Do You Even Lift? Strengthening Compiler Security Guarantees against Spectre Attacks*

XAVER FABIAN, CISPA Helmholtz Center for Information Security, Germany and University of Trento, Italy
MARCO PATRIGNANI, University of Trento, Italy
MARCO GUARNIERI, IMDEA Software Institute, Spain
MICHAEL BACKES, CISPA Helmholtz Center for Information Security, Germany

Mainstream compilers implement different countermeasures to prevent specific classes of speculative execution attacks. Unfortunately, these countermeasures either lack formal guarantees or come with proofs restricted to speculative semantics capturing only a subset of the speculation mechanisms supported by modern CPUs, thereby limiting their practical applicability. Ideally, these security proofs should target a speculative semantics capturing the effects of *all* speculation mechanisms implemented in modern CPUs. However, this is impractical and requires new secure compilation proofs to support additional speculation mechanisms.

In this paper, we address this problem by proposing a novel secure compilation framework that allows *lifting* the security guarantees provided by Spectre countermeasures from weaker speculative semantics (ignoring some speculation mechanisms) to stronger ones (accounting for the omitted mechanisms) *without* requiring new secure compilation proofs. Using our lifting framework, we performed the most comprehensive security analysis of Spectre countermeasures implemented in mainstream compilers to date. Our analysis spans 9 different countermeasures against 5 classes of Spectre attacks, which we proved secure against a speculative semantics accounting for 5 different speculation mechanisms. Our analysis highlights that fence-based and retpoline-based countermeasures can be securely lifted to the strongest speculative semantics under study. In contrast, countermeasures based on speculative load hardening cannot be securely lifted to semantics supporting indirect jump speculation.

CCS Concepts: • **Security and privacy** → **Formal security models**; **Systems security**.

Additional Key Words and Phrases: Spectre; Speculative Execution; Secure Compilation

ACM Reference Format:
Xaver Fabian, Marco Patrignani, Marco Guarnieri, and Michael Backes. 2025. Do You Even Lift? Strengthening Compiler Security Guarantees against Spectre Attacks. *Proc. ACM Program. Lang.* 9, POPL, Article 31 (January 2025), 30 pages. https://doi.org/10.1145/3704867

1 Introduction

Spectre [39] and other speculative execution attacks exploit the fact that modern CPUs speculate over the outcome of different instructions—branches [39], indirect jumps [39], stores and loads [35], and returns [41]—to bypass software-level security checks and leak sensitive information.

*This paper uses syntax highlighting accessible to both colourblind and black & white readers. For a better experience, please print or view this in colour [47].

Authors' Contact Information: Xaver Fabian, CISPA Helmholtz Center for Information Security, Saarbrücken, Germany and University of Trento, Trento, Italy, xaver.fabian@cispa.de; Marco Patrignani, University of Trento, Trento, Italy, marco.patrignani@unitn.it; Marco Guarnieri, IMDEA Software Institute, Madrid, Spain, marco.guarnieri@imdea.org; Michael Backes, CISPA Helmholtz Center for Information Security, Saarbrücken, Germany, backes@cispa.de.

ACM 2475-1421/2025/1-ART31
https://doi.org/10.1145/3704867

To mitigate these attacks, mainstream compilers like Gcc and Clang implement countermeasures in the form of secure compilation passes [14, 15, 36]. These passes modify a given program to prevent specific classes of speculative leaks. Unfortunately, the majority of these countermeasures lack formal security guarantees. Even countermeasures that come with formal security guarantees, however, are proved secure against models (called speculative semantics) that *only* capture the specific speculative leaks each countermeasure is designed to prevent. For instance, some Spectre-PHT[1] countermeasures have recently been proved secure [49] against a speculative semantics that *only* models speculation over branch instructions.

Modern CPUs, however, employ a variety of speculation mechanisms which need to be accounted for when reasoning about speculative leaks. This limits the practical applicability of existing security proofs that focus on restricted classes of speculative leaks. For instance, the security proofs from Patrignani and Guarnieri [49] ignore some speculation mechanisms (e.g., speculation over memory disambiguation or indirect jumps) implemented in all mainstream CPUs, which might compromise the proved guarantees. Unfortunately, extending security proofs to support new speculation mechanisms is far from trivial since programs that are seemingly secure when considering each speculation mechanism in isolation might still leak due to their interactions [24]. This has direct impact on existing security proofs: as we show in Section 5, the security guarantees of *speculative load hardening*, a countermeasure implemented in the Clang compiler (and the corresponding proof [49]), break when extending the underlying speculative semantics to support speculation over indirect jumps.

Thus, establishing the security guarantees of any countermeasure ideally requires proving the security of that countermeasure against attacker models capturing the effects of *all* speculation mechanisms implemented in modern CPUs. This approach, however, is impractical: (1) it requires developing new secure compilation proofs against *stronger* models (i.e., models accounting for additional speculation mechanisms) for those countermeasures that have already been proved secure against weaker models, and (2) it requires additional secure compilation proofs whenever a new speculation mechanism is discovered from reverse engineering of existing CPUs.

In this paper, we address this problem by developing a formal framework that allows us to precisely characterize when the security guarantees provided by Spectre countermeasures can be *lifted* from weaker models (ignoring some speculation mechanism) to stronger ones (accounting for the omitted mechanisms). This lifting allows us to account for further speculative mechanisms *without* requiring new secure compilation proofs. Using our lifting framework, we performed a comprehensive security analysis of the Spectre countermeasures implemented in mainstream compilers, which we proved secure against a speculative semantics accounting for all known speculation mechanisms for which formal models exist. Concretely, we make the following contributions:

- We formalise two novel speculative semantics capturing speculation over indirect jumps [39] (denoted as 👻$_{\mathcal{J}}$) and straight-line speculation [7] (denoted as 👻$_{\mathsf{SLS}}$). We present these novel semantics alongside the formalisation of the language model we use in Section 2.
- We develop a new framework for reasoning about the security of compiler-level countermeasures against leaks induced by multiple speculation mechanisms (Section 3). This framework integrates the core ideas from the composition framework from Fabian et al. [24] and from the secure compilation framework from Patrignani and Guarnieri [49] to allow reasoning about secure compilers against multiple speculative semantics. We equip our framework with a precise notion of *leakage ordering* (inspired by the notion of hardware-software contracts [30]) that precisely relates the information exposed by different speculative semantics,

[1] Spectre-PHT [39] refers to a class of speculative execution attacks exploiting speculation over branch instructions. Here, PHT stands for “Pattern History Table”, one of the microarchitectural mechanisms responsible for branch speculation.

where semantics supporting more speculation mechanisms are stronger with respect to our ordering as they leak more. The integration of all these concepts required developing new insights tailored for secure compilation, e.g., novel well-formedness conditions for composition, preservation of compiler security from composed semantics to components and from stronger semantics to weaker ones.

- We precisely characterize under which conditions the security guarantees provided by a compiler can be *lifted* from a base speculative semantics 👻$_x$ to a stronger semantics 👻$_{x+y}$, i.e., one that extends the base semantics 👻$_x$ to model the effects of additional speculation mechanisms captured by the semantics 👻$_y$ (Section 4). Our lifting theorem (Theorem 4) states that the guarantees provided by a secure compiler for the base semantics 👻$_x$ can be lifted to the extended semantics 👻$_{x+y}$ whenever three core properties are satisfied: Security in Origin (i.e., the compiler is secure w.r.t. the base semantics 👻$_x$), Independence in Extension (i.e., the compiler does not introduce further leaks under the extension semantics 👻$_y$), and Safe Nesting (i.e., there are no new leaks due to speculations arising only from the combination of semantics 👻$_x$ and 👻$_y$). Finally, to simplify proving Independence and Safe Nesting, we propose two sufficient conditions, Syntactic Independence and Trapped Speculation, that provide the same guarantees with simpler proofs but under stricter constraints.
- Using our framework, we perform a comprehensive security analysis of Spectre countermeasures in mainstream compilers (Section 5). Our analysis spans 9 countermeasures against 5 different Spectre attacks: Spectre-PHT [39], Spectre-BTB [39], Spectre-STL [35], Spectre-RSB [41, 43], and Spectre-SLS [7]. To the best of our knowledge, this is the most extensive formal analysis of compiler countermeasures against speculative attacks to date (prior studies [49] are limited to Spectre-PHT countermeasures). As part of this analysis, we precisely characterize the security guarantees of all countermeasures with respect to a combined speculative semantics accounting for five different speculation mechanisms (all those for which formal models exist). We remark that our lifting theorem (Theorem 4) is instrumental in making our security analysis feasible since we use it to lift each countermeasure's security guarantees to all possible combined semantics *without* requiring new secure compilation proofs, which significantly reduces the amount of secure compilation proofs needed.
  Our security analysis highlights that:
  - Countermeasures that block or trap speculation, i.e., fence-based [37, 38] and retpoline-based [36] approaches, are the most secure—their security guarantees can be lifted to the stronger speculative semantics we can model. Furthermore, lifting their guarantees is "easy": it can be done with minimal proof effort since Syntactic Independence and Trapped Speculation can be used to simplify lifting proofs.
  - Countermeasures that mask insecure values during speculation i.e., Speculative Load Hardening [16] (SLH) and its variants [49, 59], require careful handling of the interactions between masking code and different speculation mechanisms. For instance, we show that the guarantees of SLH cannot be lifted to models supporting speculation over indirect jumps, because the speculation flag (which tracks whether mispredictions are happening) is not tracked correctly between jumps. Even when lifting is possible, lifting SLH guarantees is more difficult than for the other countermeasures we analyzed since it requires full proofs of Independence and Safe Nesting (the simpler sufficient conditions are not applicable).
- We mechanise the core results regarding our lifting framework (not those associated with our security analysis) in the Coq proof assistant and indicate those theorems with 🐓.

The paper concludes with a discussion of the presented result (Section 6), related work (Section 7), and conclusions (Section 8). For simplicity, we only discuss key aspects of our formal models here.

$$\textit{Events } \lambda ::= \epsilon \mid \alpha? \mid \alpha! \mid \delta \mid \lightning \qquad \textit{Actions } \alpha ::= (\texttt{call } f) \mid (\texttt{ret})$$

$$\textit{\mu arch. Acts. } \delta ::= \texttt{store}(n) \mid \texttt{load}(n) \mid \texttt{pc}(l) \mid \texttt{start}_x \mid \texttt{rlb}_x$$

$$\textit{Programs } W, P ::= M, \overline{F}, \overline{I} \qquad \textit{Attackers } A ::= M, \overline{F}\,[\cdot] \qquad \textit{Imports } I ::= f$$

$$\textit{Functions } F ::= \emptyset \mid F; f \mapsto (l_{start}, c) \qquad \textit{Code } c ::= n : i \mid c_1; c_2 \qquad \textit{Codebase } C ::= \overline{F}, \overline{I}$$

$$\textit{Values } v \in \textit{Vals} = \mathbb{N} \cup \{\bot\} \qquad \textit{Expressions } e ::= v \mid x \mid \ominus e \mid e_1 \otimes e_2$$

$$\textit{Instructions } i ::= \textbf{skip} \mid x \leftarrow e \mid \textbf{load } x, e \mid \textbf{store } x, e \mid \textbf{jmp } e \mid \textbf{beqz } x, l \mid x \overset{e?}{\longleftarrow} e' \mid$$
$$\textbf{spbarr} \mid \textbf{call } f \mid \textbf{ret} \mid \textbf{load}_{prv}\ x, e \mid \textbf{store}_{prv}\ x, e$$

$$\text{Configurations } \sigma ::= \langle p, m, a \rangle \qquad \textit{Frames } B ::= \emptyset \mid \overline{n}; B$$

$$\text{Prog. States } \Omega ::= C; \overline{B}; \sigma \qquad \textit{RegisterFile } a ::= \emptyset \mid a; x \mapsto v$$

$$\textit{Registers } x \in \textit{Regs} \qquad \textit{Memory } m ::= \emptyset \mid m; n \mapsto v \text{ where } \ n \in \mathbb{Z}$$

Fig. 1. Event Model, Static and Runtime Syntax

Full details and proofs can be found in the companion report [26]. The Coq development for the mechanisation of the proofs is available at [25].

2 Language Formalisation: μAsm, Speculative Semantics, and Their Combinations

This section presents μAsm, an assembly-like language [31] that we extend with a notion of components in order to identify the unit of compilation [49]. We use μAsm as a basis for our formal framework and secure compilers.

First, we introduce the attacker model we consider (Section 2.1). Next, we present μAsm's syntax and non-speculative semantics first (Section 2.2), followed by the different speculative semantics (Section 2.3). We then show how to combine different speculative semantics (Section 2.4) to account for multiple sources of speculative leaks. Finally, we describe how different semantics can be compared in terms of leaked information (Section 2.5).

2.1 Attacker Model

We adopt a commonly-used attacker model [3, 17, 21, 24, 29–31, 49, 56]: an attacker that observes the execution of a program through events τ (Figure 1). These events model timing leaks through cache and control flow while abstracting away low-level microarchitectural details.

Events λ are either the empty event ϵ, an action $\alpha?$ or $\alpha!$ where ? denotes events *from* the component *to* the attacker and ! denotes events in the other direction, a microarchitectural action δ, or the designated event $\lightning$ denoting termination.

Action $\texttt{call}\ f?$ represents a call to a function f in the component, while $\texttt{call}\ f!$ represents a call(back) to the attacker. In contrast, action $\texttt{ret}!$ represents a return to the attacker and $\texttt{ret}?$ a return(back) to the component.

The microarchitectural actions $\texttt{store}(n)$ and $\texttt{load}(n)$ track addresses of store and loads, thereby capturing leaks through the data cache. Moreover, $\texttt{pc}(l)$ tracks the program counter during execution, thereby capturing leaks through the instruction cache. Finally, the $\texttt{start}_x$ and $\texttt{rlb}_x$ microarchitectural actions respectively denote the start and rollback of a speculative transaction [31], i.e., a set of speculatively executed instructions. Since we consider multiple speculative semantics (and their combinations), $\texttt{start}_x$ and $\texttt{rlb}_x$ actions are labelled with an identifier x denoting from which semantics the transaction originated.

Traces $\overline{\tau}$ are sequences of events λ. A trace $\overline{\tau}$ is *terminating* if it ends in $\lightning$. Given a trace $\overline{\tau}$, its *non-speculative projection* $\overline{\tau}\upharpoonright_n$ [31] consists of all observations associated with non-speculatively executed instructions and it is computed by removing all sub-sequences enclosed between $\texttt{start}_x$ and $\texttt{rlb}_x$ for any x. To reason about combined semantics [24], we also need projections $\overline{\tau}\upharpoonright_x$ that removes from the trace the contributions of a specific semantics with identifier x. Specifically, $\overline{\tau}\upharpoonright_x$ denotes the trace obtained by removing from $\overline{\tau}$ all sub-sequences enclosed $\texttt{start}_x$ and $\texttt{rlb}_x$ for a given x.

2.2 Syntax and Semantics of μAsm

μAsm's syntax is presented in Figure 1; we indicate the sequence of elements $e_1, \cdots, e_2$ as $\overline{e}$ and $\overline{e} \cdot e$ denotes a stack with top element e and rest of stack $\overline{e}$.

μAsm has a notion of components, i.e., partial programs P, and of attackers A. Components P define their memory m (defined later), a list of functions $\overline{F}$, and a list of imports $\overline{I}$, which are all the functions the component expects to be defined by an attacker. An attacker A only defines its memory and its functions. We indicate a program code base, i.e., its functions and imports, as C. A component P and attacker A can be linked to obtain a whole program $W \equiv A\,[P]$.

Functions consist of a start label l_{start} indicating the position of the code c of that function. Each function ends with a **ret** instruction. Code c is a sequence of mappings from natural-number labels to instructions i, where instructions i include skipping, (conditional) register assignments, (private) loads, (private) stores, indirect jumps, conditional branches, speculation barriers, calls, and returns.[2] Instructions can refer to expressions e, constructed by combining registers x (described below) and values v with unary and binary operators. Values come from the set *Vals* and can be natural numbers, labels, or $\perp$.

Non-Speculative States. μAsm's semantics is defined in terms of program states (Figure 1). Program states $C; \overline{B}; \sigma$ consist of a codebase C, a return frame B, and a configuration σ. C is used to look up functions, while B stores the return addresses of called functions. B consists of a stack of stacks of natural numbers n. A new empty stack $\overline{n}$ is created whenever a context switch between component and attacker happens. In this way, neither component nor attacker can manipulate the return stack of each other. Configurations σ consist of the program p, the memory m, and the register file a. The code of the program p is defined as the union of the code of all the functions and is a partial function mapping labels l to instructions i.

Memories m map memory addresses $n \in \mathbb{Z}$ to values v. The memory is split into a public part (represented by positive addresses $n \geq 0$) and a private part (represented by negative addresses $n < 0$). Attackers A can only define and access the public memory while programs P define the private memory and can access both private and public memory.

