Compiler correctness and Spectre mitigations

Julay Leatherman-Brooks

June 12, 2026



MAX PLANCK INSTITUTE
FOR SECURITY AND PRIVACY

July-December 2025

SpecIBT: Formally Verified Protection Against Speculative Control-Flow Hijacking

Jonathan Baumann[1‡] Yonghyun Kim[1‡] Yan Farba[1,2] Cătălin Hriţcu[1] Julay Leatherman-Brooks[1,3]

[1]MPI-SP, Bochum, Germany [2]Ruhr University Bochum, Germany [3]Portland State University, USA

Accepted at CSF '26!

The work was extended after I left, to cover RSB attacks, and is now known as Triosecuris.

Introduction

Motivation



Introduced in 2018

Kocher et. al., Jann Horn at Google Project Zero

Speculative side channel attack

- Control-flow misprediction
- Access secret
- Leak secret to cache (and infer its value with cache attack)

Family of attacks

Many variants, secure hardware needed

(Image from meltdownattack.com)

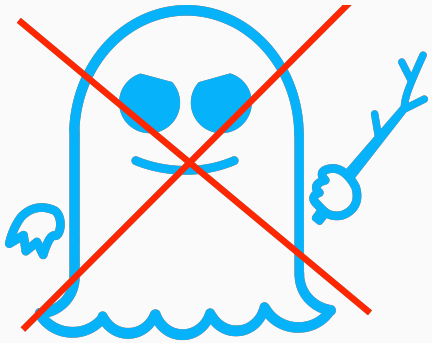
Spectre Variants

```
if (x < array1_size)
    y = array2[array1[x] * 256];
```

Spectre PHT: Pattern History Table (predicts outcome of conditional branches)

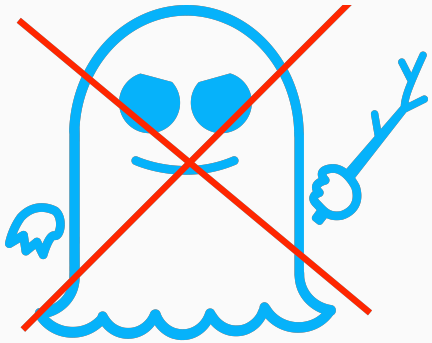
Spectre BTB: Branch Target Buffer (predicts location of indirect branches and calls)

SpecIBT



SPECTRE

PHT



SPECTRE

BTB



ROCQ

SpecIBT

Relative Security

p: source program

$(\!|\ p\ |\!)$: program transformed by SpecIBT

Relative security: $(\!|\ p\ |\!)$ does not leak more in the presence of speculation than p does in its absence.

Observational Equivalence

Absence of leaks: ***observational equivalence***.

Program states s_1 and s_2 are observationally equivalent if two program executions from s_1 and s_2 have the same output, so an observer comparing them could infer nothing about any sensitive data handled by the program.

Relative Security Again

Relative security: if two runs of p are observationally equivalent in the absence of speculation, then two runs of $(\!|\, \mathsf{p} \,|\!)$ are observationally equivalent in the presence of speculation.

Goal

A key component of the relative security proof is a compiler correctness lemma relating the concrete SpecIBT transformation to its abstract specification.

The goal of this thesis is to describe the statement and proof of this lemma, after establishing sufficient context.

Speculative Attacks and Defenses

Bounds Check: Sequential vs. Speculative

`branch` $(i < a_len)$ `to` L

$(i < a_len) \rightarrow$ branch taken

$(i \geq a_len) \rightarrow$ branch not taken

`branch ??? to` $L \rightarrow$ branch prediction, speculative execution, rollback

Incorrect prediction $\rightarrow$ ***sequentially impossible execution***

When it's an attack: ***misspeculation***

$(i \geq a_len) \wedge$ branch taken

Program vulnerable to Spectre PHT attack

```
branch (i < a_len) to L        mistrain predictor
...
```

```
L :
  r1 ← load[a + i]             OOB read (secret)
  r2 ← load[b + (r1 * 512)]    leak
```

Ultimate Speculative Load Hardening

Ultimate Speculative Load Hardening (USLH)[1] is a software-based mitigation on which SpecIBT is built. On its own, it protects against Spectre PHT attacks.

key element: ***branchless conditional expression***

Syntax: $b\,?\,e1\,:\,e2$

msf register to track misspeculation

[1] Zhang et. al., 2023

How USLH Detects Misspeculation

$\texttt{branch}\ (i < a_len)\ \texttt{to}\ L'$

$msf := (i < a_len)\ ?\ 1 : msf$

...

L' :

$msf := \neg((i < a_len)\ ?\ 1 : msf)$

$\texttt{jump}\ L$

L :

$r1 \leftarrow \texttt{load}[a + i]$

$r2 \leftarrow \texttt{load}[b + (512 * r1)]$

How USLH Mitigates Misspeculation

$$\texttt{branch}\ (i < a_len)\ \texttt{to}\ L'$$
$$msf := (i < a_len)\ ?\ 1 : msf$$
$$\ldots$$

$$L' :$$
$$msf := \neg((i < a_len)\ ?\ 1 : msf)$$
$$\texttt{jump}\ L$$

$$L :$$
$$r1 \leftarrow \texttt{load}[(msf = 1)\ ?\ 0 : a + i]$$
$$r2 \leftarrow \texttt{load}[(msf = 1)\ ?\ 0 : b + (512 * r1)]$$

