

DRAFT: Compiler Correctness and Spectre Mitigations

by

Julay Leatherman-Brooks

A thesis submitted in partial fulfillment of the
requirements for the degree of

Master of Science
in
Computer Science

Thesis Committee:
Andrew P. Tolmach, Chair
Cătălin Hrițcu
Mark P. Jones
Yao Li

Portland State University
2027

Abstract

Modern processors use speculative execution to optimize performance in the case of pipeline stalls. Speculative side-channel attacks leverage this feature to break the isolation between processes and exfiltrate sensitive information, even from programs verified not to leak in the absence of speculation. SpecIBT is a combined hardware and software defense against Spectre PHT and BTB attacks whose novel contribution is the precise detection of misspeculation on indirect calls.

Spectre mitigations are complex and easy to get wrong; the use of formal methods increases confidence in their correctness. SpecIBT has been formally verified with mechanized proofs in the Rocq Prover to satisfy *relative security*: the target program in the presence of speculation does not leak more than the source program in its absence. A key component of the relative security proof is a backward compiler correctness lemma stating that the concrete implementation of SpecIBT is correct with respect to its abstract specification. The proof of this lemma was my contribution to this work, and the topic of this thesis.

For my grandma, who was denied an education, and my mom, who has made mine possible.

Acknowledgments

I'm very grateful to the many people who helped and supported me in bringing this work to completion. Too many to name, but even if your name isn't listed here, please know that I remember and appreciate your support.

To Andrew Tolmach, my advisor: thank you for consistently going above and beyond as a mentor, generously sharing your time, expertise, and experience with me, and believing in me before I was able to believe in myself. Your support helped me grow both professionally and personally. I wish you a joyful retirement with lots of music.

To Cătălin Hrițcu: thank you for giving me the opportunity to intern with your Formally Verified Security group at MPI-SP, a life-changing experience without which this thesis would not exist. I am so grateful to have had the chance to learn from your expertise. Thank you for taking a chance on me.

I am grateful to all the wonderful people at MPI-SP who made me feel welcome. Thanks are especially due to current and former members of the FOVSEC group: Rob Blanco encouraged me to apply, Jonathan Baumann and Yonghyun Kim showed up for me every single day as mentors and friends, Jérémy Thibault was always supportive and took time out from his final day in Bochum to have an in-depth conversation with me about future plans, and Cezar Andrici and Abigail Pribisova offered warmth, friendship, and curiosity.

To Yao Li and Mark Jones: thank you both for your close reading of my thesis and suggestions on how to improve my writing. I look forward to applying these in the future. I am also deeply grateful for the mentorship and advice you've both offered over the past few years, which has helped me through challenging times.

To my SIGPLAN mentor, Patrick Lam: thank you for your insight and practical wisdom,

and especially for helping me with time management as the defense deadline drew near.

Thank you to Allison Naaktgeboren, Kevin Brosnan, Katie Casamento, Katherine Philip, Ian Kariniemi, Laura Israel, Nicholas Coltharp, Steven Libby, Grant VanDomelen, Beatrice Mitchell, Gatlin Newhouse, Chris Lu, and Gary Klimowicz, for advice, solidarity, and community. In particular, Allison and Katherine were instrumental in informing me of the existence of the MPI-SP internship and who to talk to about it.

Thank you to my dear friends John Gravois, Rose Dakin, Christina Osborn, Collin Stackhouse, Tom Diamant, Richard Melling, Karen McCracken, Joe Pomianek, Brooke Parker, Joel Brown, Devin Forest-Hines, Joel Cofield, and Jim Harris, for being there for me in so many ways. I love you all.

Thank you especially to my mom, who has generously and joyfully supported me both financially and emotionally through this degree that has taken so much longer than I expected it to. I truly could not have done this without you.

Contents

1	Introduction	1
1.1	Background	1
1.2	My Contribution	4
1.3	Roadmap	5
2	Speculative Attacks and Defenses	6
2.1	Attacks	6
2.2	Defenses	7
2.2.1	USLH and Spectre PHT	7
2.2.2	Intel CET and Spectre BTB	10
2.2.3	Threat Model and Security Definition	13
3	The MiniMIR Language	15
3.1	Syntax	15
3.2	Language Level	16
3.3	Well-formedness	17
3.4	Undefined Values	18
4	The SpecIBT transformation	19
5	Semantics	21
5.1	Sequential Semantics	21
5.2	Speculative Semantics	24
5.3	Ideal Semantics	26
6	Relative Security for SpecIBT	29
6.1	Relative Security	31
7	Backward Compiler Correctness	33
7.1	Compiler Correctness	33
7.1.1	Simulations	33
7.2	The Proof	37
7.2.1	Lemma 1	37
7.2.2	Induction proof	39
7.2.3	Single cycle proof	44
8	Related Work	55
9	Conclusion	56
10	References	57
A	Supplemental File Information	59

List of Tables

1	Branch subcases	48
2	Target single cycle, attacker takes branch, already misspeculating	48
3	Target single cycle, call, first two steps	50
4	Target single cycle, call, third and fourth steps	50
5	Source call	52

List of Figures

1	Language syntax (adapted from [1])	15
2	SpecIBT transformation (adapted from [1])	19
3	Evaluation of expressions (from [1])	21
4	Sequential semantics for MINIMIR (from [1])	22
5	Speculative semantics for MINIMIR (from [1])	24
6	Ideal semantics for MINIMIR (selected rules, adapted from [1])	27
7	Simulation diagram for SpecIBT (adapted from [2])	35

1 Introduction

1.1 Background

Programmers generally think of program execution as *sequential*; that is, the explicitly specified control flow is respected. For example, if a branch condition evaluates to false during sequential execution, the branch ought not to be taken. Sequential execution is an appropriate abstraction for everything from high-level programming languages down to ISA-level, as we expect these to express our intentions with respect to program behavior, and we expect (correct) compilation to preserve these intentions. But this abstraction breaks down at the microarchitectural level.

Modern processors are deeply pipelined, which increases throughput by breaking the processing of an instruction into stages so that multiple instructions can be processed at different stages at the same time. Pipeline stalls occur when a piece of data needed to complete a processing stage for a given instruction is not available, causing delays in the processing of subsequent instructions that depend on it. *Speculative execution* is a CPU optimization where instructions are processed before it is certain that they are supposed to be, avoiding stalls. Hardware predictors attempt to guess the most likely outcome based on recent history (for example, the outcomes of recent conditional branches are stored in the branch history buffer). An incorrect prediction's immediate result is a sequentially impossible execution, such as a branch being taken even though the branch condition is false. Once it is discovered that the prediction is incorrect, the effects of such an execution are reverted in the microarchitecture, and never reach the architecture. However, if the program accessed memory speculatively, the data will have been cached, and this effect is not reverted. Cache side-channel attacks can be used to infer the value of data accessed by the program during speculation.

Cache attacks, like speculative attacks, are an instance of the more general class of *side-channel attacks*, where the system working as intended entails an unintended leakage of information. A carefully crafted speculative attack will target a sequentially impossible execution involving secret-dependent memory accesses, and infer the value of the secret through cache manipulation [3]. There are many variants of cache attack, but all of them rely on the sharing of hardware resources between processes. Such sharing enables an attacker process to manipulate cache state in such a way that it is possible to infer information about the victim’s execution via timing attacks [4, 5]. In particular, if the victim process makes secret-dependent memory accesses, the attacker may be able to infer the value of the secret.

SpecIBT is a new mitigation presented by my co-authors and myself [1], extending the secure compilation scheme known as Ultimate Speculative Load Hardening (USLH). USLH, on its own, effectively detects and mitigates Spectre PHT attacks [6], but it is not sufficiently powerful to protect against Spectre BTB attacks, in which the attacker can misspeculate to arbitrary locations in the program, circumventing USLH protections [7]. To provide partial protection against Spectre BTB attacks, SpecIBT uses a hardware protection mechanism known as Intel Control-Flow Enforcement Technology (CET), which induces a hardware fault if the attacker misspeculates to any location other than a valid call target [8]. Because this still allows for some misspeculation to go undetected, SpecIBT adds a novel software-based protection that detects when a Spectre BTB attacker misspeculates to the *wrong* valid call target—one that differs from the sequentially intended call target [1]. Once speculation is detected, mitigations are triggered, in the form of *masking* sensitive data so that no information is leaked to the attacker. Thus SpecIBT offers *precise* detection and mitigation of Spectre PHT and BTB, preventing all speculative leaks arising from such attacks.

The original SpecIBT preprint [1] also claimed automatic protection against Spectre RSB attacks, which target the return stack buffer, via the Intel CET shadow stack. However, it

turned out that more active protections were required, which motivated the second iteration of SpecIBT, called Trio securis [9] (so named because it protects against all three variants: PHT, BTB, and RSB). This thesis only considers PHT and BTB attacks.

The absence of speculative leaks is formalized as *relative security*, where the target program, executing speculatively, does not leak more than the source program in the absence of speculative execution [10, 11]. Our formalization uses a toy language, MINIMIR, similar in level to LLVM Machine IR. It is both the source and the target language for the SpecIBT transformation. We use a sequential semantics to model execution of the source program in the absence of speculation; a speculative semantics that models a powerful attacker with full control over hardware predictors; and an *ideal semantics*, as introduced by Shivakumar et al. [12], that models both the adversarial environment of the speculative semantics and the protections afforded by SpecIBT. The source program executes in the sequential semantics and the ideal semantics, the latter of which functions as a specification that correctly expresses the program behavior we desire to enforce with the SpecIBT transformation. The transformed program executes in the adversarial environment of the speculative semantics, against which it should be protected, provided we implemented SpecIBT correctly.

Our proof of relative security is decomposed into two parts, the first one being a backward compiler correctness lemma that proves that our implementation of SpecIBT satisfies its specification as expressed by the ideal semantics. We then use an *unwinding* lemma to prove that relative security holds between the source program sequentially and ideally. This thesis describes the backward compiler correctness lemma and its proof, after establishing background for the work in which it is situated.

All proofs associated with SpecIBT/Trio securis have been mechanized in Rocq. I have provided these proofs as an artifact to accompany this thesis.

1.2 My Contribution

The research that culminated in first SpecIBT, and then Trio securis, also led to the writing of this thesis. In this section, I will describe the relation of my thesis to the published work, and clearly delineate my contribution.

I worked on SpecIBT [1] during a research internship at the Max Planck Institute for Security and Privacy (MPI-SP) in 2025. My primary contribution to SpecIBT was the compiler correctness single cycle lemma. It is 489 lines of code, corresponding to the lemma named `ultimate_slh_bcc_single_cycle_refactor`, starting at line 1616 of `MiniCET_Index.v` in the SpecIBT artifact). Most of the code in this proof is my own, with the exception of a part of the call case. The call case was not quite finished by the time my internship ended, so Yonghyun Kim, one of my mentors, completed it.

I also made other, minor contributions to the Rocq development: parts of the speculative and ideal semantics, some of the functions used to compute synchronization, the branch case of the program transformation, and a few small lemmas. Additionally, I had previously extended existing Rocq proofs of relative security for a version of the SpecIBT transformation implemented on a high-level toy language to include indirect calls and indexing on the number of steps in the target execution, as a proof of concept.

The SpecIBT preprint was published as J. Baumann, Y. Kim, Y. Farba, C. Hritcu, and J. Leatherman-Brooks, “SpecIBT: Formally verified protection against speculative control-flow hijacking”, 2026, <https://arxiv.org/abs/2601.22978v1>.

Trio securis was published as J. Baumann, Y. Kim, Y. Farba, C. Hritcu, and J. Leatherman-Brooks, “Trio securis: Formally verified protection against speculative control-flow hijacking”, CSF 2026 – 39th IEEE Computer Security Foundations Symposium, <https://doi.org/10.1109/CSF68417.2026.00036>.

The final publication briefly describes the compiler correctness lemma, gives a high-level proof sketch, and cites this thesis as providing a more detailed description (nineteen pages to the publication’s one).

The rest of my thesis establishes context for understanding the lemma and its proof, and in so doing inevitably covers some of the same ground covered by the publication. Specifically, in sections 2-5 I present contextual material that is also presented in the publication, and reuse (sometimes adapted) figures and listings from the publication. However, all of the writing in this thesis *is* my own, and represents my own understanding of the work. No text is directly incorporated from the publication, except where it appears in definitions, figure captions, etc., and all such instances are accompanied by a citation.

1.3 Roadmap

In §2 I offer an overview of speculative attacks and defenses, describe SpecIBT’s novel contribution, and briefly state and justify our threat model and security definition. Additionally, I describe the SpecIBT transformation informally by example. All remaining sections describe the formalization. In §3 I introduce the syntax of the MINIMIR language and justify the language level we chose. In §4 I give a precise definition of the transformation. The various semantics are discussed in §5. A proof sketch of relative security for SpecIBT is offered in §6. The backward compiler correctness lemma is considered in depth in §7.

2 Speculative Attacks and Defenses

2.1 Attacks

Speculative execution is triggered when a piece of data needed to complete a processing stage for a given instruction is not available, causing delays in the processing of subsequent instructions that depend on it. To avoid such delays, hardware predictors make informed guesses as to the most likely outcome for the instruction, based on recent history, and execution proceeds down a speculative path.

