SNIP: Speculative Execution and Non-Interference Preservation for Compiler Transformations

SÖREN VAN DER WALL, TU Braunschweig, Germany
ROLAND MEYER, TU Braunschweig, Germany

We address the problem of preserving non-interference across compiler transformations *under speculative semantics*. We develop a proof method that ensures the preservation uniformly across all source programs. The basis of our proof method is a new form of simulation relation. It operates over directives that model the attacker's control over the micro-architectural state, and it accounts for the fact that the compiler transformation may change the influence of the micro-architectural state on the execution (and hence the directives). Using our proof method, we show the correctness of dead code elimination. When we tried to prove register allocation correct, we identified a previously unknown weakness that introduces violations to non-interference. We have confirmed the weakness for a mainstream compiler on code from the `libsodium` cryptographic library. To reclaim security once more, we develop a novel static analysis that operates on a product of source program and register-allocated program. Using the analysis, we present an automated fix to existing register allocation implementations. We prove the correctness of the fixed register allocations with our proof method.

CCS Concepts: • **Security and privacy → Formal security models**; **Side-channel analysis and countermeasures**; • **Software and its engineering → Compilers**.

Additional Key Words and Phrases: speculative execution, compilation, verification, register allocation

ACM Reference Format:
Sören van der Wall and Roland Meyer. 2025. SNIP: Speculative Execution and Non-Interference Preservation for Compiler Transformations. *Proc. ACM Program. Lang.* 9, POPL, Article 51 (January 2025), 39 pages. https://doi.org/10.1145/3704887

1 Introduction

Cryptographic implementations must satisfy two conflicting requirements: They must compute highly performant to be of practical use and be absolutely secure for use in critical systems. Implementations optimize performance with knowledge about underlying micro-architectural hardware features such as memory access patterns that improve cache usage. Security, however, is threatened by side-channel attacks that exploit precisely these hardware features to leak sensitive information [Brumley and Boneh 2005]. To mitigate side-channel attacks, leakage of sensitive data needs to be eliminated. This confronts the programmer with two challenges: First, semantics of source-level languages do not model leaks produced by side-channels [Vu et al. 2021]. And second, even if the source-level code is secure, incautious implementation of compiler optimizations can insert new leakage, rendering efforts to secure the source program useless [Simon et al. 2018; Barthe, Blazy, Grégoire, et al. 2019]. Developers address these problems with coding guidelines such as *constant time programming* and disabling compiler optimizations. But following the guidelines is non-trivial and overlooked mistakes corrupt the guarantee for security [Al Fardan and Paterson

Authors' Contact Information: Sören van der Wall, TU Braunschweig, Braunschweig, Germany, s.van-der-wall@tu-bs.de; Roland Meyer, TU Braunschweig, Braunschweig, Germany, roland.meyer@tu-bs.de.

ACM 2475-1421/2025/1-ART51
https://doi.org/10.1145/3704887

```
1 chacha20(..., uint8* sec, uint8 bytes) {
2   uint8 buf[8];
3   stk[0] = bytes;
4   for (i = 0; i < 8; i++)
5     { buf[i] = sec[i]; }
6   ...
7   bytes = stk[0];
8   if (bytes < 64) {...} ... }
```

```
Code  Stack (-SP)
       0x48  stk
       0x40  buf
Heap
       0xE0
       ....          sensitive
       0xC8  sec     data
       0xC0
```

Code 1. Spectre-PHT. Registers are orange and memory variables teal. The stack contents are shifted by the stack pointer to appear constant. Framed instructions were inserted by register allocation.

2013]. At the same time, disabling compiler optimizations is dissatisfactory. Formal methods have shown to help with the challenges: The first challenge is overcome with novel leakage semantics that model side-channel leakage within the programming language's semantics [Molnar et al. 2006; Barthe, Betarte, et al. 2014]. For the second, novel proof methods for compilers under leakage semantics provide a guarantee that side-channel security of the source program carries over to the executable [Barthe, Grégoire, and Laporte 2018; Barthe, Blazy, Grégoire, et al. 2019; Barthe, Blazy, Hutin, et al. 2021; Barthe, Grégoire, Laporte, and Priya 2021]. Sadly, the recent discovery of Spectre attacks [Canella et al. 2019; Kocher et al. 2019] again presents a hardware feature that leakage semantics fall short of: *Speculative execution* produces side-channel leakages not captured by leakage semantics. This means both challenges were open again, and the verification community was quick to address the first: The development of speculative execution semantics has already taken place [Cauligi, Disselkoen, Gleissenthall, et al. 2020; Guarnieri, Köpf, Reineke, et al. 2021], and formal tools find speculative side-channel leakages or even prove their absence (cf. Section 7). Provably correct compilation under speculative execution semantics, however, remains an unsolved challenge that we address in this paper. It is the challenge that we address in this paper.

1.1 Background

Before we detail our contributions, we position our paper in the field of *formally verified cryptography* and provide background on *Spectre attacks* and *mitigations*. We outline *speculative execution semantics*, *non-interference* as the property that guarantees a program's side-channel security even under speculation, and *provably correct compilation* for leakage semantics without speculation.

Formally verified cryptography. The field of formally verified cryptography aims to provide cryptographic implementations that are secured not only by trust in the developer but in a machine-checkable proof of correctness and security. In order to achieve this goal, implementations are carefully crafted and three main areas of research pursue different subgoals [Barbosa et al. 2021]: **(i)** *Cryptographic protocol design* aims to provide proofs that the cryptographic protocol in itself does not reveal secrets to adversarial protocol participants, among other properties. **(ii)** *Correct and performant implementation* of the protocol aims to prove the implementation functionally correct. **(iii)** *Implementation security* investigates the compilation of implementations and the execution of binaries on real hardware in order to prove the absence of attacks that stem from the discrepancy between idealized program semantics and actual hardware semantics. This paper belongs to **(iii)**: We assume that source programs correctly implement formally verified protocols, i.e. steps **(i)** and **(ii)** are completed. We investigate whether compiler transformations preserve side-channel security. To that end, our formal semantics models side-channel leakages and speculative execution, the micro-architectural hardware components that enable the recently discovered Spectre attacks.

Spectre attacks and mitigations. Spectre attacks observe side-channel leakages that are produced during speculative execution. Speculative execution allows the processor to speculatively execute instructions from the pipeline even though they still have unevaluated instruction parameters. When the processor detects a misspeculation, i.e. it assumed incorrect values for the unevaluated parameters, it rolls back execution to erase its effect. Rollbacks are invisible to typical source-level semantics, but parts of the micro-architecture such as the cache-state are not reverted. This creates side-channel leakage observable to an attacker during the speculative execution of instructions. Spectre attacks target this in the following way: **(i)** Train some micro-architectural component to speculatively execute a code fragment that **(ii)** under misspeculation brings sensitive data into a processor's register, which **(iii)** is leaked through side-channels. The prominent example is Spectre-PHT [Kocher et al. 2019], whose source of speculation is the processor's *branch prediction* unit (Prediction History Table), and the side-channel leakage happens via the cache [Yarom and Falkner 2014; Liu et al. 2015] or the program counter [Molnar et al. 2006]. Code 1 demonstrates the attack:[1] The code is intended to load an 8-byte chunk from `sec`, which points into a stream of sensitive data, and to store it into a stack-local buffer `buf` in order to later perform computation on it. **(i)** The attacker might train the branch predictor in a way that it speculates the `for`-conditional in Line 4 and executes Line 5 an additional time even though `i = 8`. Line 5 then stores sensitive data from `&sec + 8`, say v, to `&buf + 8` which aliases with `&buf + 8 = &stk`. The speculative execution might continue with Line 7, where **(ii)** v is loaded into `bytes`. **(iii)** The register is then used in Line 8, where the branching condition is leaked, disclosing to the attacker whether $v < 64$.

The de facto approach to avoid Spectre attacks are hardware and software *mitigations*. In hardware, a simple mitigation is to disable specific speculation sources using control registers. This penalizes performance as it disables the optimization for the whole program, even when other parts of the program do not operate on sensitive data. Software mitigations have received more attention, especially for branch prediction (Spectre-PHT, cf. Section 7, *Tools*), because disabling branch prediction has severe impact on performance [Vassena et al. 2021, Evaluation]. Spectre-PHT has two known software mitigations: Speculation fence insertion and speculative load hardening [Zhang et al. 2023; Carruth 2024]. Speculation fences `sfence` instruct the processor to stop speculation and wait until all instruction's unevaluated parameters are resolved before either continuing computation in case of a correct prediction or rolling back in case of a misprediction. This prevents instructions following `sfence` to be executed speculatively altogether.[2] The mitigation is applicable to all known kinds of speculation sources. Speculative load hardening `slh` a is a mitigation unique to branch prediction. Executing `slh` a wipes the contents of register a in case of a branch misprediction, but does not stop speculative execution. In case of correct prediction or non-speculative execution, it leaves the register contents unchanged. In the binary, this semantics is achieved by tying the contents of a to a previous branching condition cond via a data dependency (in the sense of $a := \text{cond} \ ? \ a : 0$).[3] This forces the processor to evaluate cond before assigning a value to a. The processor is not guaranteed to stop speculation immediately upon learning the correct value for cond, but the value in a is now safe to be leaked. For other speculation sources (Section 7, *Speculation Sources*), speculative load hardening does not work because no similar data dependency is known. The Spectre attack from Code 1 is mitigated by inserting either `sfence` or `slh bytes` between Line 7 and 8.

[1]The attack on this code is unlikely to execute on actual hardware because the specific speculation patterns would be hard to train. We chose it because it also demonstrates a new vulnerability in register allocation that we present in this paper.
[2]This is idealized: In `x86`, for example, the instruction is realized with a memory fence `LFENCE`, which only executes after all loaded parameters to instructions are resolved - stopping the so far known speculation sources.
[3]In `x86`, a `cmov` instruction is used which does not introduce control-flow branching, so branch prediction will not speculate.

Speculative execution semantics and Non-interference. Speculative execution semantics extend leakage semantics by speculation. The achievement of leakage semantics is to incorporate a model leakage observable to the attacker in the semantics. The observable side-channel leakage depends on the leakage model. Common is the constant-time model which exposes the addresses of memory accesses and the program counter as leakage to the attacker [Barthe, Betarte, et al. 2014; Guarnieri, Köpf, Reineke, et al. 2021]. Transitions in leakage semantics (without speculation) are of the form $s \xrightarrow{\lambda} t$. They prompt a transition from s to t while capturing attacker-visible effects on the micro-architectural state in the *leakage observation* λ. Side-channel security can now be formulated as a property on the program's leakage semantics. For that, a relation declares initial states as indistinguishable to the attacker when they differ only w.r.t. sensitive data unknown to the attacker. The property is non-interference, which requires that the executions from indistinguishable initial states produce equal leakages. Non-interference guarantees side-channel security: leakages cannot depend on sensitive data in any way. Without speculation, the running example Code 1 satisfies non-interference under the constant-time leakage model: The control flow is not dependent on the secret `sec` and the addresses of memory accesses (Lines 3, 5, and 7) are independent as well.

The extension to speculative execution semantics came with a new challenge: Non-determinism. Whether the processor mispredicts and when it detects misprediction is highly hardware dependent and potentially even under the attacker's influence. As a result, there is not a single execution but instead a set of possible executions, each with a different sequence of leakages. A transition in speculative execution semantics is of the form $s \xrightarrow{\delta:\lambda} t$. Again, λ is the attacker-observable leakage. What is new is the *directive* δ that models the attacker's control over speculation [Cauligi, Disselkoen, Gleissenthall, et al. 2020; Barthe, Cauligi, et al. 2021]. The directives determine the program's speculation behavior. They provide an abstraction of the micro-architecture that the attacker can use to steer the execution whenever it depends on the micro-architectural state. In our example, the attacker steers speculation with the following sequence of directives in order to lead execution to the leakage of sensitive data:

$$\underset{\text{Line 3}}{\bullet} \;.\; \underset{\text{Lines 4 and 5}}{(\textsc{br}\,.\,\bullet)^8} \;.\; \underset{\text{Line 4}}{\textsc{sp}} \;.\; \underset{\text{Line 5}}{\textsc{su}\,\texttt{stk}\,0} \;.\; \underset{\text{Lines 4, 7, and 8}}{\textsc{br}\,.\,\bullet\,.\,\bullet} \;.$$

The intuition is the following. The first instruction is a memory access which cannot be influenced by the attacker, denoted by the directive $\bullet$. Then, the attacker steers execution so that the correct branch is taken 8 times: Directive BR executes the correct branch and $\bullet$ executes the memory assignment inside the loop. The attacker then chooses to begin a misspeculation with directive SP which enters the loop once more. This leads to an unsafe memory access during the additional loop iteration, where we let the attacker choose actual memory location with SU `stk` 0. The remaining sequence leads the execution to the leaking instruction.

