers, thereby producing insecure programs [26, 34].

In this paper, we present a methodology, inspired by recent advances in secure compilation, to reason about compiler-level countermeasures against speculative execution attacks. Armed with this methodology, we study the security of Spectre countermeasures implemented in major C compilers. We prove that the countermeasures implemented in both the Microsoft and the Clang compilers fail at preventing all speculative leaks. Then we formalise secure variants of the `lfence` and speculative load hardening countermeasures and we provide their first proof of security. Specifically, we make these contributions:

▶ We introduce a novel safety property implying the absence of speculative leaks which we dub *speculative safety* (SS, Section III). We characterize the security guarantees of SS by showing that it strictly *over-approximates* speculative non-interference (SNI), a property that precisely characterizes absence of speculative leaks and thus security against Spectre attacks [26]. Crucially, the way SS is formulated makes proving that a compiler preserves SS as simple as possible.

▶ We define two novel secure compilation criteria: *Robust Speculative Safety Preservation* ($RSSP$) and *Robust Speculative Non-Interference Preservation* ($RSNIP$, Section IV). These criteria ensure that compilers preserve SS and SNI *robustly*: roughly speaking such compilers produce code that is safe even when linked against arbitrary (potentially malicious) code. Satisfying these criteria implies that compilers correctly place countermeasures to produce programs resistant to Spectre attacks. However, $RSSP$ requires preserving a safety property (SS) and it is, therefore, much simpler to prove than $RSNIP$, which requires preserving a hyperproperty [20]. To the best of our knowledge, these are the first criteria that concretely instantiate a recent theory that phrases security of compilers as the preservation of (hyper)properties [3, 4, 50] (here: security against Spectre attacks).

▶ We present a general methodology for studying the security of compiler countermeasures against Spectre attacks; see Figure 1 (which we refine in Section IV). These compiler countermeasures translate programs of a source language into

(hardened) programs in a target language.[1]

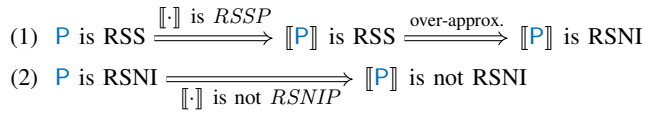

Figure 1. Our methodology to prove prove security (1) and insecurity (2) for compiler countermeasures against Spectre attacks. RSS and RSNI are the robust variants of SS and SNI (defined in Section III).

To show that a compiler countermeasure $[\![\cdot]\!]$ is secure (point (1)), we show that any source program P once compiled ($[\![P]\!]$) satisfies SNI robustly (i.e., RSNI). For this, it suffices to show that $[\![P]\!]$ satisfies SS robustly (i.e., RSS) since, as mentioned before, SS over-approximates SNI. To show that compiled programs $[\![P]\!]$ are RSS it suffices to show that (i) the compiler is *RSSP* and (ii) every source program P is RSS.

To show that a compiler countermeasure $[\![\cdot]\!]$ is insecure (point (2)), we show that it is not *RSNIP*. For this, it suffices to show that (A) some compiled code is not RSNI while (B) its source counterpart is RSNI.

Crucially, our target language has speculation but the source language *does not* (Section II-E). This makes points (ii) and (B) trivial to prove since the lack of speculative execution in the source language implies that any source program is RSS and RSNI. We formalise our source language this way to match the mental model that programmers have. Programmers do not think about speculative execution when writing source code (and they should not!) since speculation only exists in processors (i.e., the target language). It is the duty of a (secure) compiler to ensure these target features cannot be exploited.[2]

▶ With our methodology, we study the security of two existing compiler-level countermeasures against Spectre v1: (1) automated insertion of `lfences`, and (2) speculative load hardening. We show that the way these countermeasures are implemented in the Microsoft Visual C++ and Clang compilers violates *RSNIP* and is thus insecure (Section V).

▶ We also formalise secure compilers implementing countermeasure (1) and a variation of (2), and we prove their security, i.e., they are *RSSP* (Section VI). Ours are the first, provably-secure compilers against Spectre v1 attacks.

Finally, before discussing related work (Section VIII), we discuss how to extend the methodology for reasoning about countermeasures against other Spectre variants (Section VII). These countermeasures follow the same approach as the countermeasures for v1 that we discuss, so the same proof strategy we present for v1 applies for other variants as well.

For clarity, all formalisations are simplified (though we discuss all key aspects); auxiliary lemmas and proofs are omitted. Full details are in the companion technical report [8].

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

[2] Effectively, secure countermeasures can be seen as enforcing the absence of speculative leaks, i.e., they block all speculative leaks.

II. Modelling Speculative Execution

To illustrate our speculative execution model, we first introduce the classical Spectre v1 snippet (Section II-A). Using that, we define the threat model that we consider (Section II-B). Then, we present the syntax (Section II-C) and the common semantics of our languages (Section II-E). We introduce the trace model that we use to talk about observations (Section II-D), followed by the trace semantics of the source (without speculative execution, Section II-F) and of the target (with speculative execution, Section II-G). We conclude by showing how the semantics work (Example 1).

A. Spectre v1: Illustrative example

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

Listing 1. The classic Spectre v1 snippet.

To illustrate speculative execution attacks, we consider the standard Spectre v1 [35] example in Listing 1. Function `get` checks whether the index stored in variable `y` is less than the size of array `A`, stored in the global variable `size`. If so, the program retrieves `A[y]`, multiplies it by the cache line size (here: 512), and uses the result to access array `B`.

If `size` is not cached, traditional processors wait until `size` is fetched from main memory before evaluating the conditional. Rather than waiting, modern processors predict the outcome of the condition by using a branch predictor and speculatively continue the execution. Thus, the memory accesses in the **`then`** branch might be executed even if `y` $\geq$ `size`.

When `size` becomes available, the processor checks whether the prediction was correct. If not, it rolls back all changes to the architectural state and executes the correct branch. However, the speculatively-executed memory accesses leave a footprint in the microarchitectural state, particularly in the cache. This enables an adversary to retrieve `A[y]`, even for `y` $\geq$ `size`, by probing array `B` in the cache.

B. Threat Model

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

For this, we consider an *active* attacker linked against a (compiled) partial program of interest, so the attacker operates at the target level and not at the source. The partial program (or, *component*) stores sensitive information in a private heap that is not directly accessible to the attacker. For example, if the component to secure is the snippet of Listing 1, the array `A` is sensitive data and the attacker is code that runs before and after function `get`. We assume the attacker executes in another process otherwise it would be able to just execute `B[A[y]]` and access the sensitive data. The attacker can poison the branch predictor and observe microarchitectural side-effects of speculatively-executed statements in the component to leak such sensitive information. With secure countermeasures, a compiled component prevents speculative leaks, while insecure countermeasures still allow them.

We consider active attackers because they are more powerful and they let us consider more realistic compilers. Active attackers can mount confused deputy attacks [27, 52] in order to trigger a speculative leak and this makes them stronger than passive attackers considered in previous work [17, 26]. Moreover, to defend against active attackers we must consider realistic compilers that do not just compile whole programs, but *components* that are subsequently co-linked (and interact) with arbitrary code (i.e., the attacker).

Our attacker can observe microarchitectural side-effects on data and instruction caches produced by executing the component of interest. Following [26], we capture these observations in our semantics through traces that record (1) the address of all memory accesses (e.g., the addresses of `A[y]` and `B[A[y]*512]` in the example of Listing 1) and (2) the outcome of all branch and jump instructions. This over-approximates what an attacker could learn by observing accesses to both data (1) and instruction (2) caches. We focus on caches since they are the main side-channel compiler-level countermeasures are concerned about but we remark that our observation-model may also cover other side-channels, e.g., (2) over-approximates what an attacker can learn from the branch predictor state.

Finally, our target language has a speculative semantics that always mispredicts the outcome of all branching instructions executed by the component. This models the worst-case scenario in terms of leakage towards the attacker [26].

C. Syntax of our Languages L and T

The source (L) and target (T) languages for our compilers are very similar: they are both single-threaded while languages with a heap, a stack to lookup local variables, and a notion of components (our unit of compilation). Our formal language is high-level since we can still express speculation and cache side-channels without getting bogged down in the complications of assembly languages (e.g., unstructured control flow).

The common syntax of L and T is presented below; we indicate sequences of elements $e_1, \cdots, e_n$ as $\overline{e}$.

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

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

Since the semantics performs a simple taint-tracking over speculatively-accessed data (explained in Section II-E), heaps map memory addresses $n \in \mathbb{Z}$ to tainted values $v : \sigma$. 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.

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

D. Observations and Trace Model

To record relevant observations produced during the steps of the computation our semantics is *labelled* with observations. Before introducing the semantics, we formalise these observations. They have two roles: they record the control-flow between attacker and code (as required for secure compilation proofs [2, 4, 47, 50]) and they capture what the attacker can observe (as mentioned in Section II-B). This partitions observations into: *Actions* and *Cache Actions*.

$$
\begin{aligned}
&\textit{Actions } \alpha ::= \texttt{call}\ f\ v? \mid \texttt{call}\ f\ v! \mid \texttt{ret}! \mid \texttt{ret}? \\
&\textit{Cache Acts. } \delta ::= \texttt{read}(n) \mid \texttt{write}(n) \mid \texttt{if}(v) \mid \texttt{rlb}
\end{aligned}
$$

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

The `read`(n) and `write`(n) actions denote respectively read and write accesses to the heap location n. These let us model leaks through data caches without requiring to explicitly handle the cache in our semantics.

The `if`(v) action denotes that the outcome of branch instructions. This implicitly exposes the value of the program counter (i.e., which instruction we are currently executing), and thus the instruction cache content. The `rlb` action indicates the end of some speculation that has been rolled back.

Observations that are recorded via the semantics are also called labels; they are tainted (with a taint σ) and then concatenated into tainted traces $\overline{\lambda^\sigma}$. Traces have this normal form: $\alpha?^\sigma \overline{\delta^\sigma} \alpha!^\sigma$, where α^σs are tainted calls/returns and δ^σs are tainted cache actions. The alternation of ? and ! actions is due to our programs having well-bracketed control flow.

$$
\textit{Labels } \lambda ::= \epsilon \mid \alpha \mid \delta \qquad \textit{Traces } \overline{\lambda^\sigma} ::= \varnothing \mid \overline{\lambda^\sigma} \cdot \alpha^\sigma \mid \overline{\lambda^\sigma} \cdot \delta^\sigma
$$

Note that according to the normal form, cache actions δ done by the attacker are not recorded (Rule E-L-single later on). This is intuitive: the attacker has no direct access to the private heap so his cache actions cannot possibly indicate leaks. This approach is common in robust safety works [23, 25, 38, 59].

E. Non-Speculative Semantics for L and T and Taint Tracking

Both languages are given a labelled operational semantics that describes how whole programs execute. Given a compo-

nent P and an attacker A, we can link them ($A[P]$) and obtain a whole program W that contains all functions and heaps of both 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 and only the functions of $\overline{I}$.

Program States: To define the operational semantics, we need a notion of program state (Ω), which relies on the notion of a stack of local variables ($\overline{B}$). Notation-wise $\overline{e} \cdot e$ denotes a stack with top element e and rest of the stack $\overline{e}$.

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

A program state $C, H, \overline{B} \triangleright (s)_{\overline{f}}$ keeps track of a component C, a heap H, a stack of local variables $\overline{B}$, a statement s, and a stack of function names $\overline{f}$. C is used to look up function bodies and to determine which functions are the component's and which are the attacker's. The list of function names $\overline{f}$ is used to infer what code is executing: attacker or component; this information is used to determine which labels to produce. We often elide this list for the sake of clarity.

Taint Tracking: Our semantics implements a simple taint-tracking to taint observations that are executed speculatively on sensitive data, i.e., on data coming from the private heap. A value in our semantics can be tainted S (i.e., data not coming from the private heap) or U (i.e., data coming from the private heap). Taint of values respectively indicates whether it is safe or unsafe for the attacker to access that value. Tainted values are propagated as standard as programs execute.

As mentioned, taint is propagated also to observations and traces. For instance, reading a heap location can produce a safe (S) or an unsafe observation (U) depending on the taint of the read data and of the pc (which captures whether we are speculating or not). The rules below only generate taint according to the taint of data; the rules of the trace semantics (later on) update the taint of actions based on that of the pc.

Concretely, we work with the usual integrity lattice $S \leq U$ (which is the dual of the usual non-interference lattice [13, 53]). We use the least-upper-bound (lub, $\sqcup$) to propagate taint of data, while we use the greatest lower bound (glb, $\sqcap$) to generate the taint of actions based on the program counter taint. For simplicity, we report the key cases of the truth tables of these operations: $S \sqcup U = U$ and $S \sqcap U = S$.

Operational Semantics: Both languages have a big-step operational semantics for expressions (left) and a small-step, structural operational semantics for statements that generates tainted labels (right): $B \triangleright e \downarrow v : \sigma$ $\quad$ $\Omega \xrightarrow{\lambda^{\sigma}} \Omega'$

The left is read: according to the stack of local variables B, expression e reduces to value v with taint σ. These rules are standard and thus omitted (see Appendix N). We remark that values are computed as expected (though we use *0* for true in *if-zero* statements) , an expression e's taint is the lub of its variables' taint, and expressions can access only local variables in B (reading from the heap is treated as a statement).

The right is read: state Ω reduces in one step to Ω' emitting label λ tainted as σ. The most illustrative rules are below.

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

$$\frac{B \triangleright e \downarrow n : \sigma' \quad H = H_1; -|n| \mapsto v : \sigma; H_2 \quad \sigma'' = \sigma \sqcup \sigma'}{C, H, \overline{B} \cdot B \triangleright \textit{let}\ x = rd_p\ e\ \textit{in}\ s \xrightarrow{\texttt{read}(-n)^{\sigma''}} C, H, \overline{B} \cdot B \cup x \mapsto v : U \triangleright s}\ \text{(E-read-prv)}$$

$$\frac{\begin{array}{c} B \triangleright e \downarrow n : \sigma \quad H = H_1; -|n| \mapsto v' : \sigma'; H_2 \\ B \triangleright e' \downarrow v : \sigma \quad H' = H_1; -|n| \mapsto v : \sigma; H_2 \end{array}}{C, H, \overline{B} \cdot B \triangleright e :=_p e' \xrightarrow{\texttt{write}(-|n|)^{\sigma}} C, H', \overline{B} \cdot B \triangleright \textit{skip}}\ \text{(E-write-prv)}$$

Regarding taint propagation, the most interesting rule is Rule E-L-read-prv. Reading from the private heap taints the variable where the content is stored as unsafe (U) but it still produces an action tainted according to the data taint (σ'').

F. Non-Speculative Traces for L

The source language L has a big-step semantics to concatenate single steps into multiple ones and single labels in traces:

$$\Omega \xRightarrow{\overline{\lambda^{\sigma}}} \Omega'$$

This judgement is read: state Ω emits trace $\overline{\lambda^{\sigma}}$ and becomes Ω'. Since we want our source of unsafe behaviour to be speculation, the pc should be S when not speculating. Effectively, this means that the pc for any L program is always S and so traces are always tagged as S (the 'else' branch in Rule E-L-single). Additionally, we do not show any heap-related action that is performed by the attacker (the 'then' branch in Rule E-L-single, recall that functions in $\overline{\mathsf{I}}$ are defined by the attacker) as mentioned in Section II-D.

$$\frac{\begin{array}{c} \Omega \xRightarrow{\overline{\lambda_1^{\sigma}}} \Omega'' \qquad \Omega'' \xrightarrow{\alpha^{\sigma}} \Omega' \\ \Omega'' = \overline{\mathsf{F}}, \overline{\mathsf{I}}, \mathsf{H}, \mathsf{B} \triangleright (\mathsf{s})_{\overline{\mathsf{f}}.\mathsf{f}} \qquad \Omega' = \overline{\mathsf{F}}, \overline{\mathsf{I}}, \mathsf{H}', \mathsf{B}' \triangleright (\mathsf{s}')_{\overline{\mathsf{f}'}.\mathsf{f}'} \\ \text{if } \mathsf{f} == \mathsf{f}' \text{ and } \mathsf{f} \in \mathsf{I} \text{ then } \overline{\lambda_2^{\sigma}} = \overline{\lambda_1^{\sigma}} \text{ else } \overline{\lambda_2^{\sigma}} = \overline{\lambda_1^{\sigma}} \cdot \alpha^{\mathsf{S}} \end{array}}{\Omega \xRightarrow{\overline{\lambda_2^{\sigma}}} \Omega'}\ \text{(E-L-single)}$$

By relying on this semantics (for which we presented just a selection of rules), we can define the behaviour of a whole program as the trace that can be generated starting from the initial state ($\Omega_0(\cdot)$) of a program until it terminates.[3] Intuitively, the initial state of a program is the main function (which is defined by the attacker).

$$\frac{\exists \Omega.\ \Omega \text{ is stuck and } \Omega_0(\mathsf{W}) \xRightarrow{\overline{\lambda^{\sigma}}} \Omega}{\mathsf{W} \rightsquigarrow \overline{\lambda^{\sigma}}}\ \text{(E-L-trace)} \qquad \frac{\mathsf{W} \rightsquigarrow \overline{\lambda^{\sigma}}}{\mathsf{Beh}(\mathsf{W}) = \overline{\lambda^{\sigma}}}\ \text{(E-L-behaviour)}$$

G. Speculative Semantics for T

Our speculative semantics for T is inspired by the so-called "always mispredict" semantics of Guarnieri *et al.* [26]. This semantics captures the worst-case scenario, from an information

[3] Existing work defines a program's behaviour as a set of traces for non-deterministic languages [3, 4]. Since our language is fully deterministic, the behaviour is a singleton trace [37].

theoretic perspective, independently on how the attacker could potentially affect the branch prediction strategy. Whenever the semantics executes a branch instruction, it first mis-speculates by executing the wrong branch for a fixed number $\mathbf{w}$ of steps (called *speculation window*). After speculating for $\mathbf{w}$ steps, the speculative execution is terminated, the changes to the program state are rolled back, and the semantics restarts by executing the correct branch. The microarchitectural effects of the speculatively executed instructions are recorded on the trace as actions. In order to generate the correct label for the semantics, the taint of the program counter starts as $\mathbf{S}$ and it is raised to $\mathbf{U}$ when speculation happens.

For our speculative semantics, we first define a notion of speculative program state ($\mathbf{\Sigma}$) as a stack of speculation instances ($\mathbf{\Phi}$). A speculation instance records the operational state $\mathbf{\Omega}$, its remaining speculation window $\mathbf{w}$ (a natural number $\mathbf{n}$ or $\bot$ when no speculation is happening) and the taint of its program counter $\boldsymbol{\sigma}$. The maximum length of the speculation window is a global constant $\boldsymbol{\omega}$. This value depends on physical characteristics of the CPU, e.g., the size of the reorder buffer.

$$\textit{Speculation States } \mathbf{\Sigma} ::= \overline{\mathbf{\Phi}}$$
$$\textit{Speculation Instance } \mathbf{\Phi} ::= (\mathbf{\Omega}, \mathbf{w}, \boldsymbol{\sigma})$$

The execution starts in state $\mathbf{\Sigma_0} = (\mathbf{\Omega_0}, \bot, \mathbf{S})$, i.e., in the same initial state that L starts with ($\mathbf{\Omega_0}$), with the program counter tainted as $\mathbf{S}$ since no speculation has happened yet.

Given a speculation state $\mathbf{\Sigma}$, reductions happen on the state on the top of the stack (Rule E-$\mathbf{T}$-speculate-epsilon, Rule E-$\mathbf{T}$-speculate-action). Mis-speculation pushes the mis-speculating state on top of the stack and taints the program counter as $\mathbf{U}$ (Rule E-$\mathbf{T}$-speculate-if). When the speculation window is exhausted (or if the speculation reached a stuck state), speculation ends and the top of the stack is popped (Rule E-$\mathbf{T}$-speculate-rollback). The role of the $\mathbf{lfence}$ instruction is setting to zero the speculation window, so that rollbacks are triggered (Rule E-$\mathbf{T}$-speculate-lfence).

(E-$\mathbf{T}$-speculate-epsilon)
$$\frac{\mathbf{\Omega} \xrightarrow{\epsilon} \mathbf{\Omega}' \quad \mathbf{\Omega} \equiv \mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \triangleright \mathbf{s}; \mathbf{s}' \quad \mathbf{s} \not\equiv \mathbf{ifz} \cdots \text{ nor } \mathbf{lfence}}{\overline{\mathbf{\Phi}} \cdot (\mathbf{\Omega}, \mathbf{n}+1, \boldsymbol{\sigma}) \overset{\epsilon}{\rightsquigarrow} \overline{\mathbf{\Phi}} \cdot (\mathbf{\Omega}', \mathbf{n}, \boldsymbol{\sigma})}$$

(E-$\mathbf{T}$-speculate-action)
$$\frac{\mathbf{\Omega} \xrightarrow{\lambda^{\sigma}} \mathbf{\Omega}' \quad \mathbf{\Omega} \equiv \mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \triangleright \mathbf{s}; \mathbf{s}' \quad \mathbf{s} \not\equiv \mathbf{ifz} \cdots \text{ nor } \mathbf{lfence}}{\overline{\mathbf{\Phi}} \cdot (\mathbf{\Omega}, \mathbf{n}+1, \boldsymbol{\sigma}') \overset{\lambda^{\sigma \sqcap \sigma'}}{\rightsquigarrow} \overline{\mathbf{\Phi}} \cdot (\mathbf{\Omega}', \mathbf{n}, \boldsymbol{\sigma}')}$$

(E-$\mathbf{T}$-speculate-if)
$$\frac{\begin{array}{c}\mathbf{\Omega} \xrightarrow{\lambda^{\sigma}} \mathbf{\Omega}' \qquad \mathbf{\Omega} \equiv \mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \triangleright \mathbf{s}; \mathbf{s}' \\ \mathbf{s} \equiv \mathbf{ifz\ e\ then\ s''\ else\ s'''} \qquad \mathbf{j} = \mathtt{min}\,(\boldsymbol{\omega}, \mathbf{n}-1) \\ \text{if } \mathbf{B} \triangleright \mathbf{e} \downarrow \mathbf{0} \text{ then } \mathbf{\Omega''} \equiv \mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \triangleright \mathbf{s'''}; \mathbf{s}' \\ \text{if } \mathbf{B} \triangleright \mathbf{e} \downarrow \mathbf{n} \text{ and } \mathbf{n} > \mathbf{0} \text{ then } \mathbf{\Omega''} \equiv \mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \triangleright \mathbf{s''}; \mathbf{s}'\end{array}}{\overline{\mathbf{\Phi}} \cdot (\mathbf{\Omega}, \mathbf{n}, \boldsymbol{\sigma}') \overset{\lambda^{\sigma \sqcap \sigma'}}{\rightsquigarrow} \overline{\mathbf{\Phi}} \cdot (\mathbf{\Omega}', \mathbf{n}-1, \boldsymbol{\sigma}') \cdot (\mathbf{\Omega''}, \mathbf{j}, \mathbf{U})}$$

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

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

Thus, the target language $\mathbf{T}$ has a small-step speculative semantics (left) that generates single labels as well as a big-step semantics that concatenates single labels in traces (right):

$$\mathbf{\Sigma} \overset{\lambda^{\sigma}}{\rightsquigarrow} \mathbf{\Sigma}' \qquad\qquad \mathbf{\Sigma} \overset{\overline{\lambda^{\sigma}}}{\Longrightarrow} \mathbf{\Sigma}'$$

As before, actions that the attacker generates within itself are not showed (Rule E-L-single adapted to $\mathbf{T}$). Rules for traces are analogous to those of L (and therefore omitted), except that they rely on single steps made by the speculative semantics ($\overset{\cdot}{\rightsquigarrow}$) instead of operational steps ($\xrightarrow{}$) and that they propagate the taint of actions instead of setting it to S.

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

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

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

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

III. Security Conditions for Secure Speculation

In this section, we present the robust variants of SNI (Section III-A) and SS (Section III-B). We conclude by stating the relationship between the two (Section III-C).

A. Robust Speculative Non-Interference

This section instantiates Speculative Non-Inteference (SNI) from [26] in our framework, and it defines its *robust* version (RSNI), which can be used to reason about active attackers. Technically, we derive our formalization of SNI (Definition 13) from the simpler variant of Guarnieri *et al.* that is equivalent to the more general formulation [26, Proposition 1].

A program satisfies SNI [26] whenever speculatively-executed instructions do not leak more information than non-speculatively-executed instructions. Before formalizing SNI, we introduce two concepts:

- SNI is parametric in a policy that determines which information is sensitive. As mentioned in Section II-B, we assume that the private heap is sensitive while the attacker has direct access to everything else. We model this by defining a notion of low-equivalence between whole programs W and W'. We say that W and W' are *low-equivalent*, written $W' =_{\mathsf{L}} W$, if they differ only in their private heaps.
- SNI requires comparing the leakage resulting from non-speculative and speculative instructions. The *non-speculative projection* $t \restriction_{nse}$ [26] of a trace t extracts the observations

associated only with non-speculatively executed instructions. We obtain $t\restriction_{nse}$ by removing from t sub-strings enclosed between $\texttt{if}(v)$ and $\texttt{rlb}$ observations. We do not present this simple definition but rather show how it operates on traces concretely, below is $\cdot\restriction_{nse}$ applied to $\mathbf{t_{sp}}$ from Example 1.

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

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

A component P is robustly speculatively non-interferent if it is SNI no matter what valid attacker it is linked to (Definition 14). A attacker is valid (indicated as $\vdash A : atk$) if it does not define a private heap and if it does not contain instructions to read and write the private heap.

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

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

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

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

Example 2 (Listing 1 is not RSNI in T)**.** Indicate the code of Listing 1 as $\mathbf{P_1}$, consider attacker $\mathbf{A^8}$ that calls function get with 8. Since array A is in the private heap, the low-equivalent program required by Definition 13 is the same $\mathbf{A^8}$ linked with some $\mathbf{P_N}$, which is the same $\mathbf{P_1}$ with some array N with contents different from A in the heap such that A[8]≠N[8]. Whole program $\mathbf{A^8}[\mathbf{P_1}]$ generates trace $\mathbf{t_{sp}}$ from Example 1 while $\mathbf{A^8}[\mathbf{P_N}]$ generates $\mathbf{t'_{sp}}$ below. We indicate the address of array N as $\mathbf{n_N}$ and the content of N[i] as $\mathbf{v^i_N}$. Low-equivalence yields that addresses are the same ($\mathbf{n_A} + \mathbf{8} = \mathbf{n_N} + \mathbf{8}$) but contents are not ($\mathbf{v^8_A} \neq \mathbf{v^8_N}$), and thus B is accessed at different offsets ($\mathbf{n_B} + \mathbf{v^8_A} \neq \mathbf{n_B} + \mathbf{v^8_N}$).

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

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

B. Robust Speculative Safety

In our model, speculative leaks result from microarchitectural side effects (i.e., observations in our model) happening during speculative execution (when the program counter taint is U) depending on data from (or influenced by) the private heap (that is, data tainted as U). Thus, speculative leaks result in actions that are tainted U (which is the meet of the data and program counter taints). Informally, a component is RSS whenever it produces only S actions, thereby ensuring the absence of speculative leaks.

Formally, *whole* program W enjoys speculative safety (SS) if it only generates safe (S) actions in its traces (Definition 3). A component P is RSS if it upholds SS when linked against arbitrary valid attackers (Definition 4).

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

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

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

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

The snippet of Listing 1 is not RSS in T because according to the semantics, the attacker that calls get with argument 8 generates trace $\mathbf{t_{sp}}$, which has an unsafe action (Example 1). The same code in L is RSS because it never generates actions tainted as U. More generally, since L traces only have S actions (Rule E-L-single), all L programs are RSS and RSNI.

Theorem 1 (All L programs are RSS and RSNI)**.**

$$\forall \text{P} \in \text{L}.\ \vdash \text{P} : \text{RSS}\ \text{and}\ \vdash \text{P} : \text{RSNI}$$

C. The Relationship between RSNI and RSS

This section concludes by connecting RSNI and RSS. RSNI provides a precise semantics characterization of the absence of speculative leaks [26]. In contrast, RSS is an over-approximation of RSNI whose preservation through compilation is easier to prove than the RSNI-preservation. Theorem 2 states that RSS implies RSNI but the converse does not hold.

Theorem 2 (RSS is a overapproximation of RSNI)**.**

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

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

To understand point 2, consider function get_nc from Listing 2, which always accesses B[A[y]]. This code is RSNI because any two configurations that can be distinguished by looking at the traces would also be distinguished by looking at their non-speculative projections, i.e., speculatively-executed instructions do not leak additional information. However, it is not RSS because any memory access done when speculating is tagged U and thus when speculating, this snippet will also generate trace $\mathbf{t_{sp}}$ from Example 1.

```
1 void get_nc (int y) {
2   if (y < size) then B[A[y] *512] else B[A[y] *512]
3 }
```

Listing 2. Code that is RSNI but not RSS.

IV. Compiler Criteria for Spectre Security

This section defines compiler criteria that preserve RSS and RSNI following a common approach to these definitions [3,

4, 50]. We first provide a criterion that obviously implies the preservation of RSS (resp. RSNI) through compilation, which we dub *RSSP*, for *robust speculative safety preservation* (resp. *RSNIP* for robust speculative non-interference preservation). For *RSSP*, we also provide a second, *equivalent* criterion that is more amenable to proofs, which we dub *RSSC*, for *robustly- speculatively-safe compilation*. It is important to provide both kinds of criteria because the former provide a clear understanding of their security implications to the readers, while the latter are easier to prove via established proof techniques. We do not provide an equivalent criterion for *RSNIP* since we prove when countermeasures *do not* attain *RSNIP* and this kind of proof is simple already.

To state these criteria and their equivalence, we introduce a cross-language relation between traces of the two languages, which specifies when two possibly different traces have the same "meaning". Finally, we present our methodology to show how compilers can be proven secure or insecure against speculation attacks by refining Figure 1 (Figure 2).

The first criteria are clear: a compiler preserves RSS (resp. RSNI) if, given a source component that is RSS (resp. RSNI), the compiled counterpart is also RSS (resp. RSNI). Since we use *RSNIP* in its *negation* form, we report it for clarity.

Definition 5 (*RSSP*)**.**

$$\vdash [\![\cdot]\!] : \mathit{RSSP} \stackrel{\text{def}}{=} \forall \mathtt{P}.\ \text{if} \vdash \mathtt{P} : \text{RSS then} \vdash [\![\mathtt{P}]\!] : \text{RSS}$$

Definition 6 (*RSNIP*)**.**

$$\vdash [\![\cdot]\!] : \mathit{RSNIP} \stackrel{\text{def}}{=} \forall \mathtt{P}.\ \text{if} \vdash \mathtt{P} : \text{RSNI then} \vdash [\![\mathtt{P}]\!] : \text{RSNI}$$

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

$$\nvdash [\![\cdot]\!] : \mathit{RSNIP} \stackrel{\text{def}}{=} \exists \mathtt{P}.\ \vdash \mathtt{P} : \text{RSNI and} \nvdash [\![\mathtt{P}]\!] : \text{RSNI}$$

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

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

Note that finding the existentially-quantified program (and attacker) that demonstrate insecurity of a countermeasure may be hard. Fortunately, some failed attempts at proving *RSSC* can provide hints for how to do this; we provide more insights after discussing proof techniques in Section VI-C1. □

Definitions 5 and 19 are "property-ful" criteria since they explicitly refer to the property that the compiler preserves [3, 4]. Proving "property-ful" criteria can be fairly complex at times, but fortunately, it is generally possible to turn a "property-ful" definition into an *equivalent* "property-free" one [3, 4, 50]. This is often beneficial because "property-free" criteria come in so-called *backtranslation* form, which have established proof techniques [2, 4, 14, 43, 47, 50].

Our property-free security condition (*RSSC*, Definition 18 below) states that a compiler is *RSSC* if for any target-level attacker $\mathtt{A}$ that generates a trace $\overline{\lambda^\sigma}$, it is possible to build a source-level attacker $\mathtt{A}$ that generates a trace $\overline{\lambda^\sigma}$ that is related to $\overline{\lambda^\sigma}$. A source trace $\overline{\lambda^\sigma}$ and a target trace $\overline{\lambda^\sigma}$ are related (denoted with $\overline{\lambda^\sigma} \approx \overline{\lambda^\sigma}$) if the target trace contains all the actions of the source trace, plus possible interleavings of safe ($\mathbf{S}$) actions (Rules Trace-Relation-Safe and Trace-Relation-Safe-Heap). All other actions must be the same (i.e., $\equiv$, Rules Trace-Relation-Same-Act and Trace-Relation-Same-Heap).

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

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

We are now ready to formalise *RSSC*, which is equivalent to *RSSP* (Theorem 3). Importantly, this result implies that our choice for the trace relation is correct; a relation that is too strong or too weak would not let us prove this equivalence.

Definition 7 (*RSSC*)**.**

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

Theorem 3 (*RSSP* and *RSSC* are equivalent)**.**

$$\forall [\![\cdot]\!].\ \vdash [\![\cdot]\!] : \mathit{RSSP} \iff \vdash [\![\cdot]\!] : \mathit{RSSC}$$

Definition 7 requires providing an existentially-quantified source attacker $\mathtt{A}$. The general proof technique for these criteria is called *backtranslation* [4, 48], and it can either be attacker- [14, 21, 43] or trace-based [2, 47, 50]. The distinction tells us what quantified element we can use use to build the source attacker $\mathtt{A}$, either the target attacker $\mathtt{A}$ or the finite trace $\overline{\lambda}$ respectively. Fortunately, our setup is powerful enough that both techniques apply, and, for simplicity, we will adopt the attacker-based backtranslation.

Next, we present the methodology that uses our criteria to prove security and insecurity of compiler countermeasures. Recall from Section III-A that RSNI in the target language is the property we should have for components that are compiled with secure countermeasures. Conversely, components that are compiled with insecure countermeasures *are not* RSNI in the target. These intuitive ideas are represented as two chains of implications in Figure 2. The first one lists the assumptions (black dashed lines) and logical steps (theorem-annotated implications) to conclude compiler security while the second one lists assumptions and logical steps for compiler insecurity.

To show security (1), we need to prove that any compiled component is RSNI in the target language. By Theorem 2, it suffices to show that any compiled component is RSS in the target. This can be obtained by an *RSSP* compiler (i) so long as any $\mathtt{P}$ is RSS in the source (ii). By Theorem 3, for point (i) it is sufficient to show that the compiler is *RSSC*. Point (ii) holds for any $\mathtt{P}$ since they cannot speculate (Theorem 8).

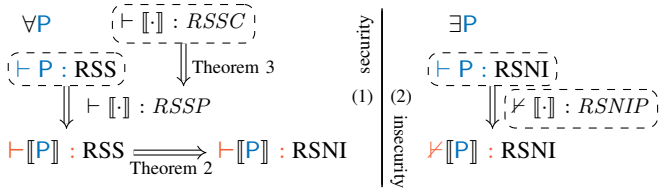

Figure 2. Our methodology to prove security (1) and insecurity (2) for compiler countermeasures against Spectre.

To show insecurity (2), we need to prove that there exists a compiled component that is *not* RSNI in the target language. For this, we need to show that the compiler is not *RSNIP* (A) given that the source component P was RSNI in the source (B). Point (A) follows from showing what mentioned in Corollary 4. As before, point (B) holds for any source component P since they cannot speculate (Theorem 8).

Thus, it suffices to show that $[\![\cdot]\!]$ is *RSSC* to know that it produces code resistant to Spectre attacks. Also, it suffices to show that $[\![\cdot]\!]$ is not *RSNIP* to know that it produces code vulnerable to Spectre attacks; this is what Section V presents.

V. Countermeasures Overview and Insecurities

In this section, we present an overview of the two main compiler-level countermeasures against Spectre v1: insertion of speculation barriers (Section V-A) and speculative load hardening (Section V-B). For each of these countermeasures, we illustrate with concrete examples that major compilers implement these countermeasures in an insecure way, that is, they violate *RSNIP* and produce programs that are *not* RSNI. For simplicity, we report the examples in the assembly language that those compilers emit.[4] All these examples are expressible in our formal languages (see Appendix L).

A. Insertion of Speculation Barriers

A simple compiler-level countermeasure against Spectre v1 attacks [29, 31, 45] is the insertion of speculation barrier—the lfence x86 instruction—after branch instructions. The lfence instruction has the effect of stopping speculative execution at the price of significant performance overhead.

This countermeasure is implemented in the Microsoft Visual C++ [45] (MSVC below) and Intel ICC [31] compilers. The ICC compiler is rather conservative in his placement of lfence instructions, as it inserts an lfence on both branches after *each* branch instruction. This makes ICC secure, as we prove in Section VI-A. In contrast, MSVC tries to reduce the number of lfences by selectively determining which branches to patch. Example 3 below illustrates how MSVC works on the standard Spectre v1 snippet while Example 4 (and as pointed out in [26, 34]) illustrates that MSVC sometimes omits necessary lfences.

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

[4] For this we still follow our colouring convention and typeset them in red when they are target code and in blue when they are source code.

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

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

Example 4 (MSVC is not *RSNIP*)**.** Consider now the code in Listing 4 below, which is adapted from from [34, Example 10]. In contrast to Listing 1, this example speculatively leaks whether `A[y]` is `0` through the branch statement in line 3.

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

Listing 4. A variant of the classic Spectre v1 snippet (Example 10 from [34]).

