Saarland University
Faculty of Mathematics and Computer Science
Department of Computer Science

Bachelor's Thesis

Verifying Spectre Countermeasures Using Hypersimulations

by
Jonathan Baumann

submitted
July 14th, 2023

Reviewers:

1. Prof. Dr. Sebastian Hack
2. Dr. Swen Jacobs

Advisor:

Julian Rosemann

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken,_____ _____

(Datum/Date) (Unterschrift/Signature)

Abstract

Several mitigations for the Spectre class of vulnerabilities have been proposed and implemented in different compilers. However, formally verifying that these compiler countermeasures are indeed effective is challenging. Existing methods overapproximate, leading to unverifiable mitigations, and can only reason about leakage models that include branch outcomes. We present a new approach based on hypersimulations that can be used to verify compiler countermeasures for a variety of leakage models, and can handle countermeasures that previously could not be verified. We also discuss an issue with Speculative Load Hardening in some leakage models. Our work has been formalized in the Coq proof assistant.

Acknowledgments

I would like to thank my advisor, Julian Rosemann, for his outstanding support during this project, being available for questions and discussions at all times. I also want to thank him for offering this project to me in the first place, and for incorporating some of my suggestions into his own work.

I'm also thankful to Prof. Sebastian Hack for allowing me to write this thesis at his group and to Dr. Swen Jacobs for sparking my interest in this topic with a very interesting seminar. I also thank them both for reviewing this thesis.

Finally, I am thankful to my friends and family for supporting me during the last weeks. I am especially thankful to Iona, Johannes, Rafaela, Oliver and Robert for proof-reading my thesis and providing helpful suggestions.

Contents

List of Figures

1 Introduction

The Spectre class of vulnerabilities, first revealed in early 2018 by Kocher et al. [2019], is a class of hardware vulnerabilities present in most recent CPUs. It exploits a feature called speculative execution, which is used by pipelined processors to avoid stalls. This is achieved by predicting, among other things, the outcomes of branches, store-to-load dependencies or even return addresses [Canella et al., 2019]. The CPU will continue executing according to the prediction. If the prediction turns out to be incorrect, the internal state will be rolled back, and execution continues along the proper path.

This entire process is supposed to be transparent, i.e., it should not be observable above the microarchitectural level. However, Kocher et al. [2019] discovered that effects of speculative execution on the cache, for instance, are not rolled back and can produce observable side-effects. This allows attackers to extract information from rolled back executions, bypassing bounds checks and other safety measures.

A variety of mitigations for Spectre, both in hardware and in software, have been proposed since then. While specific vulnerabilities have successfully been mitigated in processors using microcode updates according to Canella et al. [2019], many Spectre variations remain unmitigated. Generally, hardware mitigations require changes to the microarchitectural design, which would not only take a long time to implement, but may also be prohibitively expensive. Even if successfully implemented, however, hardware mitigations can only protect future processors, and software mitigations are necessary to protect programs running on current hardware.

Software mitigations generally fall into two categories: Tools like oo7 [Wang et al., 2018] and Spectector [Guarnieri et al., 2018] rely on program analysis after compilation, whereas others, like Speculative Load Hardening [Carruth, 2018], the mitigation implemented in the Microsoft Visual C++ compiler (MSVC)[Pardoe, 2018], or the initial mitigation proposed by Intel [Intel, 2018], are performed during compilation. Compiler-based countermeasures are generally quick to apply, but not as precise.

Of course, one way to prevent leakage through speculative execution is to prevent speculative execution altogether, as Intel's mitigation achieves. This, however, impacts performance significantly[1]. Thus, the goal of a mitigation is not just to prevent leakage through speculative execution, but to do so while reducing the performance impact. This makes mitigations more complicated, and verifying their correctness nontrivial.

Mitigations relying on program analysis, such as Spectector [Guarnieri et al., 2018], can ensure that the resulting program is secure by showing that there are no counterexamples.While this could also be done as translation validation after applying a

[1]Wang et al. [2018] report a performance overhead of 430% for the insertion of fence instructions at every branch

compiler-based countermeasure, this would negate the time benefits of using simpler compiler mitigations.

Thus, it would be preferable to prove once that a mitigation produces secure programs, removing the need to verify the result. We therefore present a general method of verifying compiler countermeasures.

1.1 Related Work

Building on work by Goguen and Meseguer [1982], noninterference is commonly used to reason about information flow security. As defined by Barthe et al. [2018], it requires that all observations produced by a program depend only on public inputs that are known to or even chosen by the attacker, and not on any secrets. To determine whether a program satisfies noninterference, it is not sufficient to look at individual traces. Instead, it is necessary to consider multiple traces to see whether they produce the same leakage.

Noninterference is thus an example of a hyperproperty [Clarkson and Schneider, 2010]. Hyperproperties generalize properties, which are sets of traces (usually given as a predicate). Clarkson and Schneider [2010] also generalized the concepts of safety and liveness properties to hypersafety and hyperliveness.

For reasoning about Spectre vulnerabilities, Guarnieri et al. [2018] defined speculative noninterference, a hyperproperty that accurately describes when a program is free of spectre vulnerabilities. Speculative noninterference requires that two executions of a program may only be distinguishable under speculative execution if they are also distinguishable nonspeculatively, in other words, equality of observable side effects must be preserved under speculative execution. Guarnieri et al. [2018] achieve this comparison by recovering the nonspeculative parts from traces with speculative execution.

Both noninterference and speculative noninterference are parametric in a leakage model, but they are most commonly used with the constant-time leakage model [Barthe et al., 2018, Guarnieri et al., 2018], which considers all control flow and the addresses of all memory accesses as visible to an attacker and therefore captures all leakage through timing and cache-timing side channels.

Cheang et al. [2019] gave an alternative hyperproperty for Spectre vulnerabilities, trace property-dependent observational determinism. Instead of recovering the nonspeculative behaviour from a speculative trace, they consider four traces, of which two are not allowed to misspeculate.

Guarnieri et al. [2018] have also introduced always-mispredict semantics as a way to capture leakage introduced by speculative execution without having to accurately model a branch predictor.

A technique to verify spectre mitigations in compilers was developed by Patrignani and Guarnieri [2021]. Their approach avoids reasoning about multiple executions by defining speculative safety, a property of a single trace, as an approximation of speculative noninterference. They then treat the mitigation as a compilation pass from a language without speculative execution to one with speculative execution. As programs

without speculative execution always satisfy speculative safety, they can prove that a mitigation is secure by proving that it preserves speculative safety, for which they use backtranslation.

However, speculative safety relies on taint tracking and does not keep track of implicit flows, as it was designed for the constant-time leakage model, where such flows would appear indirectly via the leaked branch outcome. Therefore, speculative safety can not be used with leakage models that do not include control flow, as the taint tracking would not be correct. Furthermore, even in leakage models that do leak control flow, there are programs that satisfy speculative noninterference, but not speculative safety. Thus, their approach is not able to verify all mitigations.

For classical side-channel attacks (that do not exploit speculative execution), Barthe et al. [2018] developed a technique to show that constant-time noninterference is preserved by individual compilation steps. They extended the common simulation approach used for regular properties to constant-time simulations, which now include two executions of the source program and two of the compiled program. This allows them to prove the preservation of noninterference by proving that for every step where the source traces produce equal leakage, the target traces also produce equal leakage. However, their technique requires that the traces can be executed in lockstep, so the control flow in both trace pairs must be the same. Therefore, it can not be applied to leakage models that allow different control flow.

Rosemann [2023] generalized constant-time simulations to hypersimulations. Hypersimulations can be used for a wider variety of hyperproperties, as they can deal with any number of traces, not just pairs. More importantly, however, they replace the lockstep execution of constant-time simulations with more general synchronizers, which means that they can be applied to traces with differing control flow. This makes hypersimulations applicable in more situations, in particular, for leakage models that do not include control flow.

1.2 Contributions

We propose using hypersimulations as a tool to verify spectre mitigations. Like Patrignani and Guarnieri [2021], we treat the mitigation as a compilation pass where the source and target languages have the same syntax, but different semantics: The source language does not have any kind of speculative execution, whereas the target language does.

First, we define speculative noninterference as a property of not just the compiled (mitigated) program, but of the source and compiled program together. For one, this is more accurate than the previous characterizations by Guarnieri et al. [2018] and Cheang et al. [2019]: For any characterizations which only consider the nonspeculative behaviour of a program after a mitigation has been applied, a program could be made to satisfy this definition by introducing more nonspeculative leaks instead of eliminating speculative leaks. More importantly, however, this allows us to characterize specula-

tive noninterference as the preservation of a hyperproperty, which allows us to apply hypersimulations.

We then demonstrate our approach by verifying that the mitigation proposed by Intel [Intel, 2018, 2021] produces speculative noninterferent programs not only for the constant-time leakage model (confirming the result of Patrignani and Guarnieri [2021]), but also for a leakage models that do not include all control flow, where the previous approach was not applicable.

We further demonstrate the benefit of working with hyperproperties directly by designing and verifying a mitigation that produces code satisfying speculative noninterference, but not speculative safety, and is thus not verifiable by Patrignani and Guarnieri [2021].

Finally, we demonstrate an issue with Speculative Load Hardening[Carruth, 2018] and always-mispredict semantics, which makes it unverifiable in leakage models that do not include all control flow.

1.3 Formal Verification in Coq

All proofs presented in this thesis have been formalized in and verified by the Coq proof assistant [The Coq Development Team, 2022]. The proof scripts are available at `https://compilers.cs.uni-saarland.de/projects/hypre-spectre/toc.html`.

To avoid obscuring the main proof ideas and results with details, we will not reproduce all proofs in their entirety in this thesis. Instead, we provide proof sketches that capture the main ideas, and refer to the Coq proof scripts for details.

Proofs and definitions will refer to the corresponding sections in the Coq development like this [Preserve_ct.v, `fence_preserve_ct`], indicating the file and name under which the proof or definition in question can be found. If reading this document digitally, they also link to the file online.

The Coq development also includes a work-in-progress version of the HyPre library (logical path `HyPre`) by Rosemann [2023], which is required for our proofs, but has not yet been published. The work for this thesis is found under the logical path `HypreSpectre` and consists of 1252 lines of specification and 2438 lines of proofs.

2 Regular and Speculative Semantics

2.1 Toy Language Syntax

We use a simple list-based imperative language $\mathcal{L}$, where a program is a list of statements.

We will be using common list notations, including [] for empty lists, :: for cons (appending a head to a list) and ++ for concatenating two lists. For programs specifically, however, we opt to use a semicolon instead of :: for readability reasons.

$$
\begin{array}{lll}
\mathit{Expr} \ni e ::= n & & n \in \mathbb{N} \\
\quad \mid x & & x \text{ is a variable name} \\
\quad \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 \times e_2 & & \\
\quad \mid e_1 < e_2 \mid e_1 = e_2 \mid !e & &
\end{array}
$$

$$
\begin{array}{l}
\mathit{Stmt} \ni s ::= \texttt{skip} \\
\quad \mid \texttt{fence} \\
\quad \mid x := e \\
\quad \mid \texttt{read}\ [e]\ x \\
\quad \mid \texttt{write}\ [e]\ x \\
\quad \mid \texttt{if}\ e\ \texttt{then}\ p_1\ \texttt{else}\ p_2\ \texttt{end} \\
\quad \mid \texttt{while}\ e\ \texttt{do}\ p_1\ \texttt{finally}\ p_2\ \texttt{end}
\end{array}
$$

$$
\mathit{Prog} \ni p ::= s; p \mid []
$$

Figure 2.1: Syntax of $\mathcal{L}$ [Lang.v, `stmt`]

The syntax of our toy language is shown in Figure 2.1. Note that memory accesses can not occur within expressions, but require their own instructions. This simplifies reasoning about leakage caused by memory accesses, as it means that the evaluation of expressions will not produce such leakage.

One particularity of our language is the inclusion of a `finally` clause in the `while` statement. There is no functional difference between code in the `finally` clause and code after the `while` statement, however, it does simplify the simulation relations for our mitigations (see for example Section 5.1).

2.2 Environments and Evaluation of Expressions

Our environments consist of a heap $\mathcal{H}$, which maps integers to integers, and a set of variable bindings $\mathcal{V}$ that maps variable names to integers. Both $\mathcal{V}$ and $\mathcal{H}$ initially map all variables and addresses to 0, unless otherwise specified. Thus, they are total. We use the notation $\mathcal{V}[x \mapsto v]$ to denote updating variable x to value v and similarly $\mathcal{H}[a \mapsto v]$ to denote updating address a.

When specifying heaps or variable mappings, we will give them as a list of assignments to nonzero values. This list may be empty if nothing is assigned a nonzero value, for example, the initial variable mapping is represented as [].

Since we only use integer values, $<, =$ and ! have integer results similar to how they behave in C. Other than that, all operations behave as expected.

Due to the omission of division, the evaluation of expressions is total. As it also does not produce leakage in the leakage models we consider, we can therefore use big-step semantics to evaluate expressions. We denote this evaluation by $[\![\cdot]\!]_{\mathcal{V}}$, where $\mathcal{V}$ is the set of variable bindings.

2.3 Nonspeculative Semantics

In the nonspeculative semantics, a program state $\langle \mathcal{V} \mid \mathcal{H} \mid p \rangle$ consists of a set of variable bindings $\mathcal{V}$, a heap $\mathcal{H}$ and a program p. p may be the empty program [], which signifies termination.

The nonspeculative semantics is shown in Figure 2.2. Note that our semantics reduces `while` statements to `if` statements, so that there is only one instruction where control flow can differ and speculative execution can start. Otherwise, many intermediate results would need to be proved twice, both for `if` and for `while`, instead of just once. Note also that in the nonspeculative semantics, the `fence` instruction behaves the same as `skip`.