In order to phrase non-interference on speculative execution semantics, the idea is to compare executions where the sequence of directives along the executions are equal. Then, the attacker trained the hardware in the same way and can be sure that observed differences in leakage are due to sensitive data. We define our speculative execution semantics (Section 2) and non-interference property (Section 3) in this spirit.

Secure compilation. The goal of secure compilation is to prove that a compiler pass preserves non-interference from source to target program. The current methods for leakage semantics without speculation draw from classical methods for compiler correctness which utilize *simulation* in order to argue that the target program's executions can be found in the source program's semantics [McCarthy and Painter 1967; Leroy 2009]. A traditional simulation is a relation $\prec$ between the target program's states and the source program's states. Whenever a target state t is simulated by a source state s, $t \prec s$, and has a transition $t \xrightarrow{o} v$, where o is an observable environment

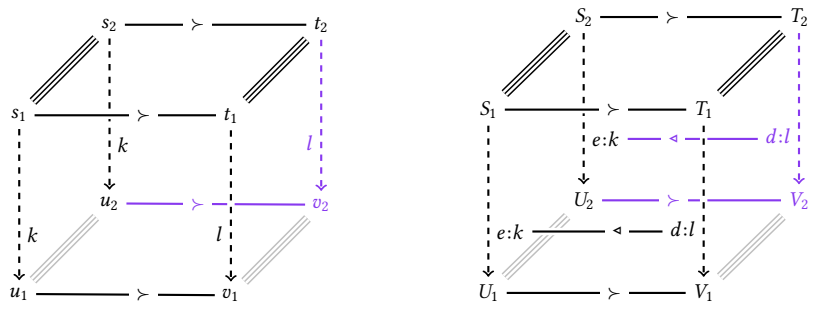

Fig. 1. Constant-time cubes. Left: leakage transforming; Right: directive transforming ($\triangleleft$).

interaction, then s has to have a next transition $s \xrightarrow{o} u$ so that $v \prec u$. Simulation ensures that the target program's execution produces the same observable environment interactions as the source program. For leakage semantics, a notion of simulation needs more: Compilers aim to preserve observable environment interactions such as system-calls, but they regularly modify side-channel leakage which creates a difference in leakage between source and target program. *Leakage transformation* [Costanzo et al. 2016; Barthe, Grégoire, and Laporte 2018; Barthe, Grégoire, Laporte, and Priya 2021] solves this issue: Given the leakages along a source program's execution, the simulation also provides a way to transform the observable leakage into the observable leakage of the corresponding target program's execution. In order to preserve non-interference, a simulation with leakage transformation needs to satisfy the *constant-time cube diagram*. The cube diagram can be seen in Figure 1 on the left. It looks at two pairs of states related by simulation $t_1 \prec s_1$ and $t_2 \prec s_2$. Then, if s_1's next transitions leak a sequence k and so do s_2's next transitions (black) then the next transitions' leakage from t_1 and t_2 must coincide as well (purple). The target leakage l does *not* need to be equal to the source leakage k. Table 1 (left) lists compilers employing simulations with leakage transformation (no speculative semantics) that satisfy the constant-time cube diagram.

In the speculative execution setting, compilers so far try to avoid Spectre attacks by running compiler passes that insert the mitigations discussed above. These passes are among the last passes in the compiler chain in order to avoid the removal of mitigations by other passes. They aim to eliminate speculative leakage by inserting mitigations conservatively, which entails significant performance overhead. Even worse: Efforts to improve performance were flawed, again leading to insecure executables [Patrignani and Guarnieri 2021]. Passes that insert mitigations are desirable because they free the developer from having to think about speculation: Before the compiler runs the mitigation pass, the semantics can be considered speculation-free. However, recent research suggests that in order to obtain minimal performance overhead, the developer needs an interface to control inserted mitigations [Shivakumar et al. 2023]. This means a new proof method for compilation is needed that works when both source and target program operate under speculative execution semantics.

To the best of our knowledge, Patrignani and Guarnieri [2021] is the only work so far to *(dis-)prove* compiler correctness under speculative execution semantics. The authors target specifically compiler passes that insert mitigations and discovered the aforementioned flaws in fence insertion and speculative load hardening. Being tailored towards mitigations, they employ assumptions on the setting that do not hold in general and that we overcome in our development. We detail the differences to our work in Table 1 (right): **Property:** The first difference lies in the property ensuring side-channel security. While speculative non-interference preservation (SNiP) is the goal,

Table 1. Left: Compilers that preserve side-channel security under leakage semantics; Compcert [Leroy 2009], Jasmin [Barthe, Grégoire, Laporte, and Priya 2021]. Right: Proof methods for compilers with speculative execution semantics; Ex. Spectres [Patrignani and Guarnieri 2021]. Green parameters are more expressive.

	Compcert	Jasmin	Ex. Spectres	This Paper
Property	NiP	NiP	STsP	SNiP
Simulation	LT-Sim	LT-Sim	LO-Sim	DT-Sim
Speculation	—	—	SW, M, TO	US, ST
Non-Det	No	No	No	Yes
Spec Source	—	—	PHT	PHT
Memory Safety	S	S	U	S
Passes	Full	Full	SLH & Fence Ins	DC & RA

their proof method preserves speculative taint safety (STsP). Taint safety is a safety property that soundly approximates non-interference. While STsP is not an approximation of SNiP, the method is appropriate for analyzing mitigation passes. Our method is designed to prove SNiP, instead. **Speculation and Non-Det:** A bigger difference is the speculative execution semantics: They assume a speculation window (SW) that limits the number of steps speculatively executed after a misprediction before a rollback occurs. Speculation windows are a restriction of the speculative execution semantics. While the restriction is reasonably chosen with respect to current hardware, it presents an under-approximation of the speculative execution semantics. They further assume that the semantics always mispredicts branches (M) to maximize speculative execution. Mispredict semantics are no further restriction of the semantics, as maximizing speculative execution also maximizes the side-channel leakages produced. Together, these assumptions form a *deterministic* restriction of the full speculative execution semantics. The focus on compiler mitigations also led them to the assumption that the *source language is speculation-free*, meaning the speculative semantics are target program only (TO). In this paper, we deal with full, unbounded speculative execution semantics (US) and the induced non-determinism in both source and target semantics (ST). **Memory:** Our work is presented for structured memory (S) and the assumption that source programs are memory safe when executed under speculation-free semantics. Memory safety is a common assumption for compilers, as unsafe memory accesses are usually considered undefined behavior in source semantics. Our proof method also works with unstructured memory (U), but we also present a static analysis whose presentation immensely benefits from structured memory. This led us to present all of our work with structured memory as the concepts behind the proof method stay the same. **Simulation:** Mitigations insert speculation barriers and do not change the code otherwise. As the source program in Patrignani and Guarnieri [2021] does not speculate, the leakages of the source program will still be fully present and unchanged in the target program. Because the target program is executed with speculative execution semantics, there can, however, be additional leakages present in the target program. This leads their work to consider leave-out simulations (LO-Sim), where source leakages are equal to target leakages with additional speculative leakages. For general compiler transformations and unbounded speculation, we introduce the more general directive transforming simulations (DT-Sim). **Passes:** Their work targets the compiler passes that insert the software mitigations against spectre from above: Speculative load hardening and fence insertion. Our work targets two general-purpose compiler transformations: Dead code elimination (DC) and transformations from the register allocation phase (RA).

1.2 Contributions

In this paper, we present *snippy simulations*, a novel proof method for preservation of non-interference under speculative execution semantics. The main challenge to overcome with speculation is the non-determinism in directive semantics. First, the definition of simulations becomes more involved: Deterministic semantics have the advantage that simulations are always bi-simulations [Milner 1971]. A simulation for deterministic semantics synchronizes the (singular) execution of the source program and the execution of the target program. For non-deterministic semantics such as speculative execution semantics, each of the target program's executions must be synchronized with a source program's execution. Second, similar to how compilers do not preserve leakage, they also modify where the attacker can steer the computation: A sequence of directives to steer execution on the source program may be unfit to steer any execution on the target program (the compiler may change instructions and with them the available directives to steer execution also change). We address this issue by introducing the new concept of *directive transformations*. Directive transformations match every executable sequence of directives in the target program with a sequence of directives executable in the source program. We then embed directive transformations into a new constant-time cube for speculative execution semantics (Figure 1, right). It is our contribution to make the constant-time cube applicable for speculative semantics.

We demonstrate our proof method on two compiler transformations: Dead code elimination and register allocation. This is the first time that these compiler passes have been formally analyzed under speculative execution semantics and to our surprise, we found a serious vulnerability in the transformations performed during the register allocation phase. The register allocation phase is located in the compiler chain where a hardware-independent intermediate representation is replaced by a concrete ISA. It transforms virtual registers into hardware registers and has to spill excess registers to the stack: In Code 1, the framed instructions constitute a spill of the register `bytes`. The program before register allocation (without Lines 3 and 7) has no side-channel leakage of sensitive data under speculative execution semantics. The program after register allocation (with Lines 3 and 7) is vulnerable to the Spectre attack presented above. This vulnerability is not unique to a singular register allocator, but more generally stems from the spilling transformation performed in this phase. In order to fix the transformations performed, we present a novel static analysis on a product of source program (before register allocation) and target program (after register allocation) that finds problematic speculative leakages introduced by spilling transformations. We then fix the problematic transformations by inserting as few mitigations as possible. The fix is automated and applies to every existing register allocator. We then show that the fixed transformations are secure by once more applying our proof method.

In short, we address the problem of non-interference preservation for compiler passes under speculative execution semantics. We make the following contributions:

- We develop a proof method for non-interference preservation *under speculative execution semantics* based on simulation relations. Technically, we address non-determinism from speculation with *directive transformations*.
- We *demonstrate* our proof method on dead code elimination.
- We show that *register allocation does not preserve non-interference* under speculative execution semantics. We confirm this for all register allocators of the `LLVM` compiler on code from the widely used `libsodium` cryptographic library.
- We propose a static analysis that *finds and automatically fixes* the vulnerabilities introduced by any register allocator. We apply our proof method to show that the fixed transformation preserves speculative non-interference.

Outline. Section 2 introduces our formulation of speculative execution semantics with leakages and directives. Section 3 defines speculative non-interference preservation (SNiP). Our proof method is presented in Section 4, and we apply it in Section 5 to prove that dead code elimination preserves non-interference. We then analyze the vulnerability we found in the register allocation transformations and present our fix in Section 6. We finish with related works in Section 7 and discuss future prospects in Section 8.

2 Language Model

We introduce our programming language and its speculative execution semantics. A program is a mapping $P : PC_P \to Inst$ from program counters to instructions. The initial program counter is $\mathsf{init} \in PC_P$. Instructions $i \in Inst$ are of the following form. We denote registers by $a, b, c, d \in Reg$ and memory variables by $\mathbf{a}, \mathbf{b} \in Var$. The subscripts $\mathsf{sc} \in PC_P$ are the successors of i. We may also call sc a successor of pc instead, if it is a successor of P pc.

$$i \in Inst \ ::= \ \mathtt{ret} \mid \mathtt{nop}_{\rangle \mathsf{sc}} \mid a := b \oplus c_{\rangle \mathsf{sc}} \mid a := \mathbf{a}[b]_{\rangle \mathsf{sc}} \mid \mathbf{a}[b] := c_{\rangle \mathsf{sc}} \mid \mathtt{br}\ b_{\rangle \mathsf{sc_t}, \mathsf{sc_f}} \mid \mathtt{sfence}_{\rangle \mathsf{sc}} \mid \mathtt{slh}\ a_{\rangle \mathsf{sc}}$$

The instructions are return (or exit), no-op, assignment, load, store, conditional branching, and the software mitigations for Spectre, speculation fences and speculative load hardening.