When compiling this code with the lfence-countermeasure enabled, MSVC produces the snippet shown in Listing 5.

```
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 5. Listing 4 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 4 trivially satisfies RSNI, because the source language L does not allow speculative execution. Its compilation in Listing 5, however, violates RSNI. To show this, consider two low-equivalent initial states Ω^0, Ω^1 where $\mathtt{y}$ is out-of-bound, A[y] is 0 in Ω^0 and 1 in Ω^1, and the value of $\mathtt{y}$ is 42 in both. The corresponding traces are:

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

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

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

B. Speculative Load Hardening

Clang implements a countermeasure called speculative load hardening [16] (SLH) that works as follows:[5]

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

SLH also has a non-interprocedural variant[6] that does not track the predicate bit across function calls but just locally. Intuitively, when a function starts executing the bit is 0 and it is set to 1 only after mis-speculated branches (as before).

Example 5 shows how SLH works. Example 6 illustrates that the SLH implementation of Clang fails in stopping all speculative leaks. Thus, SLH is not *RSNIP*. Unfortunately, its non-interprocedural variant also violates *RSNIP*, as presented in Example 7. For space constraints, we present the details (e.g., target traces) of why SLH and its non-interprocedural variant are not *RSNIP* in Appendix M. Section VI-B describes how to secure these countermeasures to emit RSNI code.

Example 5 (SLH in action)**.** Consider again the Spectre v1 snippet from Listing 1. Clang with SLH enabled compiles the program into the (simplified) assembly in Listing 6.

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

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

Example 6 (SLH is not *RSNIP*)**.** Consider the variant of Spectre v1 illustrated in Listing 7. The main difference with the standard Spectre v1 example (Listing 1) is that the first memory access is performed non-speculatively (line 2). Its value, however, is still leaked through the speculatively-executed memory access in line 4. Clang with SLH compiles this code into the snippet of Listing 8.

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

Listing 7. Another variant of the classic Spectre v1 snippet.

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

Listing 8. Compiled version of Listing 7 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

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

[6] The non-interprocedural variant of SLH, which is disabled by default, can be activated with the `-mllvm -x86-speculative-load-hardening -mllvm -x86-slh-ip=FALSE` compilation flags.

set to 0x0..0 and the value of A[y] is not protected. Therefore, speculatively executing the load in line 11 may still leak the value of A[y] speculatively, which will be different in traces generated from different, low-equivalent states. ⊡

Example 7 (Non-interprocedural SLH is not *RSNIP*)**.** The program of Listing 9 splits the memory accesses of A and B of the classical snippet across functions get and get_2.

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

Listing 9. Inter-procedural variant of the Spectre v1 snippet [41].

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]). ⊡

VI. Secure Countermeasures

We now formalise compiler countermeasures and prove them secure. We first present a high-level model of the Intel C++ compiler that inserts speculation barriers after all branching statements [31] and prove it *RSSC* (Section VI-A). Then, we formalise high-level corrections of both Clang-like SLH and of its interprocedural variant and prove them *RSSC* (Section VI-B). Finally, we discuss the proofs that both compilers are *RSSC* (Section VI-C).

A. Speculation Barriers are Secure

We model the Intel C++ compiler with $[\![\cdot]\!]^f$, a homomorphic compiler that takes a component in L and translates all of its subparts to T. While most of the compiler is straightforward (and thus omitted), its key feature is inserting an lfence at the beginning of every then and else branch of compiled code.

$$[\![\mathsf{f(x)} \mapsto \mathsf{s; return;}]\!]^f = \mathbf{f(x)} \mapsto [\![\mathsf{s}]\!]^f\mathbf{; return;}$$

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

It should come at no surprise that $[\![\cdot]\!]^f$ is *RSSC* (Theorem 18). The only source of speculation are branches (Rule E-T-speculate-if) but any branch, whether it evaluates to true or false, will execute an lfence (Rule E-T-speculate-lfence), triggering a rollback (Rule E-T-speculate-rollback). So, compiled code can perform no action during speculation. It can only perform actions when the pc is tainted as S, which makes all actions S. These actions are easy to relate to their source-level counterparts since they are generated according to the non-speculative semantics.

Theorem 4 (The lfence compiler is *RSSC*)**.** $\vdash [\![\cdot]\!]^f : RSSC$

Stronger Results for $[\![\cdot]\!]^f$*:* When considering $[\![\cdot]\!]^f$, we have two concerns: does it just preserve a safety property (RSS) and can we prove no information leaks on other side channels (we only model caches)? We now answer both.

Concerning the former question, the secure compilation criteria of Abate *et al.* [3, 4] also provide statements for compilers that preserve more, e.g., arbitrary hyperproperties [20]. We believe that $[\![\cdot]\!]^f$ does indeed attain this stronger criterion, though for the purpose of showing absence of speculative leaks, what we prove is sufficient.

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

$$\text{(E-E-speculate-eta)}\quad \frac{\Omega \xrightarrow{\epsilon} \Omega' \qquad \Omega \equiv \mathbf{C, H, \overline{B} \triangleright s; s'} \qquad \mathbf{s} \not\equiv \mathbf{ifz} \cdots \text{ nor } \mathbf{lfence}}{\overline{\Phi} \cdot \Phi \cdot (\Omega, n+1, \sigma) \overset{\eta^{\sigma}}{\rightsquigarrow} \overline{\Phi} \cdot \Phi \cdot (\Omega', n, \sigma)}$$

Denote with $[\![\cdot]\!]^f_{\mathbf{E}}$ the lfence-inserting compiler to E (which is exactly like $[\![\cdot]\!]^f$). In compiled code, *no statement* is executed speculatively, so it is easy to prove that compiled code has no leak, i.e., not even η is emitted. We can thus prove that $[\![\cdot]\!]^f_{\mathbf{E}}$ is *RSSC* with the same trace relation. Since this relation does not mention η, Theorem 5 is only possible if compiled code never emits η. This is not true for the case of SLH below (and for analogous countermeasures), since there some compiled statements are executed speculatively. This is, after all, expected, since SLH focusses on protecting the cache side channel and not, for example, timing side-channels.

Theorem 5 (The lfence compiler is *RSSC* even with more leaks)**.** $\vdash [\![\cdot]\!]^f_{\mathbf{E}} : RSSC$

B. Securing Speculative Load Hardening

The formalisation of the secure variation of SLH (which we dub SSLH) keeps track of the predicate bit in a cross-procedural way and masks inputs, rather than outputs, to statements that can leak, i.e., produce actions. We present and discuss only the most relevant cases below.

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

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

$$[\![\mathsf{e} :=_{\mathsf{p}} \mathsf{e'}]\!]^s = \begin{array}{l} \mathbf{let\ x_f = [\![e]\!]^s + 1\ in} \\ \mathbf{let\ x'_f = [\![e']\!]^s\ in} \\ \mathbf{let\ pr = rd_p\ -1\ in} \\ \mathbf{let\ x_f = 0\ (if\ pr)\ in} \\ \mathbf{let\ x'_f = 0\ (if\ pr)\ in} \\ \mathbf{x_f :=_p x'_f} \end{array}$$

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

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

Since the stack pointer is not accessible from an attacker residing in another process, the predicate bit is tracked in the first location of the private heap. 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 $-1 :=_p \cdots$). Since location -1 is reserved for the predicate bit, all memory accesses as well as the global heap are shifted by 1, so what was stored at -1 in L is stored at -2 in T etc.

Statements can lead to either direct or indirect leaks; both kinds are captured on traces and must have their inputs masked. Calling attacker functions, reading and writing the public heap are all direct leaks since their values are directly observable by the attacker. Reading and writing the private memory as well as branching are all indirect leaks since their values can be read through the caches. When these statements are compiled, we need to mask their input (masking the outputs would lead to the same vulnerability of Example 6). That is, we evaluate the sub-expressions used in those statements and store them in auxiliary variables (called $\mathbf{x_f}$). Then we look up the predicate bit (via statement $\mathbf{let\ pr = rd_p\ -1\ in\ \cdots}$) and store it in variable pr. Finally, using the conditional assignment, we set the result of those expressions to 0 if the predicate bit is 0 (true). So we can conclude that $[\![\cdot]\!]^s$ is $RSSC$.

Theorem 6 (The SSLH compiler is $RSSC$)**.** $\vdash [\![\cdot]\!]^s : RSSC$

This holds 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 is true since both cannot be tampered by the attacker, the compiler sets -1 up correctly, and the assignments right after the branches correctly update it (via the negation of the guard $\mathbf{x_f}$). Second, whenever speculation is happening, the result of any possibly-leaking expression is set to a constant 0, and constants are S since they leak no information. This last point means that any label generated when speculating will rely on S-tainted data, and so it will be tainted S and so it will not violate RSS.

NISLH: a Secure Non-Interprocedural SLH: It is also possible to secure the variant of SLH that does not carry the predicate bit across procedures. We model NISLH as $[\![\cdot]\!]^s_n$ by having the predicate bit initialized at the beginning of each function to 1 (false) in a local variable pr. Compiled code updates pr as before after every branching instruction. To ensure that pr correctly captures whether we are misspeculating, we place an lfence as the first instruction of every compiled function, otherwise the same vulnerability of Example 7 would arise.

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

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

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

Theorem 7 (The NISLH compiler is $RSSC$)**.** $\vdash [\![\cdot]\!]^s_n : RSSC$

Stronger Results for $[\![\cdot]\!]^s$: Compiler developers strive for optimising compiled code performance and we note that our SSLH is not efficient in the way masks are applied. For example, the compilation of $\mathsf{let\ x = rd_p\ 1\ in\ 2 :=_p x; 3 :=_p x}$ masks x twice, once in each assignment. Existing work has devised a tool called Blade that calculates the optimal place where to place the masking correctly [62]. To calculate the optimal position, Blade first finds sources of speculation (e.g., reading into x) as well as sinks where those sources are used (e.g., the two assignments to x). Then by using a min-max cutoff algorithm, Blade finds the minimal set of variables that need to be masked in order to eliminate paths between sources and sinks. In our example Blade emits a single masking before both assignments. We believe Blade $RSSC$ and thus secure, its proof follows the same insights of Figure 4 below. This proof requires reasoning about Blade's min-max cutoff algorithm so we leave it as future work.

C. Proof Outline of RSSC for $[\![\cdot]\!]^f$, $[\![\cdot]\!]^s$ and $[\![\cdot]\!]^s_n$

We use the simple $[\![\cdot]\!]^f$ countermeasure to illustrate the proof technique used to prove both countermeasures secure. As mentioned in Section IV, to prove that a compiler is $RSSC$ we take a target attacker (A) and create a source attacker ($\mathsf{A} = \langle\!\langle \mathsf{A} \rangle\!\rangle$) so that the two behave the same; this is called *backtranslation*. In this case, the backtranslation function ($\langle\!\langle \cdot \rangle\!\rangle$) homomorphically translates target heaps, functions, statements etc. into source ones; since it is straightforward we omit it. A benefit of our setup is that we can use the same backtranslation for both proofs since the backtranslation operates on attacker code and no countermeasure is applied to the attacker.

To prove the compiler is $RSSC$ we need to show that given a trace produced by the execution of attacker and compiled code, the backtranslated attacker and the source code produce a related trace (according to the trace relation of Section IV). This can be broken down by first setting up a cross-language relation between source and target states and then by proving that reductions preserve this relation and generate related traces. The state relation we use is fairly 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 (Appendix O). That

is, the heaps bind the same locations to the same values, the bindings bind the same variables to the same values.

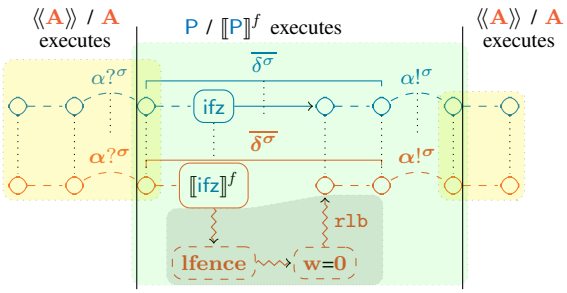

Figure 3. A diagram depicting the proof that $[\![\cdot]\!]^f$ is *RSSC*.

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

To reason about attacker code, we use a lock-step simulation: we show that starting from related states, if A does a step, then ⟨⟨A⟩⟩ does the same step and ends up in related states (yellow areas). To reason about component code, we adapt a reasoning commonly used in compiler correctness results [10, 37]. That is, if s steps and emits a trace, then $[\![s]\!]^f$ does one or more steps and emits a trace such that, the ending states as well as the traces are related (green areas, 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 (grey area). When observing target-level executions for $[\![\text{ifz}]\!]^f$, we see that the cross-language state relation is temporarily broken. After the $[\![\text{ifz}]\!]^f$ is executed, speculation starts (so the stack of target states is not a singleton and 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. This is the part where the proofs of the $[\![\cdot]\!]^s$ and $[\![\cdot]\!]^s_n$ countermeasures gets more complicated (Figure 4).

When $[\![\cdot]\!]^s$ is applied, speculation is not rolled back immediately after the then or else branch start executing. Instead, execution can continue for ω steps, spanning both attacker and compiled code and generating a trace $\overline{\lambda^\sigma}$. We need to prove that this trace is related to the empty source trace because this is only possible when all actions in $\overline{\lambda^\sigma}$ are tainted S, and so they do not leak. For this, we need to declare 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 that pr correctly captures whether we speculate or not and the mask

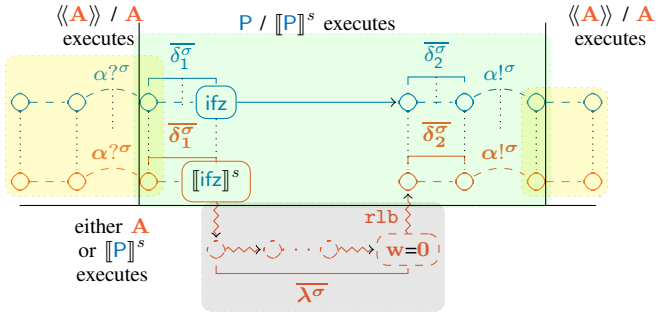

Figure 4. A diagram depicting the proof that $[\![\cdot]\!]^s$ and $[\![\cdot]\!]^s_n$ are *RSSC*.

used by $[\![\cdot]\!]^s$ is S. As already shown, both conditions hold for $[\![\cdot]\!]^s$ and $[\![\cdot]\!]^s_n$, so we can conclude that they are *RSSC*.

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

VII. Beyond Spectre v1

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

- Spectre v2 (also called Spectre-BTB) [35] exploits speculation over indirect jump instructions. The *retpoline* compiler-level countermeasure [30] replaces indirect jumps with return-based trampoline (thus the name retpoline) that leads to effectively dead code. As a result, the speculated jump executes no code and thus cannot leak anything.
- Spectre-RSB [40], in contrast, exploits speculation over return addresses (through `ret` instructions). To prevent it, Intel deployed a microcode update [30] that renders *retpoline* a valid countermeasure also against Spectre-RSB [15].
- Finally, Spectre v4 (also called Spectre-STL) [28] exploits speculation over data dependencies between in-flight store and load operations (used for memory disambiguation). To mitigate it, ARM introduced a dedicated `SSBB` speculation barrier instructions to prevent store bypasses that could potentially be injected as a compiler-level countermeasure. Intel and AMD, instead, deployed microcode updates mitigating Spectre v4 which effectively amount to changing the T semantics and are therefore out of scope for this paper.

To reason about compiler-level countermeasures against these Spectre variants, we first have to extend our speculation se-

mantics in T to capture the new kinds of speculative execution. This can be done, for instance, by adapting concepts from existing speculative semantics [9, 17, 42, 62] to our setting.

Next, we can apply our methodology to the new countermeasures. We believe that the aforementioned compiler-level countermeasures for the other Spectre variants are $RSSP$, and, therefore, the new proofs would follow the high-level structure presented in Section VI-C. Concretely, for reasoning about *retpolines* we would follow the approach of Figure 4 since they effectively do not halt execution. We would have to prove that dead code reached after a retpoline maintains the property that no sensitive data is loaded, which in intuitively true. Reasoning about `SSBB` would instead follow the same approach of Figure 3 since speculation is effectively immediately stopped.

VIII. Related Work

Speculative execution attacks: Many attacks analogous to Spectre [33, 35] have been discovered; they differ in the exploited speculation sources [28, 36, 39], the covert channels [56, 58, 61], or the target platforms [19]. We refer the reader to [15] for a survey of speculative execution attacks and their countermeasures.

Speculative semantics: Researchers recently proposed speculative semantics to model the effects of speculative execution. Several speculative semantics [9, 17, 42, 62] closely resemble microarchitectural implementations by explicitly modeling, for instance, multiple pipeline stages, reorder buffers, caches, and branch predictors, and they can reason about Spectre variants beyond Spectre v1. These semantics are significantly more complex (and potentially more brittle) than ours (which is inspired by [26]), and they would lead to much harder proofs.

Security conditions against Spectre attacks: SNI, initially proposed by Guarnieri *et al.* [26], has been used as security condition against speculative leaks also by Vassena *et al.* [62] and Balliu *et al.* [9] (with different speculative semantics). Cheang *et al.* [18], instead, proposed a similar property called trace property-dependent observational determinism. Finally, Cauligi *et al.* [17] extend the constant-time security condition to the entire speculative semantics, thereby obtaining speculative constant-time. Differently from SNI, speculative constant-time captures leaks under the non-speculative *and* the speculative semantics. Hence, speculative constant-time is inadequate for reasoning about compiler-level Spectre countermeasures which only modify a program's speculative behaviour.

Compiler-level countermeasures for Spectre v1: Apart from the already-discussed insertion of speculation barriers [5, 29], from speculative load hardening [16, 44] and from Blade [62], few countermeasures for Spectre v1 exist. Replacing branch instructions with equivalent branchless computations (using `cmov` and bit masking) is an effective countermeasure [51], which unfortunately is not generally applicable. oo7 [63] is a binary analysis tool that automatically patches speculative leaks by injective speculation barriers. Similarly to the Microsoft Visual C++ compiler, however, oo7 misses some speculative leaks (see [26]), fails in injecting all necessary speculation barriers, and ultimately violates $RSNIP$.

Secure compilation: As already mentioned, $RSSC$ (and $RSSP$) are instantiations of the robustly-safe compilation introduced by [50] and also studied by [3, 4]. Unlike their work, we deal with speculative semantics in the target language. Like [3, 50], we need to set-up a cross-language relation in order to relate source and target traces. Other formal statements for secure compilation exist, but we believe none of the existing alternatives provide as strong a guarantee as our choice.

Fully abstract compilation (FAC) is a criterion that has been widely used to prove security of compilers for different source languages (Java-like, C-like, Javascript, ML-like) to target languages enriched with many protection mechanisms (Intel SGX-like isolation, Cheri-like capabilities, tagged architectures and more) [24, 32, 47, 48, 54, 57]. To prove FAC, a compiler must preserve (and reflect) observational equivalence of source programs in their compiled counterparts [1, 48]. Recently, FAC has been questioned since it led to inefficient compiled code where the source of inefficiency was not a security vulnerability [2, 4, 32, 49]. In the setting of speculative execution, it is unclear whether being subject to speculative execution can be correctly captured by observational equivalence and thus if FAC is the right choice. In fact, if all execution is eventually rolled back, two programs that would be inequivalent due to speculation become equivalent simply because their speculative execution cannot affect their final result. Thus, we do not believe that FAC is a good candidate to talk about security (and insecurity) of Spectre countermeasures.

Constant-time-preserving compilation ($CTPC$) is a criterion that has been used to show that certain compiler (as well as optimisations) are constant-time [7, 10, 11]. To prove $CTPC$, a compiler must preserve observational non-interference, i.e., it must take pairs of attacker-indistinguishable source states into pairs of indistinguishable target states. Thus, $CTPC$ suffers from the same problems as $RSNIP$: it effectively amounts to proving the preservation of a hypersafety property across compilation, which can be more complex than preserving a safety property. Additionally, $CTPC$ has been devised for whole programs only (much like SNI) and thus it does not protect against active attackers.

Finally, robustly-safe compilation can be tailored to the case where (source) components can perform undefined behaviour [2]. Since undefined behaviour is not the source of worry in our case, we do not use this criterion.

Verifying Hypersafety as Safety Properties: Properties such as RSNI (and general non-interference) are 2-hypersafety properties [20]. Verifying whether a program satisfies a 2-hypersafety property is notoriously difficult because it effectively amounts to analysing two runs of the program. Existing approaches to this kind of verification include taint-tracking [6, 55] (which over-approximates the 2-Hypersafety property to a safety property), secure multi-execution [22] (which effectively runs the code twice in parallel) and self-composition [12, 60] (which runs the code twice sequentially). We believe the latter two approaches can also be adopted to derive a variation of RSS; we chose taint tracking for

simplicity and leave investigating the others as future work. In those approaches, as in ours, security-sensitive data needs to be marked (e.g., we taint the protected heap as U). Then, instead of carrying the taint in the trace semantics, those approaches would need to identify when code terminates and re-run by varying the security-sensitive data (effectively running the other low-equivalent program required by Definition 13).

IX. Conclusion

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

APPENDIX A
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 ($\longrightarrow$) 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 provate 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.

A. Syntax

$$
\begin{aligned}
\textit{Whole Programs}\ \mathsf{W} &::= \mathsf{H}, \overline{\mathsf{F}}, \overline{\mathsf{I}} \\
\textit{Programs}\ \mathsf{P} &::= \mathsf{H}, \overline{\mathsf{F}}, \overline{\mathsf{I}} \\
\textit{Components}\ \mathsf{C} &::= \overline{\mathsf{F}}, \overline{\mathsf{I}} \\
\textit{Contexts}\ \mathsf{A} &::= \mathsf{H}, \overline{\mathsf{F}}\,[\cdot] \\
\textit{Imports}\ \mathsf{I} &::= \mathsf{f} \\
\textit{Functions}\ \mathsf{F} &::= \mathsf{f}(\mathsf{x}) \mapsto \mathsf{s}; \mathsf{return}; \\
\textit{Operations}\ \oplus &::= + \mid - \mid \cdot \\
\textit{Comparisons}\ \otimes &::= == \mid < \mid > \\
\textit{Values}\ \mathsf{v} &::= \mathsf{n} \in \mathbb{N} \\
\textit{Expressions}\ \mathsf{e} &::= \mathsf{x} \mid \mathsf{v} \mid \mathsf{e} \oplus \mathsf{e} \mid \mathsf{e} \otimes \mathsf{e} \\
\textit{Statements}\ \mathsf{s} &::= \mathsf{skip} \mid \mathsf{s}; \mathsf{s} \mid \mathsf{let}\ \mathsf{x} = \mathsf{e}\ \mathsf{in}\ \mathsf{s} \mid \mathsf{ifz}\ \mathsf{e}\ \mathsf{then}\ \mathsf{s}\ \mathsf{else}\ \mathsf{s} \\
&\quad \mid \mathsf{call}\ \mathsf{f}\ \mathsf{e} \mid \mathsf{e} := \mathsf{e} \mid \mathsf{let}\ \mathsf{x} = \mathsf{rd}\ \mathsf{e}\ \mathsf{in}\ \mathsf{s} \mid \mathsf{let}\ \mathsf{x} = \mathsf{rd}_{\mathsf{p}}\ \mathsf{e}\ \mathsf{in}\ \mathsf{s} \mid \mathsf{e} :=_{\mathsf{p}} \mathsf{e} \\
\textit{Security Tags}\ \sigma &::= \mathsf{S} \mid \mathsf{U} \\
\textit{Heaps}\ \mathsf{H} &::= \varnothing \mid \mathsf{H}; \mathsf{n} \mapsto \mathsf{v} : \sigma \text{ where } \mathsf{n} \in \mathbb{Z} \\
\textit{Bindings}\ \mathsf{B} &::= \varnothing \mid \mathsf{B}; \mathsf{x} \mapsto \mathsf{v} : \sigma \\
\textit{Prog. States}\ \Omega &::= \mathsf{C}, \mathsf{H}, \overline{\mathsf{B}} \triangleright (\mathsf{s})_{\overline{\mathsf{f}}} \\
\textit{Labels}\ \lambda &::= \epsilon \mid \alpha \mid \delta \mid \lightning \\
\textit{Actions}\ \alpha &::= (\mathtt{call}\ \mathsf{f}\ \mathsf{v}?)^{\sigma} \mid (\mathtt{call}\ \mathsf{f}\ \mathsf{v}!)^{\sigma} \mid (\mathtt{ret}!)^{\sigma} \mid (\mathtt{ret}?)^{\sigma} \\
\textit{Heap\&Pc Act.s}\ \delta &::= (\mathtt{read}(\mathsf{n}))^{\sigma} \mid (\mathtt{write}(\mathsf{n}))^{\sigma} \mid (\mathtt{if}(\mathsf{v}))^{\sigma} \mid (\mathtt{write}(\mathsf{n} \mapsto \mathsf{v}))^{\sigma} \\
\textit{Traces}\ \overline{\lambda} &::= \varnothing \mid \overline{\lambda} \cdot \alpha \mid \overline{\lambda} \cdot \delta
\end{aligned}
$$

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

The additional condition on a trace $\overline{\lambda}$ is that it is list with this shape: $\overline{\alpha?\overline{\delta}\alpha!}$. 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 $\mathtt{write}(0)$ and $\mathtt{write}(0) \cdot \mathtt{write}(0)$.

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 $\mathsf{B} \cdot \mathsf{B}'$. We update the bindings for x in a stack B to v by writing $\mathsf{B} \cup \mathsf{x} \mapsto \mathsf{v}$. 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).

B. Dynamic Semantics

Rules Jump-Internal to 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

$$\frac{\mathsf{C} = \overline{\mathsf{F}}, \overline{\mathsf{I}}}{\mathsf{C}.\mathtt{intfs} = \overline{\mathsf{I}}}\ (\text{Intfs}) \qquad \frac{\mathsf{C} = \overline{\mathsf{F}}, \overline{\mathsf{I}}}{\mathsf{C}.\mathtt{funs} = \overline{\mathsf{F}}}\ (\text{Funs})$$

$$\frac{((\mathsf{f}' \in \overline{\mathsf{I}} \wedge \mathsf{f} \in \overline{\mathsf{I}}) \vee (\mathsf{f}' \notin \overline{\mathsf{I}} \wedge \mathsf{f} \notin \overline{\mathsf{I}}))}{\overline{\mathsf{I}} \vdash \mathsf{f}, \mathsf{f}' : \mathsf{internal}}\ (\text{L-Jump-Internal}) \qquad \frac{\mathsf{f} \in \overline{\mathsf{I}} \wedge \mathsf{f}' \notin \overline{\mathsf{I}}}{\overline{\mathsf{I}} \vdash \mathsf{f}, \mathsf{f}' : \mathsf{in}}\ (\text{L-Jump-IN}) \qquad \frac{\mathsf{f} \notin \overline{\mathsf{I}} \wedge \mathsf{f}' \in \overline{\mathsf{I}}}{\overline{\mathsf{I}} \vdash \mathsf{f}, \mathsf{f}' : \mathsf{out}}\ (\text{L-Jump-OUT})$$

$$\frac{\mathsf{A} \equiv \mathsf{H}, \overline{\mathsf{F}}\,[\cdot] \qquad \mathsf{P} \equiv \mathsf{H}', \overline{\mathsf{F}'}, \overline{\mathsf{I}} \qquad \vdash \overline{\mathsf{F}'}; \overline{\mathsf{F}}, \overline{\mathsf{I}} : \mathsf{whole} \qquad \mathsf{main} \in \mathtt{names}\left(\overline{\mathsf{F}}\right) \quad \mathtt{dom}\left(\mathsf{H}\right) \cap \mathtt{dom}\left(\mathsf{H}'\right) = \varnothing \quad \forall \mathsf{n} \mapsto \mathsf{v} : \sigma \in \mathsf{H}', \mathsf{n} < 0 \text{ and } \sigma = \mathsf{U}}{\mathsf{A}\,[\mathsf{P}] = \mathsf{H}; \mathsf{H}', \overline{\mathsf{F}}; \overline{\mathsf{F}'}, \mathtt{dom}\left(\overline{\mathsf{F}}\right)}\ (\text{L-Plug})$$

$$\frac{\mathtt{names}\left(\overline{\mathsf{F}}\right) \cap \mathtt{names}\left(\overline{\mathsf{F}'}\right) = \varnothing \qquad \mathtt{names}\left(\overline{\mathsf{I}}\right) \subseteq \mathtt{names}\left(\overline{\mathsf{F}}\right) \cup \mathtt{names}\left(\overline{\mathsf{F}'}\right) \qquad \mathtt{fv}\left(\overline{\mathsf{F}}\right) \cup \mathtt{fv}\left(\overline{\mathsf{F}'}\right) = \varnothing}{\vdash \overline{\mathsf{F}'}; \overline{\mathsf{F}}, \overline{\mathsf{I}} : \mathsf{whole}}\ (\text{L-Whole})$$

$$\frac{\mathsf{H}_0 = \mathsf{H}'' \cup \mathsf{H} \cup \mathsf{H}' \qquad \mathsf{H}' = \{\mathsf{n} \mapsto 0 : \mathsf{S} \mid \mathsf{n} \in \mathbb{N} \setminus \mathtt{dom}\left(\mathsf{H}\right)\} \qquad \mathsf{H}'' = \{-\mathsf{n} \mapsto 0 : \mathsf{U} \mid \mathsf{n} \in \mathbb{N}, -\mathsf{n} \notin \mathtt{dom}\left(\mathsf{H}\right)\}}{\Omega_0\left(\left(\mathsf{H}, \overline{\mathsf{F}}, \overline{\mathsf{I}}\right)\right) = \overline{\mathsf{F}}, \overline{\mathsf{I}}, \mathsf{H}_0, \varnothing \cdot \mathsf{x} \mapsto 0 \triangleright \mathsf{call\ main\ x}}\ (\text{L-Initial State})$$

$$\frac{\nexists \Omega', \lambda.\Omega \xrightarrow{\lambda} \Omega'}{\vdash \Omega : \bot}\ (\text{L-Terminal State})$$

1) Component Semantics:

Judgements

$\mathsf{B} \triangleright \mathsf{e} \downarrow \mathsf{v} : \sigma$	Expression e big-steps to value v tagged σ.
$\mathsf{C}, \mathsf{H}, \overline{\mathsf{B}} \triangleright (\mathsf{s})_{\overline{\mathsf{f}}} \xrightarrow{\lambda^\sigma} \mathsf{C}, \mathsf{H}', \overline{\mathsf{B}'} \triangleright (\mathsf{s}')_{\overline{\mathsf{f}'}}$	Statement s reduces to s′ and evolves the rest accordingly, emitting tagged label λ^σ.
$\Omega \xRightarrow{\overline{\lambda^\sigma}} \Omega'$	Program state Ω steps to Ω' emitting tagged trace $\overline{\lambda^\sigma}$.
$\mathsf{P} \rightsquigarrow \overline{\lambda^\sigma}$	Whole program P produces tagged trace $\overline{\lambda^\sigma}$

$\mathsf{B} \triangleright \mathsf{e} \downarrow \mathsf{v} : \sigma$

$$\frac{}{B \triangleright v \downarrow v : S}\ \text{(E-L-val)} \qquad \frac{B(x) = v : \sigma}{B \triangleright x \downarrow v : \sigma}\ \text{(E-L-var)}$$

$$\frac{B \triangleright e \downarrow n : \sigma \quad B \triangleright e' \downarrow n' : \sigma' \quad n'' = [n \oplus n'] \quad \sigma'' = \sigma \sqcup \sigma'}{B \triangleright e \oplus e' \downarrow n'' : \sigma''}\ \text{(E-L-op)}$$

$$\frac{B \triangleright e \downarrow n : \sigma \quad B \triangleright e' \downarrow n' : \sigma' \quad n'' = [n \otimes n'] \quad \sigma'' = \sigma \sqcup \sigma'}{B \triangleright e \otimes e' \downarrow n'' : \sigma''}\ \text{(E-L-comparison)}$$

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.

$$\boxed{C, H, \overline{B} \triangleright s \xrightarrow{\lambda^\sigma} C', H', \overline{B'} \triangleright s'}$$

$$\frac{}{C, H, \overline{B} \triangleright \text{skip}; s \xrightarrow{\epsilon} C, H, \overline{B} \triangleright s}\ \text{(E-L-sequence)} \qquad \frac{C, H, \overline{B} \triangleright s \xrightarrow{\lambda^\sigma} C, H', \overline{B'} \triangleright s'}{C, H \triangleright s; s'' \xrightarrow{\lambda^\sigma} C, H', \overline{B'} \triangleright s'; s''}\ \text{(E-L-step)}$$

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

$$\frac{B \triangleright e \downarrow n : \sigma \quad n > 0}{C, H, \overline{B} \cdot B \triangleright \text{ifz } e \text{ then } s \text{ else } s' \xrightarrow{(\text{if}(n))^\sigma} C, H, \overline{B} \cdot B \triangleright s}\ \text{(E-L-if-false)}$$

$$\frac{B \triangleright e \downarrow v : \sigma}{C, H, \overline{B} \cdot B \triangleright \text{let } x = e \text{ in } s \xrightarrow{\epsilon} C, H, \overline{B} \cdot B \cup x \mapsto v : \sigma \triangleright s}\ \text{(E-L-letin)}$$

$$\frac{B \triangleright e \downarrow n : \sigma \quad B \triangleright e' \downarrow v : \sigma'' \quad H = H_1; |n| \mapsto v' : \sigma'; H_2 \quad H' = H_1; |n| \mapsto v : S; H_2}{C, H, \overline{B} \cdot B \triangleright e := e' \xrightarrow{\text{write}(|n| \mapsto v)^{\sigma \sqcup \sigma''}} C, H', \overline{B} \cdot B \triangleright \text{skip}}\ \text{(E-L-write)}$$

$$\frac{B \triangleright e \downarrow n : \sigma' \quad H = H_1; |n| \mapsto v : \sigma; H_2}{C, H, \overline{B} \cdot B \triangleright \text{let } x = \text{rd } e \text{ in } s \xrightarrow{\text{read}(|n|)^{\sigma'}} C, H, \overline{B} \cdot B \cup x \mapsto v : \sigma \triangleright s}\ \text{(E-L-read)}$$

$$\frac{B \triangleright e \downarrow n : \sigma \quad B \triangleright e' \downarrow v : \sigma'' \quad n_a = -|n| \quad H = H_1; n_a \mapsto v' : \sigma'; H_2 \quad H' = H_1; n_a \mapsto v : \sigma; H_2}{C, H, \overline{B} \cdot B \triangleright e :=_p e' \xrightarrow{\text{write}(n_a)^\sigma} C, H', \overline{B} \cdot B \triangleright \text{skip}}\ \text{(E-L-write-prv)}$$

$$\frac{B \triangleright e \downarrow n : \sigma' \quad n_a = -|n| \quad H = H_1; n_a \mapsto v : \sigma; H_2 \quad \sigma'' = \sigma \sqcup \sigma'}{C, H, \overline{B} \cdot B \triangleright \text{let } x = \text{rd}_p \ e \text{ in } s \xrightarrow{\text{read}(n_a)^{\sigma''}} C, H, \overline{B} \cdot B \cup x \mapsto v : U \triangleright s}\ \text{(E-L-read-prv)}$$

$$\frac{C.\texttt{intfs} \vdash f, f' : \text{internal} \quad \overline{f'} = \overline{f''}; f' \quad f(x) \mapsto s; \text{return}; \in C.\texttt{funs} \quad B \triangleright e \downarrow v : \sigma}{C, H, \overline{B} \cdot B \triangleright (\text{call } f \ e)_{\overline{f'}} \xrightarrow{\epsilon} C, H, \overline{B} \cdot B \cdot x \mapsto v : \sigma \triangleright (s; \text{return};)_{\overline{f'}; f}}\ \text{(E-L-call-internal)}$$

$$\frac{\overline{f'} = \overline{f''}; f' \quad f(x) \mapsto s; \text{return}; \in C.\texttt{funs} \quad C.\texttt{intfs} \vdash f', f : \text{out} \quad B \triangleright e \downarrow v : \sigma}{C, H, \overline{B} \cdot B \triangleright (\text{call } f \ e)_{\overline{f'}} \xrightarrow{\text{call } f \ v!^\sigma} C, H, \overline{B} \cdot B \cdot x \mapsto v : S \triangleright (s; \text{return};)_{\overline{f'}; f}}\ \text{(E-L-callback)}$$

$$\frac{\overline{f'} = \overline{f''}; f' \quad f(x) \mapsto s; \text{return}; \in C.\texttt{funs} \quad C.\texttt{intfs} \vdash f', f : \text{in} \quad B \triangleright e \downarrow v : \sigma}{C, H, \overline{B} \cdot B \triangleright (\text{call } f \ e)_{\overline{f'}} \xrightarrow{\text{call } f \ v?^\sigma} C, H, \overline{B} \cdot B \cdot x \mapsto v : S \triangleright (s; \text{return};)_{\overline{f'}; f}}\ \text{(E-L-call)}$$

$$\frac{\overline{\mathsf{f}'} = \overline{\mathsf{f}''};\mathsf{f}' \qquad \mathsf{C}.\texttt{intfs} \vdash \mathsf{f},\mathsf{f}' : \mathsf{internal}}{\mathsf{C},\mathsf{H},\overline{\mathsf{B}}\cdot\mathsf{B} \triangleright (\mathsf{return};)_{\overline{\mathsf{f}'};\mathsf{f}} \xrightarrow{\epsilon} \mathsf{C},\mathsf{H},\overline{\mathsf{B}} \triangleright (\mathsf{skip})_{\overline{\mathsf{f}'}}}\ (\text{E-L-ret-internal})$$

$$\frac{\overline{\mathsf{f}'} = \overline{\mathsf{f}''};\mathsf{f}' \qquad \mathsf{C}.\texttt{intfs} \vdash \mathsf{f},\mathsf{f}' : \mathsf{in}}{\mathsf{C},\mathsf{H},\overline{\mathsf{B}}\cdot\mathsf{B} \triangleright (\mathsf{return};)_{\overline{\mathsf{f}'};\mathsf{f}} \xrightarrow{\texttt{ret}?^{\mathsf{S}}} \mathsf{C},\mathsf{H},\overline{\mathsf{B}} \triangleright (\mathsf{skip})_{\overline{\mathsf{f}'}}}\ (\text{E-L-retback})$$