Once the data arrives, the correct execution path can be determined. If the prediction was wrong, most effects of speculative execution are thrown away, meaning that they are never committed architecturally. However, speculative memory accesses may leave data in the cache. This enables a side channel—an information leak that occurs as a by-product of the system functioning as intended. Through attacks such as Flush+Reload [13], an attacker can leverage the difference in timing between a cache access and a main memory access to determine whether a piece of data was cached during the execution of the victim program. This can lead to the inference of details about the contents of the victim’s memory [4]. Attacker-controlled speculation can escalate the severity of leakage to include data that would never be accessed sequentially.

In order to mount a speculative attack, the attacker first identifies a suitable program point where a mispredicted branch or indirect call for which they control the inputs will lead to leakage of data that is *not sequentially accessible*. The attacker then mistrains the appropriate predictor by repeatedly executing the victim process with innocuous inputs leading to a certain prediction, and then executing it with malicious inputs for which this prediction is incorrect. This causes the processor to proceed with a sequentially impossible execution. Finally, the attacker uses a cache attack to infer the value of the data left in the cache by

this execution [3, 4].

Speculative attacks are differentiated by the specific predictor that is mistrained. The pattern history table (PHT) feeds the outcome of the most recent execution of a conditional branch into an algorithm to determine whether the branch is likely to be taken again. The branch target buffer (BTB) stores recent targets of indirect control-flow instructions, and is used to predict the next target. The return stack buffer (RSB) works similarly with respect to predicting the next return location of an indirect call. These are exploited by Spectre PHT, BTB, and RSB, respectively.

2.2 Defenses

SpecIBT is a new defense against speculative control-flow attacks. It is built on top of Ultimate Speculative Load Hardening (USLH), a software-based defense that, on its own, protects against Spectre PHT [6], and Intel’s hardware-based Control-Flow Enforcement Technology (CET), which provides a coarse-grained level of protection against misspeculation on indirect calls [8]. SpecIBT’s novel contribution is a software-based technique that improves upon Intel CET by *precisely* detecting when such misspeculation is occurring.

2.2.1 USLH and Spectre PHT

It is useful to separate defense against speculative attacks into two separate stages: *detection* and *mitigation*. USLH detects and mitigates Spectre PHT. It detects speculation by using a *predicated instruction* to check whether the branch condition evaluates to what it should for each arm of the branch: true if taken, false if not. Predicated instructions perform such checks with branchless logic, making them unsusceptible to misprediction. Misspeculation is detected when the attacker has steered execution to the incorrect arm of the branch. In this case, the dedicated misspeculation flag (*msf*) register is set, which triggers masking of

sensitive data from that point on. Leaks are still permitted to occur, but masking ensures that no information is transmitted.

Note that the *msf* register state is *monotonic*, meaning that once it is set, it remains set. We do not model rollbacks in our formalization.

Example 1. To provide an intuition for how Spectre PHT attacks work, and how USLH detects and mitigates them, we present a snippet of vulnerable source code in [Listing 1](#); partially transformed target code where detection has been implemented but not mitigation in [Listing 2](#); and fully transformed target code, including mitigation, in [Listing 3](#).

Listing 1 (Vulnerable program (adapted from [3])).

<pre>branch ($i < a_len$) to l ... </pre>
<pre>l : $r1 \leftarrow \text{load}[a + i]$ $r2 \leftarrow \text{load}[b + (512 * r1)]$ </pre>

[Listing 1](#) shows a bounds check on array a . The attacker will train the predictor with values of i that are within bounds, so that it will learn to predict the branch taken, and then supply an out-of-bounds i . We assume that all data not intended to be accessed sequentially by the program is secret. The predictor will cause the branch to be taken with the out-of-bounds i , so that $a + i$ indexes secret data, which is placed in $r1$ by the first load. The second load computes its index using the secret data, multiplied by the cache line size (512), introducing the possibility of inferring it via cache attack [3].

Listing 2 (Detection of misspeculation by USLH (adapted from [3])).

<pre>branch (i < a_len) to l' msf := (i < a_len) ? 1 : msf ...</pre>
<pre>l' : msf := (¬(i < a_len) ? 1 : msf) jump l</pre>
<pre>l : r1 ← load[a + i] r2 ← load[b + (512 * r1)]</pre>

[Listing 2](#) shows only the detection mechanism of USLH. An instruction is added inside each arm of the branch, that uses the unpredicted conditional move to check whether the branch condition evaluates to the opposite of what it should for that arm. If so, the dedicated *msf* register is set. Branch targets may be arrived at from multiple **branch** and **jump** instructions; the added block disambiguates which one so that it's clear which branch condition to check. In our running example, an out-of-bounds *i* would make the branch condition false, and this would be detected by the first instruction in *l'*. No mitigations have been added yet to prevent the leak of secret data in block *l*; this is shown in [Listing 3](#).

In [Listing 3](#), masking is applied to the branch condition if misspeculation was detected earlier, so that a value of *false* is leaked, regardless of the original value of the branch condition. If misspeculation is initiated with the branch instruction, the branch condition will be leaked, but misspeculation will be detected at the next instruction, as shown in [Listing 2](#). Then, in

the case of a taken branch, masking will be applied to both load addresses, so that the first load is no longer out of bounds. The computation of the address of the second load will thus no longer depend on a secret value, and the attacker will not be able to infer anything.

Listing 3 (Mitigation of misspeculation by USLH (adapted from [3])).

<pre> branch ((<i>msf</i> = 1) ? 0 : (i < a_len)) to l' <i>msf</i> := (i < a_len) ? 1 : <i>msf</i> ... </pre>
<pre> l' : <i>msf</i> := (¬(i < a_len) ? 1 : <i>msf</i>) jump l </pre>
<pre> l : r1 ← load[(<i>msf</i> = 1) ? 0 : (a + i)] r2 ← load[(<i>msf</i> = 1) ? 0 : (b + (512 * i))] </pre>

2.2.2 Intel CET and Spectre BTB

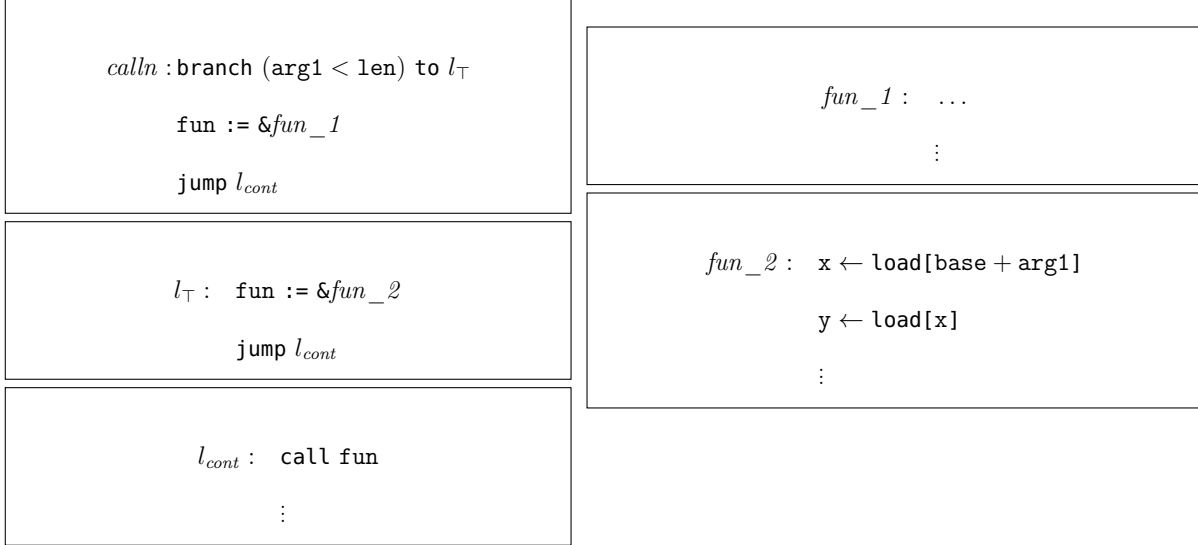
Spectre BTB is a more powerful attack than Spectre PHT, because the attacker, by mis-training the BTB, gains the ability to use indirect call instructions to divert control flow to any location in the program. Notably, USLH protections are of no use against such attacks, as the detection and mitigation described in [Example 1](#) can be speculated past by a BTB attacker [7].

In the listings below, the `&` notation is used to denote function pointers, and the `call`

instruction is an indirect call.

Example 2. Consider the following vulnerable source program:

Listing 4 (Vulnerable program (from [1])).



Either *fun_1* or *fun_2* will be stored in **fun** and called in the *l_{cont}* block, depending on whether the branch is taken or not. If *fun_2* is called, there is the possibility of leakage. The **call** instruction in the *l_{cont}* block is a potential site of misspeculation for a BTB attacker.

[Listing 5](#) below shows this program with only USLH protections added. The attacker could allow the branch to fall through, legitimately, so that *fun_1* is stored in **fun** and *l_{cont}* is jumped to, and the *msf* register remains unset. Once in *l_{cont}*, such an attack can be mounted which directly overwrites the call to *fun_1* with a call to *fun_2*. Once in the *fun_2* block, *msf* will not have been set, because the attacker’s misspeculation wasn’t detected, so potentially sensitive data will be exposed to the attacker via the pair of load instructions.

Intel CET does not fully mitigate such attacks, but it does reduce the attack surface. Intel CET statically overapproximates the set of reachable call target blocks for a given program and checks, during runtime, whether an indirect call has transferred control to the entry to

Listing 5 (Vulnerable program + USLH (from [1])).

<pre> calln : branch (arg1 < len) to l'_T msf := (arg1 < len) ? 1 : msf fun := &fun_1 jump l_cont </pre>	<pre> fun_1 : ... : </pre>
<pre> l'_T : msf := (arg1 ≥ len) ? 1 : msf jump l_T </pre>	<pre> fun_2 : x ← load[(msf = 1) ? 0 : (base + arg1)] y ← load[(msf = 1) ? 0 : x] : </pre>
<pre> l_T : fun := &fun_2 jump l_cont </pre>	
<pre> l_cont : call fun : </pre>	

one of these blocks or not. We model this by adding the **ctarget** instruction to all call target blocks (see [Listing 6](#) below) and requiring that it is encountered immediately after a call. If not, a fault occurs. This prevents misspeculation to the middle of any block, and to the beginning of a block that is not a possible function entry point with respect to the program. It does not, however, prevent the attack described above, as both *fun_1* and *fun_2* are reachable, depending on whether or not the initial branch is taken.

SpecIBT is able to *precisely* detect BTB attacks like the one above. [Listing 6](#) below shows a program instrumented with SpecIBT protections. SpecIBT’s novel contribution is to store the intended call target in the dedicated *callee* register on the caller side [1]. (Here, that means *fun_1* gets stored.) Then on the callee side, the contents of *callee* are checked against the current location, and the *msf* register is set if they don’t match. Thus when the subsequent loads execute, the data is masked and nothing is leaked to the attacker.

Listing 6 (Vulnerable program + SpecIBT (from [1])).

<pre> calln : branch (arg1 < len) to l'_T msf := (arg1 < len) ? 1 : msf fun := &fun_1 jump l_cont </pre>	<pre> fun_1 : ctarget msf := (callee ≠ &fun_1) ? 1 : msf : </pre>
<pre> l'_T : msf := (arg1 ≥ len) ? 1 : msf jump l_T </pre>	<pre> fun_2 : ctarget msf := (callee ≠ &fun_2) ? 1 : msf x ← load[(msf = 1) ? 0 : (base + arg1)] y ← load[(msf = 1) ? 0 : x] : </pre>
<pre> l_T : fun := &fun_2 jump l_cont </pre>	
<pre> l_cont : callee := fun call fun : </pre>	

2.2.3 Threat Model and Security Definition

We seek to mitigate against Spectre PHT and BTB attacks, so our threat model assumes an attacker with full control over these predictors. Our security definition follows from the fact that we seek to protect *arbitrary* programs. By “arbitrary”, we mean programs that do not enforce a distinction between secret and public data, and that do not follow the cryptographic constant-time (CCT) programming discipline. The CCT discipline requires that instructions vulnerable to timing attacks (control flow and memory access instructions, and variable-time instructions) not be secret-dependent. CCT programs typically use an information flow control type system that types registers and memory locations as holding secret or public data, and restricts the flow of secrets into public locations. Such programs

can be proven secure against timing attacks in the absence of speculation [14]. Arbitrary programs cannot be assumed to distinguish between secret and public data (any memory location might hold secret data), and might have leaks even in the absence of speculation.

Relative Security SpecIBT seeks to protect arbitrary programs from leaks arising from speculative attacks. Sequential leaks are out of scope; we only seek to ensure that no *new* leaks can be introduced during speculative execution. Given this objective, *relative security* is an appropriate choice of security definition: it states that the target program, executing speculatively, will not leak more than the source program executing sequentially [10, 11].

Formalizing relative security requires the notion of *indistinguishability*: two executions of a program are indistinguishable if they emit the same observation trace so that an attacker cannot infer information about the program by comparing them. This property is also referred to as *observational equivalence*. Relative security is stated as an implication from source to target: if two runs of the source program in the sequential semantics satisfy observational equivalence, then two runs of the target in the speculative semantics satisfy observational equivalence. This rules out handling source programs with leaks; as previously noted, these should be out of scope for our mitigation.

3 The MiniMIR Language

MINIMIR is a simple untyped imperative language with basic blocks, similar in level to LLVM Machine IR (hence its name). It is the source and target language for SpecIBT. MINIMIR is similar in some ways to assembly code, but its instructions take arbitrary expressions as arguments; this allows us to make use of necessary higher-level constructs such as the constant-time conditional.