Register files a map registers x in *Regs* to their values v. Note that the set *Regs* includes the designated registers **pc** and **sp**, modelling the program counter and the stack pointer respectively.

Non-Speculative Operational Semantics. μAsm is equipped with a big-step operational semantics [31] for expressions and a small-step operational semantics for instructions generating events. The former has judgement $a \triangleright e \downarrow v$ and means: "expression e reduces to value v under register file a." The latter has judgement $\Omega \xrightarrow{\tau} \Omega'$ meaning: "state Ω reduces in one step to Ω' emitting label τ."

Below is a selection of these rules. Rule Load executes a load instruction from address n, emitting the related microarchitectural action $\texttt{load}\ n$ exposing the accessed address n. Rule Call executes a call to function f' (whose address n' is looked up in the functions table $\mathcal{F}$), it updates the register file

[2] Technically, instruction labels are drawn from a set L of abstract labels mapped to natural numbers before execution.

$a' = a[\mathbf{pc} \mapsto n']$, and it pushes a new frame with the return address $a(\mathbf{pc}) + 1$ on the frames stack. The auxiliary function $\text{find\_fun}(n) = f$ is used to look up a function f starting from an address n in memory, while $C.\mathtt{intfs} \vdash f', f : in$ is an (omitted) judgement to determine the decorator for the call (and return) action: ? or ! depending on whether the call or return comes from the attacker to the component or vice-versa. Rule Ret resumes the computation with the **pc** set to the address (l) it pops from the frames stack.

$$\boxed{\Omega \xrightarrow{\tau} \Omega'}$$

$$\frac{p(a(\mathbf{pc})) = \mathbf{load}\ x, e \qquad x \neq \mathbf{pc} \qquad a \triangleright e \downarrow n}{C; \overline{B}; \langle p, m, a\rangle \xrightarrow{\mathtt{load}\ n} C; \overline{B}; \langle p, m, a[\mathbf{pc} \mapsto a(\mathbf{pc}) + 1, x \mapsto m(n)]\rangle}\ (\text{Load})$$

$$\frac{\begin{array}{cccc} p(a_v(\mathbf{pc})) = \mathbf{call}\ f' & \mathcal{F}(f') = n' & f' \in C.\mathtt{funs} & a' = a[\mathbf{pc} \mapsto n'] \\ a(\mathbf{pc}) = n & \text{find\_fun}(n) = f & & C.\mathtt{intfs} \vdash f', f : in \end{array}}{C; \overline{B}; \langle p, m, a\rangle \xrightarrow{\mathbf{call}\ f'?} C; (\overline{B} \cdot (a(\mathbf{pc}) + 1); \emptyset); \langle p, m, a'\rangle}\ (\text{Call})$$

$$\frac{\begin{array}{ccc} p(a(\mathbf{pc})) = \mathbf{ret} & a(\mathbf{pc}) = n & a' = a[\mathbf{pc} \mapsto l] \\ \text{find\_fun}(n) = f & \text{find\_fun}(l) = f' & C.\mathtt{intfs} \vdash f', f : in \end{array}}{C; (\overline{B} \cdot l; \overline{n}); \langle p, m, a\rangle \xrightarrow{\mathbf{ret}?} C; \overline{B}; \langle p, m, a'\rangle}\ (\text{Ret})$$

Lastly, the semantics must capture the execution of whole programs. Whole programs are the result of linking attackers and components and must have no undefined function imports. Whole programs have a (straightforward and therefore omitted) big-step semantics $\Downarrow$ that concatenates single steps into multiple ones and single labels into traces. The judgement $\Omega \Downarrow_{\overline{\tau}} \Omega'$ is read: "state Ω emits trace $\overline{\tau}$ and becomes Ω'." The behaviour of a whole program W, written $Beh_{NS}(W)$, is the set of terminating traces it produces.

Source Programs. The semantics described so far has no speculation. We use it as semantics for the source programs of all our compilers. We indicate the language of such source programs as L.

2.3 Speculative Semantics

The target languages of the compilers we consider all have different speculative semantics modeling the effects of speculatively executed instructions. The speculative semantics we define are summarised in Table 1, where we list the instruction triggering speculation for each semantics. We consider five different speculative semantics capturing branch speculation (👻$_{\mathsf{B}}$), store-bypass speculation (👻$_{\mathsf{S}}$), indirect jump speculation (👻$_{\mathcal{J}}$), speculation using a return-stack buffer (👻$_{\mathsf{R}}$), and straight-line speculation over return instructions (👻$_{\mathsf{SLS}}$). While 👻$_{\mathsf{B}}$, 👻$_{\mathsf{S}}$, and 👻$_{\mathsf{R}}$ come from prior work [24, 31], the 👻$_{\mathcal{J}}$ and 👻$_{\mathsf{SLS}}$ semantics are novel. All these semantics follow the always-mispredict approach [31]. At every instruction triggering speculative execution, the semantics first speculatively executes the *wrong* path for a bounded

Table 1. Speculative semantics with the instructions they speculate on and their effects on execution.

Semantics	Spec. Source (*specInst*)	Effect
👻$_{\mathsf{B}}$ [31]	**beqz**	branch misprediction
👻$_{\mathsf{S}}$ [24]	**store**	store bypass
👻$_{\mathcal{J}}$ (new)	(indirect) **jmp**	different jump target
👻$_{\mathsf{R}}$ [24]	**ret**	return misprediction
👻$_{\mathsf{SLS}}$ (new)	**ret**	return bypass

number of steps (called the speculation window) and then continues with the correct one. The effects of the speculatively executed instructions are visible on the trace as actions enclosed between `start` and `rlb` events. This always-mispredict approach captures the worst-case scenario in terms of leakage because it always explores all executions (and corresponding observations) associated with any possible choice of predicted values independently of the actual prediction. This over-approximates any (cache-based) observations that an attacker that can control speculation and select the predicted values among the possible predictions could observe.

All the speculative semantics we consider follow a similar structure, which we recap now. Formally, the speculative state Σ_x is a stack of speculative instances Φ_x where reductions happen only on top of the stack. Each instance Φ_x contains the program state Ω and the remaining speculation window n describing the number of instructions that can still be executed speculatively (or $\bot$ when no speculation is happening). Depending on the semantics, the instance Φ_x may track additional data, e.g., the return-stack buffer in 👻$_{\mathsf{R}}$. Below, we leave this additional information abstract and we indicate it by ...; we refer to the companion report for full definitions of speculative states. Throughout the paper, we fix the maximal speculation window, i.e., the maximum number of speculative instructions, to a global constant ω.

$$\textit{Speculative States } \Sigma_x ::= \overline{\Phi}_x \qquad\qquad \textit{Speculative Instance } \Phi_x ::= \langle \Omega, n, \ldots \rangle$$

Each semantics has an instruction that starts speculation (the central column of Table 1): whenever those instructions are executed, the semantics first pushes the mispredicted state and then the correct state onto the state Σ_x.

The (small-step) judgement for all speculative semantics is of the form $\Sigma_x \overset{\tau}{\text{👻}}_x \Sigma'_x$ and it describes how the speculative state is updated when executing instructions. For all speculative semantics, the behaviour $Beh_x(W)$ of a whole program W is the trace $\overline{\tau}$ generated by the big-step judgment $\Downarrow_x$, which executes the program W starting from its initial state until termination and collect all produced actions.

Below, we overview the small-step judgments $\Sigma_x \overset{\tau}{\text{👻}}_x \Sigma'_x$ for the speculative semantics we study. We start from the two new speculative semantics 👻$_{\mathsf{SLS}}$ (Section 2.3.1) and 👻$_{\mathcal{J}}$ (Section 2.3.2) and later present the semantics 👻$_{\mathsf{B}}$ (Section 2.3.3), 👻$_{\mathsf{S}}$ (Section 2.3.4), and 👻$_{\mathsf{R}}$ (Section 2.3.5) from prior work [24, 31]. We first describe all rules for 👻$_{\mathsf{SLS}}$, which we use to explain the structure of all our speculative semantics. In contrast, for 👻$_{\mathcal{J}}$, 👻$_{\mathsf{B}}$, 👻$_{\mathsf{S}}$, and 👻$_{\mathsf{R}}$, we only report the most significant rule for each semantics, i.e., the rule that triggers the specific form of speculation listed in Table 1.

2.3.1 *Modeling Straight-Line Speculation.*

Straight-line speculation (SLS) [5, 7] is a speculation mechanism implemented in some CPUs where return instructions are speculatively bypassed and the execution continues speculatively (after ignoring the return) for a fixed number of steps. The 👻$_{\mathsf{SLS}}$ semantics models the effect of straight-line speculation using the small-step rules below.

Speculation is started by **ret** instructions. Whenever the semantics executes **ret**, the return is speculatively bypassed and execution speculatively continues after the return (captured by Rule SLS:AM-Ret-Spec). The rule pushes on the stack of speculative states a new speculative instance $\langle \Omega'', \mathrm{j} \rangle$, from which execution will continue. Note that speculation only starts when we are inside the component. For this, the rule checks that the function f being executed is not in the imports, i.e., it is not attacker-defined: this ensures that labels are only produced when non-attacker code is executed. Otherwise, execution continues normally (Rule SLS:AM-Ret-Spec-att).

Executing instructions that do not trigger speculation updates the program state according to the non-speculative semantics, reduces the speculation window by 1, and produces actions when needed (Rule SLS:AM-NoSpec-action and Rule SLS:AM-NoSpec-epsilon). These rules are only triggered when the instruction is not a return, a fence or it is not in a set of instructions Z, which is used to track the speculation-related instructions across combinations of the semantics [24]. When the speculation window in the top instance hits 0, the speculative state is rolled back and discarded (Rule SLS:AM-Rollback) and execution continues from the speculative instance now on top.

Finally, speculation barriers terminate speculation, which we model by setting the current speculation window to 0 (Rule SLS:AM-barr-spec). In non-speculative executions (i.e., when the speculation window is $\bot$), speculation barriers are handled as **skip** (Rule SLS:AM-barr).

$$\boxed{\Phi_{\mathsf{SLS}} \overset{\tau}{\Longrightarrow}_{\mathsf{SLS}} \overline{\Phi}'_{\mathsf{SLS}}}$$

$$\frac{\begin{array}{c}p(\sigma(\mathbf{pc})) = \mathbf{ret} \quad \Omega \xrightarrow{\tau} \Omega' \quad \Omega = \overline{\mathrm{F}}; \overline{\mathrm{I}}; \overline{\mathrm{B}}; \sigma \quad \mathrm{find\_fun}(\Omega(\mathbf{pc})) = \mathbf{f} \quad \mathbf{f} \notin \overline{\mathrm{I}} \\ \Omega'' = \overline{\mathrm{F}}; \overline{\mathrm{I}}; \overline{\mathrm{B}}; \sigma'' \quad \sigma'' = \sigma[\mathbf{pc} \mapsto \Omega(\mathbf{pc}) + 1] \quad \mathbf{j} = \mathbf{min}(\omega, \mathbf{n})\end{array}}{\langle \Omega, \mathbf{n}+1 \rangle \overset{\tau}{\Longrightarrow}_{\mathsf{SLS}} \langle \Omega', \mathbf{n} \rangle \cdot \langle \Omega'', \mathbf{j} \rangle}\ (\mathsf{SLS}\text{:AM-Ret-Spec})$$

$$\frac{\begin{array}{c}p(\sigma(\mathbf{pc})) = \mathbf{ret} \quad \Omega \xrightarrow{\tau} \Omega' \\ \mathrm{find\_fun}(\Omega(\mathbf{pc})) = \mathbf{f} \quad \mathbf{f} \in \overline{\mathrm{I}}\end{array}}{\langle \Omega, \mathbf{n}+1 \rangle \overset{\tau}{\Longrightarrow}_{\mathsf{SLS}} \langle \Omega', \mathbf{n} \rangle}\ (\mathsf{SLS}\text{:AM-Ret-Spec-att})$$

$$\frac{\begin{array}{c}p(\sigma(\mathbf{pc})) \neq \mathbf{ret}, \mathbf{spbarr}, \mathbf{Z} \\ \Omega \xrightarrow{\tau} \Omega'\end{array}}{\langle \Omega, \mathbf{n}+1 \rangle \overset{\tau}{\Longrightarrow}_{\mathsf{SLS}} \langle \Omega', \mathbf{n} \rangle}\ (\mathsf{SLS}\text{:AM-NoSpec-action})$$

$$\frac{\mathbf{n} = 0 \text{ or } \vdash \Omega : \mathit{fin}}{\langle \Omega, \mathbf{n} \rangle \overset{\mathtt{rlb}_{\mathsf{SLS}}}{\Longrightarrow}_{\mathsf{SLS}}\ \varepsilon}\ (\mathsf{SLS}\text{:AM-Rollback})$$

$$\frac{\begin{array}{c}p(\sigma(\mathbf{pc})) \neq \mathbf{ret}, \mathbf{spbarr}, \mathbf{Z} \\ \Omega \xrightarrow{\epsilon} \Omega'\end{array}}{\langle \Omega, \mathbf{n}+1 \rangle \overset{\epsilon}{\Longrightarrow}_{\mathsf{SLS}} \langle \Omega', \mathbf{n} \rangle}\ (\mathsf{SLS}\text{:AM-NoSpec-epsilon})$$

$$\frac{\begin{array}{c}p(\sigma(\mathbf{pc})) = \mathbf{spbarr} \\ \Omega' = \Omega[\mathbf{pc} \mapsto \mathbf{pc} + 1]\end{array}}{\langle \Omega, \bot \rangle \overset{\epsilon}{\Longrightarrow}_{\mathsf{SLS}} \langle \Omega', \bot \rangle}\ (\mathsf{SLS}\text{:AM-barr})$$

$$\frac{\begin{array}{c}p(\sigma(\mathbf{pc})) = \mathbf{spbarr} \\ \Omega' = \Omega[\mathbf{pc} \mapsto \mathbf{pc} + 1]\end{array}}{\langle \Omega, \mathbf{n}+1 \rangle \overset{\epsilon}{\Longrightarrow}_{\mathsf{SLS}} \langle \Omega', \mathbf{0} \rangle}\ (\mathsf{SLS}\text{:AM-barr-spec})$$

2.3.2 Modeling Jump Speculation. Jump speculation allows jump instructions to speculate the address where they are jumping to [39]. To model this, Rule J:AM-Jmp-Spec creates a set of speculative instances, with one speculative instance for each of the possible jump targets in the program, by updating the program counter **pc** to that jump target.

$$\boxed{\Phi_{\mathcal{J}} \overset{\tau}{\Longrightarrow}_{\mathcal{J}} \overline{\Phi}'_{\mathcal{J}}}$$

$$\frac{\begin{array}{c}p(\sigma(\mathbf{pc})) = \mathbf{jmp}\ \mathbf{x} \quad \mathbf{x} \in \mathit{Regs} \quad \Omega \xrightarrow{\tau} \Omega' \quad \Omega = \overline{\mathrm{F}}; \overline{\mathrm{I}}; \overline{\mathrm{B}}; \sigma \\ \mathrm{find\_fun}(\Omega'(\mathbf{pc})) = \mathbf{f}' \quad \mathrm{find\_fun}(\sigma(\mathbf{pc})) = \mathbf{f} \quad \mathbf{f} \notin \overline{\mathrm{I}} \quad \overline{\mathrm{I}} \vdash \mathbf{f}, \mathbf{f}' : \mathit{internal} \\ \mathbf{j} = \mathbf{min}(\omega, \mathbf{n}) \quad \overline{\Sigma''_{\mathcal{J}}} = \bigcup_{l \in p} \langle \Omega'', \mathbf{j} \rangle \text{ where } \Omega'' = \overline{\mathrm{F}}; \overline{\mathrm{I}}; \overline{\mathrm{B}}; \sigma[\mathbf{pc} \mapsto \mathbf{l}]\end{array}}{\langle \Omega, \mathbf{n}+1 \rangle \overset{\tau}{\Longrightarrow}_{\mathcal{J}} \langle \Omega', \mathbf{n} \rangle \cdot \overline{\Sigma''_{\mathcal{J}}}}\ (\mathcal{J}\text{:AM-Jmp-Spec})$$

2.3.3 Modeling Branch Speculation. CPUs speculate over the outcome of branch instructions [39], which might result in speculatively executing the wrong branch. To model this, we rely on the $\Longrightarrow_{\mathsf{B}}$ semantics from [31] where Rule B:AM-Branch-Spec speculates on branching instructions by pushing on top of the stack of speculative states the state that is opposite of the evaluated condition.

$$\boxed{\Phi_{\mathsf{B}} \overset{\tau}{\Longrightarrow}_{\mathsf{B}} \overline{\Phi}'_{\mathsf{B}}}$$