$$\frac{\overline{\mathsf{f}'} = \overline{\mathsf{f}''};\mathsf{f}' \qquad \mathsf{C}.\texttt{intfs} \vdash \mathsf{f},\mathsf{f}' : \mathsf{out}}{\mathsf{C},\mathsf{H},\overline{\mathsf{B}}\cdot\mathsf{B} \triangleright (\mathsf{return};)_{\overline{\mathsf{f}'};\mathsf{f}} \xrightarrow{\texttt{ret}!^{\mathsf{S}}} \mathsf{C},\mathsf{H},\overline{\mathsf{B}} \triangleright (\mathsf{skip})_{\overline{\mathsf{f}'}}}\ (\text{E-L-return})$$

The provate heap is read and written at an address that is shifted by 1 to ensure that location -1 is not ever accessed. This technicality is needed for the compiler of Section J and it is explained there.

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 rollbacked). 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.

$\boxed{\Omega \xRightarrow{\overline{\lambda^{\sigma}}} \Omega'}$

$$\frac{\Omega \xRightarrow{\overline{\lambda_1^{\sigma}}} \Omega'' \qquad \Omega'' \xrightarrow{\alpha^{\sigma}} \Omega' \qquad \Omega'' = \overline{\mathsf{F}},\overline{\mathsf{I}},\mathsf{H},\mathsf{B} \triangleright (\mathsf{s})_{\overline{\mathsf{f}}\cdot\mathsf{f}} \qquad \Omega' = \overline{\mathsf{F}},\overline{\mathsf{I}},\mathsf{H}',\mathsf{B}' \triangleright (\mathsf{s}')_{\overline{\mathsf{f}'}\cdot\mathsf{f}'}}{\text{if } \mathsf{f} == \mathsf{f}' \text{ and } \mathsf{f} \in \mathsf{I} \text{ then } \overline{\lambda_2^{\sigma}} = \overline{\lambda_1^{\sigma}} \text{ else } \overline{\lambda_2^{\sigma}} = \overline{\lambda_1^{\sigma}} \cdot \alpha^{\mathsf{S}}}\ (\text{E-L-single})$$

$$\frac{\Omega \xrightarrow{\epsilon} \Omega'}{\Omega \Rightarrow \Omega'}\ (\text{E-L-silent})$$

$$\frac{\Omega \xRightarrow{\overline{\lambda_1^{\sigma_1}}} \Omega'' \qquad \Omega'' \xRightarrow{\overline{\lambda_2^{\sigma_2}}} \Omega'}{\Omega \xRightarrow{\overline{\lambda_1^{\sigma_1}}\cdot\overline{\lambda_2^{\sigma_2}}} \Omega'}\ (\text{E-L-trans})$$

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.

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

$$\frac{\exists \Omega.\, \vdash \Omega : \bot \qquad \Omega_0(\mathsf{P}) \xRightarrow{\overline{\lambda^{\sigma}}} \Omega}{\mathsf{P} \rightsquigarrow \overline{\lambda^{\sigma}}\!\!\not\downarrow}\ (\text{E-L-trace})$$

$$\frac{}{\mathsf{Beh}(\mathsf{P}) = \left\{\overline{\lambda^{\sigma}} \;\middle|\; \mathsf{P} \rightsquigarrow \overline{\lambda^{\sigma}}\right\}}\ (\text{E-L-behaviour})$$

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

a) Alternative Semantics Definition: $\boxed{\mathsf{P} \rightsquigarrow \overline{\lambda^{\sigma}}}$

$$\frac{\exists \Omega.\, \Omega_0(\mathsf{P}) \xRightarrow{\overline{\lambda^{\sigma}}} \Omega}{\mathsf{P} \rightsquigarrow \overline{\lambda^{\sigma}}}\ (\text{E-L-trace})$$

This defnition drops the condition of Ω 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 10 (Dynamic Speculative Safety (SS)) and Definition 14 (Robust Speculative Non-Interference) so we do not use this.

APPENDIX B
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 againsts 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 if L.

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

A. Syntax

All elements from L also exist in this language.

$$
\begin{aligned}
\textit{Statements } \mathbf{s} &::= \cdots \mid \mathbf{lfence} \mid \mathbf{let\ x = e\ (if\ e)\ in\ s} \\
\textit{Speculation States } \mathbf{\Sigma} &::= \mathbf{n}, \overline{(\Omega, \omega, \sigma)} \\
\textit{Actions } \boldsymbol{\alpha} &::= \mathtt{rlb} \\
\textit{Window } \boldsymbol{\omega} &::= \mathbf{n} \mid \bot \\
\textit{Droppable Names } \mathbf{D} &::= \varnothing \mid \mathbf{D}, \mathbf{x}
\end{aligned}
$$

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

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

Droppable names are a technicality needed for cross-language relations, as explained in Section I-A3.

B. Dynamic Semantics

1) Component Semantics:

$\mathbf{\Sigma} \overset{\overline{\lambda^\sigma}}{\rightsquigarrow} \mathbf{\Sigma}'$ Speculative state $\mathbf{\Sigma}$ evolves into $\mathbf{\Sigma}'$ emitting tagged action $\boldsymbol{\alpha}^\sigma$.

$$\boxed{\mathbf{C}, \mathbf{H} \triangleright \mathbf{s} \xrightarrow{\lambda^\sigma} \mathbf{C}', \mathbf{H}' \triangleright \mathbf{s}'}$$

(E-T-lfence)

$$\frac{}{\mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \triangleright \mathbf{lfence} \xrightarrow{\epsilon} \mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \triangleright \mathbf{skip}}$$

(E-T-cmove-true)

$$\frac{\mathbf{x} \in \mathtt{dom}(\mathbf{B}) \quad \mathbf{B} \triangleright \mathbf{e}' \downarrow \mathbf{0} : \boldsymbol{\sigma}' \quad \mathbf{B} \triangleright \mathbf{e} \downarrow \mathbf{v} : \boldsymbol{\sigma} \quad \boldsymbol{\sigma}'' = \boldsymbol{\sigma} \sqcup \boldsymbol{\sigma}'}{\mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \cdot \mathbf{B} \triangleright \mathbf{let\ x = e\ (if\ e')\ in\ s} \xrightarrow{\epsilon} \mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \cdot \mathbf{B} \cup \mathbf{x} \mapsto \mathbf{v} : \boldsymbol{\sigma}'' \triangleright \mathbf{s}}$$

(E-T-cmove-false)

$$\frac{\mathbf{x} \in \mathtt{dom}(\mathbf{B}) \quad \mathbf{B} \triangleright \mathbf{e}' \downarrow \mathbf{n} : \boldsymbol{\sigma}' \quad \mathbf{n} > \mathbf{0} \quad \mathbf{B}(\mathbf{x}) = \mathbf{v} : \boldsymbol{\sigma} \quad \boldsymbol{\sigma}'' = \boldsymbol{\sigma} \sqcup \boldsymbol{\sigma}'}{\mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \cdot \mathbf{B} \triangleright \mathbf{let\ x = e\ (if\ e')\ in\ s} \xrightarrow{\epsilon} \mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \cdot \mathbf{B} \cup \mathbf{x} \mapsto \mathbf{v} : \boldsymbol{\sigma}'' \triangleright \mathbf{s}}$$

In a conditional move we require that $\mathbf{x}$ is always bound afterwards.

$$\boxed{\mathbf{\Sigma} \overset{\alpha^\sigma}{\rightsquigarrow} \mathbf{\Sigma}'}$$

(E-T-speculate-epsilon)

$$\frac{\mathbf{\Omega} \xrightarrow{\epsilon} \mathbf{\Omega}' \quad \mathbf{\Omega} \equiv \mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \triangleright \mathbf{s}; \mathbf{s}' \quad \mathbf{s} \not\equiv \mathbf{ifz\ \_\ then\ \_\ else\ \_} \text{ and } \mathbf{s} \not\equiv \mathbf{lfence}}{\mathbf{w}, \overline{(\mathbf{\Omega}, \boldsymbol{\omega}, \boldsymbol{\sigma})} \cdot (\mathbf{\Omega}, \mathbf{n+1}, \boldsymbol{\sigma}) \overset{\epsilon}{\rightsquigarrow} \mathbf{w}, \overline{(\mathbf{\Omega}, \boldsymbol{\omega}, \boldsymbol{\sigma})} \cdot (\mathbf{\Omega}', \mathbf{n}, \boldsymbol{\sigma})}$$

(E-T-speculate-lfence)

$$\frac{\mathbf{\Omega} \xrightarrow{\epsilon} \mathbf{\Omega}' \quad \mathbf{\Omega} \equiv \mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \triangleright \mathbf{s}; \mathbf{s}' \quad \mathbf{s} \equiv \mathbf{lfence}}{\mathbf{w}, \overline{(\mathbf{\Omega}, \boldsymbol{\omega}, \boldsymbol{\sigma})} \cdot (\mathbf{\Omega}, \mathbf{n+1}, \boldsymbol{\sigma}) \overset{\epsilon}{\rightsquigarrow} \mathbf{w}, \overline{(\mathbf{\Omega}, \boldsymbol{\omega}, \boldsymbol{\sigma})} \cdot (\mathbf{\Omega}', \mathbf{0}, \boldsymbol{\sigma})}$$

(E-T-speculate-action)

$$\frac{\mathbf{\Omega} \xrightarrow{\lambda^{\sigma'}} \mathbf{\Omega}' \quad \mathbf{\Omega} \equiv \mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \triangleright \mathbf{s}; \mathbf{s}' \quad \mathbf{s} \not\equiv \mathbf{ifz\ \_\ then\ \_\ else\ \_} \text{ and } \mathbf{s} \not\equiv \mathbf{lfence}}{\mathbf{w}, \overline{(\mathbf{\Omega}, \boldsymbol{\omega}, \boldsymbol{\sigma})} \cdot (\mathbf{\Omega}, \mathbf{n+1}, \boldsymbol{\sigma}) \overset{\lambda^{\sigma' \sqcap \sigma}}{\rightsquigarrow} \mathbf{w}, \overline{(\mathbf{\Omega}, \boldsymbol{\omega}, \boldsymbol{\sigma})} \cdot (\mathbf{\Omega}', \mathbf{n}, \boldsymbol{\sigma})}$$

(E-T-speculate-if)

$$\frac{\Omega \xrightarrow{\alpha^\sigma} \Omega' \quad \Omega \equiv C, H, \overline{B} \cdot B \triangleright (s; s')_{\overline{f} \cdot f} \quad s \equiv \text{if } e \text{ then } s'' \text{ else } s''' \quad C \equiv \overline{F}; \overline{I} \quad f \notin \overline{I} \quad \text{if } B \triangleright e \downarrow 0 : \sigma \text{ then } \Omega'' \equiv C, H, \overline{B} \cdot B \triangleright s'''; s' \quad \text{if } B \triangleright e \downarrow n : \sigma \text{ and } n > 0 \text{ then } \Omega'' \equiv C, H, \overline{B} \cdot B \triangleright s''; s' \quad j = \texttt{min}(w, n)}{w, \overline{(\Omega, \omega, \sigma)} \cdot (\Omega, n+1, \sigma') \xrightarrow{\alpha^\sigma \sqcap \sigma'} w, \overline{(\Omega, \omega, \sigma)} \cdot (\Omega', n, \sigma') \cdot (\Omega'', j, U)}$$

(E-T-speculate-rollback)

$$\frac{}{w, \overline{(\Omega, \omega, \sigma)} \cdot (\Omega, 0, \sigma) \xrightarrow{\texttt{rlb}} w, \overline{(\Omega, \omega, \sigma)}}$$

(E-T-speculate-rollback-stuck)

$$\frac{\vdash \Sigma : \bot \quad \Sigma = w, \overline{(\Omega, \omega, \sigma)} \cdot (\Omega, W, \sigma)}{\Sigma \xrightarrow{\texttt{rlb}} (w, \overline{(\Omega, \omega, \sigma)})}$$

We allow speculation only when it happens inside the component, not in the attacker (Rule E-T-speculate-if), for this reason. Suppose a context speculates and call us with a parameter 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 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).

$\boxed{\Sigma \xRightarrow{\overline{\lambda^\sigma}} \Sigma'}$

(E-T-single)

$$\frac{(n, \Sigma) \xRightarrow{\overline{\lambda_1^\sigma}} (n', \Sigma'') \quad \Sigma'' \xrightarrow{\alpha^\sigma} \Sigma' \quad \Sigma'' = (w, \overline{(\Omega, m, \sigma)} \cdot (\overline{F}, \overline{I}, H, \overline{B} \triangleright (s)_{\overline{f} \cdot f}, n, \sigma')) \quad \Sigma' = (w, \overline{(\Omega, m, \sigma)} \cdot (\overline{F}, \overline{I}, H', \overline{B'} \triangleright (s')_{\overline{f'} \cdot f'}, n', \sigma'')) \quad \text{if } f == f' \text{ and } f \in I \text{ then } \overline{\lambda_2^\sigma} = \overline{\lambda_1^\sigma} \text{ else } \overline{\lambda_2^\sigma} = \overline{\lambda_1^\sigma} \cdot \alpha^\sigma \quad \text{if } \alpha^\sigma == \texttt{rlb}^S \text{ then } j = n' \text{ else } j = n' + 1}{(n, \Sigma) \xRightarrow{\overline{\lambda_2^\sigma}} (j, \Sigma')}$$

(E-T-silent)

$$\frac{(n, \Sigma) \xRightarrow{\overline{\lambda_1^\sigma}} (n', \Sigma'') \quad \Sigma'' \xrightarrow{\epsilon} \Sigma'}{(n, \Sigma) \xRightarrow{\overline{\lambda_1^\sigma}} (n'+1, \Sigma')}$$

(E-T-init)

$$\frac{}{(n, \Sigma) \xRightarrow{\epsilon} (n, \Sigma)}$$

Helpers

(T-Initial State)

$$\frac{H_0 = H'' \cup H \cup H' \quad H' = \{n \mapsto 0 : S \mid n \in \mathbb{N} \setminus \texttt{dom}(H)\} \quad H'' = \{-n \mapsto 0 : U \mid n \in \mathbb{N}, -n \notin \texttt{dom}(H)\} \quad \Sigma \equiv w, (\overline{F}; \overline{I}; H_0; \varnothing \cdot x \mapsto 0 \triangleright \text{call main x; skip}, \bot, S)}{\Omega_0 ((H; \overline{F}; \overline{I})) = (0, \Sigma)}$$

(E-T-trace)

$$\frac{\exists \Sigma. \vdash \Sigma : \bot_f \quad \Omega_0 (P) \xRightarrow{\overline{\lambda^\sigma}} (\_, \Sigma)}{P \rightsquigarrow \overline{\lambda^\sigma}_{\not\downarrow}}$$

(E-T-behaviour)

$$\frac{}{\text{Beh}(P) = \{\overline{\lambda^\sigma} \mid P \rightsquigarrow \overline{\lambda^\sigma}\}}$$

(T-Terminal State)

$$\frac{\Sigma = w, \overline{(\Omega, \omega, \sigma)} \cdot (\Omega, W, \sigma) \quad \nexists \Omega', \lambda. \Omega \xrightarrow{\lambda} \Omega'}{\vdash \Sigma : \bot}$$

(T-Terminal Ending State)

$$\frac{\Sigma = w, (\Omega, \bot, S) \quad \vdash \Sigma : \bot}{\vdash \Sigma : \bot_f}$$

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). 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 $U \sqcap U = U$.

C. Model Introduction

We model a cross-procedural attacker, that is, the attacker lives in a separate process. For this, we do not let it speculate because any interaction it makes when speculating is anyway captured outside speculation ($\forall$). We provide insight about an interprocedural attacker: if that is allowed, then no security is possible. One can remedy defence from an interprocedural attacker only by means of memory protection (read/write) mechanisms, but their modelling is analogous to the cross-procedural attacker.

APPENDIX C
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 8 (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.

$$\left.\begin{array}{l} -1 \mapsto 0 : S; \\ -2 \mapsto 10 : S \\ -3 \mapsto 10 : S \\ -4 \mapsto 30 : S \end{array}\right., \begin{array}{l} \overline{B}\cdot \\ y \mapsto 0 : S; \\ size \mapsto 3 : S \end{array} \triangleright \begin{array}{l} \text{ifz } (y < size) \text{ then} \\ \quad \text{let } x_a = rd_p\ 1 + y \text{ in} \\ \quad \text{let } x_b = rd_p\ 4 + x_a \text{ in} \\ \quad \text{let temp}{=}x_b \text{ in skip} \\ \text{else skip} \end{array} \Bigg\} \Omega$$

Rule E-L-if-true

since $y \mapsto 0; size \mapsto 1 \triangleright (y < size) \downarrow 0 : S$

$$\xrightarrow{(\texttt{if}(0))^S} \left.\begin{array}{l} -1 \mapsto 0 : S; \\ -2 \mapsto 10 : S \\ -3 \mapsto 10 : S \\ -4 \mapsto 30 : S \end{array}\right., \begin{array}{l} \overline{B}\cdot \\ y \mapsto 0 : S; \\ size \mapsto 3 : S \end{array} \triangleright \begin{array}{l} \text{let } x_a = rd_p\ 1 + y \text{ in} \\ \text{let } x_b = rd_p\ 4 + x_a \text{ in} \\ \text{let temp}{=}x_b \text{ in skip} \end{array} \Bigg\} \Omega_1$$

Rule E-L-read

since $\cdots \triangleright 1 + y \downarrow 1 : S$ and $S \sqcup S = S$

$$\xrightarrow{\texttt{read}(-1)^S} \left.\begin{array}{l} -1 \mapsto 0 : S; \\ -2 \mapsto 10 : S \\ -3 \mapsto 10 : S \\ -4 \mapsto 30 : S \end{array}\right., \begin{array}{l} \overline{B}\cdot \\ y \mapsto 0 : S; \\ size \mapsto 3 : S \\ x_a \mapsto 0 : U \end{array} \triangleright \begin{array}{l} \text{let } x_b = rd_p\ 4 + x_a \text{ in} \\ \text{let temp}{=}x_b \text{ in skip} \end{array} \Bigg\} \Omega_2$$

Rule E-L-read

since $\cdots \triangleright x_a \downarrow 4 : U$ and $S \sqcup U = U$

$$\xrightarrow{\texttt{read}(-4)^U} \left.\begin{array}{l} -1 \mapsto 0 : S; \\ -2 \mapsto 10 : S \\ -3 \mapsto 10 : S \\ -4 \mapsto 30 : S \end{array}\right., \begin{array}{l} \overline{B}\cdot \\ y \mapsto 0 : S; \\ size \mapsto 3 : S \\ x_a \mapsto 0 : U \\ x_b \mapsto 30 : U \end{array} \triangleright \begin{array}{l} \text{let temp}{=}x_b \text{ in skip} \end{array} \Bigg\} \Omega_3$$

Rule E-L-letin

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

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:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Omega \Rightarrow \Omega} \quad \Omega \xrightarrow{(\texttt{if}(0))^{\mathsf{S}}} \Omega_1}{\Omega \xRightarrow{(\texttt{if}(0))^{\mathsf{S}}} \Omega_1} \quad \Omega_1 \xrightarrow{\texttt{read}(\ell_{\mathsf{A}})^{\mathsf{S}}} \Omega_2}{\Omega \xRightarrow{(\texttt{if}(0))^{\mathsf{S}}\cdot\texttt{read}(\ell_{\mathsf{A}})^{\mathsf{S}}} \Omega_2} \quad \Omega_2 \xrightarrow{\texttt{read}(\ell_{\mathsf{B}})^{\mathsf{U}}} \Omega_3}{\Omega \xRightarrow{(\texttt{if}(0))^{\mathsf{S}}\cdot\texttt{read}(\ell_{\mathsf{A}})^{\mathsf{S}}\cdot\texttt{read}(\ell_{\mathsf{B}})^{\mathsf{S}}} \Omega_3} \quad \Omega_3 \xrightarrow{\epsilon} \Omega_4}{\Omega \xRightarrow{(\texttt{if}(0))^{\mathsf{S}}\cdot\texttt{read}(\ell_{\mathsf{A}})^{\mathsf{S}}\cdot\texttt{read}(\ell_{\mathsf{B}})^{\mathsf{S}}} \Omega_4}$$

If we consider the same execution in $\mathbf{T}$, 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 $\mathbf{U}$. We start from a different state with $\mathbf{y} \mapsto \mathbf{2}$.

$$\left.\begin{array}{l} -1 \mapsto 10 : \mathbf{S}; \\ -2 \mapsto 0 : \mathbf{S} \\ -3 \mapsto 10 : \mathbf{S} \\ -4 \mapsto 30 : \mathbf{S} \end{array}\right., \left.\begin{array}{l} \overline{B}\cdot \\ \mathbf{y} \mapsto 2 : \mathbf{S}; \\ \mathbf{size} \mapsto 1 : \mathbf{S} \end{array}\right. \triangleright \left.\begin{array}{l} \textbf{ifz } (\mathbf{y} < \mathbf{size}) \textbf{ then} \\ \quad \textbf{let } \mathbf{x_a} = \mathbf{rd_p}\ 0 + \mathbf{y} \textbf{ in} \\ \quad \textbf{let } \mathbf{x_b} = \mathbf{rd_p}\ 4 + \mathbf{x_a} \textbf{ in} \\ \quad \textbf{let temp}{=}\mathbf{x_b} \textbf{ in skip} \\ \textbf{else skip} \end{array}\right\} \Omega$$

since $\mathbf{y} \mapsto 2; \mathbf{size} \mapsto 1 \triangleright (\mathbf{y} < \mathbf{size}) \downarrow 1 : \mathbf{S}$

$$\xrightarrow{(\texttt{if}(1))^{\mathsf{S}}} \left.\begin{array}{l} -1 \mapsto 10 : \mathbf{S}; \\ -2 \mapsto 0 : \mathbf{S} \\ -3 \mapsto 10 : \mathbf{S} \\ -4 \mapsto 30 : \mathbf{S} \end{array}\right., \left.\begin{array}{l} \overline{B}\cdot \\ \mathbf{y} \mapsto 2 : \mathbf{S}; \\ \mathbf{size} \mapsto 1 : \mathbf{S} \end{array}\right. \triangleright \textbf{skip} \Big\} \Omega'$$

the rest of the states needed for speculation are :

$$\xrightarrow{(\texttt{if}(0))^{\mathsf{S}}} \left.\begin{array}{l} -1 \mapsto 10 : \mathbf{S}; \\ -2 \mapsto 0 : \mathbf{S} \\ -3 \mapsto 10 : \mathbf{S} \\ -4 \mapsto 30 : \mathbf{S} \end{array}\right., \left.\begin{array}{l} \overline{B}\cdot \\ \mathbf{y} \mapsto 2 : \mathbf{S}; \\ \mathbf{size} \mapsto 1 : \mathbf{S} \end{array}\right. \triangleright \left.\begin{array}{l} \textbf{let } \mathbf{x_a} = \mathbf{rd_p}\ 0 + \mathbf{y} \textbf{ in} \\ \textbf{let } \mathbf{x_b} = \mathbf{rd_p}\ 4 + \mathbf{x_a} \textbf{ in} \\ \textbf{let temp}{=}\mathbf{x_b} \textbf{ in skip} \end{array}\right\} \Omega_1$$

since $\cdots \triangleright 0 + \mathbf{y} \downarrow 2 : \mathbf{S}$ and $\mathbf{S} \sqcup \mathbf{S} = \mathbf{S}$

$$\xrightarrow{\texttt{read}(-2)^{\mathsf{S}}} \left.\begin{array}{l} -1 \mapsto 10 : \mathbf{S}; \\ -2 \mapsto 0 : \mathbf{S} \\ -3 \mapsto 10 : \mathbf{S} \\ -4 \mapsto 30 : \mathbf{S} \end{array}\right., \left.\begin{array}{l} \overline{B}\cdot \\ \mathbf{y} \mapsto 2 : \mathbf{S}; \\ \mathbf{size} \mapsto 1 : \mathbf{S} \\ \mathbf{x_a} \mapsto 0 : \mathbf{U} \end{array}\right. \triangleright \left.\begin{array}{l} \textbf{let } \mathbf{x_b} = \mathbf{rd_p}\ 4 + \mathbf{x_a} \textbf{ in} \\ \textbf{let temp}{=}\mathbf{x_b} \textbf{ in skip} \end{array}\right\} \Omega_2$$

since $\cdots \triangleright \mathbf{x_a} \downarrow 4 : \mathbf{U}$ and $\mathbf{S} \sqcup \mathbf{U} = \mathbf{U}$

$$\xrightarrow{\texttt{read}(-4)^{\mathsf{U}}} \left.\begin{array}{l} -1 \mapsto 10 : \mathbf{S}; \\ -2 \mapsto 0 : \mathbf{S} \\ -3 \mapsto 10 : \mathbf{S} \\ -4 \mapsto 30 : \mathbf{S} \end{array}\right., \left.\begin{array}{l} \overline{B}\cdot \\ \mathbf{y} \mapsto 2 : \mathbf{S}; \\ \mathbf{size} \mapsto 1 : \mathbf{S} \\ \mathbf{x_a} \mapsto 2 : \mathbf{U} \\ \mathbf{x_b} \mapsto 30 : \mathbf{U} \end{array}\right. \triangleright \textbf{let temp}{=}\mathbf{x_b} \textbf{ in skip} \Big\} \Omega_3$$

$$\xrightarrow{\epsilon} \left.\begin{array}{l} -1 \mapsto 10 : \mathbf{S}; \\ -2 \mapsto 0 : \mathbf{S} \\ -3 \mapsto 10 : \mathbf{S} \\ -4 \mapsto 30 : \mathbf{S} \end{array}\right., \left.\begin{array}{l} \overline{B}\cdot \\ \mathbf{y} \mapsto 2 : \mathbf{S}; \\ \mathbf{size} \mapsto 1 : \mathbf{S} \\ \mathbf{x_a} \mapsto 4 : \mathbf{U} \\ \mathbf{x_b} \mapsto 30 : \mathbf{U} \\ \mathbf{temp} \mapsto 30 : \mathbf{U} \end{array}\right. \triangleright \textbf{skip} \Big\} \Omega_4$$

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

$$\Sigma = (4, \varnothing \cdot (\Omega, 4, \mathbf{S}))$$
$$\Sigma_1 = (4, \varnothing \cdot (\Omega', 3, \mathbf{S}) \cdot (\Omega_1, 3, \mathbf{U}))$$
$$\Sigma_2 = (4, \varnothing \cdot (\Omega', 3, \mathbf{S}) \cdot (\Omega_2, 2, \mathbf{U}))$$
$$\Sigma_3 = (4, \varnothing \cdot (\Omega', 3, \mathbf{S}) \cdot (\Omega_3, 1, \mathbf{U}))$$
$$\Sigma_4 = (4, \varnothing \cdot (\Omega', 3, \mathbf{S}) \cdot (\Omega_4, 0, \mathbf{U}))$$
$$\Sigma' = (4, \varnothing \cdot (\Omega', 3, \mathbf{S}))$$

$$\underbrace{(4, \varnothing \cdot (\Omega, 4, \mathbf{S}))}_{\Sigma} \overset{(\mathtt{if}(0))^{S}}{\rightsquigarrow} \underbrace{(4, \varnothing \cdot (\Omega', 3, \mathbf{S}) \cdot (\Omega_1, 3, \mathbf{U}))}_{\Sigma_1}$$ by Rule E-T-speculate-if

$$\underbrace{(4, \varnothing \cdot (\Omega', 3, \mathbf{S}) \cdot (\Omega_1, 3, \mathbf{U}))}_{\Sigma_1} \overset{(\mathtt{read}(\ell_A))^{S}}{\rightsquigarrow} \underbrace{(4, \varnothing \cdot (\Omega', 3, \mathbf{S}) \cdot (\Omega_2, 2, \mathbf{U}))}_{\Sigma_2}$$ by Rule E-T-speculate-action

$$\underbrace{(4, \varnothing \cdot (\Omega', 3, \mathbf{S}) \cdot (\Omega_2, 2, \mathbf{U}))}_{\Sigma_2} \overset{(\mathtt{read}(\ell_B))^{U}}{\rightsquigarrow} \underbrace{(4, \varnothing \cdot (\Omega', 3, \mathbf{S}) \cdot (\Omega_3, 1, \mathbf{U}))}_{\Sigma_3}$$ by Rule E-T-speculate-action

$$\underbrace{(4, \varnothing \cdot (\Omega', 3, \mathbf{S}) \cdot (\Omega_3, 1, \mathbf{U}))}_{\Sigma_3} \rightsquigarrow \underbrace{(4, \varnothing \cdot (\Omega', 3, \mathbf{S}) \cdot (\Omega_4, 0, \mathbf{U}))}_{\Sigma_4}$$ by Rule E-T-speculate-epsilon

$$\underbrace{(4, \varnothing \cdot (\Omega', 3, \mathbf{S}) \cdot (\Omega_4, 0, \mathbf{U}))}_{\Sigma_4} \overset{\mathtt{rlb}}{\rightsquigarrow} \underbrace{(4, \varnothing \cdot (\Omega', 3, \mathbf{S}))}_{\Sigma'}$$ by Rule E-T-speculate-rollback

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

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

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Sigma \Rightarrow \Sigma \qquad \Sigma \overset{(\mathtt{if}(0))^{S}}{\rightsquigarrow} \Sigma_1}{\Sigma \overset{(\mathtt{if}(\mathbf{0}))^{\mathbf{S}}}{\Longrightarrow} \Sigma_1} \qquad \Sigma_1 \overset{(\mathtt{read}(\ell_A))^{S}}{\rightsquigarrow} \Sigma_2}{\Sigma \overset{(\mathtt{if}(\mathbf{0}))^{\mathbf{S}} \cdot (\mathtt{read}(\ell_{\mathbf{A}}))^{\mathbf{S}}}{\Longrightarrow} \Sigma_2} \qquad \Sigma_2 \overset{(\mathtt{read}(\ell_B))^{U}}{\rightsquigarrow} \Sigma_3}{\Sigma \overset{(\mathtt{if}(\mathbf{0}))^{\mathbf{S}} \cdot (\mathtt{read}(\ell_{\mathbf{A}}))^{\mathbf{S}} \cdot (\mathtt{read}(\ell_{\mathbf{B}}))^{\mathbf{U}}}{\Longrightarrow} \Sigma_3} \qquad \Sigma_3 \rightsquigarrow \Sigma_4}{\Sigma \overset{(\mathtt{if}(\mathbf{0}))^{\mathbf{S}} \cdot (\mathtt{read}(\ell_{\mathbf{A}}))^{\mathbf{S}} \cdot (\mathtt{read}(\ell_{\mathbf{B}}))^{\mathbf{U}}}{\Longrightarrow} \Sigma_4} \qquad \Sigma_4 \overset{\mathtt{rlb}}{\rightsquigarrow} \Sigma'}{\Sigma \overset{(\mathtt{if}(\mathbf{0}))^{\mathbf{S}} \cdot (\mathtt{read}(\ell_{\mathbf{A}}))^{\mathbf{S}} \cdot (\mathtt{read}(\ell_{\mathbf{B}}))^{\mathbf{U}} \cdot \mathtt{rlb}}{\Longrightarrow} \Sigma'}$$

⊡

Example 9 (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 Uonly when speculating. We do not share that idea so we do not do it. ⊡

APPENDIX D
DYNAMIC 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 8 (Valid Heap)**.**

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

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

Definition 9 (Attacker)**.**

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

A whole program is speculatively safe (SS) if all its actions are safe.

Definition 10 (Dynamic Speculative Safety (SS))**.**

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

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

Definition 11 (Robust Dynamic Speculative Safety (RSS))**.**

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

Theorem 8 (All L programs are RSS)**.** (Proof 1)

$$\forall \mathsf{P} \in \mathsf{L}. \vdash \mathsf{P} : \text{RSS}$$

If we consider the code of Example 8, we see that the L execution only produces S actions. This is intuitive, since there is no speculation in L.

Still in Example 8, the T execution can produce U actions, precisely when a memory access is done speculatively and based on other memory-accessed data.

APPENDIX E
REVIEW: SPECULATIVE NON-INTERFERENCE

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

Definition 12 (Low-equivalence for heaps and programs)**.**

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

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

$$\begin{aligned} \overline{\lambda}\restriction_{nse} &= (\overline{\lambda}, 0)\restriction_{nse} \\ (\varnothing, n)\restriction_{nse} &= \varnothing \\ (\alpha^{\sigma} \cdot \overline{\lambda}, 0)\restriction_{nse} &= \alpha^{\sigma} \cdot (\overline{\lambda}, 0)\restriction_{nse} \\ (\delta^{\sigma} \cdot \overline{\lambda}, 0)\restriction_{nse} &= \delta^{\sigma} \cdot (\overline{\lambda}, 0)\restriction_{nse} && \text{where } \delta^{\sigma} \neq (\mathtt{if}(v))^{\sigma} \\ ((\mathtt{if}(v))^{\sigma} \cdot \overline{\lambda}, 0)\restriction_{nse} &= (\mathtt{if}(v))^{\sigma} \cdot (\overline{\lambda}, 1)\restriction_{nse} \\ (\alpha^{\sigma} \cdot \overline{\lambda}, n+1)\restriction_{nse} &= (\overline{\lambda}, 0)\restriction_{nse} \\ (\delta^{\sigma} \cdot \overline{\lambda}, n+1)\restriction_{nse} &= (\overline{\lambda}, n+1)\restriction_{nse} && \text{where } \delta^{\sigma} \neq (\mathtt{if}(v))^{\sigma} \\ ((\mathtt{if}(v))^{\sigma} \cdot \overline{\lambda}, n+1)\restriction_{nse} &= (\overline{\lambda}, n+2)\restriction_{nse} \\ ((\mathtt{rlb})^{\sigma} \cdot \overline{\lambda}, n+1)\restriction_{nse} &= (\overline{\lambda}, n)\restriction_{nse} \end{aligned}$$

A program 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 13 (Speculative Non-interference)**.**

$$\begin{aligned} \vdash P : \text{SNI} \stackrel{\text{def}}{=}\ & \forall P' =_{\mathrm{L}} P, \forall \overline{\lambda_1} \in Beh(\Omega_0(P)), \overline{\lambda_2} \in Beh(\Omega_0(P')) \\ & \text{if } \overline{\lambda_1}\restriction_{nse} = \overline{\lambda_1}\restriction_{nse} \text{ then } \overline{\lambda_1} = \overline{\lambda_2} \end{aligned}$$

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

Definition 14 (Robust Speculative Non-Interference)**.**

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

Theorem 9 (All L programs are RSNI)**.** (Proof 2)

$$\forall \mathsf{P} \in \mathsf{L}. \vdash \mathsf{P} : \text{RSNI}$$

APPENDIX F
SECURITY CRITERIA AND THEIR IMPLICATIONS

Here, we show the relationship between speculative safety and speculative non-interference. We prove these relationships only for programs in $\mathtt{T}$, since both criteria are trivially satisfied in $\mathtt{L}$ (this immediately follows from $\mathtt{L}$ not having any speculative behavior).

A. SS implies SNI

Theorem 10 (SS implies SNI)**.**

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

Proof. Let $\mathbf{P}$ be an arbitrary program in $\mathbf{T}$ such that $\vdash \mathbf{P} : \text{SS}$. Assume, for contradiction's sake, that $\vdash \mathbf{P} : \text{SNI}$ does not hold. That is, there is another program $\mathbf{P}'$ and traces $\overline{\lambda_1} \in \mathbf{Beh}(\Omega_0(\mathbf{P}))$, $\overline{\lambda_2} \in \mathbf{Beh}(\Omega_0(\mathbf{P}'))$ such that $\mathbf{P} =_{\mathtt{L}} \mathbf{P}'$, $\overline{\lambda_1}\restriction_{nse} = \overline{\lambda_2}\restriction_{nse}$, and $\overline{\lambda_1} \neq \overline{\lambda_2}$. By unrolling the $\mathbf{T}$-Terminal State rule, we have that $\mathbf{P} \rightsquigarrow \overline{\lambda_1}$ and $\mathbf{P}' \rightsquigarrow \overline{\lambda_2}$. By unrolling the rule E-$\mathbf{T}$-trace, we have that there are Σ, Σ', $\mathbf{n}$, $\mathbf{n}'$ such that $\vdash \Sigma : \bot$, $\vdash \Sigma' : \bot$, $(\mathbf{0}, \Omega_0(\mathbf{P})) \overset{\overline{\lambda_1}}{\Longrightarrow} (\mathbf{n}, \Sigma)$, and $(\mathbf{0}, \Omega_0(\mathbf{P}')) \overset{\overline{\lambda_2}}{\Longrightarrow} (\mathbf{n}', \Sigma')$. From $\mathbf{P} =_{\mathtt{L}} \mathbf{P}'$ and lemma 1, we get $(\mathbf{0}, \Omega_0(\mathbf{P})) \approx (\mathbf{0}, \Omega_0(\mathbf{P}'))$. There are two cases:

$\mathbf{n} = \mathbf{n}'$: Then, we have $\vdash \Sigma_0 : \mathit{safe}$ and from $\vdash \mathbf{P} : \text{SS}$, we get that $\vdash \overline{\lambda_1} : \mathit{safe}$. From $(\mathbf{0}, \Omega_0(\mathbf{P})) \approx (\mathbf{0}, \Omega_0(\mathbf{P}'))$, $(\mathbf{0}, \Omega_0(\mathbf{P})) \overset{\overline{\lambda_1}}{\Longrightarrow} (\mathbf{n}, \Sigma)$, $(\mathbf{0}, \Omega_0(\mathbf{P}')) \overset{\overline{\lambda_2}}{\Longrightarrow} (\mathbf{n}', \Sigma')$, and lemma 2, we get that $\overline{\lambda_1}\restriction_{nse} <> \overline{\lambda_2}\restriction_{nse} \vee \overline{\lambda_1} = \overline{\lambda_2}$. From this and $\overline{\lambda_1}\restriction_{nse} = \overline{\lambda_2}\restriction_{nse}$, we get $\overline{\lambda_1} = \overline{\lambda_2}$ leading to a contradiction.

$\mathbf{n} \neq \mathbf{n}'$: Let $\mathbf{m}$ be the maximum value such that $(\mathbf{0}, \Omega_0(\mathbf{P})) \overset{\lambda}{\Longrightarrow} (\mathbf{m}, \Sigma_1)$ and $(\mathbf{0}, \Omega_0(\mathbf{P}')) \overset{\lambda}{\Longrightarrow} (\mathbf{m}, \Sigma_1')$. Observe that (1) such an $\mathbf{m}$ always exists, and (2) λ is a prefix to both $\vdash \overline{\lambda_1} : \mathit{safe}$ and $\vdash \overline{\lambda_2} : \mathit{safe}$ From $\vdash \mathbf{P} : \text{SS}$ and λ being a prefix of $\vdash \overline{\lambda_1} : \mathit{safe}$, we have $\vdash \lambda : \mathit{safe}$. From $(\mathbf{0}, \Omega_0(\mathbf{P})) \approx (\mathbf{0}, \Omega_0(\mathbf{P}'))$, $(\mathbf{0}, \Omega_0(\mathbf{P})) \overset{\lambda}{\Longrightarrow} (\mathbf{m}, \Sigma_1)$, $(\mathbf{0}, \Omega_0(\mathbf{P}')) \overset{\lambda}{\Longrightarrow} (\mathbf{m}, \Sigma_1')$, $\vdash \lambda : \mathit{safe}$, and lemma 2, we have that $\Sigma_1 \approx \Sigma_1'$. There are three cases:

- $\neg\exists \alpha, \sigma, \Sigma_2.\ (\mathbf{0}, \Omega_0(\mathbf{P})) \overset{\lambda\cdot\alpha^\sigma}{\Longrightarrow} (\mathbf{m}+1, \Sigma_2)$**:** From this, we get that $\Sigma_1 = \Sigma$ and therefore $\vdash \Sigma : \bot$. From this and $\Sigma_1 \approx \Sigma_1'$, we get that $\vdash \Sigma' : \bot$ holds as well. From this, we get that $\mathbf{m} = \mathbf{n} = \mathbf{n}'$, leading to a contradiction.
- $\neg\exists \alpha', \sigma', \Sigma_2'.\ (\mathbf{0}, \Omega_0(\mathbf{P}')) \overset{\lambda\cdot\alpha'^{\sigma'}}{\Longrightarrow} (\mathbf{m}+1, \Sigma_2')$**:** The proof of this case is similar to the case $\neg\exists \alpha, \sigma, \Sigma_2.\ (\mathbf{0}, \Omega_0(\mathbf{P})) \overset{\lambda\cdot\alpha^\sigma}{\Longrightarrow} (\mathbf{m}+1, \Sigma_2)$.
- $\exists \alpha, \sigma, \Sigma_2, \alpha', \sigma', \Sigma_2'.\ (\mathbf{0}, \Omega_0(\mathbf{P})) \overset{\lambda\cdot\alpha^\sigma}{\Longrightarrow} (\mathbf{m}+1, \Sigma_2) \wedge (\mathbf{0}, \Omega_0(\mathbf{P}')) \overset{\lambda\cdot\alpha'^{\sigma'}}{\Longrightarrow} (\mathbf{m}+1, \Sigma_2') \wedge \alpha^\sigma \neq \alpha'^{\sigma'}$**:** Observe that $\lambda\cdot\alpha^\sigma$ is a prefix of $\overline{\lambda_1}$. Therefore, $\vdash \lambda\cdot\alpha^\sigma : \mathit{safe}$ follows from $\vdash \mathbf{P} : \text{SS}$. From $(\mathbf{0}, \Omega_0(\mathbf{P})) \approx (\mathbf{0}, \Omega_0(\mathbf{P}'))$, $(\mathbf{0}, \Omega_0(\mathbf{P})) \overset{\lambda\cdot\alpha^\sigma}{\Longrightarrow} (\mathbf{m}+1, \Sigma_2)$, $(\mathbf{0}, \Omega_0(\mathbf{P}')) \overset{\lambda\cdot\alpha'^{\sigma'}}{\Longrightarrow} (\mathbf{m}+1, \Sigma_2')$, $\vdash \lambda\cdot\alpha^\sigma : \mathit{safe}$, and lemma 2, we have $(\lambda\cdot\alpha^\sigma)\restriction_{nse} <> (\lambda\cdot\alpha'^{\sigma'})\restriction_{nse} \vee (\Sigma_2 \approx \Sigma_2' \wedge \lambda\cdot\alpha^\sigma = \lambda\cdot\alpha'^{\sigma'})$. There are two cases:
  - $(\lambda\cdot\alpha^\sigma)\restriction_{nse} <> (\lambda\cdot\alpha'^{\sigma'})\restriction_{nse}$**:** This contradicts $\overline{\lambda_1}\restriction_{nse} = \overline{\lambda_2}\restriction_{nse}$ given that $\lambda\cdot\alpha^\sigma$ is a prefix of $\overline{\lambda_1}$ and $\lambda\cdot\alpha'^{\sigma'}$ is a prefix of $\overline{\lambda_2}$.
  - $(\Sigma_2 \approx \Sigma_2' \wedge \lambda\cdot\alpha^\sigma = \lambda\cdot\alpha'^{\sigma'})$**:** 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. □

Theorem 11 (SS implies SNI)**.**

$$\forall \mathbf{P} \in \mathtt{L}.\ \text{if}\ \vdash \mathbf{P} : \text{SS},\ \text{then}\ \vdash \mathbf{P} : \text{SNI}$$

Proof. Trivial adaptation of Theorem 10 (SS implies SNI). □

Lemma 1 (Low-equivalent programs have low-equivalent initial states)**.**

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

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

$$\begin{aligned}
\mathbf{H_0} &= \mathbf{H_1} \cup \mathbf{H} \cup \mathbf{H_2} \\
\mathbf{H_1} &= \{\mathbf{n} \mapsto \mathbf{0} : \mathbf{S} \mid \mathbf{n} \in \mathbb{N} \setminus \mathtt{dom}(\mathbf{H})\} \\
\mathbf{H_2} &= \{-\mathbf{n} \mapsto \mathbf{0} : \mathbf{U} \mid \mathbf{n} \in \mathbb{N}, -\mathbf{n} \notin \mathtt{dom}(\mathbf{H})\} \\
\Omega_0(\mathbf{P}) &\overset{\text{def}}{=} \mathbf{w}, (\overline{\mathbf{F}}; \overline{\mathbf{I}}; \mathbf{H_0}; \varnothing \cdot \mathbf{x} \mapsto \mathbf{0} \triangleright \mathbf{call\ main\ x; skip}, \bot, \mathbf{S}) \\
\mathbf{H_0'} &= \mathbf{H_1'} \cup \mathbf{H'} \cup \mathbf{H_2'} \\
\mathbf{H_1'} &= \{\mathbf{n} \mapsto \mathbf{0} : \mathbf{S} \mid \mathbf{n} \in \mathbb{N} \setminus \mathtt{dom}(\mathbf{H'})\}
\end{aligned}$$

$$\mathbf{H_2'} = \{-\mathbf{n} \mapsto \mathbf{0} : \mathbf{U} \mid \mathbf{n} \in \mathbb{N}, -\mathbf{n} \notin \mathtt{dom}\,(\mathbf{H'})\}$$

$$\mathbf{\Omega_0}\,(\mathbf{P'}) \overset{\text{def}}{=} \mathbf{w}, (\overline{\mathbf{F}}; \overline{\mathbf{I}}; \mathbf{H_0'}; \varnothing \cdot \mathbf{x} \mapsto \mathbf{0} \triangleright \mathbf{call\ main\ x; skip}, \bot, \mathbf{S})$$

Moreover, from $\mathbf{P} =_{\mathrm{L}} \mathbf{P'}$, we get $\mathbf{H} \approx \mathbf{H'}$. Therefore, $\mathbf{\Omega_0}\,(\mathbf{P}) \approx \mathbf{\Omega_0}\,(\mathbf{P'})$ immediately follows. □

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

Definition 15 (Safe-equivalence for heaps and program states)**.**

$$\vdash w, ss_0 \cdot (\Omega, m, \sigma) : unsafe \overset{\text{def}}{=} \sigma = U$$

$$\vdash w, (\Omega, m, \sigma) : safe \overset{\text{def}}{=} \sigma = S$$

$$\vdash \epsilon : safe \overset{\text{def}}{=} true$$

$$\vdash \alpha^\sigma : safe \overset{\text{def}}{=} \sigma = S$$

$$\vdash \overline{\lambda^\sigma} \cdot \alpha^\sigma : safe \overset{\text{def}}{=} \vdash \overline{\lambda^\sigma} : safe \text{ and } \vdash \alpha^\sigma : safe$$

$$\vdash H(n) : def \overset{\text{def}}{=} \exists v, \sigma.\ H(n) = v : \sigma$$

$$\vdash B(x) : def \overset{\text{def}}{=} \exists v, \sigma.\ B(x) = v : \sigma$$

$$v : \sigma \approx v' : \sigma' \overset{\text{def}}{=} \sigma = \sigma' \text{ and if } \sigma = S \text{ then } v = v'$$

$$H \approx H' \overset{\text{def}}{=} \forall n.\ \vdash H(n) : def \text{ iff } \vdash H'(n) : def \text{ and if } \vdash H(n) : def \text{ then } H(n) \approx H'(n)$$

$$B \approx B' \overset{\text{def}}{=} \forall x.\ \vdash B(x) : def \text{ iff } \vdash B'(x) : def \text{ and if } \vdash B(x) : def \text{ then } B(x) \approx B'(x)$$

$$\overline{B} \cdot B \approx \overline{B'} \cdot B' \overset{\text{def}}{=} \overline{B} \approx \overline{B'} \text{ and } B \approx B'$$

$$\Omega \approx \Omega' \overset{\text{def}}{=} \Omega \equiv C, H, \overline{B} \triangleright s \text{ and } \Omega' \equiv C, H', \overline{B'} \triangleright s \text{ and } H \approx H' \text{ and } \overline{B} \approx \overline{B'}$$

$$\varnothing \approx \varnothing$$

$$ss \approx ss' \overset{\text{def}}{=} ss \equiv ss_0 \cdot (\Omega, m, \sigma) \text{ and } ss' \equiv ss_0' \cdot (\Omega', m, \sigma) \text{ and } ss_0 \approx ss_0' \text{ and } \Omega \approx \Omega'$$

$$\Sigma \approx \Sigma' \overset{\text{def}}{=} \Sigma \equiv (w, ss) \text{ and } \Sigma' \equiv (w, ss') \text{ and } ss \approx ss'$$

1) Transitive-closure semantics $\Rightarrow$ *:*

Definition 16.

$$\alpha^\sigma <> \alpha'^{\sigma'} \text{ if } \alpha \neq \alpha' \vee \sigma \neq \sigma'$$

$$\alpha^\sigma \cdot \lambda <> \alpha'^{\sigma'} \cdot \lambda' \text{ if } \alpha^\sigma <> \alpha'^{\sigma'} \vee \lambda <> \lambda'$$

Lemma 2 (Steps of $\Rightarrow$ preserve low-equivalence or produce distinct non-speculative projections)**.**

$$\forall \mathbf{P}, \mathbf{P'}, \mathbf{\Sigma_0}, \mathbf{\Sigma_0'}, \mathbf{\Sigma_1}, \mathbf{\Sigma_1'}, \lambda, \lambda', \mathbf{n}.\ \text{if } \mathbf{\Sigma_0} = \mathbf{\Omega_0}\,(\mathbf{P}), \mathbf{\Sigma_0'} = \mathbf{\Omega_0}\,(\mathbf{P'}),$$
$$(\mathbf{0}, \mathbf{\Sigma_0}) \overset{\lambda}{\Longrightarrow} (\mathbf{n}, \mathbf{\Sigma_1}), (\mathbf{0}, \mathbf{\Sigma_0'}) \overset{\lambda'}{\Longrightarrow} (\mathbf{n}, \mathbf{\Sigma_1'}),$$