$e \in \text{exp} ::= n \in \mathbb{N}$	number
$\&l \in \mathcal{FP}$	function pointer
$X \in \mathcal{R}$	register
$\text{op}_{\mathbb{N}}(e, \dots, e)$	arithmetic operator
$b ? e : e$	constant-time conditional
$c \in \text{inst} ::= \text{skip}$	do nothing
$X := e$	assignment
branch e to l	conditional branch
jump l	unconditional jump
$X \leftarrow \text{load}[e]$	load from address
$\text{store}[e] \leftarrow e$	store to address
call e	indirect call
ctarget	call-target marker
ret	return
$b \in \text{blk} ::= (l, ([\text{inst}], \top/\perp))$	
$p \in \text{prog} ::= [\text{blk}]$	

Figure 1: Language syntax (adapted from [1])

3.1 Syntax

As shown in Figure 1, a MINIMIR program is a list of basic blocks, where a block is a tuple consisting of a list of instructions and a boolean annotation designating the block as a call target or not. Block indices are used as labels for block lookup in control-flow instructions. Instructions, as mentioned previously, take arbitrary expressions as arguments; these include

arithmetic operators, registers, constant-time conditionals, numbers, and function pointers. Constant-time conditional expressions are central to the USLH defense strategy: they allow the transformation of an instruction into a *predicated* instruction. As mentioned in §2.2.1, predicated instructions use branchless logic to evaluate conditionals, so an attacker cannot misspeculate past instructions added by SpecIBT to detect or mitigate attacks. A function pointer is a label that refers to a call target block. Function pointer expressions, denoted by $\&l$, evaluate to function pointer values, denoted by $\&l$.

Instructions include `ctarget`, our version of `endbranch`, which is a no-op in the source program and appears only at the head of call target blocks in the target program. The `call` instruction is an indirect call; `branch` and `jump` are direct.

Values are either natural numbers, function pointers, or *undefined values*. As shown in Figure 3, the only operation that is defined on function pointers is an equality comparison, which is needed for our mitigation (see Figure 2b). Undefined values are the result of any other operation performed on function pointers, or of any operation performed on a function pointer and a number. We discuss the reason for including undefined values in our language in §3.4.

3.2 Language Level

MiniMIR is a low-level intermediate language. This is by design: USLH-based compilation passes are best placed relatively late in the compilation process to reduce the chances that subsequent passes will break the protections it affords [15]. It is therefore reasonable to wonder why we did not design the language to be even more low-level, for example, completely linearized with no basic blocks, and having already translated function pointers to numbers? To see why especially the distinction between function pointers and numbers is necessary, it is useful to consider the SpecIBT transformation on indirect calls (We examine this in detail

in Figure 2b).

SpecIBT adds two instructions to the head of each call target block. Without blocks and block annotations, call target addresses would need to be explicitly declared, so that SpecIBT would know where to insert the two instructions that are added at the head of every call target block. However, because of this insertion of instructions, the call target address would need to be adjusted in the `call` instruction. Furthermore, with a linearized program, such adjustments would propagate through the entire program, requiring further, cumulative adjustments. Not only that, but any values stored in registers or memory might or might not be eventually used as call target addresses, and only those so used would need adjustment. It is easier to encode the distinction between function pointers and numbers into the language, and easier, too, to use annotated blocks to identify call targets.

3.3 Well-formedness

We define a well-formedness predicate that adds additional restrictions not expressed in the syntax. A well-formed program is nonempty and contains only well-formed blocks. A well-formed block is nonempty; ends with a `ret` or `jump` instruction (no fallthrough is allowed from block to block); and contains only well-formed instructions. An instruction is well-formed if all of its expressions are well-formed. The instructions `branch` and `jump` are well-formed only if their labels do not refer to call target blocks. Well-formedness is defined recursively on expressions such that all their subexpressions are well-formed, and function pointer expressions must refer to call target blocks (although `call` instructions are not required to have function pointer arguments; as mentioned above, this is consistent with how the argument of an indirect call is computed at runtime).

3.4 Undefined Values

If comparisons between function pointers and numbers are defined, our program transformation does not preserve sequential behavior. This is because it adds instructions that will result, during linearization, in the adjustment of function pointer addresses, so that one of the numbers in the comparison will be different and the comparison might turn out differently as a result.

However, choosing to treat such comparisons as undefined behavior also creates problems, this time directly related to SpecIBT mitigations. If misspeculation has been detected, then all loads and all stores will access address 0 of memory. This can lead to a scenario where, sequentially, a function pointer is stored to address i , a number is loaded from address j , where $i \neq j$, and then the number is used in a number-number comparison. However, speculatively, $i = j = 0$, and it will be the function pointer that is loaded, resulting in a function pointer-number comparison [1]. In this case, the source program will not encounter undefined behavior, but the target program will. This breaks relative security, because two executions of the source program will be observationally equivalent, but two executions of the target program cannot be, if one of them encounters undefined behavior.

The third option is to use undefined values. This defines comparisons between numbers and function pointers under misspeculation. The only concern would be if an undefined value were to be used as the argument of a memory access or control-flow instruction. However, under misspeculation these arguments will all be masked to 0, so this is not a concern.

4 The SpecIBT transformation

As shown in Figure 2, the SpecIBT transformation acts on programs at the instruction, block, and whole-program levels. In this section, I will briefly walk through how this works at each level.

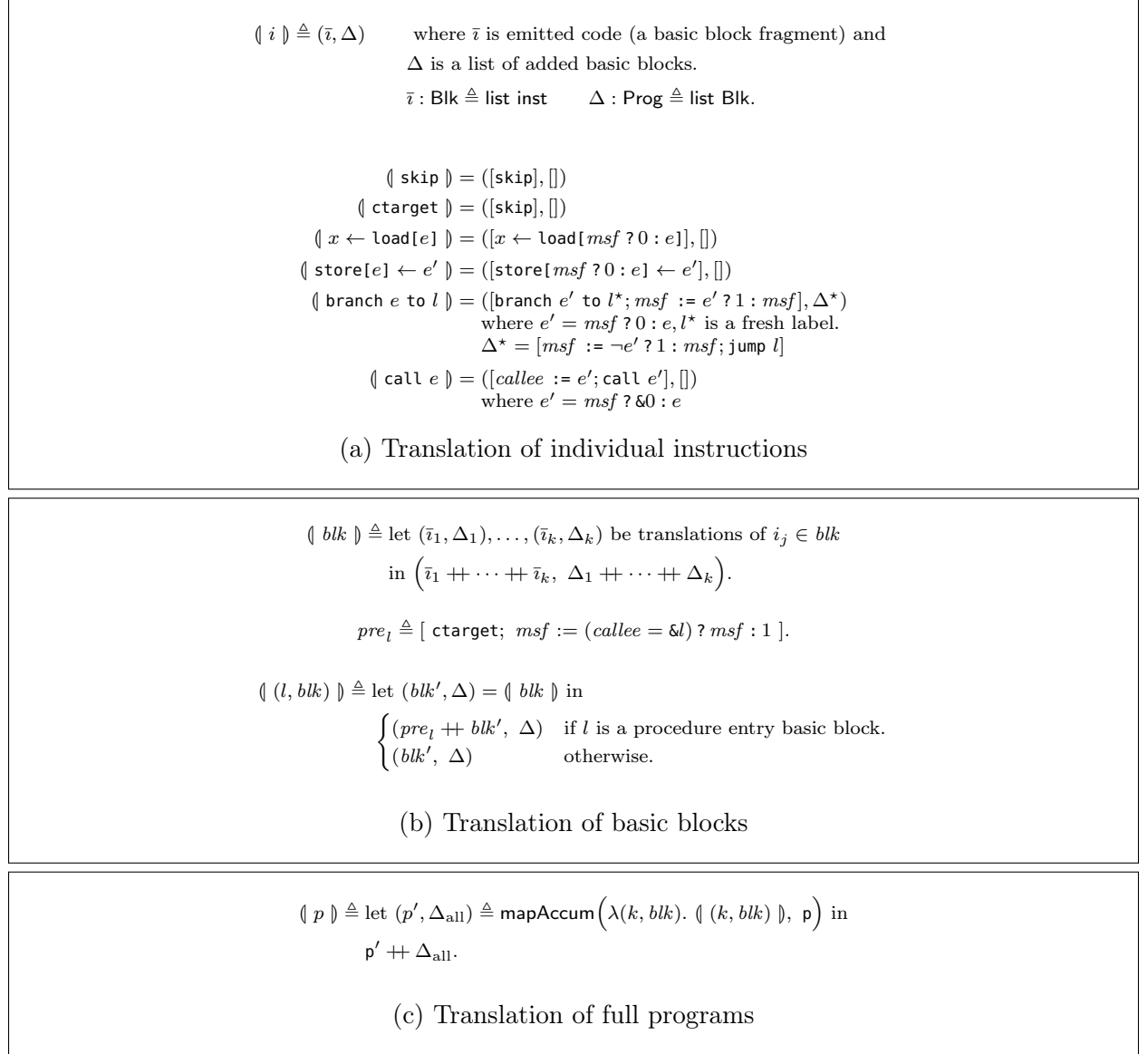


Figure 2: SpecIBT transformation (adapted from [1])

On the instruction level (Figure 2a), SpecIBT takes an instruction and returns a pair; a list of instructions and a list of added blocks. The list of instructions may contain any of the following: added instructions as well as the source instruction with its argument wrapped in a constant-time conditional (in the case of **branch** and **call**); the source instruction with its expression wrapped in a constant-time conditional (in the case of **load** and **store**); or just the original, unaltered instruction. When SpecIBT encounters a use of the **ctarget** instruction by the source program, it turns it into a **skip**. The list of added blocks will be empty except in the case of **branch** instructions, where it will contain the interposing block added by SpecIBT (see Listing 2). Because this part of the transformation is applied to a list of instructions, it returns a list of pairs.

On the block level (Figure 2b), SpecIBT first unzips the list into a single pair containing the full sequence of added instructions and the sequence of added blocks, in the order in which they were added. Then it prepends two instructions to each call target block: **ctarget**, which models Intel CET protections, and the instruction that implements SpecIBT’s novel protection mechanism on the callee side by setting the *msf* register if the stored call target label doesn’t match the block label. This part of the transformation takes a block and returns a pair containing the block’s transformed instruction list (we ignore the call target block annotation in this description) and the list of added blocks.

On the program level (Figure 2c), we start with a list of transformed blocks as defined above. **MapAccum** collects all the original blocks in order and all the added blocks in order, returns a pair containing each, and finally appends the added blocks to the transformed original blocks.

5 Semantics

Our formalization uses three semantics: sequential (Figure 4), speculative (Figure 5) and ideal (Figure 6).¹ The sequential semantics model program execution in the absence of speculation. The speculative semantics model program execution in the presence of speculation, and the ideal semantics model an environment with both speculation and SpecIBT protections built in.

$$\begin{aligned}
\text{op} &::= + \mid - \mid * \mid = \mid \leq \mid \wedge \mid \rightarrow \\
v_1 \text{ op } v_2 &= \begin{cases} l_1 = l_2 & \text{if } v_1 = \&l_1, v_2 = \&l_2, \text{op is } = \\ \text{standard} & \text{if } v_1, v_2 \in \mathbb{N} \\ \text{UV} & \text{otherwise} \end{cases} \\
\llbracket n \rrbracket_\rho &= n \\
\llbracket \&l \rrbracket_\rho &= \&l \\
\llbracket x \rrbracket_\rho &= \rho(x) \\
\llbracket e_1 \text{ op } e_2 \rrbracket_\rho &= \llbracket e_1 \rrbracket_\rho \text{ op } \llbracket e_2 \rrbracket_\rho \\
\llbracket b ? e_1 : e_2 \rrbracket_\rho &= \text{if } \llbracket b \rrbracket_\rho \neq 0 \text{ then } \llbracket e_1 \rrbracket_\rho \text{ else } \llbracket e_2 \rrbracket_\rho
\end{aligned}$$

Figure 3: Evaluation of expressions (from [1])

5.1 Sequential Semantics

The sequential semantics, shown in Figure 4, model program execution when control flow is not altered by speculative mispredictions. A sequential step is denoted as $\xrightarrow{\mathcal{O}}$, where $\mathcal{O}$ is the observation associated with the step ($\bullet$ if no observation is emitted). Multi-step execution is the reflexive, transitive closure of the single-step relation. It is written as $\xrightarrow{\mathcal{O}*}$, where the arrow with star denotes zero or more steps. Instruction fetch is denoted by $\mathbf{p}[\text{pc}] = i$, and

¹Programs are fixed in the semantics, although there is no notation indicating this.

