Towards More Efficient and Trustworthy Formally Secure Verification Against Speculative Side-Channel Attacks

Student:
Jonathan Baumann
ENS Paris-Saclay

Supervisor:
Cătălin Hriţcu
FOVSEC group, MPI-SP

General Context

Modern processors employ a performance optimization known as speculative execution, in which the processor may predict the outcome of branches or indirect jumps in order to begin executing the following instructions earlier. In case the prediction is incorrect, all incorrectly executed instructions are rolled back so that they do not affect the final result. However, speculatively executed instructions still leave traces in the microarchitectural state (e.g. via caches) that an attacker can exploit to exfiltrate secrets, leading to the well-known class of Spectre attacks.

As speculative execution vulnerabilities are fundamentally a weakness of the hardware design, intuitively, it might seem most promising to attempt to mitigate them in hardware. However, hardware mitigations are prohibitively expensive, and can not protect currently existing hardware. Therefore, it is clear that software defenses against speculative execution attacks are needed.

With no other considerations, complete software mitigations against Spectre attacks are easy: Serializing instructions such as the x86 `lfence` instruction, which does not begin executing until all previous instructions have completed and prevents later instructions from executing until it has completed [1], can fully prevent speculative execution. Therefore, attacks can be prevented by simply inserting serializing instructions after every branch[1]. This is, for example, implemented in the Intel C compiler [1], [3]. However, fully preventing speculative execution comes at a huge performance cost [4]. Thus, much work has been done on mitigations with lower overhead, whether via binary-level analysis (e.g. Spectector [5]) or in compilers [6], [7]. However, most defenses are either still prohibitively expensive (e.g. Ultimate SLH [8]) or lack formal analysis (e.g. LLVM-SLH [6]). Targeted, efficient mitigations with formal security guarantees are currently limited to the domain of cryptography, such as Selective SLH [7] for Jasmin [9]. However, even in those cases, proofs are usually only done on paper and not mechanized.

Further, most mitigations so far only target Spectre-PHT attacks (where the outcome of a branch is mispredicted), while ignoring other classes of Spectre attacks.

Research Problem

This report focuses on *mechanized* security proofs for compiler-based mitigations against speculative side-channel attacks. The aim is to investigate two aspects: One question is whether existing compiler mitigations for restricted classes of programs can be extended to handle more general classes of programs, how the security proofs need to be adjusted in this case, and how they can be mechanized in proof assistants. The other question concerns the models of speculative execution underpinning such security proofs, what relationships can be established between different models, and how they can be formalized and mechanized. Of particular interest here are relationships between abstract models and models that are closer to hardware.

Contribution

I present two main contributions.

For the first question, I present how FlexSLH, a new compiler-based Spectre mitigation which generalizes previous mitigations, can be proven to ensure *relative security* for all programs. The proof relies on generalizing the result

[1] Or other source of speculative execution, see C. Canella *et al.* [2] for an overview.

of a static information-flow analysis to a typing-like well-labeledness property, which, unlike the information-flow analysis result, is preserved during execution.

Regarding models of speculative execution, I present both a mechanization of a proof by G. Barthe *et al.* [10] establishing an equivalence between directive-based models (in which the attacker has explicit control over branch prediction at runtime) with and without rollbacks, as well as a new result that observational equivalence in the directive-based model implies observational equivalence in an always-mispredict model (as introduced by M. Guarnieri *et al.* [5] for Spectector). The latter result in particular also highlights some restrictions and a potential flaw which can easily be missed[2] when relying purely on more abstract models of speculative execution.

Arguments Supporting Validity

A core aspect of this work is to increase the trustworthiness of security proofs by using fully mechanized proofs instead of relying on paper proofs. Therefore, all work presented here has been formalized in the Rocq proof assistant[3].

The full Rocq developments can be found at https://github.com/secure-compilation/fslh-rocq for Section 3 and https://github.com/secure-compilation/comparing_speculative_semantics_rocq for Section 4. Please note, however, that some minor lemmas concerning maps are admitted in the published versions, as they are given (in this exact form) as exercises in the upcoming Security Foundations book [12] (and lectures based on it).

Summary and Future Work

I present formal, mechanized proofs of the relative security of FSLH on arbitrary (as opposed to only well-typed) programs, as well as the relationships between directive-based semantics with and without rollbacks and Spectector-style always-mispredict semantics. Both leave much potential for future work.

Regarding FSLH, the mitigation currently only exists for a small toy language, and does not yet have a proper real-world implementation. In fact, core questions remain open, such as at what point during compilation information-flow analysis should be performed so that the results remain valid. This is particularly challenging since certain optimizations in LLVM are known to break cryptographic constant-time code [13], [14], [15], e.g. by inserting unintended branches. While a full security proof for a real-world compiler may not be tractable, it may be of interest to develop security proofs of FlexSLH for (toy) languages with more advanced features, such as dynamic allocation, to ensure that it still ensures relative security in the presence of such features.

Further, FSLH so far only targets Spectre-PHT attacks, and does not mitigate other speculative execution vulnerabilities. While mitigations for other variants will be conceptually different, our long-term goal is to develop a mechanized security proof of a combined mitigation against all known speculative execution attacks.

Regarding different speculation models, this work is only a first step in bridging a large gap between the abstract models that are suited for formal proofs, and e.g. hardware-software-contracts [16], which accurately describe the hardware in question. Nonetheless, as we already uncovered some potential pitfalls when using abstract models, a first step will be to investigate whether prior work handles such cases correctly.

Notes and Acknowledgments

Section 3 is mostly adapted from Section 7 of the FSLH paper, presented at CSF 2025 [11], which received a distinguished paper award. My main contribution to this paper is in the proof of relative security for FvSLH^{v}, in particular for the ideal semantics (as my coauthors had already completed the proof of backwards compiler correctness), although I was also heavily involved in editing the final version of the paper.

As a consequence, this report will also use the same setting and notations as that paper (where applicable).

[2] For example in our own published work on FSLH [11], although it should only require adjustments to proof details and is not a flaw with the mitigation itself.

[3] Previously known as Coq. https://rocq-prover.org/

1 Setting

1.1 AWhile

For this report, we will use a simple toy language called AWhile, a variant of the While language [17] extended with arrays and branchless conditionals, which is also used in the FSLH paper [11] and the upcoming Security Foundations book [12].

$e \in aexp ::= n \in \mathbb{N}$	*number*	$c \in com ::=$ `skip`	*do nothing*
$\mid$ `x` $\in \mathcal{V}$	*scalar variable*	$\mid$ `X` $:= e$	*assignment*
$\mid op_{\mathbb{N}}(e, ..., e)$	*arithmetic operator*	$\mid c; c$	*sequence*
$\mid b \,?\, e : e$	*constant-time conditional*	$\mid$ `if` b `then` c `else` c	*conditional*
$b \in bexp ::= \mathbb{T} \mid \mathbb{F}$	*boolean*	$\mid$ `while` b `do` c	*loop*
$\mid cmp(e, e)$	*arithmetic comparison*	$\mid$ `X` $\leftarrow$ `a`$[e]$	*read from array*
$\mid op_{\mathbb{B}}(b, ..., b)$	*boolean operator*	$\mid$ `a`$[e] \leftarrow e$	*write to array*

Fig. 1. Syntax of AWhile

The syntax of this language is shown in Fig. 1. Of particular note is the constant-time conditional $b \,?\, e_1 : e_2$, which selects the value of e_1 or e_2 depending on whether b evaluates to `true`. This is similar to e.g. conditional move instructions in x86.

For sequential execution, states are triples $\langle c,\ \rho,\ \mu \rangle$ of a command, a scalar state and an array state. The scalar state ρ maps scalar variables to integer values, and is used in the evaluation of arithmetic and boolean expressions, written $[\![\cdot]\!]_\rho$. Updates are denoted $[\texttt{X} \mapsto v]\rho$. The array state μ assigns each array variable a fixed size $\mid \texttt{a} \mid_\mu$, and defines a lookup function $[\![\texttt{a}[i]]\!]_\mu$. Updates are written $[\texttt{a}[i] \mapsto v]\mu$. Unlike scalar variables, the contents of arrays can not be used directly in expressions, and are only accessible via read and write commands.

We use a standard small-step semantics for sequential execution, with the addition of optional *observations* produced during execution (see Section 1.2). We write $\langle c,\ \rho,\ \mu \rangle \xrightarrow{o} \langle c',\ \rho',\ \mu' \rangle$ (or $\langle c,\ \rho,\ \mu \rangle \xrightarrow{\bullet} \langle c',\ \rho',\ \mu' \rangle$ if no observation is produced), as well as $\langle c,\ \rho,\ \mu \rangle \xrightarrow{\mathcal{O}}{}^* \langle c',\ \rho',\ \mu' \rangle$ for multi-step execution, where $\mathcal{O}$ is a list of observations. The full sequential semantics is given in Appendix B.

1.2 Leakage Model

Our work is based on the *cryptographic constant-time* leakage model, in which an attacker can observe control flow and the addresses of all memory operations (read and write), and which is standard in the field of cryptography [18], [19], [20], [21]. This leakage model describes what a co-located attacker on the same processor is able to infer using standard cache and timing attacks such as Flush+Reload [22]. There are thus three instructions in AWhile which produce leakage:

- `if` be `then` $\cdot$ `else` $\cdot$ instructions produce an observation $branch\ b$, where b is the value of the expression be in the current state
- $X \leftarrow a[ie]$ instructions produce an observation $read$ `a` i, where the index i is the value of the expression ie
- $a[ie] \leftarrow e$ instructions produce an observation $write$ `a` i, where i is the value of the expression ie.

1.3 Security Labelings

Throughout the report, we will make use of security labelings, which assign *security levels* to variables and arrays. We only distinguish two security levels, *public* ($\mathbb{T}$) and *secret* ($\mathbb{F}$). Information may flow from public to private locations, but not the other way round, so this forms a two-point lattice with $\mathbb{T} \sqsubseteq \mathbb{F}$ and $\mathbb{T} \sqcup \mathbb{F} = \mathbb{F}$.

We typically write P for variable labelings and PA for array labelings. We further lift variable labelings P to a pair of functions $P(ae)$ and $P(be)$, which compute the security levels of arithmetic and boolean expressions respectively in the usual way. Similar to scalar states, we write updates to labelings as $[\texttt{X} \mapsto l]P$, resp. $[\texttt{a} \mapsto l]PA$.

2 Background

2.1 Speculative Execution Vulnerabilities

Speculative execution vulnerabilities, collectively known as Spectre, are vulnerabilities which arise due to a hardware optimization known as *speculative execution*. This common feature in modern pipelined (and even out-of-order) processors allows instructions to be executed on *predicted* data, thus avoiding pipeline stalls due to e.g. slow memory operations. While speculative execution does not affect the correctness of the result, as all speculative results are *rolled back* as soon as a misprediction is detected, this rollback does not apply to all microarchitectural state, e.g. caches. As a result, traditional cache-timing attacks can be used to obtain information about this speculative execution.

In this report, we focus only on Spectre-PHT[4], in which the outcome of branch instructions is predicted before the condition is completely evaluated. This enables e.g. the classic bounds-check-bypass [23]:

$$\texttt{if } i < \texttt{a\_size then j} \leftarrow \texttt{a}[i]; \texttt{x} \leftarrow \texttt{b}[j] \texttt{ else skip}$$

Here, an untrusted input `i` is used to index a (public) array a. The resulting value `j` is then used in a leakage-producing operation $\texttt{x} \leftarrow \texttt{b}[j]$. Although a bounds check protects this snippet, an attacker may be able to train the branch predictor to take the `then` branch, and then execute this code with an out-of-bounds index i pointing at secret data, which would therefore be leaked during speculative execution.

2.2 Formal Models of Speculative Execution

Many formal models of speculative execution have been proposed in the literature, varying along several aspects such as which sources of speculative execution can be modeled, how closely they model the execution of real processors (for example, Blade [24] models out-of-order execution, whereas other tools stick to more linear models of execution), and how they handle the inherent nondeterminism of the sources of speculative execution. For the latter, techniques include modeling the branch predictor as an oracle, forcing it to always mispredict [5], or allowing the attacker to control it explicitly via directives [10], [25]. For a more extensive review of different formalization approaches, see also the systematization by S. Cauligi *et al.* [26].

In this report, we will only investigate three models: The first, which is used in Section 3, is a *forward-only*, *directive-based* model, which means that the attacker has direct control over the branch predictor via directives given as input to the semantics, but we also do not model rollbacks: Once the attacker initiates speculative execution, it can not go back to nonspeculative execution. In Section 4, we will show that this is equivalent to a model *with* rollbacks, before also considering an *always-mispredict* model (inspired by M. Guarnieri *et al.* [5]).

2.3 Notions of Security Against Speculative Execution Attacks

A common notion of security, especially for cryptographic code, is *cryptographic constant-time security*, which requires that two executions must produce the same observations, as long as the initial states agree on all *public* variables and arrays (as specified by the labelings P and PA):

Definition 2.3.1 (CCT security).

$$\rho_1 \sim_P \rho_2 \land \mu_1 \sim_{PA} \mu_2 \land \langle c, \rho_1, \mu_1 \rangle \xrightarrow{\mathcal{O}_1} \langle \texttt{skip}, \cdot, \cdot \rangle$$

$$\land \langle c, \rho_2, \mu_2 \rangle \xrightarrow{\mathcal{O}_2} \langle \texttt{skip}, \cdot, \cdot \rangle \Rightarrow \mathcal{O}_1 = \mathcal{O}_2$$

Here, $\rho_1 \sim_P \rho_2$ denotes that ρ_1 and ρ_2 agree on the variables that are public according to P. Note also that we require both traces to fully execute, reaching a terminal state, as we obviously cannot require equal observations if we are comparing partial executions of different lengths[5].

[4] The name is due to the *pattern history table* used for branch prediction, following the naming convenvtion by C. Canella *et al.* [2]. This variant is also known as Spectre-v1.

[5] This also means that this definition makes no statement about executions which do not terminate, which is often not desirable. However, since we will not be using this definition in the remainder of the report, we prioritize ease of presentation.

Therefore, a straightforward way to define security in the speculative setting is to require observational equivalence in the same way, but for speculative execution:

Definition 2.3.2 (Speculative observational equivalence).

$$\langle c_1,\, \rho_1,\, \mu_1,\, b_1\rangle \approx_s \langle c_2,\, \rho_2,\, \mu_2,\, b_2\rangle \doteq \forall \mathcal{D}, \mathcal{O}_1, \mathcal{O}_2. \langle c_1,\, \rho_1,\, \mu_1,\, b_1\rangle \xrightarrow[\mathcal{D}]{\mathcal{O}_1}{}_{\mathrm{S}}^{*} \cdot \wedge \langle c_2,\, \rho_2,\, \mu_2,\, b_2\rangle \xrightarrow[\mathcal{D}]{\mathcal{O}_2}{}_{\mathrm{S}}^{*} \Rightarrow \mathcal{O}_1 = \mathcal{O}_2$$

Definition 2.3.3 (SCT (speculative constant-time) security).

$$\rho_1 \sim_P \rho_2 \wedge \mu_1 \sim_{PA} \mu_2 \Rightarrow \langle c,\, \rho_1,\, \mu_1,\, \mathbb{F}\rangle \approx_s \langle c,\, \rho_2,\, \mu_2,\, \mathbb{F}\rangle$$

Here, $\mathcal{D}$ denotes the directives that control misspeculation, and b is a flag indicating whether the current execution is misspeculating. It is set to $\mathbb{F}$ in the definition of SCT security, as there is no misspeculation prior to the start of a full program execution. We will introduce this semantics for speculative execution in more detail in Section 3, Fig. 7.

However, SCT security is not suitable for many applications, as it implies CCT security. While this is common for cryptographic code, it would be an unreasonable restriction in other contexts, e.g. operating system kernels or browsers. Thus, we need a notion of security which captures only the leakage *introduced by* speculative execution: a Spectre mitigation can not (realistically) eliminate nonspeculative leaks, but we still need to reason about the absence of speculative leaks in the presence of sequential leaks. While there are a variety of approaches to this, the one we will focus on in this report is *relative security*, which is especially well-suited to compiler mitigations. It states that if two initial states lead to attacker-indistinguishable *sequential* executions of the *source* program, then the *speculative* executions of the *mitigated* program will also be indistinguishable:

Definition 2.3.4 (Relative security).

$$\langle c,\, \rho_1,\, \mu_1\rangle \approx \langle c,\, \rho_2,\, \mu_2\rangle \Rightarrow \langle (\!|c|\!),\, \rho_1,\, \mu_1,\, \mathbb{F}\rangle \approx_s \langle (\!|c|\!),\, \rho_2,\, \mu_2,\, \mathbb{F}\rangle$$

Here, $(\!|\cdot|\!)$ denotes the transformation applied to the program. $\approx$ denotes sequential observational equivalence:

Definition 2.3.5 (Sequential observational equivalence).

$$\langle c_1,\, \rho_1,\, \mu_1\rangle \approx \langle c_2,\, \rho_2,\, \mu_2\rangle \doteq \forall \mathcal{O}_1, \mathcal{O}_2. \langle c_1,\, \rho_1,\, \mu_1\rangle \xrightarrow{\mathcal{O}_1}{}^{*} \cdot \wedge \langle c_2,\, \rho_2,\, \mu_2\rangle \xrightarrow{\mathcal{O}_2}{}^{*} \cdot \Rightarrow \mathcal{O}_1 \gtrless \mathcal{O}_2$$

In the conclusion, $\gtrless$ denotes that one of the sequences of observations must be a prefix of the other, which makes this notion also applicable to partial executions.