$$\mathbf{\Sigma_0} \approx \mathbf{\Sigma_0'}, \vdash \mathbf{\Sigma_0} : safe, \vdash \lambda : safe,$$
$$\text{then } \lambda\restriction_{nse} <> \lambda'\restriction_{nse} \vee (\mathbf{\Sigma_1} \approx \mathbf{\Sigma_1'} \wedge \lambda = \lambda')$$

Proof. Let $\mathbf{P}$, $\mathbf{P'}$, $\mathbf{\Sigma_0}$, $\mathbf{\Sigma_0'}$, $\mathbf{\Sigma_1}$, $\mathbf{\Sigma_1'}$, λ, λ', $\mathbf{n}$ be such that $\mathbf{\Sigma_0} = \mathbf{\Omega_0}\,(\mathbf{P})$, $\mathbf{\Sigma_0'} = \mathbf{\Omega_0}\,(\mathbf{P'})$, $(\mathbf{0}, \mathbf{\Sigma_0}) \overset{\lambda}{\Longrightarrow} (\mathbf{n}, \mathbf{\Sigma_1})$, $(\mathbf{0}, \mathbf{\Sigma_0'}) \overset{\lambda'}{\Longrightarrow} (\mathbf{n}, \mathbf{\Sigma_1'})$, $\mathbf{\Sigma_0} \approx \mathbf{\Sigma_0'}$, $\vdash \mathbf{\Sigma_0} : safe$, and $\vdash \lambda : safe$. We prove the lemma by induction on n:

Base case: For the base case, we consider $n = 0$. Then, both $(\mathbf{0}, \mathbf{\Sigma_0}) \overset{\lambda}{\Longrightarrow} (\mathbf{n}, \mathbf{\Sigma_1})$ and $(\mathbf{0}, \mathbf{\Sigma_0'}) \overset{\lambda'}{\Longrightarrow} (\mathbf{n}, \mathbf{\Sigma_1'})$ have been derived using the E-**T**-init rule. Therefore, $\lambda = \epsilon$, $\mathbf{\Sigma_1} = \mathbf{\Sigma_0}$, $\lambda' = \epsilon$, and $\mathbf{\Sigma_1'} = \mathbf{\Sigma_0'}$. As a result, $\lambda = \lambda'$ follows from $\lambda = \epsilon$ and $\lambda' = \epsilon$, whereas $\mathbf{\Sigma_1} \approx \mathbf{\Sigma_1'}$ follows from $\mathbf{\Sigma_1} = \mathbf{\Sigma_0}$, $\mathbf{\Sigma_1'} = \mathbf{\Sigma_0'}$, and $\mathbf{\Sigma_0} \approx \mathbf{\Sigma_0'}$. Therefore, $\lambda\restriction_{nse} <> \lambda'\restriction_{nse} \vee (\mathbf{\Sigma_1} \approx \mathbf{\Sigma_1'} \wedge \lambda = \lambda')$ holds for the base case.

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

E-T-silent: Then, $(\mathbf{0}, \mathbf{\Sigma_0}) \overset{\lambda}{\Longrightarrow} (\mathbf{n}, \mathbf{\Sigma_1})$ has been derived using the E-**T**-silent rule. Therefore, we have $(\mathbf{0}, \mathbf{\Sigma_0}) \overset{\overline{\lambda_1^\sigma}}{\Longrightarrow} (\mathbf{n-1}, \mathbf{\Sigma_2})$, $\mathbf{\Sigma_2} \overset{\epsilon}{\leftrightsquigarrow} \mathbf{\Sigma_1}$, and $\lambda = \overline{\lambda_1^\sigma} \cdot \epsilon$.

Since $\mathbf{n} > \mathbf{1}$, $(\mathbf{0}, \Sigma_0') \xRightarrow{\lambda'} (\mathbf{n}, \Sigma_1')$ has been derived using the E-T-silent or E-T-single rules. Therefore, we have $(\mathbf{0}, \Sigma_0') \xRightarrow{\overline{\lambda_1'^{\sigma'}}} (\mathbf{n-1}, \Sigma_2')$ and $\Sigma_2' \overset{\lambda_1'}{\rightsquigarrow} \Sigma_1'$. From the induction hypothesis, there are two cases:

$\overline{\lambda_1^{\sigma}}\restriction_{nse} <> \overline{\lambda_1'^{\sigma'}}\restriction_{nse}$**:** Observe that $\lambda\restriction_{nse} = \overline{\lambda_1^{\sigma}}\restriction_{nse}$ and $\lambda'\restriction_{nse}$ is either $\overline{\lambda_1'^{\sigma'}}\restriction_{nse}$ or $\overline{\lambda_1'^{\sigma'}}\restriction_{nse} \cdot \lambda_1'$. From this and $\overline{\lambda_1^{\sigma}}\restriction_{nse} <> \overline{\lambda_1'^{\sigma'}}\restriction_{nse}$, we get $\lambda\restriction_{nse} <> \lambda'\restriction_{nse}$.

$\Sigma_2 \approx \Sigma_2' \wedge \overline{\lambda_1^{\sigma}} = \overline{\lambda_1'^{\sigma'}}$**:** From $\Sigma_2 \approx \Sigma_2'$, $\Sigma_2 \overset{\epsilon}{\rightsquigarrow} \Sigma_1$, lemma 4, we get $\Sigma_2' \overset{\epsilon}{\rightsquigarrow} \Sigma_1''$ 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 $\lambda = \overline{\lambda_1^{\sigma}}$, $\lambda' = \overline{\lambda_1'^{\sigma'}} \cdot \lambda_1'$, $\overline{\lambda_1^{\sigma}} = \overline{\lambda_1'^{\sigma'}}$, and $\lambda_1' = \epsilon$. Moreover, $\Sigma_1 \approx \Sigma_1'$ follows from $\Sigma_1 \approx \Sigma_1''$ and $\Sigma_1' = \Sigma_1''$.

Therefore, $\lambda\restriction_{nse} <> \lambda'\restriction_{nse} \vee (\Sigma_1 \approx \Sigma_1' \wedge \lambda = \lambda')$ holds for this case.

E-T-single: Then, $(\mathbf{0}, \Sigma_0) \xRightarrow{\lambda} (\mathbf{n}, \Sigma_1)$ has been derived using the E-T-single rule. Therefore, we have $(\mathbf{0}, \Sigma_0) \xRightarrow{\overline{\lambda_1^{\sigma_1}}} (\mathbf{n-1}, \Sigma_2)$, $\Sigma_2 \overset{\alpha^{\sigma}}{\rightsquigarrow} \Sigma_1$, $\Sigma_2 = (\mathbf{w}, \overline{(\Omega, \mathbf{m}, \sigma)} \cdot (\overline{\mathbf{F}}, \overline{\mathbf{I}}, \mathbf{H_2}, \mathbf{B_2} \triangleright (\mathbf{s})_{\overline{\mathbf{f}}.\mathbf{f}}, \mathbf{n}, \sigma'))$, $\Sigma_1 = (\mathbf{w}, \overline{(\Omega, \mathbf{m}, \sigma)} \cdot (\overline{\mathbf{F}}, \overline{\mathbf{I}}, \mathbf{H_1}, \mathbf{B_1} \triangleright (\mathbf{s}')_{\overline{\mathbf{f}'}.\mathbf{f}'}, \mathbf{n}', \sigma''))$, and $\lambda = \overline{\lambda_1^{\sigma}}$ if $\mathbf{f} == \mathbf{f}' \wedge \mathbf{f} \in \mathbf{I}$ and $\lambda = \overline{\lambda_1^{\sigma}} \cdot \alpha^{\sigma}$ otherwise.

Since $\mathbf{n} > \mathbf{1}$, $(\mathbf{0}, \Sigma_0') \xRightarrow{\lambda'} (\mathbf{n}, \Sigma_1')$ has been derived using the E-T-silent or E-T-single rules. Therefore, we have $(\mathbf{0}, \Sigma_0') \xRightarrow{\overline{\lambda_1'^{\sigma'}}} (\mathbf{n-1}, \Sigma_2')$ and $\Sigma_2' \overset{\lambda_1'}{\rightsquigarrow} \Sigma_1'$. From the induction hypothesis, there are two cases:

$\overline{\lambda_1^{\sigma}}\restriction_{nse} <> \overline{\lambda_1'^{\sigma'}}\restriction_{nse}$**:** Observe that $\lambda\restriction_{nse}$ is either $\overline{\lambda_1^{\sigma}}\restriction_{nse}$ or $\overline{\lambda_1^{\sigma}}\restriction_{nse} \cdot \alpha^{\sigma}$ and $\lambda'\restriction_{nse}$ is either $\overline{\lambda_1'^{\sigma'}}\restriction_{nse}$ or $\overline{\lambda_1'^{\sigma'}}\restriction_{nse} \cdot \lambda_1'$. From this and $\overline{\lambda_1^{\sigma}}\restriction_{nse} <> \overline{\lambda_1'^{\sigma'}}\restriction_{nse}$, we get $\lambda\restriction_{nse} <> \lambda'\restriction_{nse}$.

$\Sigma_2 \approx \Sigma_2' \wedge \overline{\lambda_1^{\sigma}} = \overline{\lambda_1'^{\sigma'}}$**:** From $\Sigma_2 \approx \Sigma_2'$ and the determinism of $\rightsquigarrow$, then $(\mathbf{0}, \Sigma_0') \xRightarrow{\lambda'} (\mathbf{n}, \Sigma_1')$ must have been derived using the E-T-single rule (since $\Sigma_2 \overset{\alpha^{\sigma}}{\rightsquigarrow} \Sigma_1$ has produced an observation that is not ϵ). From this, $\Sigma_2 \approx \Sigma_2'$, and the determinism of $\rightsquigarrow$, we get that $\Sigma_2' \overset{\alpha'^{\sigma'}}{\rightsquigarrow} \Sigma_1'$, $\Sigma_2' = (\mathbf{w}, \overline{(\Omega', \mathbf{m}, \sigma)} \cdot (\overline{\mathbf{F}}, \overline{\mathbf{I}}, \mathbf{H_2'}, \mathbf{B_2'} \triangleright (\mathbf{s})_{\overline{\mathbf{f}}.\mathbf{f}}, \mathbf{n}, \sigma'))$, $\Sigma_1' = (\mathbf{w}, \overline{(\Omega', \mathbf{m}, \sigma)} \cdot (\overline{\mathbf{F}}, \overline{\mathbf{I}}, \mathbf{H_1'}, \mathbf{B_1'} \triangleright (\mathbf{s}')_{\overline{\mathbf{f}'}.\mathbf{f}'}, \mathbf{n}', \sigma''))$, and $\lambda' = \overline{\lambda_1'^{\sigma'}}$ if $\mathbf{f} == \mathbf{f}' \wedge \mathbf{f} \in \mathbf{I}$ and $\lambda' = \overline{\lambda_1'^{\sigma'}} \cdot \alpha'^{\sigma'}$ otherwise. There are two cases:

$\mathbf{f} == \mathbf{f}' \wedge \mathbf{f} \in \mathbf{I}$**:** Hence, $\lambda = \overline{\lambda_1^{\sigma}}$ and $\lambda' = \overline{\lambda_1'^{\sigma'}}$. Therefore, $\lambda = \lambda'$ follows from $\lambda = \overline{\lambda_1^{\sigma}}$, $\lambda' = \overline{\lambda_1'^{\sigma'}}$, and $\overline{\lambda_1^{\sigma}} = \overline{\lambda_1'^{\sigma'}}$.

From $\mathbf{f} \in \mathbf{I}$, $(\mathbf{0}, \Sigma_0) \xRightarrow{\overline{\lambda_1^{\sigma}}} (\mathbf{n-1}, \Sigma_2)$, $\Sigma_2 = (\mathbf{w}, \overline{(\Omega, \mathbf{m}, \sigma)} \cdot (\overline{\mathbf{F}}, \overline{\mathbf{I}}, \mathbf{H_2}, \mathbf{B_2} \triangleright (\mathbf{s})_{\overline{\mathbf{f}}.\mathbf{f}}, \mathbf{n}, \sigma'))$, and lemma 13, we get that for all $\mathbf{x}$, $\mathbf{v}$, σ, if $\mathbf{B_2}(\mathbf{x}) = \mathbf{v} : \sigma$, then $\sigma = \mathbf{S}$. From this, $\Sigma_2' = (\mathbf{w}, \overline{(\Omega', \mathbf{m}, \sigma)} \cdot (\overline{\mathbf{F}}, \overline{\mathbf{I}}, \mathbf{H_2'}, \mathbf{B_2'} \triangleright (\mathbf{s})_{\overline{\mathbf{f}}.\mathbf{f}}, \mathbf{n}, \sigma'))$, and $\Sigma_2 \approx \Sigma_2'$, we get that $\mathbf{B_2} = \mathbf{B_2'}$. From this, $\Sigma_2 \overset{\alpha^{\sigma}}{\rightsquigarrow} \Sigma_1$, and the determinism of $\rightsquigarrow$, we get that $\Sigma_2' \overset{\alpha^{\sigma}}{\rightsquigarrow} \Sigma_1'$. Then, $\Sigma_1 \approx \Sigma_1'$ follows from $\Sigma_2 \overset{\alpha^{\sigma}}{\rightsquigarrow} \Sigma_1$, $\Sigma_2' \overset{\alpha^{\sigma}}{\rightsquigarrow} \Sigma_1'$, $\Sigma_2 \approx \Sigma_2'$, and lemma 5.

$\neg(\mathbf{f} == \mathbf{f}' \wedge \mathbf{f} \in \mathbf{I})$**:** Hence, $\lambda = \overline{\lambda_1^{\sigma}} \cdot \alpha^{\sigma}$ and $\lambda' = \overline{\lambda_1'^{\sigma'}} \cdot \alpha'^{\sigma'}$. There are two cases:

$\vdash \Sigma_2 : \mathit{unsafe}$**:** Then, from $\vdash \lambda : \mathit{safe}$, we have $\vdash \alpha^{\sigma} : \mathit{safe}$. From $\vdash \Sigma_2 : \mathit{unsafe}$, $\Sigma_2 \overset{\alpha^{\sigma}}{\rightsquigarrow} \Sigma_1$, $\vdash \alpha^{\sigma} : \mathit{safe}$, $\Sigma_2 \approx \Sigma_2'$, $\Sigma_2' \overset{\alpha'^{\sigma}}{\rightsquigarrow} \Sigma_1'$, the determinism of $\rightsquigarrow$, and lemma 3, we get $\Sigma_1 \approx \Sigma_1'$, $\alpha^{\sigma} = \alpha'^{\sigma'}$, and $\lambda = \lambda'$ (from $\alpha^{\sigma} = \alpha'^{\sigma'}$ and $\overline{\lambda_1^{\sigma}} = \overline{\lambda_1'^{\sigma'}}$).

$\vdash \Sigma_2 : \mathit{safe}$**:** From this and $\Sigma_2 \approx \Sigma_2'$, we have $\vdash \Sigma_2' : \mathit{safe}$. From this, $\Sigma_0 = \Omega_0(\mathbf{P})$, $\Sigma_0' = \Omega_0(\mathbf{P}')$, and lemma 12, we get $\lambda\restriction_{nse} = \overline{\lambda_1^{\sigma}}\restriction_{nse} \cdot \alpha^{\sigma}$ and $\lambda'\restriction_{nse} = \overline{\lambda_1'^{\sigma'}}\restriction_{nse} \cdot \alpha'^{\sigma'}$. There are two cases:

$\alpha^{\sigma} = \alpha'^{\sigma'}$**:** Then, $\lambda = \lambda'$ follows from $\alpha^{\sigma} = \alpha'^{\sigma'}$ and $\overline{\lambda_1^{\sigma}} = \overline{\lambda_1'^{\sigma'}}$. Moreover, $\Sigma_1 \approx \Sigma_1'$ follows from $\Sigma_2 \approx \Sigma_2'$, $\Sigma_2 \overset{\alpha^{\sigma}}{\rightsquigarrow} \Sigma_1$, $\Sigma_2' \overset{\alpha'^{\sigma'}}{\rightsquigarrow} \Sigma_1'$, $\alpha^{\sigma} = \alpha'^{\sigma'}$, and lemma 5.

$\alpha^{\sigma} \neq \alpha'^{\sigma'}$**:** From this, $\lambda\restriction_{nse} = \overline{\lambda_1^{\sigma}}\restriction_{nse} \cdot \alpha^{\sigma}$, $\lambda'\restriction_{nse} = \overline{\lambda_1'^{\sigma'}}\restriction_{nse} \cdot \alpha'^{\sigma'}$, and $\overline{\lambda_1^{\sigma}}\restriction_{nse} = \overline{\lambda_1'^{\sigma'}}\restriction_{nse}$, we get that $\lambda\restriction_{nse} <> \lambda'\restriction_{nse}$.

Therefore, $\lambda\restriction_{nse} <> \lambda'\restriction_{nse} \vee (\Sigma_1 \approx \Sigma_1' \wedge \lambda = \lambda')$ holds for this case.

Therefore, $\lambda\restriction_{nse} <> \lambda'\restriction_{nse} \vee (\Sigma_1 \approx \Sigma_1' \wedge \lambda = \lambda')$ holds for the induction step.

This concludes the proof. □

2) Speculative semantics $\rightsquigarrow$:

Lemma 3 (Steps of $\rightsquigarrow$ with safe observations preserve low-equivalence for unsafe configurations)**.**

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

Proof. Let Σ_0, Σ_0', Σ_1, and λ be such that $\Sigma_0 \overset{\lambda}{\rightsquigarrow} \Sigma_1$, $\Sigma_0 \approx \Sigma_0'$, and $\vdash \lambda : \mathit{safe}$. We proceed by case distinction on the rule used to derive $\Sigma_0 \overset{\lambda}{\rightsquigarrow} \Sigma_1$:

E-T-speculate-epsilon: Then, $\Sigma_0 \stackrel{\text{def}}{=} \mathtt{w}, \overline{(\Omega, \omega, \sigma)} \cdot (\Omega_0, \mathtt{n}+1, \sigma)$, $\Omega_0 \xrightarrow{\epsilon} \Omega_1$, $\Omega_0 \equiv \mathtt{C}, \mathtt{H}, \overline{\mathtt{B}} \triangleright \mathtt{s}; \mathtt{s}'$, $\mathtt{s} \not\equiv \mathtt{s}''; \mathtt{s}'''$ and $\mathtt{s} \not\equiv \mathtt{lfence}$, and $\lambda = \epsilon$, and $\Sigma_1 \stackrel{\text{def}}{=} \mathtt{w}, \overline{(\Omega, \omega, \sigma)} \cdot (\Omega_1, \mathtt{n}, \sigma)$. From $\Sigma_0 \approx \Sigma_0'$, we get that $\Sigma_0' \stackrel{\text{def}}{=} \mathtt{w}, \overline{(\Omega', \omega, \sigma)} \cdot (\Omega_0', \mathtt{n}+1, \sigma)$ and $\Omega_0 \approx \Omega_0'$. From $\Omega_0 \xrightarrow{\epsilon} \Omega_1$, $\Omega_0 \approx \Omega_0'$, $\vdash \epsilon : \mathit{safe}$, and lemma 7, 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 $\Sigma_0' \overset{\lambda'}{\rightsquigarrow} \Sigma_1'$ where $\lambda' = \epsilon$ and $\Sigma_1' \stackrel{\text{def}}{=} \mathtt{w}, \overline{(\Omega, \omega, \sigma)} \cdot (\Omega_1', \mathtt{n}, \sigma)$. 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, $\Sigma_0 \stackrel{\text{def}}{=} \mathtt{w}, \overline{(\Omega, \omega, \sigma)} \cdot (\Omega_0, \mathtt{n}+1, \sigma)$, $\Omega_0 \xrightarrow{\lambda_0^{\sigma_0}} \Omega_1$, $\Omega_0 \equiv \mathtt{C}, \mathtt{H}, \overline{\mathtt{B}} \triangleright \mathtt{s}; \mathtt{s}'$, $\mathtt{s} \not\equiv \mathtt{s}''; \mathtt{s}'''$ and $\mathtt{s} \not\equiv \mathtt{lfence}$, and $\lambda = \boldsymbol{\lambda_0}^{\sigma_0 \sqcap \sigma}$, and $\Sigma_1 \stackrel{\text{def}}{=} \mathtt{w}, \overline{(\Omega, \omega, \sigma)} \cdot (\Omega_1, \mathtt{n}, \sigma)$. From $\Sigma_0 \approx \Sigma_0'$, we get that $\Sigma_0' \stackrel{\text{def}}{=} \mathtt{w}, \overline{(\Omega', \omega, \sigma)} \cdot (\Omega_0', \mathtt{n}+1, \sigma)$ and $\Omega_0 \approx \Omega_0'$. From $\vdash \Sigma_0 : \mathit{unsafe}$ and $\vdash \lambda : \mathit{safe}$, we get that $\vdash \boldsymbol{\lambda}^{\sigma_0} : \mathit{safe}$. From this, $\Omega_0 \approx \Omega_0'$, and lemma 7, 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 $\Sigma_0' \overset{\lambda'}{\rightsquigarrow} \Sigma_1'$ where $\lambda' = \boldsymbol{\lambda_0}^{\sigma_0 \sqcap \sigma}$ and $\Sigma_1' \stackrel{\text{def}}{=} \mathtt{w}, \overline{(\Omega, \omega, \sigma)} \cdot (\Omega_1', \mathtt{n}, \sigma)$. Hence, $\lambda = \lambda'$ follows from $\lambda' = \boldsymbol{\lambda_0}^{\sigma_0 \sqcap \sigma}$ and $\lambda' = \boldsymbol{\lambda_0}^{\sigma_0 \sqcap \sigma}$, whereas $\Sigma_1 \approx \Sigma_1'$ follows from $\Sigma_0 \approx \Sigma_0'$ and $\Omega_1 \approx \Omega_1'$.

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

From $\vdash \Sigma_0 : \mathit{unsafe}$ and $\vdash \lambda : \mathit{safe}$, we get $\vdash \alpha^{\sigma_0} : \mathit{safe}$ and $\sigma_0 = \mathtt{S}$. From this, $\Sigma_0 \approx \Sigma_0'$, and lemma 7, we get that $\Omega_0' \xrightarrow{\alpha^{\sigma_0}} \Omega_1'$ and $\Omega_1 \approx \Omega_1'$.

From $\Omega_0 \approx \Omega_0'$ and $\Omega_0 \stackrel{\text{def}}{=} \mathtt{C}, \mathtt{H}, \overline{\mathtt{B}} \cdot \mathtt{B} \triangleright (\mathtt{if\ e\ then\ s''\ else\ s'''}; \mathtt{s}')_{\overline{\mathtt{f}} \cdot \mathtt{f}}$, we have $\Omega_0 \stackrel{\text{def}}{=} \mathtt{C}, \mathtt{H}', \overline{\mathtt{B}'} \cdot \mathtt{B}' \triangleright (\mathtt{if\ e\ then\ s''\ else\ s'''}; \mathtt{s}')_{\overline{\mathtt{f}} \cdot \mathtt{f}}$ where $\mathtt{H} \approx \mathtt{H}'$, $\mathtt{B} \approx \mathtt{B}'$, and $\overline{\mathtt{B}} \approx \overline{\mathtt{B}'}$. From $\mathtt{B} \approx \mathtt{B}'$, $\mathtt{B} \triangleright \mathtt{e} \downarrow \mathtt{v} : \sigma_0$, $\sigma_0 = \mathtt{S}$, and lemma 11, we have that $\mathtt{B}' \triangleright \mathtt{e} \downarrow \mathtt{v} : \sigma_0$.

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

E-T-speculate-rollback: Then, $\Sigma_0 \stackrel{\text{def}}{=} \mathtt{w}, \overline{(\Omega, \omega, \sigma)} \cdot (\Omega, \mathtt{0}, \sigma)$, $\lambda = \mathtt{rlb}$, and $\Sigma_1 \stackrel{\text{def}}{=} \mathtt{w}, \overline{(\Omega, \omega, \sigma)}$. From $\Sigma_0 \approx \Sigma_0'$, we get that $\Sigma_0' \stackrel{\text{def}}{=} \mathtt{w}, \overline{(\Omega', \omega, \sigma)} \cdot (\Omega_0', \mathtt{0}, \sigma)$ 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 $\Sigma_0' \overset{\lambda'}{\rightsquigarrow} \Sigma_1'$ where $\lambda' = \mathtt{rlb}$ and $\Sigma_1' \stackrel{\text{def}}{=} \mathtt{w}, \overline{(\Omega', \omega, \sigma)}$. Hence, $\lambda = \lambda'$ follows from $\lambda' = \mathtt{rlb}$ and $\lambda' = \mathtt{rlb}$, whereas $\Sigma_1 \approx \Sigma_1'$ follows from $\Sigma_0 \approx \Sigma_0'$.

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

This concludes our proof. □

Lemma 4 (Steps of $\rightsquigarrow$ without observations preserve low-equivalence)**.**

$$\forall \Sigma_0, \Sigma_0', \Sigma_1, \lambda.\ \text{if } \Sigma_0 \overset{\epsilon}{\rightsquigarrow} \Sigma_1, \Sigma_0 \approx \Sigma_0'$$
$$\text{then } \exists \Sigma_1'.\ \Sigma_0' \overset{\epsilon}{\rightsquigarrow} \Sigma_1',\ \Sigma_1 \approx \Sigma_1'$$

Proof. Proof is similart to lemma 3 (only rules E-T-speculate-epsilon and E-T-speculate-lfence) but using lemma 8 instead of lemma 7. □

Lemma 5 (Steps of $\rightsquigarrow$ with same observations preserve low-equivalence)**.**

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

Proof. Let Σ_0, Σ_0', Σ_1, Σ_1', λ, λ' be such that $\Sigma_0 \overset{\lambda}{\rightsquigarrow} \Sigma_1$, $\Sigma_0' \overset{\lambda'}{\rightsquigarrow} \Sigma_1'$, $\Sigma_0 \approx \Sigma_0'$, and $\lambda = \lambda'$. We proceed by case distinction on the rule used to derive $\Sigma_0 \overset{\lambda}{\rightsquigarrow} \Sigma_1$:

E-T-speculate-epsilon: Then, $\Sigma_0 \stackrel{\text{def}}{=} \mathtt{w}, \overline{(\Omega, \omega, \sigma)} \cdot (\Omega_0, \mathtt{n}+1, \sigma)$, $\Omega_0 \xrightarrow{\epsilon} \Omega_1$, $\Omega_0 \equiv \mathtt{C}, \mathtt{H}, \overline{\mathtt{B}} \triangleright \mathtt{s}; \mathtt{s}'$, $\mathtt{s} \not\equiv \mathtt{s}''; \mathtt{s}'''$ and $\mathtt{s} \not\equiv \mathtt{lfence}$, and $\lambda = \epsilon$, and $\Sigma_1 \stackrel{\text{def}}{=} \mathtt{w}, \overline{(\Omega, \omega, \sigma)} \cdot (\Omega_1, \mathtt{n}, \sigma)$. From $\Sigma_0 \approx \Sigma_0'$, we get that $\Sigma_0' \stackrel{\text{def}}{=} \mathtt{w}, \overline{(\Omega', \omega, \sigma)} \cdot (\Omega_0', \mathtt{n}+1, \sigma)$ and $\Omega_0 \approx \Omega_0'$. From $\Omega_0 \approx \Omega_0'$ and $\Omega_0 \equiv \mathtt{C}, \mathtt{H}, \overline{\mathtt{B}} \triangleright \mathtt{s}; \mathtt{s}'$, we get that $\Omega_0' \equiv \mathtt{C}, \mathtt{H}, \overline{\mathtt{B}} \triangleright \mathtt{s}; \mathtt{s}'$. Since $\lambda = \lambda'$ and

$\lambda = \epsilon$, $\Sigma_0' \xrightarrow{\lambda'} \Sigma_1'$ has also been derived using the E-T-speculate-epsilon rule. Thus, $\Sigma_1' \stackrel{\text{def}}{=} \mathtt{w}, \overline{(\Omega', \omega, \sigma)} \cdot (\Omega_1', \mathtt{n}, \sigma)$ 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 6, 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, $\Sigma_0 \stackrel{\text{def}}{=} \mathtt{w}, \overline{(\Omega, \omega, \sigma)} \cdot (\Omega_0, \mathtt{n}+1, \sigma)$, $\Omega_0 \xrightarrow{\lambda_0^{\sigma_0}} \Omega_1$, $\Omega \equiv \mathtt{C}, \mathtt{H}, \overline{\mathtt{B}} \triangleright \mathtt{s}; \mathtt{s}'$, $\mathtt{s} \not\equiv \mathtt{s}''; \mathtt{s}'''$ and $\mathtt{s} \not\equiv \mathtt{lfence}$, and $\lambda = \lambda_0^{\sigma_0 \sqcap \sigma}$, and $\Omega_1 \stackrel{\text{def}}{=} \mathtt{w}, \overline{(\Omega, \omega, \sigma)} \cdot (\Omega_1, \mathtt{n}, \sigma)$. From $\Sigma_0 \approx \Sigma_0'$, we get that $\Sigma_0' \stackrel{\text{def}}{=} \mathtt{w}, \overline{(\Omega', \omega, \sigma)} \cdot (\Omega_0', \mathtt{n}+1, \sigma)$ and $\Omega_0 \approx \Omega_0'$. From $\Omega_0 \approx \Omega_0'$ and $\Omega_0 \equiv \mathtt{C}, \mathtt{H}, \overline{\mathtt{B}} \triangleright \mathtt{s}; \mathtt{s}'$, we get that $\Omega_0' \equiv \mathtt{C}, \mathtt{H}, \overline{\mathtt{B}} \triangleright \mathtt{s}; \mathtt{s}'$. Since $\lambda = \lambda'$ and $\lambda = \lambda_0^{\sigma_0 \sqcap \sigma}$, $\Sigma_0' \xrightarrow{\lambda'} \Sigma_1'$ has also been derived using the E-T-speculate-action rule. Thus, $\Sigma_1' \stackrel{\text{def}}{=} \mathtt{w}, \overline{(\Omega', \omega, \sigma)} \cdot (\Omega_1', \mathtt{n}, \sigma)$ and $\Omega_0' \xrightarrow{\lambda_0'^{\sigma_0'}} \Omega_1'$. From $\lambda = \lambda_0^{\sigma_0 \sqcap \sigma}$, $\lambda' = \lambda_0'^{\sigma_0' \sqcap \sigma}$, 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 9, 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 6, 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, $\Sigma_0 \stackrel{\text{def}}{=} \mathtt{w}, \overline{(\Omega, \omega, \sigma)} \cdot (\Omega_0, \mathtt{n}+1, \sigma)$, $\Omega_0 \stackrel{\text{def}}{=} \mathtt{C}, \mathtt{H}, \overline{\mathtt{B}} \cdot \mathtt{B} \triangleright\ (\mathtt{if\ e\ then\ s''\ else\ s'''}; \mathtt{s}')_{\overline{\mathtt{f}} \cdot \mathtt{f}}$, $\mathtt{C} \equiv \overline{\mathtt{F}}; \overline{\mathtt{I}}$, $\mathtt{f} \notin \overline{\mathtt{I}}$, $\Omega_0 \xrightarrow{\alpha^{\sigma_0}} \Omega_1$, $\lambda = \alpha^{\sigma_0 \sqcap \sigma}$, and $\Sigma_1 \stackrel{\text{def}}{=} \mathtt{w}, \overline{(\Omega, \omega, \sigma)} \cdot (\Omega_1, \mathtt{n}, \sigma) \cdot (\Omega_2, \mathtt{mint}(\mathtt{w}, \mathtt{n}), \mathtt{U})$ where $\Omega_2 = \mathtt{C}, \mathtt{H}, \overline{\mathtt{B}} \cdot \mathtt{B} \triangleright \mathtt{s}'''; \mathtt{s}'$ if $\mathtt{B} \triangleright \mathtt{e} \downarrow \mathtt{0} : \sigma_0$ and $\Omega_2 = \mathtt{C}, \mathtt{H}, \overline{\mathtt{B}} \cdot \mathtt{B} \triangleright \mathtt{s}''; \mathtt{s}'$ if $\mathtt{B} \triangleright \mathtt{e} \downarrow \mathtt{n} : \sigma_0 \wedge \mathtt{n} > \mathtt{0}$. From $\Sigma_0 \approx \Sigma_0'$, we get that $\Sigma_0' \stackrel{\text{def}}{=} \mathtt{w}, \overline{(\Omega', \omega, \sigma)} \cdot (\Omega_0', \mathtt{n}+1, \sigma)$ where $\overline{(\Omega', \omega, \sigma)} \approx \overline{(\Omega, \omega, \sigma)}$ and $\Omega_0 \approx \Omega_0'$. Thus, $\Omega_0' \stackrel{\text{def}}{=} \mathtt{C}, \mathtt{H}', \overline{\mathtt{B}'} \triangleright\ (\mathtt{if\ e\ then\ s''\ else\ s'''}; \mathtt{s}')_{\overline{\mathtt{f}} \cdot \mathtt{f}}$ where $\mathtt{H} \approx \mathtt{H}'$ and $\overline{\mathtt{B}} \approx \overline{\mathtt{B}'}$. Hence, $\Sigma_0' \xrightarrow{\lambda'} \Sigma_1'$ has also been derived using the E-T-speculate-action rule. Thus, $\Sigma_1' \stackrel{\text{def}}{=} \mathtt{w}, \overline{(\Omega', \omega, \sigma)} \cdot (\Omega_1', \mathtt{n}, \sigma) \cdot (\Omega_2', \mathtt{mint}(\mathtt{w}, \mathtt{n}), \mathtt{U})$ and $\Omega_0' \xrightarrow{\alpha'^{\sigma_0'}} \Omega_1'$. From $\lambda = \alpha^{\sigma_0 \sqcap \sigma}$, $\lambda' = \alpha'^{\sigma_0' \sqcap \sigma}$, 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 9, 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 6, we get $\Omega_1 \approx \Omega_1'$. Observe that from $\Omega_0 \approx \Omega_0'$ and $\Omega_0' \stackrel{\text{def}}{=} \mathtt{C}, \mathtt{H}', \overline{\mathtt{B}'} \triangleright\ (\mathtt{if\ e\ then\ s''\ else\ s'''}; \mathtt{s}')_{\overline{\mathtt{f}} \cdot \mathtt{f}}$, we get that $\alpha^{\sigma_0} = (\mathtt{if}(\mathtt{n}))^{\sigma_0} \leftrightarrow \mathtt{B} \triangleright \mathtt{e} \downarrow \mathtt{n} : \sigma_0$ and $\alpha'^{\sigma_0'} = (\mathtt{if}(\mathtt{n}'))^{\sigma_0'} \leftrightarrow \mathtt{B}' \triangleright \mathtt{e} \downarrow \mathtt{n}' : \sigma_0'$. From this and $\alpha^{\sigma_0'} = \alpha'^{\sigma_0'}$, we get $\mathtt{n} : \sigma_0 = \mathtt{n}' : \sigma_0'$. 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, $\Sigma_0 \stackrel{\text{def}}{=} \mathtt{w}, \overline{(\Omega, \omega, \sigma)} \cdot (\Omega, 0, \sigma)$, $\lambda = \mathtt{rlb}$, and $\Sigma_1 \stackrel{\text{def}}{=} \mathtt{w}, \overline{(\Omega, \omega, \sigma)}$. From $\Sigma_0 \approx \Sigma_0'$, we get that $\Sigma_0' \stackrel{\text{def}}{=} \mathtt{w}, \overline{(\Omega', \omega, \sigma)} \cdot (\Omega_0', 0, \sigma)$ where $\overline{(\Omega', \omega, \sigma)} \approx \overline{(\Omega, \omega, \sigma)}$ and $\Omega_0 \approx \Omega_0'$. Hence, $\Sigma_0' \xrightarrow{\lambda'} \Sigma_1'$ has also been derived using the E-T-speculate-rollback rule. Thus, $\Sigma_1' \stackrel{\text{def}}{=} \mathtt{w}, \overline{(\Omega', \omega, \sigma)}$. Hence, $\Sigma_1 \approx \Sigma_1'$ follows from $\overline{(\Omega', \omega, \sigma)} \approx \overline{(\Omega, \omega, \sigma)}$.

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

This concludes our proof. □

3) Non-speculative semantics $\longrightarrow$:

Lemma 6 (Steps of $\longrightarrow$ 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 Ω_0, Ω_0', λ, λ', Ω_1, Ω_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, $\Omega_0 \stackrel{\text{def}}{=} \mathtt{C}, \mathtt{H}, \overline{\mathtt{B}} \triangleright \mathtt{skip}; \mathtt{s}$, $\lambda = \epsilon$, and $\Omega_1 \stackrel{\text{def}}{=} \mathtt{C}, \mathtt{H}, \overline{\mathtt{B}} \triangleright \mathtt{s}$. From $\Omega_0 \approx \Omega_0'$, we have that $\Omega_0' \stackrel{\text{def}}{=} \mathtt{C}, \mathtt{H}', \overline{\mathtt{B}'} \triangleright \mathtt{skip}; \mathtt{s}$ where $\mathtt{H} \approx \mathtt{H}'$ and $\overline{\mathtt{B}} \approx \overline{\mathtt{B}'}$. Therefore, $\Omega_0' \xrightarrow{\lambda'} \Omega_1'$ has also been derived using the E-T-sequence where $\Omega_1' \stackrel{\text{def}}{=} \mathtt{C}, \mathtt{H}', \overline{\mathtt{B}'} \triangleright \mathtt{s}$ and $\lambda' = \epsilon$. Hence, $\Omega_1 \approx \Omega_1'$ follows from $\mathtt{H} \approx \mathtt{H}'$, $\overline{\mathtt{B}} \approx \overline{\mathtt{B}'}$, and $\mathtt{B} \approx \mathtt{B}'$.

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

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

E-T-letin: Then, $\Omega_0 \stackrel{\text{def}}{=} \mathtt{C}, \mathtt{H}, \overline{\mathtt{B}} \cdot \mathtt{B} \triangleright \mathtt{let\ x = e\ in\ s}$, $\lambda = \epsilon$, $\mathtt{B} \triangleright \mathtt{e} \downarrow \mathtt{v} : \sigma$, and $\Omega_1 \stackrel{\text{def}}{=} \mathtt{C}, \mathtt{H}, \overline{\mathtt{B}} \cdot \mathtt{B} \cup \mathtt{x} \mapsto \mathtt{v} : \sigma \triangleright \mathtt{s}$. From $\Omega_0 \approx \Omega_0'$, we have that $\Omega_0' \stackrel{\text{def}}{=} \mathtt{C}, \mathtt{H}', \overline{\mathtt{B}'} \cdot \mathtt{B}' \triangleright \mathtt{let\ x = e\ in\ s}$ where $\mathtt{H} \approx \mathtt{H}'$, $\overline{\mathtt{B}} \approx \overline{\mathtt{B}'}$, and $\mathtt{B} \approx \mathtt{B}'$. From

this, $\Omega_0' \xrightarrow{\lambda'} \Omega_1'$ has also been derived using the E-T-letin rule. Hence, $\Omega_1' \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}', \overline{\mathbf{B}'} \cdot \mathbf{B}' \cup \mathbf{x} \mapsto \mathbf{v}' : \sigma' \triangleright \mathbf{s}$ where $\mathbf{B}' \triangleright \mathbf{e} \downarrow \mathbf{v}' : \sigma'$. From lemma 10, $\mathbf{B} \approx \mathbf{B}'$, $\mathbf{B} \triangleright \mathbf{e} \downarrow \mathbf{v} : \sigma$, and $\mathbf{B}' \triangleright \mathbf{e} \downarrow \mathbf{v}' : \sigma'$, we have $\mathbf{v} : \sigma \approx \mathbf{v}' : \sigma'$. Hence, $\Omega_1 \approx \Omega_1'$ follows from $\mathbf{H} \approx \mathbf{H}'$, $\overline{\mathbf{B}} \approx \overline{\mathbf{B}'}$, $\mathbf{B} \approx \mathbf{B}'$, and $\mathbf{v} : \sigma \approx \mathbf{v}' : \sigma'$.

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

E-T-read: Then, $\Omega_0 \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \cdot \mathbf{B} \triangleright \mathbf{let\ x = rd\ e\ in\ s}$, $\mathbf{B} \triangleright \mathbf{e} \downarrow \mathbf{n} : \sigma_1$, $\mathbf{H} = \mathbf{H}_1; |\mathbf{n}| \mapsto \mathbf{v} : \sigma_0; \mathbf{H}_2$, $\lambda = \texttt{read}(|\mathbf{n}|)^{\sigma_1}$, and $\Omega_1 \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \cdot \mathbf{B} \cup \mathbf{x} \mapsto \mathbf{v} : \sigma_0 \triangleright \mathbf{s}$. From $\Omega_0 \approx \Omega_0'$, we get that $\Omega_0' \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}', \overline{\mathbf{B}'} \cdot \mathbf{B}' \triangleright \mathbf{let\ x = rd\ e\ in\ s}$, $\mathbf{H} \approx \mathbf{H}'$, $\overline{\mathbf{B}} \approx \overline{\mathbf{B}'}$, and $\mathbf{B} \approx \mathbf{B}'$. Therefore, $\Omega_0' \xrightarrow{\lambda'} \Omega_1'$ has also been derived using the E-T-read rule. Thus, $\Omega_1' \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}', \overline{\mathbf{B}'} \cdot \mathbf{B}' \cup \mathbf{x} \mapsto \mathbf{v}' : \sigma_0' \triangleright \mathbf{s}$ where $\mathbf{B}' \triangleright \mathbf{e} \downarrow \mathbf{n}' : \sigma_1'$, $\mathbf{H}'(|\mathbf{n}'|) = \mathbf{v}' : \sigma_0'$, and $\lambda' = \texttt{read}(|\mathbf{n}'|)^{\sigma_1'}$. From $\lambda = \texttt{read}(|\mathbf{n}|)^{\sigma_1}$, $\lambda' = \texttt{read}(|\mathbf{n}'|)^{\sigma_1'}$, and $\lambda = \lambda'$, we get $|\mathbf{n}| = |\mathbf{n}'|$. From $|\mathbf{n}| = |\mathbf{n}'|$, $\mathbf{H} \approx \mathbf{H}'$, $\mathbf{H}(|\mathbf{n}|) = \mathbf{v} : \sigma_0$, and $\mathbf{H}'(|\mathbf{n}'|) = \mathbf{v}' : \sigma_0'$, we get $\mathbf{v} : \sigma_0 \approx \mathbf{v}' : \sigma_0'$. Hence, $\Omega_1 \approx \Omega_1'$ follows from $\mathbf{H} \approx \mathbf{H}'$, $\overline{\mathbf{B}} \approx \overline{\mathbf{B}'}$, $\mathbf{B} \approx \mathbf{B}'$, $|\mathbf{n}| = |\mathbf{n}'|$, and $\mathbf{v} : \sigma_0 \approx \mathbf{v}' : \sigma_0'$.

E-T-write-prv: Then, we have that $\Omega_0 \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \cdot \mathbf{B} \triangleright \mathbf{e} :=_{\mathbf{p}} \mathbf{e}'$, $\mathbf{B} \triangleright \mathbf{e} \downarrow \mathbf{n} : \sigma_0$, $\mathbf{B} \triangleright \mathbf{e}' \downarrow \mathbf{v} : \sigma_1$, $\lambda = \texttt{write}(-|\mathbf{n}|)^{\sigma_0}$, $\mathbf{H} = \mathbf{H}_2; -|\mathbf{n}| \mapsto \mathbf{v}_0 : \sigma_2; \mathbf{H}_3$, $\mathbf{H}_1 = \mathbf{H}_2; -|\mathbf{n}| \mapsto \mathbf{v} : \sigma_1; \mathbf{H}_3$, and $\Omega_1 \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}_1, \overline{\mathbf{B}} \cdot \mathbf{B} \triangleright \mathbf{skip}$. From $\Omega_0 \approx \Omega_0'$, we get that $\Omega_0' \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}', \overline{\mathbf{B}'} \cdot \mathbf{B}' \triangleright \mathbf{e} :=_{\mathbf{p}} \mathbf{e}'$, $\mathbf{H} \approx \mathbf{H}'$, $\overline{\mathbf{B}} \approx \overline{\mathbf{B}'}$, and $\mathbf{B} \approx \mathbf{B}'$. Hence, $\Omega_0' \xrightarrow{\lambda'} \Omega_1'$ has also been derived using the E-T-write-prv rule. Thus, $\Omega_1' \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}_1', \overline{\mathbf{B}'} \cdot \mathbf{B}' \triangleright \mathbf{skip}$ where $\mathbf{B}' \triangleright \mathbf{e} \downarrow \mathbf{n}' : \sigma_0'$, $\mathbf{B}' \triangleright \mathbf{e}' \downarrow \mathbf{v}' : \sigma_1'$, $\mathbf{H}_1' = \mathbf{H}_2'; -|\mathbf{n}'| \mapsto \mathbf{v}' : \sigma_1'; \mathbf{H}_3$, and $\lambda' = \texttt{write}(-|\mathbf{n}'|)^{\sigma_0'}$. From $\lambda = \lambda'$, $\lambda' = \texttt{write}(-|\mathbf{n}'|)^{\sigma_0'}$, and $\lambda = \texttt{write}(-|\mathbf{n}|)^{\sigma_0}$, we get $|\mathbf{n}| : \sigma_0 = |\mathbf{n}'| : \sigma_0'$. From $\mathbf{B} \approx \mathbf{B}'$, $\mathbf{B}' \triangleright \mathbf{e}' \downarrow \mathbf{v}' : \sigma_1'$, $\mathbf{B} \triangleright \mathbf{e}' \downarrow \mathbf{v} : \sigma_1$, and lemma 10, we get $\mathbf{v} : \sigma_1 \approx \mathbf{v}' : \sigma_1'$. Hence, $\Omega_1 \approx \Omega_1'$ follows from $\mathbf{H} \approx \mathbf{H}'$, $\overline{\mathbf{B}} \approx \overline{\mathbf{B}'}$, $\mathbf{B} \approx \mathbf{B}'$, $|\mathbf{n}| : \sigma_0 = |\mathbf{n}'| : \sigma_0'$, and $\mathbf{v} : \sigma_1 \approx \mathbf{v}' : \sigma_1'$.

E-T-read-prv: Then, $\Omega_0 \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \cdot \mathbf{B} \triangleright \mathbf{let\ x = rd_p\ e\ in\ s}$, $\mathbf{B} \triangleright \mathbf{e} \downarrow \mathbf{n} : \sigma_0$, $\mathbf{H} = \mathbf{H}_1; -|\mathbf{n}| \mapsto \mathbf{v} : \sigma_1; \mathbf{H}_2$, $\lambda = \texttt{read}(-|\mathbf{n}|)^{\sigma_0}$, and $\Omega_1 \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \cdot \mathbf{B} \cup \mathbf{x} \mapsto \mathbf{v} : \mathbf{U} \triangleright \mathbf{s}$. From $\Omega_0 \approx \Omega_0'$, we get that $\Omega_0' \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}', \overline{\mathbf{B}'} \cdot \mathbf{B}' \triangleright \mathbf{let\ x = rd_p\ e\ in\ s}$ where $\mathbf{H} \approx \mathbf{H}'$, $\overline{\mathbf{B}} \approx \overline{\mathbf{B}'}$, and $\mathbf{B} \approx \mathbf{B}'$. Hence, $\Omega_0' \xrightarrow{\lambda'} \Omega_1'$ has also been derived using the E-T-read-prv rule. Thus, $\Omega_1' \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}', \overline{\mathbf{B}'} \cdot \mathbf{B}' \cup \mathbf{x} \mapsto \mathbf{v}' : \mathbf{U} \triangleright \mathbf{s}$ where $\mathbf{B}' \triangleright \mathbf{e} \downarrow \mathbf{n}' : \sigma_0'$, $\lambda' = \texttt{read}(-|\mathbf{n}'|)^{\sigma_0'}$, and $\mathbf{H}'(-|\mathbf{n}'|) = \mathbf{v}' : \sigma_1'$. From $\lambda = \lambda'$, $\lambda = \texttt{read}(-|\mathbf{n}|)^{\sigma_0}$, and $\lambda' = \texttt{read}(-|\mathbf{n}|)^{\sigma_0'}$, we have $|\mathbf{n}| : \sigma_0 = |\mathbf{n}'| : \sigma_0'$. From this and $\mathbf{H} \approx \mathbf{H}'$, we have $\mathbf{H}(-|\mathbf{n}|) \approx \mathbf{H}'(-|\mathbf{n}'|)$. Hence, $\Omega_1 \approx \Omega_1'$ follows from $\mathbf{H} \approx \mathbf{H}'$, $\overline{\mathbf{B}} \approx \overline{\mathbf{B}'}$, $\mathbf{B} \approx \mathbf{B}'$, $|\mathbf{n}| : \sigma_0 = |\mathbf{n}'| : \sigma_0'$, and $\mathbf{v} : \sigma_1 \approx \mathbf{v}' : \sigma_1'$ (observe that the latter is enough since $\mathbf{x}$ is tagged $\mathbf{U}$).

E-T-call-internal: Then, $\Omega_0 \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \cdot \mathbf{B} \triangleright (\mathbf{call\ f\ e})_{\overline{\mathbf{f}'}}$, $\lambda = \epsilon$, $\mathbf{C}.\texttt{intfs} \vdash \mathbf{f}, \mathbf{f}' : \mathbf{internal}$, $\overline{\mathbf{f}'} = \overline{\mathbf{f}''}; \mathbf{f}'$, $\mathbf{f}(\mathbf{x}) \mapsto \mathbf{s}; \mathbf{return}; \in \mathbf{C}.\texttt{funs}$, $\mathbf{B} \triangleright \mathbf{e} \downarrow \mathbf{v} : \sigma$, and $\Omega_1 \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \cdot \mathbf{B} \cdot \mathbf{x} \mapsto \mathbf{v} : \sigma \triangleright (\mathbf{s}; \mathbf{return};)_{\overline{\mathbf{f}'}; \mathbf{f}}$. From $\Omega_0 \approx \Omega_0'$, we get that $\Omega_0' \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}', \overline{\mathbf{B}'} \cdot \mathbf{B}' \triangleright (\mathbf{call\ f\ e})_{\overline{\mathbf{f}'}}$ where $\mathbf{H} \approx \mathbf{H}'$, $\overline{\mathbf{B}} \approx \overline{\mathbf{B}'}$, and $\mathbf{B} \approx \mathbf{B}'$. Hence, $\Omega_0' \xrightarrow{\lambda'} \Omega_1'$ has also been derived using the E-T-call-internal rule. Thus, $\Omega_1' \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}', \overline{\mathbf{B}'} \cdot \mathbf{B}' \cdot \mathbf{x} \mapsto \mathbf{v}' : \sigma' \triangleright (\mathbf{s}; \mathbf{return};)_{\overline{\mathbf{f}'}; \mathbf{f}}$ and $\mathbf{B}' \triangleright \mathbf{e} \downarrow \mathbf{v}' : \sigma'$. From $\mathbf{B} \approx \mathbf{B}'$, $\mathbf{B} \triangleright \mathbf{e} \downarrow \mathbf{v} : \sigma$, $\mathbf{B}' \triangleright \mathbf{e} \downarrow \mathbf{v}' : \sigma'$, and lemma 10, we get $\mathbf{v} : \sigma \approx \mathbf{v}' : \sigma'$. Hence, $\Omega_1 \approx \Omega_1'$ follows from $\mathbf{H} \approx \mathbf{H}'$, $\overline{\mathbf{B}} \approx \overline{\mathbf{B}'}$, $\mathbf{B} \approx \mathbf{B}'$, and $\mathbf{v} : \sigma \approx \mathbf{v}' : \sigma'$.