$$\begin{array}{c}
\text{SEQ_SKIP} \frac{\mathbf{p}[[pc]] = \text{skip}}{\langle pc, \rho, \mu, stk \rangle \xrightarrow{\bullet} \langle pc + 1, \rho, \mu, stk \rangle} \\
\text{SEQ_ASGN} \frac{\mathbf{p}[[pc]] = x := e \quad \llbracket e \rrbracket_{\rho} = v}{\langle pc, \rho, \mu, stk \rangle \xrightarrow{\bullet} \langle pc + 1, [x \mapsto v] \rho, \mu, stk \rangle} \\
\text{SEQ_BRANCH} \frac{\mathbf{p}[[pc]] = \text{branch } e \text{ to } l \quad \llbracket e \rrbracket_{\rho} = n \quad b = (n \neq 0)}{pc' = \text{if } b \text{ then } (l, 0) \text{ else } pc + 1} \\
\langle pc, \rho, \mu, stk \rangle \xrightarrow{OBranch\ b} \langle pc', \rho, \mu, stk \rangle \\
\text{SEQ_JUMP} \frac{\mathbf{p}[[pc]] = \text{jump } l}{\langle pc, \rho, \mu, stk \rangle \xrightarrow{\bullet} \langle (l, 0), \rho, \mu, stk \rangle} \\
\text{SEQ_LOAD} \frac{\mathbf{p}[[pc]] = x \leftarrow \text{load}[e] \quad \llbracket e \rrbracket_{\rho} = n \quad \mu[n] = v}{\langle pc, \rho, \mu, stk \rangle \xrightarrow{OLoad\ n} \langle pc + 1, [x \mapsto v] \rho, \mu, stk \rangle} \\
\text{SEQ_STORE} \frac{\mathbf{p}[[pc]] = \text{store}[e] \leftarrow e' \quad \llbracket e \rrbracket_{\rho} = n \quad \llbracket e' \rrbracket_{\rho} = v}{\langle pc, \rho, \mu, stk \rangle \xrightarrow{OStore\ n} \langle pc + 1, \rho, [n \mapsto v] \mu, stk \rangle} \\
\text{SEQ_CALL} \frac{\mathbf{p}[[pc]] = \text{call } e \quad \llbracket e \rrbracket_{\rho} = \&l}{\langle pc, \rho, \mu, stk \rangle \xrightarrow{OCall\ l} \langle (l, 0), \rho, \mu, (pc + 1) :: stk \rangle} \\
\text{SEQ_CTARGET} \frac{\mathbf{p}[[pc]] = \text{ctarget}}{\langle pc, \rho, \mu, stk \rangle \xrightarrow{\bullet} \langle pc + 1, \rho, \mu, stk \rangle} \\
\text{SEQ_RET} \frac{\mathbf{p}[[pc]] = \text{ret}}{\langle pc, \rho, \mu, pc' :: stk \rangle \xrightarrow{\bullet} \langle pc', \rho, \mu, stk \rangle} \\
\text{SEQ_TERM} \frac{\mathbf{p}[[pc]] = \text{ret}}{\langle pc, \rho, \mu, [] \rangle \xrightarrow{\bullet} \text{Term}}
\end{array}$$

Figure 4: Sequential semantics for MINIMIR (from [1])

evaluation of expressions with respect to register state by $\llbracket e \rrbracket_{\rho}$, defined in [Figure 3](#).²

The state of a running program is represented in the sequential semantics as $\langle pc, \rho, \mu, stk \rangle$, where pc is the program counter, which consists of a block label l and an offset o specifying an instruction within the block; ρ is the register state, a mapping of identifiers to values; μ is the memory state, a flat array of values accessed by index, and stk is the stack, a list

²If an expression is ill-formed and would trigger undefined behavior when evaluated, this is avoided in the case where misspeculation is already occurring and the expression is masked; the mask always has the correct type.

of program counters. Normal termination is represented in the semantics as **Term**. Normal termination occurs when $\mathbf{p} \llbracket pc \rrbracket = \mathbf{ret}$ and $stk = []$. **Term** and **Fault** are considered as final states; no sequential rules lead to faulting.

A program's current state is undefined if it is non-final and there exists no rule to step to a successor state. There is no explicit representation in the transition rules for undefined states. Undefined behavior for each rule can be inferred by enumerating the ways in which its premises could be falsified. For example, the source program $\mathbf{p}$ would encounter undefined behavior on **call** e if $\llbracket e \rrbracket_\rho \neq \&l$.

Taken branches, calls, and jumps go to the first instruction in the target block; the sequential semantics do not allow jumps to the middle of a block. As mentioned in §3.3, due to well-formedness, calls can only go to call target blocks, and branches and jumps can only go to non-call-target blocks. Call target blocks and non-call-target blocks are disjoint sets. Calls push the return address onto the stack. Returns pop the saved pc off the stack and jump to it in the successor state. Termination is defined as a return where the stack is empty. There is a rule here for the SEQ_CTARGET instruction, but it only advances the program counter; **ctarget** is treated as a no-op sequentially. The SpecIBT transformation erases all **ctarget** instructions to be sure that they only occur where the transformation places them.

In all three semantics, it is possible for a single step of execution to produce an *observation*, which is information leaked to the attacker. Multiple execution steps produce a sequence of such observations, referred to as a *trace*. Observations on a single step are represented in the semantics above the transition arrow as *OBranch* b , *OCall* l , *OLoad* n , and *OStore* n , where b is a branch condition, l is a call target label, and n is a memory address. This model of sequential leakage is justified by the fact that an attacker can infer information about a program sequentially from timing differences due to control flow or memory accesses [14]. When there is no observation produced by a step, this is represented with $\bullet$ above the arrow.

5.2 Speculative Semantics

$$\begin{array}{c}
\text{SPEC_SKIP} \frac{\mathbf{p}[[pc]] = \text{skip}}{\langle pc, \rho, \mu, stk, \perp, ms \rangle \xrightarrow[\bullet]{s} \langle pc + 1, \rho, \mu, stk, \perp, ms \rangle} \\
\\
\text{SPEC_ASGN} \frac{\mathbf{p}[[pc]] = x := e \quad \llbracket e \rrbracket_\rho = v}{\langle pc, \rho, \mu, stk, \perp, ms \rangle \xrightarrow[\bullet]{s} \langle pc + 1, [x \mapsto v]\rho, \mu, stk, \perp, ms \rangle} \\
\\
\text{SPEC_BRANCH} \frac{\mathbf{p}[[pc]] = \text{branch } e \text{ to } l \quad \llbracket e \rrbracket_\rho = n \quad b = (n \neq 0) \quad pc' = \text{if } b' \text{ then } (l, 0) \text{ else } pc + 1 \quad ms' = ms \vee (b \neq b')}{\langle pc, \rho, \mu, stk, \perp, ms \rangle \xrightarrow[\text{DBranch } b']{OBranch \ b} \langle pc', \rho, \mu, stk, \perp, ms' \rangle} \\
\\
\text{SPEC_JUMP} \frac{\mathbf{p}[[pc]] = \text{jump } l}{\langle pc, \rho, \mu, stk, \perp, ms \rangle \xrightarrow[\bullet]{s} \langle (l, 0), \rho, \mu, stk, \perp, ms \rangle} \\
\\
\text{SPEC_LOAD} \frac{\mathbf{p}[[pc]] = x \leftarrow \text{load}[e] \quad \llbracket e \rrbracket_\rho = n \quad \mu[n] = v}{\langle pc, \rho, \mu, stk, \perp, ms \rangle \xrightarrow[\bullet]{OLoad \ n} \langle pc + 1, [x \mapsto v]\rho, \mu, stk, \perp, ms \rangle} \\
\\
\text{SPEC_STORE} \frac{\mathbf{p}[[pc]] = \text{store}[e] \leftarrow e' \quad \llbracket e \rrbracket_\rho = n \quad \llbracket e' \rrbracket_\rho = v}{\langle pc, \rho, \mu, stk, \perp, ms \rangle \xrightarrow[\bullet]{OStore \ n} \langle pc + 1, \rho, [n \mapsto v]\mu, stk, \perp, ms \rangle} \\
\\
\text{SPEC_CALL} \frac{\mathbf{p}[[pc]] = \text{call } e \quad \llbracket e \rrbracket_\rho = \&l \quad ms' = ms \vee (pc' \neq (l, 0))}{\langle pc, \rho, \mu, stk, \perp, ms \rangle \xrightarrow[\text{DCall } pc']{OCall \ l} \langle pc', \rho, \mu, (pc + 1) :: stk, \top, ms' \rangle} \\
\\
\text{SPEC_RET} \frac{\mathbf{p}[[pc]] = \text{ret}}{\langle pc, \rho, \mu, pc' :: stk, \perp, ms \rangle \xrightarrow[\bullet]{s} \langle pc', \rho, \mu, stk, \perp, ms \rangle} \\
\\
\text{SPEC_TERM} \frac{\mathbf{p}[[pc]] = \text{ret}}{\langle pc, \rho, \mu, [], \perp, ms \rangle \xrightarrow[\bullet]{s} \text{Term}} \\
\\
\text{SPEC_CTARGET} \frac{\mathbf{p}[[pc]] = \text{ctarget}}{\langle pc, \rho, \mu, stk, \top, ms \rangle \xrightarrow[\bullet]{s} \langle pc + 1, \rho, \mu, stk, \perp, ms \rangle} \\
\\
\text{SPEC_FAULT} \frac{\mathbf{p}[[pc]] = i \quad i \neq \text{ctarget}}{\langle pc, \rho, \mu, stk, \top, ms \rangle \xrightarrow[\bullet]{s} \text{Fault}}
\end{array}$$

Figure 5: Speculative semantics for MINIMIR (from [1])

The speculative semantics (Fig. 5) is an extension of the sequential semantics (additions in gold font) with attacker directives, two flags added to the state, and two additional transition rules modeling Intel CET. Transitions are represented by $\xrightarrow[\mathcal{D}]{\mathcal{O}}_s$. The small s differentiates

speculative transitions from sequential ones, and $\mathcal{D}$ is an option type representing either a singleton list containing the attacker directive consumed by the step, or an empty list if there is no directive associated with the step. When there is no directive consumed by a step, this is represented with $\bullet$ below the arrow. Multistep speculative execution is represented by $\frac{\mathcal{O}}{\mathcal{D}}^*(n)$, where $\mathcal{D}$ denotes the sequence of attacker directives consumed by an n -step program execution.

Attacker Model We use directives to model a powerful speculative attacker with full control over hardware predictors. This is a sound overapproximation, meaning that we have modeled an unrealistically powerful attacker in order to guarantee that our mitigation does not fail to protect against any capability a real-world attacker might have. This approach also allows us to avoid needing to model hardware details, which would make our formalization, especially the proofs, more complex.

The attacker can trigger misspeculation with the control-flow instructions, `branch` and `call`, using directives $DBranch\ b'$ and $DCall\ pc'$, respectively. This corresponds to our threat model, where the attacks we consider are Spectre PHT and BTB. Misspeculation is occurring when the attacker’s choice of branch condition or call target differs from the branch condition or call target that would be used sequentially.

Speculative State Configuration In the speculative semantics, the flags ct and ms are added to program state: $\langle pc, \rho, \mu, stk, ct, ms \rangle$. The former is part of the modeling of Intel CET protections, and the latter mirrors the state of the msf register, and of the ideal ms flag.³ The ct flag is set to true when a `ctarget` instruction is expected, immediately after a `call` instruction, as part of detecting misspeculation on calls.

³Unlike in the ideal semantics, the ms flag is not modeling anything in the hardware, and, unlike the msf register, whether or not it is set does *not* affect the subsequent execution of the program. It is only there to facilitate the proofs. It can be assumed to be always in step with msf and ideal ms .

Fault states are represented in the semantics as **Fault**. This state is the result of misspeculating to anywhere other than the head of a call target block.

Transition Rules `SPEC_BRANCH` uses the attacker directive as the branch condition and detects misspeculation if it is different from the sequential branch condition. The sequential branch condition is leaked. `SPEC_CALL` uses the attacker directive as call target location, and detects misspeculation if its label is different from that of the sequential call target, which is leaked, or if its offset is nonzero (sequential calls only go to zero offsets). The *ct* flag is set in the successor state in expectation of a **ctarget** instruction, and the return address pushed onto the stack.

The new transition rules, `SPEC_FAULT` and `SPEC_CTARGET`, model the two possible outcomes of misspeculation on a call instruction in the presence of Intel CET protections. The former enforces faulting when $ct = \top$ but the instruction is not **ctarget**, while the latter unsets *ct* when the instruction is **ctarget**.