2.4 Software-based Mitigations

The first mitigation proposed for Spectre-PHT was to effectively disable speculation using serializing "fence" instructions such as the x86 `lfence` instruction, which enforces that all prior instructions have completed before later instructions begin executing [1], [3]. While fence instructions after every branch clearly prevent Spectre-PHT attacks, this comes with a heavy performance penalty [4], making it undesirable in practice.

Several recent works [7], [8] therefore focus on a different approach, which does not prevent speculative execution:

2.4.1 Speculative Load Hardening

Speculative load hardening, or SLH for short, is a mitigation against Spectre-PHT introduced by C. Carruth [6] for the LLVM compiler. Instead of relying on instructions which block speculative execution, it detects misspeculated execution, and conditionally protects secrets in that case.

To detect misspeculation, SLH maintains a *misspeculation flag* which tracks whether all branches along the current execution path have been taken correctly. It achieves this by using *branchless conditional instructions*, which can update the flag without being a source of speculative execution themselves. Such instructions are available in most architectures, e.g. conditional moves in x86, but can otherwise also be implemented using boolean logic.

Leaks during speculative execution are prevented by masking the values or addresses of load and store instructions using the misspeculation flag. For clarity, we also represent this masking using branchless conditionals, in practice, boolean operations are commonly used instead.

There are two variants of this masking, commonly referred to as *address hardening*[6] or *value hardening*. In *address hardening*, the masking is always applied to the *address* of load and store instructions, making sure that they will always access some safe memory location. With *value hardening*, on the other hand, the masking can under some circumstances be applied to the *loaded value* of a read instruction instead of the address. It is important to note, however, that the versions of SLH implemented in LLVM provide neither SCT nor relative security, since leakage via branch conditions is not protected.

2.4.1.1 SLH Master Recipe

In order to more easily compare different versions of SLH, we can describe them as instantiations of a common "SLH master recipe" [11], where the actual masking operations of branch conditions, read and write indices are parameterized by $[\![\cdot]\!]_{\mathbb{B}}$, $[\![\cdot]\!]^{\cdot}_{rd}$) and $[\![\cdot]\!]^{\cdot}_{wr}$ respectively. For index SLH, this looks as follows:

$$
\begin{aligned}
(\!|\texttt{skip}|\!) &\doteq \texttt{skip} \\
(\!|\texttt{x} := e|\!) &\doteq \texttt{x} := e \\
(\!|c_1; c_2|\!) &\doteq (\!|c_1|\!); (\!|c_2|\!) \\
(\!|\texttt{if } be \texttt{ then } c_1 \texttt{ else } c_2|\!) &\doteq \texttt{if } [\![be]\!]_{\mathbb{B}} \texttt{ then b} := [\![be]\!]_{\mathbb{B}} \texttt{ ? b : 1; } (\!|c_1|\!) \\
&\qquad\qquad \texttt{else b} := [\![be]\!]_{\mathbb{B}} \texttt{ ? 1 : b; } (\!|c_2|\!) \\
(\!|\texttt{while } be \texttt{ do } c|\!) &\doteq \texttt{while } [\![be]\!]_{\mathbb{B}} \texttt{ do} \\
&\qquad\qquad \texttt{b} := [\![be]\!]_{\mathbb{B}} \texttt{ ? b : 1; } (\!|c|\!); \\
&\qquad\qquad \texttt{b} := [\![be]\!]_{\mathbb{B}} \texttt{ ? 1 : b} \\
(\!|\texttt{X} \leftarrow \texttt{a}[i]|\!) &\doteq \texttt{X} \leftarrow \texttt{a}[[\![i]\!]^{\texttt{X}}_{rd}] \\
(\!|\texttt{a}[i] \leftarrow e|\!) &\doteq \texttt{a}[[\![i]\!]^{e}_{wr}] \leftarrow e
\end{aligned}
$$

Fig. 2. Index SLH (iSLH) master recipe

For value SLH, we only need to adjust the read case. The conditions under which the loaded value can be masked instead of the address are parameterized by $VC(\cdot, \cdot)$:

$$
(\!|\texttt{X} \leftarrow \texttt{a}[i]|\!) \doteq \begin{cases} \texttt{X} \leftarrow \texttt{a}[i]; \texttt{X} := b == 1 \texttt{ ? 0 : X} & if\ VC(\texttt{X}, i) \\ \texttt{X} \leftarrow \texttt{a}[[\![i]\!]^{\texttt{X}}_{rd}] & otherwise \end{cases}
$$

Fig. 3. Value SLH (vSLH) master recipe

2.4.2 Selective SLH

Selective SLH [7] is a variant of SLH which drastically increases the performance of mitigated code by selectively masking values only when they are loaded into *public variables*. It enforces SCT, but only limited to a specific class of programs, which are well-typed in a CT type system. For AWhile, we would use the following type system:

$$
\text{CT\_Skip}\ \frac{}{P; PA \vdash \texttt{skip}} \qquad \text{CT\_Asgn}\ \frac{P(e) \sqsubseteq P(\texttt{X})}{P; PA \vdash \texttt{X} := e} \qquad \text{CT\_Seq}\ \frac{P; PA \vdash c_1 \quad P; PA \vdash c_2}{P; PA \vdash c_1; s_2}
$$

$$
\text{CT\_If}\ \frac{P(be) \quad P; PA \vdash c_1 \quad P; PA \vdash c_2}{P; PA \vdash \texttt{if } be \texttt{ then } c_1 \texttt{ else } s_2} \qquad \text{CT\_While}\ \frac{P(be) \quad P; PA \vdash c}{P; PA \vdash \texttt{while } be \texttt{ do } c}
$$

$$
\text{CT\_ARead}\ \frac{P(i) \quad PA(\texttt{a}) \sqsubseteq P(\texttt{X})}{P; PA \vdash \texttt{X} \leftarrow \texttt{a}[i]} \qquad \text{CT\_AWrite}\ \frac{P(i) \quad P(e) \sqsubseteq PA(\texttt{a})}{P; PA \vdash \texttt{a}[i] \leftarrow e}
$$

Fig. 4. CT type system for Selective SLH

[6] Since our relatively high-level AWhile language operates directly on arrays and indices instead of computing addresses, we can not fully capture all subtleties of *address SLH* with this language. Therefore, we will instead use the term *index SLH* to emphasize that we only apply masking to the index, not the array itself.

We can now instantiate the SLH master recipe for Selective SLH, where the protection of the index will depend on the security level of the target variable:

$$\llbracket be \rrbracket_{\mathbb{B}} \doteq be$$
$$\llbracket i \rrbracket^{\mathtt{X}}_{rd} \doteq \begin{cases} \mathtt{b} == 1 \,?\, 0 : i & if\ P(\mathtt{X}) \\ i & otherwise \end{cases}$$
$$\llbracket i \rrbracket^{e}_{wr} \doteq \begin{cases} \mathtt{b} == 1 \,?\, 0 : i & if\ \neg P(e) \\ i & otherwise \end{cases}$$

Fig. 5a. SiSLH

$$VC(\mathtt{X}, i) \doteq P(\mathtt{X})$$
$$\llbracket be \rrbracket_{\mathbb{B}} \doteq be$$
$$\llbracket i \rrbracket^{\mathtt{X}}_{rd} \doteq i$$
$$\llbracket i \rrbracket^{e}_{wr} \doteq i$$

Fig. 5b. SvSLH

Fig. 5. SLH recipe instantiations for Selective SLH

Note that protecting write addresses in Fig. 5a is necessary, as illustrated by a counterexample presented by J. Baumann *et al.* [11].

2.4.3 Ultimate SLH

Ultimate SLH [8] is an SLH variant offering the most exhaustive protections currently available to our knowledge. Although a major aspect of Ultimate SLH is protection against leakage via variable-time instructions, we will not model such behaviors for this report. Instead, we focus on another important aspect of Ultimate SLH: The fact that it also protects *branch conditions* in order to prevent leakage via control flow[7].

For Ultimate SLH, we thus instantiate the iSLH master recipe as follows:

$$\llbracket be \rrbracket_{\mathbb{B}} \doteq \mathtt{b} == 0 \,\&\&\, be$$
$$\llbracket i \rrbracket^{\mathtt{X}}_{rd} \doteq \mathtt{b} == 1 \,?\, 0 : i$$
$$\llbracket i \rrbracket^{e}_{wr} \doteq \mathtt{b} == 1 \,?\, 0 : i$$

Fig. 6. iSLH recipe instantiation for USLH

Note that there is no vSLH version, as Ultimate SLH always protects the memory address.

Ultimate SLH enforces relative security and puts no restrictions on the mitigated program. However, this security comes at a high cost, with a performance overhead of roughly 150%[8] [8].

3 Extending FlexSLH to Arbitrary Programs

3.1 Speculative Execution Model

For this section, we will use a *forward-only, directive-based* model of speculative execution, which means that

- we do not model rollbacks; once speculative execution is initiated, it can not go back to nonspeculative execution
- the attacker exerts direct control over the branch predictor with directives that are supplied to the semantics.

The former helps to greatly simplify proofs, as the semantics does not need to keep track of previous states for rollbacks. We will show in Section 4 that this captures the same leakage that would be captured by a model with rollbacks. The latter overapproximates attacker capabilities regarding training the branch predictor, but also concerning memory layout. Importantly, it enables reasoning with a high-level language, without having to consider details regarding e.g. the memory layout.

We show here only the rules for branches, reads and writes. The full set of rules is given in Appendix C.

[7] This aspect can also be found in prior work by M. Patrignani and M. Guarnieri [27], although they do not mention the necessity of this, nor that it differs from other variants.

[8] Even with this high overhead, it still outperforms fences, which have a performance overhead of roughly 300% on the same benchmark [8]

$$\textsc{Spec\_If}\ \frac{b' = [\![be]\!]_\rho}{\langle \texttt{if}\ be\ \texttt{then}\ c_\mathbb{T}\ \texttt{else}\ c_\mathbb{F},\ \rho,\ \mu,\ b\rangle \xrightarrow[step]{branch\ b'}_\mathrm{S} \langle c_{b'},\ \rho,\ \mu,\ b\rangle}$$

$$\textsc{Spec\_If\_Force}\ \frac{b' = [\![be]\!]_\rho}{\langle \texttt{if}\ be\ \texttt{then}\ c_\mathbb{T}\ \texttt{else}\ c_\mathbb{F},\ \rho,\ \mu,\ b\rangle \xrightarrow[force]{branch\ b'}_\mathrm{S} \langle c_{\neg b'},\ \rho,\ \mu,\ \mathbb{T}\rangle}$$

$$\textsc{Spec\_Read}\ \frac{i = [\![ie]\!]_\rho \qquad v = [\![\mathtt{a}[i]]\!]_\mu \qquad i < |\ \mathtt{a}\ |_\mu}{\langle \mathtt{X} \leftarrow \mathtt{a}[ie],\ \rho,\ \mu,\ b\rangle \xrightarrow[step]{read\ \mathtt{a}\ i}_\mathrm{S} \langle \texttt{skip},\ [\mathtt{X} \mapsto v]\rho,\ \mu,\ b\rangle}$$

$$\textsc{Spec\_Read\_Force}\ \frac{i = [\![ie]\!]_\rho \qquad v = [\![\mathtt{b}[j]]\!]_\mu \qquad i \geq |\ \mathtt{a}\ |_\mu \qquad j < |\ \mathtt{b}\ |_\mu}{\langle \mathtt{X} \leftarrow \mathtt{a}[ie],\ \rho,\ \mu,\ \mathbb{T}\rangle \xrightarrow[load\ \mathtt{b}\ j]{read\ \mathtt{a}\ i}_\mathrm{S} \langle \texttt{skip},\ [\mathtt{X} \mapsto v]\rho,\ \mu,\ \mathbb{T}\rangle}$$

$$\textsc{Spec\_Write}\ \frac{i = [\![ie]\!]_\rho \qquad v = [\![ae]\!]_\rho \qquad i < |\ \mathtt{a}\ |_\mu}{\langle \mathtt{a}[ie] \leftarrow ae,\ \rho,\ \mu,\ b\rangle \xrightarrow[step]{write\ \mathtt{a}\ i}_\mathrm{S} \langle \texttt{skip},\ \rho,\ [\mathtt{a}[i] \mapsto v]\mu,\ b\rangle}$$

$$\textsc{Spec\_Write\_Force}\ \frac{i = [\![ie]\!]_\rho \qquad v = [\![ae]\!]_\rho \qquad i \geq |\ \mathtt{a}\ |_\mu \qquad j < |\ \mathtt{b}\ |_\mu}{\langle \mathtt{a}[ie] \leftarrow ae,\ \rho,\ \mu,\ \mathbb{T}\rangle \xrightarrow[store\ \mathtt{b}\ j]{write\ \mathtt{a}\ i}_\mathrm{S} \langle \texttt{skip},\ \rho,\ [\mathtt{b}[j] \mapsto v]\mu,\ \mathbb{T}\rangle}$$

Fig. 7. (Forward-only, directive-based) speculative semantics of AWhile (selected rules)

3.2 Flexible SLH

Flexible SLH [11] is a Spectre mitigation which generalizes both Selective and Ultimate SLH. It combines the ideas of selectively applying masking based on a security labeling with the additional hardening of branch conditions.

Like Selective SLH, the simplest presentation of FSLH relies on an information-flow (IFC)[9] type system, although it uses a more general one similar to D. M. Volpano *et al.* [28]:

$$\textsc{WT\_Skip}\ \frac{}{P; PA \vdash_{pc} \texttt{skip}} \qquad \textsc{WT\_Asgn}\ \frac{P(e) = l \qquad pc \sqcup l \sqsubseteq P(\mathtt{X})}{P; PA \vdash_{pc} \mathtt{X} := e} \qquad \textsc{WT\_Seq}\ \frac{P; PA \vdash_{pc} c_1 \qquad P; PA \vdash_{pc} c_2}{P; PA \vdash_{pc} c_1; s_2}$$

$$\textsc{WT\_If}\ \frac{P(be) = l \qquad P; PA \vdash_{pc \sqcup l} c_1 \qquad P; PA \vdash_{pc \sqcup l} c_2}{P; PA \vdash_{pc} \texttt{if}\ be\ \texttt{then}\ c_1\ \texttt{else}\ s_2} \qquad \textsc{WT\_While}\ \frac{P(be) = l \qquad P; PA \vdash_{pc \sqcup l} c}{P; PA \vdash_{pc} \texttt{while}\ be\ \texttt{do}\ c}$$

$$\textsc{WT\_ARead}\ \frac{P(i) = l_i \qquad pc \sqcup l_i \sqcup PA(\mathtt{a}) \sqsubseteq P(\mathtt{X})}{P; PA \vdash_{pc} \mathtt{X} \leftarrow \mathtt{a}[i]} \qquad \textsc{WT\_AWrite}\ \frac{P(i) = l_i \qquad pc \sqcup l_i \sqcup P(e) \sqsubseteq PA(\mathtt{a})}{P; PA \vdash_{pc} \mathtt{a}[i] \leftarrow e}$$

Fig. 8. IFC type system for FSLH (differences compared to Fig. 4 highlighted)

Using this type system, we can instantiate the SLH master recipe for FlexSLH:

$$[\![be]\!]_\mathbb{B} \doteq \begin{cases} \mathtt{b} == 0\ \&\&\ be & if\ \neg P(be) \\ be & otherwise \end{cases}$$

$$[\![i]\!]^{\mathtt{X}}_{rd} \doteq \begin{cases} \mathtt{b} == 1\ ?\ 0 : i & if\ P(\mathtt{X}) \vee \neg P(i) \\ i & otherwise \end{cases}$$

$$[\![i]\!]^{e}_{wr} \doteq \begin{cases} \mathtt{b} == 1\ ?\ 0 : i & if\ \neg P(e) \vee \neg P(i) \\ i & otherwise \end{cases}$$

Fig. 9a. FiSLH

$$VC(\mathtt{X}, i) \doteq P(\mathtt{X}) \wedge P(i)$$

$$[\![be]\!]_\mathbb{B} \doteq \begin{cases} \mathtt{b} == 0\ \&\&\ be & if\ \neg P(be) \\ be & otherwise \end{cases}$$

$$[\![i]\!]^{\mathtt{X}}_{rd} \doteq \begin{cases} \mathtt{b} == 1\ ?\ 0 : i & if\ \neg P(i) \\ i & otherwise \end{cases}$$

$$[\![i]\!]^{e}_{wr} \doteq \begin{cases} \mathtt{b} == 1\ ?\ 0 : i & if\ \neg P(i) \\ i & otherwise \end{cases}$$

Fig. 9b. FvSLH

Fig. 9. SLH recipe instantiations for flexible SLH (differences compared to Figure 5 highlighted)

[9] IFC stands for information-flow control. We will use this acronym, but typically omit "control" in writing.