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

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

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

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, $\Omega_0 \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \cdot \mathbf{B} \triangleright \mathbf{let\ x = e_0\ (if\ e_1)\ in\ s}$, $\mathbf{x} \in \mathtt{dom}(\mathbf{B})$, $\lambda = \epsilon$, $\Omega_1 \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \cdot \mathbf{B} \cup \mathbf{x} \mapsto \mathbf{v_0} : \boldsymbol{\sigma} \triangleright \mathbf{s}$, $\mathbf{B} \triangleright \mathbf{e_0} \downarrow \mathbf{v_0} : \boldsymbol{\sigma_0}$, $\mathbf{B} \triangleright \mathbf{e_1} \downarrow \mathbf{0} : \boldsymbol{\sigma_1}$, and $\boldsymbol{\sigma} = \boldsymbol{\sigma_0} \sqcup \boldsymbol{\sigma_1}$. From $\Omega_0 \approx \Omega_0'$, we have that $\Omega_0' \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}', \overline{\mathbf{B}'} \cdot \mathbf{B}' \triangleright \mathbf{let\ x = e_0\ (if\ e_1)\ in\ s}$ where $\mathbf{H} \approx \mathbf{H}'$, $\overline{\mathbf{B}} \cdot \mathbf{B} \approx \overline{\mathbf{B}'} \cdot \mathbf{B}'$, and $\mathbf{B} \approx \mathbf{B}'$. From $\mathbf{B} \approx \mathbf{B}'$, $\mathbf{B} \triangleright \mathbf{e_1} \downarrow \mathbf{0} : \boldsymbol{\sigma_1}$, and lemma 11, we get that $\mathbf{B}' \triangleright \mathbf{e_1} \downarrow \mathbf{n}' : \boldsymbol{\sigma}_1'$ and $\mathbf{0} : \boldsymbol{\sigma_1} \approx \mathbf{n}' : \boldsymbol{\sigma}_1'$. There are two cases:

- $\boldsymbol{\sigma_1} = \mathbf{S}$**:** Then, $\boldsymbol{\sigma}_1' = \mathbf{S}$ and $\mathbf{v}_1' = \mathbf{0}$ follows from $\mathbf{0} : \boldsymbol{\sigma_1} \approx \mathbf{n}' : \boldsymbol{\sigma}_1'$. Hence, $\Omega_0' \xrightarrow{\lambda'} \Omega_1'$ has also been derived using the E-T-cmove-true rule. Thus, $\Omega_1 \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}', \overline{\mathbf{B}'} \cdot \mathbf{B}' \cup \mathbf{x} \mapsto \mathbf{v}_0' : \boldsymbol{\sigma}' \triangleright \mathbf{s}$ where $\mathbf{B}' \triangleright \mathbf{e_0} \downarrow \mathbf{v}_0' : \boldsymbol{\sigma}_0'$ and $\boldsymbol{\sigma}' = \boldsymbol{\sigma}_0' \sqcup \boldsymbol{\sigma}_1'$. From $\mathbf{B} \approx \mathbf{B}'$, $\mathbf{B} \triangleright \mathbf{e_0} \downarrow \mathbf{v_0} : \boldsymbol{\sigma_0}$, $\mathbf{B}' \triangleright \mathbf{e_0} \downarrow \mathbf{v}_0' : \boldsymbol{\sigma}_0'$, and lemma 10, we get $\mathbf{v_0} : \boldsymbol{\sigma_0} \approx \mathbf{v}_0' : \boldsymbol{\sigma}_0'$. Hence, $\Omega_1 \approx \Omega_1'$ follows from $\mathbf{H} \approx \mathbf{H}'$, $\overline{\mathbf{B}} \approx \overline{\mathbf{B}'}$, $\mathbf{B} \approx \mathbf{B}'$, and $\mathbf{v_0} : \boldsymbol{\sigma_0} \approx \mathbf{v}_0' : \boldsymbol{\sigma}_0'$.
- $\boldsymbol{\sigma_1} = \mathbf{U}$**:** Then, $\boldsymbol{\sigma}_1' = \mathbf{U}$ holds as well. There are two cases:
  - $\mathbf{v}_1' = \mathbf{0}$**:** Hence, $\Omega_0' \xrightarrow{\lambda'} \Omega_1'$ has also been derived using the E-T-cmove-true rule. Thus, $\Omega_1 \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}', \overline{\mathbf{B}'} \cdot \mathbf{B}' \cup \mathbf{x} \mapsto \mathbf{v}_0' : \boldsymbol{\sigma}' \triangleright \mathbf{s}$ where $\mathbf{B}' \triangleright \mathbf{e_0} \downarrow \mathbf{v}_0' : \boldsymbol{\sigma}_0'$ and $\boldsymbol{\sigma}' = \boldsymbol{\sigma}_0' \sqcup \boldsymbol{\sigma}_1'$. Hence, $\Omega_1 \approx \Omega_1'$ follows from $\mathbf{H} \approx \mathbf{H}'$, $\overline{\mathbf{B}} \approx \overline{\mathbf{B}'}$, $\mathbf{B} \approx \mathbf{B}'$, and $\sigma = \sigma' = \mathbf{U}$.
  - $\mathbf{v}_1' > \mathbf{0}$**:** Hence, $\Omega_0' \xrightarrow{\lambda'} \Omega_1'$ has been derived using the E-T-cmove-false rule. Thus, $\Omega_1' \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}', \overline{\mathbf{B}'} \cdot \mathbf{B}' \cup \mathbf{x} \mapsto \mathbf{v_0} : \boldsymbol{\sigma}' \triangleright \mathbf{s}$ where $\mathbf{B(x)} = \mathbf{v}_0' : \boldsymbol{\sigma}_0'$ and $\boldsymbol{\sigma}' = \boldsymbol{\sigma}_0' \sqcup \boldsymbol{\sigma}_1'$. Hence, $\Omega_1 \approx \Omega_1'$ follows from $\mathbf{H} \approx \mathbf{H}'$, $\overline{\mathbf{B}} \approx \overline{\mathbf{B}'}$, $\mathbf{B} \approx \mathbf{B}'$, and $\sigma = \sigma' = \mathbf{U}$.

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, $\Omega_0 \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \triangleright \mathbf{s}; \mathbf{s}''$, $\mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \triangleright \mathbf{s} \xrightarrow{\lambda} \mathbf{C}, \mathbf{H_1}, \overline{\mathbf{B_1}} \triangleright \mathbf{s_1}$, and $\Omega_1 = \mathbf{C}, \mathbf{H_1}, \overline{\mathbf{B_1}} \triangleright \mathbf{s_1}; \mathbf{s}''$. From $\Omega_0 \approx \Omega_0'$, we get that $\Omega_0 \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}', \overline{\mathbf{B}'} \triangleright \mathbf{s}; \mathbf{s}''$ where $\mathbf{H} \approx \mathbf{H}'$ and $\overline{\mathbf{B}} \approx \overline{\mathbf{B}'}$. Hence, $\Omega_0' \xrightarrow{\lambda'} \Omega_1'$ has also been derived using the E-T-step rule. Therefore, we get $\Omega_0' \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}', \overline{\mathbf{B}'} \triangleright \mathbf{s}; \mathbf{s}''$, $\mathbf{C}, \mathbf{H}', \overline{\mathbf{B}'} \triangleright \mathbf{s} \xrightarrow{\lambda'} \mathbf{C}, \mathbf{H}_1', \overline{\mathbf{B}_1'} \triangleright \mathbf{s_1}$, and $\Omega_1' = \mathbf{C}, \mathbf{H}_1', \overline{\mathbf{B}_1'} \triangleright \mathbf{s_1}; \mathbf{s}''$. From $\mathbf{H} \approx \mathbf{H}'$, $\overline{\mathbf{B}} \approx \overline{\mathbf{B}'}$, $\mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \triangleright \mathbf{s} \xrightarrow{\lambda} \mathbf{C}, \mathbf{H_1}, \overline{\mathbf{B_1}} \triangleright \mathbf{s_1}$, $\mathbf{C}, \mathbf{H}', \overline{\mathbf{B}'} \triangleright \mathbf{s} \xrightarrow{\lambda'} \mathbf{C}, \mathbf{H}_1', \overline{\mathbf{B}_1'} \triangleright \mathbf{s_1}$, $\lambda = \lambda'$, and the induction hypothesis, we get that $\mathbf{H_1} \approx \mathbf{H}_1'$ and $\overline{\mathbf{B_1}} \approx \overline{\mathbf{B}_1'}$. Hence, $\Omega_1 \approx \Omega_1'$ follows from $\mathbf{H_1} \approx \mathbf{H}_1'$ and $\overline{\mathbf{B_1}} \approx \overline{\mathbf{B}_1'}$.

□

Lemma 7 (Steps of $\longrightarrow$ with safe observations preserve low-equivalence)**.**

$$\forall \Omega_0, \Omega_0', \lambda, \Omega_1.\ \text{if } \Omega_0 \xrightarrow{\lambda} \Omega_1, \Omega_0 \approx \Omega_0', \vdash \lambda : \mathit{safe}$$
$$\text{then } \exists \lambda', \Omega_1.\ \Omega_0' \xrightarrow{\lambda'} \Omega_1', \Omega_1 \approx \Omega_1',\ \lambda = \lambda'$$

Proof. Let Ω_0, Ω_0', λ, and Ω_1 be such that $\Omega_0 \approx \Omega_0'$, $\Omega_0 \xrightarrow{\lambda} \Omega_1$, and $\vdash \lambda : \mathit{safe}$. 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, $\Omega_0 \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \triangleright \mathbf{skip}; \mathbf{s}$, $\lambda = \epsilon$, and $\Omega_1 \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \triangleright \mathbf{s}$. From $\Omega_0 \approx \Omega_0'$, we have that $\Omega_0' \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}', \overline{\mathbf{B}'} \triangleright \mathbf{skip}; \mathbf{s}$ where $\mathbf{H} \approx \mathbf{H}'$ and $\overline{\mathbf{B}} \approx \overline{\mathbf{B}'}$. We can apply the E-T-sequence rule to derive $\Omega_0' \xrightarrow{\lambda'} \Omega_1'$ where $\Omega_1 \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}', \overline{\mathbf{B}'} \triangleright \mathbf{s}$ 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, $\Omega_0 \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \cdot \mathbf{B} \triangleright \mathbf{ifz\ e\ then\ s\ else\ s'}$, $\mathbf{B} \triangleright \mathbf{e} \downarrow \mathbf{0} : \sigma$ $\lambda = (\mathtt{if(0)})^{\sigma}$, and $\Omega_1 \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \cdot \mathbf{B} \triangleright \mathbf{s}$. From $\Omega_0 \approx \Omega_0'$, we have that $\Omega_0' \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}', \overline{\mathbf{B}'} \cdot \mathbf{B}' \triangleright \mathbf{ifz\ e\ then\ s\ else\ s'}$ where $\mathbf{H} \approx \mathbf{H}'$, $\overline{\mathbf{B}} \approx \overline{\mathbf{B}'}$, and $\mathbf{B} \approx \mathbf{B}'$. From $\mathbf{B} \triangleright \mathbf{e} \downarrow \mathbf{0} : \sigma$, $\mathbf{B} \approx \mathbf{B}'$, and lemma 11, we get that $\mathbf{B}' \triangleright \mathbf{e} \downarrow \mathbf{v}' : \sigma'$ and $\mathbf{0} : \sigma \approx \mathbf{v}' : \sigma'$. From $\vdash \lambda : \mathit{safe}$ and $\lambda = (\mathtt{if(0)})^{\sigma}$, it follows that $\sigma = S$ and $\lambda = (\mathtt{if(0)})^{\mathbf{S}}$. From this and $\mathbf{0} : \sigma \approx \mathbf{v}' : \sigma'$, it follows that $v' = 0$. Hence, $\mathbf{B}' \triangleright \mathbf{e} \downarrow \mathbf{0} : \mathbf{S}'$ holds. Therefore, we can apply the E-T-if-true rule to derive $\Omega_0' \xrightarrow{\lambda'} \Omega_1'$ where $\Omega_1 \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}', \overline{\mathbf{B}'} \cdot \mathbf{B}' \triangleright \mathbf{s}$ and $\lambda' = (\mathtt{if(0)})^{\mathbf{S}}$. Hence, $\Omega_1 \approx \Omega_1'$ follows from $\Omega_0 \approx \Omega_0'$ and $\lambda = \lambda'$ follows from $\lambda = (\mathtt{if(0)})^{\mathbf{S}}$ and $\lambda' = (\mathtt{if(0)})^{\mathbf{S}}$.

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

E-T-letin: Then, $\Omega_0 \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \cdot \mathbf{B} \triangleright \mathbf{let\ x = e\ in\ s}$, $\lambda = \epsilon$, $\mathbf{B} \triangleright \mathbf{e} \downarrow \mathbf{v} : \sigma$, and $\Omega_1 \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \cdot \mathbf{B} \cup \mathbf{x} \mapsto \mathbf{v} : \sigma \triangleright \mathbf{s}$. From $\Omega_0 \approx \Omega_0'$, we have that $\Omega_0' \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}', \overline{\mathbf{B}'} \cdot \mathbf{B}' \triangleright \mathbf{let\ x = e\ in\ s}$ where $\mathbf{H} \approx \mathbf{H}'$, $\overline{\mathbf{B}} \approx \overline{\mathbf{B}'}$, and $\mathbf{B} \approx \mathbf{B}'$. From $\mathbf{B} \approx \mathbf{B}'$, $\mathbf{B} \triangleright \mathbf{e} \downarrow \mathbf{v} : \sigma$, and lemma 11, we have $\mathbf{B}' \triangleright \mathbf{e} \downarrow \mathbf{v}' : \sigma'$ and $\mathbf{v} : \sigma \approx \mathbf{v}' : \sigma'$. Therefore, we can apply the rule E-T-letin to derive $\Omega_0' \xrightarrow{\lambda'} \Omega_1'$ where $\lambda' = \epsilon$ and $\Omega_1 \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}', \overline{\mathbf{B}'} \cdot \mathbf{B}' \cup \mathbf{x} \mapsto \mathbf{v}' : \sigma' \triangleright \mathbf{s}$. Hence, $\Omega_1 \approx \Omega_1'$ follows from $\Omega_0 \approx \Omega_0'$ and $\mathbf{v} : \sigma \approx \mathbf{v}' : \sigma'$ whereas $\lambda = \lambda'$ follows from $\lambda = \epsilon$ and $\lambda' = \epsilon$.

E-T-write: Then, $\Omega_0 \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}_0, \overline{\mathbf{B}} \cdot \mathbf{B} \triangleright \mathbf{e_1} := \mathbf{e_2}$, $\lambda = \mathtt{write}(|\mathbf{n}| \mapsto \mathbf{v})^{\sigma_1 \sqcup \sigma_2}$, $\mathbf{B} \triangleright \mathbf{e_1} \downarrow \mathbf{n} : \sigma_1$, $\mathbf{B} \triangleright \mathbf{e_2} \downarrow \mathbf{v} : \sigma_2$, $\mathbf{H_0} = \mathbf{H_2}; |\mathbf{n}| \mapsto \mathbf{v_0} : \sigma_0; \mathbf{H_3}$, $\mathbf{H_1} = \mathbf{H_2}; |\mathbf{n}| \mapsto \mathbf{v} : \mathbf{S}; \mathbf{H_3}$, and $\Omega_1 \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}_1, \overline{\mathbf{B}} \cdot \mathbf{B} \triangleright \mathbf{skip}$.

From $\Omega_0 \approx \Omega_0'$, $\Omega_0' \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}_0', \overline{\mathbf{B}} \cdot \mathbf{B} \triangleright \mathbf{e}_1 := \mathbf{e}_2$ where $\mathbf{H}_0 \approx \mathbf{H}_0'$, $\overline{\mathbf{B}} \approx \overline{\mathbf{B}'}$, and $\mathbf{B} \approx \mathbf{B}'$. From $\mathbf{H}_0 \approx \mathbf{H}_0'$ and $\mathbf{H}_0 = \mathbf{H}_2; |\mathbf{n}| \mapsto \mathbf{v}_0 : \sigma_0; \mathbf{H}_3$, it follows that there are $\mathbf{H}_2', \mathbf{H}_3'$ such that $\mathbf{H}_0' = \mathbf{H}_2'; |\mathbf{n}| \mapsto \mathbf{v}_0' : \sigma_0; \mathbf{H}_3'$. From $\mathbf{B} \approx \mathbf{B}'$, $\mathbf{B} \triangleright \mathbf{e}_1 \downarrow \mathbf{n} : \sigma_1$, $\mathbf{B} \triangleright \mathbf{e}_2 \downarrow \mathbf{v} : \sigma_2$, and lemma 11, we get $\mathbf{B}' \triangleright \mathbf{e}_1 \downarrow \mathbf{n}' : \sigma_1'$ and $\mathbf{B}' \triangleright \mathbf{e}_2 \downarrow \mathbf{v}' : \sigma_2'$ such that $\mathbf{n} : \sigma_1 \approx \mathbf{n}' : \sigma_1'$ and $\mathbf{v} : \sigma_2 \approx \mathbf{v}' : \sigma_2'$. From $\vdash \lambda : \mathit{safe}$, we get $\sigma_1 \sqcup \sigma_2 = \mathbf{S}$ and therefore $\sigma_1 = \mathbf{S}$ and $\sigma_2 = \mathbf{S}$. From $\sigma_1 = \mathbf{S}$, $\sigma_2 = \mathbf{S}$, $\mathbf{n} : \sigma_1 \approx \mathbf{n}' : \sigma_1'$, and $\mathbf{v} : \sigma_2 \approx \mathbf{v}' : \sigma_2'$, we therefore get that $\mathbf{n} : \sigma_1 = \mathbf{n}' : \sigma_1'$, and $\mathbf{v} : \sigma_2 = \mathbf{v}' : \sigma_2'$. Therefore, we can apply the rule E-T-write to derive $\Omega_0' \xrightarrow{\lambda'} \Omega_1'$ where $\lambda' = \texttt{write}(|\mathbf{n}'| \mapsto \mathbf{v}')^{\sigma_1' \sqcup \sigma_2'}$, $H_1' = \mathbf{H}_2'; |\mathbf{n}| \mapsto \mathbf{v}' : \mathbf{S}; \mathbf{H}_3'$, $\Omega_1 \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}_1', \overline{\mathbf{B}'} \cdot \mathbf{B}' \triangleright \mathbf{skip}$.

From $\mathbf{n} : \sigma_1 = \mathbf{n}' : \sigma_1'$, $\mathbf{v} : \sigma_2 = \mathbf{v}' : \sigma_2'$, $\lambda' = \texttt{write}(|\mathbf{n}'| \mapsto \mathbf{v}')^{\sigma_1' \sqcup \sigma_2'}$, and $\lambda = \texttt{write}(|\mathbf{n}| \mapsto \mathbf{v})^{\sigma_1 \sqcup \sigma_2}$, we immediately get that $\lambda = \lambda'$.

Finally, $\Omega_1 \approx \Omega_1'$ follows from $\Omega_0 \approx \Omega_0'$, $\Omega_1 \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}_1', \overline{\mathbf{B}'} \cdot \mathbf{B}' \triangleright \mathbf{skip}$, $\Omega_1' \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}_1', \overline{\mathbf{B}'} \cdot \mathbf{B}' \triangleright \mathbf{skip}$, $\mathbf{H}_1 = \mathbf{H}_2; |\mathbf{n}| \mapsto \mathbf{v} : \mathbf{S}; \mathbf{H}_3$, $\mathbf{H}_1' = \mathbf{H}_2'; |\mathbf{n}'| \mapsto \mathbf{v}' : \mathbf{S}; \mathbf{H}_3'$, $\mathbf{n} : \sigma_1 = \mathbf{n}' : \sigma_1'$, $\mathbf{v} : \sigma_2 = \mathbf{v}' : \sigma_2'$, $\overline{\mathbf{B}} \approx \overline{\mathbf{B}'}$, and $\mathbf{B} \approx \mathbf{B}'$.

E-T-read: Then, $\Omega_0 \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \cdot \mathbf{B} \triangleright \mathbf{let\ x = rd\ e\ in\ s}$, $\mathbf{B} \triangleright \mathbf{e} \downarrow \mathbf{n} : \sigma_1$, $\mathbf{H} = \mathbf{H}_1; |\mathbf{n}| \mapsto \mathbf{v} : \sigma_0; \mathbf{H}_2$, $\lambda = \texttt{read}(|\mathbf{n}|)^{\sigma_1}$, and $\Omega_1 \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \cdot \mathbf{B} \cup \mathbf{x} \mapsto \mathbf{v} : \sigma_0 \triangleright \mathbf{s}$. From $\mathbf{B} \approx \mathbf{B}'$, $\mathbf{B} \triangleright \mathbf{e} \downarrow \mathbf{n} : \sigma_1$, and lemma 11, we get $\mathbf{B}' \triangleright \mathbf{e} \downarrow \mathbf{n}' : \sigma_1'$ and $\mathbf{n} : \sigma_1 \approx \mathbf{n}' : \sigma_1'$. From $\vdash \lambda : \mathit{safe}$, we have that $\sigma_1 = \mathbf{S}$. From $\sigma_1 = \mathbf{S}$ and $\mathbf{n} : \sigma_1 \approx \mathbf{n}' : \sigma_1'$, we get $\mathbf{n} : \sigma_1 = \mathbf{n}' : \sigma_1'$.

From $\Omega_0 \approx \Omega_0'$ and $\mathbf{n} : \sigma_1 = \mathbf{n}' : \sigma_1'$, we get that $\Omega_0' \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}', \overline{\mathbf{B}'} \cdot \mathbf{B}' \triangleright \mathbf{let\ x = rd\ e\ in\ s}$ where $\mathbf{H}' = \mathbf{H}_1'; |\mathbf{n}| \mapsto \mathbf{v}' : \sigma_0'; \mathbf{H}_2'$, $\mathbf{H} \approx \mathbf{H}'$, $\overline{\mathbf{B}} \approx \overline{\mathbf{B}'}$, $\mathbf{B} \approx \mathbf{B}'$, and $\mathbf{v} : \sigma_0 \approx \mathbf{v}' : \sigma_0'$.

Therefore, we can apply the rule E-T-read to derive $\Omega_0' \xrightarrow{\lambda'} \Omega_1'$ where $\lambda' = \texttt{read}(|\mathbf{n}|)^{\sigma_1'}$, and $\Omega_1 \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}', \overline{\mathbf{B}'} \cdot \mathbf{B}' \cup \mathbf{x} \mapsto \mathbf{v}' : \boldsymbol{\sigma_0'} \triangleright \mathbf{skip}$. Hence, $\Omega_1 \approx \Omega_1'$ and $\lambda = \lambda'$ follow from $\mathbf{H} \approx \mathbf{H}'$, $\overline{\mathbf{B}} \approx \overline{\mathbf{B}'}$, $\mathbf{B} \approx \mathbf{B}'$, $\mathbf{v} : \sigma_0 \approx \mathbf{v}' : \sigma_0'$, and $\mathbf{n} : \sigma_1 = \mathbf{n}' : \sigma_1'$.

E-T-write-prv: Then, we have that $\Omega_0 \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \cdot \mathbf{B} \triangleright \mathbf{e} :=_{\mathbf{p}} \mathbf{e}'$, $\mathbf{B} \triangleright \mathbf{e} \downarrow \mathbf{n} : \sigma_0$, $\mathbf{B} \triangleright \mathbf{e}' \downarrow \mathbf{v} : \sigma_1$, $\lambda = \texttt{write}(-|\mathbf{n}|)^{\sigma_0}$, $\mathbf{H} = \mathbf{H}_2; -|\mathbf{n}| \mapsto \mathbf{v}_0 : \sigma_2; \mathbf{H}_3$, $\mathbf{H}_1 = \mathbf{H}_2; -|\mathbf{n}| \mapsto \mathbf{v} : \sigma_1; \mathbf{H}_3$, and $\Omega_1 \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}_1, \overline{\mathbf{B}} \cdot \mathbf{B} \triangleright \mathbf{skip}$.

From $\Omega_0 \approx \Omega_0'$, we get that $\Omega_0' \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}', \overline{\mathbf{B}'} \cdot \mathbf{B}' \triangleright \mathbf{e} :=_{\mathbf{p}} \mathbf{e}'$, $\mathbf{H}' = \mathbf{H}_2'; -|\mathbf{n}| \mapsto \mathbf{v}_0' : \sigma_2'; \mathbf{H}_3'$, $\mathbf{H} \approx \mathbf{H}'$, $\overline{\mathbf{B}} \approx \overline{\mathbf{B}'}$, and $\mathbf{B} \approx \mathbf{B}'$.

From $\vdash \lambda : \mathit{safe}$, we get that $\sigma_0 = \mathbf{S}$. From $\Omega_0 \approx \Omega_0'$, $\mathbf{B} \triangleright \mathbf{e} \downarrow \mathbf{n} : \sigma_0$, and lemma 11, we get $\mathbf{B} \triangleright \mathbf{e} \downarrow \mathbf{n}' : \sigma_0'$ and $\mathbf{n} : \sigma_0 \approx \mathbf{n}' : \sigma_0'$. From this and $\sigma_0 = \mathbf{S}$, we get $\mathbf{n} : \sigma_0 = \mathbf{n}' : \sigma_0'$.

From $\Omega_0 \approx \Omega_0'$, $\mathbf{B} \triangleright \mathbf{e}' \downarrow \mathbf{v} : \sigma_1$, and lemma 11, we get $\mathbf{B} \triangleright \mathbf{e}' \downarrow \mathbf{v}' : \sigma_1'$ and $\mathbf{v} : \sigma_1 \approx \mathbf{v}' : \sigma_1'$.

Hence, we can apply the rule E-T-write-prv to derive $\Omega_0' \xrightarrow{\lambda'} \Omega_1'$ where $\lambda = \texttt{write}(-|\mathbf{n}'|)^{\sigma_0'}$ and $\Omega_1 \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}', \overline{\mathbf{B}'} \cdot \mathbf{B}' \triangleright \mathbf{skip}$ where $\mathbf{H}_1' = \mathbf{H}_2'; -|\mathbf{n}'| \mapsto \mathbf{v}' : \sigma_1'; \mathbf{H}_3$. Therefore, $\lambda = \lambda'$ follows from $\mathbf{n} : \sigma_0 = \mathbf{n}' : \sigma_0'$ and $\Omega_1 \approx \Omega_1'$ follows from $\Omega_0 \approx \Omega_0'$, $\mathbf{n} : \sigma_0 = \mathbf{n}' : \sigma_0'$, and $\mathbf{v} : \sigma_1 \approx \mathbf{v}' : \sigma_1'$.

E-T-read-prv: Then, $\Omega_0 \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \cdot \mathbf{B} \triangleright \mathbf{let\ x = rd_p\ e\ in\ s}$, $\mathbf{B} \triangleright \mathbf{e} \downarrow \mathbf{n} : \sigma_0$, $\mathbf{H} = \mathbf{H}_1; -|\mathbf{n}| \mapsto \mathbf{v} : \sigma_1; \mathbf{H}_2$, $\lambda = \texttt{read}(-|\mathbf{n}|)^{\sigma_0}$, and $\Omega_1 \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \cdot \mathbf{B} \cup \mathbf{x} \mapsto \mathbf{v} : \mathbf{U} \triangleright \mathbf{s}$.

From $\Omega_0 \approx \Omega_0'$, we get that $\Omega_0' \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}', \overline{\mathbf{B}'} \cdot \mathbf{B}' \triangleright \mathbf{let\ x = rd_p\ e\ in\ s}$ where $\mathbf{H} \approx \mathbf{H}'$, $\overline{\mathbf{B}} \approx \overline{\mathbf{B}'}$, and $\mathbf{B} \approx \mathbf{B}'$. From $\mathbf{H} \approx \mathbf{H}'$ and $\mathbf{H} = \mathbf{H}_1; -|\mathbf{n}| \mapsto \mathbf{v} : \sigma_1; \mathbf{H}_2$, we get that $\mathbf{H} = \mathbf{H}_1'; -|\mathbf{n}| \mapsto \mathbf{v}' : \sigma_1'; \mathbf{H}_2'$ and $\mathbf{v} : \sigma_1 \approx \mathbf{v}' : \sigma'$. Moreover, from $\mathbf{B} \triangleright \mathbf{e} \downarrow \mathbf{n} : \sigma_0$, $\mathbf{B} \approx \mathbf{B}'$, and lemma 11, we get that $\mathbf{B}' \triangleright \mathbf{e} \downarrow \mathbf{n}' : \sigma_0'$ and $\mathbf{n} : \sigma_0 \approx \mathbf{n}' : \sigma_0'$.

From $\vdash \lambda : \mathit{safe}$, we get that $\sigma_0 = \mathbf{S}$. From this and $\mathbf{n} : \sigma_0 \approx \mathbf{n}' : \sigma_0'$, we get $\mathbf{n} : \sigma_0 = \mathbf{n}' : \sigma_0'$.

We can apply the rule E-T-read-prv to derive $\Omega_0' \xrightarrow{\lambda'} \Omega_1'$ where $\lambda = \texttt{read}(-|\mathbf{n}'|)^{\sigma_0'}$, and $\Omega_1' \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}', \overline{\mathbf{B}'} \cdot \mathbf{B}' \cup \mathbf{x} \mapsto \mathbf{v}' : \mathbf{U} \triangleright \mathbf{skip}$. Therefore, $\lambda = \lambda'$ follows from $\mathbf{n} : \sigma_0 = \mathbf{n}' : \sigma_0'$, whereas $\Omega_1 \approx \Omega_1'$ follows from $\Omega_0 \approx \Omega_0'$ and the fact that the values assigned to $\mathbf{x}$ has tag $\mathbf{U}$.

E-T-call-internal: Then, $\Omega_0 \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \cdot \mathbf{B} \triangleright (\mathbf{call\ f\ e})_{\overline{\mathbf{f}'}}$, $\lambda = \epsilon$, $\mathbf{C.intfs} \vdash \mathbf{f}, \mathbf{f}' : \mathbf{internal}$, $\overline{\mathbf{f}'} = \overline{\mathbf{f}''}; \mathbf{f}'$, $\mathbf{f}(\mathbf{x}) \mapsto \mathbf{s}; \mathbf{return}; \in \mathbf{C.funs}$, $\mathbf{B} \triangleright \mathbf{e} \downarrow \mathbf{v} : \sigma$, and $\Omega_1 \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \cdot \mathbf{B} \cdot \mathbf{x} \mapsto \mathbf{v} : \sigma \triangleright (\mathbf{s}; \mathbf{return};)_{\overline{\mathbf{f}'}; \mathbf{f}}$. From $\Omega_0 \approx \Omega_0'$, we get that $\Omega_0' \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}', \overline{\mathbf{B}'} \cdot \mathbf{B}' \triangleright (\mathbf{call\ f\ e})_{\overline{\mathbf{f}'}}$ where $\mathbf{H} \approx \mathbf{H}'$, $\overline{\mathbf{B}} \approx \overline{\mathbf{B}'}$, and $\mathbf{B} \approx \mathbf{B}'$. From $\mathbf{B} \approx \mathbf{B}'$, $\mathbf{B} \triangleright \mathbf{e} \downarrow \mathbf{v} : \sigma$, and lemma 11, we get that $\mathbf{B} \triangleright \mathbf{e} \downarrow \mathbf{v}' : \sigma'$ and $\mathbf{v} : \sigma \approx \mathbf{v} : \sigma'$. Therefore, we can apply the rule E-T-call-internal to derive $\Omega_0' \xrightarrow{\lambda'} \Omega_1'$ where $\lambda' = \epsilon$, and $\Omega_1' \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}', \overline{\mathbf{B}'} \cdot \mathbf{B}' \cdot \mathbf{x} \mapsto \mathbf{v}' : \sigma' \triangleright (\mathbf{s}; \mathbf{return};)_{\overline{\mathbf{f}'}; \mathbf{f}}$. 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 $\mathbf{v} : \sigma \approx \mathbf{v} : \sigma'$).

E-T-callback: Then, $\Omega_0 \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \cdot \mathbf{B} \triangleright (\mathbf{call\ f\ e})_{\overline{\mathbf{f}'}}$, $\lambda = \texttt{call f v!}^{\sigma}$, $\overline{\mathbf{f}'} = \overline{\mathbf{f}''}; \mathbf{f}'$, $\mathbf{f}(\mathbf{x}) \mapsto \mathbf{s}; \mathbf{return}; \in \mathbf{C.funs}$, $\mathbf{C.intfs} \vdash \mathbf{f}', \mathbf{f} : \mathbf{out}$, $\mathbf{B} \triangleright \mathbf{e} \downarrow \mathbf{v} : \sigma$, and $\Omega_1 = \mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \cdot \mathbf{B} \cdot \mathbf{x} \mapsto \mathbf{v} : \sigma \triangleright (\mathbf{s}; \mathbf{return};)_{\overline{\mathbf{f}'}; \mathbf{f}}$. From $\Omega_0 \approx \Omega_0'$, we get that $\Omega_0' \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}', \overline{\mathbf{B}'} \cdot \mathbf{B}' \triangleright (\mathbf{call\ f\ e})_{\overline{\mathbf{f}'}}$ where $\mathbf{H} \approx \mathbf{H}'$, $\overline{\mathbf{B}} \approx \overline{\mathbf{B}'}$, and $\mathbf{B} \approx \mathbf{B}'$. From $\mathbf{B} \approx \mathbf{B}'$, $\mathbf{B} \triangleright \mathbf{e} \downarrow \mathbf{v} : \sigma$, and lemma 11, we get that $\mathbf{B} \triangleright \mathbf{e} \downarrow \mathbf{v}' : \sigma'$ and $\mathbf{v} : \sigma \approx \mathbf{v} : \sigma'$. Therefore, we can apply the rule E-T-callback to derive $\Omega_0' \xrightarrow{\lambda'} \Omega_1'$ where $\lambda' = \texttt{call f v'!}^{\sigma'}$ and $\Omega_1' \stackrel{\text{def}}{=} \mathbf{C}, \mathbf{H}', \overline{\mathbf{B}'} \cdot \mathbf{B}' \cdot \mathbf{x} \mapsto \mathbf{v}' : \sigma' \triangleright (\mathbf{s}; \mathbf{return};)_{\overline{\mathbf{f}'}; \mathbf{f}}$. From $\vdash \lambda : \mathit{safe}$, we get that $\sigma = \mathbf{S}$. From this and $\mathbf{v} : \sigma \approx \mathbf{v} : \sigma'$, we get that $\mathbf{v} : \sigma = \mathbf{v} : \sigma'$. Hence, $\lambda = \lambda'$ (which follows from $\lambda = \texttt{call f v!}^{\sigma}$, $\lambda = \texttt{call f v'!}^{\sigma'}$, and $\mathbf{v} : \sigma = \mathbf{v} : \sigma'$), and $\Omega_1 \approx \Omega_1'$ (which follows from $\Omega_0 \approx \Omega_0'$ and $\mathbf{v} : \sigma \approx \mathbf{v} : \sigma'$).

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

E-T-ret-internal: Then, $\Omega_0 \stackrel{\text{def}}{=} C, H, \overline{B} \cdot B \triangleright (\text{return};)_{\overline{f'};f}$, $\overline{f'} = \overline{f''}; f'$, $C.\texttt{intfs} \vdash f, f' : \text{internal}$, $\lambda = \epsilon$, and $\Omega_1 = C, H, \overline{B} \triangleright (\text{skip})_{\overline{f'}}$. From $\Omega_0 \approx \Omega_0'$, we get that $\Omega_0' \stackrel{\text{def}}{=} C, H', \overline{B'} \cdot B' \triangleright (\text{return};)_{\overline{f'};f}$ where $H \approx H'$, $\overline{B} \approx \overline{B'}$, 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 $\Omega_1' \stackrel{\text{def}}{=} C, H', \overline{B'} \triangleright (\text{skip})_{\overline{f'}}$. Hence, $\lambda = \lambda'$ (which follows from $\lambda = \epsilon$ and $\lambda' = \epsilon$) and $\Omega_1 \approx \Omega_1'$ (which follows from $H \approx H'$ and $\overline{B} \approx \overline{B'}$).

E-T-retback: Then, $\Omega_0 \stackrel{\text{def}}{=} C, H, \overline{B} \cdot B \triangleright (\text{return};)_{\overline{f'};f}$, $\overline{f'} = \overline{f''}; f'$, $C.\texttt{intfs} \vdash f, f' : \text{internal}$, $\lambda = \texttt{ret?}^{S}$, and $\Omega_1 = C, H, \overline{B} \triangleright (\text{skip})_{\overline{f'}}$. From $\Omega_0 \approx \Omega_0'$, we get that $\Omega_0' \stackrel{\text{def}}{=} C, H', \overline{B'} \cdot B' \triangleright (\text{return};)_{\overline{f'};f}$ where $H \approx H'$, $\overline{B} \approx \overline{B'}$, and $B \approx B'$. Therefore, we can apply the rule E-T-ret-retback to derive $\Omega_0' \xrightarrow{\lambda'} \Omega_1'$ where $\lambda' = \texttt{ret?}^{S}$ and $\Omega_1' \stackrel{\text{def}}{=} C, H', \overline{B'} \triangleright (\text{skip})_{\overline{f'}}$. Hence, $\lambda = \lambda'$ (which follows from $\lambda = \texttt{ret?}^{S}$ and $\lambda' = \texttt{ret?}^{S}$) and $\Omega_1 \approx \Omega_1'$ (which follows from $H \approx H'$ and $\overline{B} \approx \overline{B'}$).

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

E-T-lfence: Then, $\Omega_0 \stackrel{\text{def}}{=} C, H, \overline{B} \triangleright \text{lfence}$, $\lambda = \epsilon$, and $\Omega_1 \stackrel{\text{def}}{=} C, H, \overline{B} \triangleright \text{skip}$. From $\Omega_0 \approx \Omega_0'$, we get that $\Omega_0' \stackrel{\text{def}}{=} C, H', \overline{B'} \triangleright \text{lfence}$ where $H' \approx H$ and $\overline{B} \approx \overline{B'}$. Hence, we can apply the E-T-lfence rule to derive $\Omega_0' \xrightarrow{\lambda'} \Omega_1'$ where $\lambda' = \epsilon$ and $\Omega_1' \stackrel{\text{def}}{=} C, H', \overline{B'} \triangleright \text{skip}$. Hence, $\Omega_1 \approx \Omega_1'$ and $\lambda = \lambda'$ hold.

E-T-cmove-true: Then, $\Omega_0 \stackrel{\text{def}}{=} C, H, \overline{B} \cdot B \triangleright \text{let } x = e_0 \text{ (if } e_1\text{) in } s$, $x \in \texttt{dom}(B)$, $\lambda = \epsilon$, $\Omega_1 \stackrel{\text{def}}{=} C, H, \overline{B} \cdot B \cup x \mapsto v_0 : \sigma \triangleright s$, $B \triangleright e_0 \downarrow v_0 : \sigma_0$, $B \triangleright e_1 \downarrow 0 : \sigma_1$, and $\sigma = \sigma_0 \sqcup \sigma_1$. From $\Omega_0 \approx \Omega_0'$, we have that $\Omega_0' \stackrel{\text{def}}{=} C, H', \overline{B'} \cdot B' \triangleright \text{let } x = e_0 \text{ (if } e_1\text{) in } s$ where $H \approx H'$, $\overline{B} \cdot B \approx \overline{B'} \cdot B'$, 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 11, 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 $\Omega_1 \stackrel{\text{def}}{=} C, H', \overline{B'} \cdot B' \cup x \mapsto v_0' : \sigma' \triangleright s$, 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 $\Omega_1 \stackrel{\text{def}}{=} C, H', \overline{B'} \cdot B' \cup x \mapsto v_0' : \sigma' \triangleright s$, 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 $\Omega_1 \stackrel{\text{def}}{=} C, H', \overline{B'} \cdot B' \cup x \mapsto v' : \sigma' \triangleright s$, $B(x) = v' : \sigma_0'$, 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, $\Omega_0 \stackrel{\text{def}}{=} C, H, \overline{B} \triangleright s; s''$, $C, H, \overline{B} \triangleright s \xrightarrow{\lambda} C_1, H_1, \overline{B_1} \triangleright s_1$, 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 $\overline{B} \approx \overline{B'}$. From the induction hypothesis, $\vdash \lambda : \textit{safe}$, and $\Omega_0 \approx \Omega_0'$, we get that $C, H', \overline{B'} \triangleright s \xrightarrow{\lambda'} C, H_1', \overline{B_1'} \triangleright s_1$ such that $H_1 \approx H_1'$, $\overline{B_1} \approx \overline{B_1'}$, and $\lambda = \lambda'$. Therefore, we can apply the E-L-step rule to derive $\Omega_0' \xrightarrow{\lambda'} \Omega_1'$ where $\Omega_1 \stackrel{\text{def}}{=} C, H_1', \overline{B_1'} \triangleright s_1; s''$. Observe that $\Omega_1 \approx \Omega_1'$ and $\lambda = \lambda'$ hold.

This concludes the proof of our lemma. □

Lemma 8 (Steps of $\longrightarrow$ without observations preserve low-equivalence)**.**

$$\forall \Omega_0, \Omega_0', \lambda, \Omega_1.\ \text{if } \Omega_0 \xrightarrow{\epsilon} \Omega_1, \Omega_0 \approx \Omega_0'$$
$$\text{then } \exists \Omega_1.\ \Omega_0' \xrightarrow{\epsilon'} \Omega_1', \Omega_1 \approx \Omega_1'$$

Proof. Special case of lemma 7 since $\vdash \epsilon : \textit{safe}$ is always satisfied. □

Lemma 9 (Steps of $\longrightarrow$ that agree on observation and low-equivalence cannot disagree on label)**.**

$$\forall \Omega_0, \Omega_0', \lambda, \sigma, \sigma', \Omega_1, \Omega_1'.\ \text{if } \Omega_0 \xrightarrow{\lambda^\sigma} \Omega_1, \Omega_0' \xrightarrow{\lambda^{\sigma'}} \Omega_1', \Omega_0 \approx \Omega_0'$$
$$\text{then } \sigma = \sigma'$$

Proof. By structural induction on the rules defining $\longrightarrow$. 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 10). □

4) Bindings:

Lemma 10 (Low-equivalent bindings produce low-equivalent results - 1)**.**

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

Proof. Let $\mathbf{B}$, $\mathbf{B}'$, $\mathbf{e}$, $\mathbf{v}$, $\mathbf{v}'$, $\boldsymbol{\sigma}$, and $\boldsymbol{\sigma}'$ be such that $\mathbf{B} \approx \mathbf{B}'$, $\mathbf{B} \triangleright \mathbf{e} \downarrow \mathbf{v} : \boldsymbol{\sigma}$, $\mathbf{B}' \triangleright \mathbf{e} \downarrow \mathbf{v}' : \boldsymbol{\sigma}'$. From $\mathbf{B} \approx \mathbf{B}'$, $\mathbf{B} \triangleright \mathbf{e} \downarrow \mathbf{v} : \boldsymbol{\sigma}$, and lemma 11, there are $\mathbf{v}''$, $\boldsymbol{\sigma}''$ such that $\mathbf{B}' \triangleright \mathbf{e} \downarrow \mathbf{v}'' : \boldsymbol{\sigma}''$ and $\mathbf{v} : \boldsymbol{\sigma} \approx \mathbf{v}'' : \boldsymbol{\sigma}''$. Since $\downarrow$ is deterministic, we immediately get that $\mathbf{v}'' : \boldsymbol{\sigma}'' = \mathbf{v}' : \boldsymbol{\sigma}'$. Hence, $\mathbf{v} : \boldsymbol{\sigma} \approx \mathbf{v}' : \boldsymbol{\sigma}'$. □

Lemma 11 (Low-equivalent bindings produce low-equivalent results - 2)**.**

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

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

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

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

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

- **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 $\mathbf{n_1'}$, $\mathbf{n_2}'$, σ_1', σ_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 $\mathbf{B}' \triangleright \mathbf{e} \downarrow \mathbf{v}' : \boldsymbol{\sigma}'$, 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 $\mathbf{v} : \boldsymbol{\sigma} \approx \mathbf{v}' : \boldsymbol{\sigma}'$.
  - $\sigma = \mathbf{U}$**:** 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, $\mathbf{v} : \boldsymbol{\sigma} \approx \mathbf{v}' : \boldsymbol{\sigma}'$ holds.
- **E-T-comparison:** The proof of this case is similar to the case for the E-T-op rule.

This completes the proof. □

5) Non-speculative projection $\cdot\restriction_{nse}$:

Lemma 12 (Properties of non-speculative projection $\cdot\restriction_{nse}$)**.**

$$\begin{aligned}&\forall \mathbf{P}, \boldsymbol{\Sigma}, \boldsymbol{\Sigma}', \lambda, \lambda', \mathbf{n}.\ \text{if } (\mathbf{0}, \boldsymbol{\Omega_0}\,(\mathbf{P})) \overset{\lambda}{\Longrightarrow} (\mathbf{n}, \boldsymbol{\Sigma}), \boldsymbol{\Sigma} \overset{\lambda'}{\rightsquigarrow} \boldsymbol{\Sigma}', \vdash \boldsymbol{\Sigma}' : \mathit{safe}\\ &\qquad \text{then } (\lambda \cdot \lambda')\restriction_{nse} = \lambda\restriction_{nse} \cdot \lambda'\\ &\qquad \text{if } (\mathbf{0}, \boldsymbol{\Omega_0}\,(\mathbf{P})) \overset{\lambda}{\Longrightarrow} (\mathbf{n}, \boldsymbol{\Sigma}), \boldsymbol{\Sigma} \overset{\lambda'}{\rightsquigarrow} \boldsymbol{\Sigma}', \vdash \boldsymbol{\Sigma}' : \mathit{unsafe}\\ &\qquad \text{then } (\lambda \cdot \lambda')\restriction_{nse} = \lambda\restriction_{nse}\end{aligned}$$

Proof. The lemma follows by inspection of the semantics and of the non-speculative projection. □

6) Properties of componenets:

Lemma 13 (Only safe values in componenets)**.**

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

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

B. RSS implies RSNI

Corollary 2 (RSS implies RSNI)**.**

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

Proof. Let $\mathbf{P}$ be an arbitrary program in $\mathbf{T}$ such that $\vdash \mathbf{P} : \text{RSS}$ holds. Let $\mathbf{A}$ be an arbitrary context. Then, there are two cases:

$\vdash \mathbf{A} : atk$: From $\vdash \mathbf{P} : \text{RSS}$, it follows that $\vdash \mathbf{A}[\mathbf{P}] : \text{SS}$. By applying theorem 10, we have $\vdash \mathbf{A}[\mathbf{P}] : \text{SNI}$. Hence, $\vdash \mathbf{A} : atk \Rightarrow \vdash \mathbf{A}[\mathbf{P}] : \text{SNI}$ holds.

$\nvdash \mathbf{A} : atk$: Then, $\vdash \mathbf{A} : atk \Rightarrow \vdash \mathbf{A}[\mathbf{P}] : \text{SNI}$ trivially holds.

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

C. SNI does not imply SS

Theorem 12 (SNI not imply SS)**.**

$$\exists \mathbf{P} \in \mathbf{T}. \vdash \mathbf{P} : \text{SNI} \wedge \nvdash \mathbf{P} : \text{SS}$$

Proof. Consider the following program $\mathbf{P}$:

```
ifz (y < size) then
  let x_a = rd_p 0 + y in
  let x_b = rd_p 4 + x_a in
  let temp=x_b in skip
 else
  let x_a = rd_p 0 + y in
  let x_b = rd_p 4 + x_a in
  let temp=x_b in skip
```

The above program clearly satisfies speculative non-interference for, e.g., $\mathbf{w} = \mathbf{10}$ 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 section C). □

D. RSNI does not imply RSS

Corollary 3 (RSNI not imply RSS)**.**

$$\exists \mathbf{P} \in \mathbf{T}. \vdash \mathbf{P} : \text{RSNI} \wedge \nvdash \mathbf{P} : \text{RSS}$$

Proof. The counter-example provided in theorem 12 can be directly ported to the robust setting (by moving the code in the component and having a context simply calling the component). □

Appendix G
Compiler Criteria and their Implications

Definition 17 (Robust Dynamic Speculative Safety-Preserving Compiler ($RSSP$))**.**

$$\vdash [\![\cdot]\!] : RSSP \stackrel{\text{def}}{=} \forall \mathtt{P}.\ \text{ if } \vdash \mathtt{P} : \text{RSS then } \vdash [\![\mathtt{P} : \text{RSS}]\!]$$

This gets expanded to:

$$\forall \mathtt{P}.\ \text{ if } \ \forall \mathtt{A}. \forall \overline{\lambda^\sigma} \in \mathsf{Beh}(\mathtt{A}\,[\mathtt{P}]). \forall \alpha^\sigma \in \overline{\lambda^\sigma}.\, \sigma \equiv \mathtt{S}$$
$$\text{then } \ \forall \mathbf{A}. \forall \overline{\boldsymbol{\lambda}^{\boldsymbol{\sigma}}} \in \mathbf{Beh}(\mathbf{A}\,[\![\mathtt{P}]\!]). \forall \boldsymbol{\alpha}^{\boldsymbol{\sigma}} \in \overline{\boldsymbol{\lambda}^{\boldsymbol{\sigma}}}.\, \boldsymbol{\sigma} \equiv \mathbf{S}$$

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 Section I-A3 and for slh in Section J-B1.

Trace relation $\approx$

(Trace-Relation)
$$\frac{}{\varnothing \approx \varnothing}$$

(Trace-Relation-Same-Act)
$$\frac{\overline{\lambda^\sigma} \approx \overline{\boldsymbol{\lambda}^{\boldsymbol{\sigma}}} \qquad \alpha^\sigma \overset{A}{\approx} \boldsymbol{\alpha}^{\boldsymbol{\sigma}}}{\overline{\lambda^\sigma} \cdot \alpha^\sigma \approx \overline{\boldsymbol{\lambda}^{\boldsymbol{\sigma}}} \cdot \boldsymbol{\alpha}^{\boldsymbol{\sigma}}}$$

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

(Trace-Relation-Safe-Act)
$$\frac{\overline{\lambda^\sigma} \approx \overline{\boldsymbol{\lambda}^{\boldsymbol{\sigma}}} \qquad \epsilon \overset{A}{\approx} \boldsymbol{\alpha}^{\mathbf{S}}}{\overline{\lambda^\sigma} \approx \overline{\boldsymbol{\lambda}^{\boldsymbol{\sigma}}} \cdot \boldsymbol{\alpha}^{\mathbf{S}}}$$

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

(Trace-Relation-Rollback)
$$\frac{\overline{\lambda^\sigma} \approx \overline{\boldsymbol{\lambda}^{\boldsymbol{\sigma}}} \qquad \epsilon \overset{A}{\approx} \mathtt{rlb}^\sigma}{\overline{\lambda^\sigma} \approx \overline{\boldsymbol{\lambda}^{\boldsymbol{\sigma}}} \cdot \mathtt{rlb}^\sigma}$$

Action relation $\overset{A}{\approx}$

(Action Relation - call)
$$\frac{\mathtt{f} \equiv \mathbf{f} \qquad \mathtt{v} \overset{V}{\approx} \mathbf{v} \qquad \sigma \equiv \boldsymbol{\sigma}}{\mathtt{call\ f\ v}?^\sigma \overset{A}{\approx} \mathtt{call}\ \mathbf{f}\ \mathbf{v}?^{\boldsymbol{\sigma}}}$$

(Action Relation - return)
$$\frac{}{\mathtt{ret}!^S \overset{A}{\approx} \mathtt{ret}!^{\mathbf{S}}}$$

(Action Relation - callback)
$$\frac{\mathtt{f} \equiv \mathbf{f} \qquad \mathtt{v} \overset{V}{\approx} \mathbf{v} \qquad \sigma \equiv \boldsymbol{\sigma}}{\mathtt{call\ f\ v}!^\sigma \overset{A}{\approx} \mathtt{call}\ \mathbf{f}\ \mathbf{v}!^{\boldsymbol{\sigma}}}$$

(Action Relation - returnback)
$$\frac{}{\mathtt{ret}?^S \overset{A}{\approx} \mathtt{ret}?^{\mathbf{S}}}$$

(Action Relation - read)
$$\frac{\mathtt{n} \overset{V}{\approx} \mathbf{n} \qquad \sigma \equiv \boldsymbol{\sigma}}{\mathtt{read(n)}^\sigma \overset{A}{\approx} \mathtt{read}(\mathbf{n})^{\boldsymbol{\sigma}}}$$

(Action Relation - write)
$$\frac{\mathtt{n} \overset{V}{\approx} \mathbf{n} \qquad \sigma \equiv \boldsymbol{\sigma}}{\mathtt{write(n)}^\sigma \overset{A}{\approx} \mathtt{write}(\mathbf{n})^{\boldsymbol{\sigma}}}$$

(Action Relation - write 2)
$$\frac{\mathtt{n} \overset{V}{\approx} \mathbf{n} \qquad \mathtt{v} \overset{V}{\approx} \mathbf{v} \qquad \sigma \equiv \boldsymbol{\sigma}}{\mathtt{write(n} \mapsto \mathtt{v)}^\sigma \overset{A}{\approx} \mathtt{write}(\mathbf{n} \mapsto \mathbf{v})^{\boldsymbol{\sigma}}}$$

(Action Relation - if)
$$\frac{\mathtt{n} \overset{V}{\approx} \mathbf{n} \qquad \sigma \equiv \boldsymbol{\sigma}}{\mathtt{if(n)}^\sigma \overset{A}{\approx} \mathtt{if}(\mathbf{n})^{\boldsymbol{\sigma}}}$$

(Action Relation - epsi alpha)
$$\frac{\boldsymbol{\sigma} \equiv \mathbf{S}}{\epsilon \overset{A}{\approx} \boldsymbol{\alpha}^{\boldsymbol{\sigma}}}$$

(Action Relation - epsi heap)
$$\frac{\boldsymbol{\sigma} \equiv \mathbf{S}}{\epsilon \overset{A}{\approx} \boldsymbol{\delta}^{\boldsymbol{\sigma}}}$$

(Action Relation - rlb)
$$\frac{\boldsymbol{\sigma} \equiv \mathbf{S}}{\epsilon \overset{A}{\approx} \mathtt{rlb}^{\boldsymbol{\sigma}}}$$

Definition 18 (Robust Dynamic Speculative Safety Compilation ($RSSC$))**.**

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

Theorem 13 ($RSSC$ implies $RSSP$)**.** (Proof 3) With the relation $\approx$:

$$\forall [\![\cdot]\!] \text{ if } \vdash [\![\cdot]\!] : RSSC \text{ then } \vdash [\![\cdot]\!] : RSSP$$

Theorem 14 ($RSSP$ implies $RSSC$)**.** With the relation $\approx$:

$$\forall [\![\cdot]\!] \text{ if } \vdash [\![\cdot]\!] : RSSP \text{ then } \vdash [\![\cdot]\!] : RSSC$$

Theorem 15 ($RSSC$ and $RSSP$ are equivalent)**.** With the relation $\approx$:

$$\forall [\![\cdot]\!] \vdash [\![\cdot]\!] : RSSC \iff \vdash [\![\cdot]\!] : RSSP$$

A. Criteria for Insecure Compilers

Definition 19 ($RSNIP$)**.**

$$\vdash [\![\cdot]\!] : RSNIP \stackrel{\text{def}}{=} \forall \mathtt{P}.\ \text{ if } \vdash \mathtt{P} : \text{RSNI then } \vdash [\![\mathtt{P}]\!] : \text{RSNI}$$

Corollary 4 ($\nvdash [\![\cdot]\!] : RSNIP$)**.**

$$\nvdash [\![\cdot]\!] : RSNIP \stackrel{\text{def}}{=} \exists \mathtt{P}.\ \vdash \mathtt{P} : \text{RSNI and } \nvdash [\![\mathtt{P}]\!] : \text{RSNI}$$

APPENDIX H
COMPILER INSECURITY RESULTS

A. Modelling Output-masking SLH

This SLH compiler masks upon read, which is a possible problem since the masking is not right with respect to the predstate, so you may load before an if, not mask because you're not speculating, start speculation and leak.

$$
\begin{aligned}
[\![\mathsf{call\ f\ e}]\!]^s_{\mathcal{J}} &= \mathbf{call\ f}\ [\![\mathsf{e}]\!]^s_{\mathcal{J}} \\
[\![\mathsf{e} := \mathsf{e}']\!]^s_{\mathcal{J}} &= [\![\mathsf{e}]\!]^s_{\mathcal{J}} := [\![\mathsf{e}']\!]^s_{\mathcal{J}} \\
[\![\mathsf{let\ x} = \mathsf{rd\ e\ in\ s}]\!]^s_{\mathcal{J}} &= \mathbf{let\ x = rd}\ [\![\mathsf{e}]\!]^s_{\mathcal{J}}\ \mathbf{in}\ [\![\mathsf{s}]\!]^s_{\mathcal{J}} \\
[\![\mathsf{e} :=_{\mathsf{p}} \mathsf{e}']\!]^s_{\mathcal{J}} &= [\![\mathsf{e}]\!]^s_{\mathcal{J}} + 1 :=_{\mathbf{p}} [\![\mathsf{e}']\!]^s_{\mathcal{J}} \\
[\![\mathsf{ifz\ e\ then\ s\ else\ s}']\!]^s_{\mathcal{J}} &= \mathbf{let\ x_g}{=}[\![\mathsf{e}]\!]^s_{\mathcal{J}}\ \mathbf{in} \\
&\qquad \mathbf{ifz\ x_g\ then\ let\ x = rd_p\ {-1}\ in\ {-1} :=_p x \vee \neg x_g;}\ [\![\mathsf{s}]\!]^s_{\mathcal{J}} \\
&\qquad \mathbf{else\ let\ x = rd_p\ {-1}\ in\ {-1} :=_p x \vee x_g;}\ [\![\mathsf{s}']\!]^s_{\mathcal{J}} \\
[\![\mathsf{let\ x} = \mathsf{rd_p\ e\ in\ s}]\!]^s_{\mathcal{J}} &= \mathbf{let\ x = rd_p}\ [\![\mathsf{e}]\!]^s_{\mathcal{J}} + 1\ \mathbf{in\ let\ pr = rd_p\ {-1}\ in\ let\ x = 0\ (if\ pr)\ in}\ [\![\mathsf{s}]\!]^s_{\mathcal{J}}
\end{aligned}
$$

This compiler breaks the proof of $RSSC$ (eventually) as noted here: Proof 26.
Most importantly, this compiler is not $RSNIP$.

Theorem 16 ($[\![\cdot]\!]^s_{\mathcal{J}}$ is not $RSNIP$)**.**

$$\not\vdash [\![\cdot]\!]^s_{\mathcal{J}} : RSNIP$$

Proof.

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

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

These target traces show that the compiled code is not RSNI in $\mathbf{T}$.

$$
\begin{aligned}
\mathbf{t}' = {} & \mathtt{call\ get\ 8?}^{\mathbf{S}} \cdot \mathtt{read}(1)^{\mathbf{S}} \cdot \mathtt{read}(-(\mathbf{n_a} - 1 + 42 + 1))^{\mathbf{S}} \cdot \\
& \mathtt{read}(-1)^{\mathbf{S}} \cdot \mathtt{if}(1)^{\mathbf{S}} \cdot \mathtt{read}(-1)^{\mathbf{S}} \cdot \mathtt{write}(-1)^{\mathbf{S}} \cdot \\
& \mathtt{read}(\mathbf{n_b} + \mathbf{v_a})^{\mathbf{U}} \cdot \mathtt{rlb}^{\mathbf{S}} \cdot \mathtt{read}(-1)^{\mathbf{S}} \cdot \mathtt{write}(-1)^{\mathbf{S}} \cdot \mathtt{ret!}^{\mathbf{S}} \\
\mathbf{t}' = {} & \mathtt{call\ get\ 8?}^{\mathbf{S}} \cdot \mathtt{read}(1)^{\mathbf{S}} \cdot \mathtt{read}(-(\mathbf{n_a} - 1 + 42 + 1))^{\mathbf{S}} \cdot \\
& \mathtt{read}(-1)^{\mathbf{S}} \cdot \mathtt{if}(1)^{\mathbf{S}} \cdot \mathtt{read}(-1)^{\mathbf{S}} \cdot \mathtt{write}(-1)^{\mathbf{S}} \cdot \\
& \mathtt{read}(\mathbf{n_b} + \mathbf{v'_a})^{\mathbf{U}} \cdot \mathtt{rlb}^{\mathbf{S}} \cdot \mathtt{read}(-1)^{\mathbf{S}} \cdot \mathtt{write}(-1)^{\mathbf{S}} \cdot \mathtt{ret!}^{\mathbf{S}} \\
\mathbf{t}\restriction_{nse} = \mathbf{t}'\restriction_{nse} = {} & \mathtt{call\ get\ 8?}^{\mathbf{S}} \cdot \mathtt{read}(1)^{\mathbf{S}} \cdot \mathtt{read}(-(\mathbf{n_a} - 1 + 42 + 1))^{\mathbf{S}} \cdot \\
& \mathtt{if}(1)^{\mathbf{S}} \cdot \mathtt{read}(-1)^{\mathbf{S}} \cdot \mathtt{write}(-1)^{\mathbf{S}} \cdot \mathtt{ret!}^{\mathbf{S}}
\end{aligned}
$$

□

B. Unsafe Inter-procedural SLH

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

$$
\begin{aligned}
[\![\mathsf{f(x)} \mapsto \mathsf{s; return;}]\!]^s_n &= \mathbf{f(x) \mapsto\ let\ x_{pr}{=}false\ in}\ [\![\mathsf{s}]\!]^s_n\mathbf{; return;} \\
[\![\mathsf{ifz\ e\ then\ s\ else\ s}']\!]^s_n &= \mathbf{let\ x_g}{=}[\![\mathsf{e}]\!]^s\ \mathbf{in} \\
&\qquad \mathbf{ifz\ x_g\ then\ let\ x_{pr}{=}x_{pr} \vee \neg x_g\ in}\ [\![\mathsf{s}]\!]^s_n \\
&\qquad \mathbf{else\ let\ x_{pr}{=}x_{pr} \vee x_g\ in}\ [\![\mathsf{s}']\!]^s_n \\
[\![\mathsf{let\ x} = \mathsf{rd_p\ e\ in\ s}]\!]^s_n &= \mathbf{let\ x = rd_p\ e\ in\ let\ x = 0\ (if\ x_{pr})\ in}\ [\![\mathsf{s}]\!]^s_n
\end{aligned}
$$

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

When proving Lemma 31 (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 29.

Crucially, this compiler is not $RSNIP$.

Theorem 17 ($[\![\cdot]\!]^s_n$ is not $RSNIP$)**.**

$$\not\vdash [\![\cdot]\!]^s_n : RSNIP$$

Proof.

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

$$\begin{aligned}
\mathbf{t'} &= \mathtt{call\ get\ 8?}^{\mathbf{S}} \cdot \mathtt{read}(\mathbf{1})^{\mathbf{S}} \cdot \mathtt{read}(-(\mathbf{n_a + 42}))^{\mathbf{S}} \cdot\\
&\qquad \mathtt{if}(\mathbf{1})^{\mathbf{S}} \cdot \mathtt{read}(\mathbf{n_b + v_a})^{\mathbf{U}} \cdot \mathtt{rlb}^{\mathbf{S}} \cdot \mathtt{ret!}^{\mathbf{S}}\\
\mathbf{t'} &= \mathtt{call\ get\ 8?}^{\mathbf{S}} \cdot \mathtt{read}(\mathbf{1})^{\mathbf{S}} \cdot \mathtt{read}(-(\mathbf{n_a + 42}))^{\mathbf{S}} \cdot\\
&\qquad \mathtt{if}(\mathbf{1})^{\mathbf{S}} \cdot \mathtt{read}(\mathbf{n_b + v'_a})^{\mathbf{U}} \cdot \mathtt{rlb}^{\mathbf{S}} \cdot \mathtt{ret!}^{\mathbf{S}}\\
\mathbf{t}\restriction_{nse} = \mathbf{t'}\restriction_{nse} &= \mathtt{call\ get\ 8?}^{\mathbf{S}} \cdot \mathtt{read}(\mathbf{1})^{\mathbf{S}} \cdot \mathtt{read}(-(\mathbf{n_a + 42}))^{\mathbf{S}} \cdot\\
&\qquad \mathtt{if}(\mathbf{1})^{\mathbf{S}} \cdot \mathtt{ret!}^{\mathbf{S}}
\end{aligned}$$

□

APPENDIX I
THE LFENCE COMPILER $[\![\cdot]\!]^f$

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 Section B-B1). 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.

$$[\![\mathsf{H};\overline{\mathsf{F}};\overline{\mathsf{I}}]\!]^f = [\![\mathsf{H}]\!]^f;[\![\overline{\mathsf{F}}]\!]^f;[\![\overline{\mathsf{I}}]\!]^f$$

$$[\![\varnothing]\!]^f = \varnothing$$
$$[\![\overline{\mathsf{I}}\cdot \mathsf{f}]\!]^f = [\![\overline{\mathsf{I}}]\!]^f \cdot \mathbf{f}$$

$$[\![\mathsf{H};-\mathsf{n}\mapsto \mathsf{v}:\mathsf{U}]\!]^f = [\![\mathsf{H}]\!]^f;-[\![\mathsf{n}]\!]^f \mapsto [\![\mathsf{v}]\!]^f : \mathbf{U}$$

$$\begin{aligned}
[\![\mathsf{f}(\mathsf{x})\mapsto \mathsf{s};\mathbf{return};]\!]^f &= \mathbf{f}(\mathbf{x})\mapsto [\![\mathsf{s}]\!]^f;\mathbf{return};\\
[\![\mathsf{s};\mathsf{s}']\!]^f &= [\![\mathsf{s}]\!]^f;[\![\mathsf{s}']\!]^f\\
[\![\mathsf{skip}]\!]^f &= \mathbf{skip}\\
[\![\mathsf{let}\ \mathsf{x}=\mathsf{e}\ \mathsf{in}\ \mathsf{s}]\!]^f &= \mathbf{let}\ \mathbf{x}=[\![\mathsf{e}]\!]^f\ \mathbf{in}\ [\![\mathsf{s}]\!]^f\\
[\![\mathsf{ifz}\ \mathsf{e}\ \mathsf{then}\ \mathsf{s}\ \mathsf{else}\ \mathsf{s}']\!]^f &= \mathbf{ifz}\ [\![\mathsf{e}]\!]^f\ \mathbf{then}\ \{\mathbf{lfence};[\![\mathsf{s}]\!]^f\}\ \mathbf{else}\ \{\mathbf{lfence};[\![\mathsf{s}']\!]^f\}\\
[\![\mathsf{call}\ \mathsf{f}\ \mathsf{e}]\!]^f &= \mathbf{call}\ \mathbf{f}\ [\![\mathsf{e}]\!]^f\\
[\![\mathsf{e}:=\mathsf{e}']\!]^f &= [\![\mathsf{e}]\!]^f := [\![\mathsf{e}']\!]^f\\
[\![\mathsf{let}\ \mathsf{x}=\mathsf{rd}\ \mathsf{e}\ \mathsf{in}\ \mathsf{s}]\!]^f &= \mathbf{let}\ \mathbf{x}=\mathbf{rd}\ [\![\mathsf{e}]\!]^f\ \mathbf{in}\ [\![\mathsf{s}]\!]^f\\
[\![\mathsf{e}:=_{\mathsf{p}}\mathsf{e}']\!]^f &= [\![\mathsf{e}]\!]^f :=_{\mathbf{p}} [\![\mathsf{e}']\!]^f\\
[\![\mathsf{let}\ \mathsf{x}=\mathsf{rd}_{\mathsf{p}}\ \mathsf{e}\ \mathsf{in}\ \mathsf{s}]\!]^f &= \mathbf{let}\ \mathbf{x}=\mathbf{rd_p}\ [\![\mathsf{e}]\!]^f\ \mathbf{in}\ [\![\mathsf{s}]\!]^f
\end{aligned}$$

$$\begin{aligned}
[\![\mathsf{n}]\!]^f &= \mathbf{n}\\
[\![\mathsf{e}\oplus\mathsf{e}']\!]^f &= [\![\mathsf{e}]\!]^f \oplus [\![\mathsf{e}']\!]^f\\
[\![\mathsf{e}\otimes\mathsf{e}']\!]^f &= [\![\mathsf{e}]\!]^f \otimes [\![\mathsf{e}']\!]^f
\end{aligned}$$

Theorem 18 (The lfence compiler is $RSSC$)**.** (Proof 6)

$$\vdash [\![\cdot]\!]^f : RSSC$$

Theorem 19 (All lfence-compiled programs are $RSSC$)**.** (Proof 7)

$$\forall \mathsf{P}.\vdash [\![\mathsf{P}]\!]^f : \mathrm{RSS}$$

A. 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 (Section I-A1) and a trace-based backtranslation (analogous to the one of the SLH compiler).

1) Context-based Backtranslation:

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

$$\begin{aligned}
\langle\!\langle \varnothing \rangle\!\rangle_c^f &= \varnothing\\
\langle\!\langle \mathbf{H};\mathbf{n}\mapsto \mathbf{v}:\boldsymbol{\sigma} \rangle\!\rangle_c^f &= \langle\!\langle \mathbf{H} \rangle\!\rangle_c^f;\langle\!\langle \mathbf{n} \rangle\!\rangle_c^f \mapsto \langle\!\langle \mathbf{v} \rangle\!\rangle_c^f : \langle\!\langle \boldsymbol{\sigma} \rangle\!\rangle_c^f
\end{aligned}$$

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

$$\langle\!\langle \sigma \rangle\!\rangle_c^f = \sigma$$

$$\begin{aligned}
\langle\!\langle \mathbf{n} \rangle\!\rangle_c^f &= \mathsf{n} \\
\langle\!\langle \mathbf{e} \oplus \mathbf{e}' \rangle\!\rangle_c^f &= \langle\!\langle \mathbf{e} \rangle\!\rangle_c^f \oplus \langle\!\langle \mathbf{e}' \rangle\!\rangle_c^f \\
\langle\!\langle \mathbf{e} \otimes \mathbf{e}' \rangle\!\rangle_c^f &= \langle\!\langle \mathbf{e} \rangle\!\rangle_c^f \otimes \langle\!\langle \mathbf{e}' \rangle\!\rangle_c^f
\end{aligned}$$

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

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 $RSSC$.

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 20 (Correctness of the Backtranslation for lfence)**.** (Proof 8)

$$\begin{aligned}
&\text{if} && \mathbf{A}\left[[\![\mathsf{P}]\!]^f\right] \rightsquigarrow \overline{\lambda^\sigma} \\
&\text{then} && \langle\!\langle \mathbf{A} \rangle\!\rangle_c^f[\mathsf{P}] \rightsquigarrow \overline{\lambda^\sigma} \\
&\text{and} && \overline{\lambda^\sigma} \approx \overline{\lambda^\sigma}
\end{aligned}$$

Theorem 21 (Generalised Backward Simulation for lfence)**.** (Proof 9)

$$\begin{aligned}
&\text{if} && \text{if } \mathsf{f} \in \overline{\mathsf{f}''} \text{ then } \mathbf{s} = [\![\mathsf{s}]\!]^f \text{ else } \mathsf{s} = \langle\!\langle \mathbf{s} \rangle\!\rangle_c^f \\
&\text{and} && \text{if } \mathsf{f}' \in \overline{\mathsf{f}''} \text{ then } \mathbf{s}' = [\![\mathsf{s}']\!]^f \text{ else } \mathsf{s}' = \langle\!\langle \mathbf{s}' \rangle\!\rangle_c^f \\
&\text{and} && \mathbf{\Sigma} = \mathbf{w}(\mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \triangleright (\mathbf{s}; \mathbf{s}'')_{\overline{\mathbf{f}} \cdot \mathbf{f}}, \bot, \mathbf{S}) \\
&\text{and} && \mathbf{\Sigma}' = \mathbf{w}(\mathbf{C}, \mathbf{H}', \overline{\mathbf{B}'} \triangleright (\mathbf{s}'; \mathbf{s}'')_{\overline{\mathbf{f}'} \cdot \mathbf{f}'}, \bot, \mathbf{S}) \\
&\text{and} && (\mathbf{n}, \mathbf{\Sigma}) \xRightarrow{\overline{\lambda^\sigma}} (\mathbf{n}', \mathbf{\Sigma}') \\
&\text{and} && \Omega \overset{S}{\approx}_{\overline{\mathsf{f}''}} \mathbf{\Sigma} \\
&\text{then} && \Omega = \mathsf{C}, \mathsf{H}, \overline{\mathsf{B}} \triangleright (\mathsf{s}; \mathsf{s}'')_{\overline{\mathsf{f}} \cdot \mathsf{f}} \xRightarrow{\overline{\lambda^\sigma}} \mathsf{C}, \mathsf{H}', \overline{\mathsf{B}'} \triangleright (\mathsf{s}'; \mathsf{s}'')_{\overline{\mathsf{f}'} \cdot \mathsf{f}'} = \Omega' \\
&\text{and} && \overline{\lambda^\sigma} \approx \overline{\lambda^\sigma} \\
&\text{and} && \Omega' \overset{S}{\approx}_{\overline{\mathsf{f}''}} \mathbf{\Sigma}'
\end{aligned}$$

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.

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

$$\frac{}{\varnothing \overset{H}{\approx} \varnothing}\ \text{(Heap - base)} \qquad \frac{H \overset{H}{\approx} \mathbf{H} \quad z \overset{V}{\approx} \mathbf{z} \quad v \overset{V}{\approx} \mathbf{v} \quad \sigma \equiv \boldsymbol{\sigma}}{H; z \mapsto v : \sigma \overset{H}{\approx} \mathbf{H}; \mathbf{z} \mapsto \mathbf{v} : \boldsymbol{\sigma}}\ \text{(Heap - ind)}$$

$$\frac{H \overset{H}{\approx} \mathbf{H} \quad H' \overset{H}{\approx} \mathbf{H}' \quad v \overset{V}{\approx} \mathbf{v} \quad \sigma \equiv \boldsymbol{\sigma}}{H; 0 \mapsto v : \sigma; H' \overset{H}{\approx} \mathbf{H}; \mathbf{0} \mapsto \mathbf{v} : \boldsymbol{\sigma}}\ \text{(Heap - start)} \qquad \frac{z \equiv \mathbf{z} \quad z \in \mathbb{Z}}{z \overset{V}{\approx} \mathbf{z}}\ \text{(Value - num)}$$

Bindings are related if they map same-named variables to related values. Components are related according to a list of function names $\overline{f}$ 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.

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

$$\frac{}{\varnothing \overset{B}{\approx} \varnothing}\ \text{(Binding - base)} \qquad \frac{B \overset{B}{\approx} \mathbf{B} \quad v \overset{V}{\approx} \mathbf{v} \quad \sigma \equiv \boldsymbol{\sigma}}{B \cdot x \mapsto v : \sigma \overset{B}{\approx} \mathbf{B} \cdot \mathbf{x} \mapsto \mathbf{v} : \boldsymbol{\sigma}}\ \text{(Binding - ind)}$$

$$\frac{\overline{B} \overset{B}{\approx} \overline{\mathbf{B}} \quad B \overset{B}{\approx} \mathbf{B}}{\overline{B} \cdot B \overset{B}{\approx} \overline{\mathbf{B}} \cdot \mathbf{B}}\ \text{(Bindings)}$$

$$\frac{\begin{array}{c}\forall f \in \overline{f}.\ \text{if } f(x) \mapsto s \in \overline{F} \text{ then } \mathbf{f(x)} \mapsto [\![s]\!]^f \in \overline{\mathbf{F}} \\ \forall \mathbf{f(x)} \mapsto \mathbf{s} \in \overline{\mathbf{F}} \text{ if } \mathbf{f} \notin \overline{\mathbf{f}} \text{ then } f(x) \mapsto \langle\!\langle s \rangle\!\rangle_c^f \in \overline{F} \\ \overline{I} \equiv \overline{\mathbf{I}}\end{array}}{\overline{F}; \overline{I} \overset{C}{\approx}_{\overline{f}} \overline{\mathbf{F}}; \overline{\mathbf{I}}}\ \text{(Components)}$$

$$\frac{\overline{B} \overset{B}{\approx} \overline{\mathbf{B}} \quad H \overset{H}{\approx} \mathbf{H} \quad \overline{f} \equiv \overline{\mathbf{f}} \quad C \overset{C}{\approx}_{\overline{f'''}} \mathbf{C}}{C, H, \overline{B} \triangleright (s)_{\overline{f}} \overset{S}{\approx}_{\overline{f'''}} \mathbf{w}, (\mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \triangleright (\mathbf{s})_{\overline{\mathbf{f}}}, \bot, \mathbf{S})}\ \text{(States)}$$

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 14 (Forward Simulation for Expressions in lfence)**.** (Proof 11)

$$\begin{array}{rl} \text{if} & B \triangleright e \downarrow v : \sigma \\ \text{and} & B \overset{B}{\approx} \mathbf{B} \\ \text{and} & \sigma \equiv \boldsymbol{\sigma} \\ \text{then} & \mathbf{B} \triangleright [\![e]\!]^f \downarrow [\![v]\!]^f : \boldsymbol{\sigma} \end{array}$$

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 $\mathbf{s}''$ that are not touched by the reductions in order to use this result in a general setting.

Lemma 15 (Forward Simulation for Compiled Statements in lfence)**.** (Proof 12)

$$\begin{array}{rl} \text{if} & \Omega = C, H, \overline{B} \triangleright (s; s'')_{\overline{f}} \xrightarrow{\lambda^\sigma} C, H', \overline{B'} \triangleright (s'; s'')_{\overline{f'}} = \Omega' \\ \text{and} & \Omega \overset{S}{\approx}_{\overline{f'''}} \boldsymbol{\Sigma} \\ \text{and} & \boldsymbol{\Sigma} = \mathbf{w}, (\mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \triangleright ([\![s]\!]^f; \mathbf{s}'')_{\overline{\mathbf{f}}}, \bot, \mathbf{S}) \\ \text{and} & \boldsymbol{\Sigma}' = \mathbf{w}(\mathbf{C}, \mathbf{H}', \overline{\mathbf{B}'} \triangleright ([\![s']\!]^f; \mathbf{s}'')_{\overline{\mathbf{f}'}}, \bot, \mathbf{S}) \\ \text{then} & (\mathbf{n}, \boldsymbol{\Sigma}) \xRightarrow{\overline{\lambda^\sigma}} (\mathbf{n}', \boldsymbol{\Sigma}') \\ \text{and} & \lambda^\sigma \approx \overline{\boldsymbol{\lambda^\sigma}} \qquad \text{(using the trace relation!)} \\ \text{and} & \Omega' \overset{S}{\approx}_{\overline{f'''}} \boldsymbol{\Sigma}' \end{array}$$

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 22 (Backward Simulation for Compiled Steps in lfence)**.** (Proof 10)

$$\begin{array}{rl} \text{if} & \boldsymbol{\Sigma} = \mathbf{w}, (\mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \triangleright ([\![s]\!]^f; \mathbf{s}'')_{\overline{\mathbf{f}}}, \bot, \mathbf{S}) \\ \text{and} & \boldsymbol{\Sigma}' = \mathbf{w}(\mathbf{C}, \mathbf{H}', \overline{\mathbf{B}'} \triangleright ([\![s']\!]^f; \mathbf{s}'')_{\overline{\mathbf{f}'}}, \bot, \mathbf{S}) \end{array}$$

$$
\begin{aligned}
&\text{and} && (\mathbf{n}, \mathbf{\Sigma}) \xRightarrow{\overline{\lambda^{\sigma}}} (\mathbf{n}', \mathbf{\Sigma}') \\
&\text{and} && \Omega \overset{S}{\approx}_{\overline{\mathbf{f}''}} \mathbf{\Sigma} \\
&\text{then} && \Omega = \mathsf{C}, \mathsf{H}, \overline{\mathsf{B}} \triangleright (\mathsf{s}; \mathsf{s}'')_{\overline{\mathsf{f}}} \xrightarrow{\lambda^{\sigma}} \mathsf{C}, \mathsf{H}', \overline{\mathsf{B}'} \triangleright (\mathsf{s}'; \mathsf{s}'')_{\overline{\mathsf{f}'}} = \Omega' \\
&\text{and} && \lambda^{\sigma} \approx \overline{\boldsymbol{\lambda^{\sigma}}} \qquad (\text{ using the trace relation!}) \\
&\text{and} && \Omega' \overset{S}{\approx}_{\overline{\mathbf{f}''}} \mathbf{\Sigma}'
\end{aligned}
$$

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 $\overset{V}{\approx}$ 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 16 (Backward Simulation for Backtranslated Expressions in lfence)**.** (Proof 13)

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

Lemma 17 (Backward Simulation for Backtranslated Statements)**.** (Proof 14)

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

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 21.

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 18 (Initial States are Related)**.** (Proof 15)

$$
\begin{aligned}
&\forall \mathsf{P}, \forall \overline{\mathsf{f}} = \mathtt{dom}\,(\mathsf{P}.\mathsf{F}), \forall \mathbf{A} \\
&\Omega_0 \left(\langle\!\langle \mathbf{A} \rangle\!\rangle_c^f [\mathsf{P}] \right) \overset{S}{\approx}_{\overline{\mathsf{f}}} \mathbf{\Omega_0} \left(\mathbf{A} [\![\mathsf{P}]\!]^f \right)
\end{aligned}
$$

Lemma 19 (A Value is Related to its Compilation for lfence)**.** (Proof 16)

$$
\mathsf{v} \overset{V}{\approx} [\![\mathsf{v}]\!]^f
$$

Lemma 20 (A Heap is Related to its Compilation for lfence)**.** (Proof 17)

$$
\mathsf{H} \overset{H}{\approx} [\![\mathsf{H}]\!]^f
$$

Lemma 21 (A Value is Related to its Backtranslation)**.** (Proof 18)

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

Lemma 22 (A Taint is Related to its Backtranslation)**.** (Proof 19)

$$
\langle\!\langle \boldsymbol{\sigma} \rangle\!\rangle_c^f \equiv \boldsymbol{\sigma}
$$

Lemma 23 (A Heap is Related to its Backtranslation)**.** (Proof 20)

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

B. A Stronger Property for $[\![\cdot]\!]^f$

Theorem 23 (The lfence compiler is $RSSC$ with more leaks)**.** (Proof 21)

$$\vdash [\![\cdot]\!]^f : RSSC$$

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

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

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

Appendix J
The SLH Compiler $[\![\cdot]\!]^s$

$$[\![\mathsf{H};\overline{\mathsf{F}};\overline{\mathsf{I}}]\!]^s = [\![\mathsf{H}]\!]^s \cup (-1 \mapsto \mathbf{false} : \mathbf{S});\ [\![\overline{\mathsf{F}}]\!]^s;\ [\![\overline{\mathsf{I}}]\!]^s$$

$$[\![\varnothing]\!]^s = \varnothing$$
$$[\![\overline{\mathsf{I}} \cdot \mathsf{f}]\!]^s = [\![\overline{\mathsf{I}}]\!]^s \cdot \mathbf{f}$$

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

$$[\![\mathsf{f}(\mathsf{x}) \mapsto \mathsf{s}; \mathsf{return};]\!]^s = \mathbf{f}(\mathbf{x}) \mapsto [\![\mathsf{s}]\!]^s; \mathbf{return};$$

$$[\![\mathsf{n}]\!]^s = \mathbf{n}$$
$$[\![\mathsf{e} \oplus \mathsf{e}']\!]^s = [\![\mathsf{e}]\!]^s \oplus [\![\mathsf{e}']\!]^s$$
$$[\![\mathsf{e} \otimes \mathsf{e}']\!]^s = [\![\mathsf{e}]\!]^s \otimes [\![\mathsf{e}']\!]^s$$

$$[\![\mathsf{s}; \mathsf{s}']\!]^s = [\![\mathsf{s}]\!]^s; [\![\mathsf{s}']\!]^s$$
$$[\![\mathsf{skip}]\!]^s = \mathbf{skip}$$
$$[\![\mathsf{let\ x = e\ in\ s}]\!]^s = \mathbf{let\ x} = [\![\mathsf{e}]\!]^s\ \mathbf{in}\ [\![\mathsf{s}]\!]^s$$
$$[\![\mathsf{call\ f\ e}]\!]^s = \mathbf{let\ x_f} = [\![\mathsf{e}]\!]^s\ \mathbf{in}\ \mathbf{let\ pr} = \mathbf{rd_p}\ -1\ \mathbf{in}\ \mathbf{let\ x_f} = \mathbf{0}\ (\mathbf{if\ pr})\ \mathbf{in\ call\ f\ x_f}$$
$$[\![\mathsf{e} := \mathsf{e}']\!]^s = \mathbf{let\ x_f} = [\![\mathsf{e}]\!]^s\ \mathbf{in}\ \mathbf{let\ x'_f} = [\![\mathsf{e}']\!]^s\ \mathbf{in}\ \mathbf{let\ pr} = \mathbf{rd_p}\ -1\ \mathbf{in}\ \mathbf{let\ x_f} = \mathbf{0}\ (\mathbf{if\ pr})\ \mathbf{in}\ \mathbf{let\ x'_f} = \mathbf{0}\ (\mathbf{if\ pr})\ \mathbf{in}\ \mathbf{x_f} := \mathbf{x'_f}$$
$$[\![\mathsf{let\ x} = \mathsf{rd\ e\ in\ s}]\!]^s = \mathbf{let\ x} = \mathbf{rd}\ [\![\mathsf{e}]\!]^s\ \mathbf{in}\ \mathbf{let\ pr} = \mathbf{rd_p}\ -1\ \mathbf{in}\ \mathbf{let\ x} = \mathbf{0}\ (\mathbf{if\ pr})\ \mathbf{in}\ [\![\mathsf{s}]\!]^s$$
$$[\![\mathsf{e} :=_{\mathsf{p}} \mathsf{e}']\!]^s = \mathbf{let\ x_f} = [\![\mathsf{e}]\!]^s + 1\ \mathbf{in}\ \mathbf{let\ x'_f} = [\![\mathsf{e}']\!]^s\ \mathbf{in}\ \mathbf{let\ pr} = \mathbf{rd_p}\ -1\ \mathbf{in}\ \mathbf{let\ x_f} = \mathbf{0}\ (\mathbf{if\ pr})\ \mathbf{in}\ \mathbf{let\ x'_f} = \mathbf{0}\ (\mathbf{if\ pr})\ \mathbf{in}\ \mathbf{x_f} :=_{\mathbf{p}} \mathbf{x'_f}$$
$$[\![\mathsf{ifz\ e\ then\ s\ else\ s}']\!]^s = \mathbf{let\ x_g} = [\![\mathsf{e}]\!]^s\ \mathbf{in}\ \mathbf{let\ pr} = \mathbf{rd_p}\ -1\ \mathbf{in}\ \mathbf{let\ x_g} = \mathbf{0}\ (\mathbf{if\ pr})\ \mathbf{in}\ \mathbf{ifz\ x_g}\ \mathbf{then\ let\ x} = \mathbf{rd_p}\ -1\ \mathbf{in}\ -1 :=_{\mathbf{p}} \mathbf{x} \vee \neg \mathbf{x_g}; [\![\mathsf{s}]\!]^s\ \mathbf{else\ let\ x} = \mathbf{rd_p}\ -1\ \mathbf{in}\ -1 :=_{\mathbf{p}} \mathbf{x} \vee \mathbf{x_g}; [\![\mathsf{s}']\!]^s$$

$$\begin{aligned}[\![\mathsf{let\ x = rd_p\ e\ in\ s}]\!]^s = \ & \mathbf{let\ x}{=}[\![\mathsf{e}]\!]^s{+}\mathbf{1\ in}\\ & \mathbf{let\ pr} = \mathbf{rd_p}\ {-}\mathbf{1\ in}\\ & \mathbf{let\ x} = \mathbf{0\ (if\ pr)\ in}\\ & \mathbf{let\ x} = \mathbf{rd_p\ x\ in}\ [\![\mathsf{s}]\!]^s\end{aligned}$$

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 24 (Our SLH compiler is $RSSC$)**.** (Proof 37)

$$\vdash [\![\cdot]\!]^s : RSSC$$

A. Inter-procedural SLH

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

$$\begin{aligned}[\![\mathsf{f(x) \mapsto s; return;}]\!]^s_n = \ & \mathbf{f(x) \mapsto\ lfence;}\\ & \mathbf{let\ x_{pr}{=}false\ in}\ [\![\mathsf{s}]\!]^s_n\mathbf{;\ return;}\\ [\![\mathsf{ifz\ e\ then\ s\ else\ s'}]\!]^s_n = \ & \mathbf{let\ x_g{=}}[\![\mathsf{e}]\!]^s\ \mathbf{in}\\ & \mathbf{ifz\ x_g\ then\ let\ x_{pr}{=}x_{pr} \vee \neg x_g\ in}\ [\![\mathsf{s}]\!]^s_n\\ & \mathbf{else\ let\ x_{pr}{=}x_{pr} \vee x_g\ in}\ [\![\mathsf{s'}]\!]^s_n\\ [\![\mathsf{let\ x = rd_p\ e\ in\ s}]\!]^s_n = \ & \mathbf{let\ x = rd_p\ e\ in\ let\ x = 0\ (if\ x_{pr})\ in}\ [\![\mathsf{s}]\!]^s_n\end{aligned}$$

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

When proving Lemma 31 (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 29.

Theorem 25 (Our inter-procedural SLH compiler is $RSSC$)**.** (Proof 38)

$$\vdash [\![\cdot]\!]^s_n : RSSC$$

B. Backtranslation

We can use the same context-based backtranslation of Section I-A1, what changes are the cross-language relations.

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 $\mathbf{\Sigma}$ that is a stack of operational states $\overline{\mathbf{\Omega}}$, we require that the first element of the stack is in the strong state relation ($\overset{S}{\approx}$) with a source state Ω. All other states need to be related at a weaker state relation ($\overset{S}{\lesssim}$), which only enforces that all bindings map variables to S values. That is, a target an 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 $\overline{\mathbf{D}}$ 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 $\overset{B}{\approx}$

$$\frac{}{\varnothing \overset{B}{\approx}^{\varnothing} \varnothing}\ (\text{Binding - base})$$

$$\frac{\mathsf{B} \overset{B}{\approx}^{\mathbf{D}} \mathbf{B} \qquad \mathsf{v} \overset{V}{\approx} \mathbf{v} \qquad \sigma \equiv \boldsymbol{\sigma}}{\mathsf{B} \cdot \mathsf{x} \mapsto \mathsf{v} : \sigma \overset{B}{\approx}^{\mathbf{D}} \mathbf{B} \cdot \mathbf{x} \mapsto \mathbf{v} : \boldsymbol{\sigma}}\ (\text{Binding - ind})$$

$$\frac{\mathsf{B} \overset{B}{\approx}^{\mathbf{D}} \mathbf{B} \qquad \mathbf{x} \in \mathbf{D}}{\mathsf{B} \overset{B}{\approx}^{\mathbf{D}} \mathbf{B} \cdot \mathbf{x} \mapsto \mathbf{v} : \boldsymbol{\sigma}}\ (\text{Binding - ind})$$

$$\frac{\overline{\mathsf{B}} \overset{B}{\approx}^{\overline{\mathbf{D}}} \overline{\mathbf{B}} \qquad \mathsf{B} \overset{B}{\approx}^{\mathbf{D}} \mathbf{B}}{\overline{\mathsf{B}} \cdot \mathsf{B} \overset{B}{\approx}^{\overline{\mathbf{D}} \cdot \mathbf{D}} \overline{\mathbf{B}} \cdot \mathbf{B}}\ (\text{Bindings})$$

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 $\overset{H}{\lesssim}$ for private heaps (whose domain is negative integers) to account for this.

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

$$\frac{}{\varnothing \overset{H}{\lesssim} \varnothing}\ (\text{Heap - base})$$

$$\frac{\mathsf{H} \overset{H}{\lesssim} \mathbf{H} \qquad \mathsf{z} - 1 \overset{V}{\lesssim} \mathbf{z} \qquad \mathsf{v} \overset{V}{\lesssim} \mathbf{v} \qquad \sigma \equiv \boldsymbol{\sigma}}{\mathsf{H}; \mathsf{z} \mapsto \mathsf{v} : \sigma \overset{H}{\lesssim} \mathbf{H}; \mathbf{z} \mapsto \mathbf{v} : \boldsymbol{\sigma}}\ (\text{Heap - negative})$$

$$\frac{\mathsf{H} \overset{H}{\lesssim} \mathbf{H} \qquad \mathsf{H}' \overset{H}{\approx} \mathbf{H}' \qquad \mathsf{v} \overset{V}{\approx} \mathbf{v} \qquad \sigma \equiv \boldsymbol{\sigma}}{\mathsf{H}; 0 \mapsto \mathsf{v} : \sigma; \mathsf{H}' \overset{H}{\approx} \mathbf{H}; -1 \mapsto \mathbf{false} : \mathbf{S}; \mathbf{0} \mapsto \mathbf{v} : \boldsymbol{\sigma}; \mathbf{H}'}\ (\text{Heap - start})$$

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

(States relation)
$$\frac{\Omega \overset{S}{\approx}_{\overline{f}}^{\overline{D}} (w, (\Omega, \bot, S)) \quad \forall \Omega_s \in \overline{\Omega},\ \Omega \overset{S}{\approx}_{\overline{f}} \Omega_s}{\Omega \overset{S}{\approx}_{\overline{f}}^{\overline{D}} (w, (\Omega, \bot, S) \cdot \overline{(\Omega, W, U)})}$$

(Base States)
$$\frac{\overline{B} \overset{B}{\approx}^{\overline{D}} \overline{B} \quad H \overset{H}{\approx} H \quad \overline{f} \equiv \overline{f} \quad C \overset{C}{\approx}_{\overline{f''}} C}{C, H, \overline{B} \triangleright (s)_{\overline{f}} \overset{S}{\approx}_{\overline{f''}}^{\overline{D}} w, (C, H, \overline{B} \triangleright (s)_{\overline{f}}, \bot, S)}$$

(Single state relation - ctx)
$$\frac{(\text{ if } f \notin \overline{f'} \text{ then } \vdash B : \overset{B}{\approx}) \quad C \overset{C}{\approx}_{\overline{f'}} C \quad \vdash H : \overset{H}{\approx}}{C, H, \overline{B} \triangleright (s)_{\overline{f}} \overset{S}{\approx}_{\overline{f'}} C, H, \overline{B} \cdot B \triangleright (s)_{\overline{f} \cdot f}}$$

(Heap relation same)
$$\frac{\forall n \mapsto v : \sigma \in H \text{ if } n \geq 0 \text{ then } \sigma = S \quad H(-1) = \text{true} : S}{\vdash H : \overset{H}{\approx}}$$

(Target Bindings ok base)
$$\frac{}{\vdash \varnothing : \overset{B}{\approx}}$$

(Target Bindings ok sing)
$$\frac{\vdash B : \overset{B}{\approx}}{\vdash B \cdot x \mapsto v : S : \overset{B}{\approx}}$$

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

2) Relation for Inter-procedural SLH: Binding relation $\overset{B}{\propto}$ State relation $\overset{S}{\propto}$

(States relation)
$$\frac{\Omega \overset{S}{\approx}_{\overline{f}}^{\overline{D}} (w, (\Omega, \bot, S)) \quad \forall \Omega_s \in \overline{\Omega},\ \Omega \overset{S}{\propto}_{\overline{f}} \Omega_s}{\Omega \overset{S}{\propto}_{\overline{f}}^{\overline{D}} (w, (\Omega, \bot, S) \cdot \overline{(\Omega, W, U)})}$$

(Heap relation same)
$$\frac{\forall n \mapsto v : \sigma \in H \text{ if } n \geq 0 \text{ then } \sigma = S}{\vdash H : \overset{H}{\propto}}$$

(Single state relation - ctx)
$$\frac{(\text{ if } f \notin \overline{f'} \text{ then } \vdash B : \overset{B}{\propto}) \quad C \overset{C}{\approx}_{\overline{f'}} C \quad \vdash H : \overset{H}{\propto}}{C, H, \overline{B} \triangleright (s)_{\overline{f}} \overset{S}{\propto}_{\overline{f'}} C, H, \overline{B} \cdot B \triangleright (s)_{\overline{f} \cdot f}}$$

(Target Bindings ok base)
$$\frac{}{\vdash pr \mapsto 0 : S : \overset{B}{\propto}}$$

(Target Bindings ok sing)
$$\frac{\vdash B : \overset{B}{\propto}}{\vdash B \cdot x \mapsto v : S : \overset{B}{\propto}}$$

Speculating states are related $\overset{S}{\propto}$ 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 24 (Initial States are Related for SLH)**.** (Proof 22)
$$\forall P, \forall \overline{f} = \mathtt{dom}(P.F), \forall A$$
$$\Omega_0 \left(\langle\!\langle A \rangle\!\rangle_c^f [P] \right) \overset{S}{\approx}_{\overline{f}} \Omega_0 (A[\![P]\!]^s)$$

Lemma 25 (A Value is Related to its Compilation for SLH)**.** (Proof 23)
$$v \overset{V}{\approx} [\![v]\!]^s$$

Lemma 26 (A Heap is Related to its Compilation for SLH)**.** (Proof 24)
$$H \overset{H}{\approx} [\![H]\!]^s$$

Lemma 27 (Forward Simulation for Expressions in SLH)**.** (Proof 25)
$$\begin{aligned} &\text{if} && B \triangleright e \downarrow v : \sigma \\ &\text{and} && B \overset{B}{\approx} B \\ &\text{and} && \sigma \equiv \sigma \\ &\text{then} && B \triangleright [\![e]\!]^s \downarrow [\![v]\!]^s : \sigma \end{aligned}$$

Lemma 28 (Forward Simulation for Compiled Statements in SLH)**.** (Proof 26)