5.3 Ideal Semantics

The notion of an "idealized" semantics comes from the work of Shivakumar et al. [12], where it is the starting point for the design of the concrete program transformation. The authors first identify a security property based on their threat model, and then design an ideal semantics that expresses both the adversarial environment and the protections necessary to guarantee security *in the adversarial environment*. From that semantics, they then define an implementation, which is the concrete program transformation. Finally, they prove that a source program executing ideally and a target program hardened with the concrete transformation executing speculatively will exhibit the same behaviors and are therefore equivalent. Therefore the transformation guarantees security in the adversarial environment. We adopted this notion and general approach, using our own threat model and security definition.

$$\begin{array}{c}
\text{IDEAL_BRANCH} \frac{\text{p}[[pc]] = \text{branch } e \text{ to } l \quad \text{if } ms \text{ then } 0 \text{ else } [[e]]_\rho = n \quad b = (n \neq 0) \\
pc' = \text{if } b' \text{ then } (l, 0) \text{ else } pc + 1 \quad ms' = ms \vee (b \neq b')}{\langle pc, \rho, \mu, stk, ms \rangle \xrightarrow[\text{DBranch } b']{\text{OBranch } b} \langle pc', \rho, \mu, stk, ms' \rangle} \\
\\
\text{IDEAL_LOAD} \frac{\text{p}[[pc]] = x \leftarrow \text{load}[e] \quad \text{if } ms \text{ then } 0 \text{ else } [[e]]_\rho = n \quad \mu[n] = v}{\langle pc, \rho, \mu, stk, ms \rangle \xrightarrow[\bullet]{\text{OLoad } n} \langle pc + 1, [x \mapsto v]\rho, \mu, stk, ms \rangle} \\
\\
\text{IDEAL_STORE} \frac{\text{p}[[pc]] = \text{store}[e] \leftarrow e' \quad \text{if } ms \text{ then } 0 \text{ else } [[e]]_\rho = n \quad [[e']]_\rho = v}{\langle pc, \rho, \mu, stk, ms \rangle \xrightarrow[\bullet]{\text{OStore } n} \langle pc + 1, \rho, [n \mapsto v]\mu, stk, ms \rangle} \\
\\
\text{IDEAL_CALL} \frac{\text{p}[[pc]] = \text{call } e \quad \text{if } ms \text{ then } \&0 \text{ else } [[e]]_\rho = \&l \quad ms' = ms \vee (l' \neq l) \\
|p| > l' \quad \text{IsCallTarget}(\text{p}[l']) \quad o' = 0}{\langle pc, \rho, \mu, stk, ms \rangle \xrightarrow[\text{DCall } (l', o')]{\text{OCall } l} \langle (l', o'), \rho, \mu, (pc + 1) :: stk, ms' \rangle} \\
\\
\text{IDEAL_CALL_FAULT} \frac{\text{p}[[pc]] = \text{call } e \quad \text{if } ms \text{ then } \&0 \text{ else } [[e]]_\rho = \&l \\
|p| \leq l' \vee \neg \text{IsCallTarget}(\text{p}[l']) \vee (o' \neq 0)}{\langle pc, \rho, \mu, stk, ms \rangle \xrightarrow[\text{DCall } (l', o')]{\text{OCall } l} \text{Fault}}
\end{array}$$

Figure 6: Ideal semantics for MINIMIR (selected rules, adapted from [1])

The ideal semantics (Figure 6) is an extension of the speculative semantics (additions in gold font) with protections mirroring those of the concrete SpecIBT transformation. Ideal execution is represented by $\frac{\mathcal{O}}{\mathcal{D}} \xrightarrow{i}$. Multistep ideal execution is represented by $\frac{\mathcal{O}}{\mathcal{D}} \xrightarrow{i}^*$. The ideal state configuration is the same as the speculative, but without the *ct* flag: $\langle pc, \rho, \mu, stk, ms \rangle$.

Ideal Transition Rules IDEAL_BRANCH, IDEAL_LOAD, and IDEAL_STORE are the same as their speculative counterparts, except that they enforce masking directly, using the *ms* flag. IDEAL_CALL and IDEAL_CALL_FAULT handle the cases where the attacker directive goes to a call target block or not. Instead of doing this with the **ctarget** instruction and the *ct* flag, as with the target program, the conditions under which a fault would occur or not are built directly into the transition rules as premises.

Misspeculation is tracked using the *ms* flag, because the source program does not use the *msf* register. Unlike in the speculative semantics, the *ms* flag in the ideal semantics influences

program execution by controlling when masking is enforced.

6 Relative Security for SpecIBT

[Theorem 1](#) states that, if two executions of the source program in the sequential semantics are indistinguishable based on leakage (2), then two executions of the target program in the speculative semantics will also be indistinguishable (4). We use $\mathbf{p} \vdash$ to denote that the semantics is parameterized by the program, so it is fixed for an execution. We use $\llbracket \mathbf{p} \rrbracket$ to denote a program transformed by SpecIBT.

Each source/target pair of executions must begin with related register states. The $\sim$ relation on registers in (3) is given in [Definition 6](#).

Some side conditions, collected in (1), must hold as well: the source program must be well-formed, as specified in §3, and it must not use the dedicated target registers *callee* and *msf* or the **ctarget** instruction. These conditions must hold throughout the proofs.

The remaining conditions are specific to proofs handling the program entry point. The *callee* register must indicate, for both target executions, that the program has been entered via a call to block 0, which must be designated as a call target block (notice also that the *ct* flag is set for both executions, as shown in (4)), and the *msf* register must contain 0, indicating that misspeculation is not yet occurring at the start of the program (notice also that the ideal *ms* flag is unset ($\perp$) in (4)).

Theorem 1 (Relative Security for SpecIBT (from [1])).

$$\left(\begin{array}{l} wf_prog(\mathbf{p}) \wedge callee, msf \notin VARS(\mathbf{p}) \wedge \\ IsCallTarget(\mathbf{p}[0]) \wedge \mathbf{ctarget} \notin \mathbf{p} \wedge \\ \rho'_1(callee) = \&0 \wedge \rho'_2(callee) = \&0 \wedge \\ \rho'_1(msf) = 0 \wedge \rho'_2(msf) = 0 \end{array} \right) \Rightarrow \quad (1)$$

$$\mathbf{p} \vdash \langle (0, 0), \rho_1, \mu_1, [] \rangle \approx \langle (0, 0), \rho_2, \mu_2, [] \rangle \Rightarrow \quad (2)$$

$$\rho_1 \sim \rho'_1 \wedge \rho_2 \sim \rho'_2 \Rightarrow \quad (3)$$

$$\llbracket \mathbf{p} \rrbracket \vdash \langle (0, 0), \rho'_1, \mu_1, [], \top, \perp \rangle \approx_s \langle (0, 0), \rho'_2, \mu_2, [], \top, \perp \rangle \quad (4)$$

The notion of indistinguishability is formalized as *observational equivalence*. Two program executions are observationally equivalent if, **given the same sequence of directives**, they emit the same observation trace, or if one of the observation traces is a prefix of the other. The $\approx$ and $\approx_s$ symbols are shorthand for sequential and speculative observational equivalence, respectively.

Definition 1 (Sequential Observational Equivalence (from [1])). Two sequential states $\langle s_1 \rangle$ and $\langle s_2 \rangle$ are observationally equivalent under the sequential semantics of program $\mathbf{p}$, written $\mathbf{p} \vdash \langle s_1 \rangle \approx \langle s_2 \rangle$ iff

$$\forall \mathcal{O}_1, \mathcal{O}_2. \mathbf{p} \vdash \langle s_1 \rangle \xrightarrow{\mathcal{O}_1^*} \cdot \wedge \mathbf{p} \vdash \langle s_2 \rangle \xrightarrow{\mathcal{O}_2^*} \cdot \Rightarrow \mathcal{O}_1 \leq \mathcal{O}_2$$

$\mathcal{O}_1 \leq \mathcal{O}_2$ denotes that, instead of the both lists being equal, $\mathcal{O}_1$ may be a prefix of $\mathcal{O}_2$ or vice versa. This is required because the two executions are not required to be of the same length. The use of the $\cdot$ symbol for successor states means that they don't matter enough to specify.

Definition 2 (Speculative Observational Equivalence (from [1])). Two speculative states $\langle s_1 \rangle$ and $\langle s_2 \rangle$ are observationally equivalent under the speculative semantics of program $\mathbf{p}$, written $\mathbf{p} \vdash \langle s_1 \rangle \approx_s \langle s_2 \rangle$ iff

$$\forall \mathcal{D}, \mathcal{O}_1, \mathcal{O}_2. \mathbf{p} \vdash \langle s_1 \rangle \xrightarrow[\mathcal{D}]{\mathcal{O}_1^*}_s \cdot \wedge \mathbf{p} \vdash \langle s_2 \rangle \xrightarrow[\mathcal{D}]{\mathcal{O}_2^*}_s \cdot \Rightarrow \mathcal{O}_1 \leq \mathcal{O}_2$$

We define observational equivalence in the ideal semantics, $\approx_i$, in an analogous way, with $\xrightarrow[\mathcal{D}]{\cdot}_s^*$ replaced by $\xrightarrow[\mathcal{D}]{\cdot}_i^*$.

6.1 Relative Security

To prove relative security, we first prove a backward compiler correctness result, showing equivalence between a speculative execution of the target and an ideal execution of the source.

Lemma 1 (Backwards compiler correctness, top-level (from [1])).

$$\left(\begin{array}{l} wf_prog(\mathbf{p}) \wedge callee, msf \notin VARS(\mathbf{p}) \wedge \\ IsCallTarget(\mathbf{p}[0]) \wedge \mathbf{c}target \notin \mathbf{p} \wedge \\ \rho'(callee) = \&0 \wedge \rho'(msf) = 0 \end{array} \right) \Rightarrow \quad (5)$$

$$\langle \mathbf{p} \rangle \vdash \langle (0, 0), \rho', \mu, [], \top, \perp \rangle \xrightarrow[\mathcal{D}]{\mathcal{O}}^* \quad \Rightarrow \quad (6)$$

$$\rho \sim \rho' \quad \Rightarrow \quad (7)$$

$$\mathbf{p} \vdash \langle (0, 0), \rho, \mu, [], \perp \rangle \xrightarrow[\mathcal{D}]{\mathcal{O}}^* \quad (8)$$

The goal of [Theorem 1](#), (4), unfolds to [Definition 2](#). The premises of [Definition 2](#) are target executions in the speculative semantics. We can introduce these as premises in [Theorem 1](#), and then apply [Lemma 1](#) to them. Since [Lemma 1](#) is an implication from a target execution in the speculative semantics (6) to a source execution in the ideal semantics (8), the application will result in the substitution of ideal observational equivalence for speculative in the goal of [Theorem 1](#).

This leaves us with a connection to make between the source program executing in the sequential semantics and the source program executing in the ideal semantics. Until misspeculation begins, these executions are the same, so observational equivalence in the sequential semantics will imply it in the ideal. After misspeculation begins, which, because the sequential semantics are deterministic, it will do at the same point in both executions, we use an *unwinding lemma* to establish that two ideal executions with the same directive sequence, program counter, and stack state will produce equivalent observation traces, due

to masking [1].

7 Backward Compiler Correctness

We now consider [Lemma 1](#) in depth. [§7.1](#) and [§7.1.1](#) give an overview of compiler correctness and simulations, respectively. [§7.2](#) presents the backward compiler correctness lemma, decomposing it into a top-level statement ([§7.2.1](#)); a more general statement proved by induction ([§7.2.2](#)); and a single-cycle lemma that establishes equivalence between multiple steps of the target program and a single step of the source program ([§7.2.3](#)).

7.1 Compiler Correctness

A correct compiler preserves the meaning of the source program, as determined by its observable behavior. A target program produced by a correct compiler will only produce behaviors that are a subset of source behaviors: this is known as *refinement*. Observable behavior includes divergence, normal termination, faulting, undefined behavior, and, of course, observation traces [2].⁴ We use a compiler correctness proof to establish behavioral equivalence between the **target program executing in the speculative semantics** and the **source program executing in the ideal semantics**. This is a somewhat off-label usage of compiler correctness, first introduced by Shivakumar et al. [12], where the *correctness* referred to is that of the SpecIBT transformation with respect to its specification in the ideal semantics.

7.1.1 Simulations

Simulations, a form of equivalence relation between programs, are commonly used to express compiler correctness. The strongest type of simulation, called a *bisimulation*, describes a source and target program that exhibit exactly the same behaviors.

⁴It typically also includes input and output, but MiniMIR does not have these.

Definition 3 (Bisimulation (from [2])). A source program s and target program t simulate each other iff

$$\forall B, t \Downarrow B \Leftrightarrow s \Downarrow B, \text{ where for } p \Downarrow B \text{ we say } \textit{program } p \textit{ exhibits behavior } B.$$

For each implication direction, the semantics in the premise is being *simulated*, and the semantics in the conclusion is doing the *simulating*.

This is often too strong, as some compilers might remove source behaviors (optimizations often do this), or add target behaviors, as discussed above. In these cases, it may be useful to limit the simulation to one direction of implication or the other. The directions are distinguished by whether the premise quantifies over all possible executions of the target (backward) or of the source (forward).

Definition 4 (Backward Simulation (from [2])). A source program s simulates a target program t iff

$$\forall B, t \Downarrow B \Rightarrow s \Downarrow B.$$

We use a backward simulation because it is required to be able to apply the backward compiler correctness result to the premises of [Equation 4](#) in [Theorem 1](#), as described in [§6.1](#).

Simulation Diagrams The typical structure of simulation proofs can be visualized with a simulation diagram. [Figure 7](#) shows a simulation where the target program takes one or more steps corresponding to a single step of the source. Once this is established for a single source step, it can be proved for the whole program by strong⁵ induction on the length of the target execution.

⁵The need for *strong* induction is due to the fact that we are using a single target cycle as the induction step, and when this is longer than 1 step, regular induction no longer suffices.

Let S be a source state, and T be a target state. Solid lines denote premises, and dashed lines denote conclusions. The same goes for arrows, which denote state transitions. The $+$ symbol denotes one or more steps of execution. The diagram below represents the statement

If $T \sim S$, and $T \xrightarrow{t^+} T'$, then $\exists S'$ such that $S \xrightarrow{t} S' \wedge T' \sim S'$.

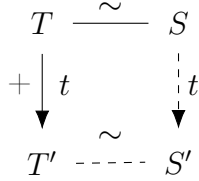


Figure 7: Simulation diagram for SpecIBT (adapted from [2])

- $T \sim S$: the state invariant $\sim$ holds for a pair of arbitrary source and target states
- $T \xrightarrow{t^+} T'$: T can take one or more steps to a successor state T' , possibly producing an observation trace t
- It must be shown that:

- $S \xrightarrow{t} S'$: S can take a single step to a successor state S' , producing trace t
- $T' \sim S'$: the state invariant holds for source and target successor states

We use this definition because SpecIBT sometimes adds instructions. We refer to $T \xrightarrow{t^+} T'$ as a *single cycle* of the target.⁶

Definition 5 (Single Cycle). The length of a single cycle for the target program depends upon which source instruction i we consider.