One oddity of this semantics is that it repeats final states (states where the program is empty) infinitely. This choice was made because hypersimulations require infinite traces. However, this does not mean that information about termination is lost, it can be reintroduced via the leakage model (see Section 3.3).

2.4 Speculative Semantics

To model speculative execution, we use an always-mispredict semantics as described by Guarnieri et al. [2018], which captures all possible leakage by considering all paths

$$\frac{}{\langle \mathcal{V} \mid \mathcal{H} \mid \texttt{skip}; p\rangle \to_{\text{ns}} \langle \mathcal{V} \mid \mathcal{H} \mid p\rangle}\ \textsc{Skip} \qquad \frac{}{\langle \mathcal{V} \mid \mathcal{H} \mid \texttt{fence}; p\rangle \to_{\text{ns}} \langle \mathcal{V} \mid \mathcal{H} \mid p\rangle}\ \textsc{Fence}$$

$$\frac{[\![e]\!]_{\mathcal{V}} = v}{\langle \mathcal{V} \mid \mathcal{H} \mid x := e; p\rangle \to_{\text{ns}} \langle \mathcal{V}[x \mapsto v] \mid \mathcal{H} \mid p\rangle}\ \textsc{Assign}$$

$$\frac{[\![a]\!]_{\mathcal{V}} = a' \qquad \mathcal{H}(a') = v}{\langle \mathcal{V} \mid \mathcal{H} \mid \texttt{read}\ [a]\ x; p\rangle \to_{\text{ns}} \langle \mathcal{V}[x \mapsto v] \mid \mathcal{H} \mid p\rangle}\ \textsc{Read}$$

$$\frac{[\![a]\!]_{\mathcal{V}} = a' \qquad \mathcal{V}(x) = v}{\langle \mathcal{V} \mid \mathcal{H} \mid \texttt{write}\ [a]\ x; p\rangle \to_{\text{ns}} \langle \mathcal{V} \mid \mathcal{H}[a' \mapsto v] \mid p\rangle}\ \textsc{Write}$$

$$\frac{[\![b]\!]_{\mathcal{V}} \neq 0}{\langle \mathcal{V} \mid \mathcal{H} \mid \texttt{if}\ b\ \texttt{then}\ p_1\ \texttt{else}\ p_2\ \texttt{end}; p\rangle \to_{\text{ns}} \langle \mathcal{V} \mid \mathcal{H} \mid p_1 +\!\!+ p\rangle}\ \textsc{IfTrue}$$

$$\frac{[\![b]\!]_{\mathcal{V}} = 0}{\langle \mathcal{V} \mid \mathcal{H} \mid \texttt{if}\ b\ \texttt{then}\ p_1\ \texttt{else}\ p_2\ \texttt{end}; p\rangle \to_{\text{ns}} \langle \mathcal{V} \mid \mathcal{H} \mid p_2 +\!\!+ p\rangle}\ \textsc{IfFalse}$$

$$\frac{}{\begin{array}{c}\langle \mathcal{V} \mid \mathcal{H} \mid \texttt{while}\ b\ \texttt{do}\ p_1\ \texttt{finally}\ p_2\ \texttt{end}; p\rangle \\ \to_{\text{ns}} \langle \mathcal{V} \mid \mathcal{H} \mid \texttt{if}\ b\ \texttt{then}\ (p_1 +\!\!+ \texttt{while}\ b\ \texttt{do}\ p_1\ \texttt{finally}\ p_2\ \texttt{end})\ \texttt{else}\ p_2\ \texttt{end}; p\rangle\end{array}}\ \textsc{While}$$

$$\frac{}{\langle \mathcal{V} \mid \mathcal{H} \mid []\rangle \to_{\text{ns}} \langle \mathcal{V} \mid \mathcal{H} \mid []\rangle}\ \textsc{Term}$$

Figure 2.2: Nonspeculative semantics of $\mathcal{L}$ [Lang.v, `smallstep_nonspec`]

along which the processor may speculatively execute. To this end, whenever a branch instruction is encountered, the semantics first begins executing the incorrect branch speculatively for some number of steps until that execution is rolled back and execution continues along the correct branch. Importantly, in the case of nested speculation, only the innermost execution is rolled back. This way, the semantics also captures leakage that occurs when the prediction is incorrect for an outer, but correct for an inner branch.

The semantics operates on configurations that consist of a stack S of speculative states, where each state consists of a variable map $\mathcal{V}$, a heap $\mathcal{H}$ and a program p as well as a speculation window n. The speculation window limits for how many steps speculative execution can continue. Nonspeculative execution is not limited and does not have a speculation window, which we indicate with $\bot$.

One step of execution depends only on the topmost element of the stack, everything below remains unchanged. Most instructions execute according to their nonspeculative semantics, but decreasing the speculation window by one, with two exceptions: The `fence` instruction directly sets the speculation window to 0 and `if` b `then` c_1 `else` c_2 `end`

$$\text{decr}(n) := \begin{cases} \bot & \text{if } n = \bot \\ n-1 & \text{otherwise} \end{cases} \qquad \text{wndw}(n) := \begin{cases} \omega & \text{if } n = \bot \\ n-1 & \text{otherwise} \end{cases}$$

$$\text{no-rollback}(n) := n > 0 \vee n = \bot \qquad \text{zero-out}(n) := \begin{cases} \bot & \text{if } n = \bot \\ 0 & \text{otherwise} \end{cases}$$

$$\frac{\text{no-rollback}(n)}{\langle n \mid \mathcal{V} \mid \mathcal{H} \mid \texttt{skip}; p\rangle :: S \to_{\text{sp}} \langle \text{decr}(n) \mid \mathcal{V} \mid \mathcal{H} \mid p\rangle :: S} \text{ AMSkip}$$

$$\frac{\text{no-rollback}(n)}{\langle n \mid \mathcal{V} \mid \mathcal{H} \mid \texttt{fence}; p\rangle :: S \to_{\text{sp}} \langle \text{zero-out}(n) \mid \mathcal{V} \mid \mathcal{H} \mid p\rangle :: S} \text{ AMFence}$$

$$\frac{\text{no-rollback}(n) \qquad [\![e]\!]_{\mathcal{V}} = v}{\langle n \mid \mathcal{V} \mid \mathcal{H} \mid x := e; p\rangle :: S \to_{\text{sp}} \langle \text{decr}(n) \mid \mathcal{V}[x \mapsto v] \mid \mathcal{H} \mid p\rangle :: S} \text{ AMAssign}$$

$$\frac{\text{no-rollback}(n) \qquad [\![a]\!]_{\mathcal{V}} = a' \qquad \mathcal{H}(a') = v}{\langle n \mid \mathcal{V} \mid \mathcal{H} \mid \texttt{read } [a]\ x; p\rangle :: S \to_{\text{sp}} \langle \text{decr}(n) \mid \mathcal{V}[x \mapsto v] \mid \mathcal{H} \mid p\rangle :: S} \text{ AMRead}$$

$$\frac{\text{no-rollback}(n) \qquad [\![a]\!]_{\mathcal{V}} = a' \qquad \mathcal{V}(x) = v}{\langle n \mid \mathcal{V} \mid \mathcal{H} \mid \texttt{write } [a]\ x; p\rangle :: S \to_{\text{sp}} \langle \text{decr}(n) \mid \mathcal{V} \mid \mathcal{H}[a' \mapsto v] \mid p\rangle :: S} \text{ AMWrite}$$

$$\frac{\text{no-rollback}(n)}{\begin{array}{c}\langle n \mid \mathcal{V} \mid \mathcal{H} \mid \texttt{while } e \texttt{ do } p_1 \texttt{ finally } p_2 \texttt{ end}; \rangle :: S \\ \to_{\text{sp}} \langle \text{decr}(n) \mid \mathcal{V} \mid \mathcal{H} \mid \texttt{if } e \texttt{ then } (p_1 +\!\!+ [\texttt{while } e \texttt{ do } p_1 \texttt{ finally } p_2 \texttt{ end}]) \texttt{ else } p_2 \texttt{ end}; p\rangle \\ :: S\end{array}} \text{ AMWhile}$$

$$\frac{\text{no-rollback}(n) \qquad [\![b]\!]_{\mathcal{V}} \neq 0}{\begin{array}{c}\langle n \mid \mathcal{V} \mid \mathcal{H} \mid \texttt{if } b \texttt{ then } p_1 \texttt{ else } p_2 \texttt{ end}; p\rangle :: S \\ \to_{\text{sp}} \langle \text{wndw}(n) \mid \mathcal{V} \mid \mathcal{H} \mid p_2 +\!\!+ p\rangle ::\langle \text{decr}(n) \mid \mathcal{V} \mid \mathcal{H} \mid p_1 +\!\!+ p\rangle :: S\end{array}} \text{ AMIfTrue}$$

$$\frac{\text{no-rollback}(n) \qquad [\![b]\!]_{\mathcal{V}} = 0}{\begin{array}{c}\langle n \mid \mathcal{V} \mid \mathcal{H} \mid \texttt{if } b \texttt{ then } p_1 \texttt{ else } p_2 \texttt{ end}; p\rangle :: S \\ \to_{\text{sp}} \langle \text{wndw}(n) \mid \mathcal{V} \mid \mathcal{H} \mid p_1 +\!\!+ p\rangle ::\langle \text{decr}(n) \mid \mathcal{V} \mid \mathcal{H} \mid p_2 +\!\!+ p\rangle :: S\end{array}} \text{ AMIfFalse}$$

$$\frac{}{\langle 0 \mid \mathcal{V} \mid \mathcal{H} \mid p\rangle :: S \to_{\text{sp}} S} \text{ AMRollback} \qquad \frac{n \neq \bot \wedge n > 0}{\langle n \mid \mathcal{V} \mid \mathcal{H} \mid []\rangle :: S \to_{\text{sp}} S} \text{ AMRollbackT}$$

$$\frac{}{\langle \bot \mid \mathcal{V} \mid \mathcal{H} \mid []\rangle :: S \to_{\text{sp}} \langle \bot \mid \mathcal{V} \mid \mathcal{H} \mid []\rangle :: S} \text{ AMTerm} \qquad \frac{}{[] \to_{\text{sp}} []} \text{ AMTerm'}$$

Figure 2.3: Speculative semantics of $\mathcal{L}$ [Lang.v, `smallstep_spec_am`]

$$\frac{}{i \to^t []}\ \textsc{Empty} \qquad \frac{}{i \to^t [] \rhd i}\ \textsc{Init} \qquad \frac{i \to^t S \rhd s \qquad s \to t}{i \to^t S \rhd s \rhd t}\ \textsc{Step}$$

Figure 2.4: Rules for generating traces

starts misspeculation by pushing a state for the incorrect branch to the top and the correct branch below, both with the speculation window decreased by 1. Finally, speculative executions are rolled back once the speculation window reaches 0 or there is no code left to execute. This happens by simply removing them from the stack so that execution can continue with the state below. The complete set of rules can be seen in Figure 2.3.

The semantics as presented here is parametric in the speculation window ω, which is used to initialize speculative execution in case the outer execution is nonspeculative. In the accompanying Coq development as well as in all examples, we will use a speculation window of 16 for illustration purposes. In practice, however, a much larger window should be chosen depending on CPU architecture. A safe value would be twice the size of the reorder buffer [Guarnieri et al., 2018].

Similar to the nonspeculative semantics, final states – nonspeculative states with the empty program – are repeated indefinitely. In order to obtain a total semantics, we also treat the empty stack this way. Note however that the empty stack can not occur as long as we start all executions with nonspeculative states, as nonspeculative states can not be rolled back.

2.5 Initial States

For our purposes, we assume that all variables are initially zero. Input is passed to the program by storing it in the heap. Given a program p and and initial heap $\mathtt{i}$, the initial state for nonspeculative execution will be $\langle \mathtt{i} \rangle_{\text{ns}} := \langle [] \mid \mathtt{i} \mid p \rangle$ and the initial state for speculative execution will be $\langle \mathtt{i} \rangle_{\text{sp}} := \langle \bot \mid [] \mid \mathtt{i} \mid p \rangle :: []$. Note that we omit the source program p in the notation, because we never have more than one source program that would need to be distinguished.

2.6 Trace Prefixes

When using hypersimulations, we reason about infinite traces of programs, or, more precisely, finite prefixes thereof.

Since both our semantics are deterministic, we can identify an infinite trace with its initial state $\langle \mathtt{i} \rangle$. We can then generate all finite prefixes of this infinite trace inductively according to the rules presented in Figure 2.4. Note that $\rhd$ behaves like :: for traces,

except the order is reversed for readability reasons. This way, new states are appended to the right.

In this thesis, we will generally use the roman alphabet for traces in the nonspeculative semantics and the greek alphabet for traces in the speculative semantics. We will write $\langle \mathtt{i} \rangle_{\text{ns}} \rightarrow^{t}_{\text{ns}} a$ to denote that a is a valid trace prefix for input `i` according to the nonspeculative semantics, and similarly $\langle \mathtt{i} \rangle_{\text{sp}} \rightarrow^{t}_{\text{sp}} \alpha$ to denote that α is a valid trace prefix for input `i` according to the speculative semantics.

Trace prefixes can be concatenated with sequences of states if the resulting sequence of states respects the semantics. We will simply write ab for a trace prefix a concatenated with a sequence of states b.