Program vulnerable to Spectre BTB

$calln$: `branch` $(i < a_len)$ `to` $L_\top$

$fun := \&fun_1$

`jump` L_{cont}

$L_\top$: $fun := \&fun_2$

`jump` L_{cont}

L_{cont} : `call` fun

$\vdots$

fun_1 : $\ldots$

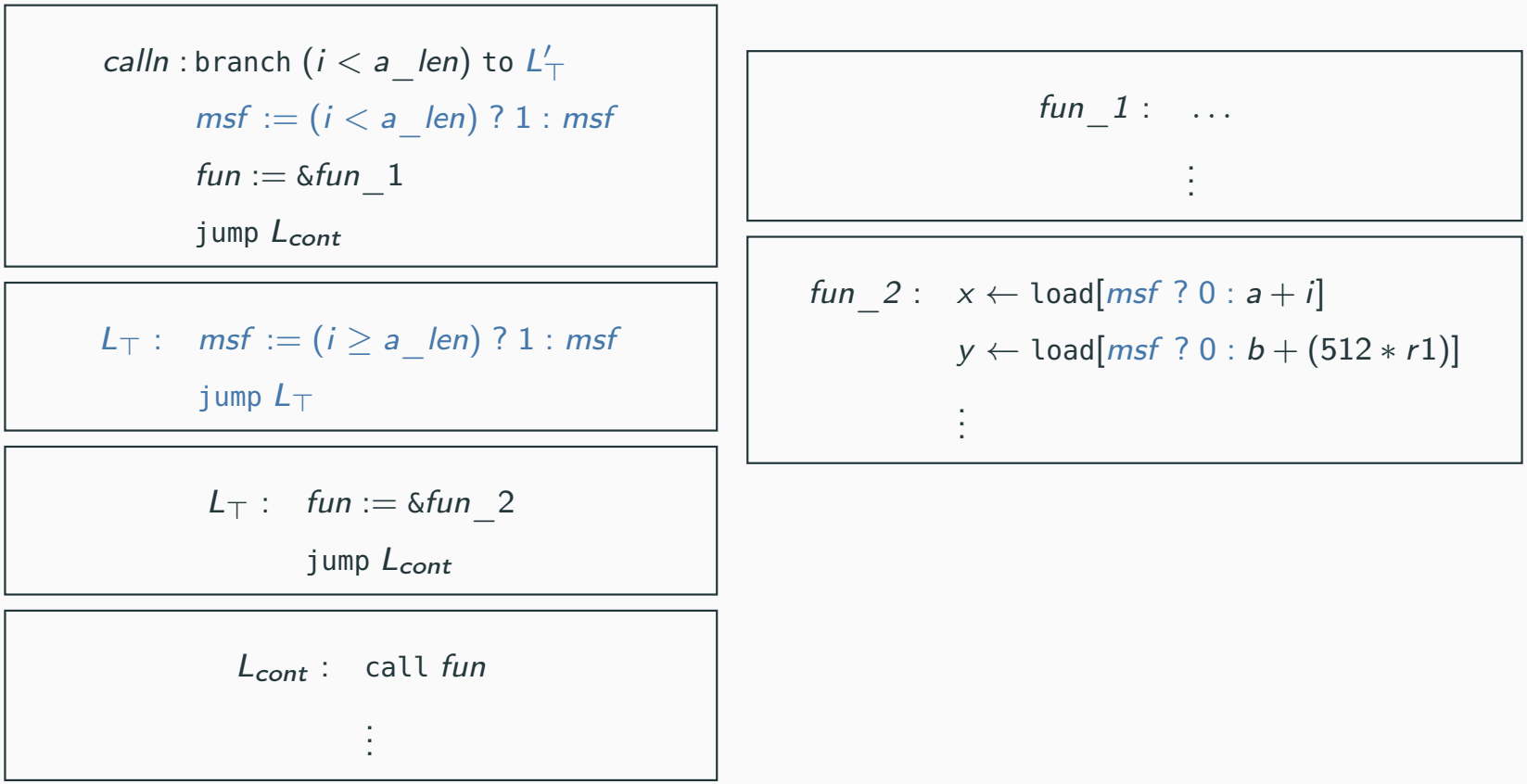
$\vdots$

fun_2 : $x \leftarrow \texttt{load}[a + i]$

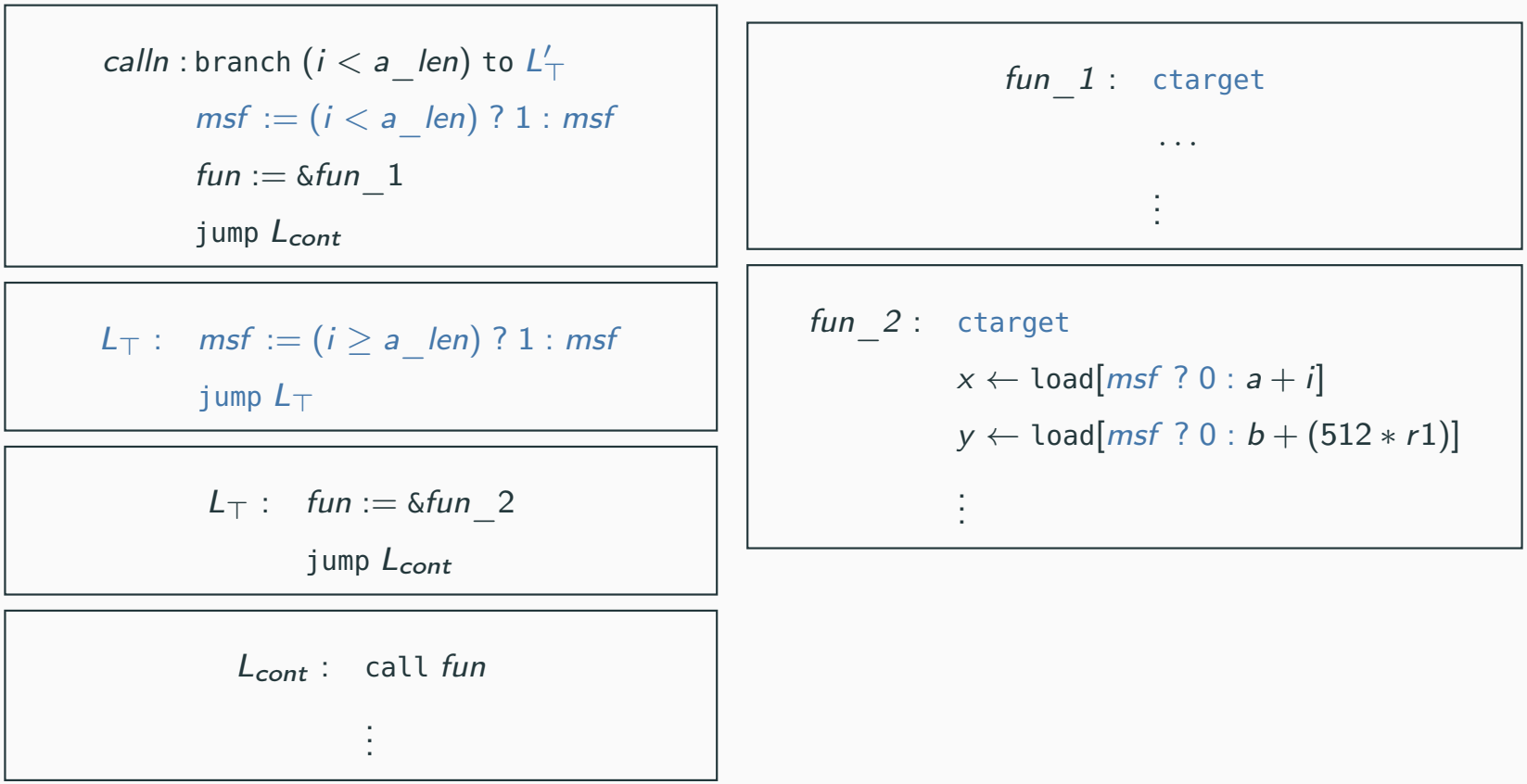
$y \leftarrow \texttt{load}[b + (512 * r1)]$

$\vdots$

Building up SpecIBT: USLH



Building up SpecIBT: USLH + Intel CET



SpecIBT fully protects against Spectre BTB

$calln$: `branch` $(i < a_len)$ `to` $L'_\top$

$msf := (i < a_len)\ ?\ 1 : msf$

$fun := \&fun_1$

`jump` L_{cont}

$L_\top$: $msf := (i \geq a_len)\ ?\ 1 : msf$

`jump` $L_\top$

$L_\top$: $fun := \&fun_2$

`jump` L_{cont}

L_{cont} : $callee := fun$

`call` fun

$\vdots$

fun_1 : `ctarget`

$msf := ((callee = fun_1)\ ?\ msf : 1)$

$\ldots$

$\vdots$

fun_2 : `ctarget`

$msf := (callee = fun_2)\ ?\ msf : 1$

$x \leftarrow$ `load`$[msf\ ?\ 0 : a + i]$

$y \leftarrow$ `load`$[msf\ ?\ 0 : b + (512 * r1)]$

$\vdots$

Summary

- Spectre PHT: USLH mitigates
- Spectre BTB: more powerful, USLH alone can't mitigate
- Intel CET partially protects
- **SpecIBT precisely detects and mitigates Spectre BTB (and PHT) attacks**