(B:AM-Branch-Spec)

$$\frac{\begin{array}{c} p(\sigma(\mathbf{pc})) = \mathbf{beqz}\ \mathbf{x}, \mathbf{l} \qquad \Omega \xrightarrow{\tau} \Omega' \qquad \Omega = \overline{\mathbf{F}}; \overline{\mathbf{I}}; \overline{\mathbf{B}}; \sigma \qquad \text{find\_fun}(\sigma(\mathbf{pc})) = \mathbf{f} \qquad \mathbf{f} \notin \overline{\mathbf{I}} \\ \Omega'' = \overline{\mathbf{F}}; \overline{\mathbf{I}}; \overline{\mathbf{B}}; \sigma'' \qquad \sigma'' = \sigma[\mathbf{pc} \mapsto \mathbf{l}'] \qquad \mathbf{j} = \mathbf{min}(\omega, \mathbf{n}) \\ \text{if } \sigma'(\mathbf{pc}) = \mathbf{l} \text{ then } \mathbf{l}' = \sigma(\mathbf{pc}) + 1 \text{ otherwise } \mathbf{l}' = \mathbf{l} \end{array}}{\langle \Omega, \mathbf{n} + 1 \rangle \overset{\tau}{\rightsquigarrow}_{\mathrm{B}} \langle \Omega', \mathbf{n} \rangle \cdot \langle \Omega'', \mathbf{j} \rangle}$$

2.3.4 Modeling Store-bypass Speculation. Modern CPUs write stores to main memory asynchronously to reduce delays caused by the memory subsystem. This may result in speculatively bypassing store instructions and fetching stale information from the CPU's load-store queue [35]. To model this, we rely on the $\rightsquigarrow_{\mathrm{S}}$ semantics from [24] where Rule S:AM-Store-Spec speculates on stores by adding a state at the top of the stack where a store has been skipped.

$$\boxed{\Phi_{\mathrm{S}} \overset{\tau}{\rightsquigarrow}_{\mathrm{S}} \overline{\Phi}'_{\mathrm{S}}}$$

(S:AM-Store-Spec)

$$\frac{\begin{array}{c} p(\sigma(\mathbf{pc})) = \mathbf{store}\ \mathbf{x}, \mathbf{e} \qquad \Omega \xrightarrow{\tau} \Omega' \qquad \Omega = \overline{\mathbf{F}}; \overline{\mathbf{I}}; \overline{\mathbf{B}}; \sigma \qquad \text{find\_fun}(\sigma(\mathbf{pc})) = \mathbf{f} \qquad \mathbf{f} \notin \overline{\mathbf{I}} \\ \Omega'' = \overline{\mathbf{F}}; \overline{\mathbf{I}}; \overline{\mathbf{B}}; \sigma'' \qquad \sigma'' = \sigma[\mathbf{pc} \mapsto \Omega(\mathbf{pc}) + 1] \qquad \mathbf{j} = \mathbf{min}(\omega, \mathbf{n}) \end{array}}{\langle \Omega, \mathbf{n} + 1 \rangle \overset{\tau}{\rightsquigarrow}_{\mathrm{S}} \langle \Omega', \mathbf{n} \rangle \cdot \langle \Omega'', \mathbf{j} \rangle}$$

2.3.5 Modeling Return Speculation. CPUs also speculate on the outcome of return instructions [41]. For this, they rely on a microarchitectural data structure called the return-stack-buffer (RSB). The speculative semantics $\rightsquigarrow_{\mathrm{R}}$, taken from [24], models this kind of speculation by extending the speculative instances with a return stack buffer $\mathbb{R}$, which is a list of return locations. Rule R:AM-Ret-Spec, which is the one starting speculation, works by speculatively returning to the 'wrong' location $\mathbf{l}$ at the top of the RSB whenever $\mathbf{l}$ differs from the expected return address $\mathbf{B}(0)$ upon encountering a return instruction. Note also that the semantics pushes a return address to the RSB whenever a **call** instruction is executed (not shown in the rules below).

$$\boxed{\Phi_{\mathrm{R}} \overset{\tau}{\rightsquigarrow}_{\mathrm{R}} \overline{\Phi}'_{\mathrm{R}}}$$

(R:AM-Ret-Spec)

$$\frac{\begin{array}{c} p(\sigma(\mathbf{pc})) = \mathbf{ret} \qquad \Omega \xrightarrow{\tau} \Omega' \qquad \Omega = \overline{\mathbf{F}}; \overline{\mathbf{I}}; \overline{\mathbf{B}}; \sigma \qquad \Omega' = \overline{\mathbf{F}}; \overline{\mathbf{I}}; \overline{\mathbf{B}'}; \sigma' \qquad \mathbb{R} = \mathbb{R}' \cdot \mathbf{l} \\ \mathbf{l} \neq \mathbf{B}(0) \qquad \Omega'' = \overline{\mathbf{F}}; \overline{\mathbf{I}}; \overline{\mathbf{B}'}; \sigma'' \qquad \mathbf{j} = \mathbf{min}(\omega, \mathbf{n}) \\ \text{find\_fun}(\sigma(\mathbf{pc})) = \mathbf{f} \qquad \text{find\_fun}(\sigma'(\mathbf{pc})) = \mathbf{f}' \qquad \mathbf{f}, \mathbf{f}' \notin \overline{\mathbf{I}} \qquad \sigma'' = \sigma[\mathbf{pc} \mapsto \mathbf{l}] \end{array}}{\langle \Omega, \mathbb{R}, \mathbf{n} + 1 \rangle \overset{\tau}{\rightsquigarrow}_{\mathrm{R}} \langle \Omega', \mathbb{R}', \mathbf{n} \rangle \cdot \langle \Omega'', \mathbb{R}', \mathbf{j} \rangle}$$

2.4 Combining Speculative Semantics

To reason about leaks resulting from multiple speculation sources, we rely on the combination framework from Fabian et al. [24]. This framework allows combining multiple speculative semantics (for different speculation sources) into a combined semantics that allows reasoning about all these kinds of speculation. For instance, combining $\rightsquigarrow_{\mathrm{B}}$ and $\rightsquigarrow_{\mathrm{S}}$ yields the composed semantics $\rightsquigarrow_{\mathrm{B+S}}$ that speculates on both **beqz** and **store** instructions. We remark that a composed semantics is "stronger than its parts", that is, it may explore speculative actions that only arise from the interaction of its component semantics. As shown in [24], some programs contain leaks only under the composed semantics, even though the programs are leak-free when considering the base semantics in isolation.

The core component of this combination framework is the notion of *well-formed composition*, which needs to be tailored to the specific properties that combinations need to preserve. Here we report two properties that well-formed combinations need to satisfy; we will introduce a third, novel, well-formedness condition in Section 3 to deal with preservation of the security properties.

Definition 1 (Well-Formed Composition – Part 1)**.** *The composition of semantics* 👻$_x$ *and* 👻$_y$ *is well-formed, denoted with* $\vdash$ 👻$_{x+y}$: *WFC, if it satisfies the following properties and the property later defined in Definition 6:*

- ***Confluence [24]:*** *If* $\Sigma_{x+y} \xrightarrow{\tau'}$👻$_{x+y}$ Σ'_{x+y} *and* $\Sigma_{x+y} \xrightarrow{\tau''}$👻$_{x+y}$ Σ''_{x+y}, *then* $\Sigma'_{x+y} = \Sigma''_{x+y}$ *and* $\tau' = \tau''$.
- ***Projection Preservation [24]:*** $Beh_x(P) = Beh_{x+y}(P)\upharpoonright_y$ *and* $Beh_y(P) = Beh_{x+y}(P)\upharpoonright_x$.

Confluence ensures the determinism of the combined semantics (where Σ_{x+y} is the operational state for the combined semantics), whereas Projection Preservation ensures that the speculative behaviour of the source semantics can be recovered from the behavior of the composed semantics.

2.5 Leakage Ordering

Each speculative semantics in Table 1, as well as their compositions, capture different "attacker models", where the attacker can observe the effects (visible on the traces) of the speculative instructions modelled by the semantics. To reason about the strength of these attacker models, we follow [30] and introduce a partial order in terms of leakage between the different semantics. In particular, we say that semantics 👻$_1$ is weaker than another semantics 👻$_2$, written 👻$_1 \sqsubseteq$ 👻$_2$ iff 👻$_2$ leaks more than 👻$_1$, i.e., if any two initial configurations that result in different traces for 👻$_1$ also result in different traces for 👻$_2$.[3]

Figure 2 depicts all the semantics studied in this paper as well as their combinations ordered according to the amount of leaked information, where there is an edge from semantics 👻$_1$ to 👻$_2$ whenever 👻$_1 \sqsubseteq$ 👻$_2$. In the figure, the weakest semantics is 👻$_{NS}$, since it only exposes information about non-speculative instructions. In contrast, 👻$_{B+S}$ exposes speculation on both **beqz** and **store** instructions and is thus stronger than its components 👻$_B$ and 👻$_S$. Note that there is no single strongest semantics in Figure 2 due to limitations in the combination framework we use [24], which does not allow for combining semantics speculating on the same instruction like 👻$_R$ and 👻$_{SLS}$.

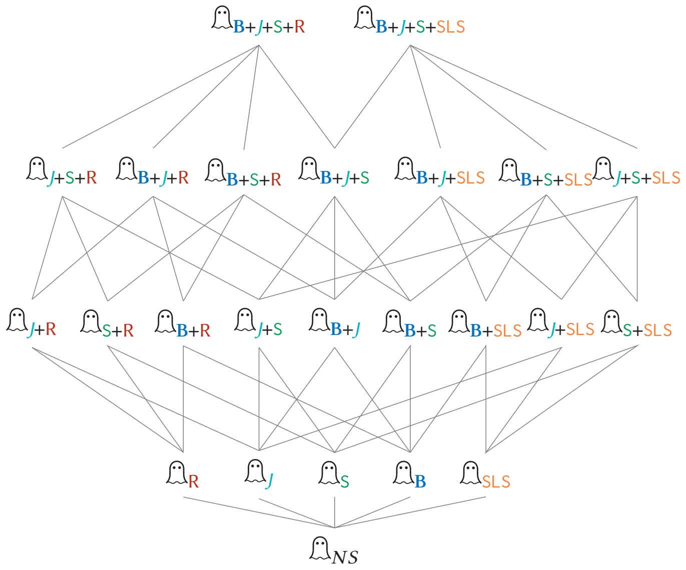


Fig. 2. Ordering of μAsm semantics. A semantics higher in the order is stronger, i.e., exposes more information.

[3] For readers familiar with [30], we flipped the relation $\sqsubseteq$ from the original paper.

3 Security Notions

In this section, we present the security notions used in our framework. Since our goal is studying the security of compiler countermeasures against different classes of speculative leaks, we extend the secure compilation framework from Patrignani and Guarnieri [49] to work with μAsm and with all speculative semantics from Section 2 and their combinations.

All our security definitions rely on a notion of robustness, typical for secure compilation work [2], which we explain first (Section 3.1). Next, we introduce two security notions for programs (Section 3.2): Robust Speculative Non-Interference (RSNI) and Robust Speculative Safety (RSS). We continue by presenting the secure compilation criteria (Section 3.3). We conclude by introducing Relation Preservation, the novel, remaining property for well-formed compositions (Section 3.4), which together with Confluence and Projection Preservation from Section 2.4 precisely characterize the core properties of composed semantics.

3.1 Robustness

All our security definitions are *robust* [1, 2, 27, 28, 48, 51, 55], i.e., they quantify over *every* possible valid attacker. In particular, μAsm defines partial programs P (which specify a set of functions to be imported) that are linked to the attacker-controlled context A (which defines the imported functions), leading to a whole program $W \equiv A\,[P]$. Thus, the attacker is also code executed together with the partial program P.

In this work, we say that a component *satisfies a property robustly* iff it satisfies the property for all possible valid attackers, where an attacker is valid, written $\vdash A : atk$, if it does not define a private memory and does not contain instructions that read and write to the private memory. This notion of robustness allows for separate compilation of our partial programs P.

3.2 Security Notions for Whole Programs

In this section, we extend the notions of RSNI (Section 3.2.1) and RSS (Section 3.2.2) to μAsm and all semantics from Section 2.

3.2.1 Robust Speculative Non-Interference (RSNI). RSNI is the application of Speculative Non-Interference to the robust setting. Speculative Non-Interference (SNI) is a class of security properties [30, 31] that compares the information leaked by instructions executed speculatively and non-speculatively. Intuitively, a program satisfies SNI iff it does not leak more information under the speculative semantics than under the non-speculative semantics. Thus, SNI semantically characterizes security against leaks introduced by speculatively executed instructions.

RSNI is parametric in (1) a policy denoting the sensitive information and (2) in the speculative semantics 👻_x, which models the speculative behaviour of programs. The policy describes which parts of the program state are public. In our case, only the private part of the memory M is sensitive. Thus, two programs P and P' are low-equivalent, written $P =_L P'$, if they only differ in their private memory. RSNI compares the leakage between non-speculative traces and speculative traces. In a nutshell, a program P satisfies RSNI (Definition 2) for a speculative semantics 👻_x if for any low-equivalent program P' that generates the same non-speculative trace, the two programs generate the same speculative traces as well robustly. Here the (omitted) function Ω_0 is used to initialize the machine state for whole programs.

Definition 2 (Robust Speculative Non-Interference [49] (RSNI)).

$$\begin{aligned}\text{👻}_x \vdash P\colon \mathit{RSNI} \overset{\text{def}}{=} \forall A, W'.\ &\mathit{if} \vdash A\colon \mathit{atk}\ \mathit{and}\ A\,[P] =_L W'\ \mathit{and}\ \mathit{Beh}_x(\Omega_0\,(A\,[P]))\upharpoonright_n = \mathit{Beh}_x(\Omega_0\,(W'))\upharpoonright_n \\ &\mathit{then}\ \mathit{Beh}_x(\Omega_0\,(A\,[P])) = \mathit{Beh}_x(\Omega_0\,(W'))\end{aligned}$$

We remark that all programs satisfy RSNI for the non-speculative semantics $👻_{NS}$ because there is no speculation and, thus, $\overline{\tau}\upharpoonright_n = \overline{\tau}$ for all its traces [49, Theorem 3.4].

3.2.2 Robust Speculative Safety. SNI is a hyperproperty and requires reasoning about pairs of traces. To simplify secure compilation proofs, we follow [49] and over-approximate RSNI using robust speculative safety (RSS), a safety property which only requires reasoning about single traces.

Just like RSNI, RSS is the application of Speculative Safety to the robust setting. Speculative Safety uses taint tracking, tainting values as “safe” (denoted by S) if the value can be speculatively leaked without violating RSNI (e.g., the public memory is safe), or “unsafe” (denoted by U) otherwise. Furthermore, taints are propagated during computation. We instantiate taint tracking for all the speculative semantics in Section 2.3; we refer the interested reader to [49, Section 3.2] for the taint tracking rules since these rules are virtually unchanged.

RSS ensures that programs P robustly generate only safe (S) actions in their traces.

Definition 3 (Robust Speculative Safety [49] (RSS))**.**

$$👻_x \vdash P : RSS \overset{\text{def}}{=} \forall A, \tau, \lambda^{\sigma}.\ if \vdash A : atk\ and\ \tau \in Beh_x(\Omega_0\,(A\,[P]))\ and\ \lambda^{\sigma} \in \tau,\ then\ \sigma \equiv S$$

Again, RSS trivially holds for the non-speculative semantics $👻_{NS}$ [49, Theorem 3.9] because there is no speculation.

Theorem 1, which we proved for all speculative semantics defined in Table 1, precisely connects RSNI and RSS by showing that RSS over-approximates RSNI.

Theorem 1 (RSS Overapproximates RSNI). *For all semantics $👻_x$ in Table 1, if $👻_x \vdash P : RSS$ then $👻_x \vdash P : RSNI$.*

3.3 Secure Compilation Criteria

We now present *robust speculative safety preservation* (*RSSP*) and *robust speculative non-interference preservation* (*RSNIP*), two criteria defined in [49] for reasoning about compiler guarantees against speculative leaks, which we make parametric in the underlying speculative semantics. Note that in this paper we use the term “compiler” to refer to an individual compilation pass on μAsm programs (rather than to a compiler from a high-level to a low-level language as is more usual).

A compiler preserves RSS for a given semantics $👻_x$ if given a source component that is RSS under the non-speculative semantics, the compiled counterpart is also RSS under $👻_x$.

Definition 4 (Robust speculative safety preservation [49] (*RSSP*))**.**

$$👻_x \vdash [\![\cdot]\!] : RSSP \overset{\text{def}}{=} \forall P \in L.\ if\, 👻_{NS} \vdash P : RSS\ then\ 👻_x \vdash [\![P]\!] : RSS$$

Similarly, a compiler preserves RSNI for a given semantics $👻_x$ if given a source component that is RSNI under the non-speculative semantics, the compiled counterpart is also RSNI under $👻_x$.