3 Leakage Models

3.1 General Assumptions

In all our examples (and all leakage models we use), we treat the variables as not observable to an attacker. We split the heap into a public and a private part similar to Patrignani and Guarnieri [2021], with the private part in the negative addresses and the public part in the nonnegative addresses. This means that data from a positive address can safely be leaked as it is known to the attacker anyway, while data from a negative address should not be leaked. We will also use the heap for input, so secret inputs will be stored at negative addresses and public or attacker-controlled inputs at nonnegative addresses.

3.2 Low-Equivalence

In order to detect leakage of secrets, we want to compare executions of a program where all public (attacker-controlled and attacker-visible) data is the same, but secret data may be different. If the attacker can not detect a difference, then the secrets do not leak.

Since we use heaps as inputs, we thus define low-equivalence as a relation on heaps: Two heaps are low-equivalent, written $\sim_L$, if their public (low-security) parts, i.e. everything at a nonnegative address, are the same.

$$\mathcal{H}_1 \sim_L \mathcal{H}_2 \Leftrightarrow \forall n \in \mathbb{N}.\mathcal{H}_1(n) = \mathcal{H}_2(n)$$

3.3 Leakage Functions

In previous methods [Barthe et al., 2018, Patrignani and Guarnieri, 2021], the leakage was included in the semantics, which means that the semantics would have to be adjusted for every leakage model. This way, the executions can directly produce a sequence of observations without having to produce a complete trace.

Since hypersimulations require the complete traces anyway, we can instead model leakage as a function from a trace prefix to a list of observations. This makes the leakage model independent of the semantics, so we can reason about different models without any changes to the semantics.

We define our leakage functions on individual states, such that they always describe the leakage that will be produced by the next execution step. They will be lifted to

$$
\begin{aligned}
\ell_{\mathrm{ct}}(\langle \mathcal{V} \mid \mathcal{H} \mid \texttt{read}\ [a]\ x; p\rangle) &= \text{read } [\![a]\!]_{\mathcal{V}} \\
\ell_{\mathrm{ct}}(\langle \mathcal{V} \mid \mathcal{H} \mid \texttt{write}\ [a]\ x; p\rangle) &= \begin{cases} \text{write } \mathcal{V}(x) \text{ to } [\![a]\!]_{\mathcal{V}} & \text{if } [\![a]\!]_{\mathcal{V}} \geqslant 0 \\ \text{write } [\![a]\!]_{\mathcal{V}} & \text{otherwise} \end{cases} \\
\ell_{\mathrm{ct}}(\langle \mathcal{V} \mid \mathcal{H} \mid \texttt{if}\ b\ \texttt{then}\ p_1\ \texttt{else}\ p_2\ \texttt{end}; p\rangle) &= \begin{cases} \text{branch } \top & \text{if } [\![b]\!]_{\mathcal{V}} \neq 0 \\ \text{branch } \bot & \text{otherwise} \end{cases} \\
\ell_{\mathrm{ct}}(\langle \mathcal{V} \mid \mathcal{H} \mid [] \rangle) &= \text{end} \\
\ell_{\mathrm{ct}}(\langle \mathcal{V} \mid \mathcal{H} \mid p\rangle) &= \varepsilon \qquad \text{otherwise}
\end{aligned}
$$

$$
\begin{aligned}
\ell_{\mathrm{ct}}(\langle 0 \mid \mathcal{V} \mid \mathcal{H} \mid p\rangle) &= \varepsilon \\
\ell_{\mathrm{ct}}(\langle n \mid \mathcal{V} \mid \mathcal{H} \mid \texttt{read}\ [a]\ x; p\rangle) &= \text{read } [\![a]\!]_{\mathcal{V}} \\
\ell_{\mathrm{ct}}(\langle n \mid \mathcal{V} \mid \mathcal{H} \mid \texttt{write}\ [a]\ x; p\rangle) &= \begin{cases} \text{write } \mathcal{V}(x) \text{ to } [\![a]\!]_{\mathcal{V}} & \text{if } [\![a]\!]_{\mathcal{V}} \geqslant 0 \\ \text{write } [\![a]\!]_{\mathcal{V}} & \text{otherwise} \end{cases} \\
\ell_{\mathrm{ct}}(\langle n \mid \mathcal{V} \mid \mathcal{H} \mid \texttt{if}\ b\ \texttt{then}\ p_1\ \texttt{else}\ p_2\ \texttt{end}; p\rangle) &= \begin{cases} \text{branch } \top & \text{if } [\![b]\!]_{\mathcal{V}} \neq 0 \\ \text{branch } \bot & \text{otherwise} \end{cases} \\
\ell_{\mathrm{ct}}(\langle \bot \mid \mathcal{V} \mid \mathcal{H} \mid [] \rangle) &= \text{end} \\
\ell_{\mathrm{ct}}(\langle n \mid \mathcal{V} \mid \mathcal{H} \mid p\rangle) &= \varepsilon \qquad \text{otherwise}
\end{aligned}
$$

Figure 3.1: Leakage function ℓ_{ct} on speculative [Leak.v, `leak_ct'`] and nonspeculative [Leak.v, `leak_ct`] states

trace prefixes by concatenating the individual leakages. ε represents the empty word, i.e. the absence of an observation, and will disappear during concatenation.

In our examples, we will use three different leakage functions: ℓ_{ct} (Figure 3.1) corresponds to the constant-time leakage model. Therefore, it includes the outcome of branch instructions and accessed memory addresses. ℓ_{lm} (Figure 3.2), in contrast, still includes accessed memory addresses, but it replaces complete information about the control flow (branch outcomes) with only loop headers. Finally, ℓ_{mem} (Figure 3.3) only includes the accessed memory addresses.

As mentioned in Chapter 2, our traces repeat final states indefinitely. Thus, all our leakage functions also produce the end observation for final states[1] to signify that the execution has terminated.

Since we treat the public heap as visible throughout the entire execution, not just at the end, write observations to the public heap also include the written value, whereas those to the private heap only include the address. To model an attacker that can

[1] While this means that the end observation is also repeated indefinitely, this does not convey any more information than producing the observation just once or allowing traces to end, which would both require special handling with our technique.

$$
\begin{aligned}
\ell_{\mathrm{lm}}(\langle \mathcal{V} \mid \mathcal{H} \mid \texttt{read}\ [a]\ x; p\rangle) &= \text{read } [\![a]\!]_{\mathcal{V}} \\
\ell_{\mathrm{lm}}(\langle \mathcal{V} \mid \mathcal{H} \mid \texttt{write}\ [a]\ x; p\rangle) &= \begin{cases} \text{write } \mathcal{V}(x) \text{ to } [\![a]\!]_{\mathcal{V}} & \text{if } [\![a]\!]_{\mathcal{V}} \geqslant 0 \\ \text{write } [\![a]\!]_{\mathcal{V}} & \text{otherwise} \end{cases} \\
\ell_{\mathrm{lm}}(\langle \mathcal{V} \mid \mathcal{H} \mid \texttt{while}\ b\ \texttt{do}\ p_1\ \texttt{finally}\ p_2\ \texttt{end}; p\rangle) &= \text{loop} \\
\ell_{\mathrm{lm}}(\langle \mathcal{V} \mid \mathcal{H} \mid []\rangle) &= \text{end} \\
\ell_{\mathrm{lm}}(\langle \mathcal{V} \mid \mathcal{H} \mid p\rangle) &= \varepsilon \qquad \text{otherwise}
\end{aligned}
$$

$$
\begin{aligned}
\ell_{\mathrm{lm}}(\langle 0 \mid \mathcal{V} \mid \mathcal{H} \mid p\rangle) &= \varepsilon \\
\ell_{\mathrm{lm}}(\langle n \mid \mathcal{V} \mid \mathcal{H} \mid \texttt{read}\ [a]\ x; p\rangle) &= \text{read } [\![a]\!]_{\mathcal{V}} \\
\ell_{\mathrm{lm}}(\langle n \mid \mathcal{V} \mid \mathcal{H} \mid \texttt{write}\ [a]\ x; p\rangle) &= \begin{cases} \text{write } \mathcal{V}(x) \text{ to } [\![a]\!]_{\mathcal{V}} & \text{if } [\![a]\!]_{\mathcal{V}} \geqslant 0 \\ \text{write } [\![a]\!]_{\mathcal{V}} & \text{otherwise} \end{cases} \\
\ell_{\mathrm{lm}}(\langle n \mid \mathcal{V} \mid \mathcal{H} \mid \texttt{while}\ b\ \texttt{do}\ p_1\ \texttt{finally}\ p_2\ \texttt{end}; p\rangle) &= \text{loop} \\
\ell_{\mathrm{lm}}(\langle \bot \mid \mathcal{V} \mid \mathcal{H} \mid []\rangle) &= \text{end} \\
\ell_{\mathrm{lm}}(\langle n \mid \mathcal{V} \mid \mathcal{H} \mid p\rangle) &= \varepsilon \qquad \text{otherwise}
\end{aligned}
$$

Figure 3.2: Leakage function ℓ_{lm} on speculative [Leak.v, `leak_lm'`] and nonspeculative [Leak.v, `leak_lm`] states

$$
\begin{aligned}
\ell_{\mathrm{mem}}(\langle \mathcal{V} \mid \mathcal{H} \mid \texttt{read}\ [a]\ x; p\rangle) &= \text{read } [\![a]\!]_{\mathcal{V}} \\
\ell_{\mathrm{mem}}(\langle \mathcal{V} \mid \mathcal{H} \mid \texttt{write}\ [a]\ x; p\rangle) &= \begin{cases} \text{write } \mathcal{V}(x) \text{ to } [\![a]\!]_{\mathcal{V}} & \text{if } [\![a]\!]_{\mathcal{V}} \geqslant 0 \\ \text{write } [\![a]\!]_{\mathcal{V}} & \text{otherwise} \end{cases} \\
\ell_{\mathrm{mem}}(\langle \mathcal{V} \mid \mathcal{H} \mid []\rangle) &= \text{end} \\
\ell_{\mathrm{mem}}(\langle \mathcal{V} \mid \mathcal{H} \mid p\rangle) &= \varepsilon \qquad \text{otherwise}
\end{aligned}
$$

$$
\begin{aligned}
\ell_{\mathrm{mem}}(\langle 0 \mid \mathcal{V} \mid \mathcal{H} \mid p\rangle) &= \varepsilon \\
\ell_{\mathrm{mem}}(\langle n \mid \mathcal{V} \mid \mathcal{H} \mid \texttt{read}\ [a]\ x; p\rangle) &= \text{read } [\![a]\!]_{\mathcal{V}} \\
\ell_{\mathrm{mem}}(\langle n \mid \mathcal{V} \mid \mathcal{H} \mid \texttt{write}\ [a]\ x; p\rangle) &= \begin{cases} \text{write } \mathcal{V}(x) \text{ to } [\![a]\!]_{\mathcal{V}} & \text{if } [\![a]\!]_{\mathcal{V}} \geqslant 0 \\ \text{write } [\![a]\!]_{\mathcal{V}} & \text{otherwise} \end{cases} \\
\ell_{\mathrm{mem}}(\langle \bot \mid \mathcal{V} \mid \mathcal{H} \mid []\rangle) &= \text{end} \\
\ell_{\mathrm{mem}}(\langle n \mid \mathcal{V} \mid \mathcal{H} \mid p\rangle) &= \varepsilon \qquad \text{otherwise}
\end{aligned}
$$

Figure 3.3: Leakage function ℓ_{mem} on speculative [Leak.v, `leak_mem'`] and nonspeculative [Leak.v, `leak_mem`] states

only see the final state of the heap, one could alternatively only leak the addresses, and include the state of the public heap in the termination observation.

3.4 Leakage Equivalence

For finite trace prefixes, we can simply evaluate the leakage function to determine whether they produce the same leakage. Therefore, we first introduce the following two relations on trace prefixes, parametric in a leakage function ℓ:

$$\begin{aligned} a =_\ell b \quad &:= \quad \ell(a) = \ell(b) \\ a \lesssim_\ell b \quad &:= \quad \ell(a) \text{ is a prefix of } \ell(b) \text{ or vice versa} \end{aligned}$$

Infinite traces, however, can not be compared directly. We therefore define leakage equivalence of two traces as follows:

termination insensitive leakage equivalence Two traces t_1 and t_2 are termination insensitive leakage equivalent under a leakage model ℓ if, for any finite prefixes a_1 and a_2 such that $t_1 \rightarrow^t a_1$ and $t_2 \rightarrow^t a_2$, it holds that $a_1 \lesssim_\ell a_2$.

termination sensitive leakage equivalence Two traces t_1 and t_2 are termination sensitive leakage equivalent under a leakage model ℓ if they are termination insensitive leakage equivalent and, additionally, for all prefixes a_1 and a_2 such that $t_1 \rightarrow^t a_1$ and $t_2 \rightarrow^t a_2$, there exist sequences of states b_1 and b_2 such that $t_1 \rightarrow^t a_1 b_1, t_2 \rightarrow^t a_2 b_2$ and $a_1 b_1 =_\ell a_2 b_2$.

Intuitively, the first definition simply accounts for the fact that trace prefixes can correspond to any point during execution by only requiring that the leakage that has already been produced can not be different. Of course, since this must hold for all trace prefixes, this also means that future leakages will be the same.

However, there is one case in which this is not sufficient, which only occurs if the leakage model does not contain control flow. In that case, it is possible for a program to enter an infinite loop that does not produce any further leakage. The other trace could then produce arbitrary leakage and the traces would still be considered leakage equivalent. Therefore, we have the second definition, which additionally requires that we can extend the two trace prefixes such that they will produce the exact same leakage. Intuitively, this means that the trace that currently has less leakage will eventually "catch up".