This version of FSLH ensures relative safety [11, Sections 5 and 6], but only for programs that are well-typed in this type system. The proof is fully mechanized in Rocq.

3.3 FvSLH$^\forall$: Generalizing FSLH Using Static Information-Flow Analysis

This section is adapted from J. Baumann et al. *[11, Section 7].*

The main limitation of the version of FSLH presented above is that the presented type system rejects some programs, as it requires the labelings P and PA to remain constant throughout the program. To lift this restriction, we replace it with a flow-sensitive information-flow analysis similar to the algorithmic version of S. Hunt and D. Sands's [29, Section 4] flow-sensitive type system. In contrast to the type system in Fig. 8, this IFC analysis updates the labelings with each assignment, resulting in different labelings used to compute the IFC levels of individual expressions at different program points. We call this version FvSLH$^\forall$ to clearly distinguish it from the versions above.

We record the results of the IFC analysis as annotations to our commands, as they are used both to apply protections as well as in the security proof. We therefore extend the the syntax of AWhile as follows:

$$
\begin{aligned}
\overline{c} \in \mathit{acom} ::= {} & \texttt{skip} \mid \texttt{X} := e \mid \overline{c};_{@(P,PA)}\, \overline{c} \\
\mid {} & \texttt{if}\ b_{@l}\ \texttt{then}\ \overline{c}\ \texttt{else}\ \overline{c} \\
\mid {} & \texttt{while}\ b_{@l}\ \texttt{do}\ \overline{c}_{@(P,PA)} \\
\mid {} & \texttt{X}_{@l_{\texttt{x}}} \leftarrow \texttt{a}[e_{@l_i}] \\
\mid {} & \texttt{a}[e_{@l_i}] \leftarrow e \\
\mid {} & \mathit{branch}\ l\ \overline{c}
\end{aligned}
$$

Fig. 10. Syntax of annotated commands

- Branch conditions, array indices, and target variables of array reads are labelled with their statically determined security level. These annotations are directly used by FvSLH$^\forall$.
- Sequence and loop commands are annotated with intermediate labelings used to define well-labeledness (see Section 3.4.2).
- Finally, we introduce a new (annotated) command $\mathit{branch}\ l\ \overline{c}$, which wraps $\overline{c}$ with an additional label l. This annotation is not produced by the IFC analysis, but by an intermediate formalism in the proof, which we call *ideal semantics* (see Section 3.4.1), and which will use this annotation to help track implicit flows.

$$
\begin{aligned}
\langle\!\langle \texttt{skip} \rangle\!\rangle^{P,PA}_{pc} &\doteq (\texttt{skip}, P, PA) \\
\langle\!\langle \texttt{X} := e \rangle\!\rangle^{P,PA}_{pc} &\doteq (\texttt{X} := e, P, PA) \\
\langle\!\langle c_1; c_2 \rangle\!\rangle^{P,PA}_{pc} &\doteq \left(\overline{c_1};_{@(P_1,PA_1)}\, \overline{c_2}, P_2, PA_2\right) \ \mathit{where}\ (\overline{c_1}, P_1, PA_1) = \langle\!\langle c_1 \rangle\!\rangle^{P,PA}_{pc} \\
&\qquad\qquad \mathit{and}\ (\overline{c_2}, P_2, PA_2) = \langle\!\langle c_2 \rangle\!\rangle^{P_1,PA_1}_{pc} \\
\langle\!\langle \texttt{if}\ be\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \rangle\!\rangle^{P,PA}_{pc} &\doteq \left(\texttt{if}\ be_{@P(be)}\ \texttt{then}\ \overline{c_1}\ \texttt{else}\ \overline{c_2}, P_1 \sqcup P_2, PA_1 \sqcup PA_2\right) \\
&\qquad\qquad \mathit{where}\ (\overline{c_1}, P_1], PA_1) = \langle\!\langle c_1 \rangle\!\rangle^{P,PA}_{pc \sqcup P(be)} \\
&\qquad\qquad \mathit{and}\ (\overline{c_2}, P_2, PA_2) = \langle\!\langle c_2 \rangle\!\rangle^{P,PA}_{pc \sqcup P(be)} \\
\langle\!\langle \texttt{while}\ be\ \texttt{do}\ c \rangle\!\rangle^{P,PA}_{pc} &\doteq \left(\texttt{while}\ be_{@P_{fix}(be)}\ \texttt{do}\ \overline{c}_{@(P_{fix},PA_{fix})}\right)\ \mathit{where} \\
&\qquad\qquad (P_{fix}, PA_{fix}) = \boldsymbol{fix}\ \Big(\lambda(P', PA').\ \boldsymbol{let}\ (\overline{c}, P'', PA'') = \langle\!\langle c \rangle\!\rangle^{P',PA'}_{pc \sqcup P'(be)} \\
&\qquad\qquad\qquad\qquad \boldsymbol{in}\ (P'', PA'') \sqcup (P, PA)\Big) \\
\langle\!\langle \texttt{X} \leftarrow \texttt{a}[i] \rangle\!\rangle^{P,PA}_{pc} &\doteq \left(\texttt{X}_{@pc \sqcup P(i) \sqcup PA(\texttt{a})} \leftarrow \texttt{a}[i_{@P(i)}], [\texttt{X} \mapsto pc \sqcup P(i) \sqcup PA(a)]P, PA\right) \\
\langle\!\langle \texttt{a}[i] \leftarrow e \rangle\!\rangle^{P,PA}_{pc} &\doteq \left(\texttt{a}[i_{@P(i)}] \leftarrow e, P, [\texttt{a} \mapsto PA(\texttt{a}) \sqcup pc \sqcup P(i) \sqcup P(e)]PA\right)
\end{aligned}
$$

Fig. 11. Flow-sensitive IFC analysis generating annotated commands

Our IFC analysis is a straightforward extension of the one of S. Hunt and D. Sands [29] to AWhile while also adding annotations to the commands, and is shown in Fig. 11. We write $(P_1, PA_1) \sqsubseteq (P_2, PA_2)$, $P_1 \sqcup P_2$, and $PA_1 \sqcup PA_2$ for the pointwise liftings from labels to maps. For assignments, loads, and stores, the analysis simply updates the labelings and annotates expressions with the appropriate labels. It also takes into account the program counter label *pc*, which helps prevent implicit flows. For conditionals, both branches are analyzed recursively, raising the *pc* label by the label of the condition. We then take the join of the two resulting labelings, which is a necessary overapproximation, as we cannot determine statically which branch will be taken.

Similarly, for loops, we must account for arbitrarily many iterations. This is achieved by a fixpoint labeling, i.e., a labeling such that analysis of the loop body with this labeling results in an equal or more precise final labeling, which implies that the annotations produced by the analysis remain correct for execution in this final labeling (where correctness of annotations is formalized in Section 3.4.2). Further, this fixpoint labeling must also be less precise than the initial labeling to ensure that the computed annotations are correct for the first iteration. We compute the fixpoint labeling with a simple fixpoint iteration, ensuring termination by additionally providing the number of variables and arrays assigned by c as an upper bound. This is sound, as each iteration that does not yield a fixpoint must change some array or variable from public to secret.

Once the information-flow analysis has produced an annotated command, these annotations can be used to apply the protections. We denote the corresponding translation function by $(\!|\cdot|\!)^{FvSLH^\forall}$. Its behavior is similar to the FvSLH instantiation shown in Fig. 9b, but using the provided labels instead of computing security levels of expressions:

$$
\begin{aligned}
(\!|\texttt{skip}|\!)^{FvSLH^\forall} &\doteq \texttt{skip} \\
(\!|\texttt{x} := \texttt{ae}|\!)^{FvSLH^\forall} &\doteq \texttt{x} := \texttt{ae} \\
(\!|\overline{c_1};_{@(P,PA)}\ \overline{c_2}|\!)^{FvSLH^\forall} &\doteq (\!|\overline{c_1}|\!)^{FvSLH^\forall}; (\!|\overline{c_2}|\!)^{FvSLH^\forall} \\
(\!|\texttt{if}\ be_{@l}\ \texttt{then}\ \overline{c_\mathbb{T}}\ \texttt{else}\ \overline{c_\mathbb{F}}|\!)^{FvSLH^\forall} &\doteq \texttt{if}\ [\![be]\!]^l_\mathbb{B} \\
&\qquad \texttt{then b} := [\![be]\!]^l_\mathbb{B}\ \texttt{?b}:1; (\!|c_\mathbb{T}|\!)^{FvSLH^\forall} \\
&\qquad \texttt{else b} := [\![be]\!]^l_\mathbb{B}\ \texttt{?}1:\texttt{b}; (\!|c_\mathbb{F}|\!)^{FvSLH^\forall} \\
(\!|\texttt{while}\ be_{@l}\ \texttt{do}\ \overline{c}_{@(P,PA)}|\!)^{FvSLH^\forall} &\doteq \texttt{while}\ [\![be]\!]^l_\mathbb{B} \\
&\qquad \texttt{b} := [\![be]\!]^l_\mathbb{B}\ \texttt{?b}:1; (\!|\overline{c}|\!)^{FvSLH^\forall} \\
&\qquad \texttt{b} := [\![be]\!]^l_\mathbb{B}\ \texttt{?}1:\texttt{b} \\
(\!|\texttt{X}_{@l_\texttt{x}} \leftarrow \texttt{a}[i_{@l_i}]|\!)^{FvSLH^\forall} &\doteq \begin{cases} \texttt{X} \leftarrow \texttt{a}[i]; \texttt{X} := \texttt{b} == 1\ \texttt{?}0:\texttt{X} & if\ l_\texttt{x} \wedge l_i \\ \texttt{X} \leftarrow \texttt{a}\left[[\![i]\!]^{l_\texttt{x},l_i}_{rd}\right] & otherwise \end{cases} \\
(\!|\texttt{a}[i_{@l_i}] \leftarrow e|\!)^{FvSLH^\forall} &\doteq \texttt{a}\left[[\![i]\!]^{l_i}_{wr}\right] \leftarrow e
\end{aligned}
$$

$$
\begin{aligned}
[\![be]\!]^\mathbb{T}_\mathbb{B} &\doteq be \\
[\![be]\!]^\mathbb{F}_\mathbb{B} &\doteq \texttt{b} = 0 \wedge be \\
[\![i]\!]^{\mathbb{F},\mathbb{T}}_{rd} &\doteq i \\
[\![i]\!]^{\cdot,\cdot}_{rd} &\doteq \texttt{b} = 1\ \texttt{?}\ 0 : i \quad otherwise \\
[\![i]\!]^\mathbb{T}_{wr} &\doteq i \\
[\![i]\!]^\mathbb{F}_{wr} &\doteq \texttt{b} = 1\ \texttt{?}\ 0 : i
\end{aligned}
$$

Fig. 12. Translation function applying FvSLH$^\forall$ protections

3.4 Proof of Relative Security

The proof of relative security for FvSLH$^\forall$ relies on the following core ideas:

- An *ideal semantics*, which captures the same protections as the FvSLH$^\forall$ transformation, but as restrictions within the semantics. This enables to prove separately the correctness of the transformation with respect to the ideal semantics, and relative security of the ideal semantics.
- A *well-labeledness predicate*, similar to a typing judgment, which describes whether annotations are permissible (although not necessarily precise) with respect to the information flow. This is required in order to prove that the labels produced by the information-flow analysis remain valid during execution.
- The proof of relative security for the ideal semantics, which decomposes executions into three parts: a non-speculative part which behaves exactly the same as the sequential semantics, a step which initiates speculation, and a speculative part, during which the observations are fully determined by the directives and public values.

3.4.1 Ideal Semantics

We simplify the proof of relative security by using an *ideal semantics*, which applies the same protections as FvSLH$^{\forall}$, but as restrictions in the semantics instead of as a code transformation. As the FvSLH$^{\forall}$ code transformation depends on the annotations produced by the information-flow analysis, the ideal semantics operates on annotated commands. Further, we also instrument this ideal semantics to dynamically track the labels of variables and arrays, which allows us to prove that ideal executions preserve agreement on *public*-labelled variables. The ideal semantics (Fig. 13) therefore differs from the speculative semantics in Fig. 7 in the following ways:

- It operates on annotated commands.
- States additionally include a pc label, a variable labeling P, and an array labeling PA, which are updated with every step.
- Masking of indices, values and branch conditions based on the annotated labels is performed within the semantics.

$$\textsc{Ideal\_If}\quad \frac{b' = (l \vee \neg b) \wedge [\![be]\!]_\rho}{\langle \texttt{if}\ be_{@l}\ \texttt{then}\ \overline{c_\mathbb{T}}\ \texttt{else}\ \overline{c_\mathbb{F}},\ \rho,\ \mu,\ b,\ pc,\ P,\ PA\rangle \xrightarrow[step]{branch\ b'}_{\mathrm{i}} \langle branch\ pc\ \ \overline{c_{b'}},\ \rho,\ \mu,\ b,\ pc \sqcup l,\ P,\ PA\rangle}$$

$$\textsc{Ideal\_If\_Force}\quad \frac{b' = (l \vee \neg b) \wedge [\![be]\!]_\rho}{\langle \texttt{if}\ be_{@l}\ \texttt{then}\ \overline{c_\mathbb{T}}\ \texttt{else}\ \overline{c_\mathbb{F}},\ \rho,\ \mu,\ b,\ pc,\ P,\ PA\rangle \xrightarrow[force]{branch\ b'}_{\mathrm{i}} \langle branch\ pc\ \ \overline{c_{\neg b'}},\ \rho,\ \mu,\ \mathbb{T},\ pc \sqcup l,\ P,\ PA\rangle}$$

$$\textsc{Ideal\_Read}\quad \frac{i = \begin{cases} 0 & if\ \neg(l_i) \wedge b \\ [\![ie]\!]_\rho & otherwise \end{cases} \qquad v = \begin{cases} 0 & if\ l_\mathtt{X} \wedge l_i \wedge b \\ [\![\mathtt{a}[i]]\!]_\mu & otherwise \end{cases} \qquad i < |\ \mathtt{a}\ |_\mu}{\langle \mathtt{X}_{@l_\mathtt{X}} \leftarrow \mathtt{a}[ie_{@l_i}],\ \rho,\ \mu,\ b,\ pc,\ P,\ PA\rangle \xrightarrow[step]{read\ \mathtt{a}\ i}_{\mathrm{i}} \langle \mathtt{skip},\ [\mathtt{X} \mapsto v]\rho,\ \mu,\ b,\ pc,\ [\mathtt{X} \mapsto l_\mathtt{X}]P,\ PA\rangle}$$

$$\textsc{Ideal\_Read\_Force}\quad \frac{i = [\![ie]\!]_\rho \qquad v = \begin{cases} 0 & if\ l_\mathtt{X} \\ [\![\mathtt{b}[j]]\!]_\mu & otherwise \end{cases} \qquad i \geq |\ \mathtt{a}\ |_\mu \qquad j < |\ \mathtt{b}\ |_\mu}{\langle \mathtt{X}_{@l_\mathtt{X}} \leftarrow \mathtt{a}[ie_{@\mathbb{T}}],\ \rho,\ \mu,\ \mathbb{T},\ pc,\ P,\ PA\rangle \xrightarrow[load\ \mathtt{b}\ j]{read\ \mathtt{a}\ i}_{\mathrm{i}} \langle \mathtt{skip},\ [\mathtt{X} \mapsto v]\rho,\ \mu,\ \mathbb{T},\ pc,\ [\mathtt{X} \mapsto l_\mathtt{X}]P,\ PA\rangle}$$

$$\textsc{Ideal\_Seq\_Skip}\quad \frac{terminal\ \overline{c_1}}{\langle \overline{c_1};_{@(P',PA')}\ \overline{c_2},\ ...,\ pc,\ ...\rangle \xrightarrow[\bullet]{}_{\mathrm{i}} \langle \overline{c_2},\ ...,\ pc\text{-}after\ \overline{c_1}\ \ pc,\ ...\rangle} \qquad \textsc{Ideal\_Branch}\quad \frac{\langle \overline{c},\ ...\rangle \xrightarrow[d]{o}_{\mathrm{i}} \langle \overline{c'},\ ...\rangle}{\langle branch\ l\ \overline{c},\ ...\rangle \xrightarrow[d]{o}_{\mathrm{i}} \langle branch\ l\ \overline{c'},\ ...\rangle}$$

Fig. 13. Ideal semantics for FvSLH$^{\forall}$ (selected rules, see Appendix D)

In order to properly track implicit flows during execution, the pc label needs to be saved and restored correctly, which we achieve using *branch* annotations: When entering a branch $\overline{c}$ in rules Ideal_If and Ideal_If_Force in Fig. 13, $\overline{c}$ is wrapped in an annotated command $branch\ l\ \overline{c}$, where pc is the program counter label before entering the branch. The command $\overline{c}$ underneath this annotation then takes steps as usual (Ideal_Branch). This leads to terminated commands (`skip`) potentially being wrapped in an arbitrary number of such branch annotations, so we also introduce the predicate *terminal* to describe such commands. The rule Ideal_Seq_Skip is adjusted accordingly, allowing any terminal program on the left. In this rule, the function $pc\text{-}after\ \overline{c}\ \ pc$ determines the label of the outermost branch annotation and uses it to restore the program counter label to the one before entering the branch. If there is no branch annotation, $pc\text{-}after\ \overline{c}\ \ pc$ defaults to the current program counter label pc. We choose to use branch annotations in this way as it allows us to keep a one-to-one correspondence between steps of the sequential and ideal semantics.