Definition 5 (Robust speculative non-interference preservation [49] (*RSNIP*))**.**

$$👻_x \vdash [\![\cdot]\!] : RSNIP \overset{\text{def}}{=} \forall P \in L.\ if\, 👻_{NS} \vdash P : RSNI\ then\ 👻_x \vdash [\![P]\!] : RSNI$$

We conclude by stating Theorem 2, which presents two new results. It states that (1) whenever a compiler preserves the security for a stronger semantics (i.e., one that exposes more information), then it also preserves security for weaker ones, and dually that (2) whenever a compiler does not preserve security for a weaker semantics, then it does not preserve security for stronger ones.

Theorem 2 (Leakage ordering and *RSNIP*, 🐓). *The following statements hold for any $👻_1, 👻_2$:*

- *If $👻_2 \vdash [\![\cdot]\!] : RSNIP$ and $👻_1 \sqsubseteq 👻_2$ then $👻_1 \vdash [\![\cdot]\!] : RSNIP$.*
- *If $👻_1 \nvdash [\![\cdot]\!] : RSNIP$ and $👻_1 \sqsubseteq 👻_2$ then $👻_2 \nvdash [\![\cdot]\!] : RSNIP$.*

3.4 Well-Formed Compositions and Compilers

We now introduce Relation Preservation, the last property—in addition to Confluence and Projection Preservation (see Section 2.4)—for well-formed compositions of speculative semantics. We remark that while Confluence and Projection Preservation come directly from [24], Relation Preservation is new and tailored to ensure, together with the other two properties, that RSS overapproximates RSNI for any well-formed composition.

Definition 6 (Well-Formed Composition – Part 2)**.**

Relation Preservation

$$\textit{If } \Sigma_{x+y} \approx \Sigma'_{x+y} \textit{ and } \Sigma_{x+y} \Downarrow^{\overline{\tau}}_{x+y} \Sigma^{\dagger}_{x+y} \textit{ and } \vdash \overline{\tau} : \textit{safe then } \Sigma'_{x+y} \Downarrow^{\overline{\tau}}_{x+y} \Sigma''_{x+y} \textit{ and } \Sigma^{\dagger}_{x+y} \approx \Sigma''_{x+y}.$$

To explain Relation Preservation we need to mention two technical details: the state relation $\approx$ and the judgement $\vdash \overline{\tau} : \textit{safe}$. Judgement $\vdash \overline{\tau} : \textit{safe}$ means that all actions on the trace are tainted as safe (S). Intuitively, the relation $\approx$ relates two states iff their registers and memories locations have the same taint and all elements tainted S have the same values in the two states. We note that we can derive Relation Preservation in a general manner whenever the source semantics enjoy Relation Preservation as well (like our semantics B, $\mathcal{J}$, S, R, and SLS).

The main result of this section is Theorem 3, which states that for any well-formed composition, RSS overapproximates RSNI. We remark that we prove Theorem 3 once and for all by exploiting (1) the well-formedness properties and (2) the fact that all compositions in Figure 2 are well-formed, rather than having to prove the implication for each of the composed semantics.

Theorem 3 (RSS Overapproximates RSNI for Compositions). *If* $\vdash 👻_{x+y} : \textit{WFC}$ *and* $👻_{x+y} \vdash P : \textit{RSS}$*, then* $👻_{x+y} \vdash P : \textit{RSNI}$*.*

Corollary 1 relates the security of a compiler for a well-formed composition with the security of its composing semantics. In particular, if a compiler is *RSSP* for a well-formed composition $👻_{x+y}$, then it is also *RSSP* for the composing semantics $👻_{x}$ and $👻_{y}$. Dually, if a compiler is not *RSSP* for a component, then it is not *RSSP* for any composition.

Corollary 1 (RSSP and compositions, 🐓)**.** *The following statements hold for any well-formed* $👻_{x+y}$*:*

- *If* $👻_{x+y} \vdash [\![\cdot]\!] : \textit{RSSP}$ *then* $👻_{x} \vdash [\![\cdot]\!] : \textit{RSSP}$ *and* $👻_{y} \vdash [\![\cdot]\!] : \textit{RSSP}$*.*
- *If* $👻_{x} \nvdash [\![\cdot]\!] : \textit{RSSP}$ *or* $👻_{y} \nvdash [\![\cdot]\!] : \textit{RSSP}$ *then* $👻_{x+y} \nvdash [\![\cdot]\!] : \textit{RSSP}$*.*

We remark that an analogue of Corollary 1 holds for *RSNIP*.

4 Lifting Compiler Guarantees

Compiler countermeasures against speculative leaks are often developed and proven secure against a specific speculative semantics. For instance, countermeasures against Spectre-PHT have been proven secure against the $👻_{\mathrm{B}}$ semantics [49] modelling speculation over branch instructions. CPUs, however, may employ other speculation mechanisms, whose details might even be unknown when the countermeasure is designed, beyond those originally targeted by the countermeasure.

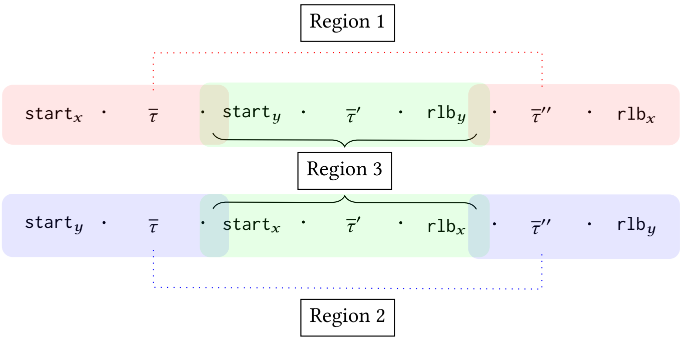
Ensuring the security of a countermeasure hence requires continuously validating their guarantees (e.g., through proofs) against stronger and stronger semantics, as soon as new speculation mechanisms are discovered and modeled. For instance, in the context of Spectre-PHT attacks, countermeasures need to be proved secure against stronger semantics like $👻_{\mathrm{B}+\mathcal{J}+\mathrm{S}+\mathrm{R}}$ rather than $👻_{\mathrm{B}}$ as done in [49]. To reduce the burden of re-proving security whenever a new speculation mechanism is discovered, we need ways of lifting security guarantees from weaker to stronger semantics, which supports more speculations.

In this section, we address this issue by precisely characterizing under which conditions the scope of a countermeasure can be securely extended to other speculation mechanisms. More precisely, we study whenever the security guarantees provided by a compiler targeting a semantics $👻_x$ can be lifted to a stronger semantics $👻_y$, i.e., $👻_x \sqsubseteq 👻_y$. First, we introduce the preconditions for our lifting theorem, i.e., the notions of Independence and Safe Nesting, as well as our main result: the Lifted Compiler Preservation Theorem (Theorem 4) characterizing when security guarantees can be lifted to stronger semantics (Section 4.1). Then, we introduce Syntactic Independence (Section 4.2) and Trapped Speculation (Section 4.3), a set of sufficient conditions for Independence and Safe Nesting respectively. As we show in Section 5, these preconditions can be used in many cases to significantly simplify proofs of Independence and Safe Nesting in practice.

4.1 Lifting Theorem

In this section, we precisely characterize the sufficient conditions for lifting the security guarantees provided by a compiler targeting a semantics $👻_x$ to a stronger semantics $👻_{x+y}$. We start by providing a high-level intuition about how two component semantics $👻_x$ and $👻_y$ can interact when composed as $👻_{x+y}$ (Section 4.1.1). Next, we formalize the notions of Independence (Section 4.1.2) and Safe Nesting (Section 4.1.3), which precisely characterize when a compiler's guarantees can be lifted from $👻_x$ to $👻_{x+y}$. Then, we introduce Conditional Robust Speculative Safety Preservation (CRSSP), a new secure compilation criterion ensuring that RSS is preserved in the composed semantics $👻_{x+y}$ by the compiler for $👻_x$ only for those programs that already satisfy RSS for $👻_y$. We conclude by stating and explaining our lifting theorem (Section 4.1.5) which is the main result of this section.

4.1.1 Interplay of Semantics. To understand the challenges involved in lifting security guarantees from a weaker semantics $👻_x$ to a stronger semantics $👻_{x+y}$, one needs to consider the interactions of $👻_x$ and $👻_y$.



Region 1
$\mathtt{start}_x \cdot \overline{\tau} \cdot \mathtt{start}_y \cdot \overline{\tau}' \cdot \mathtt{rlb}_y \cdot \overline{\tau}'' \cdot \mathtt{rlb}_x$
Region 3
$\mathtt{start}_y \cdot \overline{\tau} \cdot \mathtt{start}_x \cdot \overline{\tau}' \cdot \mathtt{rlb}_x \cdot \overline{\tau}'' \cdot \mathtt{rlb}_y$
Region 2

Fig. 3. Interplay of semantics $👻_x$ and $👻_y$ when executing a program under the combined semantics $👻_{x+y}$.

Figure 3 depicts two portions of traces produced when executing a program under the composed semantics $👻_{x+y}$. The first trace (top) starts with a speculative transaction from semantics $👻_x$ (highlighted in red and starting with action $\mathtt{start}_x$). Inside the red speculative transaction, there is a *nested* speculative transaction from semantics $👻_y$ (highlighted in green, starting with action $\mathtt{start}_y$ and ending with action $\mathtt{rlb}_y$). After the termination of the nested transaction, the outer red transaction continues until the end of its speculative window (action $\mathtt{rlb}_x$). Dually, the second trace (bottom) starts with a speculative transaction from $👻_y$ followed by a nested transaction from $👻_x$.

Thus, there are three different regions that might result in leaks:

- **Region 1**: the speculative transaction started by $👻_x$ (highlighted in red),
- **Region 2**: the speculative transaction started by $👻_y$ (highlighted in blue), and
- **Region 3**: the nested transactions (highlighted in green).

Note that while regions 1 and 2 are generated by a single semantics, the nested speculative transactions in Region 3 only arise in the combined semantics $👻_{x+y}$. While leaks in region 1 are fixed by proving the security of a compiler for the speculative semantics $👻_x$, i.e., $👻_x \vdash [\![\cdot]\!] : \mathit{RSSP}$, we need additional conditions to ensure the absence of leaks in Regions 2 and 3. Thus, we introduce the notions of Independence with respect to a speculative semantics $👻_y$ and Safe Nesting which ensure the absence of leaks in Regions 2 and 3 respectively.

4.1.2 Independence. Intuitively, when trying to lift our compiler security from the speculative semantics $👻_x$ to a stronger semantics $👻_{x+y}$, we need to ensure that the compiler does not introduce new leaks for the extension semantics $👻_y$. The Independence property precisely characterizes this aspect: A compiler $[\![\cdot]\!]$ for the origin semantics $👻_x$ is called independent for the extension semantics $👻_y$ iff the compiler does not introduce further leaks under the extension semantics $👻_y$.

Definition 7 (Independence in Extension).

$$👻_y \vdash [\![\cdot]\!] : I \overset{\text{def}}{=} \forall P.\ \mathit{if}\, 👻_y \vdash P : \mathit{RSS}\ \mathit{then}\ 👻_y \vdash [\![P]\!] : \mathit{RSS}$$

Note that Independence differs from *RSSP* (Definition 4) in that the former uses $👻_y$ in the pre- and post-condition, whereas the latter employs the $👻_{NS}$ in its pre-condition.

As stated in Corollary 2, a compiler that is *RSSP* for $👻_x$ is also Independent w.r.t. this semantics.

Corollary 2 (Self Independence, 🐓). *If* $👻_x \vdash [\![\cdot]\!] : \mathit{RSSP}$*, then* $👻_x \vdash [\![\cdot]\!] : I$.

4.1.3 Safe Nested Speculation. Given a combined semantics $👻_{x+y}$, a program P has Safe Nested Speculations (denoted with $👻_{x+y} \vdash P : \text{safeN}$) if all actions inside nested speculative transactions are safe. Safe nesting, therefore, ensures that there are no unsafe interactions between the composing semantics $👻_x$ and $👻_y$.

Definition 8 (Safe Nested Speculation).

$$\begin{aligned}
👻_{x+y} \vdash P : \mathit{safeN} \overset{\text{def}}{=} \forall \overline{\tau} \in \mathit{Beh}_{x+y}(P),\ \mathit{if}& \\
&\mathtt{start}_a \cdot \overline{\tau}' \cdot \mathtt{rlb}_a\ \mathit{is\ a\ subtrace\ of}\ \overline{\tau}\ \mathit{and}\ \mathtt{start}_b \cdot \overline{\tau}'' \cdot \mathtt{rlb}_b\ \mathit{is\ a\ subtrace\ of}\ \overline{\tau}' \\
&\mathit{then}\ \vdash \overline{\tau}'' : \mathit{safe\ where}\ a \in \{x, y\}\ \mathit{and}\ b \in \{x, y\} \setminus \{a\}
\end{aligned}$$

Finally, we say that a compiler satisfies the Safe Nested Speculation property, written $👻_{x+y} \vdash [\![\cdot]\!] : \text{safeN}$, iff all its compiled programs satisfy Definition 8, i.e., $\forall P.\ 👻_{x+y} \vdash [\![P]\!] : \text{safeN}$.

4.1.4 Conditional Robust Speculative Safety Preservation. Often compilers implementing Spectre countermeasures are developed to prevent leaks introduced by a *specific* speculation mechanism. Hence, when lifting their security guarantees to a semantics that accounts for additional speculation mechanisms, compiled programs might still contain some leaks that the compiler was not designed to prevent in the first place, i.e., leaks caused *only* by the additional mechanisms. However, *RSSP* is too strict of a criterion here, since it does not distinguish between the different speculation mechanisms that cause the leak. To account for this, we propose a new secure compilation criterion called Conditional Robust Speculative Safety Preservation (*CRSSP*, Definition 9). As the name indicates, *CRSSP* is a variant of *RSSP* that restricts RSS preservation *only* to those programs that do not contain leaks caused only by the additional mechanisms.

Definition 9 (Conditional Robust Speculative Safety Preservation (*CRSSP*)).

$$👻_x, 👻_y \vdash [\![\cdot]\!] : \mathit{CRSSP} \overset{\text{def}}{=} \forall P \in L.\ \mathit{if}\, 👻_{NS} \vdash P : \mathit{RSS}\ \mathit{and}\ 👻_y \vdash P : \mathit{RSS},\ \mathit{then}\ 👻_{x+y} \vdash [\![P]\!] : \mathit{RSS}.$$

4.1.5 Lifting Theorem. We are now ready to introduce the main result of this section, that is, our lifting theorem (Theorem 4). Intuitively, we can lift the security guarantees of a compiler $[\![\cdot]\!]$ targeting semantics 👻_x to a stronger WFC semantics 👻_{x+y} for a program P provided that (1) the compiler $[\![\cdot]\!]$ is RSSP for 👻_x, (2) it fulfills Independence for the extension semantics 👻_y, and (3) it fulfills Safe Nesting for the combined semantics 👻_{x+y}.

Theorem 4 (Lifted Compiler Preservation, 🐓). *If* $\text{👻}_x \vdash [\![\cdot]\!] : \mathit{RSSP}$ *and* $\text{👻}_y \vdash [\![\cdot]\!] : I$ *and* $\text{👻}_{x+y} \vdash [\![\cdot]\!] : \mathit{safeN}$ *and* $\vdash \text{👻}_{x+y} : \mathit{WFC}$ *, then* $\text{👻}_x, \text{👻}_y \vdash [\![\cdot]\!] : \mathit{CRSSP}$*.*

We remark that our lifted guarantees hold only for programs P that are initially secure w.r.t. the extension semantics 👻_y. That is, the compiler enjoys $\text{👻}_x, \text{👻}_y \vdash [\![\cdot]\!] : \mathit{CRSSP}$ not the stronger property $\text{👻}_{x+y} \vdash [\![\cdot]\!] : \mathit{RSSP}$. The reason is that while compiler $[\![\cdot]\!]$ does not introduce further leaks under the extension semantics 👻_y (due to Independence), it might not prevent 👻_y-leaks already present in the source program.

For the traces depicted in Figure 3, Theorem 4 ensures RSS (for any program satisfying RSS for the extension semantics 👻_y) under the composed semantics as follows. For Region 1, RSS follows from $\text{👻}_x \vdash [\![\cdot]\!] : \mathit{RSSP}$. For Region 2, RSS follows from the program being originally RSS under 👻_y and from $[\![\cdot]\!]$ fulfilling Independence for extension 👻_y. Finally, for Region 3, RSS follows from the compiled program having Safe Nesting.

Theorem 4 allows us to lift the security guarantees of our secure compilers to stronger semantics, without worrying about unexpected leaks introduced by other speculation mechanisms (captured by the extension semantics) and, crucially, *without* requiring new secure compilation proofs.

Next, we give sufficient conditions for Independence (Section 4.2) and Safe Nesting (Section 4.3).