Formalization

MiniMIR

MiniMIR: simple, untyped imperative language with basic blocks

MIR: Machine Intermediate Representation, a low-level language in LLVM

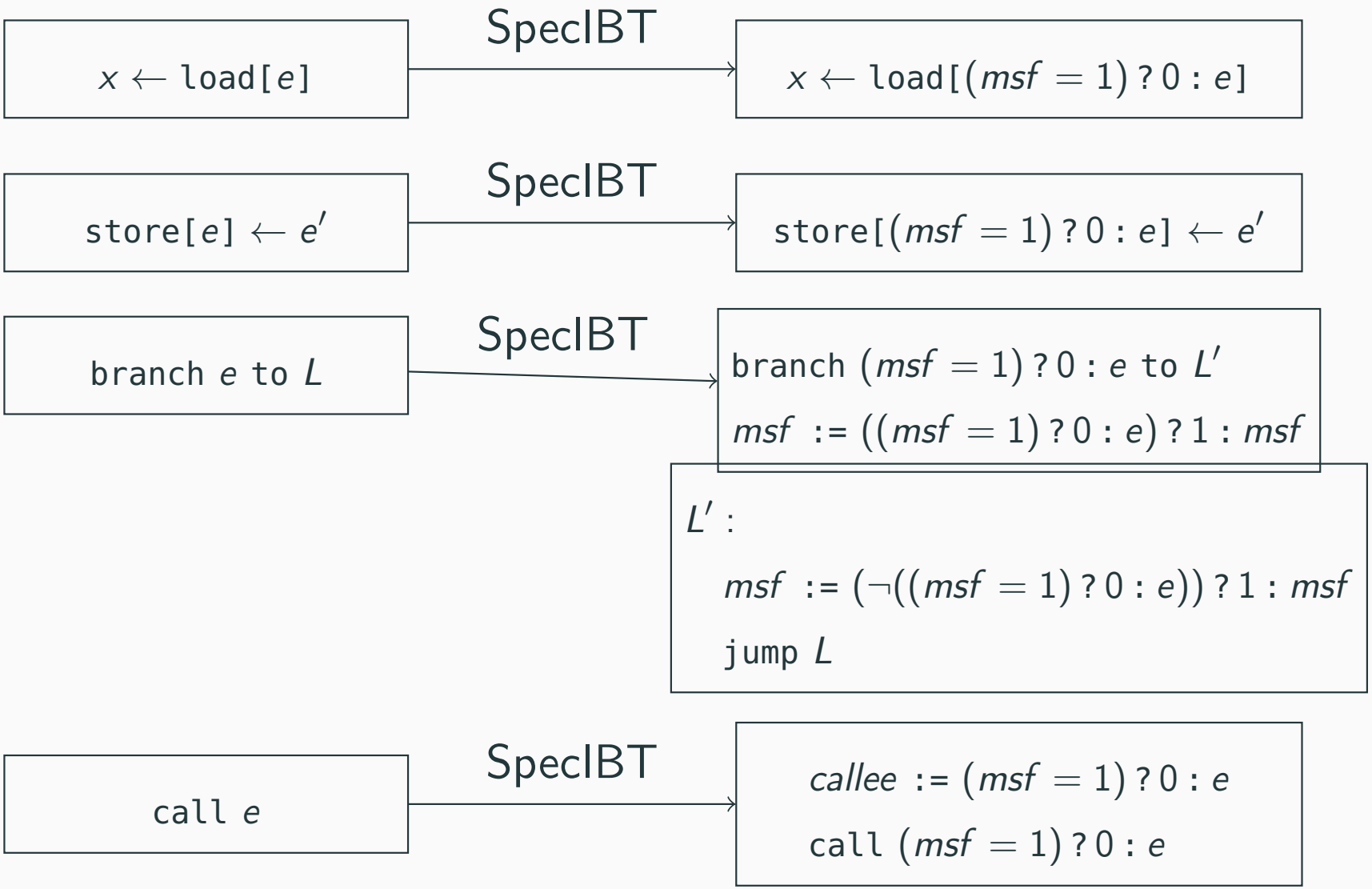
SpecIBT: MiniMIR $\rightarrow$ MiniMIR

MiniMIR: Expressions, Values

$e \in \texttt{exp} ::= n \in \mathbb{N}$		number
	$\mid \texttt{\&}L \in \mathcal{FP}$	function pointer
	$\mid \texttt{X} \in \mathcal{R}$	register
	$\mid \mathrm{op}_{\mathbb{N}}(e, \ldots, e)$	arithmetic operator
	$\mid b \texttt{ ? } e \texttt{ : } e$	branchless conditional
$v \in \texttt{val} ::= n \in \mathbb{N}$		number
	$\mid \texttt{\&}L \in \mathcal{FP}$	function pointer value
	$\mid \texttt{UV}$	undefined value

$c \in \texttt{inst} ::= \texttt{skip}$		do nothing
	$\mid \texttt{X := } e$	assignment
	$\mid \texttt{branch } e \texttt{ to } L$	conditional branch
	$\mid \texttt{jump } L$	unconditional jump
	$\mid \texttt{X} \leftarrow \texttt{load[}e\texttt{]}$	load from address
	$\mid \texttt{store[}e\texttt{]} \leftarrow e$	store to address
	$\mid \texttt{call } e$	call
	$\mid \texttt{ctarget}$	call-target marker
	$\mid \texttt{ret}$	return
$b \in \texttt{blk} ::= (L, (\texttt{[inst]}, \top/\bot))$		block
$p \in \texttt{prog} ::= \texttt{[blk]}$		program

How SpecIBT Transforms Instructions



Otherwise, $\forall i$, the SpecIBT transformation returns i.

How SpecIBT Transforms Call Target Blocks

For `call` instructions, we also expect the following two target instructions at the head of any call target block with label L:

L :

`ctarget`

msf := $(callee = L)\,?\,msf : 1$

Semantics

MiniMIR programs execute in three different semantics: sequential, speculative, and ideal.