To simplify working with termination sensitive leakage equivalence, we also define the following relation on trace prefixes [Prefix_sens.v, `ll_pre_sens`]:

$$a \lesssim^+_{\ell \mathtt{i}} b := a \lesssim_\ell b \land (\exists\, a'\, b'. \langle \mathtt{i}_a \rangle \rightarrow^t aa' \land \langle \mathtt{i}_b \rangle \rightarrow^t bb' \land aa' =_\ell bb')$$

This relation encapsulates all the additional requirements, such that the traces produced by the inputs $\mathtt{i}_a$ and $\mathtt{i}_b$ are terminations sensitive leakage equivalent if all finite trace prefixes a and b satisfy $a \lesssim^+_{\ell \mathtt{i}} b$. Note that this relation is additionally parametric in the initial program and input, which are needed to enforce that the traces are correct according to the semantics.

3.5 Noninterference and Speculative Noninterference

We can now define noninterference and speculative noninterference based on leakage equivalence:

noninterference A program is noninterferent if two traces are leakage equivalent whenever they start in low-equivalent states.

$$\mathtt{i}_1 \sim_L \mathtt{i}_2 \Rightarrow (\forall\, a\, b.\, \langle \mathtt{i}_1 \rangle \rightarrow^t a \wedge \langle \mathtt{i}_2 \rangle \rightarrow^t b \Rightarrow a \lesssim^+_{\ell \mathtt{i}} b)$$

speculative noninterference A pair of a source and a compiled program is speculatively noninterferent if leakage equivalence of two source traces implies leakage equivalence of target traces starting in the same environments.

$$\begin{aligned}\forall\, \mathtt{i}_1\, \mathtt{i}_2.\, (\forall\, a\, b.\, \langle \mathtt{i}_1 \rangle_{\mathrm{ns}} \rightarrow^t_{\mathrm{ns}} a \wedge \langle \mathtt{i}_2 \rangle_{\mathrm{ns}} \rightarrow^t_{\mathrm{ns}} b \Rightarrow a \lesssim^+_{\ell \mathtt{i}} b) \\ \Rightarrow (\forall\, \alpha\, \beta.\, \langle \mathtt{i}_1 \rangle_{\mathrm{sp}} \rightarrow^t_{\mathrm{sp}} \alpha \wedge \langle \mathtt{i}_2 \rangle_{\mathrm{sp}} \rightarrow^t_{\mathrm{sp}} \beta \Rightarrow \alpha \lesssim^+_{\ell \mathtt{i}} \beta)\end{aligned}$$

Both definitions can also be instantiated with termination insensitive leakage equivalence.

Note that Barthe et al. [2018] also distinguish between termination insensitive and termination sensitive noninterference, but their distinction is different. Our definition of termination sensitive noninterference can be seen as a generalization of theirs that does not require equal control flow. Their version of termination insensitive noninterference, however, only requires that the same observations are produced if both executions terminate, but imposes no restrictions otherwise. In contrast, our definition requires that as long as both traces do produce leakage, they must be the same, and only allows that one trace may produce more observations than the other.

3.6 Security of Mitigations

A mitigation can be considered secure if it always produces speculatively noninterferent programs. Speculative noninterference, as we define it, is a property of the source and compiled versions of a program and requires that leakage equivalence was preserved across compilation for the program in question. Thus, if we want to show that a mitigation always produces speculatively noninterferent programs, we need to prove that it preserves leakage equivalence.

4 Using Hypersimulations

4.1 Simulations

Hypersimulations require a simulation between the source and target program. We use manysteps simulations as defined by Barthe et al. [2018]. There is also a more general concept of general simulations, which allow that the target trace takes less steps than the source trace. General simulations can also be used with hypersimulations [Rosemann, 2023], but we omit them since manysteps simulations are enough for our purposes.

A simulation is a relation $\approx$ between states in the source semantics and states in the target semantics. It is defined with respect to a synchronizer function sync, which specifies how many steps in the target semantics need to be executed to match one step in the source semantics. Together, they must satisfy the following properties:

- For any state a, $\mathsf{sync}(a)$ must be greater than zero.
- For any input, the initial states in the source and target semantics must be related.
- For related states $a \approx \alpha$, the state a' obtained by executing one step of the source semantics on a and the state α' obtained by executing $\mathsf{sync}(a)$ steps of the target semantics on α must again be related ($a' \approx \alpha'$).

4.2 Hypersimulations

Hypersimulations extend simulations to tuples of traces by having multiple simulations in parallel. A hypersimulation thus depends on three relations: an equivalence relation $\equiv_S$ on trace prefixes in the source language, an equivalence relation $\equiv_C$ on trace prefixes in the target language, and a simulation relation $\approx$ between trace prefixes in the source and target language.

The defining property of a hypersimulation is that when starting with source trace prefixes a_1 and a_2 and target trace prefixes α_1 and α_2 such that $a_1 \equiv_S a_2$, $\alpha_1 \equiv_C \alpha_2$, $a_1 \approx \alpha_1$ and $a_2 \approx \alpha_2$, it must be possible to extend all trace prefixes according to the semantics and simulation relation such that both the source and target relation are satisfied once more[1]. A visualization of a hypersimulation can be seen in Figure 4.1 (red lines mark the proof obligations to show that $\equiv_S, \equiv_C$ and $\approx$ form a hypersimulation).

[1] We present only hypersimulations for 2-hyperproperties, with 2 source traces and 2 target traces. Hypersimulations can also be used for k-hyperproperties for any k, in which case they require k source traces and k target traces.

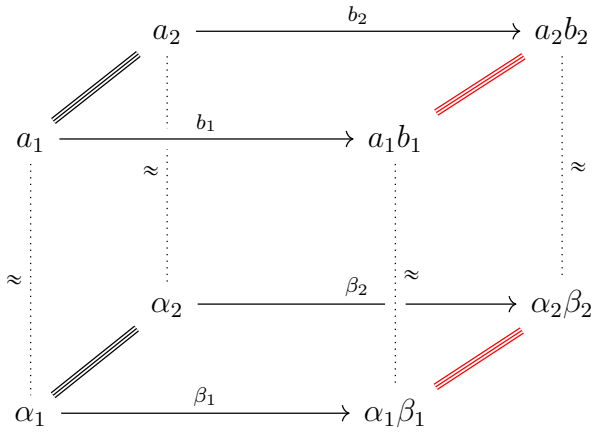


Figure 4.1: Diagram representing a hypersimulation

Since hypersimulations, unlike constant-time simulations, allow that the source traces have different control flow, it is not desirable to always execute the traces in lockstep (i.e. always the same number of steps). Instead, hypersimulations require some method of determining how many steps each trace should be executed for. The easiest method, which is sufficient in our case, is the use of a barrier predicate, where each trace will be executed until the next barrier (state for which the barrier predicate is true). This requires us to prove that the traces will still satisfy the relations at that point (see Figure 4.2c). We call one such step to the next barriers a hyperstep. However, this alone is not sufficient, as it leaves open what happens when no barrier is encountered. Thus, we have two options: We can either prove that it is always possible to reach another barrier (Figure 4.2a), or alternatively, that if one trace can not reach another barrier, the source and target relations will be satisfied for any future state (Figure 4.2b). Finally, all three relations must also be satisfied by the initial empty trace prefixes. We will not mention this explicitly in our examples, as it is trivial for the relations we use.

4.3 Proving Preservation of Leakage Equality using Hypersimulations

From a hypersimulation, we can only conclude that $\equiv_C$ will *always eventually* hold, i.e. there will always be some point in the future where it will be satisfied. However, to prove speculative noninterference, we need to show that $\lesssim_\ell$ (or $\lesssim^+_{\ell\mathtt{i}}$) *always* holds for target trace prefixes, i.e. it must hold at every point during execution.

As such, a hypersimulation does not directly prove the preservation of leakage equality. To show that $\lesssim_\ell$ holds on the target traces, we can use the fact that it is a safety hyperproperty [Rosemann, 2023], meaning that if it holds at any point during execution,

$$
\begin{aligned}
&\forall\, \mathtt{i}\, a.\, \langle \mathtt{i} \rangle_{\mathrm{ns}} \rightarrow^{t}_{\mathrm{ns}} a \\
\Rightarrow\ &\exists\, b\, b'\, x.\, b = b' \rhd x \\
&\wedge \langle \mathtt{i} \rangle_{\mathrm{ns}} \rightarrow^{t}_{\mathrm{ns}} ab \\
&\wedge \mathrm{B}(x)
\end{aligned}
$$

(a) Option 1: Traces always encounter barriers

$$
\begin{aligned}
&\forall\, i\, a_1\, a_2\, \alpha_1\, \alpha_2. \\
&\langle \mathtt{i}_1 \rangle_{\mathrm{ns}} \rightarrow^{t}_{\mathrm{ns}} a_1 \wedge \langle \mathtt{i}_2 \rangle_{\mathrm{ns}} \rightarrow^{t}_{\mathrm{ns}} a_2 \\
\Rightarrow\ & a_1 \equiv_S a_2 \\
\Rightarrow\ & (\forall\, b. \langle \mathtt{i}_i \rangle_{\mathrm{ns}} \rightarrow^{t}_{\mathrm{ns}} a_i b \Rightarrow (\forall n, \neg\, \mathrm{B}(b_n))) \\
\Rightarrow\ & \langle \mathtt{i}_1 \rangle_{\mathrm{sp}} \rightarrow^{t}_{\mathrm{sp}} \alpha_1 \wedge \langle \mathtt{i}_2 \rangle_{\mathrm{sp}} \rightarrow^{t}_{\mathrm{sp}} \alpha_2 \\
\Rightarrow\ & \alpha_1 \equiv_C \alpha_2 \\
\Rightarrow\ & \forall\, b_1\, b_2. \langle \mathtt{i}_1 \rangle_{\mathrm{ns}} \rightarrow^{t}_{\mathrm{ns}} a_1 b_1 \wedge \langle \mathtt{i}_2 \rangle_{\mathrm{ns}} \rightarrow^{t}_{\mathrm{ns}} a_2 b_2 \\
&\Rightarrow\ \forall\, \beta_1\, \beta_2. \langle \mathtt{i}_1 \rangle_{\mathrm{sp}} \rightarrow^{t}_{\mathrm{sp}} \alpha_1 \beta_1 \wedge \langle \mathtt{i}_2 \rangle_{\mathrm{sp}} \rightarrow^{t}_{\mathrm{sp}} \alpha_2 \beta_2 \\
&\quad\Rightarrow\ a_1 b_1 \approx \alpha_1 \beta_1 \wedge a_2 b_2 \approx \alpha_2 \beta_2 \\
&\qquad\Rightarrow\ a_1 b_1 \equiv_S a_2 b_2 \\
&\qquad\quad \wedge\, \alpha_1 \beta_1 \equiv_C \alpha_2 \beta_2
\end{aligned}
$$

(b) Option 2: If one trace will not encounter another barrier, the source and target relations will hold

$$
\begin{aligned}
&\forall\, a_1\, a_2\, \alpha_1\, \alpha_2. \\
&\langle \mathtt{i}_1 \rangle_{\mathrm{ns}} \rightarrow^{t}_{\mathrm{ns}} a_1 \wedge \langle \mathtt{i}_2 \rangle_{\mathrm{ns}} \rightarrow^{t}_{\mathrm{ns}} a_2 \\
\Rightarrow\ & a_1 \equiv_S a_2 \\
\Rightarrow\ & \langle \mathtt{i}_1 \rangle_{\mathrm{sp}} \rightarrow^{t}_{\mathrm{sp}} \alpha_1 \wedge \langle \mathtt{i}_2 \rangle_{\mathrm{sp}} \rightarrow^{t}_{\mathrm{sp}} \alpha_2 \\
\Rightarrow\ & \alpha_1 \equiv_C \alpha_2 \\
\Rightarrow\ & \forall\, b_1\, b_2. \langle \mathtt{i}_1 \rangle_{\mathrm{ns}} \rightarrow^{t}_{\mathrm{ns}} a_1 b_1 \wedge \langle \mathtt{i}_2 \rangle_{\mathrm{ns}} \rightarrow^{t}_{\mathrm{ns}} a_2 b_2 \\
&\Rightarrow\ \forall\, \beta_1\, \beta_2. \langle \mathtt{i}_1 \rangle_{\mathrm{sp}} \rightarrow^{t}_{\mathrm{sp}} \alpha_1 \beta_1 \wedge \langle \mathtt{i}_2 \rangle_{\mathrm{sp}} \rightarrow^{t}_{\mathrm{sp}} \alpha_2 \beta_2 \\
&\quad\Rightarrow\ a_1 b_1 \approx \alpha_1 \beta_1 \wedge a_2 b_2 \approx \alpha_2 \beta_2 \\
&\quad\Rightarrow\ \forall\, x_1\, x_2\, b_1'\, b_2'. b_1 = b_1' \rhd x_1 \wedge b_2 = b_2' \rhd x_2 \\
&\qquad\Rightarrow \mathrm{B}(x_1) \wedge \mathrm{B}(x_2) \\
&\qquad\Rightarrow (\forall n. \neg\, \mathrm{B}\,(b'_{1,n})) \wedge (\forall n. \neg\, \mathrm{B}\,(b'_{2,n})) \\
&\qquad\Rightarrow\ a_1 b_1 \equiv_S a_2 b_2 \\
&\qquad\quad \wedge\, \alpha_1 \beta_1 \equiv_C \alpha_2 \beta_2
\end{aligned}
$$

(c) The hyperstep predicate: The source and target relation are satisfied after one hyperstep to the next barriers

Figure 4.2: Predicates for hypersimulations using barrier synchronization. The hyperstep predicate is required for both options.

it must hold at all prior points[2]. If we use $=_\ell$ as the target relation in the hypersimulation, we can then use the fact that $=_\ell$ implies $\leqslant_\ell$ to conclude that $\leqslant_\ell$ must always hold at some point in the future. Therefore, since it is a safety hyperproperty, $\leqslant_\ell$ must always hold.