4.2 Syntactic Independence: Independence for Free

To simplify the task of proving Independence, we now introduce *Syntactic Independence* (SI) (Definition 10), a syntactic sufficient condition for Independence. As the name suggests, SI can be checked by *syntactic* inspection of the compiler $[\![\cdot]\!]$ and of the extension semantics 👻_y.

Before formalizing SI, we introduce some notation. Given a compiler $[\![\cdot]\!]$, we denote by $\mathit{injInst}([\![\cdot]\!])$ the set of instructions that the compiler inserts during compilation. For instance, for a simple compiler $[\![\cdot]\!]^f_{\mathbf{B}}$ that inserts **spbarr** instructions after branch instructions to prevent speculation [49], $\mathit{injInst}([\![\cdot]\!]^f_{\mathbf{B}})$ is the set $\{\mathbf{spbarr}\}$. Given a semantics 👻, we denote by $\mathit{specInst}(\text{👻})$ the set of instructions that trigger speculation in 👻. For instance, for semantics 👻_{SLS}, which models straight-line speculation over return instructions, the set $\mathit{specInst}(\text{👻}_{\mathsf{SLS}})$ is $\{\mathbf{ret}\}$.

We are now ready to formalize Syntactic Independence (Definition 10). In a nutshell, a compiler $[\![\cdot]\!]$ is syntactically independent for a semantics 👻 (denoted with $\text{👻} \vdash [\![\cdot]\!] : \mathit{SI}$) if the compiler does not insert (1) any instructions that trigger speculation under 👻, and (2) any instructions that produce data-dependent actions or modify the program state (except for the program counter **pc** and the stack pointer **sp**). The first requirement ensures that the compiler does not introduce new (potentially unsafe) speculative transactions, whereas the second requirement ensures that the compiler does not introduce unsafe actions into existing safe speculative transactions under 👻.

Definition 10 (Syntactic Independence)**.**

$$\text{👻} \vdash [\![\cdot]\!] : \mathit{SI} \overset{\text{def}}{=} \mathit{injInst}([\![\cdot]\!]) \cap \mathit{specInst}(\text{👻}) = \emptyset \ \mathit{and}\ \mathit{injInst}([\![\cdot]\!]) \cap \{\mathbf{beqz}, \mathbf{jmp}, \mathbf{store}, \mathbf{load}, \longleftarrow\} = \emptyset$$

For instance, the compiler $[\![\cdot]\!]^f_{\mathbf{B}}$ mentioned above is SI w.r.t. 👻_{SLS} since $\mathit{injInst}([\![\cdot]\!]^f_{\mathbf{B}}) \cap \mathit{specInst}(\text{👻}_{\mathsf{SLS}}) = \{\mathbf{spbarr}\} \cap \{\mathbf{ret}\} = \emptyset$ and $\{\mathbf{spbarr}\} \cap \{\mathbf{beqz}, \mathbf{jmp}, \mathbf{store}, \mathbf{load}, \longleftarrow\} = \emptyset$.

Lemma 1 connects Syntactic Independence and Independence.

Lemma 1 (SI Implies Independence)**.** *If* $\text{👻}_y \vdash [\![\cdot]\!] : SI$*, then* $\text{👻}_y \vdash [\![\cdot]\!] : I$*.*

We remark that checking SI is significantly simpler than manually proving Independence. As we show in Section 5.3, SI plays a critical role in reducing the amount of Independence proofs necessary to carry out our security analysis. Despite its restrictiveness, SI is applicable to two classes of compiler countermeasures—`lfence`-based countermeasures [38] and return-trampoline countermeasures [36]—that work by stalling speculative execution. For more complex countermeasures, e.g., speculative load hardening [16], that aim at preventing speculative leaks (rather than preventing speculation altogether), SI is not applicable since the compiler might instrument the program with additional instructions that modify the program state. In this case, in our security analysis we fall-back to standard Independence proofs.

4.3 Trapped Speculation: Fulfilling Safe Nesting

Showing that the compiled program fulfils the Safe Nesting condition is challenging because it requires reasoning about both the compiler as well as the interactions between component semantics. To help with this, we now introduce a sufficient condition on compilers that ensures that all compiled programs enjoy the Safe Nesting property.

Definition 11 (Trapped Speculation of Compiler)**.**

$$\text{👻}_x \vdash [\![\cdot]\!] : \mathit{trappedSpec} \overset{\text{def}}{=} \forall P, \overline{\tau} \in \mathit{Beh}_x([\![P]\!]), \tau \in \overline{\tau}\upharpoonright_{se}.\ \exists n.\ \tau = \texttt{rlb}_x\ n \ \mathit{or}\ \tau = \texttt{start}_x\ n$$

In a nutshell, a compiler satisfies Definition 11 iff it *traps speculation*, which we model by requiring that the only speculative actions τ produced by compiled programs are either $\texttt{start}_x\ n$ (i.e., beginning of speculation) or $\texttt{rlb}_x\ n$ (i.e., end of speculation). This, thus, implies that there are no unsafe actions between the start of a speculation transaction and its rollback as required by Safe Nesting. For example, a compiler inserting fences into the program stops speculation immediately and fulfils our definition of Trapped Speculation. Similarly, a compiler inserting a so-called return trampoline [36] (which traps speculation in a loop) also fulfills Trapped Speculation.

Definition 11 relies on the speculative projection function $\upharpoonright_{se}$, which removes all non-speculative observations from the trace and is defined as the inverse of the non-speculative projection $\upharpoonright_{n}$.

Lemma 2 connects Trapped Speculation (Definition 11) with Safe Nesting (Definition 8).

Lemma 2 (Trapped Speculation Implies Safe Nesting, 🐓)**.** *If* $\text{👻}_x \vdash [\![\cdot]\!] : \mathit{trappedSpec}$ *then* $\text{👻}_{x+y} \vdash [\![\cdot]\!] : \mathit{safeN}$*.*

As we show in Section 5.4, Definition 11 significantly reduces the proof burden. In particular, rather than having to reason about the combined semantics 👻_{x+y} when showing Safe Nesting, we can just reason about the compiler $[\![\cdot]\!]$ for semantics 👻_x.

With this formal setup, we now move on to our security analysis, which demonstrates how to attain *CRSSP* for a number of countermeasures, by relying on the notions of Independence, Syntactic Independence, Safe Nesting, and Trapped Speculation.

5 Countermeasures Analysis

In this section, we present a comprehensive analysis of the security guarantees provided by Spectre countermeasures implemented in major compilers. Our analysis covers 9 countermeasures (summarized in Section 5.1) and 5 classes of Spectre attacks: Spectre-PHT [39], Spectre-BTB [39], Spectre-RSB [41, 43], Spectre-STL [35], and Spectre-SLS [7]. Using our secure compilation framework, we precisely characterize the security guarantees provided by these countermeasures against the five speculative semantics from Section 2 and their combinations.

Table 2. Analyzed compiler countermeasures.

Name	Symbol	Base Semantics	Source
Fences for Returns Straight-Line	$[\![\cdot]\!]^{f}_{\mathsf{SLS}}$	👻$_{\mathsf{SLS}}$	GCC/Clang
Retpoline for Jumps	$[\![\cdot]\!]^{rpl}_{J}$	👻$_{J}$	GCC/Clang/[36]
Retpoline with fence for Jumps	$[\![\cdot]\!]^{rplf}_{J}$	👻$_{J}$	Gcc/[36]
Retpoline for Returns	$[\![\cdot]\!]^{rpl}_{\mathsf{R}}$	👻$_{\mathsf{R}}$	Gcc/[43]
Fences for Returns	$[\![\cdot]\!]^{f}_{\mathsf{R}}$	👻$_{\mathsf{R}}$	[43]
Fences for Stores	$[\![\cdot]\!]^{f}_{\mathsf{S}}$	👻$_{\mathsf{S}}$	[37]
Ultimate SLH for Branches	$[\![\cdot]\!]^{USLH}_{\mathbf{B}}$	👻$_{\mathbf{B}}$	[59] (extends Clang's SLH)
Strong SLH for Branches	$[\![\cdot]\!]^{SSLH}_{\mathbf{B}}$	👻$_{\mathbf{B}}$	[49] (extends Clang's SLH)
Fences for Branches	$[\![\cdot]\!]^{f}_{\mathbf{B}}$	👻$_{\mathbf{B}}$	ICC/Clang

Theorem 4 plays a key role in this analysis since we use it to lift security guarantees from simpler semantics to their combinations, thereby significantly reducing the number of secure compilation proofs that need to be carried out. We remark that to apply the lifting theorem, which allows us to lift the security guarantees of the compiler to stronger semantics, we need to show (1) Security in Source, (2) Independence in Extension, and (3) Safe Nesting, which is what we do next.

First, we tackle Security in Source and prove that the compilers are *RSSP* w.r.t. their base speculative semantics (Section 5.2). Then we focus on Independence, and show Syntactic Independence for four of our compilers and fall-back to full Independence proofs for the remaining five (Section 5.3). Next, we analyze Safe Nesting and show that Trapped Speculation (Definition 11) applies in all cases except two (Section 5.4). Finally, we combine these results by evaluating the strongest security guarantees that can be achieved for these compilers using Theorem 4 (Section 5.5).

5.1 The Compilers

Table 2 summarizes the Spectre countermeasures that we analyze. These countermeasures are often implemented as compilation passes at the end of the compilation process (e.g., SLH is a `MachineFunctionPass` in Clang). Next, we describe the compilers in more detail and refer to our technical report for their full definitions.

Fences Against Straight-Line Speculation ($[\![\cdot]\!]^{f}_{\mathsf{SLS}}$). Modern CPUs can speculatively bypass **ret** instructions [5, 7]. Compilers like Gcc and Clang (with option -mharden-sls=all) prevent this by injecting a speculation barrier after every **ret** instruction. Since the **ret** instruction is an unconditional change in control flow, the barrier[4] will not be executed architecturally but only when straight-line speculation is happening. We model this countermeasure in the $[\![\cdot]\!]^{f}_{\mathsf{SLS}}$ compiler that inserts a barrier after every **ret** instruction.

Retpoline for Indirect Jumps ($[\![\cdot]\!]^{rpl}_{J}$, $[\![\cdot]\!]^{rplf}_{J}$). Spectre-BTB attacks [39] exploit speculation over indirect jumps. The retpoline countermeasure [36] replaces all indirect jumps in the code with a return trampoline, i.e., with a construct that traps the speculation in an infinite loop. Retpoline is available in all general compilers like Clang (-mretpoline) and Gcc (with option -mindirect-branch)

[4] For x86, the int3 single-byte instruction is used to reduce the binary size.

and is widely deployed because current developed hardware mitigations are not enough to protect against indirect jump speculation [10]. We consider two models of the retpoline countermeasure. The $[\![\cdot]\!]^{rpl}_{J}$ compiler replaces every indirect jump instruction **jmp** e with a return trampoline. Additionally, we consider the $[\![\cdot]\!]^{rplf}_{J}$ compiler, which inserts an additional fence after **ret** instructions in trampolines (to prevent straight-line speculation). This compiler corresponds to activating both flags -mindirect-branch and -mharden-sls=all in Gcc.

Retpoline for Returns ($[\![\cdot]\!]^{rpl}_{\mathsf{R}}$)*.* A variant of the retpoline countermeasure has been proposed to prevent Spectre-RSB attacks [43]. This countermeasure (implemented in Gcc with option -mfunction-return) replaces each **ret** instruction with a return trampoline; trapping misprediction caused by **ret** instructions. We modeled this countermeasure in the $[\![\cdot]\!]^{rpl}_{\mathsf{R}}$ compiler.

Fences for Returns ($[\![\cdot]\!]^{f}_{\mathsf{R}}$)*.* Maisuradze and Rossow [43] propose to add an **lfence** instruction after every **call** instruction. This ensures that mis-speculations over **ret** instructions involving the Return Stack Buffer will always land on one of the injected speculation barriers, thereby preventing speculative leaks. We model this countermeasure in the $[\![\cdot]\!]^{f}_{\mathsf{R}}$ compiler, which replaces every **call** f instruction with **call** f; **spbarr**.

Fences for Stores ($[\![\cdot]\!]^{f}_{\mathsf{S}}$)*.* To prevent speculation over store-to-load bypasses [37] (also known as Spectre-STL), Intel suggested to insert the **lfence** instruction after every store, thereby ensuring that all stores are committed to main memory and preventing speculation. However, no mainstream compiler implements this countermeasure due to the high performance overhead. We model this countermeasure in the $[\![\cdot]\!]^{f}_{\mathsf{S}}$ compiler, which replaces (1) every **store** x, e instruction with **store** x, e; **spbarr**, and (2) every $\mathbf{store}_{prv}$ x, e instruction with $\mathbf{store}_{prv}$ x, e; **spbarr**.

(Strong) Speculative Load Hardening (SSLH, $[\![\cdot]\!]^{SSLH}_{\mathsf{B}}$*).* Modern CPUs speculate over the outcome of branch instructions [39]. Clang (with option -mspeculative-load-hardening) protects against these speculative leaks by (1) using a speculation flag that tracks whenever misprediction is currently happening or not, and (2) using the flag to conditionally mask loads and stores to prevent the leaks [16]. Patrignani and Guarnieri [49] investigated the security of SLH and showed it insecure with respect to 👻$_{\mathsf{B}}$. They proposed an improved version called strong-SLH and prove it secure with respect to 👻$_{\mathsf{B}}$ semantics. We evaluate their compiler[5] $[\![\cdot]\!]^{SSLH}_{\mathsf{B}}$ in our framework to see if we can lift the security guarantees to stronger semantics.

Ultimate Speculative Load Hardening (USLH, $[\![\cdot]\!]^{USLH}_{\mathsf{B}}$*).* Zhang et al. [59] showed that variable-latency arithmetic instructions can leak secret information under speculation, and this is not prevented by speculative-load hardening [16] or by its strong-variant $[\![\cdot]\!]^{SSLH}_{\mathsf{B}}$ [49]. To prevent these speculative leaks, they propose the "ultimate speculative load hardening" compiler, which extends strong-SLH by additionally masking inputs to variable-latency arithmetic instructions.

We modelled the core aspects of ultimate USLH in the $[\![\cdot]\!]^{USLH}_{\mathsf{B}}$ compiler. For this, we extended μAsm to support a dedicated instruction $x \leftarrow_{VL} y\ op\ z$ denoting variable-latency computations. Furthermore, we extended μAsm events to include a new observation $y\ \mathsf{op}\ z$ that is emitted by the new instruction $x \leftarrow_{VL} y\ op\ z$.

$$\textit{Instructions}\ i ::= \cdots \mid x \leftarrow_{VL} y\ op\ z \qquad\qquad \mu\textit{arch. Acts.}\ \delta ::= \cdots \mid x\ \mathsf{op}\ y$$

[5] Technically, Patrignani and Guarnieri [49] targeted a While language while we have an assembly-like language μAsm. However, the translation is straightforward.

These extensions augment the constant time observer (used in our model in Section 2.1 as well as in the strong-SLH formalization in [49]) to capture leaks related to variable-latency instructions. We denote the 👻_{B} semantics extended with the new observer as $\text{👻}_{\mathsf{B}}^{ct+vl}$, and we have the following leakage ordering: $\text{👻}_{\mathsf{B}}^{ct} \sqsubseteq \text{👻}_{\mathsf{B}}^{ct+vl}$.

Fences for Branches ($[\![\cdot]\!]_{\mathsf{B}}^{f}$). Another approach to prevent leaks due to branch misprediction is injecting `lfence` instructions after branch instructions. Compilers like Intel ICC (with flag -mconditional-branch=all-fix) and Clang (with flag -x86-speculative-load-hardening-lfence) implement this countermeasure. This countermeasure was already modelled (and proved secure for 👻_{B}) in [49] as $[\![\cdot]\!]_{\mathsf{B}}^{f}$[5], which replaces every **beqz** x, l with **beqz** x, l; **spbarr** and we want to investigate the applicability of the lifting theorem to $[\![\cdot]\!]_{\mathsf{B}}^{f}$'s security guarantees as well.

5.2 Security of the Compilers

Here, we report the results of the security analysis of each compiler with respect to their base speculative semantics, as indicated in Table 2. Theorem 5 states that each compiler is *RSSP* w.r.t. its base speculative semantics. Even though we present all results altogether, we remark that each point in Theorem 5 corresponds to an independent secure compilation proof.