Since the ideal semantics matches the masking behavior of $(\!|\cdot|\!)^{FvSLH^{\forall}}$, the following compiler correctness result holds for the transformation:

Lemma 3.4.1 (Backwards compiler correctness for FvSLH$^{\forall}$).

$$\begin{aligned} &\big(\forall \mathtt{a}.\ |\ \mathtt{a}\ |_\mu > 0\big) \wedge \mathtt{b} \notin VARS(c) \wedge \rho(\mathtt{b}) = [\![b]\!]_\mathbb{N} \\ &\Rightarrow \langle (\!|\overline{c}|\!)^{FvSLH^{\forall}},\ \rho,\ \mu,\ b\rangle \xrightarrow[\mathcal{D}]{\mathcal{O}}_{\mathrm{S}}{}^{*} \langle c',\ \rho',\ \mu',\ b'\rangle \\ &\Rightarrow \exists \overline{c''}\, P'\, PA'\, pc'.\langle c,\ \rho,\ \mu,\ b,\ pc,\ P,\ PA\rangle \xrightarrow[\mathcal{D}]{\mathcal{O}}_{\mathrm{i}}{}^{*} \langle \mathtt{skip},\ [\mathtt{b} \mapsto \rho(\mathtt{b})]\rho',\ \mu',\ pc',\ p',\ PA'\rangle \\ &\wedge \big(c' = \mathtt{skip} \Rightarrow terminal\ \overline{c''} \wedge \rho'(\mathtt{b}) = [\![b']\!]_\mathbb{N}\big) \end{aligned}$$

3.4.2 Well-labeledness

Since the ideal semantics relies on the labels included in our annotated commands, we can only show relative security if the annotations have been computed correctly, i.e. a secret value can never be labeled as public. While this is the case for the initial annotated program obtained by our IFC analysis, we need a property that is preserved during execution, and thus cannot reference the analysis directly. We therefore define a well-labeledness judgment $P_1, PA_1 \rightsquigarrow P_2, PA_2 \vdash_{pc} \overline{c}$ of an annotated command $\overline{c}$ with respect to an initial labeling (P_1, PA_1) and pc label and a final labeling (P_2, PA_2) with the derivation rules given in Fig. 14. The rules largely match the behavior of $\langle\!\langle \cdot \rangle\!\rangle_P^{PA,pc}$, except instead of saving computed labels, we only make sure that they don't contradict the already annotated ones.

$$\textsc{WL\_Skip}\ \frac{(P_1, PA_1) \sqsubseteq (P_2, PA_2)}{P_1, PA_1 \rightsquigarrow P_2, PA_2 \vdash_{pc} \texttt{skip}} \qquad \textsc{WL\_Asgn}\ \frac{([\texttt{X} \mapsto P_1(e)]P_1, PA_1) \sqsubseteq (P_2, PA_2)}{P_1, PA_1 \rightsquigarrow P_2, PA_2 \vdash_{pc} \texttt{X} := e}$$

$$\textsc{WL\_Seq}\ \frac{\mathit{branch\text{-}free}\ \overline{c_2} \quad P_1, PA_1 \rightsquigarrow P', PA' \vdash_{pc} \overline{c_1} \quad P', PA' \rightsquigarrow P_2, PA_2 \vdash_{(\mathit{pc\text{-}after}\ \overline{c_1}\ pc)} \overline{c_2}}{P_1, PA_1 \rightsquigarrow P_2, PA_2 \vdash_{pc} \overline{c_1};_{@(P', PA')} \overline{c_2}}$$

$$\textsc{WL\_If}\ \frac{P_1(be) \sqsubseteq l_{be} \quad \mathit{branch\text{-}free}\ \overline{c_1} \quad \mathit{branch\text{-}free}\ \overline{c_2} \quad P_1, PA_1 \rightsquigarrow P_2, PA_2 \vdash_{pc \sqcup l_{be}} \overline{c_1} \quad P_1, PA_1 \rightsquigarrow P_2, PA_2 \vdash_{pc \sqcup l_{be}} \overline{c_2}}{P_1, PA_1 \rightsquigarrow P_2, PA_2 \vdash_{pc} \texttt{if}\ be_{@l_{be}}\ \texttt{then}\ \overline{c_1}\ \texttt{else}\ \overline{c_2}}$$

$$\textsc{WL\_While}\ \frac{P_1(be) \sqsubseteq l_{be} \quad \mathit{branch\text{-}free}\ \overline{c} \quad (P_1, PA_1) \sqsubseteq (P', PA') \quad (P', PA') \sqsubseteq (P_2, PA_2) \quad P', PA' \rightsquigarrow P', PA' \vdash_{pc \sqcup l_{be}} \overline{c_1}}{P_1, PA_1 \rightsquigarrow P_2, PA_2 \vdash_{pc} \texttt{while}\ be_{@l_{be}}\ \texttt{do}\ \overline{c}_{@(P', PA')}}$$

$$\textsc{WL\_ARead}\ \frac{P_1(e) \sqsubseteq l_i \quad pc \sqsubseteq l_{\texttt{X}} \quad l_i \sqsubseteq l_{\texttt{X}} \quad PA_1(\texttt{a}) \sqsubseteq l_{\texttt{X}} \quad ([\texttt{X} \mapsto l_{\texttt{X}}]P_1, PA_1) \sqsubseteq (P_2, PA_2)}{P_1, PA_1 \rightsquigarrow P_2, PA_2 \vdash_{pc} \texttt{X}_{@l_{\texttt{X}}} \leftarrow \texttt{a}[\texttt{e}_{@l_i}]}$$

$$\textsc{WL\_AWrite}\ \frac{P_1(i) \sqsubseteq l_i \quad (P_1, [\texttt{a} \mapsto PA_1(\texttt{a}) \sqcup pc \sqcup l_i \sqcup P_1(e)]PA_1) \sqsubseteq (P_2, PA_2)}{P_1, PA_1 \rightsquigarrow P_2, PA_2 \vdash_{pc} \texttt{a}[i_{@l_i}] \leftarrow e}$$

$$\textsc{WL\_Branch}\ \frac{P_1, PA_1 \rightsquigarrow P_2, PA_2 \vdash_{pc} \overline{c}}{P_1, PA_1 \rightsquigarrow P_2, PA_2 \vdash_{pc} \mathit{branch}\ l\ \overline{c}}$$

Fig. 14. Well-labeledness of annotated commands

The most interesting case is that of loops (WL_While). Instead of doing another fixpoint iteration, we confirm that the annotated labeling is indeed a fixpoint by using it for both the initial and final labeling for the loop body.

Like in the ideal semantics, in WL_Seq, we use $\mathit{pc\text{-}after}\ \overline{c_1}\ l$ to determine the correct program counter label after executing the first part of a sequence. However, this would create problems for the preservation result below if we had branch annotations at arbitrary locations, so we prohibit branch annotations in nonsensical locations using the predicate $\mathit{branch\text{-}free}\ \overline{c}$.

Note that well-labeledness does not require labels to match *precisely*, instead, they may overapproximate. Specifically, the initial labeling can be made more precise and the final labeling less precise while preserving well-labeledness.

Lemma 3.4.2 (IFC analysis produces well-labeled programs).

$$\langle\!\langle c \rangle\!\rangle_{pc}^{P,PA} = (\overline{c}, P', PA') \Rightarrow P, PA \rightsquigarrow P', PA' \vdash_{pc} \overline{c}$$

Proof sketch. By induction on c; most cases follow easily from the definition of well-labeledness. The loop case follows by a nested induction on the number of public variables and arrays, with two base cases: If there are no public variables or arrays left or if the set of public variables and arrays does not shrink in one iteration, then a fixpoint has been reached; in both cases, well-labeledness of the loop body in the fixpoint labeling follows by the outer induction hypothesis. □

Lemma 3.4.3 ($\rightarrow_{\mathsf{i}}$ preserves well-labeledness).

$$P, PA \rightsquigarrow P'', PA'' \vdash_{pc} \overline{c} \quad \Rightarrow$$
$$\langle \overline{c},\ \rho,\ \mu,\ b,\ pc,\ P,\ PA \rangle \xrightarrow[\mathcal{D}]{\mathcal{O}}_{\mathsf{i}} \langle \overline{c'},\ \rho,\ \mu,\ b,\ pc',\ P',\ PA' \rangle \Rightarrow$$
$$P, PA \rightsquigarrow P'', PA'' \vdash_{pc'} \overline{c'}$$

Proof sketch. By induction on the annotated command. The loop case relies on the fact that well-labeledness is preserved when making the initial labeling more precise, as the labeling when reaching the loop is generally more precise than the loop's fixed-point labeling. The sequence case is similar. □

3.4.3 Key Theorems

We have already shown that FvSLH$^{\forall}$ is correct w.r.t. the ideal semantics (Lemma 3.4.1), so it remains to show the relative security of the ideal semantics.

Lemma 3.4.4 ($\rightarrow_{\mathsf{i}}^*$ ensures relative security).

$$\langle\!\langle c \rangle\!\rangle_P^{PA,\mathbb{T}} = (\overline{c}, P', PA') \wedge \rho_1 \sim_P \rho_2 \quad \Rightarrow$$
$$\langle c,\ \rho_1,\ \mu_1 \rangle \approx \langle c,\ \rho_2,\ \mu ... 2 \rangle \quad \Rightarrow$$
$$\langle \overline{c},\ \rho_1,\ \mu_1,\ \mathbb{F},\ \mathbb{T},\ P,\ PA \rangle \approx_i \langle \overline{c},\ \rho_2,\ \mu_2,\ \mathbb{F},\ \mathbb{T},\ P,\ PA \rangle$$

Proof sketch. Unfolding $\approx_i$ exposes two ideal executions with shared directives $\mathcal{D}$, producing observations $\mathcal{O}_1$ and $\mathcal{O}_2$. The goal is to establish their equality.

If the attacker does not force misspeculation, i.e. $\mathcal{D}$ only contains *step* directives, the goal follows immediately, since the executions without misspeculation are identical to sequential executions, and $\langle c,\ \rho_1,\ \mu_1 \rangle \approx \langle c,\ \rho_2,\ \mu_2 \rangle$. Otherwise, the directives must be of the form $\mathcal{D} = [step; ...; step] \cdot [force] \cdot \mathcal{D}'$, neatly decomposing both executions into three parts:

- A nonspeculative prefix, which behaves the same as sequential execution. Thus, these parts produce the same observations, since $\langle c,\ \rho_1,\ \mu_1 \rangle \approx \langle c,\ \rho_2,\ \mu_2 \rangle$ and there is exactly one observation for each directive.
- A step with the *force* directive, initiating misspeculation. This step produces a *branch* observation for both traces, but we can infer that both executions must take the same branch, since we know that the same observation is produced during sequential execution.
- A mispredicted execution suffix. Applying Lemma 3.4.2, we obtain well-labeledness of $\overline{c}$, which is preserved for both executions by Lemma 3.4.3. We can prove that for this part, all observations depend only on the attacker directives and public values: Since masking depends on annotations, and annotations are the same in both executions, any potential leakage is either *masked* in both executions (and thus equal), or labelled as *public*, in which case well-labeledness implies that the leakage is also labelled *public* according to the dynamic tracking, and thus equal for both executions (since $\rho_1 \sim_P \rho_2$, and ideal execution preserves agreement on *public*-labelled variables). We refer to this result as *unwinding of speculative execution.* □

We compose Lemma 3.4.1 and Lemma 3.4.4 to prove the relative security of FvSLH$^{\forall}$. While Lemma 3.4.4 requires the annotated command to be the result of the IFC analysis, this analysis always succeeds, so we obtain relative security *for all programs.*

Theorem 3.4.5 (Relative security of FvSLH$^{\forall}$ for all programs).

$$\mathtt{b} \notin VARS(c) \wedge \rho_1(\mathtt{b}) = 0 \wedge \rho_2(\mathtt{b}) = 0 \quad \Rightarrow$$
$$\left(\forall \mathtt{a}.\ |\ \mathtt{a}\ |_{\mu_1} > 0\right) \wedge \left(\forall \mathtt{a}.\ |\ \mathtt{a}\ |_{\mu_2} > 0\right) \quad \Rightarrow$$
$$\rho_1 \sim_P \rho_2 \wedge \mu_1 \sim_{PA} \mu_2 \quad \Rightarrow$$
$$\langle c,\ \rho_1,\ \mu_1 \rangle \approx \langle c,\ \rho_2,\ \mu_2 \rangle \quad \Rightarrow$$
$$\langle\!\langle c \rangle\!\rangle_{\mathbb{T}}^{P,PA} = (\overline{c}, P', PA') \quad \Rightarrow$$
$$\langle (\!|\overline{c}|\!)^{FvSLH^{\forall}},\ \rho_1,\ \mu_1,\ \mathbb{F} \rangle \approx_s \langle (\!|c|\!)^{FvSLH^{\forall}},\ \rho_2,\ \mu_2,\ \mathbb{F} \rangle$$

4 Correspondences Between Different Speculative Execution Semantics

In this section, we will present two more models of speculative execution, and establish relationships between them. We will begin with a directive-based model that includes rollbacks, adapting (and mechanizing) an equivalence proof by G. Barthe *et al.* [10], before moving to an always-mispredict model (as introduced by M. Guarnieri *et al.* [5]).

However, before proceeding further, we will make one change to the forward-only semantics presented in Fig. 7:

$$\textsc{Spec\_Read}\ \frac{d = step \vee d = read\ \mathtt{b}\ j \quad i = [\![ie]\!]_\rho \quad v = [\![\mathtt{a}[i]]\!]_\mu \quad i < |\ \mathtt{a}\ |_\mu}{\langle \mathtt{X} \leftarrow \mathtt{a}[ie],\ \rho,\ \mu,\ b\rangle \xrightarrow[d]{read\ \mathtt{a}\ i}_{\mathrm{S}} \langle \mathtt{skip},\ [\mathtt{X} \mapsto v]\rho,\ \mu,\ b\rangle}$$

$$\textsc{Spec\_Write}\ \frac{d = step \vee d = write\ \mathtt{b}\ j \quad i = [\![ie]\!]_\rho \quad v = [\![ae]\!]_\rho \quad i < |\ \mathtt{a}\ |_\mu}{\langle \mathtt{a}[ie] \leftarrow ae,\ \rho,\ \mu,\ b\rangle \xrightarrow[d]{write\ \mathtt{a}\ i}_{\mathrm{S}} \langle \mathtt{skip},\ \rho,\ [\mathtt{a}[i] \mapsto v]\mu,\ b\rangle}$$

Fig. 15. Modified Spec_Read and Spec_Write rules for forward-only, directive-based semantics of AWhile

We allow the *normal, unforced* executions to consume not only *step*, but also arbitrary *read* `b` j (resp. *write* `b` j) directives, so long as the index is in bounds. This does not introduce any nondeterminism, since the Spec_Read_Force and Spec_Write_Force rules only apply if the index is out-of-bounds. The importance of this change will be explained in Section 4.2.2 - 4.2.3.