Semantics. We introduce two semantics: Speculation-free $s \xrightarrow{\delta:\lambda} t$ (Rules 2.1) and speculative $S \overset{\delta:\lambda}{\dashrightarrow} T$ (Rules 2.2). The transitions are labelled by leakage $\lambda \in Leak$ and directives $\delta \in Direct$. Our leakages stem from the constant-time leakage model which leaks the addresses of memory accesses as well as branching conditions. Directives resolve non-determinism for the speculative semantics, i.e. when speculation starts and ends or where unsafe memory accesses (out-of-bounds) actually access memory. Directives are considered under the attacker's control.

A speculation-free state is a tuple $(\mathsf{pc}, \rho, \mu) \in State$ that tracks the program counter $\mathsf{pc} \in PC$, register contents $\rho : Reg \to Val$, and memory $\mu : Mem \to Val$. The semantics is given in Rules 2.1 and is fairly standard. Memory is structured and without dynamic allocation. Each variable $\mathbf{a}$ has static size and for an offset address $\mathsf{n} \in Adr \subseteq Val$, we write $\mathsf{n} \in |\mathbf{a}|$ to indicate that n lies within $\mathbf{a}$'s size. The memory is $Mem = \{(\mathbf{a}, \mathsf{n}) \mid \mathsf{n} \in |\mathbf{a}|\}$. Following the leakage model, loads (LOAD and LOAD-UNSAFE) and stores (STORE and STORE-UNSAFE) leak the accessed address used via LD n and ST n, and branching (BRANCH) leaks its condition with BR b. The directives LU $\mathbf{b}$ m (LOAD-UNSAFE) and SU $\mathbf{b}$ m (STORE-UNSAFE) let the attacker control the address for unsafe memory accesses.

The source of speculation are $\mathtt{br}\ b$ instructions triggering branch-prediction (PHT). Our semantics models speculation only for misspeculated branches. The reason for this is that a correctly predicted branch will later commit and the resulting architectural state and the observable leakages will coincide with an execution that did not speculate in the first place. With no difference in correct speculation and speculation-free execution there is no need to model correctly speculated branches separately, and we will use the terms speculation and misspeculation interchangeably. Speculation thus starts with a branch misprediction, and later ends with a rollback to the state before speculation.[4] A speculating state tracks all active mispredictions in a stack of states $S, T, \ldots \in SState = State^*$. The semantics $S \overset{\delta:\lambda}{\dashrightarrow} T$ is provided in Rules 2.2. With $P\,s$, we access s's instruction P pc, when $s = (\mathsf{pc}, \rho, \mu)$. The mitigation instructions $\mathtt{sfence}$ and $\mathtt{slh}\ a$ are *speculation sensitive*, as their semantics depend on whether the current state is speculating. Their semantics is according to our explanation in Section 1: A speculation fence $\mathtt{sfence}$ disallows speculation, so SFENCE only executes in states currently not speculating. SLH performs speculative load hardening $\mathtt{slh}\ a$, which wipes a register only if the state is currently speculating. The remaining instructions

[4]This means our semantics allows for another speculation immediately after rollback. This could be avoided with an additional flag to store whether a state has already been mispredicted.

Rules 2.1: Speculation-free Semantics

NOP
$$\frac{P\,\mathsf{pc} = \mathsf{nop}_{\rangle \mathsf{sc}}}{(\mathsf{pc}, \rho, \mu) \xrightarrow{\bullet:\bullet} (\mathsf{sc}, \rho, \mu)}$$

ASGN
$$\frac{P\,\mathsf{pc} = a := b \oplus c_{\rangle \mathsf{sc}} \quad \mathsf{v} = \rho\, b \oplus \rho\, c}{(\mathsf{pc}, \rho, \mu) \xrightarrow{\bullet:\bullet} (\mathsf{sc}, \rho[a \mapsto \mathsf{v}], \mu)}$$

BRANCH
$$\frac{P\,\mathsf{pc} = \mathsf{br}\, b_{\rangle \mathsf{sc_t}, \mathsf{sc_f}} \quad \mathsf{b} = (0 = \rho\, b)}{(\mathsf{pc}, \rho, \mu) \xrightarrow{\mathrm{BR:BR\,b}} (\mathsf{sc_b}, \rho, \mu)}$$

LOAD
$$\frac{P\,\mathsf{pc} = a := \mathbf{a}[b]_{\rangle \mathsf{sc}} \quad \mathsf{n} = \rho\, b \in |\mathbf{a}| \quad \mathsf{v} = \mu\, \mathbf{a}\, \mathsf{n}}{(\mathsf{pc}, \rho, \mu) \xrightarrow{\bullet:\mathrm{LD\,n}} (\mathsf{sc}, \rho[a \mapsto \mathsf{v}], \mu)}$$

STORE
$$\frac{P\,\mathsf{pc} = \mathbf{a}[b] := c_{\rangle \mathsf{sc}} \quad \mathsf{n} = \rho\, b \in |\mathbf{a}| \quad \mathsf{v} = \rho\, c}{(\mathsf{pc}, \rho, \mu) \xrightarrow{\bullet:\mathrm{ST\,n}} (\mathsf{sc}, \rho, \mu[(\mathbf{a}, \mathsf{n}) \mapsto \mathsf{v}])}$$

LOAD-UNSAFE
$$\frac{P\,\mathsf{pc} = a := \mathbf{a}[b]_{\rangle \mathsf{sc}} \quad \mathsf{n} = \rho\, b \notin |\mathbf{a}| \quad \mathsf{v} = \mu\, \mathbf{b}\, \mathsf{m}}{(\mathsf{pc}, \rho, \mu) \xrightarrow{\mathrm{LU\,b\,m:LD\,n}} (\mathsf{sc}, \rho[a \mapsto \mathsf{v}], \mu)}$$

STORE-UNSAFE
$$\frac{P\,\mathsf{pc} = \mathbf{a}[b] := c_{\rangle \mathsf{sc}} \quad \mathsf{n} = \rho\, b \notin |\mathbf{a}| \quad \mathsf{v} = \rho\, c}{(\mathsf{pc}, \rho, \mu) \xrightarrow{\mathrm{SU\,b\,m:ST\,n}} (\mathsf{sc}, \rho, \mu[(\mathbf{b}, \mathsf{m}) \mapsto \mathsf{v}])}$$

Rules 2.2: Speculating Semantics

STEP
$$\frac{P\,s \text{ speculation insensitive} \quad s \xrightarrow{\delta:\lambda} t}{S.s \xrightarrow{\delta:\lambda} S.t}$$

SPEC
$$\frac{P\,\mathsf{pc} = \mathsf{br}\, a_{\rangle \mathsf{sc_t}, \mathsf{sc_f}} \quad \mathsf{b} = (0 = \rho\, a)}{S.(\mathsf{pc}, \rho, \mu) \xrightarrow{\mathrm{SP:BR}\,\neg\mathsf{b}} S.(\mathsf{pc}, \rho, \mu).(\mathsf{sc}_{\neg\mathsf{b}}, \rho, \mu)}$$

ROLLBACK
$$\frac{|S| \geq 1}{S.s \xrightarrow{\mathrm{RB:RB}} S}$$

SFENCE
$$\frac{P\,\mathsf{pc} = \mathsf{sfence}_{\rangle \mathsf{sc}}}{(\mathsf{pc}, \rho, \mu) \xrightarrow{\bullet:\bullet} (\mathsf{sc}, \rho, \mu)}$$

SLH
$$\frac{P\,\mathsf{pc} = \mathsf{slh}\, a_{\rangle \mathsf{sc}} \quad \mathsf{v} = |S| > 0\ ?\ 0 : \rho\, a}{S.(\mathsf{pc}, \rho, \mu) \xrightarrow{\bullet:\bullet} S.(\mathsf{sc}, \rho[a \mapsto \mathsf{v}], \mu)}$$

are *speculation insensitive*. STEP executes them on the currently speculating state, i.e. the top-most state in the stack of states. The directives that determine whether a misprediction happens or not are SP and BR. SP demands misprediction performed by SPEC. A copy of the current state is pushed on top of the current state and the program counter is set to the incorrect branch. Otherwise, BRANCH executes on directive BR for a correct branching. ROLLBACK rolls back execution to before the last misprediction. It can be triggered with a RB directive in any state that is currently speculating. There is no bound on the length of a speculation.

We write $S \xrightarrow{d:l}{}^{*} T$ for finite executions and $S \xrightarrow{d:l} \infty$ for diverging executions. We use l and d for both finite and infinite sequences, i.e. $l \in \mathit{Leak}^{*} \cup \mathit{Leak}^{\infty}$ and $d \in \mathit{Direct}^{*} \cup \mathit{Direct}^{\infty}$. We call any $(\mathsf{init}, \rho, \mu)$ initial and (pc, ρ, μ) with $P\,\mathsf{pc} = \mathtt{ret}$ final. The behavior of a program consists of the directives and events along any execution from an initial state. The speculation-free semantics is deterministic, so its behavior is a single execution; the speculative behavior is non-deterministic and its behavior forms a set of executions.

$$\mathsf{Beh}\, P\, s \triangleq \begin{cases} (d{:}l) & s \xrightarrow{d:l}{}^{*} t, t \text{ final} \\ (d{:}l) & s \xrightarrow{d:l} \infty \end{cases} \qquad \mathsf{SBeh}\, P\, S \triangleq \begin{array}{r} \{(d{:}l) \mid S \xrightarrow{d:l}{}^{*} T, T \text{ final}\} \\ \cup\, \{(d{:}l) \mid S \xrightarrow{d:l} \infty\} \end{array}$$

We call P safe if no memory access is unsafe, i.e. for every initial state s no directives LU **a** n, SU **a** n occur in the speculation-free behavior $\mathsf{Beh}\, P\, s$. For the remaining paper we assume safe programs. Note that this does not mean that speculating memory accesses need to be safe. As seen in Code 1, Spectre Attacks utilize the fact that safe programs are not safe under speculative semantics.

Example 1. Code 2 contains a simplified version of Code 1. The secret is already in a register ρ `secret` = v and to be stored to `buf` at offset `b`. Assume the offset is out of bounds, ρ `b` = 8 $\notin$ |`buf`|. From a state $s = (2, \rho, \mu)$, where the first instruction was already executed, i.e. ρ `a` = f $\neq$ 0, the following transitions are available: First, a speculation is started with SPEC and s is copied with program counter set to the incorrect branch 3. Next, STORE-UNSAFE executes on directive su `stk` 0, $\mu' = \mu[(\texttt{stk}, 0) \mapsto \text{v}]$. The LOAD then brings the secret to a register, $\rho' = \rho[\texttt{bytes} \mapsto \text{v}]$. Finally, BRANCH leaks whether the secret is 0,

```
1 a = (b < buf_size)
2 br (a),3, 4
3     buf[b] = secret
4 bytes = stk[0]
5 br (bytes),6, 6
6 ret
```

Code 2. Simplified Code 1

$$s \xrightarrow{\text{sp:br f}} s.(3, \rho, \mu) \xrightarrow{\text{su stk 0:st 8}} s.(4, \rho, \mu') \xrightarrow{\bullet\text{:ld 0}} s.(5, \rho', \mu') \xrightarrow{\text{br:br v=0}} s.(6, \rho', \mu') .$$

Speculative semantics exhibit two important properties: First, due to the constant-time leakage model, speculative semantics reveal the program counter to the attacker: The program counter can be deduced from the leakage of conditionals in rules BRANCH and SPEC. Second, directives resolve all non-determinism introduced by speculation. To express the first property, we write $S \equiv T$ to mean that S and T are at the same program point. For speculation-free states, $s \equiv \text{pc}$ means that s is at program counter pc, $s = (\text{pc}, \rho, \mu)$. Then, $s \equiv t$ means that s and t share the program counter, $s \equiv \text{pc} \equiv t$. For speculating states, we write $S \equiv T$ if each pair of configurations in their speculation stack is at the same program point. Formally, $\varepsilon \equiv \varepsilon$, and $S.s \equiv T.t$ if $S \equiv T$ and $s \equiv t$. The following two lemmas express the properties.