Theorem 5 (Compiler Security). *The following statements hold:*

- *(SLS: Fence is secure)* $\text{👻}_{\mathsf{SLS}} \vdash [\![\cdot]\!]_{\mathsf{SLS}}^{f} : RSSP$
- *(R: Retpoline is secure)* $\text{👻}_{\mathsf{R}} \vdash [\![\cdot]\!]_{\mathsf{R}}^{rpl} : RSSP$
- *(R: Fence is secure)* $\text{👻}_{\mathsf{R}} \vdash [\![\cdot]\!]_{\mathsf{R}}^{f} : RSSP$
- *(J: Retpoline is secure)* $\text{👻}_{J} \vdash [\![\cdot]\!]_{\mathcal{J}}^{rpl} : RSSP$
- *(J: Retpoline with fence is secure)* $\text{👻}_{J} \vdash [\![\cdot]\!]_{\mathcal{J}}^{rplf} : RSSP$
- *(S: Fence is secure)* $\text{👻}_{\mathsf{S}} \vdash [\![\cdot]\!]_{\mathsf{S}}^{f} : RSSP$
- *(B: USLH is secure)* $\text{👻}_{\mathsf{B}}^{ct+vl} \vdash [\![\cdot]\!]_{\mathsf{B}}^{USLH} : RSSP$
- *(B: SSLH is secure [49])* $\text{👻}_{\mathsf{B}} \vdash [\![\cdot]\!]_{\mathsf{B}}^{SSLH} : RSSP$
- *(B: Fence is secure [49])* $\text{👻}_{\mathsf{B}} \vdash [\![\cdot]\!]_{\mathsf{B}}^{f} : RSSP$

5.3 Independence of the Compilers

We now investigate whether our compilers satisfy the Independence in Extension condition. Due to space constraints, we do not report on the Independence and Syntactic Independence for all our compilers and combinations and refer the interested reader to our companion report. Instead, Theorem 6 reports the strongest possible semantics for which we can prove (Syntactic) Independence for each compiler.

Theorem 6 (Compiler Independence). *The following statements hold:*

- *(SLS: Fence Independence)* $\text{👻}_{\mathsf{B}+J+\mathsf{S}+\mathsf{SLS}} \vdash [\![\cdot]\!]_{\mathsf{SLS}}^{f} : SI$
- *(R: Fence Independence)* $\text{👻}_{\mathsf{B}+J+\mathsf{S}+\mathsf{R}} \vdash [\![\cdot]\!]_{\mathsf{R}}^{f} : SI$
- *(S: Fence Independence)* $\text{👻}_{\mathsf{B}+J+\mathsf{S}+\mathsf{R}} \vdash [\![\cdot]\!]_{\mathsf{S}}^{f} : SI$ *and* $\text{👻}_{\mathsf{B}+J+\mathsf{S}+\mathsf{SLS}} \vdash [\![\cdot]\!]_{\mathsf{S}}^{f} : SI$
- *(B: Fence Independence)* $\text{👻}_{\mathsf{B}+J+\mathsf{S}+\mathsf{R}} \vdash [\![\cdot]\!]_{\mathsf{B}}^{f} : SI$ *and* $\text{👻}_{\mathsf{B}+J+\mathsf{S}+\mathsf{SLS}} \vdash [\![\cdot]\!]_{\mathsf{B}}^{f} : SI$
- *(B: USLH Independence)* $\text{👻}_{\mathsf{B}+J+\mathsf{S}+\mathsf{R}} \vdash [\![\cdot]\!]_{\mathsf{B}}^{USLH} : I$ *and* $\text{👻}_{\mathsf{B}+J+\mathsf{S}+\mathsf{SLS}} \vdash [\![\cdot]\!]_{\mathsf{B}}^{USLH} : I$
- *(B: SSLH Independence)* $\text{👻}_{\mathsf{B}+J+\mathsf{S}+\mathsf{R}} \vdash [\![\cdot]\!]_{\mathsf{B}}^{SSLH} : I$ *and* $\text{👻}_{\mathsf{B}+J+\mathsf{S}+\mathsf{SLS}} \vdash [\![\cdot]\!]_{\mathsf{B}}^{SSLH} : I$
- *(R: Retpoline Independence)* $\text{👻}_{\mathsf{B}+J+\mathsf{S}+\mathsf{R}} \vdash [\![\cdot]\!]_{\mathsf{R}}^{rpl} : I$

- *($\mathcal{J}$: Retpoline Independence)* $\text{👻}_{\mathsf{B+J+S+R}} \vdash [\![\cdot]\!]^{rpl}_{\mathcal{J}} : I$
- *($\mathcal{J}$: Retpoline with Fence Independence)* $\text{👻}_{\mathsf{B+J+S+R}} \vdash [\![\cdot]\!]^{rplf}_{\mathcal{J}} : I$ *and* $\text{👻}_{\mathsf{B+J+S+SLS}} \vdash [\![\cdot]\!]^{rplf}_{\mathcal{J}} : I$

As stated in Theorem 6, for compilers $[\![\cdot]\!]^{f}_{\mathsf{SLS}}$, $[\![\cdot]\!]^{f}_{\mathsf{R}}$, $[\![\cdot]\!]^{f}_{\mathsf{S}}$, and $[\![\cdot]\!]^{f}_{\mathsf{B}}$, we directly proved that Syntactic Independence holds even for the strongest possible combined semantics. Given that Syntactic Independence implies Independence (Lemma 1), this allows us to derive Independence results for all these compilers through simple syntactic checks.

For the $[\![\cdot]\!]^{rpl}_{\mathcal{J}}$, $[\![\cdot]\!]^{rplf}_{\mathcal{J}}$ and $[\![\cdot]\!]^{rpl}_{\mathsf{R}}$ compilers, instead, we have to fall back to a full Independence proof for the strongest semantics. The reason is that these compilers add **ret** instructions to the code which could interact with R or SLS, i.e., $\mathit{injInst}([\![\cdot]\!]) \cap \mathit{specInst}(\text{👻}_{\mathsf{R}}) = \{\mathbf{ret}\}$, which violates SI. However, for weaker combinations not including R or SLS, SI applies.

We also remark that Independence does not hold for the retpoline compiler $[\![\cdot]\!]^{rpl}_{\mathcal{J}}$ and the straight-line speculation semantics 👻_{SLS}, i.e., $\text{👻}_{\mathsf{SLS}} \nvdash [\![\cdot]\!]^{rpl}_{\mathcal{J}} : I$. The reason is that the **ret** instructions injected by $[\![\cdot]\!]^{rpl}_{\mathcal{J}}$ as part of return trampolines can be speculatively bypassed under 👻_{SLS}. The additional speculation barrier injected by the strengthened compiler $[\![\cdot]\!]^{rplf}_{\mathcal{J}}$ fixes this issue and allows recovering Independence w.r.t. SLS as well, i.e., $\text{👻}_{\mathsf{SLS}} \vdash [\![\cdot]\!]^{rplf}_{\mathcal{J}} : I$.

Finally, for the SLH compilers $[\![\cdot]\!]^{USLH}_{\mathsf{B}}$ and $[\![\cdot]\!]^{SSLH}_{\mathsf{B}}$, we again had to perform full Independence proofs since these compilers inject instructions, which violate SI requirements, for tracking the speculation flag and for masking **store** and **load** instructions.

5.4 Safe Nesting of the Compilers

The last condition to fulfill for lifting security guarantees is Safe Nesting. Rather than directly proving Safe Nesting, we study which compilers trap speculation according to Definition 11. Theorem 7 precisely characterize which compilers enjoy this property. Combining Theorem 7 with the fact that trapped speculation implies Safe Nesting (Lemma 2), gives us a precise characterization of which compilers enjoy the Safe Nesting property.

Theorem 7 (Compiler Safe Nesting). *The following statements hold:*

- *(SLS: Fence traps speculation)* $\text{👻}_{\mathsf{SLS}} \vdash [\![\cdot]\!]^{f}_{\mathsf{SLS}} : \mathit{trappedSpec}$
- *(R: Retpoline traps speculation)* $\text{👻}_{\mathsf{R}} \vdash [\![\cdot]\!]^{rpl}_{\mathsf{R}} : \mathit{trappedSpec}$
- *(R: Fence traps speculation)* $\text{👻}_{\mathsf{R}} \vdash [\![\cdot]\!]^{f}_{\mathsf{R}} : \mathit{trappedSpec}$
- *($\mathcal{J}$: Retpoline traps speculation)* $\text{👻}_{\mathsf{J}} \vdash [\![\cdot]\!]^{rpl}_{\mathcal{J}} : \mathit{trappedSpec}$
- *($\mathcal{J}$: Retpoline with fence traps speculation)* $\text{👻}_{\mathsf{J}} \vdash [\![\cdot]\!]^{rplf}_{\mathcal{J}} : \mathit{trappedSpec}$
- *(S: Fence traps speculation)* $\text{👻}_{\mathsf{S}} \vdash [\![\cdot]\!]^{f}_{\mathsf{S}} : \mathit{trappedSpec}$
- *(B: Fence traps speculation)* $\text{👻}_{\mathsf{B}} \vdash [\![\cdot]\!]^{f}_{\mathsf{B}} : \mathit{trappedSpec}$

For all our compilers that rely on inserting **spbarr** as a countermeasure, proving Trapped Speculation was easy since the speculation barriers immediately stop any speculative transaction. In contrast, for the compilers that inject retpolines, Trapped Speculation follows from the fact that the return-trampoline traps speculation in a loop that does not produce visible events.

Trapped Speculation, however, does not hold for SLH-based countermeasures because these countermeasures do not block speculative execution but rather prevent leaks during speculation. For $[\![\cdot]\!]^{SSLH}_{\mathsf{B}}$ and $[\![\cdot]\!]^{USLH}_{\mathsf{B}}$, therefore, we need to directly prove Safe Nesting for each combined semantics, which requires reasoning about all interactions of component semantics. Theorem 8 reports the strongest semantics for which safe nesting holds.

Theorem 8 (Compiler Safe Nesting). *The following statements hold:*

- *(B: SSLH Safe Nesting)* $\text{👻}_{\mathsf{B+S+R}} \vdash \llbracket\cdot\rrbracket^{SSLH}_{\mathsf{B}} : safeN$ *and* $\text{👻}_{\mathsf{B+S+SLS}} \vdash \llbracket\cdot\rrbracket^{SSLH}_{\mathsf{B}} : safeN$
- *(B: USLH Safe Nesting)* $\text{👻}^{ct+vl}_{\mathsf{B+S+R}} \vdash \llbracket\cdot\rrbracket^{USLH}_{\mathsf{B}} : safeN$ *and* $\text{👻}^{ct+vl}_{\mathsf{B+S+SLS}} \vdash \llbracket\cdot\rrbracket^{USLH}_{\mathsf{B}} : safeN$

Intuitively, Safe Nesting holds for the cases in Theorem 8 because the SLH compilers preserve the invariant that the speculation flag is correctly set even when considering speculative transactions caused by semantics like 👻_{S} or 👻_{R}.

In contrast, Safe Nesting does not hold for combinations including $\text{👻}_{\mathcal{J}}$. This follows from the fact that SSLH/USLH compilers do not correctly propagate the value of the speculation flag during indirect jumps. As we state in Theorem 9, this also leads to both $\llbracket\cdot\rrbracket^{SSLH}_{\mathsf{B}}$ and $\llbracket\cdot\rrbracket^{USLH}_{\mathsf{B}}$ being insecure w.r.t. any combination of semantics containing $\text{👻}_{\mathcal{J}}$ and 👻_{B}, i.e., any speculative semantics supporting branch and indirect jump speculation.

Theorem 9 (SLH Insecurity w.r.t $\mathcal{J}$). *For any speculative semantics* 👻 *such that* $\text{👻}_{\mathsf{B}+\mathcal{J}} \sqsubseteq \text{👻}$, $\text{👻} \nvdash \llbracket\cdot\rrbracket^{SSLH}_{\mathsf{B}} : RSNIP$ *and* $\text{👻} \nvdash \llbracket\cdot\rrbracket^{USLH}_{\mathsf{B}} : RSNIP.$

We prove Theorem 9 by finding a source program (which we present in the companion report for space constraints), that after compilation with $\llbracket\cdot\rrbracket^{SSLH}_{\mathsf{B}}/\llbracket\cdot\rrbracket^{USLH}_{\mathsf{B}}$ still leaks due to indirect jump speculation. The gist of the insecure program is that while the target of the indirect jump is masked by $\llbracket\cdot\rrbracket^{SSLH}_{\mathsf{B}}/\llbracket\cdot\rrbracket^{USLH}_{\mathsf{B}}$ (to prevent leaks), this does not prevent speculation of indirect jumps that can be used to bypass the instructions tracking the speculation flag inserted by the compiler. Note that this issue could be fixed by relying on hardware support for control-flow-integrity like Intel-CET [52] or ARM-BTI [8], which restrict the targets of indirect jumps to a fixed set of addresses *even under speculation*. This should be sufficient to ensure that attackers cannot speculatively bypass instructions setting the speculation flags and it should allow us to derive Safe Nesting for the SLH compilers w.r.t. combinations including $\text{👻}_{\mathcal{J}}$. We leave investigating this for future work.

5.5 Lifting Security Guarantees

We conclude our security analysis by using the results from Sections 5.2–5.4 together with Theorem 4 to study how far we can lift the security guarantees provided by each compiler. This allows us to precisely characterize the security of these countermeasures even under stronger semantics.

Theorem 10 summarizes the strongest lifted security guarantees one can derive for each of the studied compilers using our lifting theorem (Theorem 4). After an explanation of the result, we present it visually in Figure 4.

Theorem 10 (Lifted security guarantees). *The following statements hold:*

- *(S: Fence)* $\text{👻}_{\mathsf{S}}, \text{👻}_{\mathsf{B}+\mathcal{J}+\mathsf{R}} \vdash \llbracket\cdot\rrbracket^{f}_{\mathsf{S}} : CRSSP$ *and* $\text{👻}_{\mathsf{S}}, \text{👻}_{\mathsf{B}+\mathcal{J}+\mathsf{SLS}} \vdash \llbracket\cdot\rrbracket^{f}_{\mathsf{S}} : CRSSP$
- *(SLS: Fence)* $\text{👻}_{\mathsf{SLS}}, \text{👻}_{\mathsf{B}+\mathcal{J}+\mathsf{S}} \vdash \llbracket\cdot\rrbracket^{f}_{\mathsf{SLS}} : CRSSP$
- *(R: Fence)* $\text{👻}_{\mathsf{R}}, \text{👻}_{\mathsf{B}+\mathcal{J}+\mathsf{S}} \vdash \llbracket\cdot\rrbracket^{f}_{\mathsf{R}} : CRSSP$
- *($\mathcal{J}$: Retpoline)* $\text{👻}_{\mathcal{J}}, \text{👻}_{\mathsf{B+S+R}} \vdash \llbracket\cdot\rrbracket^{rpl}_{\mathcal{J}} : CRSSP$
- *($\mathcal{J}$: Retpoline with fences)* $\text{👻}_{\mathcal{J}}, \text{👻}_{\mathsf{B+S+R}} \vdash \llbracket\cdot\rrbracket^{rplf}_{\mathcal{J}} : CRSSP$ *and* $\text{👻}_{\mathcal{J}}, \text{👻}_{\mathsf{B+S+SLS}} \vdash \llbracket\cdot\rrbracket^{rplf}_{\mathcal{J}} : CRSSP$
- *(R: Retpoline)* $\text{👻}_{\mathsf{R}}, \text{👻}_{\mathsf{B}+\mathcal{J}+\mathsf{S}} \vdash \llbracket\cdot\rrbracket^{rpl}_{\mathsf{R}} : CRSSP$
- *(B: Fence)* $\text{👻}_{\mathsf{B}}, \text{👻}_{\mathcal{J}+\mathsf{S}+\mathsf{R}} \vdash \llbracket\cdot\rrbracket^{f}_{\mathsf{B}} : CRSSP$ *and* $\text{👻}_{\mathsf{B}}, \text{👻}_{\mathcal{J}+\mathsf{S}+\mathsf{SLS}} \vdash \llbracket\cdot\rrbracket^{f}_{\mathsf{B}} : CRSSP$
- *(B: USLH)* $\text{👻}^{ct+vl}_{\mathsf{B}}, \text{👻}^{ct+vl}_{\mathsf{S+R}} \vdash \llbracket\cdot\rrbracket^{USLH}_{\mathsf{B}} : CRSSP$ *and* $\text{👻}^{ct+vl}_{\mathsf{B}}, \text{👻}^{ct+vl}_{\mathsf{S+SLS}} \vdash \llbracket\cdot\rrbracket^{USLH}_{\mathsf{B}} : CRSSP$

- *(B: SSLH)* $\text{👻}_{\mathsf{B}}, \text{👻}_{\mathsf{S+R}} \vdash \llbracket\cdot\rrbracket^{SSLH}_{\mathsf{B}} : CRSSP$ *and* $\text{👻}_{\mathsf{B}}, \text{👻}_{\mathsf{S+SLS}} \vdash \llbracket\cdot\rrbracket^{SSLH}_{\mathsf{B}} : CRSSP$

For all our compilers except the SLH ones, we can lift the security guarantees up to $\text{👻}_{\mathsf{B}+\mathit{J}+\mathsf{S+R}}$ or $\text{👻}_{\mathsf{B}+\mathit{J}+\mathsf{S+SLS}}$, the two strongest combined speculative semantics (as shown in Figure 2), i.e., the strongest attacker models considered in our paper.