For termination sensitive speculative noninterference, we can use the same approach if we show that $\leqslant^+_{\ell\mathtt{i}}$ is a safety hyperproperty and that $=_\ell$ implies $\leqslant^+_{\ell\mathtt{i}}$.

Lemma 4.3.1 [Prefix_sens.v, `vl_ll_pre_sens_safety`]. *$\leqslant^+_{\ell\mathtt{i}}$ is a safety hyperproperty.*

Proof. Assume a, b, c and d such that b is a prefix of a, d is a prefix of c and $a \leqslant^+_{\ell\mathtt{i}} c$. We have to show $b \leqslant^+_{\ell\mathtt{i}} d$.

Since $\leqslant_\ell$ is a safety hyperproperty, we can already conclude $b \leqslant_\ell d$. It remains to show that $\exists\, b'\, d'. \langle \mathtt{i}_b \rangle \rightarrow^t bb' \wedge \langle \mathtt{i}_d \rangle \rightarrow^t dd' \wedge bb' =_\ell dd'$.

From $a \leqslant^+_{\ell\mathtt{i}} c$ we know that there are a' and c' such that $aa' =_\ell cc'$. Since b is a prefix of a, there is h_1 such that $a = bh_1$. Similarly, there is h_2 such that $c = dh_2$.

By picking $h_1 a'$ for b' and $h_2 c'$ for d', $bh_1 a' = aa' =_\ell cc' = dh_2 c'$ concludes the proof. □

Lemma 4.3.2 [Prefix_sens.v, `vl_ll_eq_ll_pre_sens`]. *For all $\mathtt{i}_a, \mathtt{i}_b, a$ and b with $\langle \mathtt{i}_a \rangle \rightarrow^t a$ and $\langle \mathtt{i}_b \rangle \rightarrow^t b$, $a =_\ell b \Rightarrow a \leqslant^+_{\ell\mathtt{i}} b$.*

Proof. We already know that $a \leqslant_\ell b$.
It remains to show that $\exists\, a'\, b'. \langle \mathtt{i}_a \rangle \rightarrow^t aa' \wedge \langle \mathtt{i}_b \rangle \rightarrow^t bb' \wedge aa' =_\ell bb'$.
We can pick [] for both a' and b', the rest follows by assumption. □

To prove that a mitigation is effective, we first need to define a corresponding simulation relation. Then, to show that the mitigation preserves leakage equivalence, we can choose a suitable source relation $\equiv_S$ and show that it forms a hypersimulation with $=_\ell$ as the target relation and the simulation relation.

[2] This definition is slightly different than the one by Clarkson and Schneider [2010], as we define it on finite trace prefixes, not infinite traces. However, it still captures the core idea that once a safety property is violated, it can not be recovered.

5 Eager Insertion of Fences is Secure

We will demonstrate the approach described in the previous chapter by showing that Intel's mitigation, the insertion of serializing fence instructions after every branch [Intel, 2018, 2021], is secure under all three leakage models presented in the previous section.

5.1 Modelling Intel's Mitigation

$$
\begin{aligned}
(\!|\texttt{if } b \texttt{ then } p_1 \texttt{ else } p_2 \texttt{ end}; p|\!)_{\text{fence}} &= \texttt{if } b \texttt{ then fence}; (\!|p_1|\!)_{\text{fence}} \\
&\quad\ \texttt{else fence}; (\!|p_2|\!)_{\text{fence}} \texttt{ end}; (\!|p|\!)_{\text{fence}} \\
(\!|\texttt{while } b \texttt{ do } p_1 \texttt{ finally } p_2 \texttt{ end}; p|\!)_{\text{fence}} &= \texttt{while } b \texttt{ do fence}; (\!|p_1|\!)_{\text{fence}} \\
&\quad\ \texttt{finally fence}; (\!|p_2|\!)_{\text{fence}} \texttt{ end}; (\!|p|\!)_{\text{fence}} \\
(\!|s; p|\!)_{\text{fence}} &= s; (\!|p|\!)_{\text{fence}} \qquad \text{otherwise} \\
(\!|[]|\!)_{\text{fence}} &= []
\end{aligned}
$$

Figure 5.1: Compilation function that inserts fences [Mitigation.v, `comp_fence`]

In order to reason about a mitigation, we first have to model it for our toy language. In this case, we can do so with a simple recursive function $(\!|\cdot|\!)_{\text{fence}}$ that appends a `fence` instruction in front of both cases of a branch, see Figure 5.1. The simulation relation $\approx_{\text{fence}}$ for this mitigation relates any source state $\langle \mathcal{V} \mid \mathcal{H} \mid p \rangle$ to the compiled program in the same environment: $\langle \bot \mid \mathcal{V} \mid \mathcal{H} \mid (\!|p|\!)_{\text{fence}} \rangle :: []$.

We can also define a synchronizer function $\mathsf{sync}_{\text{fence}}$ that, given a source state, determines how many steps the compiled program needs to take to match the next step of the source program. Whenever the source program executes a step that is not an `if` statement, the compiled program needs to execute one step to match. For `if` statements, the compiled program needs four steps: It starts speculating, reaches a `fence`, rolls back the execution and needs to execute another `fence` instruction.

We can then show that $\approx_{\text{fence}}$ is indeed a simulation relation w.r.t. $\mathsf{sync}_{\text{fence}}$:

$$\mathsf{sync}_{\text{fence}}(\langle \mathcal{V} \mid \mathcal{H} \mid \texttt{if } b \texttt{ then } p_1 \texttt{ else } p_2 \texttt{ end}; p\rangle) = 4$$
$$\mathsf{sync}_{\text{fence}}(\langle \mathcal{V} \mid \mathcal{H} \mid p\rangle) = 1 \qquad \text{otherwise}$$

Figure 5.2: Synchronizer for $(\!|\cdot|\!)_{\text{fence}}$ [Mitigation.v, `cf_sim_step`]

Lemma 5.1.1 [Mitigation.v, `cf_sim_sync_correct`]. *$\approx_{\text{fence}}$ is a simulation relation w.r.t. $\mathsf{sync}_{\text{fence}}$.*

Proof. We need to show that if we start with states $a \approx_{\text{fence}} \alpha$ and execute one step on a and $\mathsf{sync}_{\text{fence}}(a)$ steps on α, we end up in states a' and α' with $a' \approx_{\text{fence}} \alpha'$.

Note that for $(\!|\cdot|\!)_{\text{fence}}$, the following holds: $(\!|p_1 +\!\!+ p_2|\!)_{\text{fence}} = (\!|p_1|\!)_{\text{fence}} +\!\!+ (\!|p_2|\!)_{\text{fence}}$. With this, the proof follows by case distinction on the program and computing the appropriate number of steps. □

Note that, as mentioned in Section 2.1, this simulation relation would not work without the `finally` clause for `while` statements. The `finally` clause allows us to insert fences at its beginning (see Figure 5.1), rather than after the `while` statement. Without it, applying $(\!|\cdot|\!)_{\text{fence}}$ to a `while` instruction and then taking a step in the speculative semantics would yield a syntactically different program[1] than first taking a step in the nonspeculative semantics and then applying $(\!|\cdot|\!)_{\text{fence}}$.

We also use $\mathsf{sync}_{\text{fence}}$ to lift $\approx_{\text{fence}}$ to trace prefixes: The initial states must be related according to $\approx_{\text{fence}}$. Following that, the next pair of states indicated by $\mathsf{sync}_{\text{fence}}$ must again be related. This repeats until the end of the trace and must account for all states, i.e. the last states of the two traces must be related. We will use the same notation, $\approx_{\text{fence}}$, for the relation lifted to traces.

Lemma 5.1.2 [Preserve_ct.v, `sim_same_leakage`]. *Executing one step in the source program and the corresponding number of steps in the target program produces the same leakage under ℓ_{ct}, ℓ_{lm} [Preserve_lm.v, `sim_same_leakage`] and ℓ_{mem} [Preserve_mem.v, `sim_same_leakage`].*

Proof. By case distinction on the current statement in the source program. For an `if` statement, the leakage is the same and the following `fence` instructions and rollback do not produce any further leakage. For all other statements, the same statement will be executed in the target program, yielding the same leakage. □

[1] Consider a program `while` c `do` b `end`; [] that would compile to `while` c `do fence`; b `end`; `fence`; []. After one step of execution, this would become `if` c `then fence`; $b +\!\!+$ [`while` c `do fence`; b `end`] `else` [] `end`; `fence`; []. However, executing one step nonspeculatively first and then applying the mitigation would instead yield `if` c `then fence`; $b +\!\!+$ [`while` c `do fence`; b `end`] `else fence`; [] `end`; [].

5.2 Speculative Noninterference under Constant-Time Leakage Model

To prove that $(\!|\cdot|\!)_{\text{fence}}$ produces speculative noninterferent programs, i.e. that it preserves termination sensitive leakage equivalence, we use the method described in Section 4.3.

For the constant-time leakage model, this means that we need a hypersimulation with $=_{\ell_{\text{ct}}}$ as the target relation to conclude that $\lesssim^{+}_{\ell_{\text{ct}}\,\mathtt{i}}$ holds for all target trace prefixes. Concretely, we will also be using $=_{\ell_{\text{ct}}}$ as the source relation, so we need to show that $(=_{\ell_{\text{ct}}}, =_{\ell_{\text{ct}}}, \approx_{\text{fence}})$ forms a hypersimulation. To this end, we use the predicate described in Figure 4.2a.

This requires us to define a barrier predicate to determine for how many steps the source execution continues. We define the barrier predicate B_{ct} that is true for `if`, `read` and `write` statements as well as final states.

Lemma 5.2.1 [Preserve_ct.v, `barrier_leak`]. *The barrier predicate* B_{ct} *is true for a state iff that state produces leakage under* ℓ_{ct}*, i.e. adding this state to a trace yields a longer sequence of observations.*

Furthermore, we need to show that every source trace will always encounter another barrier:

Lemma 5.2.2 [Preserve_ct.v, `src_term`]. *For any trace prefix* $i \to^{t}_{\text{ns}} a$*, there is a sequence of states* $bs \rhd b$ *such that* $i \to^{t}_{\text{ns}} a(bs \rhd b)$ *and* b *satisfies* B_{ct}*.*

Proof. We can define a recursive function that calculates how many steps it will take to reach a barrier. This is guaranteed to terminate, as the program gets smaller at every step – in the case of a `while` statement, we know that the next step will execute an `if` statement, so we do not need recursion.

Once we have determined the number of steps until the next barrier, we can execute that number of steps in the semantics to obtain the sequence $bs \rhd b$. □

Finally, it remains to show that the source and target relations are still satisfied after one hyperstep:

Theorem 5.2.3 [Preserve_ct.v, `fence_preserve_ct`]. $(\!|\cdot|\!)_{\text{fence}}$ *produces programs that satisfy termination sensitive speculative noninterference w.r.t.* ℓ_{ct}*.*

Proof. We use the predicate from Figure 4.2a with B_{ct} to show that $=_{\ell_{\text{ct}}}, =_{\ell_{\text{ct}}}$ and $\approx_{\text{fence}}$ form a hypersimulation. We know from Lemma 5.2.2 that it is always possible to reach another barrier. Thus, it remains to show that $=_{\ell_{\text{ct}}}$ is still satisfied after one hyperstep:

- We have that a_1 and a_2 as well as α_1 and α_2 have the same leakage, and that b_1 and b_2 are sequences of states where the barrier predicate is true only for the last state. By Lemma 5.2.1, we thus know that $\ell_{\text{ct}}(a_1 b_1)$ and $\ell_{\text{ct}}(a_2 b_2)$ have the same length. Since we assume leakage equivalence on the source traces, this implies $\ell_{\text{ct}}(a_1 b_1) = \ell_{\text{ct}}(a_2 b_2)$.

- From Lemma 5.1.2, we know that β_1 and β_2 must produce the same leakage as b_1 and b_2. Thus, we can conclude that $\ell_{\text{ct}}(\alpha_1\beta_1) = \ell_{\text{ct}}(\alpha_2\beta_2)$.

We conclude that the target traces are leakage equivalent as described in Section 4.3. □

5.3 Speculative Noninterference under ℓ_{lm}

In the previous proof, the source traces are actually executed in lockstep, as the constant-time leakage model forces the control flow to be the same. This was actually a requirement of both the approach by Patrignani and Guarnieri [2021] as well as the constant-time simulations by Barthe et al. [2018]. Hypersimulations, however, have no such restriction, so we can also evaluate $(\!|\cdot|\!)_{\text{fence}}$ under leakage models that do not restrict control flow.

As a first step, consider the ℓ_{lm} leakage model which does not include all branches, but instead only loops. This way, the control flow may be different between the two source traces, but their termination behaviour must still be the same. We can therefore again use the predicate shown in Figure 4.2a and the proof will not differ much from the proof for the constant-time leakage model.

Since we are working with ℓ_{lm}, we will use $=_{\ell_{\text{lm}}}$ as both the source and target relation. We start by defining the barrier predicate B_{lm}, which is true for `while` , `read` and `write` instructions as well as final states.

Lemma 5.3.1 [Preserve_ct.v, `barrier_leak`]. B_{lm} *is true for a state iff that state produces leakage under* ℓ_{lm}.