LEMMA 1 (PROGRAM-COUNTER-LEAKAGE). *If two same-point states $S_1 \equiv S_2$ execute with the same directives and leakages, $S_1 \xrightarrow{d:l}{}^{*} T_1$, $S_2 \xrightarrow{d:l}{}^{*} T_2$, then the resulting states are also same-point, $T_1 \equiv T_2$.*

LEMMA 2 (DIRECTIVE-DETERMINISM). *For all S and δ there exist at most one T and λ with $S \xrightarrow{\delta:\lambda} T$.*

Notation. Similar to how we access the current instruction with $P\,s$, we write $f\,s$ instead of f pc when $s = (\text{pc}, \rho, \mu)$ for any $f : PC \to A$. Further, we extend not only the $\equiv$-relation to speculating states, but any relation $R \subseteq State \times State$ is extended to a relation on $SState$ in the obvious way: $\varepsilon\ R\ \varepsilon$ and $S.s\ R\ T.t$, if $S\ R\ T$ and $s\ R\ t$.

3 Non-intereference Properties

We define non-interference and non-interference preservation for our speculative semantics. We require the initial state's memory to be partitioned into *public* and *sensitive* data through a security level assignment $\text{sec} : Var \to SecLvl$ to a lattice $SecLvl = (\{\mathsf{L}, \mathsf{H}\}, \mathsf{L} \leq \mathsf{H})$. Sensitive data (H) is considered unknown to the attacker, and we say that initial states are indistinguishable to the attacker, $(\text{init}, \rho, \mu) =_{\text{sec}} (\text{init}, \rho, \mu')$, when the memory coincides on all variables considered public. That is, for all $\mathbf{a} \in Var$ with $\text{sec}\,\mathbf{a} = \mathsf{L}$, $\mu\,\mathbf{a} = \mu'\,\mathbf{a}$.

Our formulation of speculative non-interference (SNi, Definition 1) requires indistinguishable initial states $s_1 =_{\text{sec}} s_2$ to produce equal behavior. That means that for both initial states **(i)** the sequences of executable directives are the same, and **(ii)** for each sequence of executable directives, the observable leakage is the same. Let us explain the necessity for the first condition. As long as the second condition is satisfied, any two executions $s_1 \xrightarrow{d:l} T_1$ and $s_2 \xrightarrow{d:l} T_2$ will stay in the same program point $T_1 \equiv T_2$ (Lemma 1) and synchronously execute the same instructions. If at T_1 the set of executable directives is different to those in T_2, then the instruction has to be a memory access. All other instructions have the same set of executable directives, independent of register and memory contents. This means that one state executes an unsafe memory accesses (LOAD-UNSAFE or STORE-UNSAFE) while the other executes a safe memory accesses (LOAD or STORE). However, unsafe and safe memory accesses both leak the address used. A difference in executable directive thus amounts to different leakage.

Definition 1 (SNi). *A program is speculatively non-interferent,* $P \vDash \mathsf{SNi}$*, if all indistinguishable initial states* $S_1 =_{\mathsf{sec}} S_2$ *have the same behavior* $\mathsf{SBeh}\, P\, S_1 = \mathsf{SBeh}\, P\, S_2$.

Our goal is to prove preservation of non-interference for compiler transformations. We model compiler transformations $[.]$ that map a source program P to the transformed target program $[P]$. Transformations may modify the structure of initial states from source program P to target program $[P]$. For example, a pass that realizes the architecture's calling convention relocates function parameters to specific registers. We require each pass to come with a relation $\prec$ on initial states that identifies the initial states of $[P]$ with the initial states of P. In order to define preservation of non-interference, the relation has to respect sec in the following sense:[5]

Definition 2. *A relation* $\prec \subseteq \mathit{SState}_{[P]} \times \mathit{SState}_P$ *respects* sec *if every initial* $t \in \mathit{SState}_{[P]}$ *is mapped to an initial* $s \in \mathit{SState}_P$ *with* $t \prec s$*, and for all pairs of initial states* $t_1 \prec s_1$ *and* $t_2 \prec s_2$*:* $t_1 =_{\mathsf{sec}} t_2$ *if and only if* $s_1 =_{\mathsf{sec}} s_2$.

Speculative non-interference preservation for a transformation $[.]$ asks whether for all source programs P, $P \vDash \mathsf{SNi}$ entails $[P] \vDash \mathsf{SNi}$. However, defining preservation in this way leads to potentially surprising outcomes. Even if the source program P fails to be SNi, it can have some indistinguishable initial states which produce equal leakage. One would expect that a speculative non-interference preserving compiler transformation preserves this equal leakage to the target program $[P]$. But the above definition gives no such guarantee: If P fails to be SNi, there are no guarantees for $[P]$ at all. To counteract that, our definition of speculative non-interference preservation is more precise [Patrignani and Guarnieri 2021]. It requires preservation of equal leakage for every pair of source program's and target program's initial states individually. In particular, this definition entails that if $P \vDash \mathsf{SNi}$ then also $[P] \vDash \mathsf{SNi}$.

Definition 3 (SNiP). *A program translation* $[.]$ *with* sec*-respecting mapping* $\prec$ *is* SNi*-preserving,* $[.] \vDash \mathsf{SNiP}$*, if all initial states* $t_1 =_{\mathsf{sec}} t_2$ *of* $[P]$ *with initial source states* $t_1 \prec s_1$ *and* $t_2 \prec s_2$ *of equal behavior* $\mathsf{SBeh}\, P\, s_1 = \mathsf{SBeh}\, P\, s_2$ *also have equal target behavior,* $\mathsf{SBeh}\, [P]\, t_1 = \mathsf{SBeh}\, [P]\, t_2$.

4 Proving Speculative Non-interference Preservation

We present our proof method for speculative non-interference preservation. We introduce *snippy simulations* which ensure that a code transformation preserves speculative non-interference:

Theorem 2. *If for all* P *there is a snippy simulation* $(\prec, \lhd)$ *between* $[P]$ *and* P*, then* $[.] \vDash \mathsf{SNiP}$.

In order to reach that goal, we first define simulations that transform directives to cope with the fact that compilers do not preserve executable sequences of directives. We then introduce the constraints a snippy simulation needs to additionally satisfy and finally prove Theorem 2. This reduces proving a transformation SNiP to proving that it has a snippy simulation for each program P. In Section 5, we show how to craft a snippy simulation that is parametric in P, reducing proof effort to a once-and-for-all proof.

4.1 Simulation with Directive Transformation

The new feature in our work is *directive transformation*. Conceptually, a simulation between the target program $[P]$ and the source program P shall replay any execution of $[P]$ in P. A directive sequence d selects a single execution in $[P]$ (Lemma 2). Our simulation wants to select a corresponding execution in P. However, the directives d may not be executable in P, or it might select an inappropriate execution. Instead, a different sequence of directives may be necessary on the source program, since transformations $[.]$ are not designed to preserve them.

[5] One could also have a second security assignment on the target program, but for simplicity we assume they are the same.

Source P

```
1 if (i < buf_size)
2     a = buf[i];
3 a = 0;
4 ret;
```

Target $[P]$

```
a if (i < buf_size)
b     nop;
c a = 0;
d ret;
```

Code 3. Example code transformation from dead code elimination.

Example 3. We accompany our formal development with the example transformation in Code 3, where an unnecessary a := buf[i] instruction is replaced by a nop. Consider an initial target state $t = (\mathsf{a}, \rho, \mu)$ where $\rho\ \mathsf{i} \notin |\mathsf{buf}|$, and the source state $s = (1, \rho, \mu)$. The directives SP.●.● are executable from t. But the same sequence cannot be executed from s: An unsafe load necessitates a directive LU a m for any a and m. Thus, a transformed sequence of directives SP.LU a m.● is executed.

A simulation with directive transformation (dt−sim) is a relation on states $T \prec S$ where a target state $T \in \mathit{SState}_{[P]}$ is related to a source state $S \in \mathit{SState}_P$. The directive transformation is a family of relations $\triangleleft_{T \prec S} \subseteq \mathit{Direct}^* \times \mathit{Direct}^*$. We characterize dt−sim in Rules 4.1. Consider any states $T \in \mathit{SState}_{[P]}$ and $S \in \mathit{SState}_P$ with $T \prec S$. To express that T is simulated by S means the following: Either both states are final (FINAL), or we have to explore all sequences of executable directives $T \xrightarrow{d:l}{}^* V$ in $[P]$ up to some bound (TGT). For each explored sequence of directives we apply the directive transformation $\triangleleft_{T \prec S}$. Then, we need to replay the execution with a sequence of executable directives $S \xrightarrow{e:k}{}^* U$ in P (SRC), so that $V \prec U$. Formally, we write $\langle \prec, \triangleleft \vdash T \prec_\mathrm{t} S : d \rangle$ to express that we are exploring executions in $[P]$, have already seen a sequence of directives d and arrived at target state T. We can now either bound the exploration with DIRECT-TF, or continue exploration via TGT. With DIRECT-TF we look up a directive transformation for the explored d and swap to $\langle \prec, \triangleleft \vdash V \prec_\mathrm{s} S : e \rangle$. This states that we are seeking to replay the explored sequence with its transformation e from S. If e is executable $S \xrightarrow{e:k}{}^* U$ in P, COIND checks that the states reached from exploration in $[P]$ and replay in P are again related, $V \prec U$. The notions of $\boxed{\prec}$ and $\boxed{\dashrightarrow}$ make sure that both exploration in $[P]$ and replaying in P take at least one execution step. The guarded version $\boxed{\prec}$ requires at least one application of TGT or SRC to become the unguarded version $\boxed{\dashrightarrow}$. Only then, DIRECT-TF and COIND become applicable. We write $\boxed{\dashrightarrow}$ for any of $\boxed{\prec}$ or $\boxed{\dashrightarrow}$.

DEFINITION 4 (dt−sim). *A simulation with directive transformation $(\prec, \triangleleft)$ consists of a relation $\prec \subseteq \mathit{SState}_{[P]} \times \mathit{SState}_P$ and a family $\triangleleft = (\triangleleft_{T \prec S})_{(T,S) \in \prec}$ so that for all initial $t \in \mathit{SState}_{[P]}$, there is an initial $s \in \mathit{SState}_P$ with $t \prec s$, and for all $T \prec S$, $\langle \boxed{\prec}, \triangleleft_{T \prec S} \vdash T \prec_\mathrm{t} S : \varepsilon \rangle$ can be proven in Rules 4.1.*[6]

Example 4. Consider once again the transformation in Code 3 and the initial states t and s from Example 3. Further, let $u = (4, \rho', \mu)$ and $v = (\mathsf{d}, \rho', \mu)$ with $\rho' = \rho[\mathsf{a} \mapsto 0]$. We want to prove that $t \prec s$ is justified, i.e. we need to construct $\triangleleft_{t \prec s}$ so that $\langle \boxed{\prec}, \triangleleft_{t \prec s} \vdash t \prec_\mathrm{t} s : \varepsilon \rangle$ is derivable. We drop the subscript and just write $\triangleleft$. Exploration via TGT yields (among others) two sequences of directives executable from t in $[P]$: SP.●.● as in Example 3 and BR. The corresponding execution takes us to $t \xrightarrow{\mathrm{SP.●.●:BR\ f.●.●}}{}^* t.v$ and $t \xrightarrow{\mathrm{BR:BR\ f}} v$, respectively. After exploration with TGT, we are thus left to prove $\langle \boxed{\dashrightarrow}, \triangleleft \vdash t.v \prec_\mathrm{t} s : \mathrm{SP.●.●} \rangle$ and $\langle \boxed{\dashrightarrow}, \triangleleft \vdash v \prec_\mathrm{t} s : \mathrm{BR} \rangle$. For the first sequence, we transform the directives as in Example 3: SP.LU sec 0.● $\triangleleft$ SP.●.●. For the other case, we do not need a transformation, so BR $\triangleleft$ BR. With DIRECT-TF, we are left with deriving $\langle \boxed{\prec}, \triangleleft \vdash t.v \prec_\mathrm{s} s : \mathrm{SP.LU\ sec\ 0.●} \rangle$ and $\langle \boxed{\prec}, \triangleleft \vdash v \prec_\mathrm{s} s : \mathrm{BR} \rangle$. Indeed, P can replay the directives with $s \xrightarrow{\mathrm{SP.LU\ sec\ 0.●:BR\ f.LD\ n.●}}{}^* s.u$ using SRC, where $\mathsf{n} = \rho\ \mathsf{i}$ and $s \xrightarrow{\mathrm{BR:BR\ f}} u$, respectively. To now utilize COIND we need $v \prec u$ and $t.v \prec s.u$. Each would again have to be justified independently. Justifying the first is easy with FINAL, while the other needs another application of TGT, DIRECT-TF, and SRC and can then utilize $t \prec s$.