Sequential: **source program** without speculation

Speculative: **target program** with speculation

Ideal: **source program** with speculation *and* SpecIBT protections (abstract specification for SpecIBT)

Leakage Model

All three semantics model **leakage** through a **side channel**.



$$
\begin{array}{rll}
\mathcal{O} ::= & \mathit{OBranch}\ b & \text{branch condition} \\
\mid & \mathit{OCall}\ L & \text{function pointer} \\
\mid & \mathit{OLoad}\ n & \text{memory address} \\
\mid & \mathit{OStore}\ n & \text{memory address}
\end{array}
$$

Attacker Model

The speculative and ideal semantics model a powerful attacker who has **control over the microarchitecture**.



$$\begin{aligned} \mathcal{D} ::= {} & \mathit{DBranch}\ b' && \text{branch condition} \\ \mid {} & \mathit{DCall}\ pc' && \text{function pointer} \end{aligned}$$

Notation: Program State

Sequential: $\langle pc, \rho, \mu, stk \rangle$

Speculative: $\langle pc, \rho, \mu, stk, ct \rangle$

Ideal: $\langle pc, \rho, \mu, stk, ms \rangle$

pc: program counter, has block label, offset into block

ρ, μ: map from registers or memory locations to contents

stk: stack, a list of program counters

Sequential Semantics Example Rule

$$\textsc{Seq_Call}\ \frac{\mathtt{p}[\![pc]\!] = \mathtt{call}\ e \quad [\![e]\!]_\rho = \&L}{\langle pc, \rho, \mu, stk\rangle \xrightarrow{OCall\ L} \langle (L, 0), \rho, \mu, (pc + 1) :: stk\rangle}$$

$\mathtt{p}[\![pc]\!] = \mathtt{call}\ e$: instruction fetch

$[\![e]\!]_\rho$: expression evaluation wrt register state

$\&L$: L is a function pointer value

$\xrightarrow{OCall\ L}$: sequential transition step emitting observation of function pointer L

Speculative Semantics Example Rule

$$\textsc{Spec_Call}\ \frac{\mathtt{p}[\![pc]\!] = \mathtt{call}\ e \quad [\![e]\!]_\rho = \&L}{\langle pc, \rho, \mu, stk, ct\rangle \xrightarrow[DCall\ pc']{OCall\ L}_s \langle pc', \rho, \mu, (pc+1) :: stk, \top\rangle}$$

$\xrightarrow[DCall\ pc']{OCall\ L}_s$: speculative step consuming directive pc', emitting observation of L

ct: "call target" flag. When set, `ctarget` expected as next instruction

Ideal Semantics Example Rule

$$\textsc{Ideal_Call}\ \frac{\mathtt{p}[\![pc]\!] = \mathtt{call}\ e \quad \text{if } ms \text{ then } \&0 \text{ else } [\![e]\!]_\rho = \&L \quad ms' = ms \vee (l' \neq l) \quad |\mathtt{p}| > L' \quad \mathit{IsCallTarget}(\mathtt{p}[L']) \quad o' = 0}{\langle pc, \rho, \mu, stk, ms\rangle \xrightarrow[\mathit{DCall}\ (L', o')]{\mathit{OCall}\ l}_i \langle (L', o'), \rho, \mu, (pc+1) :: stk, ms'\rangle}$$

ms: same role as *msf* register in target program: detection and mitigation

if *ms* then &0 else $[\![e]\!]_\rho = \&L$: evaluate e, mask if misspeculating

$|\mathtt{p}| > L'$, $\mathit{IsCallTarget}(\mathtt{p}[L'])$, $o' = 0$: does pc' go to head of call target block?

Relative Security

Sequential Observational Equivalence

Two **sequential states** $\langle s_1 \rangle$ and $\langle s_2 \rangle$ are observationally equivalent under the sequential semantics of program p, written $\langle s_1 \rangle \approx \langle s_2 \rangle$ iff

$$
\begin{aligned}
&\forall \mathcal{O}_1, \mathcal{O}_2 \\
&\langle s_1 \rangle \xrightarrow{\mathcal{O}_1}{}^{*} _ \Rightarrow \\
&\langle s_2 \rangle \xrightarrow{\mathcal{O}_2}{}^{*} _ \Rightarrow \\
&\mathcal{O}_1 \lessgtr \mathcal{O}_2,
\end{aligned}
$$

where $\xrightarrow{\mathcal{O}}{}^{*}$ denotes a multistep sequential execution emitting observation trace $\mathcal{O}$.

Speculative Observational Equivalence

Two **speculative states** $\langle s_1 \rangle$ and $\langle s_2 \rangle$ are observationally equivalent under the speculative semantics of program p, written $\langle s_1 \rangle \approx_s \langle s_2 \rangle$ iff

$$\begin{aligned}
&\forall \mathcal{D}, \mathcal{O}_1, \mathcal{O}_2 \\
&\langle s_1 \rangle \xrightarrow[\mathcal{D}]{\mathcal{O}_1}{}^*_s _ \Rightarrow \\
&\langle s_2 \rangle \xrightarrow[\mathcal{D}]{\mathcal{O}_2}{}^*_s _ \Rightarrow \\
&\mathcal{O}_1 \lessgtr \mathcal{O}_2,
\end{aligned}$$

where $\xrightarrow[\mathcal{D}]{\mathcal{O}}{}^*_s$ denotes a multistep speculative execution consuming directive sequence $\mathcal{D}$ and emitting observation trace $\mathcal{O}$.

Relative Security for SpecIBT

If the source program is sequentially observationally equivalent with respect to start states s_1 and s_2, then the target program must be speculatively observationally equivalent with respect to start states s_1' and s_2'.

$$\begin{aligned} & \mathsf{p} \vdash \langle s_1 \rangle \approx \langle s_2 \rangle && \Rightarrow \\ & s_1 \sim s_1' \ \wedge \ s_2 \sim s_2' && \Rightarrow \\ & (\!| \, \mathsf{p} \, |\!) \vdash \langle s_1' \rangle \approx_s \langle s_2' \rangle \end{aligned}$$

The $\sim$ relation accounts for how SpecIBT transforms the program.