Then, similar to Lemma 5.2.2, we need to show that the source traces will always encounter another barrier:

Lemma 5.3.2 [Preserve_ct.v, `src_term`]. *For any trace prefix* $i \to^t_{\text{ns}} a$*, there is a sequence of states* $bs \rhd b$ *such that* $i \to^t_{\text{ns}} a(bs \rhd b)$ *and* b *satisfies* B_{lm}.

Proof. Since loops produce observations, a program can not diverge without reaching a barrier. Thus, we can again define a recursive function that calculates how many steps it will take to reach a barrier. This is guaranteed to terminate, as the `while` statement is a base case.

Once we have determined the number of steps until the next barrier, we can execute that number of steps in the semantics to obtain the sequence $bs \rhd b$. □

Theorem 5.3.3 [Preserve_ct.v, `fence_preserve_lm`]. $(\!|\cdot|\!)_{\text{fence}}$ *produces programs that satisfy termination sensitive speculative noninterference w.r.t.* ℓ_{lm}.

Proof. We use the predicate from Figure 4.2a and Lemma 5.3.2 to show that $=_{\ell_{\text{lm}}}, =_{\ell_{\text{lm}}}$ and $\approx_{\text{fence}}$ form a hypersimulation.

It remains to show that the source and target traces still satisfy ℓ_{lm} after one hyperstep. The proof is analogous to that of Theorem 5.2.3, using B_{lm} and Lemma 5.3.1 instead of Lemma 5.2.1. □

5.4 Speculative Noninterference under ℓ_{mem}

We can also go further and not include any control flow in the leakage model: ℓ_{mem} only includes memory accesses and termination.

As the source and target relation, we pick $=_{\ell_{\text{mem}}}$, allowing us to conclude that $\lesssim^{+}_{\ell_{\text{mem}}\mathtt{i}}$ is always satisfied for the target traces. The barrier predicate B_{mem} will be true for memory accesses and final states.

Lemma 5.4.1 [Preserve_mem.v, `barrier_leak`]. B_{mem} *is true for a state iff that state produces leakage under* ℓ_{mem}.

With this model, we can not use the same version of the predicate as before, as it is possible for traces to diverge without producing any leakage. Therefore, we need to use the version of the predicate shown in Figure 4.2b.

Lemma 5.4.2. *Assuming termination sensitive leakage equivalence under* ℓ_{mem} *and two trace prefixes* $a_1 =_{\ell_{\text{mem}}} a_2$*, if one trace will not reach another barrier according to* B_{mem}*, neither trace will produce any further leakage (inlined in [Preserve_mem.v,* ***mem_hypersim****]).*

Proof. Since we assume termination sensitive leakage equivalence, we have $a_1 \lesssim^{+}_{\ell_{\text{mem}}\mathtt{i}} a_2$. From this, we know that no matter how we extend the two trace prefixes, we can always extend them further such that the new prefixes produce the same leakage. If one trace can not reach another barrier, then by Lemma 5.4.1, we know that its extension does not contain any leakage-producing states. Thus, the extension has not produced any further leakage.

Since the leakage of the extended prefixes must be the same and the leakage of the original trace prefixes was the same, we conclude that the extension of the other trace has also not produced any further leakage. This concludes the proof, as it holds for all possible extensions of the trace prefixes. □

Lemma 5.4.3. *If* $a_1 =_{\ell_{\text{mem}}} a_2$ *and both traces will not produce any further leakage, then for all* b_1, b_2 *with* $i_1 \rightarrow^{t}_{\text{ns}} a_1 b_1, i_2 \rightarrow^{t}_{\text{ns}} a_2 b_2$*, we have* $a_1 b_1 =_{\ell_{\text{mem}}} a_2 b_2$ *(inlined in [Preserve_mem.v,* ***mem_hypersim****]).*

Proof. Follows directly since neither b_1 nor b_2 can produce any leakage. Thus, $\ell_{\text{ct}}(a_1 b_1) = \ell_{\text{ct}}(a_1) = \ell_{\text{ct}}(a_2) = \ell_{\text{ct}}(a_2 b_2)$. □

To complete the proof, we again need to prove that $=_{\ell_{\text{mem}}}$ is still satisfied after a hyperstep.

Theorem 5.4.4 [Preserve_mem.v, `fence_preserve_mem`]. $(\!|\cdot|\!)_{\text{fence}}$ *produces programs that satisfy termination sensitive speculative noninterference w.r.t.* ℓ_{mem}.

Proof. We use the predicate described in Figure 4.2b with B_{mem} to show that $=_{\ell_{\text{mem}}}$, $=_{\ell_{\text{mem}}}$ and $\approx_{\text{fence}}$ form a hypersimulation.

Lemma 5.4.2 and Lemma 5.4.3 show that if we have related source trace prefixes and at least one of them will not encounter another barrier, then the source relation will be

satisfied at all future points. It follows from Lemma 5.1.2 that the same must then be true for the target traces.

Thus, it remains to show that $=_{\ell_{\text{mem}}}$ is still satisfied after a hyperstep, the proof of which is analogous to that in Theorem 5.2.3. □

6 Relaxed Insertion of Fences is also Secure

In the previous chapter, we have shown that our approach can be applied to a wider variety of leakage models than the approach by Patrignani and Guarnieri [2021]. However, their approach is also limited in a different way: Since speculative safety is an overapproximation of speculative noninterference, there are programs which are speculatively noninterferent, but do not satisfy speculative safety. They demonstrate this with an example program that has two branches, which both leak the same attacker-controlled address. We have constructed a similar example in our language in Figure 6.1.

$$\texttt{read}\ [1]\ x; \texttt{if}\ x \geqslant 0\ \texttt{then read}\ [x]\ y\ \texttt{else read}\ [x]\ y\ \texttt{end}; \texttt{write}\ [1]\ y$$

Figure 6.1: Program that satisfies speculative noninterference, but not speculative safety

The program takes a memory address as input from the attacker and performs a check which triggers speculative execution, but it reads from that address and returns the value independent of the result. The taint tracking used for speculative safety taints the value in variable y as unsafe, as it was read speculatively from an attacker-controlled address. Speculative safety is thus violated because an unsafe value is leaked during speculative execution (when it is written to address 1).

However, this program is speculatively noninterferent without any mitigation applied: The same observation that is produced during speculative execution is also produced during nonspeculative execution. Therefore, if two traces can be distinguished, they can also be distinguished without speculative execution.

As our approach does not use such an overapproximation, it can handle these cases. To demonstrate this, we designed a relaxed mitigation that is still based on the insertion of `fence` instructions, but can omit them if both branches produce the exact same leakage. We will then prove that this mitigation produces speculatively noninterferent programs in the constant-time leakage model.

6.1 Relaxed Insertion of Fences

To keep our mitigation simple, we rely purely on syntactical analysis. For every `if` statement, we check whether both branches have the exact same sequence of `read` and `write` statements within the speculation window. However, we also need to rule out

$$
\begin{aligned}
\mathsf{leak\text{-}lookahead}(0, p) &= [] \\
\mathsf{leak\text{-}lookahead}(n, []) &= \bot \\
\mathsf{leak\text{-}lookahead}(n, \texttt{fence}; p) &= \bot \\
\mathsf{leak\text{-}lookahead}(n, \texttt{if } c \texttt{ then } p_1 \texttt{ else } p_2 \texttt{ end}; p) &= \bot \\
\mathsf{leak\text{-}lookahead}(n, \texttt{while } c \texttt{ do } p_1 \texttt{ finally } p_1 \texttt{ end}; p) &= \bot \\
\mathsf{leak\text{-}lookahead}(n, x := e; p) &= \bot \\
\mathsf{leak\text{-}lookahead}(n, \texttt{read } [a]\ x; p) &= \text{read } x \text{ from } a \\
&\quad :: \mathsf{leak\text{-}lookahead}(n-1, p) \\
\mathsf{leak\text{-}lookahead}(n, \texttt{write } [a]\ x; p) &= \text{write } x \text{ to } a \\
&\quad :: \mathsf{leak\text{-}lookahead}(n-1, p) \\
\mathsf{leak\text{-}lookahead}(n, \texttt{skip}; p) &= \mathsf{leak\text{-}lookahead}(n-1, p)
\end{aligned}
$$

Figure 6.2: The leak-lookahead function [Mitigation.v, `leak_lookahead`]

any changes to the environment, so if we encounter any `assign` instructions, we still insert a `fence`. We also rule out nested `if` statements, as these would require the analysis to be much more complex: inner branches would have to be checked for different speculation window sizes, and rollbacks of inner nested speculative executions make the general behaviour much more complex.

To keep the proofs simple, we also rule out `fence` instructions in the source program as well as branches that end before the speculation window does. This way, we only need to consider the two cases where speculative execution is either rolled back immediately or will proceed to the end of the speculation window. Furthermore, if we did not insert `fence` instructions for branches that end before the speculation window, $(\!|p_1 +\!\!+ p_2|\!)_{\text{relaxed}} = (\!|p_1|\!)_{\text{relaxed}} +\!\!+ (\!|p_2|\!)_{\text{relaxed}}$ would not hold.

For the syntactical analysis, we use the function leak-lookahead (Figure 6.2). This function returns either a list of observations that will be produced during speculative execution or $\bot$ if it encounters any statements that force us to insert a `fence`. Note that while we use the same notation, the observations produced here are not the same as those produced by our leakage models: While the leakage models yield the actual values at runtime, leak-lookahead yields expressions that will be compared syntactically.

The simulation relation once again relates a program in an environment to the compiled program in the same environment: $\langle \mathcal{V} \mid \mathcal{H} \mid p \rangle \approx_{\text{relaxed}} \langle \bot \mid \mathcal{V} \mid \mathcal{H} \mid (\!|p|\!)_{\text{relaxed}} \rangle :: []$.

The synchronizer function $\mathsf{sync}_{\text{relaxed}}$ for this simulation is shown in Figure 6.4. To determine the number of steps needed for speculative execution in the case of `if` statements, the function performs the same checks as during compilation to determine whether `fence` instructions have been inserted. It then returns either 4 for an immediate rollback, or 18 for the execution of the entire speculation window and subsequent rollback.

$$(\!|\texttt{if } c \texttt{ then } p_1 \texttt{ else } p_2 \texttt{ end}; p|\!)_{\text{relaxed}}$$
$$= \begin{cases} \texttt{if } c \texttt{ then } (\!|p_1|\!)_{\text{relaxed}} \texttt{ else } (\!|p_2|\!)_{\text{relaxed}} \texttt{ end}; (\!|p|\!)_{\text{relaxed}} \\ \qquad \text{if } \mathsf{leak\text{-}lookahead}(16, p_1) = \mathsf{leak\text{-}lookahead}(16, p_2) \neq \bot \\ \texttt{if } c \texttt{ then fence}; (\!|p_1|\!)_{\text{relaxed}} \texttt{ else fence}; (\!|p_2|\!)_{\text{relaxed}} \texttt{ end}; (\!|p|\!)_{\text{relaxed}} \\ \qquad \text{otherwise} \end{cases}$$

$$(\!|\texttt{while } c \texttt{ do } p_1 \texttt{ finally } p_2 \texttt{ end}; p|\!)_{\text{relaxed}}$$
$$= \begin{cases} \texttt{while } c \texttt{ do } (\!|p_1|\!)_{\text{relaxed}} \texttt{ finally } (\!|p_2|\!)_{\text{relaxed}} \texttt{ end}; (\!|p|\!)_{\text{relaxed}} \\ \qquad \text{if } \mathsf{leak\text{-}lookahead}(16, p_1) = \mathsf{leak\text{-}lookahead}(16, p_2) \neq \bot \\ \texttt{while } c \texttt{ do fence}; (\!|p_1|\!)_{\text{relaxed}} \texttt{ finally fence}; (\!|p_2|\!)_{\text{relaxed}} \texttt{ end}; (\!|p|\!)_{\text{relaxed}} \\ \qquad \text{otherwise} \end{cases}$$

$$(\!|s; p|\!)_{\text{relaxed}} = s; (\!|p|\!)_{\text{relaxed}} \qquad \text{otherwise}$$
$$(\!|[]|\!)_{\text{relaxed}} = []$$

Figure 6.3: Compilation function implementing our relaxed mitigation [Mitigation.v, `comp_relaxed`]

$$\mathsf{sync}_{\text{relaxed}}(\langle \mathcal{V} \mid \mathcal{H} \mid \texttt{if } b \texttt{ then } p_1 \texttt{ else } p_2 \texttt{ end}; p \rangle) = \begin{cases} 18 & \text{if } \mathsf{leak\text{-}lookahead}(p_1) \\ & \quad = \mathsf{leak\text{-}lookahead}(p_2) \neq \bot \\ 4 & \text{otherwise} \end{cases}$$
$$\mathsf{sync}_{\text{relaxed}}(\langle \mathcal{V} \mid \mathcal{H} \mid p \rangle) = 1 \qquad \text{otherwise}$$

Figure 6.4: Synchronizer function for $(\!|\cdot|\!)_{\text{relaxed}}$ [Mitigation.v, `cr_sim_step`]

To prove that $\approx_{\text{relaxed}}$ is indeed a simulation relation w.r.t. $\mathsf{sync}_{\text{relaxed}}$, we need three facts:

Lemma 6.1.1 [**Mitigation.v, `comp_relaxed_app`**].
$(\!|a +\!\!+ b|\!)_{\text{relaxed}} = (\!|a|\!)_{\text{relaxed}} +\!\!+ (\!|b|\!)_{\text{relaxed}}$

Proof. By induction. Note that this only works because we place `fence` instructions for branches that are shorter than the speculation window, so that we do not need to consider the code after an `if` statement. □

Lemma 6.1.2 [**Mitigation.v, `exec_no_nested_spec`**]. *If* $\mathsf{leak\text{-}lookahead}(n, p) \neq \bot$, *then executing* n *steps starting in* $\langle n \mid \mathcal{V} \mid \mathcal{H} \mid p\rangle :: s :: []$ *will result in the state* $s :: []$.