[6] Our way to define simulations is inspired by recent work to unify stuttering [Cho et al. 2023].

RULES 4.1: CHARACTERIZING SIMULATIONS

$$\textsc{tgt}\ \frac{\forall T \xrightarrow{\delta:\lambda} V.\ \boxed{\prec}, \triangleleft \vdash V \prec_t S : d.\delta \quad T \text{ not final}}{\boxed{\prec}, \triangleleft \vdash T \prec_t S : d} \qquad \textsc{direct-tf}\ \frac{\boxed{\prec}, \triangleleft \vdash T \prec_s S : e \quad e \triangleleft d}{\boxed{\prec}, \triangleleft \vdash T \prec_t S : d}$$

$$\textsc{src}\ \frac{\exists S \xrightarrow{\delta:\lambda} U.\ \boxed{\prec}, \triangleleft \vdash T \prec_s U : e}{\boxed{\prec}, \triangleleft \vdash T \prec_s S : \delta.e} \qquad \textsc{coind}\ \frac{T \prec S}{\boxed{\prec}, \triangleleft \vdash T \prec_s S : \varepsilon} \qquad \textsc{final}\ \frac{T, S \text{ final}}{\boxed{\prec}, \triangleleft \vdash T \prec_t S : \varepsilon}$$

RULES 4.2: SIMULATION INTERVALS

$$\textsc{sync}\ \frac{T \xrightarrow{d:l}{}^{*} V \quad T \prec S \quad V \prec U \quad S \xrightarrow{e:k}{}^{*} U \qquad \langle T \prec_t S\rangle \overset{d}{\Leftarrow} \langle V \prec_t S\rangle \overset{e \triangleleft d}{\Longleftarrow} \langle V \prec_s S\rangle \overset{e}{\Leftarrow} \langle V \prec_s U\rangle}{S \xrightarrow{e:k}_s U \quad (S,T) \xrightarrow{e:k \triangleleft d:l} (U,V) \quad T \xrightarrow{d:l}_t V}$$

Simulation Intervals. Our goal is to formulate snippy simulations as a constraint on simulations with directive transformation. We define it with simulation intervals. A simulation interval for states $T \prec S$ is a pair of an explored sequence of directives from T in $[P]$ and the corresponding replay from S in P. Formally, we define simulation intervals through a synchronized product, whose transitions are the simulation intervals. The states of the synchronized product are of shape (S, T) so that $T \prec S$. Its transitions are of the form $(S, T) \xrightarrow{e:k \triangleleft d:l} (U, V)$, where e is the directive transformation of an explored d in $[P]$. In order to formally define the transition relation, consider a proof tree that justifies $T \prec S$, i.e. that derives $\langle \boxed{\prec}, \triangleleft_{T \prec S} \vdash T \prec_t S : \varepsilon\rangle$. We use the notation

$$\langle T \prec_t S\rangle \overset{d}{\Leftarrow} \langle V \prec_t S\rangle \overset{e \triangleleft d}{\Longleftarrow} \langle V \prec_s S\rangle \overset{e}{\Leftarrow} \langle V \prec_s U\rangle$$

to state that the proof tree contains the nodes $\langle \boxed{\prec}, \triangleleft_{S \prec T} \vdash V \prec_t S : d\rangle$, $\langle \boxed{\prec}, \triangleleft_{S \prec T} \vdash V \prec_s S : e\rangle$, and $\langle \boxed{\prec}, \triangleleft_{S \prec T} \vdash V \prec_s U : \varepsilon\rangle$ on one path. In particular, this means that $T \xrightarrow{d:l}{}^{*} V$ and $S \xrightarrow{e:k}{}^{*} U$ for appropriate l, k, and $e \triangleleft_{T \prec S} d$. We define a synchronized transition relation that executes both in a single step.

DEFINITION 5. *Given a simulation $(\prec, \triangleleft)$, its simulation interval transition is defined by Rule* SYNC.

$$\xrightarrow{\cdot \triangleleft \cdot} \quad \subseteq \quad \prec \times \mathit{Direct}^* \times \mathit{Leak}^* \times \mathit{Direct}^* \times \mathit{Leak}^* \times \prec$$

SYNC further defines the transition relations $\xrightarrow{\cdot}_s$ and $\xrightarrow{\cdot}_t$ as the projection of simulation intervals to source and target program. Transitive closures of the transition relations are defined as usual. We say that a simulation is lock-step if simulation intervals are single step: $\xrightarrow{\cdot}_t, \xrightarrow{\cdot}_s \subseteq \xrightarrow{\cdot}$.

Example 5. The simulation intervals resulting from Example 4 for $t \prec s$ are:

$$(s,t) \xrightarrow{\text{SP.LU secret } 0.\bullet : \text{BR f.LD n.}\bullet \ \triangleleft\ \text{SP.}\bullet.\bullet : \text{BR f.}\bullet.\bullet} (s.u, t.v) \qquad (s,t) \xrightarrow{\text{BR} : \text{BR f} \ \triangleleft\ \text{BR} : \text{BR f}} (u,v)$$

The following lemma states that our formulation of simulations is sound. That is, we find all of $[P]$'s behavior in the projection of the simulation interval transition relation $\xrightarrow{\cdot}_t$. This lets us perform (co-)induction on SBeh $[P]$ S with $\xrightarrow{\cdot}_t$ rather than $\xrightarrow{\cdot}$, which we will utilize in our proof of Theorem 2. The same is not true for the source program's behavior and $\xrightarrow{\cdot}_s$.

LEMMA 3. *If T occurs in $\prec$,* SBeh $[P]$ $T = \{(d{:}l) \mid T \xrightarrow{d:l}{}^{*}_t X, X \text{ final}\} \cup \{(d{:}l) \mid T \xrightarrow{d:l}_t \infty\}$.

4.2 Snippy Simulations

So far, simulations are very liberal: Simulation merely require that a sequence of directives in $[P]$ can be transformed via $\lhd$ into a sequence of directives in P. The length and contained directives can change when applying $\lhd$ and there are no restrictions on how the leakage changes when applying $\lhd$. In this section, we establish *snippy simulations*, a constant-time cube constraint [Barthe, Grégoire, and Laporte 2018] on simulations for speculative semantics that entails SNiP when satisfied.

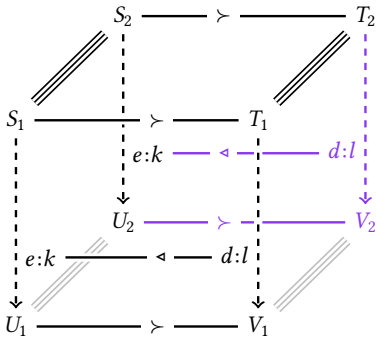

Fig. 2. Whenever the black conditions are met, a snippy simulation $\prec$ also explores the purple execution and simulates it by the source execution. Gray conditions follow from the semantics.

The intuition for snippy simulations can be explained as follows. In order to prove that a transformation $[.]$ satisfies SNiP, we are given a source program P and the target program $[P]$, as well as four initial states: Two target initial states $t_1 \equiv t_2$ and two simulating source states $t_1 \prec s_1$ and $t_2 \prec s_2$, so that $\mathsf{SBeh}\, P\, s_1 = \mathsf{SBeh}\, P\, s_2$. The goal is to prove that the equality of behavior carries over to the target program. Given a simulation $(\prec, \lhd)$ we so far know how to replay any sequence of directives d from t_1 transformed on s_1 (Lemma 3). Consider a simulation interval $(s_1, t_1) \xrightarrow{e:k \lhd d:l} (U_1, V_1)$. Due to same behavior of s_1 and s_2, the source directives can also be executed from s_2, $s_2 \xrightarrow{e:k} U_2$. Snippy simulations now state that, in such a situation, the simulation interval for s_2 and t_2 also contains $(s_2, t_2) \xrightarrow{e:k \lhd d:l} (U_2, V_2)$. That means, t_2 can also execute d and produce the same leakage. And further, $(\prec, \lhd)$ does also explore d from t_2, not a longer or shorter sequence. Figure 2 demonstrates the constraint in the general case, where states need not be initial. The simulation interval of S_1 and T_1 and the ability for another source state S_2 to mimic the behavior are in black. The constraint is in purple: $(\prec, \lhd)$ has to also provide the same simulation interval for any other state $T_2 \succ S_2$ at the same program point as T_1.

With snippy simulations defined, we conclude the section with the proof of Theorem 2.

Definition 6. *A snippy simulation $(\prec, \lhd)$ is* sec*-respecting and satisfies the diagram in Figure 2. That is, for all $S_1 \equiv S_2$ and $T_1 \equiv T_2$ with $T_1 \prec S_1$, $T_2 \prec S_2$, and $(S_1, T_1) \xrightarrow{e:k \lhd d:l} (U_1, V_1)$,*

$$S_2 \xrightarrow{e:k}{}^{*} U_2 \qquad \textit{implies the existence of } V_2 \textit{ with} \qquad (S_2, T_2) \xrightarrow{e:k \lhd d:l} (U_2, V_2)\,.$$

Proof of Theorem 2. Consider a program P and a snippy simulation $(\prec, \lhd)$. We need to prove the following: For all initial $t_1 =_{\mathsf{sec}} t_2$ with $t_1 \prec s_1$ and $t_2 \prec s_2$: When $\mathsf{SBeh}\, [P]\, S_1 = \mathsf{SBeh}\, [P]\, s_2$, then also $\mathsf{SBeh}\, [P]\, t_1 = \mathsf{SBeh}\, [P]\, t_2$. We claim a stronger statement: Whenever $T_1 \prec S_1$, $T_2 \prec S_2$, $S_1 \equiv S_2$, and $T_1 \equiv T_2$, and $\mathsf{SBeh}\, P\, S_1 = \mathsf{SBeh}\, P\, S_2$: Then $\mathsf{SBeh}\, [P]\, T_1 \subseteq \mathsf{SBeh}\, [P]\, T_2$ holds.

This is sufficient: Consider initial target states $t_1 =_{\mathsf{sec}} t_2$ as well as source states $t_1 \prec s_1$ and $t_2 \prec s_2$ with $\mathsf{SBeh}\, P\, s_1 = \mathsf{SBeh}\, P\, s_2$. Initial states are all at the same program point, so the requirements of the claim are satisfied and $\mathsf{SBeh}\, [P]\, t_1 \subseteq \mathsf{SBeh}\, [P]\, t_2$ holds. By symmetry, $\mathsf{SBeh}\, [P]\, t_1 = \mathsf{SBeh}\, [P]\, t_2$.

We prove our claim coinductively on $\mathsf{SBeh}\, [P]\, T_1$ split into simulation intervals (Lemma 3). The case of a final T_1, i.e. $\mathsf{SBeh}\, [P]\, T_1 = \{(\varepsilon{:}\varepsilon)\}$, $T_2 \equiv T_1$ is final, too, and thus $(\varepsilon{:}\varepsilon) \in \mathsf{SBeh}\, [P]\, T_2$. In the (co-)inductive case, let $(e.d{:}k.l) \in \mathsf{SBeh}\, [P]\, T_1$ with $T_1 \xrightarrow{e:k} t\, V_1$ from a simulation interval $(S_1, T_1) \xrightarrow{f:m \lhd e:k} (U_1, V_1)$. From $\mathsf{SBeh}\, P\, S_1 = \mathsf{SBeh}\, P\, S_2$ follows $S_2 \xrightarrow{f:m}{}^{*} U_2$. Lemma 1 gives $U_1 \equiv U_2$ and same behavior of S_1 and S_2 entails $\mathsf{SBeh}\, P\, U_1 = \mathsf{SBeh}\, P\, U_2$. Snippyness then yields the simulation interval $(S_2, T_2) \xrightarrow{f:m \lhd e:k} (U_2, V_2)$, i.e. $T_2 \xrightarrow{e:k}{}^{*} V_2$. Lemma 1 gives $V_1 \equiv V_2$. We can now apply (co-)induction hypothesis for $\mathsf{SBeh}\, [P]\, V_1 \subseteq \mathsf{SBeh}\, [P]\, V_2$, which implies $(e.d{:}k.l) \in \mathsf{SBeh}\, [P]\, V_2$. Together with $T_2 \xrightarrow{e:k}{}^{*} V_2$, we arrive at $(e.d{:}k.l) \in \mathsf{SBeh}\, [P]\, T_2$. □

5 Case Study: Dead Code Elimination

In this section, we prove that dead code elimination $[.]^{\mathrm{dc}}$ satisfies SNiP as a demonstration of the proof method and a warm-up for the next section, where we tackle register allocation transformations. We first give a short rundown on the transformation of $[.]^{\mathrm{dc}}$ before crafting a snippy simulation that is parametric in the source program P. Dead code elimination is responsible for removing instructions from P whose computed values are not utilized anywhere. It is the result of a data flow analysis that finds removable instructions. Typically, the analysis follows constant propagation in order to identify as many aliasing memory accesses as possible. To support that, we assume that load and store instructions $a := \mathbf{a}[\mathsf{n}]$ and $\mathbf{a}[\mathsf{n}] := c$ can hold a constant address $\mathsf{n} \in \mathit{Adr}$ instead of a register, where we require $\mathsf{n} \in |\mathbf{a}|$. For the remaining section, fix an arbitrary program P with entry point init.