For the SLH compilers $\llbracket\cdot\rrbracket^{USLH}_{\mathsf{B}}$ and $\llbracket\cdot\rrbracket^{SSLH}_{\mathsf{B}}$, we are able to lift the security guarantees only up to $\text{👻}_{\mathsf{B+S+R}}$ and $\text{👻}_{\mathsf{B+S+SLS}}$. Lifting the guarantees of $\llbracket\cdot\rrbracket^{USLH}_{\mathsf{B}}$ and $\llbracket\cdot\rrbracket^{SSLH}_{\mathsf{B}}$ to stronger semantics including 👻_{J} is not possible because Safe Nesting is not fulfilled (cf. Section 5.4).

Finally, for $\llbracket\cdot\rrbracket^{f}_{\mathsf{SLS}}$, $\llbracket\cdot\rrbracket^{f}_{\mathsf{R}}$ and $\llbracket\cdot\rrbracket^{rpl}_{\mathsf{R}}$, we cannot lift security to both $\text{👻}_{\mathsf{B}+\mathit{J}+\mathsf{S+R}}$ and $\text{👻}_{\mathsf{B}+\mathit{J}+\mathsf{S+SLS}}$ because we cannot compose 👻_{R} and 👻_{SLS} due to limitations of the combination framework (cf. Section 2.4).

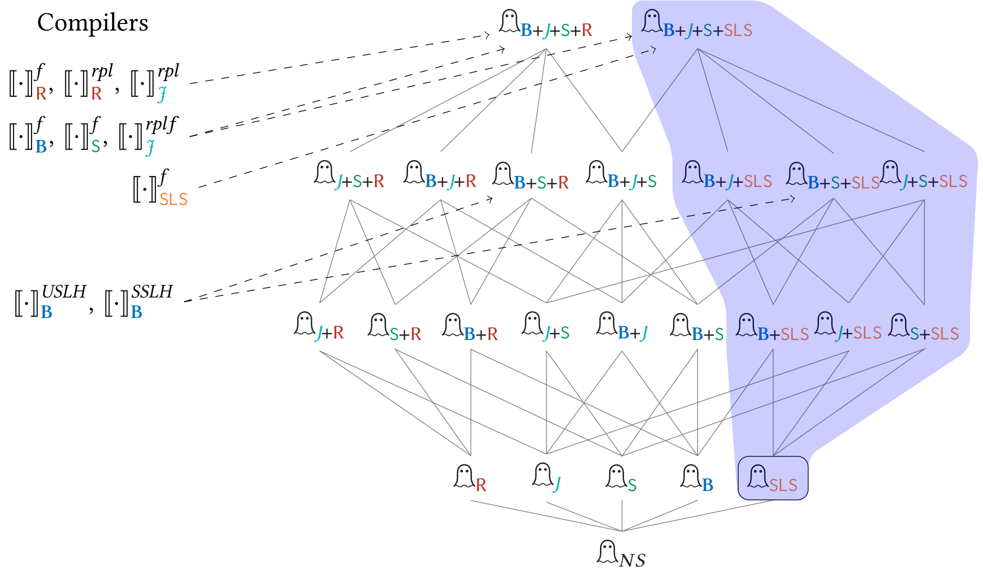

Compilers
$\llbracket\cdot\rrbracket^{f}_{\mathsf{R}}$, $\llbracket\cdot\rrbracket^{rpl}_{\mathsf{R}}$, $\llbracket\cdot\rrbracket^{rpl}_{\mathit{J}}$
$\llbracket\cdot\rrbracket^{f}_{\mathsf{B}}$, $\llbracket\cdot\rrbracket^{f}_{\mathsf{S}}$, $\llbracket\cdot\rrbracket^{rplf}_{\mathit{J}}$
$\llbracket\cdot\rrbracket^{f}_{\mathsf{SLS}}$
$\llbracket\cdot\rrbracket^{USLH}_{\mathsf{B}}$, $\llbracket\cdot\rrbracket^{SSLH}_{\mathsf{B}}$
$\text{👻}_{\mathsf{B}+\mathit{J}+\mathsf{S+R}}$
$\text{👻}_{\mathsf{B}+\mathit{J}+\mathsf{S+SLS}}$
$\text{👻}_{\mathit{J}+\mathsf{S+R}}$
$\text{👻}_{\mathsf{B}+\mathit{J}+\mathsf{R}}$
$\text{👻}_{\mathsf{B+S+R}}$
$\text{👻}_{\mathsf{B}+\mathit{J}+\mathsf{S}}$
$\text{👻}_{\mathsf{B}+\mathit{J}+\mathsf{SLS}}$
$\text{👻}_{\mathsf{B+S+SLS}}$
$\text{👻}_{\mathit{J}+\mathsf{S+SLS}}$
$\text{👻}_{\mathit{J}+\mathsf{R}}$
$\text{👻}_{\mathsf{S+R}}$
$\text{👻}_{\mathsf{B+R}}$
$\text{👻}_{\mathit{J}+\mathsf{S}}$
$\text{👻}_{\mathsf{B}+\mathit{J}}$
$\text{👻}_{\mathsf{B+S}}$
$\text{👻}_{\mathsf{B+SLS}}$
$\text{👻}_{\mathit{J}+\mathsf{SLS}}$
$\text{👻}_{\mathsf{S+SLS}}$
👻_{R}
👻_{J}
👻_{S}
👻_{B}
👻_{SLS}
👻_{NS}

Fig. 4. On the left is a visualization of Theorem 10, where the list of compilers is connected with a dashed line to the strongest semantic their security is lifted to. On the right, the blue area of the leakage ordering represents where Theorem 4 is applicable for $\llbracket\cdot\rrbracket^{f}_{\mathsf{SLS}}$. We can lift the security guarantees of $\llbracket\cdot\rrbracket^{f}_{\mathsf{SLS}}$ from the base semantics 👻_{SLS} to all composed semantics in the highlighted area.

We remark that our lifting theorem allowed us to derive strong compiler guarantees, i.e., *CRSSP* w.r.t. $\text{👻}_{\mathsf{B}+\mathit{J}+\mathsf{S+R}}$ or $\text{👻}_{\mathsf{B}+\mathit{J}+\mathsf{S+SLS}}$, *without* requiring new secure compilation proofs, thereby significantly reducing the proving effort. Note that carrying out secure compilation proofs can be complex because each proof requires setting up multiple cross-language relations (for states, values, actions, etc.) [23] as well as defining an invariant that holds for speculation (both in securely-compiled code and in attacker code) [49].

We also stress that the lifted security guarantees from Theorem 10 are expressed in terms of the *CRSSP* criterion, not in terms of the stronger *RSSP* criterion. That is, the lifted security guarantees only holds for programs that are initially secure w.r.t. the extension semantics since the compiler might not prevent leaks under the extension semantics already present in the source program.

As a concrete example, we use Figure 4 to visualize (1) the effects of Theorem 10 and (2), this lifting for the fence compiler $\llbracket\cdot\rrbracket^{f}_{\mathsf{SLS}}$. With respect to (1), the list of compilers on the left is connected to the strongest semantics to which they can be lifted. With respect to (2), the blue shaded area depicts the semantics to which we can lift the security guarantees using Theorem 4 together with Independence and Safe Nesting. Our lifting approach allowed us to derive secure compilation results for the 7 semantics in the shaded area with only 1 secure compilation proof (for the base semantics 👻_{SLS}), 1 simple proof of Trapped Speculation, and other 6 simple proofs of Syntactic Independence (by syntactic inspection). Without our approach, proving the same results would have required 7 fully-fledged secure compilation proofs. We remark, that the compiler $[\![\cdot]\!]^{f}_{\text{SLS}}$ only inserts fences against 👻_{SLS} and we require the precondition of $\text{👻}_{\text{B+}J\text{+SLS}} \vdash P : \text{RSS}$ of *CRSSP* to ensure security w.r.t to the other semantics. Thus, *CRSSP* states that the security guarantees of $[\![\cdot]\!]^{f}_{\text{SLS}}$ are preserved for the combinations. A way of removing this precondition would be by composing different compiler countermeasures together, which we discuss in the following section.

6 Discussion

Scope of the Security Analysis. Lifting the results of our security analysis to real-world CPUs and compilers is only possible to the extent that our models faithfully represent the target systems.

In terms of CPUs and speculative leaks, any information flow in the target CPU that is not captured by our speculative semantics (and their combinations) might invalidate our security proofs in practice. In particular, all speculative semantics from Table 1 consider a commonly-used attacker model [3, 17, 21, 24, 29–31, 49, 56] that captures a cache-based attacker by exposing control-flow and memory accesses along non-speculative and speculative execution paths. Any leaks not reflected in control-flow and memory accesses might, therefore, be missed by our models. Regarding speculation, for the two new semantics modeling speculation over indirect jumps (👻_{J}) and straight-line speculation (👻_{SLS}), the main simplification is in the modeling of speculative indirect jumps where we over-approximate the effect of prediction structures (e.g., BTB) by allowing any speculative target. Note, however, that we only support valid instructions in the compiled program as targets of speculative jumps; speculatively jumping in the middle of program instructions is not captured by 👻_{J}. For the other semantics ($\text{👻}_{\mathbf{B}}$, 👻_{S}, and 👻_{R}), we directly employ state-of-the-art models from prior work, whose limitations are discussed in [24, 31].

In terms of compilers, any divergence between our models in Section 5.1 and their actual implementations might, again, invalidate our results. One important simplification of our SLH compilers $[\![\cdot]\!]^{USLH}_{\mathbf{B}}$ and $[\![\cdot]\!]^{SSLH}_{\mathbf{B}}$ is that the speculation flag is *always* stored in a dedicated register. In contrast, the actual SLH implementation in Clang [16] uses a general purpose register for storing the speculation flag which, in some cases, might be spilled to memory. This might result in unexpected leaks in case speculation over store-bypasses (👻_{S}) might result in loading a stale value for the speculation flag from memory.

Using the Framework. At this point, the reader may wonder how one can use this framework when the next version of Spectre comes out. For example, recent work [46] has discovered that CPUs also speculate on division operations (since this does not lead to actual attacks, we ignored this speculative semantics in our security analysis). Let us indicate a semantics capturing leaks resulting from that speculation over divisions with $\text{👻}_{\mathbf{D}}$. The ordering of Figure 2 would have many new elements, including a top element $\text{👻}_{\mathbf{B}+J+\text{S}+\text{R}+\mathbf{D}}$. Since none of the considered compilers introduce a division operation, it would be sufficient to prove Syntactic Independence for them, reuse all the theorems from Sections 5.2 and 5.4 and then apply Theorem 4 in order to obtain that those compilers are *CRSSP* for the new speculative semantics $\text{👻}_{\mathbf{B}+J+\text{S}+\text{R}+\mathbf{D}}$.

Limitations. Our framework currently suffers from three core limitations:

- *Composing semantics:* Some of the speculative semantics studied in this paper cannot be composed together, e.g., 👻_{R} and 👻_{SLS}. This is due to a restriction of the underlying combination framework [24], which does not allow combining semantics that speculate on the same instruction. This limits the applicability of our lifting theorem in some cases, e.g., we cannot lift security guarantees from 👻_{R} to a semantics containing also 👻_{SLS} (and vice versa).
- *Target security properties:* Our framework is currently limited to compilers that focus on preventing different classes of speculative leaks, which is reflected in some core aspects of our lifting theorems (e.g., its reliance on *RSSP* and *CRSSP*). We believe, however, that our framework can be extended to other classes of security properties *beyond speculative leaks*, such as memory safety (👻_{MS}) [9] and cryptographic constant time (👻_{CT}, equivalent to our 👻_{NS} semantics) [40]. In the case of 👻_{MS} and 👻_{CT}, however, existing semantics that express these leaks can be composed in a much simpler way than speculative semantics, without any nesting. Thus, proving that e.g., countermeasures for 👻_{MS} are *CRSSP* for $\text{👻}_{\mathsf{MS+CT}}$, only requires reasoning about Independence, because Safe Nesting trivially holds. We leave reasoning about these results (and composing the resulting semantics with the presented ones) as future work.
- *From Single to Multiple Compilers:* Our lifting theorem preserves *CRSSP*, which means that the lifted security guarantees only hold for programs that are initially secure for the extension semantics. To lift this restriction, one could *compose multiple compilers* where each compiler prevents leaks for a specific speculative semantics. The current framework, however, does not provide a way of securely composing compiler passes. Despite this, we believe that this framework provides a first step towards reasoning about the application of several Spectre countermeasures. In fact, we *speculate* that we can use existing results on composing secure compilers [42] to prove that if a compiler $[\![\cdot]\!]_1$ is *CRSSP* with respect to a semantics, and another compiler $[\![\cdot]\!]_2$ is *CRSSP* with respect to the same semantics, then the two can be composed ($[\![\,[\![\cdot]\!]_2\,]\!]_1$) and the result is *CRSSP* with respect to the same semantics. The presented work can then be used to (1) first lift *single* compiler countermeasures to their strongest semantics and then (2) compose those countermeasures to obtain that the composition is also *CRSSP* for the strongest semantics. This would allow for the verification of full compilers, instead of single compiler passes as we do here. We leave investigating the theory of secure compilation applied to speculative semantics, as well as its application to the results of this paper via points (1) and (2) for future work.

7 Related Work

Speculative Execution Attacks. After Spectre [39] has been disclosed to the public in 2018, researchers have identified many other speculative execution attacks [6, 10, 13, 41, 43, 57, 58]. We refer the reader to Canella et al. [14] for a survey of existing attacks.

Speculative Semantics. There are many semantics capturing the effects of speculatively executed instructions [11, 17, 20, 22, 24, 29, 31, 44, 49, 50, 56]. These semantics differ in the level of microarchitectural details that are modelled (e.g., from program-level models [31] to those closer to simplified CPU designs [29]) and the languages that are used (e.g., from models targeting While languages [49] to those targeting assembly-style languages [31]); see [18] for a survey of speculative semantics.

As indicated in Table 1, our branch speculation semantics 👻_{B} is from [31], whereas our store-bypass speculation 👻_{S} and return misprediction 👻_{R} semantics are from [24]. These semantics and our new semantics $\text{👻}_{\mathcal{J}}$ and 👻_{SLS} all follow the *always-mispredict* strategy [31], which explore mispredicted paths for a fixed number of steps before continuing the architectural execution.

Security Properties for speculative leaks. Researchers have proposed many program-level properties for security against speculative leaks, which can be classified into three main groups [18]:

(1) Non-interference definitions ensure the security of speculative *and* non-speculative instructions. E.g., speculative constant-time [17] extends constant-time to transient instructions as well.

(2) Relative non-interference definitions [19, 29, 31, 32, 54] ensure that transient instructions do not leak more information than non-transient ones. E.g., speculative non-interference [31], which we inherit from the secure compilation framework we build on [49], restricts the information leaked by speculatively executed instructions (without constraining what can be leaked non-speculatively).

(3) Definitions that formalise security as a safety property [49, 50], which may over-approximate definitions from the groups above.

Secure compilation for speculative leaks. Our secure compilation framework extends the work by Patrignani and Guarnieri [49] (which is restricted to branch speculation 👻$_{\mathbf{B}}$) with support for new speculative semantics and their combinations [24]. In particular, the *CRSSP* secure compilation criterion from Definition 9 is an extension of the *RSSP* criterion from [49].

In Section 5, we analyzed Spectre countermeasures implemented in mainstream compilers (or variants of them) or suggested by hardware vendors. Next, we review further countermeasures.

Barthe et al. [12] and Shivakumar et al. [53] extend Jasmin [3, 4] to protect constant-time programs against leaks induced by branch speculation. In contrast, Blade [56] is a countermeasure against Spectre-PHT targeting Wasm [33] which uses a flow-sensitive security-type system to minimize the amount of protect statements (either fences or SLH) needed to secure programs. Differently from our work, which targets speculative non-interference, these works target speculative constant-time.

Swivel [45] is a compiler hardening pass for Wasm that protects against multiple Spectre attacks (Spectre-PHT, Spectre-BTB, and Spectre-RSB). However, it lacks a formal model and security proof.

In concurrent work, Mosier et al. [44] proposed Serberus, a set of compiler passes that—in combination with hardware support (e.g., Intel CET-IBT and a shadow stack for return addresses)—offer protection against Spectre-PHT, Spectre-BTB, Spectre-RSB, Spectre-STL and predictive store forwarding [6]. For any whole program satisfying static constant time (a stricter variant of constant-time), Serberus ensures that its compiled counterpart is speculative constant-time. Differently from our security analysis, where we lift secure compilation guarantees from weaker semantics to stronger combined semantics, Serberus' security proof directly targets a speculative semantics incorporating all supported speculation mechanisms. Their proof already targets a semantics at the "top" of the leakage order and does not need lifting (until the discovery of new speculation mechanisms).