Proving Relative Security

We prove the relative security theorem by first transforming the conclusion from speculative to ideal observational equivalence.

We do this with the backward compiler correctness lemma, which states equivalence between speculative executions of the target and ideal executions of the source program.

Remember, the conclusion of the relative security theorem is speculative observational equivalence. This is what it looks like unfolded:

$$
\begin{aligned}
&\forall \mathcal{D}, \mathcal{O}_1, \mathcal{O}_2 \\
&\langle s_1 \rangle \xrightarrow[\mathcal{D}]{\mathcal{O}_1}{}^{*}_{s} _ \Rightarrow \\
&\langle s_2 \rangle \xrightarrow[\mathcal{D}]{\mathcal{O}_2}{}^{*}_{s} _ \Rightarrow \\
&\mathcal{O}_1 \leqslant \mathcal{O}_2
\end{aligned}
$$

After having applied the compiler correctness lemma, the relative security theorem looks like this:

$$
\begin{aligned}
& \mathsf{p} \vdash \langle s_1 \rangle \approx \langle s_2 \rangle && \Rightarrow \\
& s_1 \sim i_1 \ \wedge \ s_2 \sim i_2 && \Rightarrow \\
& \mathsf{p} \vdash \langle i_1 \rangle \approx_i \langle i_2 \rangle
\end{aligned}
$$

where i_1, i_2 are ideal state configurations.

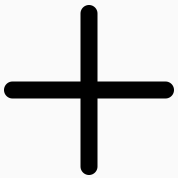
We finish proving relative security with another key lemma, which is out of scope for this talk.

Compiler Correctness Lemma

Ideal Semantics

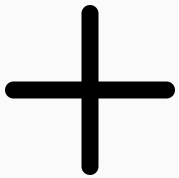


Leakage





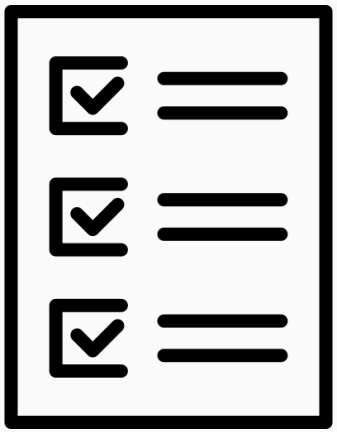
Attacker





SpecIBT
Protections

Ideal Semantics: Abstract Specification for SpecIBT

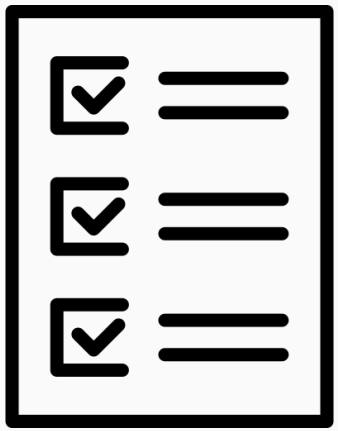


Ideal semantics:
abstract specification



SpecIBT: concrete
implementation

Backward Compiler Correctness Lemma



Ideal semantics:
abstract specification



Backward
compiler
correctness
lemma



SpecIBT: concrete
implementation

This notion of ideal semantics as abstract specification comes to us from the work of Shivakumar et. al., 2022.

Backward Compiler Correctness Lemma

Compiler correctness: prove *semantic preservation*, using a *simulation*.

$(\!|\ \mathsf{p}\ |\!) \vdash \langle\ t\ \rangle \xrightarrow[\mathcal{D}]{\mathcal{O}}{}^{*}_{s} \langle\ t'\ \rangle$ $\Rightarrow$ target program, concrete SpecIBT protections

$\langle\ t\ \rangle \sim \langle\ s\ \rangle$ $\Rightarrow$ relation on start states

$\mathsf{p} \vdash \langle\ s\ \rangle \xrightarrow[\mathcal{D}]{\mathcal{O}}{}^{*}_{i} \langle\ s'\ \rangle.$ source program, abstract specification of SpecIBT protections

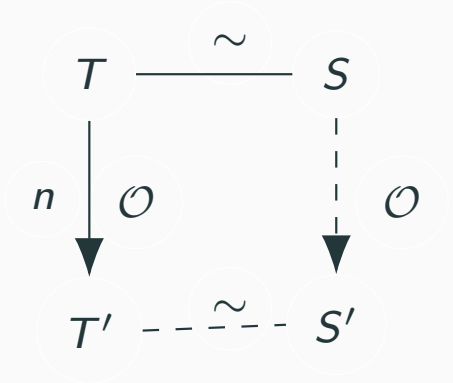
Compiler Correctness Proof

Overview of Proof

- by strong induction on length of target execution
- $\forall \xrightarrow[\mathcal{D}]{\mathcal{O}}{}^{*}_{s}, \exists \xrightarrow[\mathcal{D}]{\mathcal{O}}{}^{*}_{i}$
- prove that a single source step can take place, corresponding to a **target cycle**, then use induction hypothesis
- **Target cycle:** the 1-4 instructions it takes the target program to do the equivalent of a single source step
- **Target cycle** length: how many instructions did SpecIBT add?

Simulation Diagram[2]

If $T \sim S$, and $T \xrightarrow{\mathcal{O}}{}^{n} T'$, then $\exists\, S'$ such that $S \xrightarrow{\mathcal{O}} S' \wedge T' \sim S'$.