Flow analysis. The static analysis for dead code elimination is a Liveness analysis. Since we will later define our own flow analysis when fixing the weaknesses in register allocation, we recall flow analysis in a more general setting. A flow analysis searches for a fixed-point solution to a system of flow inequalities in order to obtain approximate knowledge about all executions of a program. Formally, a *forward/backward flow analysis* finds a solution $\mathcal{X}$ to the inequalities (fwd)/(bwd), where pc and sc range over program counters so that pc is a predecessor to sc.

$$\begin{aligned} &\mathcal{X}\,\mathsf{sc} \geq F_{\mathsf{pc}}\,(\mathcal{X}\,\mathsf{pc}) \\ &\mathcal{X}\,\mathsf{init} \geq \mathcal{X}_0 \end{aligned} \qquad \text{(fwd)} \qquad\qquad \begin{aligned} &\mathcal{X}\,\mathsf{pc} \geq F_{\mathsf{sc}}\,(\mathcal{X}\,\mathsf{sc}) \\ &\mathcal{X}\,\mathsf{pc} \geq \mathcal{X}_0 \end{aligned} \quad P\,\mathsf{pc} = \texttt{ret} \qquad \text{(bwd)}$$

Flow values stem from a semi-lattice $(L, \leq)$ and a solution $\mathcal{X} : \mathit{PC} \to L$ finds a flow value for each program point. In case of a forward analysis, $\mathcal{X}\,\mathsf{pc}$ denotes the flow value at pc before the execution of $P\,\mathsf{pc}$. For a backward analysis $\mathcal{X}\,\mathsf{pc}$ denotes the flow value at pc after the execution of $P\,\mathsf{pc}$. The flow value $\mathcal{X}_0$ is the initial flow value of entry / exit points of the program. The functions $F_{\mathsf{pc}} : L \to L$ are monotonic and constitute the *transfer* of the flow values along instructions.

A flow analysis can have *additional constraints*. Additional constraints are of shape $\mathcal{X}\,\mathsf{pc} \leq l$. They additionally require flow values of certain program counters pc not to exceed a bound $l \in L$. If the least solution to the flow analysis does not satisfy the additional constraints, no solution does.

Dead Code Elimination. Liveness analysis is a backward flow analysis. The flow values are the sets of registers and memory locations that are live at any given program point in that their current value could be used later. The flow lattice is $L = \mathcal{P}(\mathit{Reg} \cup \mathit{Mem})$, and the initial flow value at any exit point is $\mathcal{X}_0 = \mathit{Reg} \cup \mathit{Mem}$, but can be different dependent on calling conventions. The transfer functions are folklore, so we instead formulate the guarantee that comes with a solution. Note that the guarantee holds for speculative semantics, too, because the analysis is branch-independent.

Proposition 6. *Whenever $s \xrightarrow{d:t}{}^{*} U.u$ in P, if $P\,u$ uses a register b, then $b \in F_u\,(\mathcal{X}\,u)$, and if $P\,u$ loads a memory location $\mathbf{a}$ with offset n, then $(\mathbf{a}, \mathsf{n}) \in F_u\,(\mathcal{X}\,u)$.*

The transformation $[.]^{\mathrm{dc}}$ uses a Liveness analysis solution $\mathcal{X}$ of the backward flow inequalities (bwd) to remove unnecessary instructions. A function $dc : L \to \mathit{Inst} \to \mathit{Inst}$ inspects the flow value at a given program point and removes an instruction if it writes a register or memory location that is not live. The transformation of P is then defined per program point with $[P]^{\mathrm{dc}}\,\mathsf{pc} = dc\,(\mathcal{X}\,\mathsf{pc})\,(P\,\mathsf{pc})$.

$$dc\,l\,(a := b \oplus c_{\rangle\mathsf{sc}}) = \begin{cases} \mathsf{nop}_{\rangle\mathsf{sc}} & a \notin l \\ a := b \oplus c_{\rangle\mathsf{sc}} & a \in l \end{cases} \qquad dc\,l\,(a := \mathbf{a}[x]_{\rangle\mathsf{sc}}) = \begin{cases} \mathsf{nop}_{\rangle\mathsf{sc}} & a \notin l \\ a := \mathbf{a}[x]_{\rangle\mathsf{sc}} & a \in l \end{cases}$$

$$dc\,l\,(\mathbf{a}[x] := c_{\rangle\mathsf{sc}}) = \begin{cases} \mathsf{nop}_{\rangle\mathsf{sc}} & x = \mathsf{n}, (\mathbf{a}, \mathsf{n}) \notin l \\ \mathbf{a}[x] := c_{\rangle\mathsf{sc}} & \text{otherwise} \end{cases} \qquad dc\,l\,i = i \quad \text{for other } i$$

Dead Code Simulation. We now craft a snippy simulation $(\prec, \lhd)$ between $[P]^{\mathrm{dc}}$ and P. The simulation is parametric in P as it depends on the Liveness analysis $\mathcal{X}$. That way, we craft a single simulation relation and provide one for every source program P and its transformation $[P]^{\mathrm{dc}}$. First, we define the simulation $t \prec s$ and directive transformation $\lhd_{t \prec s}$ for speculation-free states, and afterwards lift them to speculating states. The simulation identifies states whenever they differ only on dead registers and memory locations:

$$(\mathrm{pc}, \rho, \mu) \prec (\mathrm{pc}, \rho', \mu') \iff \begin{array}{r} \forall a \in F_{\mathrm{pc}}\,(\mathcal{X}\,\mathrm{pc}).\ \rho\, a = \rho'\, a \\ \wedge\quad \forall (\mathbf{a}, \mathsf{n}) \in F_{\mathrm{pc}}\,(\mathcal{X}\,\mathrm{pc}).\ \mu\, \mathbf{a}\, \mathsf{n} = \mu'\, \mathbf{a}\, \mathsf{n} \end{array}$$

In order to define the directive transformation $\lhd_{t \prec s}$, we need to think about the shape of simulation intervals. Because $[.]^{\mathrm{dc}}$ leaves the control flow fully intact, we can choose to create a lockstep simulation. With that, we can choose $\lhd_{t \prec s}$ to be the identity relation on *Direct* and add transformations where $[.]^{\mathrm{dc}}$ replaced an instruction with nop. Let $i = P\,s$ and $i' = [P]^{\mathrm{dc}}\,t$. We set:

$$\lhd_{t \prec s} = id_{Direct} \cup \begin{cases} \{(\textsc{lu}\,\mathbf{b}\,\mathsf{m}, \bullet) \mid \mathbf{b} \in Var, \mathsf{m} \in Adr\} & i = a := \mathbf{a}[b]_{\rangle \mathrm{sc}},\quad i' = \mathsf{nop}_{\rangle \mathrm{sc}} \\ \{(\textsc{su}\,\mathbf{b}\,\mathsf{m}, \bullet) \mid \mathbf{b} \in Var, \mathsf{m} \in Adr\} & i = \mathbf{a}[b] := c_{\rangle \mathrm{sc}},\quad i' = \mathsf{nop}_{\rangle \mathrm{sc}} \\ \varnothing & \text{otherwise} \end{cases}$$

For speculating states, we simply lift $\prec$ by setting $T.t \prec S.s$ if $T \prec S$ and $t \prec s$ (and $\varepsilon \prec \varepsilon$ for the base case). For the directive transformation, we delegate to the executing states $\lhd_{T.t \prec S.s} = \lhd_{t \prec s}$.

Theorem 7. *$(\prec, \lhd)$ is a snippy lockstep simulation.*

Proof. We need to prove that $(\prec, \lhd)$ **(i)** is a simulation (Definition 4), **(ii)** is snippy (Definition 6), and **(iii)** respects sec (Definition 2). The first part **(i)** is considerably easier than in the general case, because the simulation is lockstep, i.e. both P and $[P]^{\mathrm{dc}}$ only perform a single step before finding new states in $\prec$. Consider $T_1.t_1 \prec S_1.s_1$, $T_2.t_2 \prec S_2.s_2$, $T_1.t_1 \equiv T_2.t_2$ $S_1.s_1 \equiv S_2.s_2$ and $T_1.t_1 \xrightarrow{\delta:\lambda} V_1$, where $t_1 \equiv t_2 \equiv s_1 \equiv s_2 \equiv \mathrm{pc}$. We need to show that there is $\gamma \lhd_{t_1 \prec s_1} \delta$ so that $S_1.s_1 \xrightarrow{\gamma:\kappa} U_1$ and $V_1 \prec U_1$. Second we prove snippyness **(ii)**: We additionally consider the existence of $S_2.s_2 \xrightarrow{\gamma:\kappa} U_2$. We then need to show that $T_2.t_2 \xrightarrow{\delta:\lambda} V_2$ and $\gamma \lhd_{t_2 \prec s_2} \delta$ (which implies $(s_2, t_2) \xrightarrow{\delta:\lambda \lhd \gamma:\kappa} (u_2, v_2)$).

We first split off the case $\delta = \gamma = \textsc{rb}$, where $|T_1| > 0$. For **(i)**, we need to prove $S_1.s_1 \xrightarrow{\textsc{rb}:\textsc{rb}} S_1$ because $\textsc{rb} \lhd_{t_1 \prec s_1} \textsc{rb}$. But $T_1 \prec S_1$ implies $|T_1| = |S_1| > 0$ which meets the premise for the transition. For **(ii)**, additionally consider $S_2.s_2 \xrightarrow{\textsc{rb}:\textsc{rb}} S_2$. We need to show $T_2.t_2 \xrightarrow{\textsc{rb}:\textsc{rb}} T_2$. With the same argument as before, $|T_1| = |S_1| = |S_2| = |T_2| > 0$ satisfies the premise for the transition. Proving **(i)** and **(ii)** in the case of $\delta \neq \textsc{rb} \neq \gamma$ is a large case distinction on $\langle P\,\mathrm{pc}, [P]^{\mathrm{dc}}\,\mathrm{pc}\rangle$.