- If i is a **branch** whose condition is true, SpecIBT adds two instructions in the inter-

⁶Here and in all subsequent discussions of the compiler correctness proof, *source* is a shorthand for *source program executing in the speculative semantics*, and *target* is a shorthand for *target program executing in the speculative semantics*.

posing block, and the length of the single cycle is 3.

- If i is a **branch** whose condition is false, SpecIBT appends an instruction, and the length of the single cycle is 2.
- If i is a **call**, SpecIBT prepends an instruction and adds two additional instructions at the head of the call target block. The length of the single cycle is 4.
- Otherwise, the length of the single cycle is 1.

(Refer to [Figure 2](#) for the specific instructions added.)

State Invariant State invariants impose conditions on how program states are related when equality is too strong a constraint. The state invariant, in our setting, is mostly about handling synchronization between the source and target programs, given that SpecIBT may add instructions. This means that the program counters won't always match, and therefore the contents of stacks might not either. Synchronization is also necessary because the speculative semantics model the detection of misspeculation with the *msf* register, while the ideal semantics abstract this to the *ms*⁷ flag; these must be shown to be synchronized with respect to both the timing and the outcome of detection.⁸ Registers require a relation that isn't equality only because the target program uses the dedicated registers *msf* and *callee*, which the source must not use. Memory is assumed to be the same for each program.

Definition 6. The state invariant is defined as a relation $\sim$ parameterized by a source program and indexed by ideal and speculative state configurations, written as $\langle pc, \rho, \mu, stk, ms \rangle \sim \langle pc', \rho', \mu, stk', ct, ms \rangle$. Individual aspects of state are considered as follows:

- A source program counter pc and target program counter pc' are synchronized, written

$$pc \sim pc', \text{ iff}$$

⁷Both the speculative and the ideal semantics contain the *ms* flag; it only affects program state in the ideal semantics.

⁸The *msf* register uses numbers to represent the boolean values of the *ms* flag: 1 for true, 0 for false.

- Their labels are the same. (SpecIBT doesn't move instructions to different blocks.)
- The offset of pc' is obtained by:
 - * Adding 2 if the label of pc refers to a call target block. (SpecIBT adds two instructions to the head of each call target block.)
 - * Adding 1 for each **branch** or **call** instruction in the block prior to the instruction pointed to by $p[pc]$. (SpecIBT adds one instruction for each of these.)
- $\rho \sim \rho'$ iff $\rho = \rho'$, excepting the dedicated target registers msf and $callee$, which the source doesn't use, and $msf = 1$ if $ms = \top$, 0 if $\perp$.
- $stk \sim stk'$ iff stk and stk' contain the same number of program counters and these are pairwise synchronized.

7.2 The Proof

This section contains in-depth descriptions of the proofs of [Lemma 1 \(§7.2.1\)](#), [Lemma 2 \(§7.2.2\)](#), and [Lemma 3 \(§7.2.3\)](#).

7.2.1 Lemma 1

Before we begin, it is worth restating Lemma 1 here, for convenience:

SideConditions $\Rightarrow$

$$\langle p \rangle \vdash \langle (0, 0), \rho', \mu, [], \top, \perp \rangle \xrightarrow{\mathcal{O}_s^*} \Rightarrow$$

$$\rho \sim \rho' \Rightarrow$$

$$p \vdash \langle (0, 0), \rho, \mu, [], \perp \rangle \xrightarrow{\mathcal{O}_i^*}.$$

Proof of [Lemma 1](#). [Lemma 1](#) states that observational equivalence holds between target and

source executions starting from

$$\langle (0, 0), \rho', \mu, [], \top, \perp \rangle \text{ and } \langle (0, 0), \rho, \mu, [], \perp \rangle,$$

respectively, given the same sequence of directives for both, $\rho \sim \rho'$, and the collection of side conditions in (5).

Lemma 1 is not sufficiently general to be proved by induction. Lemma 2 is a more general statement of correctness, requiring the $\sim$ relation to hold between source and target starting state configurations, not just register states. We can't begin with it because source and target program counters are not synchronized at the program entry point: the target execution begins halfway through the usual single cycle (see Definition 5) for a `call` instruction. Once the target program has gone two steps, synchronization will hold for all aspects of program state and the simulation can begin. In this proof, we must show observational equivalence for target executions of length $n \leq 2$.

- If $n = 0$, the target execution consumes no directive and produces no observation, which the source can match by taking a reflexive step.
- If $n = 1$, we know that the target step was on a `ctarget` instruction, by (5) and from our knowledge of how SpecIBT transforms call target blocks (see §4). Therefore the SPEC_CTARGET rule applies (see Figure 5). Because it consumes no directive and produces no observation, the source once again preserves observational equivalence with a reflexive step.
- If $n = 2$, we know that the second target step was on the instruction $msf := (callee = \&0) ? msf : 1$, by our knowledge of how SpecIBT transforms call target blocks and because the previous instruction was `ctarget`. The SPEC_ASGN rule applies, updating the msf register with the outcome of the *callee* check and advancing the target program counter to $(0, 2)$.

A synchronization point (where [Definition 6](#) holds) has now been reached, with zero or more target steps remaining. We then apply [Lemma 2](#), the more general backward compiler correctness proof by induction, to handle the simulation. The more general state invariant is satisfied, because the only register assignment was to *msf*, and the target program counter is at (0, 2) to the source's (0, 0) in a call target block.

□

7.2.2 Induction proof

Lemma 2 (Backwards compiler correctness, general (from [1])).

$$\left(\begin{array}{l} wf_prog(p) \wedge \\ callee, msf \notin VARS(p) \wedge \\ ctarget \notin p \end{array} \right) \Rightarrow \quad (9)$$

$$\llbracket p \rrbracket \vdash \langle sc1 \rangle \xrightarrow[\mathcal{D}]{\mathcal{O}_s^*} \langle sc2 \rangle \quad \Rightarrow \quad (10)$$

$$\langle sc1 \rangle \sim \langle ic1 \rangle \quad \Rightarrow \quad (11)$$

$$p \vdash \langle ic1 \rangle \xrightarrow[\mathcal{D}]{\mathcal{O}_i^*} \langle ic2 \rangle \quad (12)$$

[Lemma 2](#) states that, given a target execution of n steps which consumes a sequence of attacker directives and emits a sequence of observations, a state invariant which holds on starting source and target states, and some side conditions, there exists a source execution which, consuming the same sequence of directives, emits the same observations.

Proof of [Lemma 2](#). We prove [Lemma 2](#) by strong induction on the length of the target execution, which we abbreviate as $LenTgt$. The base case, where $LenTgt = 0$, is easily satisfied by the source also taking zero steps. In both cases, the successor state is the start state, no directives are consumed, and no observations are produced. The rest of the proof assumes that $LenTgt > 0$.

The application of the induction hypothesis is only necessary if the remaining target execution is longer than a single step. Therefore, this proof **cases over the length of the target execution** ($LenTgt$) **relative to the length of a single cycle** ($LenCycle$). $LenCycle$ is determined by the number of target instructions added by SpecIBT, which is based on the source instruction, as specified in [Definition 5](#). We introduce a premise, $StepsInCycle$, that computes the length of a cycle accordingly; it is always between 1 and 4, inclusive.

It is also important to know *which* target instructions are added for which values of program counter so that we can link a speculative transition rule to each step in the target execution. This is defined through a *match* relation ([Definition 7](#)) on source and target instructions. A matching target instruction is the one that begins the target single cycle, but the match relation also specifies instructions for the rest of the cycle, with the exception of the `call` instruction, because we do not know statically whether the call will fault or not.

There are three possibilities for the inductive case: the target execution might be shorter than, the same length as, or longer than a single cycle. We consider each of these.

- If $LenTgt < LenCycle$, we must show that, for all partial target executions of length $< LenCycle$, observational equivalence holds between target and source. $LenCycle = 1$ is an impossible case because $LenTgt > 0$. This rules out all instructions except `branch` and `call`.

- Instruction is `branch`: $LenTgt$ is known to be nonzero and also strictly less than $LenCycle$; therefore we consider partial executions of length 1 and 2 if the branch is taken, and of length 1 if the branch is not taken.

- By [Definition 7](#), we know that the first target step uses the `SPEC_BRANCH` transition rule, so we can also assume its premises (see [Figure 5](#)).
- These are similar enough to the premises of `IDEAL_BRANCH` (see [Figure 6](#)) that we can infer it as the ideal transition rule; the only difference is in the usage of

msf vs ms , and we know by the state invariant (see [Definition 6](#)) that these are aligned.

- The observation generated by each **branch** step, given the same directive, is the same: (*OBranch* b , where b is the sequential value of the branch condition).
- The only case left to consider is if the branch is taken and the target program goes one more step. By the match relation, we know the transition rule used will be SPEC_ASGN. This does not produce an observation; therefore observational equivalence is preserved for all possible partial executions of the target.
- Instruction is **call**: $LenTgt$ is known to be nonzero and also strictly less than $LenCycle$; therefore we consider partial executions of length 1, 2, and 3.
 - $LenTgt = 1$: by [Definition 7](#), the SPEC_ASGN transition rule applies, with state updates $pc' \rightarrow pc' + 1$ and $\rho' \rightarrow [callee \mapsto (msf = 1) ? 0 : e]\rho'$. No observation is emitted, so a reflexive source step preserves equivalence.
 - $LenTgt = 2$ or 3 : by [Definition 7](#), the SPEC_CALL rule applies. We have to consider both the second and third target step here, because whether a fault occurs depends on the attacker directive, which is consumed in the SPEC_CALL step. But the fault itself does not occur until the third step, which is either SPEC_CTARGET or SPEC_FAULT, depending on whether or not a **ctarget** instruction is encountered.

Because the source program does not use the **ctarget** instruction, the premises for its IDEAL_CALL and IDEAL_CALL_FAULT rules have to express the same conditions for faulting or not by referring to the attacker directive. A fault occurs if any of the following are the case:

- * The attacker directive's label is out of bounds with respect to the source

program. For the target program, this ensures that a `ctarget` instruction was not encountered, because the added blocks are not call target blocks (see Figure 2), and the only other possibility is that the directive was out of bounds with respect to the target program as well.

- * The attacker directive's label does not refer to a call target block. This also ensures that a `ctarget` instruction was not encountered by the target program, because SpecIBT only adds `ctarget` to the head of call target blocks.
- * The offset is nonzero. `ctarget` is only added at offset zero, so this also ensures that the target does not encounter a `ctarget`.

Therefore, the source and target will fault under the same circumstances. The only other requirement for observational equivalence is the observation emitted, which in the case of `SPEC_CALL`, `IDEAL_CALL`, and `IDEAL_CALL_FAULT` is the sequentially intended call target location. Because masking proceeds in lock-step given $msf \sim ms$, either both expressions will be masked or neither of them will be. Therefore observational equivalence holds for partial target cycles of 2 and 3 steps. Finally, the remaining premises of the ideal transition rules are satisfied by those of the speculative transition rules, by converting msf to ms .

– If $LenTgt = LenCycle$, we have a target execution of between 1 and 4 steps, and we must show that the source can execute a single step that preserves observational equivalence. We case over the possible types of final target state: running, undefined, fault, or normal termination.⁹

- **Running state:** We apply Lemma 3, introducing (16) as a premise stating that the source takes a single step, producing the same observation trace as the target and also

⁹The state `Undef` is an artifact of the testing process.

ending in a running state. Since this is also the proof goal, we are done.

- **Undefined state:** This is an impossible case because no transition rule in the speculative semantics ends in an explicitly specified undefined state.
- **Fault state:** This is also an impossible case because the only speculative transition rule where the successor state is a fault is `SPEC_FAULT`. But if this occurs, the target execution is over in 3 steps, but a single cycle for `call` is 4 steps, and it is assumed that the target execution is the length of a single cycle.
- **Normal termination:** This is an impossible case unless the instruction is `ret` and the stack is empty, in which case the transition rule `SPEC_TERM` applies. The source can take a corresponding step.¹⁰ Because it produces the same observation trace as the target and also ends in `Term`, we are done.

– If $LenTgt > LenCycle$, we can apply [Lemma 3](#) to prove a single step of the simulation, and the induction hypothesis to prove the rest. We therefore split the target execution into two parts, where the first has length $LenCycle$, and the second has length $LenTgt - LenCycle$. The sequences of directives and observations are also split accordingly. That such a split is possible is proven by induction on the length of a single cycle.

We have the necessary conditions to apply [Lemma 3](#), as its premises are those of [Lemma 2](#) (except *StepsInCycle*, which was introduced earlier). Applying it gives us the conjunction (16) as a premise, the right conjunct of which satisfies [Definition 6](#) as required by the induction hypothesis, since the successor states of the single step are the start states of the rest of the execution. Now we have the necessary conditions to apply the induction hypothesis.

Induction hypothesis The induction hypothesis states that [Lemma 2](#) is the case for all target executions of length m , where $m \leq n$ and $n = LenTgt - 1$. The target execution

¹⁰We do not show this rule in [Figure 6](#), but it applies under the same conditions as `SPEC_TERM`.

we're considering has length $LenTgt - LenCycle$, so the inequality to be satisfied once the induction hypothesis has been applied becomes $(LenTgt - LenCycle) \leq (LenTgt - 1)$. It is satisfied, because we know by (13) (*StepsInCycle*) that $LenCycle$ is, at minimum, 1. (This demonstrates why we *need* strong induction in the first place—recall that $LenCycle$ is anywhere from 1 to 4, and regular induction would only allow us to prove a result for single cycles of length 1.)