Proof. By induction. $\mathsf{leak\text{-}lookahead}(n, p) \neq \bot$ allows us to rule out nested speculation as well as `fence` instructions, which would increase or decrease the number of steps needed to reach $s :: []$. □

Lemma 6.1.3 [**Mitigation.v, `leak_extend`**].
$\mathsf{leak\text{-}lookahead}(n, p) = \mathsf{leak\text{-}lookahead}(n, p +\!\!+ [\texttt{while } c \texttt{ do } p_1 \texttt{ finally } p_2 \texttt{ end}])$

Proof. Either, $\mathsf{leak\text{-}lookahead}(n, p) = \bot$ or $\mathsf{leak\text{-}lookahead}(n, p) = l$ for some l. We prove each case via induction on p. □

Lemma 6.1.4 [**Mitigation.v, `cr_sim_sync_correct`**]. $\approx_{\text{relaxed}}$ *is a simulation relation w.r.t.* $\mathsf{sync}_{\text{relaxed}}$.

Proof. We need to prove that if we start with states $a \approx_{\text{relaxed}} \alpha$ and execute one step on a and $\mathsf{sync}_{\text{relaxed}}(a)$ steps on α, we end up in states a' and α' with $a' \approx_{\text{relaxed}} \alpha'$.

We show this by case distinction on the next instruction. For most statements, this follows by Lemma 6.1.1. For `while` statements, we additionally need Lemma 6.1.3. For `if` statements, we have two options: If `fence` instructions were inserted, we can calculate the next 4 steps immediately. If no fences were inserted, we apply Lemma 6.1.2. □

6.2 Speculative Noninterference under ℓ_{ct}

To prove that $(\!|\cdot|\!)_{\text{relaxed}}$ preserves leakage equivalence, we again use the method described in Section 4.3. We will use a hypersimulation using $\approx_{\text{relaxed}}$ and $=_{\ell_{\text{ct}}}$ as the target relation, however, our source relation will additionally require that the last states of each trace have the same program. This is, of course, guaranteed by the constant-time leakage model as the traces execute in lockstep, however, we previously did not require this explicitly. We will call this relation $\equiv_{\text{rel}}$.

Also, we will not be using B_{ct} as before. Instead, we will use a barrier predicate that is always true, meaning we will examine each step individually.

As every step is a barrier, we can use the predicate from Figure 4.2a: Proving that every trace prefix can be extended to another barrier is trivial, as it only needs to be extended by a single step.

We will start by proving that a single step in the nonspeculative semantics preserves $\equiv_{\text{rel}}$:

Lemma 6.2.1 [**Preserve_ct_relaxed.v, `srcrel_step`**]. *For leakage equivalent traces* $\langle \mathtt{i}_1 \rangle_{\text{ns}}$ *and* $\langle \mathtt{i}_2 \rangle_{\text{ns}}$*, all trace prefixes* $\langle \mathtt{i}_1 \rangle_{\text{ns}} \rightarrow^t_{\text{ns}} a_1 \rhd s_1$ *and* $\langle \mathtt{i}_1 \rangle_{\text{ns}} \rightarrow^t_{\text{ns}} a_2 \rhd s_2$ *with* $a_1 \rhd s_1 \equiv_{\text{rel}} a_2 \rhd s_2$*, and all states* s_1', s_2' *such that* $\langle \mathtt{i}_1 \rangle_{\text{ns}} \rightarrow^t_{\text{ns}} a_1 \rhd s_1 \rhd s_2'$ *and* $\langle \mathtt{i}_2 \rangle_{\text{ns}} \rightarrow^t_{\text{ns}} a_2 \rhd s_2 \rhd s_2'$*, it must hold that* $a_1 \rhd s_1 \rhd s_1' \equiv_{\text{rel}} a_2 \rhd s_2 \rhd s_2'$.

Proof. We need to show that the new trace prefixes produce the same leakage, and that the program is still equal in both traces.

- We can show that the programs in s_1' and s_2' are still equal by case distinction on the program in s_1, which is equal to that in s_2 by assumption. In all cases but the `if` statement, the program of the following state is already uniquely determined by the current program.

  For the `if` statement, we use the fact that the leakage of $a_1 \rhd s_1$ and $a_2 \rhd s_2$ includes which branch will be taken. As their leakage is the same by assumption, we know that the same branch is taken. Thus, s_1' and s_2' will have the same program.

- Like in previous proofs, we will use that we have assumed leakage equivalence of the source traces. Thus, we only need to show that the two trace prefixes produce the same amount of leakage.

  By assumption, we know that $a_1 \rhd s_1$ and $a_2 \rhd s_2$ have the same (and therefore same amount of) leakage. Thus, we only need to show that s_1' and s_2' either both produce leakage, or both do not. This follows from the fact that they have the same program, as there are no instructions that only sometimes produce an observation.

□

We can now go on to proving that after the corresponding number of steps (as indicated by $\mathsf{sync}_{\text{relaxed}}$), the target trace prefixes still satisfy $=_{\ell_{\text{ct}}}$. Unlike the proofs presented previously, we can not conclude this directly from the source trace prefixes, as source and target trace prefixes do not produce the same leakages in this example.

Instead, we use the fact that the trace prefixes before the current step satisfy $=_{\ell_{\text{ct}}}$, and prove that that the leakage produced by this step is the same for both trace prefixes.

This relies on the following lemma:

Lemma 6.2.2 [**Preserve_ct_relaxed.v, `leak_lookahead_same_leakage`**]. *If* $\mathsf{leak\text{-}lookahead}(16, p_1) = \mathsf{leak\text{-}lookahead}(16, p_2) \neq \bot$*, then executing 16 steps of* $(\!|p_1|\!)_{\text{relaxed}}$ *speculatively produces the same leakage as executing 15 steps of* p_2 *nonspeculatively.*

Note that 16 steps of nonspeculative execution correspond to only 15 steps of nonspeculative execution as the 16th step is a rollback, which has no corresponding step in the nonspeculative execution.

Proof. For the proof, we generalize the first parameter of $\mathsf{leak\text{-}lookahead}$ and prove that if $\mathsf{leak\text{-}lookahead}(a, p_1) = \mathsf{leak\text{-}lookahead}(b+1, p_2) \neq \bot$, executing a steps of $(\!|p_1|\!)_{\text{relaxed}}$ speculatively produces the same leakage as executing b steps of p_2 nonspeculatively. We show this by induction on a and b:

- If a is 0, then $\mathsf{leak\text{-}lookahead}(0, p_1) = []$. By induction on b, we find that the first $b + 1$ instructions of p_2 must all be `skip`, as no other sequence of instructions would satisfy $\mathsf{leak\text{-}lookahead}(b + 1, p_2) = []$. No leakage is produced.

- Otherwise, if b is 0, then $\mathsf{leak\text{-}lookahead}(1, p_2)$ is either $[]$ or contains exactly one observation.
  - If $\mathsf{leak\text{-}lookahead}(1, p_2)$ is $[]$, we know (as before) that both p_1 and p_2 only execute `skip` instructions and no leakage will be produced.
  - Otherwise, we consider the first statement of p_1. If it produces no observation, we apply the inductive hypothesis for a. If it produces the same observation as p_2, we can show by induction that the remaining steps executed in p_1 are all `skip` instructions and do not produce further leakage. If it produces any other observation, we have a contradiction.

- If both a and b are greater than 0, we check the first statements of p_1 and p_2:
  - If the first statement of p_1 does not produce any leakage, we apply the inductive hypothesis for a for the remainder of p_1.
  - Otherwise, if the first statement of p_2 does not produce any leakage, we apply the inductive hypothesis for b.
  - If both first statements produce the same leakage, we apply the induction hypothesis for a for the remainder of p_1 and p_2. If they produce different leakage, we have a contradiction.

□

With this, we can now prove that the target trace prefixes also satisfy $=_{\ell_{\mathrm{ct}}}$.

Lemma 6.2.3 [Preserve_ct_relaxed.v, `srcrel_impl_tgtrel`]. *If $\equiv_{\mathrm{rel}}$ is preserved for one step of the nonspeculative traces, then $=_{\ell_{\mathrm{ct}}}$ is preserved after the corresponding number of steps (according to $\mathsf{sync}_{\mathrm{relaxed}}$) of the speculative traces.*

Proof. We know that the traces satisfied $=_{\ell_{\mathrm{ct}}}$ before, so we only need to look at the newly added states. For most instructions, $\mathsf{sync}_{\mathrm{relaxed}}$ is 1, so we only have one state to consider. That state will produce the exact same leakage as the corresponding source state, so we can use the fact that the source trace prefixes satisfy $=_{\ell_{\mathrm{ct}}}$.

For `if` statements, there are two options: Either, there is a `fence` at the beginning of both branches and 4 steps are taken, or there is no `fence` and 18 steps (the execution of the `if` instruction plus 16 steps of speculative execution and the rollback) are taken. In the first case, we can calculate these 4 steps, of which only the first will produce leakage (fences and rollbacks do not). In the second case, we apply Lemma 6.2.2. □

Theorem 6.2.4 [Preserve_ct_relaxed.v, `relaxed_preserve_ct`]. *$(\!|\cdot|\!)_{\mathrm{relaxed}}$ produces programs that satisfy termination sensitive speculative noninterference w.r.t. ℓ_{ct}.*

Proof. We use the predicate from Figure 4.2a. As every step is a barrier, we know that every trace prefix will always reach another barrier in just one step.

We need to show that $\equiv_{\text{rel}}$, $=_{\ell_{\text{ct}}}$ and $\approx_{\text{relaxed}}$ form a hypersimulation. This requires us to show that $\equiv_{\text{rel}}$ and $=_{\ell_{\text{ct}}}$ are preserved, which follows from Lemma 6.2.1 and Lemma 6.2.3.

Thus, we conclude that the target traces are leakage equivalent under ℓ_{ct} as described in Section 4.3. □

6.3 Further Relaxations and Applications

As presented here, this mitigation is rather restricted. However, with further work, some of these restrictions could be lifted: Manually inserted fences could be handled better if a more general version of Lemma 6.2.2 was shown. This would, for example, allow manually placing fences as late as possible instead of at the beginning of a branch. It should also be possible to lift the restrictions on short branches, leading to improvements in more situations.

Additionally, one could look at optimizing certain cases. For example, while we need to restrict changes to the environment to ensure equal expressions will yield equal addresses, we could allow changes that are not followed by leakage-producing instructions within the speculation window. It might also be possible to incorporate results from other techniques, such as alias analysis for equality of addresses, to further reduce the number of `fence` instructions inserted.

Apart from being used on its own, such a mitigation could also be very useful in conjunction with other strategies for inserting fences. This mitigation does not place fences in ideal locations on its own, but it can verify (and fix) fences inserted previously by the developer or some other automated method. As a result, more efficient mitigations could easily be created and verified without further proof work by combining an unverified strategy for inserting fences with a subsequent pass of this mitigation.

7 Speculative Load Hardening is not Secure

We have shown that we can verify the mitigation proposed by Intel [2018], the insertion of fence instructions, even when we remove control flow from the leakage model. In this chapter, we demonstrate why the same does not work for another popular mitigation, Speculative Load Hardening [Carruth, 2018].

7.1 Modelling Speculative Load Hardening

Speculative Load Hardening works by introducing additional data dependencies between branch conditions and memory addresses. This is achieved by repeating the same calculation as for the branch condition when calculating the address for a memory access.

Since existing CPUs predict the outcome of the branch, not the outcome of the calculation, the CPU may still start speculating; however, it can not execute the memory access without fetching all the data for the branch condition to be evaluated properly. This should be enough to trigger a rollback before the memory access happens, but if it is not, the incorrect branch outcome in the calculation of the memory address will yield some safe address that does not contain any secrets.

In $\mathcal{L}$, we can model Speculative Load Hardening with a special variable `_spec`. This variable will initially be 0, but it will be updated after every branch so that it will only be 0 if the branch was taken correctly. This means that this variable will remain 0 throughout the nonspeculative execution, however, it will be 1 during speculative execution. We can now protect all memory accesses by multiplying them with the negation of this variable. During nonspeculative execution, this will be a multiplication by 1 yielding the original address. However, during speculative execution, this will always yield 0, which is part of the public heap and can safely be leaked.

The corresponding compilation function can be seen in Figure 7.1.