[2] adapted from X. Leroy, 2009

Single Cycle Lemma

$$
\begin{aligned}
&\mathit{LenCycle} = n, \text{ etc.} && \Rightarrow \\
&(\!| \, \mathsf{p} \, |\!) \vdash \langle \, t \, \rangle \xrightarrow[\mathcal{D}]{\mathcal{O}}{}^{n}_{s} \langle \, t' \, \rangle && \Rightarrow \\
&\langle \, t \, \rangle \sim \langle \, s \, \rangle && \Rightarrow \\
&\mathsf{p} \vdash \langle \, s \, \rangle \xrightarrow[\mathcal{D}]{\mathcal{O}}{}_{i} \langle \, s' \, \rangle \wedge \langle \, t' \, \rangle \sim \langle \, s' \, \rangle.
\end{aligned}
$$

State Invariant

$\langle pc, \rho, \mu, stk, ms \rangle \sim \langle pc', \rho', \mu', stk', ct \rangle :=$

$pc \sim pc'$: +2 if call target block, +1 for each prior `branch` and `call` instruction.

$\rho \sim \rho'$: equality, except source doesn't use dedicated target registers *callee* and *msf*. Target *msf* register is equivalent to ideal *ms* flag.

$\mu \sim \mu'$: equality

$stk \sim stk'$: same length, program counters pairwise related

Example case: Call

Steps in target cycle: 4

$\mathtt{p}[\![pc]\!] = \mathtt{call}\ e$ matches $(\!|\ \mathtt{p}\ |\!)[\![pc']\!] = callee\ \mathtt{:=}\ (msf = 1)\,\mathtt{?}\,0\,\mathtt{:}\,e$, $(\!|\ \mathtt{p}\ |\!)[\![pc' + 1]\!] = \mathtt{call}\ (msf = 1)\,\mathtt{?}\,0\,\mathtt{:}\,e$

I :

$(\!|\ \mathtt{p}\ |\!)[\![(L, 0)]\!] = \mathtt{ctarget}$

$(\!|\ \mathtt{p}\ |\!)[\![(L, 1)]\!] = msf\ \mathtt{:=}\ (callee = L)\,\mathtt{?}\,msf\,\mathtt{:}\,1$

Target Cycle: Step 1

SpecIBT stores the function pointer in the dedicated *callee* register.

$(\!|\ \mathsf{p}\ |\!)[\![pc']\!] = \mathit{callee} := (\mathit{msf} = 1)\ ?\ 0 : e$

(SPEC_ASGN transition rule)

State changes: $[\mathit{callee} \mapsto \mathit{msf} = 1\ ?\ 0 : e]\rho$

Target Cycle: Step 2

$(\!|\ \mathtt{p}\ |\!)[\![pc' + 1]\!] = \mathtt{call}\ (msf = 1)\ \mathtt{?}\ 0\ \mathtt{:}\ e$

(SPEC_CALL transition rule)

State changes:

- $OCall\ msf = 1\ \mathtt{?}\ 0\ \mathtt{:}\ e$
- $pc' + 2 :: stk'$

Behaviors: $DCall\ (al, ao)$

Fault?

Fault: observable behavior

Faulting target execution: 3 steps $\rightarrow$ contradiction

Call Target Block

For `call` instructions, we expect the following two target instructions at the head of any call target block with label L:

$$
\begin{array}{l}
L: \\
\quad (\!|\ \mathtt{p}\ |\!)[\![(L, 0)]\!] = \mathtt{ctarget} \\
\quad (\!|\ \mathtt{p}\ |\!)[\![(L, 1)]\!] = \mathit{msf}\ \mathtt{:=}\ (\mathit{callee} = L)\ \mathtt{?}\ \mathit{msf}\ \mathtt{:}\ 1
\end{array}
$$

Target Cycle: Step 3

3rd step: `ctarget`

$(\!|\ \mathsf{p}\ |\!)[\![(al, 0)]\!] = \texttt{ctarget}$ ($\textsc{Spec_CTarget}$ transition rule)

No state changes.

Target Cycle: Step 4

4th step: SpecIBT detects misspeculation to the *wrong* call target block.

$(\!|\ \mathsf{p}\ |\!)[\![(al, 1)]\!] = msf\ \mathsf{:=}\ (callee = al)\ \mathsf{?}\ msf\ \mathsf{:}\ 1$

(SPEC_CASGN transition rule)

State changes: $[msf \mapsto ((callee = al)\ \mathsf{?}\ msf\ \mathsf{:}\ 1)]\rho$

Target Result State and Behaviors

Cumulative state changes at the end of the target cycle:

- Program counter: $(al, 2)$
- Register state: $[callee \mapsto (msf = 1)\ ?\ 0 : e]\rho, [msf \mapsto ((callee = al)\ ?\ msf : 1)]\rho$
- Stack: $pc' + 2 :: stk'$

Target behavior:

- Emitted $OCall\ (msf = 1)\ ?\ 0 : e$
- Did not fault

Single Ideal Source Step

$$\textsc{Ideal_Call}\ \frac{\mathtt{p}[\![pc]\!] = \mathtt{call}\ e \quad \text{if } ms \text{ then } \&0 \text{ else } [\![e]\!]_\rho = \&L \quad ms' = ms \vee (l' \neq l) \quad |\mathtt{p}| > L' \quad \mathit{IsCallTarget}(\mathtt{p}[L']) \quad o' = 0}{\langle pc, \rho, \mu, stk, ms\rangle \xrightarrow[\mathit{DCall}\ (L', o')]{\mathit{OCall}\ L}_i \langle (L', o'), \rho, \mu, (pc+1) :: stk, ms'\rangle}$$

ms: same role as *msf* register in target program: detection and mitigation

if *ms* then &0 else $[\![e]\!]_\rho = \&L$: evaluate e, mask if misspeculating

$|\mathtt{p}| > L'$, $\mathit{IsCallTarget}(\mathtt{p}[L'])$, $o' = 0$: does pc' go to head of call target block?