4.1 Equivalence Between Forward-Only and Rollback Semantics

4.1.1 Rollback Semantics

In order to support rollbacks, our semantics must keep track of previous states that a rollback would revert to. Thus, this semantics will not operate on states, but on *configurations*, which are stacks of states. Execution normally only affects the topmost state of the configuration, except for forced mispredictions, which add the mispredicted branch to the top of the stack above the correct branch, and rollbacks, which pop the topmost state from the stack.

$$\textsc{Rb\_Seq\_Step}\ \frac{\langle c_1,\ \rho,\ \mu,\ b\rangle \cdot S \xrightarrow[d]{o}_{\mathrm{rb}} \langle c_{1'},\ \rho',\ \mu',\ b'\rangle \cdot S}{\langle c_1; c_2,\ \rho,\ \mu,\ b\rangle \cdot S \xrightarrow[d]{o}_{\mathrm{rb}} \langle c_{1'}; c_2,\ \rho',\ \mu',\ b'\rangle \cdot S}$$

$$\textsc{Rb\_Seq\_Grow}\ \frac{\langle c_1,\ \rho,\ \mu,\ b\rangle \cdot S \xrightarrow[d]{o}_{\mathrm{rb}} \langle c_{1'},\ \rho',\ \mu',\ b'\rangle \cdot \langle c_{1''},\ \rho'',\ \mu'',\ b''\rangle \cdot S}{\langle c_1; c_2,\ \rho,\ \mu,\ b\rangle \cdot S \xrightarrow[d]{o}_{\mathrm{rb}} \langle c_{1'}; c_2,\ \rho',\ \mu',\ b'\rangle \cdot \langle c_{1''}; c_2,\ \rho'',\ \mu'',\ b''\rangle \cdot S}$$

$$\textsc{Rb\_Seq\_Skip}\ \frac{}{\langle \mathtt{skip}; c,\ \rho,\ \mu,\ b\rangle \cdot S \xrightarrow[\bullet]{\bullet}_{\mathrm{rb}} \langle c,\ \rho,\ \mu,\ b\rangle \cdot S}$$

$$\textsc{Rb\_If}\ \frac{b' = [\![be]\!]_\rho}{\langle \mathtt{if}\ be\ \mathtt{then}\ c_{\mathbb{T}}\ \mathtt{else}\ c_{\mathbb{F}},\ \rho,\ \mu,\ b\rangle \cdot S \xrightarrow[step]{branch\ b'}_{\mathrm{rb}} \langle c_{b'},\ \rho,\ \mu,\ b\rangle \cdot S}$$

$$\textsc{Rb\_If\_Force}\ \frac{b' = [\![be]\!]_\rho}{\langle \mathtt{if}\ be\ \mathtt{then}\ c_{\mathbb{T}}\ \mathtt{else}\ c_{\mathbb{F}},\ \rho,\ \mu,\ b\rangle \cdot S \xrightarrow[force]{branch\ b'}_{\mathrm{rb}} \langle c_{\neg b'},\ \rho,\ \mu,\ \mathbb{T}\rangle \cdot \langle c_{b'},\ \rho,\ \mu,\ b\rangle \cdot S}$$

$$\textsc{Rb\_Read}\ \frac{d = step \vee d = read\ \mathtt{b}\ j \quad i = [\![ie]\!]_\rho \quad v = [\![\mathtt{a}[i]]\!]_\mu \quad i < |\ \mathtt{a}\ |_\mu}{\langle \mathtt{X} \leftarrow \mathtt{a}[ie],\ \rho,\ \mu,\ b\rangle \cdot S \xrightarrow[d]{read\ \mathtt{a}\ i}_{\mathrm{rb}} \langle \mathtt{skip},\ [\mathtt{X} \mapsto v]\rho,\ \mu,\ b\rangle \cdot S}$$

$$\textsc{Rb\_Read\_Force}\ \frac{i = [\![ie]\!]_\rho \quad v = [\![\mathtt{b}[j]]\!]_\mu \quad i \geq |\ \mathtt{a}\ |_\mu \quad j < |\ \mathtt{b}\ |_\mu}{\langle \mathtt{X} \leftarrow \mathtt{a}[ie],\ \rho,\ \mu,\ \mathbb{T}\rangle \cdot S \xrightarrow[load\ \mathtt{b}\ j]{read\ \mathtt{a}\ i}_{\mathrm{rb}} \langle \mathtt{skip},\ [\mathtt{X} \mapsto v]\rho,\ \mu,\ \mathbb{T}\rangle \cdot S}$$

$$\textsc{Rb\_Rollback}\ \frac{}{\langle c,\ \rho,\ \mu,\ b\rangle \cdot \langle c',\ \rho',\ \mu',\ b'\rangle \cdot S \xrightarrow[rollback]{rollback}_{\mathrm{rb}} \langle c',\ \rho',\ \mu',\ b'\rangle \cdot S}$$

Fig. 16. Directive-based semantics with rollbacks (selected rules, see Appendix E)

4.1.2 Equivalence Proof

Recall the definition of $\approx_s$ in Definition 2.3.2. For clarity, we will now write $\approx_{fwd}$ for the forward-only semantics (Fig. 15) and define $\approx_{rb}$ for the semantics with rollbacks (Fig. 16) in the same way.

We will establish that the two semantics are equivalent regarding observational equivalence, i.e.

$$\langle c,\, \rho_1,\, \mu_1,\, \mathbb{F}\rangle \approx_{fwd} \langle c,\, \rho_2,\, \mu_2,\, \mathbb{F}\rangle \Leftrightarrow \langle c,\, \rho_1,\, \mu_1,\, \mathbb{F}\rangle \cdot [] \approx_{rb} \langle c,\, \rho_2,\, \mu_2,\, \mathbb{F}\rangle \cdot []$$

(where [] denotes the empty stack). The proof largely follows the paper proof by G. Barthe *et al.* [10] for a similar language and semantics, which, to our knowledge, has not been mechanized. However, one notable difference is that our semantics has silent steps, which slightly complicates the proofs.

We will cover the right-to-left direction first, as it is much easier.

Theorem 4.1.1 (Observational equivalence in semantics with rollbacks implies forward-only observational equivalence).

$$\langle c,\, \rho_1,\, \mu_1,\, \mathbb{F}\rangle \approx_{rb} \langle c,\, \rho_2,\, \mu_2,\, \mathbb{F}\rangle \Rightarrow \langle c,\, \rho_1,\, \mu_1,\, \mathbb{F}\rangle \cdot [] \approx_{fwd} \langle c,\, \rho_2,\, \mu_2,\, \mathbb{F}\rangle \cdot []$$

Proof sketch. Unfolding $\approx_{fwd}$, we have a pair of forward-only executions with shared directives $\mathcal{D}$, producing observations $\mathcal{O}_1$ and $\mathcal{O}_2$. We further have that all executions in the rollback semantics, starting in the same initial states with equal directives, produce equal observations.

Thus, it suffices to show that for every forward-only execution, there is an execution in the rollback semantics with the same directives producing the same observations:

$$\forall c, \rho, \mu, b, \mathcal{D}, \mathcal{O}, c', \rho', \mu', b'.\langle c,\, \rho,\, \mu,\, b\rangle \xrightarrow[\mathcal{D}]{\mathcal{O}}{}_{\mathrm{S}}{}^{*}\langle c',\, \rho',\, \mu',\, b'\rangle$$

$$\Rightarrow \forall S.\exists S'.\langle c,\, \rho,\, \mu,\, b\rangle \cdot S \xrightarrow[\mathcal{D}]{\mathcal{O}}{}_{\mathrm{rb}}{}^{*}\langle c',\, \rho',\, \mu',\, b'\rangle \cdot S'$$

The proof proceeds by induction on the multi-step execution, with nested induction on the single step. □

The other direction is a lot more complicated, requiring several intermediate lemmas.

First, suppose we have two executions for directives $\mathcal{D} \cdot \mathit{rollback}$, producing the *same observations* $\mathcal{O} \cdot \mathit{rollback}$. We can then "cut out" the suffixes of those executions that are rolled back, producing two shorter executions that still reach the same final state. For bookkeeping purposes, we require that $\mathcal{D}$ does not include any $\mathit{rollback}$, so that we can be sure that the resulting executions are free of rollbacks.

Lemma 4.1.2 (Removing rolled-back execution suffixes).

$$\begin{aligned}
&\mathit{rollback} \notin \mathcal{D} && \Rightarrow\\
&\langle c,\, \rho_1,\, \mu_1,\, \mathbb{F}\rangle \cdot [] \xrightarrow[\mathcal{D}]{\mathcal{O}}{}_{\mathrm{rb}}{}^{*}\langle c_{1'},\, \rho_{1'},\, \mu_{1'},\, b_{1'}\rangle \cdot \langle c_{1''},\, \rho_{1''},\, \mu_{1''},\, b_{1''}\rangle \cdot S_1 \xrightarrow[\mathit{rollback}]{\mathit{rollback}}{}_{\mathrm{rb}} \langle c_{1''},\, \rho_{1''},\, \mu_{1''},\, b_{1''}\rangle \cdot S_1 \Rightarrow\\
&\langle c,\, \rho_2,\, \mu_2,\, \mathbb{F}\rangle \cdot [] \xrightarrow[\mathcal{D}]{\mathcal{O}}{}_{\mathrm{rb}}{}^{*}\langle c_{2'},\, \rho_{2'},\, \mu_{2'},\, b_{2'}\rangle \cdot \langle c_{2''},\, \rho_{2''},\, \mu_{2''},\, b_{2''}\rangle \cdot S_2 \xrightarrow[\mathit{rollback}]{\mathit{rollback}}{}_{\mathrm{rb}} \langle c_{2''},\, \rho_{2''},\, \mu_{2''},\, b_{2''}\rangle \cdot S_2 \Rightarrow\\
&\exists \mathcal{D}', \mathcal{O}'.\quad \langle c,\, \rho_1,\, \mu_1,\, \mathbb{F}\rangle \cdot [] \xrightarrow[\mathcal{D}']{\mathcal{O}'}{}_{\mathrm{rb}}{}^{*}\langle c_{1''},\, \rho_{1''},\, \mu_{1''},\, b_{1''}\rangle \cdot S_1 && \wedge\\
&\qquad\qquad \langle c,\, \rho_2,\, \mu_2,\, \mathbb{F}\rangle \cdot [] \xrightarrow[\mathcal{D}']{\mathcal{O}'}{}_{\mathrm{rb}}{}^{*}\langle c_{2''},\, \rho_{2''},\, \mu_{2''},\, b_{2''}\rangle \cdot S_2 && \wedge\\
&\qquad\qquad \mathit{rollback} \notin \mathcal{D}' \wedge \mid \mathcal{D}' \mid \leq \mid \mathcal{D} \mid
\end{aligned}$$

Proof sketch. By induction (from the right) on $\mathcal{D}$. In the inductive case, $\mathcal{D} = \mathcal{D}' \cdot d$, we decompose $\langle c,\, \rho_1,\, \mu_1,\, \mathbb{F}\rangle \cdot [] \xrightarrow[\mathcal{D}' \cdot d]{\mathcal{O}}{}_{\mathrm{rb}}{}^{*}\langle c_{1'},\, \rho_{1'},\, \mu_{1'},\, b_{1'}\rangle \cdot \langle c_{1''},\, \rho_{1''},\, \mu_{1''},\, b_{1''}\rangle \cdot S_1$ into three parts: A multi-step execution with $\mathcal{D}'$, a single step with d, and another silent multi-step execution after. We distinguish two cases:

- If $d \neq \mathit{force}$, since we already know that $d \neq \mathit{rollback}$, neither the single step with d nor the silent multi-step execution after affect anything below the topmost state on the stack. Thus, the multi-step execution with $\mathcal{D}'$ results in a configuration $\langle c_1^*,\, \rho_1^*,\, \mu_1^*,\, b_1^*\rangle \cdot \langle c_{1''},\, \rho_{1''},\, \mu_{1''},\, b_{1''}\rangle \cdot S_1$ (same for the second execution). Since we can easily construct a step $\langle c_1^*,\, \rho_1^*,\, \mu_1^*,\, b_1^*\rangle \cdot \langle c_{1''},\, \rho_{1''},\, \mu_{1''},\, b_{1''}\rangle \xrightarrow[\mathit{rollback}]{\mathit{rollback}}{}_{\mathrm{rb}} \langle c_{1''},\, \rho_{1''},\, \mu_{1''},\, b_{1''}\rangle$, we can thus apply the induction hypothesis.

- If $d = \mathit{force}$, then we know that this was the step in which the topmost state was added to the stack. We further know that the state underneath, the one that the rollback results in, is the one with the correct branch. Using rule Rb_If, we can thus construct an execution that reaches the same final state, but with no rollback.

□

Using induction, we can use this lemma to "cut out" *all* parts of an execution that were rolled back:

Corollary 4.1.2.1 (Skipping all rolled-back executions).

$$\begin{aligned}
&\langle c,\, \rho_1,\, \mu_1,\, \mathbb{F}\rangle \cdot [] \xrightarrow[\mathcal{D}]{\mathcal{O}}{}_{\mathrm{rb}}{}^{*} \langle c_{1''},\, \rho_{1''},\, \mu_{1''},\, b_{1''}\rangle \cdot S_1 && \Rightarrow \\
&\langle c,\, \rho_2,\, \mu_2,\, \mathbb{F}\rangle \cdot [] \xrightarrow[\mathcal{D}]{\mathcal{O}}{}_{\mathrm{rb}}{}^{*} \langle c_{2''},\, \rho_{2''},\, \mu_{2''},\, b_{2''}\rangle \cdot S_2 && \Rightarrow \\
&\exists \mathcal{D}', \mathcal{O}'.\quad \langle c,\, \rho_1,\, \mu_1,\, \mathbb{F}\rangle \cdot [] \xrightarrow[\mathcal{D}']{\mathcal{O}'}{}_{\mathrm{rb}}{}^{*} \langle c_{1''},\, \rho_{1''},\, \mu_{1''},\, b_{1''}\rangle \cdot S_1 \ \wedge \\
&\qquad\qquad \langle c,\, \rho_2,\, \mu_2,\, \mathbb{F}\rangle \cdot [] \xrightarrow[\mathcal{D}']{\mathcal{O}'}{}_{\mathrm{rb}}{}^{*} \langle c_{2''},\, \rho_{2''},\, \mu_{2''},\, b_{2''}\rangle \cdot S_2 \ \wedge \\
&\qquad\qquad \mathit{rollback} \notin \mathcal{D}'
\end{aligned}$$

Proof sketch. By induction on $|\ \mathcal{D}\ |$. In the inductive case, we decompose $\mathcal{D}$ as $\mathcal{D}' \cdot d$, and correspondingly the multi-step executions into a multi-step part for $\mathcal{D}'$, a single step with directive d, and a silent multi-step execution after. We distinguish two cases:

- If $d \neq \mathit{rollback}$, we apply the induction hypothesis to the multi-step executions for $\mathcal{D}'$. This yields a pair of rollback-free executions resulting in the same states, which we recompose with the single steps for d and the silent suffixes to obtain the desired result.
- If $d = \mathit{rollback}$, we first apply the induction hypothesis to the multi-step execution for $\mathcal{D}'$. This produces a pair of rollback-free executions, which then allows us to apply Lemma 4.1.2.

□

This now allows us to prove the other direction of the equivalence:

Theorem 4.1.3 (Observational equivalence in forward-only semantics implies observational equivalence in semantics with rollbacks).

$$\langle c,\, \rho_1,\, \mu_1,\, \mathbb{F}\rangle \approx_{\mathit{fwd}} \langle c,\, \rho_2,\, \mu_2,\, \mathbb{F}\rangle \Rightarrow \langle c,\, \rho_1,\, \mu_1,\, \mathbb{F}\rangle \cdot [] \approx_{\mathit{rb}} \langle c,\, \rho_2,\, \mu_2,\, \mathbb{F}\rangle \cdot []$$

Proof sketch. Unfolding $\approx_{\mathit{rb}}$, we have a pair of executions in the rollback semantics with shared directives $\mathcal{D}$, producing observations $\mathcal{O}_1$ and $\mathcal{O}_2$. We further have that all executions in the forward-only semantics, starting in the same initial states with equal directives, produce equal observations.

We distinguish two cases:

- If $\mathcal{O}_1 = \mathcal{O}_2$, we are done.
- If $\mathcal{O}_1 \neq \mathcal{O}_2$, then there is a (possibly empty) common prefix $\mathcal{O}$ and two observations $o_1 \neq o_2$ such that $\mathcal{O}_1 = \mathcal{O} \cdot o_1 \cdot \mathcal{O}_{1'}$ and $\mathcal{O}_2 = \mathcal{O} \cdot o_2 \cdot \mathcal{O}_{2'}$. We decompose the executions into a multi-step execution producing $\mathcal{O}$, single steps producing o_1 resp. o_2, and suffixes for $\mathcal{O}_1$ and $\mathcal{O}_2$.

  Using Corollary 4.1.2.1, we obtain rollback-free executions for $\mathcal{O}$ reaching the same final states. Similar to the proof of Theorem 4.1.1, these can be translated into forward-only executions by induction on the multi-step execution with nested induction on the command.

  Further, we know that the directive for the single steps producing o_1 and o_2 can not be *rollback*, as this would imly that both observations are *rollback*, but $o_1 \neq o_2$. Therefore, we obtain forward-only versions of these steps in the same way.

  Composing the obtained forward-only executions results in two executions starting in $\langle c,\, \rho_1,\, \mu_1,\, \mathbb{F}\rangle$ and $\langle c,\, \rho_2,\, \mu_2,\, \mathbb{F}\rangle$ and producing observations $\mathcal{O} \cdot o_1 \neq \mathcal{O} \cdot o_2$, which contradicts the assumption that $\langle c,\, \rho_1,\, \mu_1,\, \mathbb{F}\rangle \approx_{\mathit{fwd}} \langle c,\, \rho_2,\, \mu_2,\, \mathbb{F}\rangle$.

□

4.2 Obervational Equivalence in Directive-Based Semantics Implies Observational Equivalence in Always-Mispredict Semantics

4.2.1 Always-Mispredict Semantics

We now introduce an always-mispredict semantics for AWhile, inspired by M. Guarnieri *et al.* [5]. As the name implies, this semantics *always* triggers misprediction at every branch, instead of allowing an attacker to specify. It also models a speculation window, which decreases with every instruction, and triggers a rollback once exhausted. When first initiating misspeculation, the speculation window is set to a maximum size w_{max}, which we provide as a parameter to the semantics.