7.2 Leaking Branch Outcomes through "Safe" Memory Accesses

The issue lies in the supposedly safe memory accesses that occur if the execution is not rolled back in time. While the data accessed is known to the attacker, if we consider a leakage model that does not include branch outcomes, it is possible to leak those via

$$
\begin{aligned}
\mathrm{protect}(a) &= a \cdot !\texttt{\_spec} \\
\mathrm{upd}(e) &= \texttt{\_spec} := \texttt{\_spec} + !e \\
\\
(\!|\texttt{read}\ [a]\ x; p|\!)_{\mathrm{SLH}} &= \texttt{read}\ [\mathrm{protect(a)}]\ x; (\!|p|\!)_{\mathrm{SLH}} \\
(\!|\texttt{write}\ [a]\ x; p|\!)_{\mathrm{SLH}} &= \texttt{write}\ [\mathrm{protect}(a)]\ x; (\!|p|\!)_{\mathrm{SLH}} \\
(\!|\texttt{if}\ b\ \texttt{then}\ p_1\ \texttt{else}\ p_2\ \texttt{end}; p|\!)_{\mathrm{SLH}} &= \texttt{if}\ b\ \texttt{then}\ \mathrm{upd}(b); (\!|p_1|\!)_{\mathrm{SLH}} \\
&\quad\ \texttt{else}\ \mathrm{upd}(!b); (\!|p_2|\!)_{\mathrm{SLH}}\ \texttt{end}; (\!|p|\!)_{\mathrm{fence}} \\
(\!|\texttt{while}\ b\ \texttt{do}\ p_1\ \texttt{finally}\ p_2\ \texttt{end}; p|\!)_{\mathrm{SLH}} &= \texttt{while}\ b\ \texttt{do}\ \mathrm{upd}(b); (\!|p_1|\!)_{\mathrm{SLH}} \\
&\quad\ \texttt{finally}\ \mathrm{upd}(!b); (\!|p_2|\!)_{\mathrm{SLH}}\ \texttt{end}; (\!|p|\!)_{\mathrm{SLH}} \\
\\
(\!|s; p|\!)_{\mathrm{SLH}} &= s; (\!|p|\!)_{\mathrm{SLH}} \qquad \text{otherwise}
\end{aligned}
$$

Figure 7.1: Speculative Load Hardening for $\mathcal{L}$ [SLH.v, `speculative_load_hardening`]

the number of memory accesses. Consider Figure 7.2, where a program performs the same memory access in both branches, but at different points. One of these lies within the speculation window, whereas the other does not. Thus, when the attacker observes a memory access at location 0 before the access at the proper address, they know which branch was taken.

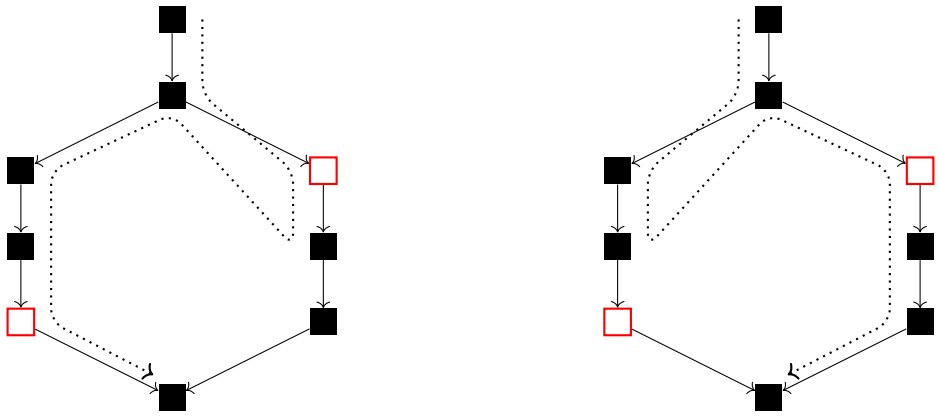
An example program is shown in Figure 7.3. Here, the attacker is able to determine whether the value stored at address -1 in the private heap is 0.

7.3 Implications of this Result

This counterexample does not necessarily demonstrate an issue with Speculative Load Hardening. As mentioned in Section 7.1, the expectation is that a rollback will be triggered before the memory access occurs. However, always-mispredict semantics [Guarnieri et al., 2018] like we use here deliberately do not accurately model the CPU and instead assume the worst possible behaviour.

It is therefore likely that the leak presented here would not occur on real hardware, meaning that Speculative Load Hardening would in fact be secure even if the leakage model does not include control flow. Proving so, however, would require semantics that more accurately model the hardware in question, which would require that hardware manufacturers guarantee certain behaviours. A promising approach in this direction are hardware-software contracts as proposed by Guarnieri et al. [2021].

On the other hand, this example illustrates that developers working with constant-time leakage models should adhere to them strictly. It might be tempting to look at leakage that would occur through control flow, like the secret value in x in our example, and decide to ignore it, thinking it would be too hard to exploit it in practice.



(a) A memory access may be performed during speculative execution.

(b) No memory access is encountered during speculative execution.

Figure 7.2: Leaking control flow through the number of memory accesses. The dotted line represents the flow of the execution, including misspeculation. On the left hand side, a memory access is performed during speculative execution, producing an additional observation. This allows an attacker to deduce that the left branch was taken, as this can not occur if the right branch is taken.

However, our counterexample shows that such a leak could become easier to exploit using speculative execution: in this case, if the attacker can not observe the control flow directly, they can still infer which branch was taken via the memory accesses performed. A mitigation that was not designed with this aspect in mind might no longer offer sufficient protection in this case. We therefore recommend that developers adhere strictly to the leakage models for which the mitigation they are using was designed.

```
p_SLH := read [−1] x;
         if x
         then
             read [1] y;
             skip;
             skip;
             skip
         else
             skip;
             skip;
             skip;
             skip;
             skip;
             skip;
             skip;
             skip;
             skip;
             skip;
             skip;
             skip;
             skip;
             skip;
             skip;
             skip;
             skip;
             read [1] y
         end
```

(a) Sample program

$$\mathcal{H}_1 := [-1 \mapsto 0] \qquad \mathcal{H}_2 := [-1 \mapsto 1]$$

(b) Low-equivalent input heaps

Let a_1, a_2 be sufficiently long traces

$$\langle [] \mid \mathcal{H}_1 \mid p_{SLH} \rangle \rightarrow^{t}_{\text{ns}} a_1, \quad \langle [] \mid \mathcal{H}_2 \mid p_{SLH} \rangle \rightarrow^{t}_{\text{ns}} a_2.$$

$$\ell_{\text{ct}}(a_1) = \text{read } -1 \cdot \text{read } 1 \cdot \text{end} \cdots$$

$$\ell_{\text{ct}}(a_2) = \text{read } -1 \cdot \text{read } 1 \cdot \text{end} \cdots$$

(c) Nonspeculative leakage

Let α_1, α_2 be sufficiently long traces

$$\langle [] \mid \mathcal{H}_1 \mid p_{SLH} \rangle \rightarrow^{t}_{\text{sp}} \alpha_1, \quad \langle [] \mid \mathcal{H}_2 \mid p_{SLH} \rangle \rightarrow^{t}_{\text{sp}} \alpha_2.$$

$$\ell_{\text{ct}}(\alpha_1) = \text{read } -1 \cdot \text{read } 0 \cdot \text{read } 1 \cdot \text{end} \cdots$$

$$\ell_{\text{ct}}(\alpha_2) = \text{read } -1 \cdot \text{read } 1 \cdot \text{end} \cdots$$

Figure 7.3: Sample program and two low-equivalent input heaps. As shown on the right, the same sequence of observations is produced by nonspeculative execution, but speculative execution produces different traces [SLH.v, `SLH_not_preserve_lm`].

8 Conclusion

In this thesis, we have demonstrated that hypersimulations can be used to verify compiler-based Spectre countermeasures. We have demonstrated that this approach is more powerful than the previous method presented by Patrignani and Guarnieri [2021] in several ways:

In Chapter 5, we have shown how hypersimulations can be used with leakage models that do not include control flow by verifying Intel's mitigation for three leakage models. We have first confirmed previous results that the mitigation is secure under the widely used constant-time leakage model. Then, we have loosened the restriction on control flow by including only loop headers and not all branches in the leakage model. This no longer enforced equal control flow, but it did still enforce that every trace prefix can always be extended to produce more leakage. We were therefore able to prove that Intel's mitigation is also secure under this leakage model with only minimal adjustments to the proof, whereas previous methods were not applicable to this leakage model at all. Finally, we considered a leakage model that did not include any information on control flow. This did require us to use a different method of showing the hypersimulation, however, it was still not much more complex than the proof for the constant-time leakage model.

In Chapter 6, we have designed and verified a mitigation that was not verifiable with the previous approach. The main idea of the mitigation is that fences do not need to be inserted if the leakage will be the same on both branches. This means that the mitigated code does not necessarily satisfy speculative safety as defined by Patrignani and Guarnieri [2021], however, we have shown that it is speculatively noninterferent for the constant-time leakage model.

In Chapter 7, we demonstrated with a counterexample that we can not verify Speculative Load Hardening for leakage models that do not include control flow. With Speculative Load Hardening, memory accesses during speculative execution still produce observations, which may allow an attacker to recover information on the control flow. However, we expect that these observations are the result of always-mispredict semantics overapproximating possible leakages, so Speculative Load Hardening could be verified using semantics that model the CPU more accurately based on vendor guarantees.

8.1 Further Work

We have only demonstrated the approach with a very minimalistic language. Further work is needed to apply this approach to more realistic languages. In particular, while

modelling programs as lists of statements keeps proofs simple, a program-counter based model with jump and branch instructions would be much more realistic.

While using specifically designed programming languages is already useful for verifying whether mitigations conceptually work, the goal should eventually be to verify implementations in real compilers. The most likely candidate for this would be CompCert[Leroy, 2009], a verified C compiler which has formally defined semantics for all intermediate steps, and simulations between them. To our knowledge, CompCert does not currently have any Spectre mitigations. However, with some work having been done to verify that it preserves constant-time leakage equivalence [Barthe et al., 2018], a verified mitigation seems to be a reasonable addition.

One aspect of the work of Patrignani and Guarnieri [2021] that we have not considered here is the ability to model control flow to and from attacker-provided code. In their approach, the component being protected by the mitigation can both call attacker-defined functions and have functions that can be called by the attacker. This would be useful to protect libraries that might be linked against malicious code. Reasoning directly with hyperproperties instead of approximating with regular properties will likely introduce additional complexity here, however, it should in principle be possible assuming that the attacker behaves the same on traces it can not distinguish.

Another possibility is the use of different semantics for the target language. Our approach does not require the use of an always-mispredict semantics, rather, any deterministic semantics can be used. While always-mispredict semantics can be used to verify mitigations across a wide variety of hardware, modelling specific hardware more accurately, for example based on hardware-software contracts [Guarnieri et al., 2021], would also be useful and could allow more precise mitigations. On the other hand, the always-mispredict semantics could also be extended to cover more sources of speculation, and therefore more Spectre variants. Fabian et al. [2022] have shown how to define such semantics in a modular way.

Finally, the mitigation designed in Chapter 6 is very limited and, in its current state, only serves as a proof of concept. Several potential improvements have already been described in that chapter, however, they would be most useful if implemented for a more realistic programming language. As mentioned before, a less restrictive version of this mitigation could also be very useful as a second pass after some other strategy of inserting fences, which then would not need to be verified.

Bibliography

Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. Secure compilation of side-channel countermeasures: The case of cryptographic "constant-time". In *31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018*, pages 328–343. IEEE Computer Society, 2018. doi: 10.1109/CSF.2018.00031. URL https://doi.org/10.1109/CSF.2018.00031.

Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. In Nadia Heninger and Patrick Traynor, editors, *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, pages 249–266. USENIX Association, 2019. URL https://www.usenix.org/conference/usenixsecurity19/presentation/canella.

Chandler Carruth. Speculative Load Hardening, 2018. URL https://llvm.org/docs/SpeculativeLoadHardening.html.

Kevin Cheang, Cameron Rasmussen, Sanjit A. Seshia, and Pramod Subramanyan. A formal approach to secure speculation. In *32nd IEEE Computer Security Foundations Symposium, CSF 2019, Hoboken, NJ, USA, June 25-28, 2019*, pages 288–303. IEEE, 2019. doi: 10.1109/CSF.2019.00027. URL https://doi.org/10.1109/CSF.2019.00027.

Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *J. Comput. Secur.*, 18(6): 1157–1210, 2010. doi: 10.3233/JCS-2009-0393. URL https://doi.org/10.3233/JCS-2009-0393.

Xaver Fabian, Marco Guarnieri, and Marco Patrignani. Automatic detection of speculative execution combinations. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pages 965–978. ACM, 2022. doi: 10.1145/3548606.3560555. URL https://doi.org/10.1145/3548606.3560555.

J. A. Goguen and J. Meseguer. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy*, pages 11–11, 1982. doi: 10.1109/SP.1982.10014.

Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. SPECTECTOR: principled detection of speculative information flows. *CoRR*, abs/1812.08639, 2018. URL `http://arxiv.org/abs/1812.08639`.

Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. Hardware-software contracts for secure speculation. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*, pages 1868–1883. IEEE, 2021. doi: 10.1109/SP40001.2021.00036. URL `https://doi.org/10.1109/SP40001.2021.00036`.

Intel. Using Intel compilers to mitigate speculative execution side-channel issues, 2018. URL `https://www.intel.com/content/www/us/en/developer/articles/troubleshooting/using-intel-compilers-to-mitigate-speculative-execution-side-channel-issues.html`.

Intel. Intel C++ compiler classic developer guide and reference, 2021. URL `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`.

Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 1–19. IEEE, 2019. doi: 10.1109/SP.2019.00002. URL `https://doi.org/10.1109/SP.2019.00002`.

Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009. doi: 10.1145/1538788.1538814. URL `https://doi.org/10.1145/1538788.1538814`.

Andrew Pardoe. Spectre mitigations in MSVC, 2018. URL `https://devblogs.microsoft.com/cppblog/spectre-mitigations-in-msvc/`.

Marco Patrignani and Marco Guarnieri. Exorcising spectres with secure compilers. In Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi, editors, *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, pages 445–461. ACM, 2021. doi: 10.1145/3460120.3484534. URL `https://doi.org/10.1145/3460120.3484534`.

Julian Rosemann. Private communications, not yet released. All relevant proofs are included in the Coq development, 2023.

The Coq Development Team. The Coq proof assistant, September 2022. URL `https://coq.inria.fr`.

Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. oo7: Low-overhead defense against spectre attacks via binary analysis. *CoRR*, abs/1807.05843, 2018. URL `http://arxiv.org/abs/1807.05843`.