The application of the induction hypothesis gives us the premise that, for any target execution is longer than a single cycle, there is a corresponding source execution that preserves observational equivalence. This is also the proof goal (12), so we are done.

□

7.2.3 Single cycle proof

Lemma 3 (Backwards compiler correctness, single cycle (adapted from [1])).

$$\left(\begin{array}{l} wf_prog(p) \wedge \\ callee, msf \notin VARS(p) \wedge \\ ctarget \notin p \wedge StepsInCycle(p, ic1, \mathcal{D}) = n \end{array} \right) \Rightarrow \quad (13)$$

$$\llbracket p \rrbracket \vdash \langle sc1 \rangle \xrightarrow[\mathcal{D}]{\mathcal{O}_s^n} \langle sc2 \rangle \quad \Rightarrow \quad (14)$$

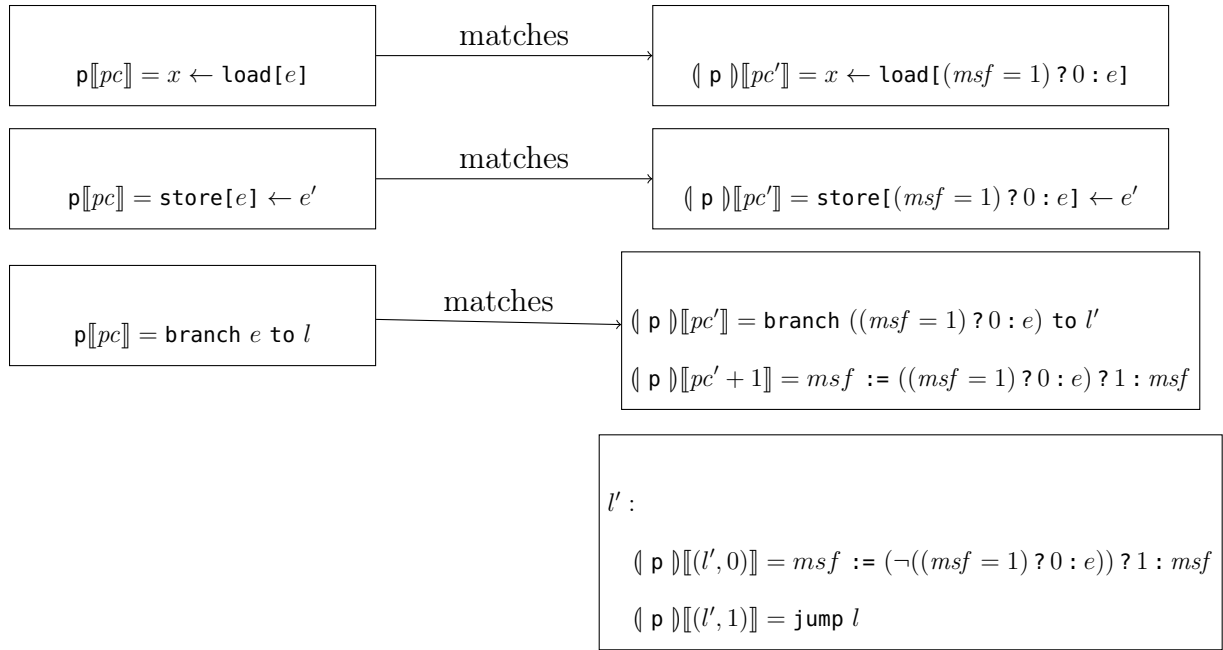
$$\langle sc1 \rangle \sim \langle ic1 \rangle \quad \Rightarrow \quad (15)$$

$$p \vdash \langle ic1 \rangle \xrightarrow[\mathcal{D}]{\mathcal{O}_i} \langle ic2 \rangle \wedge \langle sc2 \rangle \sim \langle ic2 \rangle \quad (16)$$

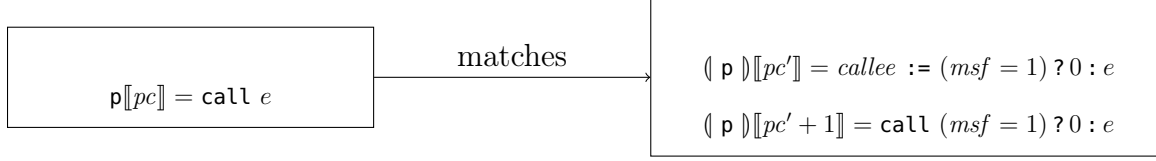
Lemma 3 states that, given a target execution whose length is a single cycle, and given that the state invariant (see Definition 6) holds between target and source starting states, we must show that the source can take a single step such that observational equivalence (see Definition 2) holds between target and source, and the state invariant is preserved in target and source successor states. The usual side conditions are also assumed.

Proof of Lemma 3. In order to reason about a single cycle (see Definition 5) of target execution, we must first establish which target instructions get executed for a given source instruction. Knowing by Equation 15 that starting target and source program counters are synchronized according to Definition 6, we define a *match relation* that defines which target instruction begins a single cycle (this instruction is said to *match* the corresponding source instruction). It further defines, for **branch** and **call**, which extra instructions are added for which values of target program counter.

Definition 7 (Match Relation). Given source and target program counters pc and pc' , where $pc \sim pc'$, and $\mathbf{p} \llbracket pc \rrbracket = i$ means instruction i is fetched from program $\mathbf{p}$ with program counter pc :

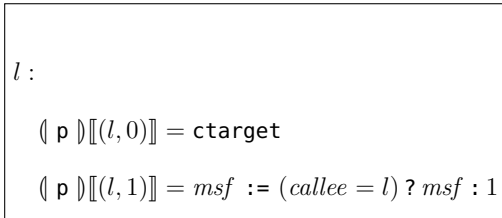


where, supposing $\mathbf{p}$ has n blocks total and **branch** e to l is the m^{th} **branch** instruction in $\mathbf{p}$, $l' = (n - 1) + m$.



Otherwise, $\forall i, p[pc] = i$ matches $\langle p \rangle[pc'] = i$.

Note: for `call` instructions, we also expect the following two target instructions at the head of any call target block with label l :



However, the match relation does *not* specify this, because it is only one of two possible outcomes. The attacker might jump to a location that isn't the beginning of a call target block, in which case a fault will occur. This distinction is handled in the proofs.

It is worth reiterating the assumptions we have collected thus far:

- The state invariant ([Definition 6](#)) holds between source and target start states: program counters, registers, memories, the stack, and between the speculative msf register and the ideal ms flag ([15](#)).
- The matching relation ([Definition 7](#)) holds between a given source instruction and the corresponding target single cycle, showing which instructions are executed by the target, and therefore which speculative transition rules are used.
- The target executes for the length of a single cycle ([13](#)).
- Side conditions hold as detailed in ([14](#)). The target execution length is the same as that of a single cycle.

We proceed by casing over source instruction type. For each source instruction, we infer the instructions in the target *single cycle*, using [Definition 7](#). Given the premises above, we must show that the ideal step on the source instruction is such that observational equivalence holds for source and target executions, and the state invariant ([Definition 6](#)) holds for source and target successor states.

In each case, we execute the target, keeping track of the premises of each step, which we can also assume; the state updates specified by each transition rule, which will define the target successor state; the type of state the target execution ends in; and the observation trace emitted by the target steps. This information is used to infer the source step and justify it (target premises tend to be similar to source premises, for matching steps, with a few exceptions), and to evaluate whether the state invariant holds at the end.

Branch We must show that the single step of the source and single cycle of the target, as specified by [Definition 7](#), are observationally equivalent and preserve the state invariant. We make case distinctions on whether or not misspeculation is already occurring, and if it isn't, whether or not the attacker's choice of branch condition differs from the sequential branch condition (if it does, misspeculation is initiated).¹¹ There are six subcases that arise from this, shown in [Table 1](#). Note that *AB* stands for *attacker branch*, and *SB* for *sequential branch*.¹²

[Table 1](#) shows the six possible subcases for a target branch instruction executing speculatively, and summarizes the state changes that occur in the target program during each possible cycle. The case where the attacker chooses to take the branch and misspeculation is already occurring is presented in more detail as an example.

¹¹Recall that misspeculation is *monotonic*; once *msf* and *ms* are set, they remain set.

¹²A reminder that the *msf* register has value 0 if unset and 1 if set, corresponding to the ideal *ms* flag's $\perp$ and $\top$ values, respectively.

	AB taken	AB not taken
$msf = 1$	leak: <i>false</i> (masked) msf (already) set	leak: <i>false</i> (masked) msf (already) set
$msf = 0 \wedge SB$ taken	leak: <i>true</i> msf not set	leak: <i>true</i> msf set
$msf = 0 \wedge SB$ not taken	leak: <i>false</i> msf set	leak: <i>false</i> msf not set

Table 1: Branch subcases

Example 3 (Branch subcase). [Table 2](#) walks through the 3-step target execution where the attacker chooses to take the branch and misspeculation is already occurring. We must prove observational equivalence between this series of steps and the step taken on the source instruction **branch** e **to** l in the ideal semantics, and that the state invariant holds for the resulting source and target states.

PC	Instruction	Transition rule	Effects on State
$pc' \sim pc$	branch $((msf = 1) ? 0 : e)$ to l'	SPEC_BRANCH	$OBranch$ false
$(l', 0)$	$msf := \neg 0 ? 1 : msf$	SPEC_ASGN	$[msf \mapsto 1]\rho'$
$(l', 1)$	jump l	SPEC_JUMP	$pc' = (l, 0)$

Table 2: Target single cycle, attacker takes branch, already misspeculating

The target program start state is $\langle pc', \rho', \mu, stk', \perp, \top \rangle$. Since $msf \sim ms$, by [Definition 6](#), $\rho'(msf) = 1$. Hence, due to masking, the branch condition evaluates to 0 in [Table 2](#). Had misspeculation not already been occurring, it would have been detected in the next step. As it is, setting msf to 1 is redundant. The **jump** brings the target execution to the synchronization point.

Target state at the end of the single cycle is $\langle (l, 0), [msf \mapsto 1]\rho, \mu, stk, \perp, \top \rangle$, and the observation emitted is $OBranch$ **false**.

We can construct a corresponding source step in the ideal semantics. Because the source instruction is **branch** e **to** l , the IDEAL_BRANCH transition rule applies; however, it is

necessary to have all the premises. These are:

- $p\llbracket pc \rrbracket = \text{branch } e \text{ to } l$: true, by the case we're in.
- if ms then 0 else $\llbracket e \rrbracket_\rho = n \wedge (n \neq 0) = b$, where b = the branch condition. Given that $msf \sim ms$, this is also true: $ms = \top$, so $n = 0$ and $b = \text{false}$.
- $pc' = \text{if } b' \text{ then } (l, 0) \text{ else } pc + 1$, where b' is the attacker directive: identical to its speculative counterpart
- $ms' = ms \vee (b \neq b')$: identical to its speculative counterpart

Since we have all the premises, we can apply `IDEAL_BRANCH`. The resulting ideal configuration is $\langle (l, 0), \rho, \mu, stk, \top \rangle$, and the observation emitted is *OBranch false*. By [Definition 6](#), $\langle (l, 0), \rho, \mu, stk, \top \rangle \sim \langle (l, 0), [msf \mapsto 1]\rho, \mu, stk, \perp, \top \rangle$, and the observations are the same for both executions. This is what we needed to show to prove this subcase.

The other five cases proceed much like this one.

Call The source instruction is `call e`. We must show that the single step of the source and single cycle of the target, as specified by [Definition 7](#), are observationally equivalent (including whether a fault occurs under the same circumstances for each) and preserve the state invariant.

By *StepsInCycle*, the number of steps in the target single cycle is 4, and by [Equation 14](#), the target execution takes 4 steps. By [Definition 7](#), the target steps we can assume are shown in [Table 3](#). By the premises of `SPEC_CALL`, we know that e evaluates to $\&l$.

Given attacker directive $D\text{Call}(l'', o'')$:

The target program start state is $\langle pc', \rho', \mu, stk', \perp, ms \rangle$. We can assume the premises of `SPEC_ASGN` and `SPEC_CALL` (see [Figure 5](#)).

PC	Instruction	Transition rule	Effects on State
$pc' \sim pc$	$callee := (msf = 1) ? 0 : e$	SPEC_ASGN	$[callee \mapsto (msf = 1) ? 0 : e]\rho'$
$pc' + 1$	$call ((msf = 1) ? 0 : e)$	SPEC_CALL	$stk' \rightarrow (pc' + 2) :: stk' \wedge$ $ct = \top \wedge$ $OCall((msf = 1) ? 0 : e) \wedge$ $ms = ms \vee (l \neq l'' \vee o'' \neq 0)$

Table 3: Target single cycle, call, first two steps

By the end of the second step, the target program state is

$$\langle (l'', o''), [callee \mapsto (msf = 1) ? 0 : e]\rho', \mu, (pc' + 2) :: stk', \top, ms \vee (l \neq l'' \vee o'' \neq 0),$$

and observation $OCall((msf = 1) ? 0 : e)$ has been emitted.