$$\begin{aligned} &\text{if} && \Omega = C, H, \overline{B} \triangleright (s; s'')_{\overline{f}} \xrightarrow{\lambda^\sigma} C, H', \overline{B'} \triangleright (s'; s'')_{\overline{f'}} = \Omega' \\ &\text{and} && \Omega \overset{S}{\approx}_{\overline{f''}}^{\overline{D}} \Sigma \\ &\text{and} && \Sigma = w(C, H, \overline{B} \triangleright ([\![s]\!]^s; s'')_{\overline{f}}, \bot, S) \\ &\text{and} && \Sigma' = w(C, H', \overline{B'} \triangleright ([\![s']\!]^s; s'')_{\overline{f'}}, \bot, S) \\ &\text{then} && (n, \Sigma) \xrightarrow{\overline{\lambda^\sigma}} (n', \Sigma') \\ &\text{and} && \lambda^\sigma \approx \overline{\lambda^\sigma} \quad (\text{ using the trace relation!}) \end{aligned}$$

and $\exists \overline{\mathbf{D'}} \supseteq \overline{\mathbf{D}}.\ \Omega' \overset{s}{\underset{\overline{\mathbf{f''}}}{\diamond}} \overline{\mathbf{D'}} \Sigma'$

Theorem 26 (Backward Simulation for Compiled Statements in SLH)**.** (Proof 27)

if $(\mathbf{n}, \Sigma) \xrightarrow{\overline{\lambda^\sigma}} (\mathbf{n'}, \Sigma')$
and $\Omega \overset{s}{\underset{\overline{\mathbf{f''}}}{\diamond}} \overline{\mathbf{D}} \Sigma$
and $\Sigma = \mathbf{w}(\mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \triangleright (\llbracket \mathsf{s} \rrbracket^s ; \mathbf{s''})_{\overline{\mathbf{f}}}, \bot, \mathbf{S})$
and $\Sigma' = \mathbf{w}(\mathbf{C}, \mathbf{H'}, \overline{\mathbf{B'}} \triangleright (\llbracket \mathsf{s'} \rrbracket^s ; \mathbf{s''})_{\overline{\mathbf{f'}}}, \bot, \mathbf{S})$
then $\Omega = \mathsf{C}, \mathsf{H}, \overline{\mathsf{B}} \triangleright (\mathsf{s}; \mathsf{s''})_{\overline{\mathsf{f}}} \xrightarrow{\lambda^\sigma} \mathsf{C}, \mathsf{H'}, \overline{\mathsf{B'}} \triangleright (\mathsf{s'}; \mathsf{s''})_{\overline{\mathsf{f'}}} = \Omega'$
and $\lambda^\sigma \approx \overline{\lambda^\sigma}$ (using the trace relation!)
and $\exists \overline{\mathbf{D'}} \supseteq \overline{\mathbf{D}}.\ \Omega' \overset{s}{\underset{\overline{\mathbf{f''}}}{\diamond}} \overline{\mathbf{D'}} \Sigma'$

Lemma 29 (Expression Reductions with Safe Bindings are Safe)**.** (Proof 28)

if $\vdash \mathbf{B} : \overset{B}{\diamond}$
then $\mathbf{B} \triangleright \mathbf{e} \downarrow \mathbf{v} : \mathbf{S}$

Lemma 30 (Any Speculation from Related States is Safe)**.** (Proof 34)

if $\Omega \overset{s}{\underset{\overline{\mathbf{f_c}}}{\diamond}} \overline{\mathbf{D}} \Sigma$
and $\Sigma = \mathbf{w}(\mathbf{C}, \mathbf{H_b}, \overline{\mathbf{B_b}} \triangleright (\mathbf{s_b})_{\overline{\mathbf{f_b}}}, \bot, \mathbf{S}) \cdot \overline{(\mathbf{C}, \mathbf{H'}, \overline{\mathbf{B'}} \triangleright (\mathbf{s'})_{\overline{\mathbf{f'}}}, \omega', \mathbf{U})} \cdot (\mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \triangleright (\mathbf{s})_{\overline{\mathbf{f}} \cdot \mathbf{f}}, \omega, \mathbf{U})$
and $\Sigma' = \mathbf{w}(\mathbf{C}, \mathbf{H_b}, \overline{\mathbf{B_b}} \triangleright (\mathbf{s_b})_{\overline{\mathbf{f_b}}}, \bot, \mathbf{S})$
and $\mathbf{n'} \leq \mathbf{n} + \mathbf{w}$
and let $\overline{\omega'} = \omega'_1, \cdots, \omega'_{\mathbf{k}}, \omega'_1 + \cdots + \omega'_{\mathbf{k}} + \omega \leq \mathbf{w}$
then $(\mathbf{n}, \Sigma) \xrightarrow{\overline{\lambda^\sigma}} (\mathbf{n'}, \Sigma')$
and $\varnothing \approx \overline{\lambda^\sigma}$
and $\Omega \overset{s}{\underset{\overline{\mathbf{f_c}}}{\diamond}} \overline{\mathbf{D}} \Sigma'$

Lemma 31 (Speculation Lasts at Most Omega)**.** (Proof 29)

if $\Omega \overset{s}{\underset{\overline{\mathbf{f_c}}}{\diamond}} \overline{\mathbf{D}} \Sigma$
and $\Sigma = \mathbf{w}(\mathbf{C}, \mathbf{H_b}, \overline{\mathbf{B_b}} \triangleright (\mathbf{s_b})_{\overline{\mathbf{f_b}}}, \bot, \mathbf{S}) \cdot \overline{(\mathbf{C}, \mathbf{H'}, \overline{\mathbf{B'}} \triangleright (\mathbf{s'})_{\overline{\mathbf{f'}}}, \omega', \mathbf{U})} \cdot (\mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \triangleright (\mathbf{s})_{\overline{\mathbf{f}} \cdot \mathbf{f}}, \omega, \mathbf{U})$
and $\Sigma' = \mathbf{w}(\mathbf{C}, \mathbf{H_b}, \overline{\mathbf{B_b}} \triangleright (\mathbf{s_b})_{\overline{\mathbf{f_b}}}, \bot, \mathbf{S}) \cdot \overline{(\mathbf{C}, \mathbf{H'}, \overline{\mathbf{B'}} \triangleright (\mathbf{s'})_{\overline{\mathbf{f'}}}, \omega', \mathbf{U})} \cdot (\mathbf{C}, \mathbf{H''}, \overline{\mathbf{B''}} \triangleright (\mathbf{s''})_{\overline{\mathbf{f''}} \cdot \mathbf{f''}}, \mathbf{0}, \mathbf{U})$
and $\mathbf{n'} \leq \mathbf{n} + \omega$
then $(\mathbf{n}, \Sigma) \xrightarrow{\overline{\lambda^\sigma}} (\mathbf{n'}, \Sigma')$
and $\varnothing \approx \overline{\lambda^\sigma}$
and $\Omega \overset{s}{\underset{\overline{\mathbf{f_c}}}{\diamond}} \overline{\mathbf{D}} \Sigma'$

Lemma 32 (Context Speculation Lasts at Most Omega)**.** (Proof 30)

if $\Omega \overset{s}{\underset{\overline{\mathbf{f_c}}}{\diamond}} \overline{\mathbf{D}} \Sigma$
and $\Sigma = \mathbf{w}(\mathbf{C}, \mathbf{H_b}, \overline{\mathbf{B_b}} \triangleright (\mathbf{s_b})_{\overline{\mathbf{f_b}}}, \bot, \mathbf{S}) \cdot \overline{(\mathbf{C}, \mathbf{H'}, \overline{\mathbf{B'}} \triangleright (\mathbf{s'})_{\overline{\mathbf{f'}}}, \omega', \mathbf{U})} \cdot (\mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \triangleright (\mathbf{s})_{\overline{\mathbf{f}} \cdot \mathbf{f}}, \omega, \mathbf{U})$
and $\Sigma' = \mathbf{w}(\mathbf{C}, \mathbf{H_b}, \overline{\mathbf{B_b}} \triangleright (\mathbf{s_b})_{\overline{\mathbf{f_b}}}, \bot, \mathbf{S}) \cdot \overline{(\mathbf{C}, \mathbf{H'}, \overline{\mathbf{B'}} \triangleright (\mathbf{s'})_{\overline{\mathbf{f'}}}, \omega', \mathbf{U})} \cdot (\mathbf{C}, \mathbf{H''}, \overline{\mathbf{B''}} \triangleright (\mathbf{s''})_{\overline{\mathbf{f''}} \cdot \mathbf{f''}}, \mathbf{0}, \mathbf{U})$
and $\mathbf{n'} \leq \mathbf{n} + \omega$
and $\mathbf{f}, \mathbf{f''} \notin \overline{\mathbf{f_c}}$
then $(\mathbf{n}, \Sigma) \xrightarrow{\overline{\lambda^\sigma}} (\mathbf{n'}, \Sigma')$
and $\varnothing \approx \overline{\lambda^\sigma}$
and $\Omega \overset{s}{\underset{\overline{\mathbf{f_c}}}{\diamond}} \overline{\mathbf{D}} \Sigma'$

Lemma 33 (Single Context Speculation is Safe)**.** (Proof 31)

if $\Omega \overset{s}{\underset{\overline{\mathbf{f_c}}}{\diamond}} \overline{\mathbf{D}} \Sigma$

and $\Sigma = \mathbf{w}(\mathbf{C}, \mathbf{H_b}, \overline{\mathbf{B_b}} \triangleright (\mathbf{s_b})_{\overline{\mathbf{f_b}}}, \bot, \mathbf{S}) \cdot \overline{(\mathbf{C}, \mathbf{H'}, \overline{\mathbf{B'}} \triangleright (\mathbf{s'})_{\overline{\mathbf{f'}}}, \omega', \mathbf{U})} \cdot (\mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \triangleright (\mathbf{s})_{\overline{\mathbf{f}}\cdot\mathbf{f}}, \omega, \mathbf{U})$

and $\Sigma' = \mathbf{w}(\mathbf{C}, \mathbf{H_b}, \overline{\mathbf{B_b}} \triangleright (\mathbf{s_b})_{\overline{\mathbf{f_b}}}, \bot, \mathbf{S}) \cdot \overline{(\mathbf{C}, \mathbf{H'}, \overline{\mathbf{B'}} \triangleright (\mathbf{s'})_{\overline{\mathbf{f'}}}, \omega', \mathbf{U})} \cdot (\mathbf{C}, \mathbf{H''}, \overline{\mathbf{B''}} \triangleright (\mathbf{s''})_{\overline{\mathbf{f''}}\cdot\mathbf{f''}}, \omega - \mathbf{1}, \mathbf{U})$

and $\mathbf{f}, \mathbf{f''} \notin \overline{\mathsf{f_c}}$

then $(\Sigma) \overset{\alpha^\sigma}{\rightsquigarrow} (\Sigma')$

and $\varnothing \approx \alpha^\sigma$

and $\Omega \overset{S}{\approx}_{\overline{\mathsf{f_c}}}^{\overline{\mathbf{D}}} \Sigma'$

Lemma 34 (Compiled Speculation Lasts at Most Omega)**.** (Proof 32)

if $\Omega \overset{S}{\approx}_{\overline{\mathsf{f_c}}}^{\overline{\mathbf{D}}} \Sigma$

and $\Sigma = \mathbf{w}(\mathbf{C}, \mathbf{H_b}, \overline{\mathbf{B_b}} \triangleright (\mathbf{s_b})_{\overline{\mathbf{f_b}}}, \bot, \mathbf{S}) \cdot \overline{(\mathbf{C}, \mathbf{H'}, \overline{\mathbf{B'}} \triangleright (\mathbf{s'})_{\overline{\mathbf{f'}}}, \omega', \mathbf{U})} \cdot (\mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \triangleright (\llbracket \mathsf{s} \rrbracket^s)_{\overline{\mathbf{f}}\cdot\mathbf{f}}, \omega, \mathbf{U})$

and $\Sigma' = \mathbf{w}(\mathbf{C}, \mathbf{H_b}, \overline{\mathbf{B_b}} \triangleright (\mathbf{s_b})_{\overline{\mathbf{f_b}}}, \bot, \mathbf{S}) \cdot \overline{(\mathbf{C}, \mathbf{H'}, \overline{\mathbf{B'}} \triangleright (\mathbf{s'})_{\overline{\mathbf{f'}}}, \omega', \mathbf{U})} \cdot (\mathbf{C}, \mathbf{H''}, \overline{\mathbf{B''}} \triangleright (\llbracket \mathsf{s''} \rrbracket^s)_{\overline{\mathbf{f''}}\cdot\mathbf{f''}}, \mathbf{0}, \mathbf{U})$

and $\mathbf{n'} \leq \mathbf{n} + \omega$

and $\mathbf{f}, \mathbf{f''} \in \overline{\mathsf{f_c}}$

then $(\mathbf{n}, \Sigma) \overset{\overline{\lambda^\sigma}}{\Longrightarrow} (\mathbf{n'}, \Sigma')$

and $\varnothing \approx \overline{\lambda^\sigma}$

and $\Omega \overset{S}{\approx}_{\overline{\mathsf{f_c}}}^{\overline{\mathbf{D}}} \Sigma'$

Lemma 35 (Compiled Speculation is Safe)**.** (Proof 33)

if $\Omega \overset{S}{\approx}_{\overline{\mathsf{f_c}}}^{\overline{\mathbf{D}}} \Sigma$

and $\Sigma = \mathbf{w}(\mathbf{C}, \mathbf{H_b}, \overline{\mathbf{B_b}} \triangleright (\mathbf{s_b})_{\overline{\mathbf{f_b}}}, \bot, \mathbf{S}) \cdot \overline{(\mathbf{C}, \mathbf{H'}, \overline{\mathbf{B'}} \triangleright (\mathbf{s'})_{\overline{\mathbf{f'}}}, \omega', \mathbf{U})} \cdot (\mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \triangleright (\llbracket \mathsf{s} \rrbracket^s)_{\overline{\mathbf{f}}\cdot\mathbf{f}}, \omega, \mathbf{U})$

and $\Sigma' = \mathbf{w}(\mathbf{C}, \mathbf{H_b}, \overline{\mathbf{B_b}} \triangleright (\mathbf{s_b})_{\overline{\mathbf{f_b}}}, \bot, \mathbf{S}) \cdot \overline{(\mathbf{C}, \mathbf{H'}, \overline{\mathbf{B'}} \triangleright (\mathbf{s'})_{\overline{\mathbf{f'}}}, \omega', \mathbf{U})} \cdot (\mathbf{C}, \mathbf{H''}, \overline{\mathbf{B''}} \triangleright (\llbracket \mathsf{s''} \rrbracket^s)_{\overline{\mathbf{f''}}\cdot\mathbf{f''}}, \omega'', \mathbf{U})$

and $\mathbf{f}, \mathbf{f''} \in \overline{\mathsf{f_c}}$

then $(\mathbf{n}, \Sigma) \overset{\overline{\lambda^\sigma}}{\Longrightarrow} (\mathbf{n'}, \Sigma')$

and $\varnothing \approx \overline{\lambda^\sigma}$

and either $\Omega \overset{S}{\approx}_{\overline{\mathsf{f_c}}}^{\overline{\mathbf{D}}} \Sigma'$

or $\omega'' = \mathbf{0}$

Theorem 27 (Correctness of the Backtranslation for SLH)**.** (Proof 36)

if $\mathbf{A}\,[\llbracket \mathsf{P} \rrbracket^s] \rightsquigarrow \overline{\lambda^\sigma}$

then $\langle\!\langle \mathbf{A} \rangle\!\rangle_c^f\,[\mathsf{P}] \rightsquigarrow \overline{\lambda^\sigma}$

and $\overline{\lambda^\sigma} \approx \overline{\lambda^\sigma}$

Theorem 28 (Generalised Backward Simulation for SLH)**.** (Proof 35)

if if $\mathsf{f} \in \overline{\mathsf{f''}}$ then $\mathbf{s} = \llbracket \mathsf{s} \rrbracket^s$ else $\mathbf{s} = \langle\!\langle \mathsf{s} \rangle\!\rangle_c^f$

and if $\mathsf{f'} \in \overline{\mathsf{f''}}$ then $\mathbf{s'} = \llbracket \mathsf{s'} \rrbracket^s$ else $\mathbf{s'} = \langle\!\langle \mathsf{s'} \rangle\!\rangle_c^f$

and $\Sigma = \mathbf{w}(\mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \triangleright (\mathbf{s}; \mathbf{s''})_{\overline{\mathbf{f}}\cdot\mathbf{f}}, \bot, \mathbf{S})$

and $\Sigma' = \mathbf{w}(\mathbf{C}, \mathbf{H'}, \overline{\mathbf{B'}} \triangleright (\mathbf{s'}; \mathbf{s''})_{\overline{\mathbf{f'}}\cdot\mathbf{f'}}, \bot, \mathbf{S})$

and $(\mathbf{n}, \Sigma) \overset{\overline{\lambda^\sigma}}{\Longrightarrow} (\mathbf{n'}, \Sigma')$

and $\Omega \overset{S}{\approx}_{\overline{\mathsf{f''}}}^{\overline{\mathbf{D}}} \Sigma$

then $\Omega = \mathsf{C}, \mathsf{H}, \overline{\mathsf{B}} \triangleright (\mathsf{s}; \mathsf{s''})_{\overline{\mathsf{f}}\cdot\mathsf{f}} \overset{\overline{\lambda^\sigma}}{\Longrightarrow} \mathsf{C}, \mathsf{H'}, \overline{\mathsf{B'}} \triangleright (\mathsf{s'}; \mathsf{s''})_{\overline{\mathsf{f'}}\cdot\mathsf{f'}} = \Omega'$

and $\overline{\lambda^\sigma} \approx \overline{\lambda^\sigma}$

and $\exists \overline{\mathbf{D'}} \supseteq \overline{\mathbf{D}}.\ \Omega' \overset{S}{\approx}_{\overline{\mathsf{f''}}}^{\overline{\mathbf{D'}}} \Sigma'$

Lemma 36 (Compiled Speculation is Safe Inter-procedurally)**.** (Proof 39)

if $\Omega \overset{S}{\propto}_{\overline{\mathsf{f_c}}}^{\overline{\mathbf{D}}} \Sigma$

and $\mathbf{\Sigma} = \mathbf{w}(\mathbf{C}, \mathbf{H_b}, \overline{\mathbf{B_b}} \triangleright (\mathbf{s_b})_{\overline{\mathbf{f_b}}}, \bot, \mathbf{S}) \cdot \overline{(\mathbf{C}, \mathbf{H'}, \overline{\mathbf{B'}} \triangleright (\mathbf{s'})_{\overline{\mathbf{f'}}}, \omega', \mathbf{U})} \cdot (\mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \triangleright ([\![\mathsf{s}]\!]^{s}_{n})_{\overline{\mathbf{f}}\cdot\mathbf{f}}, \omega, \mathbf{U})$

and $\mathbf{\Sigma'} = \mathbf{w}(\mathbf{C}, \mathbf{H_b}, \overline{\mathbf{B_b}} \triangleright (\mathbf{s_b})_{\overline{\mathbf{f_b}}}, \bot, \mathbf{S}) \cdot \overline{(\mathbf{C}, \mathbf{H'}, \overline{\mathbf{B'}} \triangleright (\mathbf{s'})_{\overline{\mathbf{f'}}}, \omega', \mathbf{U})} \cdot (\mathbf{C}, \mathbf{H''}, \overline{\mathbf{B''}} \triangleright ([\![\mathsf{s''}]\!]^{s}_{n})_{\overline{\mathbf{f''}}\cdot\mathbf{f''}}, \omega'', \mathbf{U})$

and $\mathbf{f}, \mathbf{f''} \in \overline{\mathsf{f_c}}$

then $(\mathbf{n}, \mathbf{\Sigma}) \xRightarrow{\overline{\lambda^{\sigma}}} (\mathbf{n'}, \mathbf{\Sigma'})$

and $\varnothing \approx \overline{\lambda^{\sigma}}$

and either $\Omega \propto^{s}_{\overline{\mathsf{f_c}}}{}^{\overline{\mathbf{D}}} \mathbf{\Sigma'}$

or $\omega'' = \mathbf{0}$

APPENDIX K
PROOFS FOR COUNTERMEASURES AND CRITERIA

Proof of Theorem 8 (All L programs are RSS).
Trivial: the L semantics only emits S actions. □

♠

Proof of Theorem 9 (All L programs are RSNI).
By Theorem 8 (All L programs are RSS) and by Theorem 11 (SS implies SNI). □

♠

Proof of Theorem 13 (RSSC implies RSSP).
We have (HPSSC)

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

We need to prove:

$$\forall \mathsf{P}.\ \text{if } \forall \mathsf{A}. \forall \overline{\lambda^{\sigma}} \in \mathsf{Beh}(\mathsf{A}\,[\mathsf{P}]). \forall \alpha^{\sigma} \in \overline{\lambda^{\sigma}}. \sigma \equiv \mathsf{S}$$
$$\text{then } \forall \mathbf{A}. \forall \overline{\boldsymbol{\lambda^{\sigma}}} \in \mathbf{Beh}(\mathbf{A}\,[[\mathsf{P}]]). \forall \boldsymbol{\alpha^{\sigma}} \in \overline{\boldsymbol{\lambda^{\sigma}}}. \boldsymbol{\sigma} \equiv \mathbf{S}$$

We proceed by contradiction:

$$\forall \mathsf{A}. \forall \overline{\lambda^{\sigma}} \in \mathsf{Beh}(\mathsf{A}\,[\mathsf{P}]). \forall \alpha^{\sigma} \in \overline{\lambda^{\sigma}}. \sigma \equiv \mathsf{S} \text{ (HPS)}$$
$$\text{and } \exists \mathbf{A}. \exists \overline{\boldsymbol{\lambda^{\sigma}}} \in \mathbf{Beh}(\mathbf{A}\,[[\mathsf{P}]]). \exists \boldsymbol{\alpha^{\sigma}} \in \overline{\boldsymbol{\lambda^{\sigma}}} \text{ (HPT)}$$
$$\text{and } \boldsymbol{\sigma} \equiv \mathbf{U} \text{ HPU}$$

We instantiate HPSSC with HPS and HPT so we get that $\overline{\lambda^{\sigma}} \approx \overline{\boldsymbol{\lambda^{\sigma}}}$.
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 8, 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. □

♠

Proof of Theorem 14 (RSSP implies RSSC).
We have (RSSP)

$$\forall \mathsf{P}.\ \text{if } \forall \mathsf{A}. \forall \overline{\lambda^{\sigma}} \in \mathsf{Beh}(\mathsf{A}\,[\mathsf{P}]). \forall \alpha^{\sigma} \in \overline{\lambda^{\sigma}}. \sigma \equiv \mathsf{S}$$
$$\text{then } \forall \mathbf{A}. \forall \overline{\boldsymbol{\lambda^{\sigma}}} \in \mathbf{Beh}(\mathbf{A}\,[[\mathsf{P}]]). \forall \boldsymbol{\alpha^{\sigma}} \in \overline{\boldsymbol{\lambda^{\sigma}}}. \boldsymbol{\sigma} \equiv \mathbf{S}$$

We need to prove:

$$\forall \mathsf{P}, \mathbf{A}, \overline{\boldsymbol{\lambda^{\sigma}}}, \exists \mathsf{A}, \overline{\lambda^{\sigma}}.$$
$$\text{if } \mathbf{A}\,[[\mathsf{P}]] \rightsquigarrow \overline{\boldsymbol{\lambda^{\sigma}}} \text{ then } \mathsf{A}\,[\mathsf{P}] \rightsquigarrow \overline{\lambda^{\sigma}} \text{ and } \overline{\lambda^{\sigma}} \approx \overline{\boldsymbol{\lambda^{\sigma}}}$$

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

$$\exists \mathbf{A}, \overline{\boldsymbol{\lambda^{\sigma}}}, \forall \mathsf{A}, \overline{\lambda^{\sigma}}.$$
$$\mathbf{A}\,[[\mathsf{P}]] \rightsquigarrow \overline{\boldsymbol{\lambda^{\sigma}}} \text{ and } \mathsf{A}\,[\mathsf{P}] \rightsquigarrow \overline{\lambda^{\sigma}} \text{ (HPSR) and } \overline{\lambda^{\sigma}} \not\approx \overline{\boldsymbol{\lambda^{\sigma}}} \text{ (HPA)}$$

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 $\mathbf{U}$.
- 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 $\mathbf{U}$.

We can therefore conclude that $\exists \mathbf{A}.\, \exists \overline{\lambda^{\sigma}} \in \mathbf{Beh}(\mathbf{A}\,[\![\mathsf{P}]\!]).\, \exists \alpha^{\sigma} \in \overline{\lambda^{\sigma}}.\, \sigma \equiv \mathbf{U}$ (HPU).

We instantiate RSSP with HPSR and conclude

$$\forall \mathbf{A}.\, \forall \overline{\lambda^{\sigma}} \in \mathbf{Beh}(\mathbf{A}\,[\![\mathsf{P}]\!]).$$
$$\forall \alpha^{\sigma} \in \overline{\lambda^{\sigma}}.\, \sigma \equiv \mathbf{S} \text{ (HPS)}$$

We obtain the contradiction between HPU and HPS. □

Proof of Theorem 15 (RSSC and RSSP are equivalent).

By Theorem 13 (*RSSC* implies *RSSP*) and Theorem 14 (*RSSP* implies *RSSC*). □

Proof of Theorem 18 (The lfence compiler is RSSC).

Instantiate A with $\langle\!\langle \mathbf{A} \rangle\!\rangle_c^f$.

This holds by Theorem 20 (Correctness of the Backtranslation for lfence). □

Proof of Theorem 19 (All lfence-compiled programs are RSSC).

By Theorem 8 we have HPS: $\forall \mathsf{P}. \vdash \mathsf{P} : \text{RSS}$.

By Theorem 18 we have HPC: $\vdash [\![\cdot]\!]^f : \mathit{RSSC}$.

By Theorem 13 with HPC we have HPP: $\vdash [\![\cdot]\!]^f : \mathit{RSSP}$.

By Definition 17 (Robust Dynamic Speculative Safety-Preserving Compiler (RSSP)) of HPP with HPS we conclude that $\forall \mathsf{P}. \vdash [\![\mathsf{P}]\!]^f : \text{RSS}$. □

Proof of Theorem 20 (Correctness of the Backtranslation for lfence).

This holds by Lemma 18 (Initial States are Related) and by Theorem 21 (Generalised Backward Simulation for lfence). □

Proof of Theorem 21 (Generalised Backward Simulation for lfence).

We proceed by induction on the reduction $\xRightarrow{\overline{\lambda^{\sigma}}}$

Base no reductions, this case is trivial

Inductive we have n target steps to states $\mathbf{\Sigma_i} = \mathbf{C}, \mathbf{H_i}, \overline{\mathbf{B_i}} \triangleright (\mathsf{s_i}; \mathsf{s_i''})_{\overline{\mathsf{f_i}} \cdot \mathsf{f_i}}$ and $\Omega_\mathsf{i} = \mathsf{C}, \mathsf{H_i}, \overline{\mathsf{B_i}} \triangleright (\mathsf{s_i}; \mathsf{s_i''})_{\overline{\mathsf{f_i}} \cdot \mathsf{f_i}}$
where $\Omega_\mathsf{i} \overset{\mathcal{S}}{\approx} \mathbf{\Sigma_i}$ and the traces produced so far are related via $\approx$.
We proceed by case analysis on $\mathsf{f_i}$

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

in $\overline{f''}$ (in the compiled component) This holds by Theorem 22 (Backward Simulation for Compiled Steps in lfence);

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 $\mathbf{\Sigma_i} = \mathbf{C}, \mathbf{H_i}, \overline{\mathbf{B_i}} \cdot \mathbf{B_i} \triangleright (\llbracket \mathtt{call}\ f'\ e \rrbracket^f ; \mathbf{s_i''})_{\overline{\mathbf{f_i}} \cdot \mathbf{f_i}}$
so by Rule E-**T**-speculate-action with Rule E-L-call(in the target ofc) we know
$\mathbf{\Sigma_i} \xrightarrow{(\mathtt{call}\ f'\ \llbracket v \rrbracket^f ?)^S} \mathbf{\Sigma'} = \mathbf{w}, (\mathbf{C}, \mathbf{H'}, \overline{\mathbf{B'}} \cdot \mathbf{x} \mapsto \llbracket v \rrbracket^f \triangleright (\mathbf{s_f}; \mathbf{s_i''})_{\overline{\mathbf{f'}} \cdot \mathbf{f'}}, \bot, \mathbf{S})$
and we know $\mathbf{B_i} \triangleright \llbracket e \rrbracket^f \downarrow \llbracket v \rrbracket^f$ (HPE)
where $\mathbf{f}(\mathbf{x}) \mapsto \mathbf{s_f} \in \mathbf{C}$ and $\overline{\mathbf{B'}} = \overline{\mathbf{B_i}} \cdot \mathbf{B_i}$ and $\overline{\mathbf{f'}} = \overline{\mathbf{f_i}} \cdot \mathbf{f_i}$ and $\mathbf{H'}$=$\mathbf{H_i}$
and the taint is $\mathbf{S}$ by definition of $\sqcap$ since by $\overset{S}{\approx}$ the pc taint is $\mathbf{S}$.
By Lemma 16 (Backward Simulation for Backtranslated Expressions in lfence), HPE yields $B_i \triangleright e \downarrow v$ (HPSE)
We have that $\Omega_i = C, H_i, \overline{B_i} \cdot B_i \triangleright (\mathtt{call}\ f\ e; s_i'')_{\overline{f_i} \cdot f_i}$
by definition of $\langle\langle \cdot \rangle\rangle_c^f$ we know that $f(x) \mapsto \langle\langle s_f \rangle\rangle_c^f \in C$
we take $\overline{B'} = \overline{B_i} \cdot B_i$ so that by hypothesis we have that $\overline{B'} \overset{B}{\approx} \overline{\mathbf{B'}}$ (HPB)
we take $\overline{f'} = \overline{f_i} \cdot f_i$ so that by hypothesis we have that $\overline{f'} \equiv \overline{\mathbf{f'}}$ (HPF)
we take $H' = H_i$ so that by hypothesis we have that $H' \overset{H}{\approx} \mathbf{H'}$ (HPH)
By Rule E-L-call(in the source) with HPSE and the hypotheses above, we have that
$\Omega_i \xrightarrow{(\mathtt{call}\ f'\ v?)^S} \Omega' = C, H', \overline{B'} \cdot x \mapsto v \triangleright (\langle\langle s_f \rangle\rangle_c^f ; s_i'')_{\overline{f'} \cdot f'}$
We need to prove that:

- $\Omega' \overset{S}{\approx} \mathbf{\Sigma'}$, which by Rule States means proving that:
  - $\overline{B'} \cdot x \mapsto v \overset{B}{\approx} \overline{\mathbf{B'}} \cdot \mathbf{x} \mapsto \llbracket v \rrbracket^f$, which holds by (HPB) and by Rule Binding - ind with Rule Value - num with Lemma 19 (A Value is Related to its Compilation for lfence);
  - $\overline{f'} \equiv \overline{\mathbf{f'}}$, which holds by (HPF);
  - $C \overset{C}{\approx}_{\overline{f''}} \mathbf{C}$, which holds by $\overset{S}{\approx}$ of the initial states since components do not change;
  - $H' \overset{H}{\approx} \mathbf{H'}$, which holds by (HPH)
- $(\mathtt{call}\ f'\ v?)^S \overset{A}{\approx} (\mathtt{call}\ f'\ \llbracket v \rrbracket^f ?)^{\mathbf{S}}$, which by Rule Action Relation - call means proving that:
  - $f' \equiv \mathbf{f'}$, which holds;
  - $v \overset{V}{\approx} \llbracket v \rrbracket^f$, which holds by Lemma 19;
  - $S \equiv \mathbf{S}$

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 $\mathbf{\Sigma_i} = \mathbf{C}, \mathbf{H_i}, \overline{\mathbf{B_i}} \cdot \mathbf{B_i} \triangleright (\mathbf{return};; \llbracket \mathbf{s_f} \rrbracket^f ; \mathbf{s_i''})_{\overline{\mathbf{f_i}} \cdot \mathbf{f_i}}$
so by Rule E-**T**-speculate-action with Rule E-L-return(in the target ofc) we know
$\mathbf{\Sigma_i} \xrightarrow{(\mathtt{ret!})^S} \mathbf{\Sigma'} = \mathbf{w}, (\mathbf{C}, \mathbf{H_i}, \overline{\mathbf{B_i}} \triangleright (\llbracket \mathbf{s_f} \rrbracket^f ; \mathbf{s_i''})_{\overline{\mathbf{f_i}}}, \bot, \mathbf{S})$
We have that $\Omega_i = C, H_i, \overline{B_i} \cdot B_i \triangleright (\mathtt{return};; s_f; s_i'')_{\overline{f_i} \cdot f_i}$
By Rule E-L-call(in the source), we have that
$\Omega_i \xrightarrow{(\mathtt{ret!})^S} \Omega' = C, H_i, \overline{B_i} \triangleright (s_f; s_i'')_{\overline{f_i}}$
We need to prove that:

- $\Omega' \overset{S}{\approx} \mathbf{\Sigma'}$, which by Rule States means proving that:
  - $\overline{B_i} \overset{B}{\approx} \overline{\mathbf{B_i}}$, which holds by hypothesis;
  - $\overline{f_i} \equiv \overline{\mathbf{f_i}}$, which holds by hypothesis;
  - $C \overset{C}{\approx}_{\overline{f''}} \mathbf{C}$, which holds by $\overset{S}{\approx}$ of the initial states since components do not change;
  - $H_i \overset{H}{\approx} \mathbf{H_i}$, which holds by hypothesis
- $(\mathtt{ret!})^S \overset{A}{\approx} (\mathtt{ret!})^{\mathbf{S}}$, which by Rule Action Relation - return is trivially true.

so this case holds.

not in $\overline{f''}$ (in the context) This holds by Lemma 17 (Backward Simulation for Backtranslated Statements);

□

♠

Proof of Theorem 22 (Backward Simulation for Compiled Steps in lfence).
By contradiction, assume the source ends up in a different state Ω'' such that $\Omega'' \neq \Omega'$.
By Lemma 15 (Forward Simulation for Compiled Statements in lfence) we have that the target also ends up in state Σ'' such that $\Omega'' \overset{S}{\approx}_{\overline{f''}} \Sigma''$ 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 14 (Forward Simulation for Expressions in lfence).
Trivial induction on e. □

♠

Proof of Lemma 15 (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 14 (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 $[\![s']\!]^f$.
- **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 $[\![s']\!]^f$.
- **write** by Lemma 14 (Forward Simulation for Expressions in lfence) and Rule Action Relation - write 2;
- **private write** by Lemma 14 (Forward Simulation for Expressions in lfence) and Rule Action Relation - write.

Inductive sequencing by IH;
- **letin** by IH and Lemma 14 (Forward Simulation for Expressions in lfence);
- **if zero** by IH and Lemma 14 (Forward Simulation for Expressions in lfence).
  By definition of $[\![\cdot]\!]^f$, 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 14 (Forward Simulation for Expressions in lfence) and Rule Action Relation - read;
- **let private read** by IH and Lemma 14 (Forward Simulation for Expressions in lfence) and Rule Action Relation - read;
- **conditional letin** by IH and Lemma 14 (Forward Simulation for Expressions in lfence).

□

♠

Proof of Lemma 16 (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 17 (Backward Simulation for Backtranslated Statements).
The proof proceeds by structural induction on $\mathbf{s}$.

Base skip trivial;
call Two cases arise:
1) $\mathbf{f}$ is defined by the component.
This holds by Lemma 16 (Backward Simulation for Backtranslated Expressions in lfence) and relatedness of heaps.
2) $\mathbf{f}$ is defined by the context.
This cannot arise as in the source we don't step to a backtranslated statement $\langle\!\langle \mathbf{s}' \rangle\!\rangle_c^f$.

return Two cases arise:
1) $\mathbf{f}$ is defined by the component.
This holds by relatedness of heaps.
2) $\mathbf{f}$ is defined by the context.
This cannot arise as in the source we don't step to a backtranslated statement $\langle\!\langle \mathbf{s}' \rangle\!\rangle_c^f$.

write by Lemma 16 (Backward Simulation for Backtranslated Expressions in lfence).
One complexity is showing that the action taint σ=$\mathbf{S}$, but this follows from $\overset{S}{\approx}$ which tells that the pc taint is $\mathbf{S}$.
private write this cannot arise by Definition 9;
lfence trivial.
Inductive sequencing by IH;
letin by IH and Lemma 16 (Backward Simulation for Backtranslated Expressions in lfence);
if zero by IH and Lemma 16 (Backward Simulation for Backtranslated Expressions in lfence).
In this case, Rule E-$\mathbf{T}$-speculate-if is not applicable, so we cannot speculate.
let read by IH and Lemma 16 (Backward Simulation for Backtranslated Expressions in lfence).
One complexity is showing that the action taint σ=$\mathbf{S}$, but this follows from $\overset{S}{\approx}$ which tells that the pc taint is $\mathbf{S}$.
let private read this cannot arise by Definition 9;
conditional letin by IH and Lemma 16 (Backward Simulation for Backtranslated Expressions in lfence).

□

♠

Proof of Lemma 18 (Initial States are Related).
By definition of $\overset{S}{\approx}$, 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 20 (A Heap is Related to its Compilation for lfence);
  - for the context heap, Definition 9 tells us that its domain is natural numbers, so this holds by Lemma 23;
- components are related via $\overset{C}{\approx}$ by simple inspection of $[\![\cdot]\!]^f$ and $\langle\!\langle \cdot \rangle\!\rangle_c^f$;
- the target taint is $\mathbf{S}$: this holds by Rule $\mathbf{T}$-Initial State;
- the target window is $\bot$: this holds by Rule $\mathbf{T}$-Initial State;

□

♠

Proof of Lemma 19 (A Value is Related to its Compilation for lfence).
Trivial analyisis of the compiler. □

♠

Proof of Lemma 20 (A Heap is Related to its Compilation for lfence).
Trivial analyisis of the compiler. □

♠

Proof of Lemma 21 (A Value is Related to its Backtranslation).
Trivial analyisis of the backtranslation. □

♠

Proof of Lemma 22 (A Taint is Related to its Backtranslation).
Trivial analyisis of the backtranslation. □

♠

Proof of Lemma 23 (A Heap is Related to its Backtranslation).
Trivial analyisis of the backtranslation with Lemmas 21 and 22. □

♠

Proof of Theorem 23 (The lfence compiler is RSSC with more leaks). Instantiate A with $\langle\!\langle \mathbf{A} \rangle\!\rangle_c^f$.
This holds by an adaptation of Theorem 20 (Correctness of the Backtranslation for lfence) to the extra reduction, which in turn holds by Lemma 18 (Initial States are Related) and by an adaptation of Theorem 21 (Generalised Backward Simulation for lfence) to the extra reduction, which in turn holds by Lemma 16 (Backward Simulation for Backtranslated Expressions in lfence) and by an adaptation of Theorem 22 (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. □

♠

Proof of Lemma 24 (Initial States are Related for SLH).
Analogous to the proof of Lemma 18 (Initial States are Related) but with Lemma 25 (A Value is Related to its Compilation for SLH) and Lemma 26 (A Heap is Related to its Compilation for SLH). □

♠

Proof of Lemma 25 (A Value is Related to its Compilation for SLH).
Trivial analysis of $[\![\cdot]\!]^s$. □

♠

Proof of Lemma 26 (A Heap is Related to its Compilation for SLH).
Trivial analysis of $[\![\cdot]\!]^s$ 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 27 (Forward Simulation for Expressions in SLH). Trivial induction on e. □

♠

Proof of Lemma 28 (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 27 (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 27 (Forward Simulation for Expressions in SLH).

private assign This follows from Lemma 27 (Forward Simulation for Expressions in SLH).

return Trivial.

Inductive sequencing By IH.

let-in By IH and Lemma 27 (Forward Simulation for Expressions in SLH).

if then else In this case we have this source reduction, wlog assume HE $\mathsf{B} \triangleright \mathsf{e} \downarrow \mathsf{true} : \sigma$

$$\mathsf{C}, \mathsf{H}, \overline{\mathsf{B}} \cdot \mathsf{B} \triangleright \mathsf{ifz\ e\ then\ s\ else\ s'}; \mathsf{s''} \xrightarrow{(\mathtt{if}(0))^S} \mathsf{C}, \mathsf{H}, \overline{\mathsf{B}} \cdot \mathsf{B} \triangleright \mathsf{s}; \mathsf{s''}$$

By Lemma 27 (Forward Simulation for Expressions in SLH) with HE we get HET : $\mathbf{B} \triangleright [\![\mathsf{e}]\!]^s \downarrow [\![\mathsf{true}]\!]^s : \sigma$.
In the target we have these reductions (by HET):

$$\begin{aligned}
&\Sigma = \\
&\mathbf{w}(\mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \cdot \mathbf{B}, \bot, \mathbf{S}) \triangleright [\![\mathsf{ifz\ e\ then\ s\ else\ s'}]\!]^s; \mathbf{s''} \\
\equiv &\mathbf{w}(\mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \cdot \mathbf{B}, \bot, \mathbf{S}) \triangleright \mathbf{let}\ \mathbf{x_g} = [\![\mathsf{e}]\!]^s\ \mathbf{in} \\
&\qquad \mathbf{let}\ \mathbf{pr} = \mathbf{rd_p}\ {-1}\ \mathbf{in} \\
&\qquad \mathbf{let}\ \mathbf{x_g} = \mathbf{0}\ (\mathbf{if}\ \mathbf{pr})\ \mathbf{in} \\
&\qquad \mathbf{ifz}\ \mathbf{x_g}\ \mathbf{then\ let}\ \mathbf{x} = \mathbf{rd_p}\ {-1}\ \mathbf{in}\ {-1} :=_{\mathbf{p}} \mathbf{x} \vee \neg \mathbf{x_g}; [\![\mathsf{s}]\!]^s \\
&\qquad\ \mathbf{else\ let}\ \mathbf{x} = \mathbf{rd_p}\ {-1}\ \mathbf{in}\ {-1} :=_{\mathbf{p}} \mathbf{x} \vee \mathbf{x_g}; [\![\mathsf{s'}]\!]^s \\
&\quad \text{let } \mathbf{B'} = \mathbf{B} \cdot \mathbf{x_g} \mapsto \mathbf{true} : \sigma \\
\rightsquigarrow &\mathbf{w}(\mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \cdot \mathbf{B'}, \bot, \mathbf{S}) \triangleright \mathbf{let}\ \mathbf{pr} = \mathbf{rd_p}\ {-1}\ \mathbf{in} \\
&\qquad \mathbf{let}\ \mathbf{x_g} = \mathbf{0}\ (\mathbf{if}\ \mathbf{pr})\ \mathbf{in} \\
&\qquad \mathbf{ifz}\ \mathbf{x_g}\ \mathbf{then\ let}\ \mathbf{x} = \mathbf{rd_p}\ {-1}\ \mathbf{in}\ {-1} :=_{\mathbf{p}} \mathbf{x} \vee \neg \mathbf{x_g}; [\![\mathsf{s}]\!]^s \\
&\qquad\ \mathbf{else\ let}\ \mathbf{x} = \mathbf{rd_p}\ {-1}\ \mathbf{in}\ {-1} :=_{\mathbf{p}} \mathbf{x} \vee \mathbf{x_g}; [\![\mathsf{s'}]\!]^s \\
&\quad \text{since } \mathbf{H}(-1) \mapsto \mathbf{false} : \mathbf{S} \\
&\quad \text{let } \mathbf{B''} = \mathbf{B'} \cup \mathbf{pr} \mapsto \mathbf{false} : \mathbf{S} \\
\xrightarrow{(\mathtt{read}(-1))^S} &\mathbf{w}(\mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \cdot \mathbf{B''}, \bot, \mathbf{S}) \triangleright \mathbf{let}\ \mathbf{x_g} = \mathbf{0}\ (\mathbf{if}\ \mathbf{pr})\ \mathbf{in} \\
&\qquad \mathbf{ifz}\ \mathbf{x_g}\ \mathbf{then\ let}\ \mathbf{x} = \mathbf{rd_p}\ {-1}\ \mathbf{in}\ {-1} :=_{\mathbf{p}} \mathbf{x} \vee \neg \mathbf{x_g}; [\![\mathsf{s}]\!]^s \\
&\qquad\ \mathbf{else\ let}\ \mathbf{x} = \mathbf{rd_p}\ {-1}\ \mathbf{in}\ {-1} :=_{\mathbf{p}} \mathbf{x} \vee \mathbf{x_g}; [\![\mathsf{s'}]\!]^s \\
&\quad \text{since } \mathbf{pr} \mapsto \mathbf{false} : \mathbf{S} \\
\rightsquigarrow &\mathbf{w}(\mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \cdot \mathbf{B''}, \bot, \mathbf{S}) \triangleright \mathbf{ifz}\ \mathbf{x_g}\ \mathbf{then\ let}\ \mathbf{x} = \mathbf{rd_p}\ {-1}\ \mathbf{in}\ {-1} :=_{\mathbf{p}} \mathbf{x} \vee \neg \mathbf{x_g}; [\![\mathsf{s}]\!]^s; \mathbf{s''} \\
&\qquad\ \mathbf{else\ let}\ \mathbf{x} = \mathbf{rd_p}\ {-1}\ \mathbf{in}\ {-1} :=_{\mathbf{p}} \mathbf{x} \vee \mathbf{x_g}; [\![\mathsf{s'}]\!]^s
\end{aligned}$$

By Rule E-T-speculate-if

$$\begin{aligned}
\xrightarrow{(\mathtt{if}(0))^S} &\mathbf{w}\ (\mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \cdot \mathbf{B''}, \bot, \mathbf{S} \triangleright \mathbf{let}\ \mathbf{x} = \mathbf{rd_p}\ {-1}\ \mathbf{in}\ {-1} :=_{\mathbf{p}} \mathbf{x} \vee \neg \mathbf{x_g}; [\![\mathsf{s}]\!]^s; \mathbf{s''}) \\
&\quad \cdot (\mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \cdot \mathbf{B''}, \mathbf{w}, \mathbf{U} \triangleright \mathbf{let}\ \mathbf{x} = \mathbf{rd_p}\ {-1}\ \mathbf{in}\ {-1} :=_{\mathbf{p}} \mathbf{x} \vee \mathbf{x_g}; [\![\mathsf{s'}]\!]^s\ \ ; \mathbf{s''}) \\
&\quad \text{since } \mathbf{H}(-1) \mapsto \mathbf{false} : \mathbf{S} \\
&\quad \text{let } \mathbf{B'''} = \mathbf{B''} \cdot \mathbf{x} \mapsto \mathbf{false} : \mathbf{S} \\
\xrightarrow{(\mathtt{read}(-1))^S} &\mathbf{w}\ (\mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \cdot \mathbf{B''}, \bot, \mathbf{S} \triangleright \mathbf{let}\ \mathbf{x} = \mathbf{rd_p}\ {-1}\ \mathbf{in}\ {-1} :=_{\mathbf{p}} \mathbf{x} \vee \neg \mathbf{x_g}; [\![\mathsf{s}]\!]^s; \mathbf{s''}) \\
&\quad \cdot (\mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \cdot \mathbf{B'''}, \mathbf{w}, \mathbf{U} \triangleright {-1} :=_{\mathbf{p}} \mathbf{x} \vee \mathbf{x_g}; [\![\mathsf{s'}]\!]^s\ \ ; \mathbf{s''}) \\
&\quad \text{since } \mathbf{B'''} \triangleright \mathbf{x} \vee \mathbf{x_g} \downarrow \mathbf{true} : \mathbf{S}\ \text{let } \mathbf{H'} = \mathbf{H} \cup {-1} \mapsto \mathbf{true} : \mathbf{S}
\end{aligned}$$