- ▶ $\langle a := \mathbf{a}[b]_{\rangle \mathrm{sc}}, \mathsf{nop}_{\rangle \mathrm{sc}}\rangle$ There are two subcases: $\rho_{s_1}\, b = \mathsf{n}$ is within $|\mathbf{a}|$ or not. We present the subcase $\mathsf{n} \notin |\mathbf{a}|$. For **(i)**, consider $t_1 \xrightarrow{\bullet:\bullet} v_1$. We need to show that $\textsc{lu}\,\mathbf{b}\,\mathsf{m}$ is executable in s_1, because $\textsc{lu}\,\mathbf{b}\,\mathsf{m} \lhd_{t_1 \prec s_1} \bullet$. Indeed, we have $s_1 \xrightarrow{\textsc{lu}\,\mathbf{b}\,\mathsf{m}:\textsc{ld}\,\mathsf{n}} u_1$ because $\rho_{s_1}\, b = \mathsf{n} \notin |\mathbf{a}|$. We also need to show that $v_1 \prec u_1$. From the definition of *dc*, we know $a \notin \mathcal{X}\,\mathrm{pc}$ because the instruction was replaced by nop. The transfer $F_{\mathrm{pc}}\,(\mathcal{X}\,\mathrm{pc})$ is again $\mathcal{X}\,\mathrm{pc}$ because a load to a dead register makes no registers or memory locations live. Together with (bwd), we get $F_{\mathrm{pc}}\,(\mathcal{X}\,\mathrm{pc}) = \mathcal{X}\,\mathrm{pc} \supseteq F_{\mathrm{sc}}\,(\mathcal{X}\,\mathrm{sc})$. Registers and memory of t_1 and v_1 are equal (nop) and s_1 and u_1 only differ on a. This means $t_1 \prec s_1$ implies $v_1 \prec u_1$ because $a \notin \mathcal{X}\,\mathrm{pc} \supseteq F_{\mathrm{sc}}\,(\mathcal{X}\,\mathrm{sc})$. For **(ii)**, further consider $s_2 \xrightarrow{\textsc{lu}\,\mathbf{b}\,\mathsf{m}:\textsc{ld}\,\mathsf{n}} u_2$. We need to show that $t_2 \xrightarrow{\bullet:\bullet} v_2$ and $v_2 \prec u_2$. The former is immediate from the semantics. For the latter, we apply the same arguments as in **(i)**. All other cases are very similar.

For **(iii)**, we need to restrict our attention to Liveness analysis where all registers and memory locations are initially live.[7] Every solution $\mathcal{X}$ is easily modified to satisfy this restriction. The proof that $\prec$ respects sec is then straightforward: For initial states t and s, $t \prec s$ if and only if $t = s$. □

[7] A weaker formalization of sec-respecting relations would lift this restriction, but incur more presentational overhead.

6 Fixing Weaknesses in Register Allocation

Register allocation happens in the compilation phase that moves from an IR to the hardware instructions of the target architecture. It moves from the unbounded number of virtual registers occurring in the IR version of the program to the finite set of hardware registers. In order to do so, register allocation performs a relocation of register contents. For each program point, a subset of the virtual registers is selected to be kept as hardware registers and the remaining virtual registers are spilled to the stack. The literature describes various approaches towards selecting the set of registers to be spilled [Chaitin et al. 1981; Traub et al. 1998; Poletto and Sarkar 1999; Tichadou and Rastello 2022], leading to different transformations depending on the chosen algorithm. In this paper, we express the transformations from register allocation more generally as a set of constraints. A transformation constitutes a viable register allocation if it satisfies the constraints. Practical allocation algorithms produce viable allocations making our results apply to all of them.

We first present the constraints a register allocation transformation needs to satisfy. We then demonstrate how the transformation violates speculative non-interference preservation. We continue to develop a static analysis that finds potential violations. Finally, we fix the violations and craft a snippy simulation for the fixed transformation and prove that it satisfies SNiP.

6.1 Register Allocation

Register allocation transforms a source program P by inserting *shuffle instructions*.

$$si \in \mathit{Shuffle\ Inst} \ ::= \ a \coloneqq b_{\rangle \mathsf{sc}} \mid a \coloneqq \mathbf{stk}[\mathsf{l}]_{\rangle \mathsf{sc}} \mid \mathbf{stk}[\mathsf{l}] \coloneqq b_{\rangle \mathsf{sc}} \mid \mathtt{slh}\, a_{\rangle \mathsf{sc}} \mid \mathtt{sfence}_{\rangle \mathsf{sc}}$$

The instructions extend *Inst* and are inserted in between the existing instructions in order to maintain the register relocation with the instructions move $a \coloneqq b$, fill $a \coloneqq \mathbf{stk}[\mathsf{l}]$, and spill $\mathbf{stk}[\mathsf{l}] \coloneqq b$. We also include $\mathtt{slh}\, a$ and $\mathtt{sfence}$ because we need them later to fix the transformation. The semantics are as expected: A move $a \coloneqq b$ relocates contents from b to a, $a \coloneqq \mathbf{stk}[\mathsf{l}]$ reloads a spilled register from a constant address $\mathsf{l} \in \mathit{Adr}$ in the stack, and $\mathbf{stk}[\mathsf{l}] \coloneqq b$ spills a register to the stack. We model the stack frame's section used for spilled registers with a fresh memory variable $\mathbf{stk}$ not occurring in P. We assume that $\mathbf{stk}$ has appropriate size to fit all spilled registers and is typed $\mathsf{sec}\ \mathbf{stk} = \mathsf{L}$. Shuffle code is always straight line, so for a shuffle sequence $sh \in \mathit{Shuffle\ Inst}^*$ we introduce the notion $P\ \mathsf{pc} = sh_{\rangle \mathsf{sc}}$ to express that P executes sh and then ends in sc.

Constraints for a valid transformation. Register allocation inserts shuffle code between existing instructions to realize the register relocation. Formally, a target program $[P]^{\mathsf{ra}} : \mathit{PC}_{[P]^{\mathsf{ra}}} \to \mathit{Inst}$ is a register allocation if there exist functions Φ and Ψ. The first is an injection $\Phi : \mathit{PC}_P \to \mathit{PC}_{[P]^{\mathsf{ra}}}$ of the original instructions of P to their counterparts in $[P]^{\mathsf{ra}}$. The second function is a relocation mapping $\Psi : \mathit{PC}_{[P]^{\mathsf{ra}}} \to \mathit{Reg} \rightharpoonup (\mathit{Reg} \cup \mathit{Stk})$, where $\mathit{Stk} = \{(\mathbf{stk}, \mathsf{l}) \mid \mathsf{l} \in |\mathbf{stk}|\} \subseteq \mathit{Mem}$ is the stack frame for spilled variables. At each program counter $\mathsf{pc}' \in \mathit{PC}_{[P]^{\mathsf{ra}}}$, $\Psi\ \mathsf{pc}'\ a$ is the relocation of the virtual register a from P to the hardware register or stack location in $[P]^{\mathsf{ra}}$.[8] The functions are subject to the following conditions. *Instruction matching:* The Φ-injected instructions must operate on the same registers up to relocation by Ψ. *Shuffle conformity:* The relocation Ψ must conform to the (shuffle) instructions in $[P]^{\mathsf{ra}}$. *Obeying Liveness:* Every live register in P is mapped under Ψ and no location is doubly allocated. To state the conditions formally, we introduce notation for the defined and used registers of an instruction. A register b is used by $i \in \mathit{Inst}$ if the register is read out by the corresponding rule in Rules 2.1. Similarly, a is defined by i, if it is written in that rule. For example, $a \coloneqq \mathbf{a}[b]$ uses $\{b\}$ and defines $\{a\}$.

$$\mathsf{uses}_i = \{b \mid b \in \mathit{Reg} \text{ is used by } i\} \qquad \mathsf{def}_i = \{a \mid b \in \mathit{Reg} \text{ is defined by } i\}$$

[8] In general, the content of a register a could be relocated to multiple locations. For simplicity of presentation, we forbid that.

Source P

```
1 a = (b < buf_size)
2 br (a)›3, 4
3     buf[b] = secret
4 br (bytes)›5, 5
5 ret
```

Target $[P]^{\mathrm{ra}}$

```
a a = (b < buf_size)
b stk[0] = bytes
c br (a)›d, e
d     buf[b] = secret
e a = stk[0]
f br (a)›g, g
g ret
```

Code 4. An Example register allocation. Lines b and e are the inserted shuffle code.

Instruction Matching requires that source instructions from P reappear in $[P]^{\mathrm{ra}}$: For every $\mathsf{pc} \in PC_P$ and $\mathsf{pc}' = \Phi\,\mathsf{pc}$, $i = P\,\mathsf{pc}$ has to match with $i' = [P]^{\mathrm{ra}}\,\mathsf{pc}'$. To match, the instruction i' must be the same, but registers $b \in \mathrm{uses}_i$ are replaced with $\Psi\,\mathsf{pc}'\,b \in \mathit{Reg}$. Similarly, registers $a \in \mathrm{def}_i$ are replaced with $\Psi\,\mathsf{sc}'\,a \in \mathit{Reg}$, where sc' is the successor of pc'. Defined registers are found in the successor's relocation, because they are live only after executing i'. All other instructions at program points $\mathsf{pc}' \notin \mathrm{img}(\Phi)$ must be shuffle instructions. Given $s \in \mathit{State}_P$ and $t \in \mathit{State}_{[P]^{\mathrm{ra}}}$, we write $s\ \Phi\ t$ when $s \equiv \mathsf{pc}$ and $\Phi\,\mathsf{pc} \equiv t$. We extend the notation to $S\ \Phi\ T$ in the expected way.

Shuffle Conformity requires that the relocation Ψ is upheld by the instructions in $[P]^{\mathrm{ra}}$. Consider an instruction $i' = [P]^{\mathrm{ra}}\,\mathsf{pc}'$ at $\mathsf{pc}' \in PC_{[P]^{\mathrm{ra}}}$ and let sc' be a successor. First, registers and stack locations untouched by i' must stay at the same location in Ψ: For any register d with $\Psi\,\mathsf{pc}'\,d \notin \mathrm{uses}_{i'}$ and $\Psi\,\mathsf{sc}'\,d \notin \mathrm{def}_{i'}$, $\Psi\,\mathsf{pc}'\,d = \Psi\,\mathsf{sc}'\,d$. Second, if i' is a shuffle instruction there are additional requirements: Shuffle instructions move one source register's content from one location in $[P]^{\mathrm{ra}}$ to another. As a consequence, if i' is a shuffle instruction, there must be a source register a being moved and the location moved to must be free. We write $\mathrm{free}_{\mathsf{r}'}\,d'$ and $\mathrm{free}_{\mathsf{r}'}\,(\mathbf{stk}, \mathsf{l})$ for d', $(\mathbf{stk}, \mathsf{l}) \notin \mathrm{img}(\Psi\,\mathsf{r}')$. Dependent on i', we add the following constraints, (a is the source register):

$$
\begin{array}{lllll}
i' = a' := b'_{\mathsf{sc}'}: & \exists a.\Psi\,\mathsf{pc}'\,a = b' & \wedge \Psi\,\mathsf{r}'\,a = a' & \wedge \mathrm{free}_{\mathsf{pc}'}\,a' \\
i' = a' := \mathbf{stk}[\mathsf{l}]_{\mathsf{sc}'}: & \exists a.\Psi\,\mathsf{pc}'\,a = (\mathbf{stk}, \mathsf{l}) & \wedge \Psi\,\mathsf{r}'\,a = a' & \wedge \mathrm{free}_{\mathsf{pc}'}\,a' \\
i' = \mathbf{stk}[\mathsf{l}] := b'_{\mathsf{sc}'}: & \exists a.\Psi\,\mathsf{pc}'\,a = b' & \wedge \Psi\,\mathsf{r}'\,a = (\mathbf{stk}, \mathsf{l}) & \wedge \mathrm{free}_{\mathsf{pc}'}\,(\mathbf{stk}, \mathsf{l}) \\
i' = \mathtt{slh}\,a'_{\mathsf{sc}'}: & \exists a.\Psi\,\mathsf{pc}'\,a = a' & \wedge \Psi\,\mathsf{r}'\,a = a' &
\end{array}
$$

Obeying Liveness means that all live variables in P must be allocated. There must be a Liveness solution $\mathcal{X}$ (Proposition 6) for P so that for all locations $\mathsf{pc} \in PC_P$, all registers $a \in \mathcal{X}\,\mathsf{pc}$ live at pc are allocated, i.e. $\Psi\,(\Phi\,\mathsf{pc})\,a \neq \bot$. Further, a location cannot be allocated twice, i.e. for all $\mathsf{pc} \in PC_{[P]^{\mathrm{ra}}}$, $\Psi\,\mathsf{pc}$ forms an injection on the live registers at pc.

Definition 7. *A transformation from P to $[P]^{\mathrm{ra}}$ is a register allocation if there are instruction matching, shuffle conform, and Liveness obeying* (Φ, Ψ).