Hetterich et al. [34] proposes switchpoline, an alternative to retpoline, to protect ARM cores from Spectre-BTB. It transforms indirect calls into direct calls and uses a switch statement to select the correct call target. Interestingly, the authors argue about the importance of compatible countermeasures and ensure that switchpoline is fully compatible with other Spectre countermeasures, which is in line with our Independence property (Definition 7).

8 Conclusion

This paper presented a secure compilation framework for reasoning about the security against leaks introduced by different speculation mechanisms modeled as (combinations of) speculative semantics. In particular, we developed a *lifting theorem* that allows us to lift a compiler's security guarantees from a weaker base speculative semantics to a stronger extended speculative semantics that accounts for more speculation mechanisms. Additionally, we precisely characterized the security guarantees provided by 9 Spectre-countermeasures implemented in mainstream compilers against 23 different speculative semantics covering combinations of 5 different speculation mechanisms. Our lifting theorem was instrumental in allowing us to precisely characterize each countermeasure's guarantees against all combined semantics *without* requiring additional secure compilation proofs (beyond the proof of security against each compiler's base semantics).

Acknowledgments

This work was partially supported by the Spanish Ministry of Science and Innovation under the project TED2021-132464B-I00 PRODIGY; the Spanish Ministry of Science and Innovation under the Ramón y Cajal grant RYC2021-032614-I; the Spanish Ministry of Science and Innovation under the project PID2022-142290OB-I00 ESPADA; the Italian Ministry of Education through funding for the Rita Levi Montalcini grant (call of 2019); as well as a gift from Intel.

References

[1] Martín Abadi. 1999. Secrecy by Typing in Security Protocols. *J. ACM* (1999). https://doi.org/10.1145/324133.324266

[2] Carmine Abate, Roberto Blanco, Deepak Garg, Catalin Hritcu, Marco Patrignani, and Jérémy Thibault. 2019. Journey Beyond Full Abstraction: Exploring Robust Property Preservation for Secure Compilation. In *Proceedings of the 32nd IEEE Computer Security Foundations Symposium (CSF '19)*. IEEE. https://doi.org/10.1109/CSF.2019.00025

[3] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. 2017. Jasmin: High-Assurance and High-Speed Cryptography. In *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. ACM. https://doi.org/10.1145/3133956.3134078

[4] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, Vincent Laporte, Tiago Oliveira, and Pierre-Yves Strub. 2020. The last mile: High-assurance and high-speed cryptographic implementations. In *2020 IEEE Symposium on Security and Privacy (SP) (S&P '20)*. IEEE. https://doi.org/10.1109/SP40000.2020.00028

[5] AMD. 2020. Whitepaper Straight-line Speculation. https://www.amd.com/system/files/documents/technical-guidance-for-mitigating-branch-type-confusion.pdf.

[6] AMD. 2021. Security analysis of AMD predictive store forwarding. https://www.amd.com/system/files/documents/security-analysis-predictive-store-forwarding.pdf. Accessed: 2024-03-11.

[7] ARM. 2020. Whitepaper Straight-line Speculation. https://developer.arm.com/documentation/102825/0100/.

[8] ARM. 2021. Arm Armv9-A A64 Instruction Set Architecture. https://developer.arm.com/documentation/100076/0100/A64-Instruction-Set-Reference/A64-General-Instructions/BTI?

[9] Arthur Azevedo de Amorim, Cătălin Hriţcu, and Benjamin C. Pierce. 2018. The Meaning of Memory Safety. In *Principles of Security and Trust (POST '18)*, Lujo Bauer and Ralf Küsters (Eds.). Springer. https://doi.org/10.1007/978-3-319-89722-6_4

[10] Enrico Barberis, Pietro Frigo, Marius Muench, Herbert Bos, and Cristiano Giuffrida. 2022. Branch History Injection: On the Effectiveness of Hardware Mitigations Against Cross-Privilege Spectre-v2 Attacks. In *Proceedings of the 31st USENIX Security Symposium (USENIX Security '22)*. USENIX Association. https://www.usenix.org/conference/usenixsecurity22/presentation/barberis

[11] Gilles Barthe, Marcel Böhme, Sunjay Cauligi, Chitchanok Chuengsatiansup, Daniel Genkin, Marco Guarnieri, David Mateos Romero, Peter Schwabe, David Wu, and Yuval Yarom. 2024. Testing side-channel security of cryptographic implementations against future microarchitectures. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24)*. ACM.

[12] Gilles Barthe, Sunjay Cauligi, Benjamin Grégoire, Adrien Koutsos, Kevin Liao, Tiago Oliveira, Swarn Priya, Tamara Rezk, and Peter Schwabe. 2021. High-Assurance Cryptography in the Spectre Era. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (S&P '21)*. IEEE. https://doi.org/10.1109/SP40001.2021.00046

[13] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. 2019. SMoTherSpectre: Exploiting Speculative Execution through Port Contention. In *Proceedings of the 26th ACM SIGSAC Conference on Computer and Communications Security (CCS '19)*. ACM. https://doi.org/10.1145/3319535.3363194

[14] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *Proceedings of the 28th USENIX Security Symposium (USENIX Security '19)*. USENIX Association. https://www.usenix.org/conference/usenixsecurity19/presentation/canella

[15] Claudio Canella, Sai Manoj Pudukotai Dinakarrao, Daniel Gruss, and Khaled N. Khasawneh. 2020. Evolution of Defenses against Transient-Execution Attacks. In *Proceedings of the 2020 on Great Lakes Symposium on VLSI (GLSVLSI '20)*. ACM. https://doi.org/10.1145/3386263.3407584

[16] Chandler Carruth. 2018. Speculative Load Hardening. https://llvm.org/docs/SpeculativeLoadHardening.html

[17] Sunjay Cauligi, Craig Disselkoen, Klaus v. Gleissenthall, Dean Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. 2020. Constant-Time Foundations for the New Spectre Era. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '20)*. ACM. https://doi.org/10.1145/3385412.3385970

[18] Sunjay Cauligi, Craig Disselkoen, Daniel Moghimi, Gilles Barthe, and Deian Stefan. 2022. SoK: Practical Foundations for Software Spectre Defenses. In *Proceedings of the 43rd IEEE Symposium on Security and Privacy (S&P '22)*. IEEE. https://doi.org/10.1109/SP46214.2022.9833707

[19] Kevin Cheang, Cameron Rasmussen, Sanjit Seshia, and Pramod Subramanyan. 2019. A Formal Approach to Secure Speculation. In *Proceedings of the 32nd IEEE Computer Security Foundations Symposium (CSF '19)*. IEEE. https://doi.org/10.1109/CSF.2019.00027

[20] Robert J. Colvin and Kirsten Winter. 2019. An Abstract Semantics of Speculative Execution for Reasoning About Security Vulnerabilities. In *Proceedings of the 19th Refinement Workshop (Refine '19)*. Springer. https://doi.org/10.1007/978-3-030-54997-8_21

[21] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. 2021. Hunting the Haunter — Efficient relational symbolic execution for Spectre with Haunted RelSE. In *Proceedings of the 28th Annual Network and Distributed System Security Symposium (NDSS '21)*. The Internet Society.

[22] Craig Disselkoen, Radha Jagadeesan, Alan Jeffrey, and James Riely. 2019. The Code That Never Ran: Modeling Attacks on Speculative Evaluation. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P '19)*. IEEE. https://doi.org/10.1109/SP.2019.00047

[23] Akram El-Korashy, Roberto Blanco, Jeremy Thibault, Adrien Durier, Deepak Garg, and Catalin Hritcu. 2022. SecurePtrs: Proving Secure Compilation with Data-Flow Back-Translation and Turn-Taking Simulation. In *Proceedings of the 35th IEEE Computer Security Foundations Symposium (CSF '22)*. IEEE. https://doi.org/10.1109/CSF54842.2022.9919680

[24] Xaver Fabian, Marco Guarnieri, and Marco Patrignani. 2022. Automatic Detection of Speculative Execution Combinations. In *Proceedings of the 29th ACM SIGSAC Conference on Computer and Communications Security (CCS '22)*. ACM. https://doi.org/10.1145/3548606.3560555

[25] Xaver Fabian, Marco Guarnieri, Marco Patrignani, and Michael Backes. 2024. https://github.com/XFabian/secure_comp_lifting

[26] Xaver Fabian, Marco Patrignani, Marco Guarnieri, and Michael Backes. 2024. Do You Even Lift? Strengthening Compiler Security Guarantees Against Spectre Attacks. arXiv:2405.10089 [cs.PL] https://arxiv.org/abs/2405.10089

[27] Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. 2007. A Type Discipline for Authorization Policies. *ACM Trans. Program. Lang. Syst.* 29 (2007). https://doi.org/10.1145/1275497.1275500

[28] Andrew D. Gordon and Alan Jeffrey. 2003. Authenticity by Typing for Security Protocols. *J. Comput. Secur.* 11 (2003). http://dl.acm.org/citation.cfm?id=959088.959090

[29] Roberto Guanciale, Musard Balliu, and Mads Dam. 2020. Inspectre: Breaking and fixing microarchitectural vulnerabilities by formal analysis. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS '20)*. ACM, 1853–1869. https://doi.org/10.1145/3372297.3417246

[30] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. 2021. Hardware-software contracts for secure speculation. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy (S&P '21)*. IEEE. https://doi.org/10.1109/SP40001.2021.00036

[31] Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. 2020. Spectector: Principled Detection of Speculative Information Flows. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P '20)*. https://doi.org/10.1109/SP40000.2020.00011

[32] Shengjian Guo, Yueqi Chen, Peng Li, Yueqiang Cheng, Huibo Wang, Meng Wu, and Zhiqiang Zuo. 2020. SpecuSym: Speculative Symbolic Execution for Cache Timing Leak Detection. In *Proceedings of the 42nd ACM/IEEE International Conference on Software Engineering (ICSE '20)*. ACM. https://doi.org/10.1145/3377811.3380428

[33] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '17)*. ACM. https://doi.org/10.1145/3062341.3062363

[34] Lorenz Hetterich, Markus Bauer, Michael Schwarz, and Christian Rossow. 2024. Switchpoline: A Software Mitigation for Spectre-BTB and Spectre-BHB on ARMv8. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security (ASIA CCS '24)*. ACM. https://doi.org/10.1145/3634737.3637662

[35] J. Horn. 2018. Speculative execution, variant 4: Speculative store bypass. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528. Accessed: 2021-04-11.

[36] Intel. 2018. Retpoline: A Branch Target Injection Mitigation. https://www.intel.com/content/dam/develop/external/us/en/documents/retpoline-a-branch-target-injection-mitigation.pdf

[37] Intel. 2018. Speculative Store Bypass. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/speculative-store-bypass.html

[38] Intel. 2018. Using Intel Compilers to Mitigate Speculative Execution Side-Channel Issues. https://software.intel.com/en-us/articles/using-intel-compilers-to-mitigate-speculative-execution-side-channel-issues

[39] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P '19)*. https://doi.org/10.1109/SP.2019.00002

[40] Paul C. Kocher. 1996. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology (CRYPTO '96)*, Neal Koblitz (Ed.). Springer Berlin Heidelberg. https://doi.org/10.1007/3-540-68697-5_9

[41] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2018. Spectre Returns! Speculation Attacks Using the Return Stack Buffer. In *Proceedings of the 12th USENIX Workshop on Offensive Technologies (WOOT '18)*. USENIX Association. https://www.usenix.org/conference/woot18/presentation/koruyeh

[42] Matthis Kruse and Marco Patrignani. 2022. Composing Secure Compilers.

[43] Giorgi Maisuradze and Christian Rossow. 2018. Ret2spec: Speculative Execution Using Return Stack Buffers. In *Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. ACM. https://doi.org/10.1145/3243734.3243761

[44] N. Mosier, H. Nemati, J. C. Mitchell, and C. Trippel. 2024. Serberus: Protecting Cryptographic Code from Spectres at Compile-Time. In *Proceedings of the 45th IEEE Symposium on Security and Privacy (S&P '24)*. IEEE. https://doi.org/10.1109/SP54263.2024.00048

[45] Shravan Narayan, Craig Disselkoen, Daniel Moghimi, Sunjay Cauligi, Evan Johnson, Zhao Gang, Anjo Vahldiek-Oberwagner, Ravi Sahita, Hovav Shacham, Dean Tullsen, and Deian Stefan. 2021. Swivel: Hardening WebAssembly against Spectre. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security '21)*. USENIX Association. https://www.usenix.org/conference/usenixsecurity21/presentation/narayan

[46] Oleksii Oleksenko, Marco Guarnieri, Boris Köpf, and Mark Silberstein. 2023. Hide and Seek with Spectres: Efficient discovery of speculative information leaks with random testing. In *Proceedings of the 44th IEEE Symposium on Security and Privacy (S&P 2023)*. IEEE. https://doi.org/10.1109/SP46215.2023.10179391

[47] Marco Patrignani. 2020. Why should anyone use colours? or, syntax highlighting beyond code snippets. *arXiv preprint arXiv:2001.11334* (2020).

[48] Marco Patrignani and Sam Blackshear. 2023. Robust Safety for Move. In *Proceedings of the 36th IEEE Computer Security Foundations Symposium (CSF '23)*. IEEE. https://doi.org/10.1109/CSF57540.2023.00045

[49] Marco Patrignani and Marco Guarnieri. 2021. Exorcising Spectres with Secure Compilers. In *Proceedings of the 28th ACM Conference on Computer and Communications Security (CCS '21)*. ACM. https://doi.org/10.1145/3460120.3484534

[50] Hernán Ponce de León and Johannes Kinder. 2022. Cats vs. Spectre: An Axiomatic Approach to Modeling Speculative Execution Attacks. In *Proceedings of the 43rd IEEE Symposium on Security and Privacy (S&P '22)*. IEEE. https://doi.org/10.1109/SP46214.2022.9833774

[51] Michael Sammler, Deepak Garg, Derek Dreyer, and Tadeusz Litak. 2020. The high-level benefits of low-level sandboxing. *Proc. ACM Program. Lang.* (2020). https://doi.org/10.1145/3371100

[52] Vedvyas Shanbhogue, Deepak Gupta, and Ravi Sahita. 2019. Security Analysis of Processor Instruction Set Architecture for Enforcing Control-Flow Integrity. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy (HASP '19)*. ACM. https://doi.org/10.1145/3337167.3337175

[53] B. Shivakumar, G. Barthe, B. Gregoire, V. Laporte, T. Oliveira, S. Priya, P. Schwabe, and L. Tabary-Maujean. 2023. Typing High-Speed Cryptography against Spectre v1. In *Proceedings of the 44th IEEE Symposium on Security and Privacy (S&P '23)*. IEEE. https://doi.org/10.1109/SP46215.2023.10179418

[54] Basavesh Ammanaghatta Shivakumar, Jack Barnes, Gilles Barthe, Sunjay Cauligi, Chitchanok Chuengsatiansup, Daniel Genkin, Sioli O'Connell, Peter Schwabe, Rui Qi Sim, and Yuval Yarom. 2023. Spectre Declassified: Reading from the Right Place at the Wrong Time. In *Proceedings of the 44th IEEE Symposium on Security and Privacy (S&P '23)*. IEEE. https://doi.org/10.1109/SP46215.2023.10179355

[55] David Swasey, Deepak Garg, and Derek Dreyer. 2017. Robust and Compositional Verification of Object Capability Patterns. In *Proceedings of the 2017 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. (OOPSLA '17)*. ACM. https://doi.org/10.1145/3133913

[56] Marco Vassena, Craig Disselkoen, Klaus von Gleissenthall, Sunjay Cauligi, Rami Gökhan Kıcı, Ranjit Jhala, Dean Tullsen, and Deian Stefan. 2021. Automatically Eliminating Speculative Leaks from Cryptographic Code with Blade. *Proc. ACM Program. Lang.* (2021). https://doi.org/10.1145/3434330

[57] Johannes Wikner, Daniël Trujillo, and Kaveh Razavi. 2023. Phantom: Exploiting Decoder-detectable Mispredictions. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23)*. ACM. https://doi.org/10.1145/3613424.3614275

[58] Tao Zhang, Kenneth Koltermann, and Dmitry Evtyushkin. 2020. Exploring Branch Predictors for Constructing Transient Execution Trojans. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. ACM. https://doi.org/10.1145/3373376.3378526

[59] Zhiyuan Zhang, Gilles Barthe, Chitchanok Chuengsatiansup, Peter Schwabe, and Yuval Yarom. 2023. Ultimate SLH: taking speculative load hardening to the next level. In *Proceedings of the 32nd USENIX Conference on Security Symposium (USENIX Security '23)*. USENIX Association. https://www.usenix.org/conference/usenixsecurity23/presentation/zhang-zhiyuan-slh

Received 2024-07-11; accepted 2024-11-07