$$\xrightarrow{(\texttt{write}(-1))^S} \mathbf{w}\ (\mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \cdot \mathbf{B}'', \bot, \mathbf{S} \triangleright \mathbf{let}\ \mathbf{x} = \mathbf{rd_p}\ -1\ \mathbf{in}\ -1 :=_{\mathbf{p}} \mathbf{x} \vee \neg \mathbf{x_g}; [\![\mathsf{s}]\!]^s; \mathbf{s}'')$$
$$\cdot(\mathbf{C}, \mathbf{H}', \overline{\mathbf{B}} \cdot \mathbf{B}''', \mathbf{w}, \mathbf{U} \triangleright [\![\mathsf{s}']\!]^s \quad ; \mathbf{s}'')$$

Call this last state $\mathbf{\Sigma_i}$.
Let $\overline{\mathbf{D}} = \overline{\mathbf{D}} \cdot \mathbf{D}$, consider $\overline{\mathbf{D}'} = \overline{\mathbf{D}} \cdot \mathbf{D}, \mathbf{x_g}$.
We can easily prove that $\Omega \overset{S}{\underset{\mathsf{f_c}}{\approx}} \overline{\mathbf{D}'}\ \mathbf{\Sigma_i}$ since by HP the first states are related at $\overset{S}{\underset{\mathsf{f_c}}{\approx}} \overline{\mathbf{D}'}$ and the speculating states in $\mathbf{\Sigma_i}$ only have safe bindings.

> Note: this is where the proof for $[\![\cdot]\!]^s_{\mathcal{S}}$ 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 $\mathbf{B}$ with all it can have, including $\mathbf{U}$ bindings. If we do not require the bindings to all be safe when speculating, the speculation lemmas break (Lemma 35 (Compiled Speculation is Safe)): without this we cannot show that actions are safe.

By Lemma 30 (Any Speculation from Related States is Safe) we know that:

$$(\mathbf{n}'', \mathbf{\Sigma_i}) \xRightarrow{\overline{\lambda^\sigma}} (\mathbf{n}', \mathbf{\Sigma}'')$$

Where

$$\mathbf{\Sigma}'' = \mathbf{w}\,(\mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \cdot \mathbf{B}'', \bot, \mathbf{S} \triangleright \mathbf{let}\ \mathbf{x} = \mathbf{rd_p}\ -1\ \mathbf{in}\ -1 :=_{\mathbf{p}} \mathbf{x} \vee \neg \mathbf{true}; [\![\mathsf{s}]\!]^s; \mathbf{s}'')$$

and (HL): $\varnothing \approx \overline{\lambda^\sigma}$
and (HS): $\Omega \overset{S}{\underset{\mathsf{f_c}}{\approx}} \overline{\mathbf{D}'}\ \mathbf{\Sigma}''$
and that the first element in the stack of states in $\mathbf{\Sigma}''$ is the same as the first element in the stack of states of $\mathbf{\Sigma}'$, which is still $\overset{S}{\approx}$ with Ω.
The reductions proceed as follows:

$$\mathbf{\Sigma}''$$
$$\xrightarrow{(\texttt{read}(-1))^S} (\mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \cdot \mathbf{B}'', \bot, \mathbf{S} \triangleright -1 :=_{\mathbf{p}} \mathbf{false} \vee \neg \mathbf{true}; [\![\mathsf{s}]\!]^s; \mathbf{s}'')$$
$$\text{since } \_ \triangleright \mathbf{false} \vee \neg \mathbf{true} \downarrow \mathbf{false} : \mathbf{S}$$
$$\xrightarrow{(\texttt{write}(-1))^S} (\mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \cdot \mathbf{B}'', \bot, \mathbf{S} \triangleright [\![\mathsf{s}]\!]^s; \mathbf{s}'')$$

At this point, we have this target trace:

$$(\mathbf{n}, \mathbf{\Sigma}) \xRightarrow{(\texttt{read}(-1))^{\mathbf{S}} \cdot (\texttt{if}(\mathbf{0}))^{\mathbf{S}} \cdot (\texttt{read}(-1))^{\mathbf{S}} \cdot (\texttt{write}(-1))^{\mathbf{S}}} (\mathbf{n}'', \mathbf{\Sigma_i}) \xRightarrow{\overline{\lambda^\sigma}} (\mathbf{n}'', \mathbf{\Sigma}'')$$
$$\xRightarrow{(\texttt{read}(-1))^{\mathbf{S}} \cdot (\texttt{write}(-1))^{\mathbf{S}}} (\mathbf{n}', \mathbf{\Sigma}')$$

i.e.,

$$(\mathbf{n}, \mathbf{\Sigma}) \xRightarrow{(\texttt{read}(-1))^{\mathbf{S}} \cdot (\texttt{if}(\mathbf{0}))^{\mathbf{S}} \cdot (\texttt{read}(-1))^{\mathbf{S}} \cdot (\texttt{write}(-1))^{\mathbf{S}} \cdot \overline{\lambda^\sigma} \cdot (\texttt{read}(-1))^{\mathbf{S}} \cdot (\texttt{write}(-1))^{\mathbf{S}}} (\mathbf{n}', \mathbf{\Sigma}')$$

We need to show that this trace is $\approx$ to the source trace $(\texttt{if}(0))^{\mathsf{S}}$.
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 $\overline{\lambda^\sigma}$ 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 27 (Forward Simulation for Expressions in SLH).

private read By IH and Lemma 27 (Forward Simulation for Expressions in SLH).

□

♠

Proof of Theorem 26 (Backward Simulation for Compiled Statements in SLH).

Analogous to Theorem 22 (Backward Simulation for Compiled Steps in lfence) with Lemma 28 (Forward Simulation for Compiled Statements in SLH). □

♠

Proof of Lemma 29 (Expression Reductions with Safe Bindings are Safe).
Trivial induction on $\mathbf{e}$, the only nontrivial case is when $\mathbf{e} = \mathbf{x}$ but this follows from the safety of bindings. □

♠

Proof of Lemma 31 (Speculation Lasts at Most Omega).
This proceeds by cases on $\mathbf{f}$ and $\mathbf{f}''$:

Both in $\overline{\mathsf{f}_\mathsf{c}}$ These are compiled reductions, this holds by Lemma 34 (Compiled Speculation Lasts at Most Omega);
Both not in $\overline{\mathsf{f}_\mathsf{c}}$ These are context reductions, this holds by Lemma 32 (Context Speculation Lasts at Most Omega);
$\mathbf{f} \in \overline{\mathsf{f}_\mathsf{c}}$ and $\mathbf{f}'' \notin \overline{\mathsf{f}_\mathsf{c}}$ 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 analising the case of $[\![\cdot]\!]^s$ for call, we have:
$\mathbf{let\ x_f = [\![e]\!]^s\ in\ let\ x_f = 0\ (if\ pr)\ in\ call\ f\ x_f}$
We have two cases: $\mathbf{B} \triangleright [\![\mathsf{e}]\!]^s \downarrow \mathbf{v} : \boldsymbol{\sigma}$ or $\mathbf{B} \triangleright [\![\mathsf{e}]\!]^s \downarrow \mathbf{e}' : \boldsymbol{\sigma}$, i.e., the execution of $[\![\mathsf{e}]\!]^s$ 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 $\mathbf{H}(-1) \mapsto \mathbf{true} : \mathbf{S}$ so the code above will step as follows:
(for simplicity we only keep track of the top of the stack of execution states)

$$\begin{aligned}
&(\mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \cdot \mathbf{B} \triangleright \mathbf{let\ x_f = [\![e]\!]^s\ in\ let\ x_f = 0\ (if\ pr)\ in\ call\ f\ x_f}), \omega, \mathbf{U} \\
&\quad \text{assuming } \mathbf{B} \triangleright [\![\mathsf{e}]\!]^s \downarrow \mathbf{v} : \boldsymbol{\sigma} \\
\longrightarrow\ &(\mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \cdot \mathbf{B} \cdot \mathbf{x_f} \mapsto \mathbf{v} : \boldsymbol{\sigma} \triangleright \mathbf{let\ x_f = 0\ (if\ pr)\ in\ call\ f\ x_f}), \omega - 1, \mathbf{U} \\
&\quad \text{since } \mathbf{H}(-1) \mapsto \mathbf{true} : \mathbf{S} \\
\longrightarrow\ &(\mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \cdot \mathbf{B} \cdot \mathbf{x_f} \mapsto \mathbf{0} : \mathbf{S} \triangleright \mathbf{call\ f\ x_f}), \omega - 2, \mathbf{U} \\
&\quad \text{where } \overline{\mathbf{B}'} \text{ is the current stack and the body of } \mathbf{f} \text{ is } \mathbf{s} \\
\xrightarrow{\texttt{call f 0!}^\mathbf{S}}\ &(\mathbf{C}, \mathbf{H}, \overline{\mathbf{B}'} \cdot \mathbf{x} \mapsto \mathbf{0} : \mathbf{S} \triangleright \mathbf{s}), \omega - 3, \mathbf{U}
\end{aligned}$$

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 $[\![\cdot]\!]^s_n$ 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.
> $[\![\cdot]\!]^s_n$ 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 33 (Single Context Speculation is Safe)) and the binding created in a call is safe (Rule E-L-call).
$\mathbf{f}'' \in \overline{\mathsf{f}_\mathsf{c}}$ and $\mathbf{f} \notin \overline{\mathsf{f}_\mathsf{c}}$ 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 32 (Context Speculation Lasts at Most Omega).
By induction on the reduction and with Lemma 33 (Single Context Speculation is Safe). □

♠

Proof of Lemma 33 (Single Context Speculation is Safe).
By induction on $\mathbf{s}$:

Base skip
assignment
lfence
Inductive call
sequence
letin
if
read
cmove

All cases are trivial, all expressions evaluate to $\mathbf{S}$ due to Lemma 29 (Expression Reductions with Safe Bindings are Safe) and no reduction can load $\mathbf{U}$ values, so the conditions are met.
All actions are tagged $\mathbf{S}$ so they can be related to $\varnothing$. □

♠

Proof of Lemma 34 (Compiled Speculation Lasts at Most Omega).
By induction on the reduction with Lemma 35 (Compiled Speculation is Safe) □

♠

Proof of Lemma 35 (Compiled Speculation is Safe).
By induction on $\mathbf{s}$:

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 $\omega = \mathbf{0}$ then this trivially holds by Rule E-T-init.
If $\omega = \mathbf{n+1}$ then we have two cases:

- $\mathbf{f}$ is compiled code.
  We have two cases:
  1) $\mathbf{B} \triangleright [\![\mathbf{e}]\!]^s \downarrow \mathbf{v} : \boldsymbol{\sigma}$
     We have two cases here:
     a) $\omega > \mathbf{3}$
        By HP we have that $\mathbf{H}(-\mathbf{1}) \mapsto \mathbf{true} : \mathbf{S}$ so we have:
        (for simplicity we only keep track of the top of the stack of execution states)

$$
\begin{aligned}
&(\mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \cdot \mathbf{B} \triangleright \mathbf{let}\ \mathbf{x_f} = [\![\mathbf{e}]\!]^s\ \mathbf{in\ let}\ \mathbf{x_f} = \mathbf{0}\ (\mathbf{if\ pr})\ \mathbf{in\ call\ f}\ \mathbf{x_f}), \omega, \mathbf{U} \\
&\quad \text{assuming } \mathbf{B} \triangleright [\![\mathbf{e}]\!]^s \downarrow \mathbf{v} : \boldsymbol{\sigma} \\
\longrightarrow\ &(\mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \cdot \mathbf{B} \cdot \mathbf{x_f} \mapsto \mathbf{v} : \boldsymbol{\sigma} \triangleright \mathbf{let}\ \mathbf{x_f} = \mathbf{0}\ (\mathbf{if\ pr})\ \mathbf{in\ call\ f}\ \mathbf{x_f}), \omega - \mathbf{1}, \mathbf{U} \\
&\quad \text{since } \mathbf{H}(-\mathbf{1}) \mapsto \mathbf{true} : \mathbf{S} \\
\longrightarrow\ &(\mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \cdot \mathbf{B} \cdot \mathbf{x_f} \mapsto \mathbf{0} : \mathbf{S} \triangleright \mathbf{call\ f}\ \mathbf{x_f}), \omega - \mathbf{2}, \mathbf{U}
\end{aligned}
$$

where $\overline{B'}$ is the current stack and the body of f is $[\![s]\!]^s$

$$\xrightarrow{\texttt{call f 0!}^S} (C, H, \overline{B'} \cdot x \mapsto 0 : S \triangleright [\![s]\!]^s), \omega - 3, U$$

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) $B \triangleright [\![e]\!]^s \downarrow e : \sigma$

If $[\![e]\!]^s$ 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, \overline{B} \cdot B \triangleright \\
& \texttt{let } x_g = [\![e]\!]^s \texttt{ in} \\
& \quad \texttt{let } pr = rd_p\ -1 \texttt{ in} \\
& \quad \texttt{let } x_g = 0\ (\texttt{if } pr) \texttt{ in} \\
& \quad \texttt{ifz } x_g \texttt{ then let } x = rd_p\ -1 \texttt{ in } -1 :=_p x \vee \neg x_g; [\![s]\!]^s \\
& \quad\quad \texttt{else let } x = rd_p\ -1 \texttt{ in } -1 :=_p x \vee x_g; [\![s']\!]^s \\
\longrightarrow\ & C, H, \overline{B} \cdot B \cdot x_g \mapsto v : \sigma \triangleright \\
& \texttt{let } pr = rd_p\ -1 \texttt{ in} \\
& \quad \texttt{let } x_g = 0\ (\texttt{if } pr) \texttt{ in} \\
& \quad \texttt{ifz } x_g \texttt{ then let } x = rd_p\ -1 \texttt{ in } -1 :=_p x \vee \neg x_g; [\![s]\!]^s \\
& \quad\quad \texttt{else let } x = rd_p\ -1 \texttt{ in } -1 :=_p x \vee x_g; [\![s']\!]^s \\
\longrightarrow\ & C, H, \overline{B} \cdot B \cdot x_g \mapsto v : \sigma \cdot pr \mapsto \texttt{true} : S \triangleright \\
& \quad \texttt{let } x_g = 0\ (\texttt{if } pr) \texttt{ in} \\
& \quad \texttt{ifz } x_g \texttt{ then let } x = rd_p\ -1 \texttt{ in } -1 :=_p x \vee \neg x_g; [\![s]\!]^s \\
& \quad\quad \texttt{else let } x = rd_p\ -1 \texttt{ in } -1 :=_p x \vee x_g; [\![s']\!]^s \\
& \quad \text{this step is key: note that } x_g \text{ becomes } S \\
\xrightarrow{\texttt{if(0)}^S}\ & C, H, \overline{B} \cdot B \cdot x_g \mapsto 0 : S \cdot pr \mapsto \texttt{true} : S \triangleright \\
& \quad \texttt{ifz } x_g \texttt{ then let } x = rd_p\ -1 \texttt{ in } -1 :=_p x \vee \neg x_g; [\![s]\!]^s \\
& \quad\quad \texttt{else let } x = rd_p\ -1 \texttt{ in } -1 :=_p x \vee x_g; [\![s']\!]^s \\
\xrightarrow{\texttt{read}(-1)^S}\ & C, H, \overline{B} \cdot B \cdot x_g \mapsto 0 : S \cdot pr \mapsto \texttt{true} : S \triangleright \\
& \texttt{let } x = rd_p\ -1 \texttt{ in } -1 :=_p x \vee \neg x_g; [\![s]\!]^s \\
\xrightarrow{\texttt{read}(-1)^S}\ & C, H, \overline{B} \cdot B \cdot x_g \mapsto 0 : S \cdot pr \mapsto \texttt{true} : S \cdot x \mapsto \texttt{true} : S \triangleright \\
& -1 :=_p x \vee \neg x_g; [\![s]\!]^s \\
\xrightarrow{\texttt{write}(-1)^S}\ & C, H \cup -1 \mapsto \texttt{true} : S, \overline{B} \cdot B \cdot x_g \mapsto 0 : S \cdot pr \mapsto \texttt{true} : S \cdot x \mapsto \texttt{true} : S \triangleright \\
& [\![s]\!]^s
\end{aligned}$$

The rest holds by IH so long as the states are related by $\overset{S}{\sim}$, which is trivially true and if the trace is related to $\varnothing$ by $\approx$.
We have this trace

$$\xrightarrow{\texttt{read}(-1)^S \cdot \texttt{if}(0)^S \cdot \texttt{read}(-1)^S \cdot \texttt{write}(-1)^S}$$

and each action is related to $\varnothing$ by Rules Action Relation - epsi alpha and Action Relation - epsi heap, so the whole trace is related to $\varnothing$.

Thus this case holds.

read This is analogous to the if case.

private read This is analogous to the if case.

□

♠

Proof of Lemma 30 (Any Speculation from Related States is Safe).

This proof proceeds by induction on the stack of configurations.

Base Empty stack:

By Lemma 31 (Speculation Lasts at Most Omega) we have the first reductions and $\varnothing \approx \overline{\lambda^{\sigma}}$:

$$\begin{aligned} & \mathbf{w}(\mathbf{C}, \mathbf{H_b}, \overline{\mathbf{B_b}} \triangleright (\mathbf{s_b})_{\overline{\mathbf{f_b}}}, \bot, \mathbf{S}) \cdot (\mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \triangleright (\mathbf{s})_{\overline{\mathbf{f}}}, \omega, \mathbf{U}) \\ \xrightarrow{\overline{\lambda^{\sigma}}} \; & \mathbf{w}(\mathbf{C}, \mathbf{H_b}, \overline{\mathbf{B_b}} \triangleright (\mathbf{s_b})_{\overline{\mathbf{f_b}}}, \bot, \mathbf{S}) \cdot (\mathbf{C}, \mathbf{H'}, \overline{\mathbf{B'}} \triangleright (\mathbf{s'})_{\overline{\mathbf{f'}}}, \mathbf{0}, \mathbf{U}) \\ \xrightarrow{\texttt{rlb}} \; & \mathbf{w}(\mathbf{C}, \mathbf{H_b}, \overline{\mathbf{B_b}} \triangleright (\mathbf{s_b})_{\overline{\mathbf{f_b}}}, \bot, \mathbf{S}) \end{aligned}$$

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 28 (Generalised Backward Simulation for SLH).

We have these kinds of reductions:

- compiled-to-compiled code: this holds by Theorem 26 (Backward Simulation for Compiled Statements in SLH);
- backtranslated-to-backtranslated code: this holds by Lemma 17 (Backward Simulation for Backtranslated Statements);
- compiled-to-backtranslated or backtranslated-to-compiled code: this is analogous to the cases discussed in Section I-A4 (Simulation and Relation for Backtranslated Code) since these do not trigger any speculation.

□

♠

Proof of Theorem 27 (Correctness of the Backtranslation for SLH).

By Theorem 28 (Generalised Backward Simulation for SLH) with Lemma 24 (Initial States are Related for SLH). □

♠

Proof of Theorem 24 (Our SLH compiler is RSSC).

Instantiate A with $\langle\!\langle \mathbf{A} \rangle\!\rangle_c^f$.

This holds by Theorem 27 (Correctness of the Backtranslation for SLH). □

♠

Proof of Theorem 25 (Our inter-procedural SLH compiler is RSSC).

Instantiate A with $\langle\!\langle \mathbf{A} \rangle\!\rangle_c^f$.

This holds by a variation of Theorem 27 (Correctness of the Backtranslation for SLH) to account for the different state relation ($\overset{s}{\propto}$) which in turns holds by a variation of both Theorem 28 (Generalised Backward Simulation for SLH) and Lemma 24 (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 26 (Backward Simulation for Compiled Statements in SLH) and Lemma 17 (Backward Simulation for Backtranslated Statements) and Section I-A4 (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 28 (Forward Simulation for Compiled Statements in SLH), which relies on

Lemma 30 (Any Speculation from Related States is Safe) and then on Lemma 31 (Speculation Lasts at Most Omega) and then on an adaptation of Lemma 35 (Compiled Speculation is Safe), where the new trace relation plays a role.

We provide only the proof of the last theorem (in Lemma 36 (Compiled Speculation is Safe Inter-procedurally)) since it is the only one with any change. □

♠

Proof of Lemma 36 (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 $\omega = \mathbf{0}$ then this trivially holds by Rule E-T-init.

If $\omega = \mathbf{n} + \mathbf{1}$ then we have two cases:

- f is compiled code.
  We have two cases:
  1) $\mathbf{B} \triangleright [\![e]\!]^s \downarrow \mathbf{v} : \boldsymbol{\sigma}$
     We have two cases here:
     a) $\omega > \mathbf{1}$
        Compiled code starts with an lfence so the execution is immediately rolled back and this case holds by Rule E-T-speculate-rollback.
     b) $\omega \leq \mathbf{1}$
        In this case the execution will run out of steps and the case holds by Rule E-T-speculate-rollback.
  2) $\mathbf{B} \triangleright [\![e]\!]^s \downarrow \mathbf{e} : \boldsymbol{\sigma}$
     If $[\![e]\!]^s$ 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 33 of Lemma 35 (Compiled Speculation is Safe).

read This is analogous to the if case.

private read This is analogous to the if case.

□

♠

APPENDIX L
T PROGRAMS FOR SECTION V SNIPPETS

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

1) Snippet of Listing 1:

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

2) Snippet of Listing 2:

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

3) Snippet of Listing 3:

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

4) Snippet of Listing 4:

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

5) Snippet of Listing 5:

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

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

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

7) Snippet of Listing 7:

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

8) Snippet of Listing 8:

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

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

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

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

APPENDIX M
FORMAL DETAILS THAT SLH IS NOT *RSNIP*

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

The attacker is the same:

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

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

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

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

$$\begin{aligned} t\restriction_{nse} = t'\restriction_{nse} = {} & \texttt{call get 8?}^\text{S} \cdot \texttt{read(1)}^\text{S} \cdot \texttt{read}(-(n_a - 1 + 8 + 1))^\text{S} \cdot \\ & \texttt{if(1)}^\text{S} \cdot \texttt{read}(-1)^\text{S} \cdot \texttt{write}(-1)^\text{S} \cdot \texttt{ret!}^\text{S} \end{aligned}$$

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

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

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

$$\begin{aligned} t\restriction_{nse} = t'\restriction_{nse} = {} & \texttt{call get 8?}^\text{S} \cdot \texttt{read(1)}^\text{S} \cdot \texttt{read}(-(n_a + 8))^\text{S} \cdot \\ & \texttt{if(1)}^\text{S} \cdot \texttt{ret!}^\text{S} \end{aligned}$$

APPENDIX N
OMITTED SEMANTICS RULES

The semantics relies on these auxiliary rules.

$$\frac{((f' \in \overline{I} \wedge f \in \overline{I}) \vee (f' \notin \overline{I} \wedge f \notin \overline{I}))}{\overline{I} \vdash f, f' : internal}\ \text{(Jump-Internal)} \qquad \frac{f \in \overline{I} \wedge f' \notin \overline{I}}{\overline{I} \vdash f, f' : in}\ \text{(Jump-IN)} \qquad \frac{f \notin \overline{I} \wedge f' \in \overline{I}}{\overline{I} \vdash f, f' : out}\ \text{(Jump-OUT)}$$

Below, we use $[\cdot]$ as the standard operator interpretation.

$$\boxed{B \triangleright e \ \downarrow\ v : \sigma}$$

$$\frac{}{B \triangleright v \ \downarrow\ v : S}\ \text{(E-val)} \qquad \frac{B(x) = v : \sigma}{B \triangleright x \ \downarrow\ v : \sigma}\ \text{(E-var)}$$

$$\frac{B \triangleright e \ \downarrow\ n : \sigma \quad B \triangleright e' \ \downarrow\ n' : \sigma' \quad n'' = [n \oplus n'] \quad \sigma'' = \sigma \sqcup \sigma'}{B \triangleright e \oplus e' \ \downarrow\ n'' : \sigma''}\ \text{(E-op)}$$

$$\frac{B \triangleright e \ \downarrow\ n : \sigma \quad B \triangleright e' \ \downarrow\ n' : \sigma' \quad n'' = [n \otimes n'] \quad \sigma'' = \sigma \sqcup \sigma'}{B \triangleright e \otimes e' \ \downarrow\ n'' : \sigma''}\ \text{(E-comparison)}$$

For technical reasons, writing the public heap produces an observation that records both address and value being written since both are accessible by the attacker.

$$\boxed{C, H, \overline{B} \triangleright s \xrightarrow{\lambda^\sigma} C', H', \overline{B'} \triangleright s'}$$

$$\frac{}{C, H, \overline{B} \triangleright skip; s \xrightarrow{\epsilon} C, H, \overline{B} \triangleright s}\ \text{(E-sequence)}$$

$$\frac{C, H, \overline{B} \triangleright s \xrightarrow{\lambda^\sigma} C, H', \overline{B'} \triangleright s'}{C, H \triangleright s; s'' \xrightarrow{\lambda^\sigma} C, H', \overline{B'} \triangleright s'; s''}\ \text{(E-step)}$$

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

$$\frac{B \triangleright e \ \downarrow\ n : \sigma \quad n > 0}{C, H, \overline{B} \cdot B \triangleright ifz\ e\ then\ s\ else\ s' \xrightarrow{(\mathtt{if}(n))^\sigma} C, H, \overline{B} \cdot B \triangleright s}\ \text{(E-if-false)}$$

$$\frac{B \triangleright e \ \downarrow\ v : \sigma}{C, H, \overline{B} \cdot B \triangleright let\ x = e\ in\ s \xrightarrow{\epsilon} C, H, \overline{B} \cdot B \cup x \mapsto v : \sigma \triangleright s}\ \text{(E-letin)}$$

$$\frac{B \triangleright e \ \downarrow\ n : \sigma \quad B \triangleright e' \ \downarrow\ v : \sigma'' \quad H = H_1; |n| \mapsto v' : \sigma'; H_2 \quad H' = H_1; |n| \mapsto v : S; H_2}{C, H, \overline{B} \cdot B \triangleright e := e' \xrightarrow{\mathtt{write}(|n| \mapsto v)^{\sigma \sqcup \sigma''}} C, H', \overline{B} \cdot B \triangleright skip}\ \text{(E-write)}$$

$$\frac{B \triangleright e \ \downarrow\ n : \sigma' \quad H = H_1; |n| \mapsto v : \sigma; H_2}{C, H, \overline{B} \cdot B \triangleright let\ x = rd\ e\ in\ s \xrightarrow{\mathtt{read}(|n|)^{\sigma'}} C, H, \overline{B} \cdot B \cup x \mapsto v : \sigma \triangleright s}\ \text{(E-read)}$$

$$\frac{B \triangleright e \ \downarrow\ n : \sigma \quad B \triangleright e' \ \downarrow\ v : \sigma'' \quad n_a = -|n| \quad H = H_1; n_a \mapsto v' : \sigma'; H_2 \quad H' = H_1; n_a \mapsto v : \sigma; H_2}{C, H, \overline{B} \cdot B \triangleright e :=_p e' \xrightarrow{\mathtt{write}(n_a)^\sigma} C, H', \overline{B} \cdot B \triangleright skip}\ \text{(E-write-prv)}$$

$$\frac{B \triangleright e \ \downarrow\ n : \sigma' \quad n_a = -|n| \quad H = H_1; n_a \mapsto v : \sigma; H_2 \quad \sigma'' = \sigma \sqcup \sigma'}{C, H, \overline{B} \cdot B \triangleright let\ x = rd_p\ e\ in\ s \xrightarrow{\mathtt{read}(n_a)^{\sigma''}} C, H, \overline{B} \cdot B \cup x \mapsto v : U \triangleright s}\ \text{(E-read-prv)}$$

$$\frac{C.\mathtt{intfs} \vdash f, f' : internal \quad \overline{f'} = \overline{f''}; f' \quad f(x) \mapsto s; return; \in C.\mathtt{funs} \quad B \triangleright e \ \downarrow\ v : \sigma}{C, H, \overline{B} \cdot B \triangleright (call\ f\ e)_{\overline{f'}} \xrightarrow{\epsilon} C, H, \overline{B} \cdot B \cdot x \mapsto v : \sigma \triangleright (s; return;)_{\overline{f'};f}}\ \text{(E-call-internal)}$$

$$\frac{\overline{f'} = \overline{f''}; f' \quad f(x) \mapsto s; return; \in C.\mathtt{funs} \quad C.\mathtt{intfs} \vdash f', f : out \quad B \triangleright e \ \downarrow\ v : \sigma}{C, H, \overline{B} \cdot B \triangleright (call\ f\ e)_{\overline{f'}} \xrightarrow{\mathtt{call}\ f\ v!^\sigma} C, H, \overline{B} \cdot B \cdot x \mapsto v : S \triangleright (s; return;)_{\overline{f'};f}}\ \text{(E-callback)}$$

$$\frac{\overline{f'} = \overline{f''}; f' \quad f(x) \mapsto s; return; \in C.\mathtt{funs} \quad C.\mathtt{intfs} \vdash f', f : in \quad B \triangleright e \ \downarrow\ v : \sigma}{C, H, \overline{B} \cdot B \triangleright (call\ f\ e)_{\overline{f'}} \xrightarrow{\mathtt{call}\ f\ v?^\sigma} C, H, \overline{B} \cdot B \cdot x \mapsto v : S \triangleright (s; return;)_{\overline{f'};f}}\ \text{(E-call)}$$

$$\frac{\overline{f'} = \overline{f''}; f' \quad C.\mathtt{intfs} \vdash f, f' : internal}{C, H, \overline{B} \cdot B \triangleright (return;)_{\overline{f'};f} \xrightarrow{\epsilon} C, H, \overline{B} \triangleright (skip)_{\overline{f'}}}\ \text{(E-ret-internal)}$$

$$\frac{\overline{f'} = \overline{f''}; f' \quad C.\mathtt{intfs} \vdash f, f' : in}{C, H, \overline{B} \cdot B \triangleright (return;)_{\overline{f'};f} \xrightarrow{\mathtt{ret}?^S} C, H, \overline{B} \triangleright (skip)_{\overline{f'}}}\ \text{(E-retback)}$$

$$\frac{\overline{f'} = \overline{f''}; f' \quad C.\mathtt{intfs} \vdash f, f' : out}{C, H, \overline{B} \cdot B \triangleright (return;)_{\overline{f'};f} \xrightarrow{\mathtt{ret}!^S} C, H, \overline{B} \triangleright (skip)_{\overline{f'}}}\ \text{(E-return)}$$

APPENDIX O
CROSS-LANGUAGE STATE RELATION

We present first the relations for the lfence compiler (Section O-A) and then those for SLH (Section O-B).

A. Relations for the Lfence-Compiler

$$\boxed{\text{Heap relation } \overset{H}{\approx} \quad \text{Value relation } \overset{V}{\approx}}$$

$$\frac{}{\varnothing \overset{H}{\approx} \varnothing}\ \text{(Heap - base)} \qquad \frac{\mathsf{H} \overset{H}{\approx} \mathbf{H} \quad \mathsf{z} \overset{V}{\approx} \mathbf{z} \quad \mathsf{v} \overset{V}{\approx} \mathbf{v} \quad \sigma \equiv \boldsymbol{\sigma}}{\mathsf{H}; \mathsf{z} \mapsto \mathsf{v} : \sigma \overset{H}{\approx} \mathbf{H}; \mathbf{z} \mapsto \mathbf{v} : \boldsymbol{\sigma}}\ \text{(Heap - ind)}$$

$$\frac{\mathsf{H} \overset{H}{\approx} \mathbf{H} \quad \mathsf{H}' \overset{H}{\approx} \mathbf{H}' \quad \mathsf{v} \overset{V}{\approx} \mathbf{v} \quad \sigma \equiv \boldsymbol{\sigma}}{\mathsf{H}; 0 \mapsto \mathsf{v} : \sigma; \mathsf{H}' \overset{H}{\approx} \mathbf{H}; \mathbf{0} \mapsto \mathbf{v} : \boldsymbol{\sigma}; \mathbf{H}'}\ \text{(Heap - start)} \qquad \frac{\mathsf{z} \equiv \mathbf{z} \quad z \in \mathbb{Z}}{\mathsf{z} \overset{V}{\approx} \mathbf{z}}\ \text{(Value - num)}$$

The component and state relations are indexed by the list of functions defined by the compiled code.

- $\boxed{\text{Binding relation } \overset{B}{\approx} \quad \text{Component relation } \overset{C}{\approx} \quad \text{State relation } \overset{S}{\approx}}$

$$\frac{}{\varnothing \overset{B}{\approx} \varnothing}\ (\text{Binding - base}) \qquad \frac{\mathbf{B} \overset{B}{\approx} \mathbf{B} \quad \mathbf{v} \overset{V}{\approx} \mathbf{v} \quad \sigma \equiv \sigma}{\mathbf{B} \cdot \mathbf{x} \mapsto \mathbf{v} : \sigma \overset{B}{\approx} \mathbf{B} \cdot \mathbf{x} \mapsto \mathbf{v} : \sigma}\ (\text{Binding - ind})$$

$$\frac{\overline{\mathbf{B}} \overset{B}{\approx} \overline{\mathbf{B}} \quad \mathbf{B} \overset{B}{\approx} \mathbf{B}}{\overline{\mathbf{B}} \cdot \mathbf{B} \overset{B}{\approx} \overline{\mathbf{B}} \cdot \mathbf{B}}\ (\text{Bindings})$$

$$\frac{\begin{array}{c}\forall \mathsf{f} \in \overline{\mathsf{f}}.\ \text{if } \mathsf{f}(\mathsf{x}) \mapsto \mathsf{s} \in \overline{\mathsf{F}} \text{ then } \mathbf{f}(\mathbf{x}) \mapsto [\![\mathsf{s}]\!]^{f} \in \overline{\mathbf{F}} \\ \forall \mathbf{f}(\mathbf{x}) \mapsto \mathbf{s} \in \overline{\mathbf{F}} \text{ if } \mathbf{f} \notin \overline{\mathsf{f}} \text{ then } \mathsf{f}(\mathsf{x}) \mapsto \langle\!\langle \mathsf{s} \rangle\!\rangle^{f}_{c} \in \overline{\mathsf{F}} \\ \overline{\mathsf{I}} \equiv \overline{\mathsf{I}}\end{array}}{\overline{\mathsf{F}}; \overline{\mathsf{I}} \overset{C}{\approx}_{\overline{\mathsf{f}}} \overline{\mathbf{F}}; \overline{\mathbf{I}}}\ (\text{Components})$$

$$\frac{\overline{\mathbf{B}} \overset{B}{\approx} \overline{\mathbf{B}} \quad \mathbf{H} \overset{H}{\approx} \mathbf{H} \quad \overline{\mathsf{f}} \equiv \overline{\mathsf{f}} \quad \mathbf{C} \overset{C}{\approx}_{\overline{\mathbf{f}}''} \mathbf{C}}{\mathsf{C}, \mathsf{H}, \overline{\mathsf{B}} \triangleright (\mathsf{s})_{\overline{\mathsf{f}}} \overset{S}{\approx}_{\overline{\mathbf{f}}''} (\mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \triangleright (\mathsf{s})_{\overline{\mathbf{f}}}, \bot, \mathbf{S})}\ (\text{States})$$

B. Relations for SLH

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

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

$$\frac{}{\varnothing \overset{B}{\approx}^{\varnothing} \varnothing}\ (\text{Binding - base}) \qquad \frac{\mathbf{B} \overset{B}{\approx}^{\mathbf{D}} \mathbf{B} \quad \mathbf{v} \overset{V}{\approx} \mathbf{v} \quad \sigma \equiv \sigma}{\mathbf{B} \cdot \mathbf{x} \mapsto \mathbf{v} : \sigma \overset{B}{\approx}^{\mathbf{D}} \mathbf{B} \cdot \mathbf{x} \mapsto \mathbf{v} : \sigma}\ (\text{Binding - ind})$$

$$\frac{\mathbf{B} \overset{B}{\approx}^{\mathbf{D}} \mathbf{B} \quad \mathbf{x} \in \mathbf{D}}{\mathbf{B} \overset{B}{\approx}^{\mathbf{D}} \mathbf{B} \cdot \mathbf{x} \mapsto \mathbf{v} : \sigma}\ (\text{Binding - ind}) \qquad \frac{\overline{\mathbf{B}} \overset{B}{\approx}^{\overline{\mathbf{D}}} \overline{\mathbf{B}} \quad \mathbf{B} \overset{B}{\approx}^{\mathbf{D}} \mathbf{B}}{\overline{\mathbf{B}} \cdot \mathbf{B} \overset{B}{\approx}^{\overline{\mathbf{D}} \cdot \mathbf{D}} \overline{\mathbf{B}} \cdot \mathbf{B}}\ (\text{Bindings})$$

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

$$\frac{}{\varnothing \overset{H}{\approx} \varnothing}\ (\text{Heap - base}) \qquad \frac{\mathbf{H} \overset{H}{\approx} \mathbf{H} \quad \mathbf{z} - 1 \overset{V}{\approx} \mathbf{z} \quad \mathbf{v} \overset{V}{\approx} \mathbf{v} \quad \sigma \equiv \sigma}{\mathsf{H}; \mathsf{z} \mapsto \mathsf{v} : \sigma \overset{H}{\approx} \mathbf{H}; \mathbf{z} \mapsto \mathbf{v} : \sigma}\ (\text{Heap - negative})$$

$$\frac{\mathbf{H} \overset{H}{\approx} \mathbf{H} \quad \mathbf{H}' \overset{H}{\approx} \mathbf{H}' \quad \mathbf{v} \overset{V}{\approx} \mathbf{v} \quad \sigma \equiv \sigma}{\mathsf{H}; 0 \mapsto \mathsf{v} : \sigma; \mathsf{H}' \overset{H}{\approx} \mathbf{H}; -1 \mapsto \mathbf{false} : \mathbf{S}; \mathbf{0} \mapsto \mathbf{v} : \sigma; \mathbf{H}'}\ (\text{Heap - start})$$

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

$$\frac{\Omega \overset{S}{\approx}_{\overline{\mathsf{f}}}^{\overline{\mathbf{D}}} ((\mathbf{\Omega}, \bot, \mathbf{S})) \quad \forall \mathbf{\Omega_s} \in \overline{\mathbf{\Omega}},\ \Omega \overset{S}{\approx}_{\overline{\mathsf{f}}} \mathbf{\Omega_s}}{\Omega \overset{S}{\approx}_{\overline{\mathsf{f}}}^{\overline{\mathbf{D}}} ((\mathbf{\Omega}, \bot, \mathbf{S}) \cdot \overline{(\mathbf{\Omega}, \mathbf{W}, \mathbf{U})})}\ (\text{States relation})$$

$$\frac{\overline{\mathbf{B}} \overset{B}{\approx}^{\overline{\mathbf{D}}} \overline{\mathbf{B}} \quad \mathbf{H} \overset{H}{\approx} \mathbf{H} \quad \overline{\mathsf{f}} \equiv \overline{\mathsf{f}} \quad \mathbf{C} \overset{C}{\approx}_{\overline{\mathbf{f}}''} \mathbf{C}}{\mathsf{C}, \mathsf{H}, \overline{\mathsf{B}} \triangleright (\mathsf{s})_{\overline{\mathsf{f}}} \overset{S}{\approx}_{\overline{\mathbf{f}}''}^{\overline{\mathbf{D}}} (\mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \triangleright (\mathsf{s})_{\overline{\mathbf{f}}}, \bot, \mathbf{S})}\ (\text{Base States})$$

$$\frac{(\text{if } \mathbf{f} \notin \overline{\mathbf{f}'} \text{ then } \vdash \mathbf{B} : \overset{B}{\diamond}) \quad \mathbf{C} \overset{C}{\approx}_{\overline{\mathbf{f}'}} \mathbf{C} \quad \vdash \mathbf{H} : \overset{H}{\diamond}}{\mathsf{C}, \mathsf{H}, \overline{\mathsf{B}} \triangleright (\mathsf{s})_{\overline{\mathsf{f}}} \overset{S}{\diamond}_{\overline{\mathbf{f}}'} \mathbf{C}, \mathbf{H}, \overline{\mathbf{B}} \cdot \mathbf{B} \triangleright (\mathsf{s})_{\overline{\mathbf{f}} \cdot \mathbf{f}}}\ (\text{Single state relation - ctx})$$

$$\frac{\forall \mathbf{n} \mapsto \mathbf{v} : \sigma \in \mathbf{H} \text{ if } \mathbf{n} \geq \mathbf{0} \text{ then } \sigma = \mathbf{S} \quad \mathbf{H}(-1) = \mathbf{true} : \mathbf{S}}{\vdash \mathbf{H} : \overset{H}{\diamond}}\ (\text{Heap relation same})$$

$$\frac{}{\vdash \varnothing : \overset{B}{\diamond}}\ (\text{Target Bindings ok base}) \qquad \frac{\vdash \mathbf{B} : \overset{B}{\diamond}}{\vdash \mathbf{B} \cdot \mathbf{x} \mapsto \mathbf{v} : \mathbf{S} : \overset{B}{\diamond}}\ (\text{Target Bindings ok sing})$$

References

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

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

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

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

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

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

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

[8] Anonymous Authors. Exorcising Spectres with Secure Compilers – Technical Report with Proofs and Details. https://www.dropbox.com/s/m0scanttbo95i80/tr.pdf?dl=0 Viewer Info has been disabled for double-blind review.

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

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

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

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

[13] Ken Biba. Integrity considerations for secure computer systems. page 68, 04 1977.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[30] Intel. Retpoline: A branch target injection mitigation. https://software.intel.com/security-software-guidance/api-app/sites/ 2018.

[31] Intel. Using Intel Compilers to Mitigate Speculative Execution Side-Channel Issues. https://software.intel.com/en-us/articles/using-intel-compilers-to-m 2018.

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

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

[34] Paul Kocher. Spectre mitigations in Microsoft's C/C++ compiler. https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigat 2018.

[35] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *S&P '19*. IEEE, 2019.

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

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

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

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

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

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

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

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

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

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

[46] Marco Patrignani. Why should anyone use colours? or, syntax highlighting beyond code snippets, 2020.

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

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

[49] Marco Patrignani and Deepak Garg. Secure Compilation and Hyperproperties Preservation. In *CSF 2017*, 2017.

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

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

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

[53] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J.Sel. A. Commun.*, 21(1):5–19, September 2006.

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

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

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

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

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

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

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

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

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

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