In the absence of attacker directions, the attacker can also not specify the locations of out-of-bounds memory accesses. Instead, this semantics is parametric in a memory layout L, which we model as simply a list of variable names specifying an order of arrays. We assume that all arrays are placed one after the other[10]. A function $accessed\text{-}location_\mu^{L(\mathtt{A},i)}$ returns the resulting memory location as a tuple $(\mathtt{b}, j)$ of a (potentially different) array and index within that array, or $\bot$ if the the index is too large to hit any array (in which case the execution gets stuck).

Instead of a boolean flag for misspeculation, our states now include a speculation window $w \in \mathbb{N} \cup \{\bot\}$, where $\bot$ denotes that the current execution is not misspeculating. A predicate *enabled* describes when a state is allowed to take a step (i.e. when $w \neq 0$). Further, we use two helper functions: $decr\ w$, which decrements the window while misspeculating, but leaves $\bot$ unchanged, and $specwin_{w_{max}}\ w$, which sets the window for a new misprediction to $w - 1$ if the current state is already misspeculating, or w_{max} if the current window is $\bot$.

$$\text{AM\_Asgn}\ \frac{enabled\ w \qquad v = [\![ae]\!]_\rho}{\langle \mathtt{X} := ae,\ \rho,\ \mu,\ w\rangle \cdot S \xrightarrow{\bullet}_{am^L_{w_{max}}} \langle \mathtt{skip},\ [\mathtt{X} \mapsto v]\rho,\ \mu,\ decr\ w\rangle \cdot S}$$

$$\text{AM\_Seq\_Step}\ \frac{\langle c_1,\ \rho,\ \mu,\ w\rangle \cdot S \xrightarrow{o}_{am^L_{w_{max}}} \langle c_{1'},\ \rho',\ \mu',\ b'\rangle \cdot S}{\langle c_1; c_2,\ \rho,\ \mu,\ w\rangle \cdot S \xrightarrow{o}_{am^L_{w_{max}}} \langle c_{1'}; c_2,\ \rho',\ \mu',\ w'\rangle \cdot S}$$

$$\text{AM\_Seq\_Grow}\ \frac{\langle c_1,\ \rho,\ \mu,\ w\rangle \cdot S \xrightarrow{o}_{am^L_{w_{max}}} \langle c_{1'},\ \rho',\ \mu',\ w'\rangle \cdot \langle c_{1''},\ \rho'',\ \mu'',\ w''\rangle \cdot S}{\langle c_1; c_2,\ \rho,\ \mu,\ w\rangle \cdot S \xrightarrow{o}_{am^L_{w_{max}}} \langle c_{1'}; c_2,\ \rho',\ \mu',\ w'\rangle \cdot \langle c_{1''}; c_2,\ \rho'',\ \mu'',\ w''\rangle \cdot S}$$

$$\text{AM\_Seq\_Skip}\ \frac{enabled\ w}{\langle \mathtt{skip}; c,\ \rho,\ \mu,\ w\rangle \cdot S \xrightarrow{\bullet}_{am^L_{w_{max}}} \langle c,\ \rho,\ \mu,\ w\rangle \cdot S}$$

$$\text{AM\_While}\ \frac{enabled\ w}{\langle \mathtt{while}\ be\ \mathtt{do}\ c,\ \rho,\ \mu,\ w\rangle \cdot S \xrightarrow{\bullet}_{am^L_{w_{max}}} \langle \mathtt{if}\ be\ \mathtt{then}\ c; \mathtt{while}\ be\ \mathtt{do}\ c\ \mathtt{else}\ \mathtt{skip},\ \rho,\ \mu,\ w\rangle \cdot S}$$

$$\text{AM\_If}\ \frac{enabled\ w \qquad b' = [\![be]\!]_\rho}{\langle \mathtt{if}\ be\ \mathtt{then}\ c_{\mathbb{T}}\ \mathtt{else}\ c_{\mathbb{F}},\ \rho,\ \mu,\ w\rangle \cdot S \xrightarrow{branch\ b'}_{am^L_{w_{max}}} \langle c_{\neg b'},\ \rho,\ \mu,\ specwin_{w_{max}}\ w\rangle \cdot \langle c_{b'},\ \rho,\ \mu,\ decr\ w\rangle \cdot S}$$

$$\text{AM\_Read}\ \frac{enabled\ w \quad i = [\![ie]\!]_\rho \quad accessed\text{-}location_\mu^{L(\mathtt{a},i)} = (\mathtt{b}, j) \quad v = [\![\mathtt{b}[j]]\!]_\mu}{\langle \mathtt{X} \leftarrow \mathtt{a}[ie],\ \rho,\ \mu,\ w\rangle \cdot S \xrightarrow{read\ \mathtt{a}\ i}_{am^L_{w_{max}}} \langle \mathtt{skip},\ [\mathtt{X} \mapsto v]\rho,\ \mu,\ decr\ w\rangle \cdot S}$$

$$\text{AM\_Write}\ \frac{enabled\ w \quad i = [\![ie]\!]_\rho \quad v = [\![ae]\!]_\rho \quad accessed\text{-}location_\mu^{L(\mathtt{a},i)} = (\mathtt{b}, j)}{\langle \mathtt{a}[ie] \leftarrow ae,\ \rho,\ \mu,\ w\rangle \cdot S \xrightarrow{write\ \mathtt{a}\ i}_{am^L_{w_{max}}} \langle \mathtt{skip},\ \rho,\ [\mathtt{b}[j] \mapsto v]\mu,\ decr\ w\rangle \cdot S}$$

$$\text{AM\_Rollback}\ \frac{\neg\ enabled\ w}{\langle c,\ \rho,\ \mu,\ w\rangle \cdot \langle c',\ \rho',\ \mu',\ w'\rangle \cdot S \xrightarrow{rollback}_{am^L_{w_{max}}} \langle c',\ \rho',\ \mu',\ w'\rangle \cdot S}$$

Fig. 17. Always-mispredict semantics for AWhile

[10] We do not explicitly require that the layout contains all arrays used in the program. If it does not, the semantics will simply get stuck.

4.2.2 Observational Equivalence in Directive-Based Semantics Implies Observational Equivalence in Always-Mispredict Semantics, with Restrictions

We first define observational equivalence for the always-mispredict semantics:

Definition 4.2.1 (Observational equivalence in the always-mispredict semantics).

$$\langle c_1,\, \rho_1,\, \mu_1,\, b_1\rangle \approx_{am} \langle c_2,\, \rho_2,\, \mu_2,\, b_2\rangle \doteq \quad \forall \mathcal{O}_1, \mathcal{O}_2. \langle c_1,\, \rho_1,\, \mu_1,\, b_1\rangle \xrightarrow{\mathcal{O}_1}{}_{am^L_{w_{max}}}{}^* \cdot \wedge \langle c_2,\, \rho_2,\, \mu_2,\, b_2\rangle \xrightarrow{\mathcal{O}_2}{}_{am^L_{w_{max}}}{}^*$$
$$\Rightarrow |\mathcal{O}_1| = |\mathcal{O}_2| \Rightarrow \mathcal{O}_1 = \mathcal{O}_2$$

Note that in the absence of directives, which would ensure that both executions proceed equally far, we require an additional assumption that the lengths of $\mathcal{O}_1$ and $\mathcal{O}_2$ are equal. This is equivalent to conluding $\mathcal{O}_1 \gtrless \mathcal{O}_2$ as in Definition 2.3.5, but slightly easier for the proof.

It is clear that the attacker in the directive-based semantics is more powerful than in the always-mispredict semantics, as it can directly control the locations of out-of-bounds memory accesses, whereas the always-mispredict semantics determines those locations based on a memory layout. However, we might still expect that states which satisfy observational equivalence in the directive-based semantics ($\langle c,\, \rho_1,\, \mu_1,\, \mathbb{F}\rangle \cdot [] \approx_{rb} \langle c,\, \rho_2,\, \mu_2,\, \mathbb{F}\rangle \cdot []$) will also satisfy observational equivalence in the always-mispredict semantics ($\langle c,\, \rho_1,\, \mu_1,\, \bot\rangle \cdot [] \approx_{am} \langle c,\, \rho_2,\, \mu_2,\, \bot\rangle \cdot []$). However, this is only true under some restrictions:

- μ_1 and μ_2 must agree on the lengths of arrays, so that out-of-bounds accesses result in the same locations for both executions
- $\langle c,\, \rho_1,\, \mu_1,\, \mathbb{F}\rangle$ and $\langle c,\, \rho_2,\, \mu_2,\, \mathbb{F}\rangle$ must not get stuck in nonspeculative states in in the directive-based semantics, in other words, whenever the misspeculation flag is set to $\mathbb{F}$, it must be possible to proceed either with no directive or with a *step* directive. This is necessary because the directive-based semantics gets stuck if an out-of-bounds access is encountered during nonspeculative execution, whereas the always-mispredict semantics will continue with a memory location determined by the layout, after which we have no more assumptions on the execution. We thus need to rule out such cases.

We will thus prove the following:

Theorem 4.2.2 (Observational equivalence in directive-based semantics implies observational equivalence in always-mispredict semantics).

$$\begin{aligned}
&\left(\forall \mathtt{a}, |\mathtt{a}|_{\mu_1} = |\mathtt{a}|_{\mu_2}\right) && \Rightarrow\\
&\Bigg(\forall \mathcal{D}, \mathcal{O}_1, c_{1'}, \rho_{1'}, \mu_{1'}, S_1.\quad \langle c,\, \rho_1,\, \mu_1,\, \mathbb{F}\rangle \cdot [] \xrightarrow[\mathcal{D}]{\mathcal{O}_1}{}_{\mathrm{rb}}{}^* \langle c_{1'},\, \rho_{1'},\, \mu_{1'},\, \mathbb{F}\rangle \cdot S_1\\
&\qquad\qquad \Rightarrow \langle c_{1'},\, \rho_{1'},\, \mu_{1'},\, \mathbb{F}\rangle \xrightarrow[\bullet]{\bullet}{}_{\mathrm{rb}} \cdot \vee \langle c_{1'},\, \rho_{1'},\, \mu_{1'},\, \mathbb{F}\rangle \xrightarrow[step]{\cdot}{}_{\mathrm{rb}} \cdot\Bigg) && \Rightarrow\\
&\Bigg(\forall \mathcal{D}, \mathcal{O}_2, c_{2'}, \rho_{2'}, \mu_{2'}, S_2. \langle c,\, \rho_2,\, \mu_2,\, \mathbb{F}\rangle \cdot [] \xrightarrow[\mathcal{D}]{\mathcal{O}_2}{}_{\mathrm{rb}}{}^* \langle c_{2'},\, \rho_{2'},\, \mu_{2'},\, \mathbb{F}\rangle \cdot S_2\\
&\qquad\qquad \Rightarrow \langle c_{2'},\, \rho_{2'},\, \mu_{2'},\, \mathbb{F}\rangle \xrightarrow[\bullet]{\bullet}{}_{\mathrm{rb}} \cdot \vee \langle c_{2'},\, \rho_{2'},\, \mu_{2'},\, \mathbb{F}\rangle \xrightarrow[step]{\cdot}{}_{\mathrm{rb}} \cdot\Bigg) && \Rightarrow\\
&\langle c,\, \rho_1,\, \mu_1,\, \mathbb{F}\rangle \cdot [] \approx_{rb} \langle c,\, \rho_2,\, \mu_2,\, \mathbb{F}\rangle \cdot [] && \Rightarrow\\
&\langle c,\, \rho_1,\, \mu_1,\, \bot\rangle \cdot [] \approx_{am} \langle c,\, \rho_2,\, \mu_2,\, \bot\rangle \cdot []
\end{aligned}$$

Proof sketch. Unfolding $\approx_{am}$, we obtain a pair of traces $\langle c,\, \rho_1,\, \mu_1,\, \bot\rangle \cdot [] \xrightarrow{\mathcal{O}_1}{}_{am^L_{w_{max}}} \langle c_{1'},\, \rho_{1'},\, \mu_{1'},\, w_{1'}\rangle \cdot S_1$ and $\langle c,\, \rho_2,\, \mu_2,\, \bot\rangle \cdot [] \xrightarrow{\mathcal{O}_2}{}_{am^L_{w_{max}}} \langle c_{2'},\, \rho_{2'},\, \mu_{2'},\, w_{2'}\rangle \cdot S_2$ such that $|\mathcal{O}_1| = |\mathcal{O}_2|$.

The goal is to show equality of $\mathcal{O}_1$ and $\mathcal{O}_2$.

Since $\langle c,\, \rho_1,\, \mu_1,\, \mathbb{F}\rangle \cdot [] \approx_{rb} \langle c,\, \rho_2,\, \mu_2,\, \mathbb{F}\rangle \cdot []$, it is sufficient to prove that there exists a sequence of directives $\mathcal{D}$ which produces corresponding executions in the rollback semantics:

$$\exists \mathcal{D}, b_{1'}, b_{2'}, S_{1'}, S_{2'}. \langle c, \rho_1, \mu_1, \mathbb{F} \rangle \cdot [] \xrightarrow[\mathcal{D}]{\mathcal{O}_1}_{\text{rb}} \langle c_{1'}, \rho_{1'}, \mu_{1'}, b_{1'} \rangle \cdot S_{1'}$$

$$\wedge \langle c, \rho_1, \mu_1, \mathbb{F} \rangle \cdot [] \xrightarrow[\mathcal{D}]{\mathcal{O}_1}_{\text{rb}} \langle c_{1'}, \rho_{1'}, \mu_{1'}, b_{1'} \rangle \cdot S_{1'}$$

$$\wedge (w_{1'} = \bot \Leftrightarrow b_{1'} = \mathbb{F}) \wedge (w_{1'} = \bot \Leftrightarrow b_{1'} = \mathbb{F}) \wedge S_1 \equiv S_{1'} \wedge S_2 \equiv S_{2'}$$

where $S_1 \equiv S_{1'}$ denotes that S_1 and $S_{1'}$ agree on the program, scalar and array state of each state in the stack, and the misspeculation flag is false iff the speculation window is $\bot$.

Proof by induction (from the right) on $\mathcal{O}_1$ and using the fact that $|\mathcal{O}_1| = |\mathcal{O}_2|$. The base case is easy, as silent steps are essentially the same in both semantics. In the inductive case, we have $\mathcal{O}_1 = \mathcal{O}_{1'} \cdot o_1$ and $\mathcal{O}_2 = \mathcal{O}_{2'} \cdot o_2$. We decompose both executions into three parts: A multi-step execution producing $\mathcal{O}_{1'}$ resp. $\mathcal{O}_{2'}$, a single step producing observation o_1 resp. o_2, and a silent multi-step execution suffix. We apply the induction hypothesis to the first multi-step executions to obtain corresponding directive-based executions. Directive-based executions for the silent suffixes are obtained similar to the base case.

However, before we can make any statements regarding the the single steps which produce o_1 and o_2, we first need to establish that they are at the same point during execution. Towards this end, we first apply observational equivalence in the directive-based semantics to conclude that $\mathcal{O}_{1'} = \mathcal{O}_{2'}$. This implies that the control flow is the same for both executions. Together with the fact that the next step after produces leakage, meaning that both execution have taken all possible silent steps, this allows us to prove that the resulting configurations agree on the program and speculation window of all states (proof by induction on one execution, inversion of the other, and nested induction on the command).

Now, we can proceed using several case distinctions:

- If the speculation window is 0, we know that both single steps must be a rollback, and thus $o_1 = o_2 = \mathit{rollback}$. Thus, we can construct the corresponding step in the rollback semantics using Rb_Rollback.
- Otherwise, we distinguish based on o_1.
  - If $o_1 = \mathit{branch}\ b$, we also know that $o_2 = \mathit{branch}\ b'$ for some b', since both single steps execute the same command. We thus know that both steps must use the rule AM_If (potentially underneath AM_Seq_Grow), thus, we can choose the directive force and construct an equivalent step (by induction on the command, using rules Rb_If_Force and Rb_Seq_Grow).
  - If $o_1 = \mathit{read}\ \mathtt{a}\ i$, we also know that $o_2 = \mathit{read}\ \mathtt{a}\ i'$. However, we do not know whether the indices are in-bounds, or not. We distinguish based on whether we are currently misspeculating:
    - If the speculation window is $\bot$, we can use the assumption that nonspeculative execution in the directive-based semantics does not get stuck. We thus obtain two single steps in the directive-based semantics using the step directive. This, in turn, implies that the indices i and i' are in-bounds, which then means that these steps indeed correspond to the single steps in the always-mispredict semantics.
    - Otherwise, we perform a case distinction on whether i and i' are in-bounds:
      - If both indices are in-bounds, we can construct corresponding steps in the directive-based semantics using the step directive.
      - If both indices are out-of-bounds, we pick the directive $\mathit{load}\ \mathtt{b}\ j$, where $\mathtt{b}[j]$ is the accessed location of one of the two steps, and construct the corresponding steps in the directive-based semantics. Then, we apply observational equivalence in the directive-based semantics to obtain that $\mathit{read}\ \mathtt{a}\ i = \mathit{read}\ \mathtt{a}\ i'$. Since all arrays have the same length in both executions, this means that both steps in the always-mispredict semantics also access the same location, and thus correspond to the directive-based steps we've constructed.
      - If only one index is out-of-bounds, we pick the directive $\mathit{load}\ \mathtt{b}\ j$ for the location $\mathtt{b}[j]$ that is accessed by this index. *This is where the change from Fig. 15 comes into play, as it allows us to construct steps in the directive-based semantics for both executions using the same directive, despite the index being in-bounds for one of them. Without this change, we would not be able to choose a single directive for both executions, and thus would not be able to proceed with the proof.* Once again, we apply observational equivalence in the directive-based semantics to obtain

that $read\ \mathtt{a}\ i = read\ \mathtt{a}\ i'$. We thus obtain a contradiction, as `a` has the same length in both executions, so the index can't be both in-bounds and out-of-bounds.