If $DCall(l'', o'')$ goes to the head of a call target block, the next instruction is **ctarget**. Otherwise, execution faults. However, by Equation 14 and *StepsInCycle*, the target single cycle for **call** is 4 steps. Therefore, faulting is an impossible case and $DCall(l'', o'')$ must go to the head of a call target block. By Figure 2b, the first two instructions are **ctarget** and $msf := (callee = l'') ? msf : 1$. The transition rules and effects on state are shown in Table 4.

PC	Instruction	Transition rule	Effects on State
$(l'', 0)$	ctarget	SPEC_CTARGET	$ct = \perp$
$(l'', 1)$	$msf := (callee = l'') ? msf : 1$	SPEC_ASGN	$[msf \mapsto (callee = l'') ? msf : 1]\rho'$

Table 4: Target single cycle, call, third and fourth steps

SpecIBT detects misspeculation on the *wrong* call target block in the fourth step of target execution. By the end of the single cycle, at the synchronization point, the target program state is

$$\langle (l'', 2), [msf \mapsto (callee = l'') ? msf : 1][callee \mapsto (msf = 1) ? 0 : e]\rho', \\ \mu, (pc' + 2) :: stk', \perp, ms \vee (pc' \neq (l'', o'')) \rangle.$$

Now we must show that the source program, given source instruction **call** e , can take a step

in the ideal semantics, such that observational equivalence holds, and the state invariant is preserved when a fault does not occur.

Observational equivalence holds with respect to faulting behavior By `SPEC_FAULT`, the target faults whenever $ct = \top \wedge (\llbracket \mathbf{p} \rrbracket(l'', o'') \neq \mathbf{ctarget})$. The source does not use the `ctarget` instruction in this way (it is a no-op if it happens to appear), and the ideal semantics do not contain the ct flag. Instead, the premises of `IDEAL_CALL_FAULT` enforce that a fault occurs for any one of three reasons: (1) $l'' \geq |\mathbf{p}|$, or (2) $\neg \text{IsCallTarget}(\mathbf{p}[l''])$, or (3) $o'' \neq 0$. These are the same conditions under which a `ctarget` instruction placed by `SpecIBT` will *not* be encountered (see [Figure 2b](#)).

`IDEAL_CALL` specifies that calls will succeed under the logical negation of the conditions specified by `IDEAL_CALL_FAULT` (see [Figure 6](#)). It would seem that this ensures that source and target will fault, or not, under the same conditions. However, there is a small complication. The length of the source program is less than or equal to the target program ($|\mathbf{p}| \leq |\llbracket \mathbf{p} \rrbracket|$). If the source program contains `branch` instructions, its length is strictly less than that of the target program. It is reasonable to wonder whether a call that is in-bounds for the target is also in-bounds for the source, and whether a call target block in the target is also a call target block in the source. Because the `SpecIBT` transformation does not alter either the index or the call target annotation of existing blocks, but only appends new branch target blocks (see [Figure 2a](#)), the conditions under which a fault occurs remain the same for both source and target, even though their lengths are not necessarily the same. Observational equivalence holds with respect to the type of successor state reached.

Source Execution When the source program doesn't fault, the `IDEAL_CALL` transition rule is used. The source start state is $\langle pc, \rho, \mu, stk, ms \rangle$. The premises of this rule are goals to be satisfied before it can be applied. For attacker directive $D\text{Call}(l'', o'')$:

- $|p| > l''$
- $IsCallTarget(p[l''])$
- $o'' = 0$
- $p[[pc]] = \text{call } e$
- if ms then $\&0$ else $\llbracket e \rrbracket_\rho = \&l$
- $ms = ms \vee (l \neq l'')$

Given that the target execution completed a single cycle without faulting, and given the discussion in the previous section, we know that (l'', o'') satisfies the first three requirements. We already have a premise that satisfies the fourth requirement; otherwise we would not be in the **call** case.

The last two goals are premises of SPEC_CALL.

- SPEC_CALL premise: $\llbracket e \rrbracket_\rho = \&l$, where $e = (msf = 1) ? \&0 : e$
IDEAL_CALL goal: if ms then $\&0$ else $\llbracket e \rrbracket_\rho = \&l$
- SPEC_CALL premise: $ms = ms \vee ((l'', o'') \neq (l, 0))$
IDEAL_CALL goal: $ms = ms \vee (l'' \neq l)$

In the first case, the premise and the goal are essentially identical because, by [Definition 6](#), $msf \sim ms$. In the second case, we already know $o'' = 0$, because the target didn't fault (remember they both use the same directive), so the premise and goal are the same. Therefore, we can assume that the source can take a step as defined by IDEAL_CALL, and shown in [Table 5](#).

PC	Instruction	Transition rule	Effects on State
pc	call e	IDEAL_CALL	if ms then $\&0$ else $\llbracket e \rrbracket_\rho \wedge ms = ms \vee (l'' \neq l)$

Table 5: Source call

Given attacker directive $DCall(l'', o'')$:

The source successor state is $\langle (l'', 0), \rho, \mu, (pc + 1) :: stk, ms \vee (l'' \neq l) \rangle$.

Observational Equivalence We must show that the source and target executions emit the same observation. This will apply for both faulting and non-faulting executions, because the observation is emitted before the fault takes place. The speculative semantics (see [Figure 5](#)) dictate that the observation emitted by the target program will be of l if the msf register was already set in the starting state, and of $\&0$ if it wasn't. In the source program, the ideal semantics (see [Figure 6](#)) dictate that masking of l occurs if the ideal ms flag is set in the starting state. By [Definition 6](#), these will always be in the same state in the starting state, so the observations emitted will also always be the same.

State Invariant Preserved We must show that the state invariant is preserved in source and target executions that do not fault. The target successor state is

$\langle (l'', 2), [msf \mapsto (callee = l'') ? msf : 1][callee \mapsto (msf = 1) ? 0 : e]\rho', \mu, (pc' + 2) :: stk', \perp, ms \vee (pc' \neq (l'', o'')) \rangle$.

The source successor state is $\langle (l'', 0), \rho, \mu, (pc + 1) :: stk, ms \vee (l'' \neq l) \rangle$.

- $(l'', 0) \sim (l'', 2)$: synchronized as per [Definition 6](#), since we are at a call target block.
- $\rho \sim [msf \mapsto (callee = l'') ? msf : 1][callee \mapsto (msf = 1) ? 0 : e]\rho'$: synchronized, since the only updates have been to msf and $callee$, which are excluded from the relation, and $\rho \sim \rho'$ by [\(15\)](#).
- $(pc + 1) :: stk \sim (pc' + 2) :: stk'$: $(pc + 1) \sim (pc' + 2)$, because one instruction is added to the target when the source instruction is a `call`. By [\(15\)](#), $stk \sim stk'$.
- $ms \sim msf$: given a non-faulting execution path, speculation is detected in each semantics on the basis of whether $l'' = l$. We know that $ms \sim msf$ at the start, so it must

be the case that $ms \sim msf$ in the successor states.

Therefore, the state invariant is preserved in source and target executions that do not fault.

□

8 Related Work

Compiler correctness and ideal semantics Our overall proof structure, and especially the use of the ideal semantics as a specification for our transformation, owes much to the work of Shivakumar et al. [12]. Differences are as expected, given that they are seeking to protect only cryptographic programs in a source language that uses an information flow control type system to enable proofs that source programs are sequentially free of leaks. For example, their security definition, *relative noninterference*, assumes the distinction between secret and public data, whereas we seek to protect programs that make no such distinction. Their transformation, like ours, has speculative load hardening as a basis, but their variant applies masking under more limited conditions than ours, because the source programs they consider are already relatively well-protected. They have formal proofs of the correctness of their transformation, but not mechanized proofs.

From sections 2 and 3 of the work of Xavier Leroy [2], we adopt the notions of semantic preservation, simulation relation, and safety preservation, the backward compiler correctness proof technique, and (adapted) simulation diagram.

Relative security Our use of relative security as a suitable security definition for programs that might be sequentially leaky comes from the work of Baumann et al. [10]. The definition states that the target program, executing speculatively, should not leak more than the source program, executing sequentially. This is appropriate for a compiler transformation on a program whose source code might contain nonspeculative leaks, which are not in scope for our mitigation. Previous work has used the name "relative security", but not in conjunction with this definition.

9 Conclusion

This thesis has described the role played by compiler correctness in formal proofs of relative security. The ideal semantics define a specification for the SpecIBT transformation such that source programs executing ideally should enjoy the same protections against speculative attack as transformed programs executing in the speculative semantics. We introduced the backward compiler correctness technique used to prove equivalence for such executions, including the idea of a simulation diagram describing the method of proof for a single step of simulation between target and source. We then examined the nested proofs in great detail, from the top-level statement covering the first two steps of the target program to a simulation point from the program entry point, to the more general lemma statement, to the single-cycle statement whose proof follows the simulation diagram.

Security, correctness, and safety are not only important, but subtle, and proving these properties is complex and difficult. Publications with an audience of experts tend to elide many details, dispatching proofs like the one considered here in a single paragraph. This document might serve as a reminder of the level of complexity being abstracted away by high-level sketches, and a justification, if one is needed, for the continued drive toward mechanized specifications and proofs, and the increased certainty that they provide.

10 References

- [1] J. Baumann, Y. Kim, Y. Farba, C. Hritcu, and J. Leatherman-Brooks, “SpecIBT: Formally verified protection against speculative control-flow hijacking,” 2026. [Online]. Available: <https://arxiv.org/abs/2601.22978v1>
- [2] X. Leroy, “A formally verified compiler back-end,” *Journal of Automated Reasoning*, vol. 43, no. 4, pp. 363–446, 2009. [Online]. Available: <http://xavierleroy.org/publi/compcert-backend.pdf>
- [3] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, “Spectre attacks: Exploiting speculative execution,” in *2019 IEEE Symposium on Security and Privacy, SP*. IEEE, 2019, pp. 1–19. [Online]. Available: <https://doi.org/10.1109/SP.2019.00002>
- [4] Y. Yarom, *Microarchitectural Attacks*. Wiley-ISTE, January 2025.
- [5] B. Amos, N. Gilboa, and A. Levy, “Spectre without shared memory,” in *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, ser. SAC ’19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1944–1951. [Online]. Available: <https://doi.org/10.1145/3297280.3297470>
- [6] Z. Zhang, G. Barthe, C. Chuengsatiansup, P. Schwabe, and Y. Yarom, “Ultimate SLH: Taking speculative load hardening to the next level,” in *32nd USENIX Security Symposium*, J. A. Calandrino and C. Troncoso, Eds. USENIX Association, 2023, pp. 7125–7142. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity23/presentation/zhang-zhiyuan-slh>
- [7] C. Carruth, “Speculative Load Hardening: A Spectre variant #1 mitigation technique,” LLVM Reference, Sep. 2018. [Online]. Available: <https://llvm.org/docs/SpeculativeLoadHardening.html>
- [8] Intel Corporation, “Intel® 64 and IA-32 architectures software developer’s manual: Volume 1: Basic architecture: Chapter 18 Control-Flow Enforcement Technology (CET),” 2025. [Online]. Available: <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html>
- [9] J. Baumann, Y. Kim, Y. Farba, C. Hritcu, and J. Leatherman-Brooks, “TrioSecuris: Formally verified protection against speculative control-flow hijacking,” To appear at CSF’26, Jul. 2026. [Online]. Available: <https://doi.org/10.48550/arXiv.2601.22978>
- [10] J. Baumann, R. Blanco, L. Ducruet, S. Harwig, and C. Hritcu, “FSLH: flexible mechanized speculative load hardening,” in *38th IEEE Computer Security Foundations Symposium, CSF 2025, Santa Cruz, CA, USA, June 16-20, 2025*. IEEE, 2025, pp. 569–584. [Online]. Available: <https://doi.org/10.1109/CSF64896.2025.00023>
- [11] B. Dongol, M. Griffin, A. Popescu, and J. Wright, “Relative security: Formally modeling and (dis)proving resilience against semantic optimization vulnerabilities,”

- in *37th IEEE Computer Security Foundations Symposium, CSF*. IEEE, 2024, pp. 403–418. [Online]. Available: <https://doi.org/10.1109/CSF61375.2024.00027>
- [12] B. A. Shivakumar, J. Barnes, G. Barthe, S. Cauligi, C. Chuengsatiansup, D. Genkin, S. O’Connell, P. Schwabe, R. Q. Sim, and Y. Yarom, “Spectre declassified: Reading from the right place at the wrong time,” in *44th IEEE Symposium on Security and Privacy, SP*. IEEE, 2023, pp. 1753–1770. [Online]. Available: <https://doi.org/10.1109/SP46215.2023.10179355>
- [13] Y. Yarom and K. Falkner, “FLUSH+RELOAD: A high resolution, low noise, l3 cache Side-Channel attack,” in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 719–732. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom>
- [14] C. Hritcu, *Software Foundations: Security Foundations*. Draft textbook volume, Oct. 2025. [Online]. Available: <https://mpi-sp-foe-2025.github.io/book-secf>
- [15] M. Schneider, D. Lain, I. Puddu, N. Dutly, and S. Capkun, “Breaking bad: How compilers break constant-time implementations,” To appear at AsiaCCS, 2025. [Online]. Available: <https://dl.acm.org/doi/10.1145/3708821.3733909>

Appendix A Supplemental File Information

The following supplemental files accompany this thesis.

Artifact

The Rocq development containing the compiler correctness lemma and surrounding work described by this thesis.

File type: `.zip`, `.v`, `.md`

File name: `specibt.zip`

File size: 82 KB

Required software: See `README.md` for required software and build instructions.