Ideal Result State and Behaviors

State changes at the end of the ideal step:

- Program counter: $(al, 0)$
- Stack: $pc + 1 :: stk$
- Misspeculation flag: $ms' = ms \vee (l' \neq l)$

Ideal behavior:

- Emitted *OCall L*
- Did not fault

Semantic Preservation

Target: $\mathit{OCall}\,(\mathit{msf} = 1)\,?\,0 : e$

Source: $\mathit{OCall}\ L$, if ms then $\&0$ else $[\![e]\!]_\rho = \&L$

State Invariant

The state invariant $\sim$ holds between the resulting source and target states.

$pc \sim pc'$: $+2$ if call target block, $+1$ for each prior `branch` and `call` instruction.

✓ $(al, 0) \sim (al, 2)$

$\rho \sim \rho'$: equality, except source doesn't use dedicated target registers *callee* and *msf*.

✓ The only updates were to *callee* and *msf*.

Target *msf* register is equivalent to ideal *ms* flag.

✓ These are both set if the attacker directive doesn't match the intended location, which is masked in both cases if misspeculation is already occurring.

$stk \sim stk'$: program counters pairwise related

✓$pc + 1 \sim pc' + 2$

Single Cycle Lemma

$$
\begin{aligned}
&\mathit{LenCycle} = n, \text{ etc.} && \Rightarrow \\
&(\!| \, \mathsf{p} \, |\!) \vdash \langle \, t \, \rangle \xrightarrow[\mathcal{D}]{\mathcal{O}}{}^{n}_{s} \langle \, t' \, \rangle && \Rightarrow \\
&\langle \, t \, \rangle \sim \langle \, s \, \rangle && \Rightarrow \\
&\mathsf{p} \vdash \langle \, s \, \rangle \xrightarrow[\mathcal{D}]{\mathcal{O}}{}_{i} \langle \, s' \, \rangle \wedge \langle \, t' \, \rangle \sim \langle \, s' \, \rangle.
\end{aligned}
$$

Backward Compiler Correctness

Compiler correctness: prove *semantic preservation*, using a *simulation*.

$(\!|\ \mathsf{p}\ |\!) \vdash \langle\ t\ \rangle \xrightarrow[\mathcal{D}]{\mathcal{O}}{}^{*}_{s} \langle\ t'\ \rangle$ $\Rightarrow$ target program, concrete SpecIBT protections

$\langle\ t\ \rangle \sim \langle\ s\ \rangle$ $\Rightarrow$ relation on start states

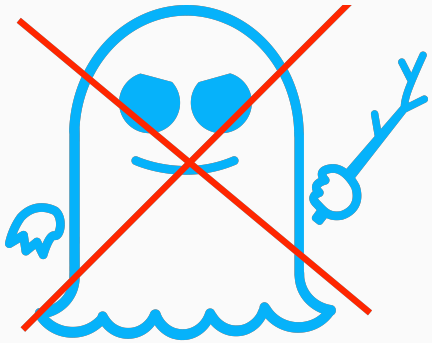
$\mathsf{p} \vdash \langle\ s\ \rangle \xrightarrow[\mathcal{D}]{\mathcal{O}}{}^{*}_{i} \langle\ s'\ \rangle.$ source program, abstract specification of SpecIBT protections

Relative Security for SpecIBT

If the source program is sequentially observationally equivalent with respect to start states s_1 and s_2, then the program transformed by SpecIBT must be speculatively observationally equivalent with respect to start states s_1' and s_2'.

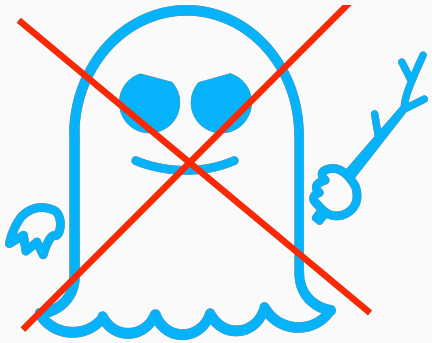
$$\begin{aligned} & \mathsf{p} \vdash \langle s_1 \rangle \approx \langle s_2 \rangle && \Rightarrow \\ & s_1 \sim s_1' \ \wedge \ s_2 \sim s_2' && \Rightarrow \\ & (\!| \, \mathsf{p} \, |\!) \vdash \langle s_1' \rangle \approx_s \langle s_2' \rangle \end{aligned}$$

SpecIBT: Formally Verified Spectre Mitigation



SPECTRE

PHT



SPECTRE

BTB



ROCQ

SpecIBT

Thank you!

attribution: icons from flaticon.com by Freepik, Andrean Prabowo, Fir3Ghost, rukanicon, and Pixel perfect.