- The case $o_1 = write\ \mathtt{a}\ i$ proceeds similar to the previous case.

□

4.2.3 Conclusions

In my opinion, the most interesting part of the previous subsection is not the result, but the restrictions we had to add to prove it.

While it is reasonable in our setting to assume that arrays have the same length, this would not apply for languages which support e.g. dynamic allocation. This may indicate that, for such languages, it will be necessary to model the memory layout at least to some extent.

Regarding the restriction that the directive-based execution must not get stuck in nonspeculative states, it reveals that the directive-based semantics implicitly assumes that the program is sequentially memory-safe. While this is not necessarilty a severe restriction - especially in the context of compiler-based mitigations, as it is hard to provide any guarantees if a program exhibits undefined behaviour - it would certainly be preferrable to be explicit about this assumption[11].

Most important, however, is the change we had to make to the semantics in Fig. 15, which is essential for the proof of Theorem 4.2.2 to conclude. Without this change, the semantics simply does not capture leakage due to different memory addresses if only one of those addresses is in-bounds. This is potentially a serious flaw in formal methods, also quite subtle and therefore easy to miss; indeed, it was missed for our work on FSLH (Section 3, [11]) (although this should only affect some minor lemmas, not the overall result).

5 Conclusions and Future Work

This report covered two results towards making efficient Spectre mitigations available for all programs and towards increasing the trustworthiness of the theoretical foundations: In Section 3, I presented a proof of relative security for FvSLH$^{\forall}$, a compiler-based mitigation against Specte-PHT which uses an information-flow analysis to apply protections selectively to only operations which might leak secrets. In Section 4, I present an equivalence proof between directive-based speculation models with and without rollbacks, and further, a proof that observational equivalence in the directive-based model implies observational equivalence in an always-mispredict model, albeit under some restrictions. All proofs have been mechanized in the Rocq proof assistant and are available online[12][13].

As mentioned in Section 4.2.3, the work presented in Section 4 uncovered a flaw with the model used in the security proof of FSLH (Section 3, [11]). While I am reasonably confident that this is not a flaw with the mitigation itself - in particular, the *unwinding* result should still hold - the proofs will need to be adjusted for the fixed model (Fig. 15), requiring modifications to minor lemmas. It will then also be interesting to investigate prior work for similar issues.

Apart from this, the always-mispredict semantics presented here is still very abstract and pretty far removed from real hardware. The limitations and issues we already uncovered with just this step indicate that it would be very beneficial to further close the gap between abstract models and real hardware with mechanized formal proofs.

For FSLH, as mentioned on page 2 of this report, there are still open questions regarding a real-world implementation of this mitigation. At the same time, we are looking at incorporating mitigations against other Spectre vulnerabilities, with the long-term aim of developing a combined mitigation against all known speculative execution attacks with a mechanized proof of relative security.

A References

[1] Intel, "Using Intel Compilers to Mitigate Speculative Execution Side-Channel Issues." Accessed: Jan. 30, 2023. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/troubleshooting/using-intel-compilers-to-mitigate-speculative-execution-side-channel-issues.html

[11] Our own paper [11] falls short in this regard.

[12] https://github.com/secure-compilation/fslh-rocq

[13] https://github.com/secure-compilation/comparing_speculative_semantics_rocq

[2] C. Canella *et al.*, "A Systematic Evaluation of Transient Execution Attacks and Defenses," in *28th USENIX Security Symposium*, N. Heninger and P. Traynor, Eds., USENIX Association, 2019, pp. 249–266. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/canella

[3] Intel, "Intel C++ Compiler Classic Developer Guide and Reference." Accessed: Feb. 01, 2023. [Online]. Available: https://www.intel.com/content/www/us/en/develop/documentation/cpp-compiler-developer-guide-and-reference/top/compiler-reference/compiler-options/code-generation-options/mconditional-branch-qconditional-branch.html

[4] P. Kocher, "Spectre Mitigations in Microsoft's C/C++ Compiler ." Accessed: Aug. 14, 2025. [Online]. Available: https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html

[5] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez, "Spectector: Principled Detection of Speculative Information Flows," in *2020 IEEE Symposium on Security and Privacy, SP*, IEEE, 2020, pp. 1–19. doi: 10.1109/SP40000.2020.00011.

[6] C. Carruth, "Speculative Load Hardening: A Spectre Variant #1 Mitigation Technique." [Online]. Available: https://llvm.org/docs/SpeculativeLoadHardening.html

[7] B. A. Shivakumar *et al.*, "Spectre Declassified: Reading from the Right Place at the Wrong Time," in *44th IEEE Symposium on Security and Privacy, SP*, IEEE, 2023, pp. 1753–1770. doi: 10.1109/SP46215.2023.10179355.

[8] Z. Zhang, G. Barthe, C. Chuengsatiansup, P. Schwabe, and Y. Yarom, "Ultimate SLH: Taking Speculative Load Hardening to the Next Level," in *32nd USENIX Security Symposium*, J. A. Calandrino and C. Troncoso, Eds., USENIX Association, 2023, pp. 7125–7142. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/zhang-zhiyuan-slh

[9] J. B. Almeida *et al.*, "Jasmin: High-Assurance and High-Speed Cryptography," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, in CCS '17. Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 1807–1823. doi: 10.1145/3133956.3134078.

[10] G. Barthe *et al.*, "High-Assurance Cryptography in the Spectre Era," in *42nd IEEE Symposium on Security and Privacy, SP*, IEEE, 2021, pp. 1884–1901. doi: 10.1109/SP40001.2021.00046.

[11] J. Baumann, R. Blanco, L. Ducruet, S. Harwig, and C. Hritcu, "FSLH: Flexible Mechanized Speculative Load Hardening." [Online]. Available: https://arxiv.org/abs/2502.03203

[12] C. Hritcu and R. Blanco, "Security Foundations," 0.1 ed., in Software Foundations., 2025. Accessed: Jul. 17, 2025. [Online]. Available: https://mpi-sp-foe-2025.github.io/book-secf/

[13] F. Willi, "Identifying Compiler Optimizations that Break Constant Time Programming Techniques," 2025. [Online]. Available: https://ethz.ch/content/dam/ethz/special-interest/infk/inst-infsec/appliedcrypto/education/theses/semester-project_fiona-willi.pdf

[14] A. Geimer and C. Maurice, "Fun with flags: How Compilers Break and Fix Constant-Time Code." [Online]. Available: https://arxiv.org/abs/2507.06112

[15] M. Schneider, D. Lain, I. Puddu, N. Dutly, and S. Capkun, "Breaking Bad: How Compilers Break Constant-Time~Implementations." [Online]. Available: https://arxiv.org/abs/2410.13489

[16] M. Guarnieri, B. Köpf, J. Reineke, and P. Vila, "Hardware-Software Contracts for Secure Speculation," in *42nd IEEE Symposium on Security and Privacy, SP*, IEEE, 2021, pp. 1868–1883. doi: 10.1109/SP40001.2021.00036.

[17] "Simple Imperative Programs." in Software Foundations. 2022. Accessed: Jan. 30, 2023. [Online]. Available: https://softwarefoundations.cis.upenn.edu/lf-current/Imp.html

[18] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying Constant-Time Implementations," in *25th USENIX Security Symposium*, T. Holz and S. Savage, Eds., USENIX Association, 2016, pp. 53–70. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/almeida

[19] G. Barthe, G. Betarte, J. D. Campo, C. Luna, and D. Pichardie, "System-Level Non-interference of Constant-Time Cryptography. Part II: Verified Static Analysis and Stealth Memory," *J. Autom. Reason.*, vol. 64, no. 8, pp. 1685–1729, 2020, doi: 10.1007/S10817-020-09548-X.

[20] S. Cauligi *et al.*, "FaCT: a DSL for timing-sensitive computation," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, K. S. McKinley and K. Fisher, Eds., ACM, 2019, pp. 174–189. doi: 10.1145/3314221.3314605.

[21] L.-A. Daniel, S. Bardin, and T. Rezk, "Binsec/Rel: Symbolic Binary Analyzer for Security with Applications to Constant-Time and Secret-Erasure," *ACM Trans. Priv. Secur.*, vol. 26, no. 2, pp. 1–42, 2023, doi: 10.1145/3563037.

[22] Y. Yarom and K. Falkner, "FLUSH+RELOAD: a high resolution, low noise, L3 cache side-channel attack," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, in SEC'14. San Diego, CA: USENIX Association, 2014, pp. 719–732.

[23] P. Kocher *et al.*, "Spectre Attacks: Exploiting Speculative Execution," in *2019 IEEE Symposium on Security and Privacy, SP*, IEEE, 2019, pp. 1–19. doi: 10.1109/SP.2019.00002.

[24] M. Vassena *et al.*, "Automatically eliminating speculative leaks from cryptographic code with Blade," *Proc. ACM Program. Lang.*, vol. 5, no. POPL, pp. 1–30, 2021, doi: 10.1145/3434330.

[25] S. Cauligi *et al.*, "Constant-time foundations for the new Spectre era," in *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI*, A. F. Donaldson and E. Torlak, Eds., ACM, 2020, pp. 913–926. doi: 10.1145/3385412.3385970.

[26] S. Cauligi, C. Disselkoen, D. Moghimi, G. Barthe, and D. Stefan, "SoK: Practical Foundations for Software Spectre Defenses," in *43rd IEEE Symposium on Security and Privacy, SP 2022*, IEEE, 2022, pp. 666–680. doi: 10.1109/SP46214.2022.9833707.

[27] M. Patrignani and M. Guarnieri, "Exorcising Spectres with Secure Compilers," in *2021 ACM SIGSAC Conference on Computer and Communications Security, CCS*, Y. Kim, J. Kim, G. Vigna, and E. Shi, Eds., ACM, 2021, pp. 445–461. doi: 10.1145/3460120.3484534.

[28] D. M. Volpano, C. E. Irvine, and G. Smith, "A Sound Type System for Secure Flow Analysis," *J. Comput. Secur.*, vol. 4, no. 2/3, pp. 167–188, 1996, doi: 10.3233/JCS-1996-42-304.

[29] S. Hunt and D. Sands, "On flow-sensitive security types," in *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, J. G. Morrisett and S. L. P. Jones, Eds., ACM, 2006, pp. 79–90. doi: 10.1145/1111037.1111045.

B Sequential Semantics of AWhile

$$\textsc{Seq\_Asgn}\ \frac{v = [\![ae]\!]_\rho}{\langle \mathtt{X} := ae,\ \rho,\ \mu\rangle \xrightarrow{\bullet} \langle \mathtt{skip},\ [\mathtt{X} \mapsto v]\rho,\ \mu\rangle} \qquad \textsc{Seq\_Seq\_Step}\ \frac{\langle c_1,\ \rho,\ \mu\rangle \xrightarrow[\bullet]{o}_{\mathrm{S}} \langle c_{1'},\ \rho',\ \mu'\rangle}{\langle c_1; c_2,\ \rho,\ \mu\rangle \xrightarrow{o} \langle c_{1'}; c_2,\ \rho',\ \mu'\rangle}$$

$$\textsc{Seq\_Seq\_Skip}\ \frac{}{\langle \mathtt{skip}; c,\ \rho,\ \mu\rangle \xrightarrow{\bullet} \langle c,\ \rho,\ \mu\rangle}$$

$$\textsc{Seq\_While}\ \frac{}{\langle \mathtt{while}\ be\ \mathtt{do}\ c,\ \rho,\ \mu\rangle \xrightarrow{\bullet} \langle \mathtt{if}\ be\ \mathtt{then}\ c; \mathtt{while}\ be\ \mathtt{do}\ c\ \mathtt{else}\ \mathtt{skip},\ \rho,\ \mu\rangle}$$

$$\textsc{Seq\_If}\ \frac{b' = [\![be]\!]_\rho}{\langle \mathtt{if}\ be\ \mathtt{then}\ c_{\mathbb{T}}\ \mathtt{else}\ c_{\mathbb{F}},\ \rho,\ \mu\rangle \xrightarrow{branch\ b'} \langle c_{b'},\ \rho,\ \mu\rangle}$$

$$\textsc{Seq\_Read}\ \frac{i = [\![ie]\!]_\rho \qquad v = [\![\mathtt{a}[i]]\!]_\mu \qquad i < |\ \mathtt{a}\ |_\mu}{\langle \mathtt{X} \leftarrow \mathtt{a}[ie],\ \rho,\ \mu\rangle \xrightarrow{read\ \mathtt{a}\ i} \langle \mathtt{skip},\ [\mathtt{X} \mapsto v]\rho,\ \mu\rangle}$$

$$\textsc{Seq\_Write}\ \frac{i = [\![ie]\!]_\rho \qquad v = [\![ae]\!]_\rho \qquad i < |\ \mathtt{a}\ |_\mu}{\langle \mathtt{a}[ie] \leftarrow ae,\ \rho,\ \mu\rangle \xrightarrow{write\ \mathtt{a}\ i} \langle \mathtt{skip},\ \rho,\ [\mathtt{a}[i] \mapsto v]\mu\rangle}$$

Fig. 18. Sequential semantics of AWhile

C Forward-Only, Directive-Based Speculative Semantics of AWhile

$$\textsc{Spec\_Asgn}\ \frac{v = [\![ae]\!]_\rho}{\langle \mathtt{X} := ae,\ \rho,\ \mu,\ b\rangle \xrightarrow[\bullet]{}_{\mathrm{S}} \langle \mathtt{skip},\ [\mathtt{X} \mapsto v]\rho,\ \mu,\ b\rangle}$$

$$\textsc{Spec\_Seq\_Step}\ \frac{\langle c_1,\ \rho,\ \mu,\ b\rangle \xrightarrow[d]{o}_{\mathrm{S}} \langle c_{1'},\ \rho',\ \mu',\ b'\rangle}{\langle c_1; c_2,\ \rho,\ \mu,\ b\rangle \xrightarrow[d]{o}_{\mathrm{S}} \langle c_{1'}; c_2,\ \rho',\ \mu',\ b'\rangle}$$

$$\textsc{Spec\_Seq\_Skip}\ \frac{}{\langle \mathtt{skip}; c,\ \rho,\ \mu,\ b\rangle \xrightarrow[\bullet]{}_{\mathrm{S}} \langle c,\ \rho,\ \mu,\ b\rangle}$$

$$\textsc{Spec\_While}\ \frac{}{\langle \mathtt{while}\ be\ \mathtt{do}\ c,\ \rho,\ \mu,\ b\rangle \xrightarrow[\bullet]{}_{\mathrm{S}} \langle \mathtt{if}\ be\ \mathtt{then}\ c; \mathtt{while}\ be\ \mathtt{do}\ c\ \mathtt{else}\ \mathtt{skip},\ \rho,\ \mu,\ b\rangle}$$

$$\textsc{Spec\_If}\ \frac{b' = [\![be]\!]_\rho}{\langle \mathtt{if}\ be\ \mathtt{then}\ c_{\mathbb{T}}\ \mathtt{else}\ c_{\mathbb{F}},\ \rho,\ \mu,\ b\rangle \xrightarrow[step]{branch\ b'}_{\mathrm{S}} \langle c_{b'},\ \rho,\ \mu,\ b\rangle}$$

$$\textsc{Spec\_If\_Force}\ \frac{b' = [\![be]\!]_\rho}{\langle \mathtt{if}\ be\ \mathtt{then}\ c_{\mathbb{T}}\ \mathtt{else}\ c_{\mathbb{F}},\ \rho,\ \mu,\ b\rangle \xrightarrow[force]{branch\ b'}_{\mathrm{S}} \langle c_{\neg b'},\ \rho,\ \mu,\ \mathbb{T}\rangle}$$

$$\textsc{Spec\_Read}\ \frac{i = [\![ie]\!]_\rho \qquad v = [\![\mathtt{a}[i]]\!]_\mu \qquad i < |\ \mathtt{a}\ |_\mu}{\langle \mathtt{X} \leftarrow \mathtt{a}[ie],\ \rho,\ \mu,\ b\rangle \xrightarrow[step]{read\ \mathtt{a}\ i}_{\mathrm{S}} \langle \mathtt{skip},\ [\mathtt{X} \mapsto v]\rho,\ \mu,\ b\rangle}$$

$$\textsc{Spec\_Read\_Force}\ \frac{i = [\![ie]\!]_\rho \qquad v = [\![\mathtt{b}[j]]\!]_\mu \qquad i \geq |\ \mathtt{a}\ |_\mu \qquad j < |\ \mathtt{b}\ |_\mu}{\langle \mathtt{X} \leftarrow \mathtt{a}[ie],\ \rho,\ \mu,\ \mathbb{T}\rangle \xrightarrow[load\ \mathtt{b}\ j]{read\ \mathtt{a}\ i}_{\mathrm{S}} \langle \mathtt{skip},\ [\mathtt{X} \mapsto v]\rho,\ \mu,\ \mathbb{T}\rangle}$$

$$\textsc{Spec\_Write}\ \frac{i = [\![ie]\!]_\rho \qquad v = [\![ae]\!]_\rho \qquad i < |\ \mathtt{a}\ |_\mu}{\langle \mathtt{a}[ie] \leftarrow ae,\ \rho,\ \mu,\ b\rangle \xrightarrow[step]{write\ \mathtt{a}\ i}_{\mathrm{S}} \langle \mathtt{skip},\ \rho,\ [\mathtt{a}[i] \mapsto v]\mu,\ b\rangle}$$

$$\textsc{Spec\_Write\_Force}\ \frac{i = [\![ie]\!]_\rho \qquad v = [\![ae]\!]_\rho \qquad i \geq |\ \mathtt{a}\ |_\mu \qquad j < |\ \mathtt{b}\ |_\mu}{\langle \mathtt{a}[ie] \leftarrow ae,\ \rho,\ \mu,\ \mathbb{T}\rangle \xrightarrow[store\ \mathtt{b}\ j]{write\ \mathtt{a}\ i}_{\mathrm{S}} \langle \mathtt{skip},\ \rho,\ [\mathtt{b}[j] \mapsto v]\mu,\ \mathbb{T}\rangle}$$

Fig. 19. (Forward-only, directive-based) speculative semantics of AWhile (selected rules)

D Ideal Semantics for FvSLH$^{\forall}$

$$\textsc{Ideal\_Asgn}\ \frac{v = [\![ae]\!]_\rho}{\langle \mathtt{X} := ae,\ \rho,\ \mu,\ b,\ pc,\ P,\ PA\rangle \xrightarrow[\bullet]{\bullet}_{\mathrm{S}} \langle \mathtt{skip},\ [\mathtt{X} \mapsto v]\rho,\ \mu,\ b,\ pc,\ [\mathtt{X} \mapsto pc \sqcup P(ae)]P,\ PA\rangle}$$

$$\textsc{Ideal\_Seq\_Step}\ \frac{\langle \overline{c_1},\ \rho,\ \mu,\ b,\ pc,\ P,\ PA\rangle \xrightarrow[d]{o}_{\mathrm{S}} \langle \overline{c_{1'}},\ \rho',\ \mu',\ b',\ pc'',\ P'',\ PA''\rangle}{\langle \overline{c_1};_{@(P',PA')}\ \overline{c_2},\ \rho,\ \mu,\ b,\ pc,\ P,\ PA\rangle \xrightarrow[d]{o}_{\mathrm{S}} \langle \overline{c_{1'}};_{@(P',PA')}\ \overline{c_2},\ \rho',\ \mu',\ b',\ pc'',\ P'',\ PA''\rangle}$$

$$\textsc{Ideal\_Seq\_Skip}\ \frac{terminal\ \overline{c_1}}{\langle \overline{c_1};_{@(P',PA')}\ \overline{c_2},\ \rho,\ \mu,\ b,\ pc,\ P,\ PA\rangle \xrightarrow[\bullet]{\bullet}_{\mathrm{i}} \langle \overline{c_2},\ \rho,\ \mu,\ b,\ pc\text{-}after\ \overline{c_1}\ \ pc,\ P,\ PA\rangle}$$

$$\textsc{Ideal\_While}\ \frac{}{\begin{array}{l}\langle \mathtt{while}\ be_{@l}\ \mathtt{do}\ \overline{c}_{@(P',PA')},\ \rho,\ \mu,\ b,\ pc,\ P,\ PA\rangle \xrightarrow[\bullet]{\bullet}_{\mathrm{S}} \\ \langle \mathtt{if}\ be_{@l}\ \mathtt{then}\ c;_{@(P',PA')}\ \mathtt{while}\ be_{@l}\ \mathtt{do}\ \overline{c}_{@(P',PA')}\ \mathtt{else}\ \mathtt{skip},\ \rho,\ \mu,\ b\rangle\end{array}}$$

$$\textsc{Ideal\_If}\ \frac{b' = (l \vee \neg b) \wedge [\![be]\!]_\rho}{\langle \mathtt{if}\ be_{@l}\ \mathtt{then}\ \overline{c_{\mathbb{T}}}\ \mathtt{else}\ \overline{c_{\mathbb{F}}},\ \rho,\ \mu,\ b,\ pc,\ P,\ PA\rangle \xrightarrow[step]{branch\ b'}_{\mathrm{i}} \langle branch\ pc\ \ \overline{c_{b'}},\ \rho,\ \mu,\ b,\ pc \sqcup l,\ P,\ PA\rangle}$$

$$\textsc{Ideal\_If\_Force}\ \frac{b' = (l \vee \neg b) \wedge [\![be]\!]_\rho}{\langle \mathtt{if}\ be_{@l}\ \mathtt{then}\ \overline{c_{\mathbb{T}}}\ \mathtt{else}\ \overline{c_{\mathbb{F}}},\ \rho,\ \mu,\ b,\ pc,\ P,\ PA\rangle \xrightarrow[force]{branch\ b'}_{\mathrm{i}} \langle branch\ pc\ \ \overline{c_{\neg b'}},\ \rho,\ \mu,\ \mathbb{T},\ pc \sqcup l,\ P,\ PA\rangle}$$

$$\textsc{Ideal\_Read}\ \frac{i = \begin{cases} 0 & if\ \neg(l_i) \wedge b \\ [\![ie]\!]_\rho & otherwise \end{cases} \qquad v = \begin{cases} 0 & if\ l_{\mathtt{X}} \wedge l_i \wedge b \\ [\![\mathtt{a}[i]]\!]_\mu & otherwise \end{cases} \qquad i < |\ \mathtt{a}\ |_\mu}{\langle \mathtt{X}_{@l_{\mathtt{X}}} \leftarrow \mathtt{a}[ie_{@l_i}],\ \rho,\ \mu,\ b,\ pc,\ P,\ PA\rangle \xrightarrow[step]{read\ \mathtt{a}\ i}_{\mathrm{i}} \langle \mathtt{skip},\ [\mathtt{X} \mapsto v]\rho,\ \mu,\ b,\ pc,\ [\mathtt{X} \mapsto l_{\mathtt{X}}]P,\ PA\rangle}$$

$$\textsc{Ideal\_Read\_Force}\ \frac{i = [\![ie]\!]_\rho \qquad v = \begin{cases} 0 & if\ l_{\mathtt{X}} \\ [\![\mathtt{b}[j]]\!]_\mu & otherwise \end{cases} \qquad i \geq |\ \mathtt{a}\ |_\mu \qquad j < |\ \mathtt{b}\ |_\mu}{\langle \mathtt{X}_{@l_{\mathtt{X}}} \leftarrow \mathtt{a}[ie_{@\mathbb{T}}],\ \rho,\ \mu,\ \mathbb{T},\ pc,\ P,\ PA\rangle \xrightarrow[load\ \mathtt{b}\ j]{read\ \mathtt{a}\ i}_{\mathrm{i}} \langle \mathtt{skip},\ [\mathtt{X} \mapsto v]\rho,\ \mu,\ \mathbb{T},\ pc,\ [\mathtt{X} \mapsto l_{\mathtt{X}}]P,\ PA\rangle}$$

$$\textsc{Ideal\_Write}\ \frac{i = \begin{cases} 0 & if\ \neg l_i \wedge b \\ [\![ie]\!]_\rho & otherwise \end{cases} \qquad v = [\![ae]\!]_\rho \qquad i < |\ \mathtt{a}\ |_\mu}{\langle \mathtt{a}[ie_{@l_{ie}}] \leftarrow ae,\ \rho,\ \mu,\ b,\ pc,\ P,\ PA\rangle \xrightarrow[step]{write\ \mathtt{a}\ i}_{\mathrm{i}} \langle \mathtt{skip},\ \rho,\ [\mathtt{a}[i] \mapsto v]\mu,\ b,\ pc,\ P,\ [\mathtt{a} \mapsto P(\mathtt{a}) \sqcup pc \sqcup l_{ie} \sqcup P(ae)]\ PA\rangle}$$

$$\textsc{Ideal\_Write\_Force}\ \frac{i = [\![ie]\!]_\rho \qquad v = [\![ae]\!]_\rho \qquad i \geq |\ \mathtt{a}\ |_\mu \qquad j < |\ \mathtt{b}\ |_\mu}{\langle \mathtt{a}[ie_{@\mathbb{T}}] \leftarrow ae,\ \rho,\ \mu,\ \mathbb{T},\ pc,\ P,\ PA\rangle \xrightarrow[store\ \mathtt{b}\ j]{write\ \mathtt{a}\ i}_{\mathrm{i}} \langle \mathtt{skip},\ \rho,\ [\mathtt{b}[j] \mapsto v]\mu,\ \mathbb{T},\ pc,\ P,\ [\mathtt{a} \mapsto PA(\mathtt{a}) \sqcup pc \sqcup P(ae)]\ PA\rangle}$$

$$\textsc{Ideal\_Branch}\ \frac{\langle \overline{c},\ \rho,\ \mu,\ b,\ pc,\ P,\ PA\rangle \xrightarrow[d]{o}_{\mathrm{i}} \langle \overline{c'},\ \rho',\ \mu',\ b',\ pc',\ P',\ PA'\rangle}{\langle branch\ l\ \overline{c},\ \rho,\ \mu,\ b,\ pc,\ P,\ PA\rangle \xrightarrow[d]{o}_{\mathrm{i}} \langle branch\ l\ \overline{c'},\ \rho',\ \mu',\ b',\ pc',\ P',\ PA'\rangle}$$

Fig. 20. Ideal semantics for FvSLH$^{\forall}$

E Directive-based Semantics with Rollbacks

$$\textsc{Rb\_Asgn}\ \frac{v = [\![ae]\!]_\rho}{\langle \mathtt{X} := ae,\ \rho,\ \mu,\ b\rangle \cdot S \xrightarrow[\bullet]{\bullet}_{\mathrm{rb}} \langle \mathtt{skip},\ [\mathtt{X} \mapsto v]\rho,\ \mu,\ b\rangle \cdot S}$$

$$\textsc{Rb\_Seq\_Step}\ \frac{\langle c_1,\ \rho,\ \mu,\ b\rangle \cdot S \xrightarrow[d]{o}_{\mathrm{rb}} \langle c_{1'},\ \rho',\ \mu',\ b'\rangle \cdot S}{\langle c_1; c_2,\ \rho,\ \mu,\ b\rangle \cdot S \xrightarrow[d]{o}_{\mathrm{rb}} \langle c_{1'}; c_2,\ \rho',\ \mu',\ b'\rangle \cdot S}$$

$$\textsc{Rb\_Seq\_Grow}\ \frac{\langle c_1,\ \rho,\ \mu,\ b\rangle \cdot S \xrightarrow[d]{o}_{\mathrm{rb}} \langle c_{1'},\ \rho',\ \mu',\ b'\rangle \cdot \langle c_{1''},\ \rho'',\ \mu'',\ b''\rangle \cdot S}{\langle c_1; c_2,\ \rho,\ \mu,\ b\rangle \cdot S \xrightarrow[d]{o}_{\mathrm{rb}} \langle c_{1'}; c_2,\ \rho',\ \mu',\ b'\rangle \cdot \langle c_{1''}; c_2,\ \rho'',\ \mu'',\ b''\rangle \cdot S}$$

$$\textsc{Rb\_Seq\_Skip}\ \frac{}{\langle \mathtt{skip}; c,\ \rho,\ \mu,\ b\rangle \cdot S \xrightarrow[\bullet]{\bullet}_{\mathrm{rb}} \langle c,\ \rho,\ \mu,\ b\rangle \cdot S}$$

$$\textsc{Rb\_While}\ \frac{}{\langle \mathtt{while}\ be\ \mathtt{do}\ c,\ \rho,\ \mu,\ b\rangle \cdot S \xrightarrow[\bullet]{\bullet}_{\mathrm{rb}} \langle \mathtt{if}\ be\ \mathtt{then}\ c; \mathtt{while}\ be\ \mathtt{do}\ c\ \mathtt{else}\ \mathtt{skip},\ \rho,\ \mu,\ b\rangle \cdot S}$$

$$\textsc{Rb\_If}\ \frac{b' = [\![be]\!]_\rho}{\langle \mathtt{if}\ be\ \mathtt{then}\ c_{\mathbb{T}}\ \mathtt{else}\ c_{\mathbb{F}},\ \rho,\ \mu,\ b\rangle \cdot S \xrightarrow[step]{branch\ b'}_{\mathrm{rb}} \langle c_{b'},\ \rho,\ \mu,\ b\rangle \cdot S}$$

$$\textsc{Rb\_If\_Force}\ \frac{b' = [\![be]\!]_\rho}{\langle \mathtt{if}\ be\ \mathtt{then}\ c_{\mathbb{T}}\ \mathtt{else}\ c_{\mathbb{F}},\ \rho,\ \mu,\ b\rangle \cdot S \xrightarrow[force]{branch\ b'}_{\mathrm{rb}} \langle c_{\neg b'},\ \rho,\ \mu,\ \mathbb{T}\rangle \cdot \langle c_{b'},\ \rho,\ \mu,\ b\rangle \cdot S}$$

$$\textsc{Rb\_Read}\ \frac{d = step \vee d = read\ \mathtt{b}\ j \quad i = [\![ie]\!]_\rho \quad v = [\![\mathtt{a}[i]]\!]_\mu \quad i < |\ \mathtt{a}\ |_\mu}{\langle \mathtt{X} \leftarrow \mathtt{a}[ie],\ \rho,\ \mu,\ b\rangle \cdot S \xrightarrow[d]{read\ \mathtt{a}\ i}_{\mathrm{rb}} \langle \mathtt{skip},\ [\mathtt{X} \mapsto v]\rho,\ \mu,\ b\rangle \cdot S}$$

$$\textsc{Rb\_Read\_Force}\ \frac{i = [\![ie]\!]_\rho \quad v = [\![\mathtt{b}[j]]\!]_\mu \quad i \geq |\ \mathtt{a}\ |_\mu \quad j < |\ \mathtt{b}\ |_\mu}{\langle \mathtt{X} \leftarrow \mathtt{a}[ie],\ \rho,\ \mu,\ \mathbb{T}\rangle \cdot S \xrightarrow[load\ \mathtt{b}\ j]{read\ \mathtt{a}\ i}_{\mathrm{rb}} \langle \mathtt{skip},\ [\mathtt{X} \mapsto v]\rho,\ \mu,\ \mathbb{T}\rangle \cdot S}$$

$$\textsc{Rb\_Write}\ \frac{d = step \vee d = write\ \mathtt{b}\ j \quad i = [\![ie]\!]_\rho \quad v = [\![ae]\!]_\rho \quad i < |\ \mathtt{a}\ |_\mu}{\langle \mathtt{a}[ie] \leftarrow ae,\ \rho,\ \mu,\ b\rangle \cdot S \xrightarrow[d]{write\ \mathtt{a}\ i}_{\mathrm{rb}} \langle \mathtt{skip},\ \rho,\ [\mathtt{a}[i] \mapsto v]\mu,\ b\rangle \cdot S}$$

$$\textsc{Rb\_Write\_Force}\ \frac{i = [\![ie]\!]_\rho \quad v = [\![ae]\!]_\rho \quad i \geq |\ \mathtt{a}\ |_\mu \quad j < |\ \mathtt{b}\ |_\mu}{\langle \mathtt{a}[ie] \leftarrow ae,\ \rho,\ \mu,\ \mathbb{T}\rangle \cdot S \xrightarrow[store\ \mathtt{b}\ j]{write\ \mathtt{a}\ i}_{\mathrm{rb}} \langle \mathtt{skip},\ \rho,\ [\mathtt{b}[j] \mapsto v]\mu,\ \mathbb{T}\rangle \cdot S}$$

$$\textsc{Rb\_Rollback}\ \frac{}{\langle c,\ \rho,\ \mu,\ b\rangle \cdot \langle c',\ \rho',\ \mu',\ b'\rangle \cdot S \xrightarrow[rollback]{rollback}_{\mathrm{rb}} \langle c',\ \rho',\ \mu',\ b'\rangle \cdot S}$$

Fig. 21. Directive-based semantics with rollbacks