Example 8. Code 4 contains an example register allocation. It is a simplified version of Code 1, which still exhibits the weakness of register allocation. The left program starts with a secret value in register secret and public values in b and bytes. It stores the secret into buf when the address b is in bounds of |buf|. Then, it leaks bytes at 4. The right program is after register allocation. Register allocation has inserted the spill and fill instructions b and e. The instruction injection Φ can be seen from side by side alignment. It is the mapping $\{1 \mapsto \mathsf{b}, 2 \mapsto \mathsf{c}, 3 \mapsto \mathsf{d}, 4 \mapsto \mathsf{f}, 5 \mapsto \mathsf{g}\}$. The register mapping Ψ at a and b makes no relocations: $\Psi\,\mathsf{a} = \Psi\,\mathsf{b} = \mathit{id}_{\mathit{Reg}}$. The spill instruction at b relocates bytes to the stack. It remains spilled from c to e: $\Psi\,\mathsf{c}\,\mathsf{bytes} = \ldots = \Psi\,\mathsf{e}\,\mathsf{bytes} = (\mathsf{stk}, 0)$. Further, because a is not live after 2 it is not allocated from c to g: $\Psi\,\mathsf{c}\,\mathsf{a} = \ldots = \Psi\,\mathsf{g}\,\mathsf{a} = \bot$. Instead, the register allocation reuses a in f: The fill instruction at e relocates bytes to a: $\Psi\,\mathsf{f}\,\mathsf{bytes} = \mathsf{a}$.

6.2 Register Allocation is not SNiP

Transformations that satisfy the constraints for a valid register allocation are known to be non-interference preserving, provided that the semantics are speculation-free [Barthe, Blazy, Grégoire, et al. 2019; Barthe, Grégoire, Laporte, and Priya 2021]. We first assumed that the same is true for speculative semantics as well, because register allocation only inserts shuffle instructions. While shuffle instructions produce leakages, they are constant address loads and stores which means they only leak constant values. To our surprise, when we tried to form a snippy simulation relation for register allocation, we realized that it was not SNiP. Code 4 demonstrates a minimalistic example for how register allocation introduces weaknesses into $[P]^{\text{ra}}$. The left program is SNi, because the leakage at Line 4 depends on the register `bytes` that holds a public value untouched since the start of the execution. The right program, however, is susceptible to the same attack as described in Section 1 for Code 1: Speculatively executing `c` when `b` holds a value out of |`buf`| can store the secret value from `secret` to `stk[0]`. That value is then loaded at `e` into `a` and leaked at `f`. The fault for this attack is not with the constant leakage of shuffle instructions. The weakness occurs from spilling itself, i.e. from the relocation of registers to memory: Conceptually, speculative execution can access memory everywhere because it performs unsafe memory accesses. But it cannot read or write register contents. Register allocation moves registers into memory, effectively granting unsafe memory operations access to spilled registers.

We were able to reproduce the weakness with a real compiler. Code 4 is inspired by Code 1 which is an excerpt of the `libsodium` function `chacha20_encrypt_bytes` with slight modifications. The function is responsible for encryption of data using the `Chacha20` stream cipher and, similar to Code 1, first copies the data into a stack-local buffer before executing the encryption algorithm on it. Our tests were done on `LLVM 17`, and we tested each of `LLVM`'s register allocators (i.e. `basic`, `greedy`, `fast`, and `pbqp`). They *all* insert the instructions `b` and `e` (Code 4) when compiling our program without other optimizations.[9] This means the vulnerability is inherent to register allocation, and cannot avoided by just opting towards a particular register allocator.

6.3 Poison-Tracking Product

We develop a static analysis that reveals weaknesses like the one in Section 6.2. The analysis operates on a product construction between source program and register-allocated program. To motivate the construction, we inspect the simulation for register allocation under speculation-free semantics, and explain what fails under speculative semantics. Under speculation-free semantics, a simulation for register allocation matches a state s of the source program P to a state t of the target program $[P]^{\text{ra}}$ whenever the instruction is matched, $s\ \Phi\ t$, and the values of all registers and memory locations in s and t are equal up to relocation Ψ. When P is memory safe, one can then prove that any instruction executed from t can be replayed from s. The simulation preserves non-interference: Apart from the leakages inserted by shuffle instructions (which are constant addresses), the leakage that arises from execution in t is equal to the leakage from s. The previous section showed that it is not enough to just extend this approach to speculating states: Speculation introduces unsafe memory operations that access the Ψ-mapped stack locations in $[P]^{\text{ra}}$. Such an access cannot be simulated by the source program P, as it has the accessed value in a register where the memory operation cannot access it. As a result, if we executed P and $[P]^{\text{ra}}$ in parallel we will eventually see speculating states S and T with different values in registers of S and their Ψ-mapped location in T. Further execution propagates the differences to other locations.

The goal in fixing register allocation is to make sure that differences in value do not lead to leakage of sensitive data. An immediate mitigation would be to insert an `sfence` instruction before every

[9] Targeting `x86-64`. We used the `opt` passes `mem2reg`,`simplify-cfg`,`module-inline` before register allocation with `llc`.

load and store operation. This would eliminate speculating unsafe memory accesses altogether. However, this introduces many sfence instructions and reduces performance more than necessary. Instead, we construct a product of P and $[P]^{\mathsf{ra}}$ that tracks differences in values between S and T. We call registers and memory locations that hold different values *poisoned*, because leaking them in $[P]^{\mathsf{ra}}$ might be unsafe: If S and T execute an instruction that leaks the value of a poisoned register, we cannot rely on P being SNi to justify that the leakage from T is safe. Leakage of healthy (not poisoned) registers, however, is safe for that reason. As a result, we only need to protect instructions that leak a poisoned register's value.

Poison types. Poison types represent the registers and memory locations where P and $[P]^{\mathsf{ra}}$ can have different values. Consider states $s \in \mathit{State}_P$ and $t \in \mathit{State}_{[P]^{\mathsf{ra}}}$. A poison type $\mathsf{ps} \in \mathit{Ptype}$ is a function $(\mathit{Reg} \cup \mathit{Mem} \setminus \mathit{Stk}) \to \mathit{Poison}$. It assigns a poison value to each register and memory location of s. The poison values $\mathit{Poison} = \{\mathsf{p}, \mathsf{wp}, \mathsf{h}\}$ have the following meaning: Healthy registers and memory locations (h) are equal between s and t up to relocation by Ψ. Poisoned registers and memory locations (p) can differ between s and t up to relocation by Ψ. Finally, registers and memory locations can be weakly poisoned (wp). This poison value is introduced because of slh a. If slh a occurs as a shuffle instruction, executing it speculatively sets a to 0. But because it is a shuffle instruction it does not occur in P. That makes a's value different between P and $[P]^{\mathsf{ra}}$. However, *every* execution in $[P]^{\mathsf{ra}}$ that executes this shuffle instruction speculatively also sets a to 0. This makes it safe to leak in $[P]^{\mathsf{ra}}$, even though the value differs between P and $[P]^{\mathsf{ra}}$. We thus type registers and memory locations that hold a 0 due to an slh a instruction weakly poisoned (wp). Formally, we express that states $s \in \mathit{State}_P$ and $t \in \mathit{State}_{[P]^{\mathsf{ra}}}$ are equal up to a poison-type ps and relocation Ψ with the notation $s\ \Psi^{\mathsf{ps}}\ t$. We then extend the notation to speculating states $S \in \mathit{SState}_P$ and $T \in \mathit{SState}_{[P]^{\mathsf{ra}}}$ of equal speculation depth $|S| = |T|$. For that, we have sequences of poison types $\mathsf{P} \in \mathit{Ptype}^*$, one poison type for each level of speculation $|S| = |T| = |\mathsf{P}|$. To formalize the notation $S\ \Psi^{\mathsf{P}}\ T$, write $\mathsf{ps}[\mathsf{pv}]$ for the set of all registers and memory locations typed pv by ps, $\mathsf{ps}[\mathsf{pv}] = \{a, (\mathbf{a}, \mathsf{n}) \mid \mathsf{ps}\ a = \mathsf{pv}, \mathsf{ps}\ \mathbf{a}\ \mathsf{n} = \mathsf{pv}\}$. Further, we write $[\![a]\!]_t = \rho_t\ a$ and $[\![(\mathbf{a}, \mathsf{n})]\!]_t = \mu_t\ \mathbf{a}\ \mathsf{n}$. Let $s = (\mathsf{pc}, \rho, \mu) \in \mathit{State}_P$ and $t = (\mathsf{pc}', \rho', \mu') \in \mathit{State}_{[P]^{\mathsf{ra}}}$ range over states of P and $[P]^{\mathsf{ra}}$. We define:

$$
\begin{array}{lll}
 & \forall a \in \mathsf{ps}[\mathsf{h}].\ [\![\Psi\ \mathsf{pc}'\ a]\!]_t = \rho\ a & \\
s\ \Psi^{\mathsf{ps}}\ t & \wedge\quad \forall a \in \mathsf{ps}[\mathsf{wp}].\ [\![\Psi\ \mathsf{pc}'\ a]\!]_t = 0 & S.s\ \Psi^{\mathsf{P},\mathsf{ps}}\ T.t \qquad S\ \Psi^{\mathsf{P}}\ T \\
\quad \textit{if and only if} & \wedge\quad \forall (\mathbf{a}, \mathsf{n}) \in \mathsf{ps}[\mathsf{h}].\ \mu'\ \mathbf{a}\ \mathsf{n} = \mu\ \mathbf{a}\ \mathsf{n} & \quad \textit{if and only if} \quad \wedge\ s\ \Psi^{\mathsf{ps}}\ t \\
 & \wedge\ \forall (\mathbf{a}, \mathsf{n}) \in \mathsf{ps}[\mathsf{wp}].\ \mu'\ \mathbf{a}\ \mathsf{n} = 0 &
\end{array}
$$

The definition of $s\ \Psi^{\mathsf{ps}}\ t$ meets the intuition: Healthy registers and memory locations need to coincide between s and t and weakly poisoned locations must be 0 in t. For poisoned locations, there are no requirements. For speculating states, the definition is applied at each speculation level. We write h, wp, and p for the poison types that assign the respective poison value everywhere.

Example 9. We observe the poisoned values in the attack from Section 6.2 in Code 4 when running P and $[P]^{\mathsf{ra}}$ side by side. The register allocation (Φ, Ψ) is described in Example 8. The initial states are $s = (1, \rho, \mu)$ and $t = (\mathsf{a}, \rho, \mu)$, with $\rho\ \mathsf{b} \notin |\mathsf{buf}|$ and $\rho\ \mathsf{secret} = \mathsf{v} \neq \rho\ \mathsf{bytes}$. We also assume that already $\rho\ \mathsf{a} = \mathsf{f}$ (so that the updates in 1 and a have no effect). After executing the store instruction at 3 with the directive su buf 0, P's state is $u = (4, \rho, \mu[(\mathsf{buf}, 0) \mapsto \mathsf{v}])$. Similarly, after executing the store instruction at d with the directive su stk 0, $[P]^{\mathsf{ra}}$'s state is $v = (\mathsf{e}, \rho, \mu[(\mathsf{stk}, 0) \mapsto \mathsf{v}])$. We see a poisoned value for bytes: At e, it is still located in $\Psi\ \mathsf{e}\ \mathsf{bytes} = (\mathsf{stk}, 0)$ due to the spill at b. But the values differ with $\rho_v\ \mathsf{bytes} \neq \mathsf{v} = \mu'\ \mathsf{stk}\ 0$. Further, $\mu_u\ \mathsf{buf}\ 0 \neq \mu_v\ \mathsf{buf}\ 0$ holds a poisoned value. With a poison type that is fully healthy except on bytes and $(\mathsf{buf}, 0)$, $\mathsf{ps} = \mathsf{h}[\mathsf{bytes}, (\mathsf{buf}, 0) \mapsto \mathsf{p}]$, we have $u\ \Psi^{\mathsf{ps}}\ v$.

Rules 6.1: Stack

POISON-LOAD-STKUNSAFE

$$\frac{P\,s = a := \mathbf{a}[b]_{\rightarrow\mathsf{sc}} \quad \mathsf{ps}\,b = \mathsf{h} \quad \mathsf{ps}' = \mathsf{ps}[a \mapsto \mathsf{p}] \quad s \xrightarrow{\text{LU}\,\mathbf{b}\,\mathsf{l}:\text{LD}\,\mathsf{n}} u \quad t \xrightarrow{\text{LU}\,\mathbf{stk}\,\mathsf{m}:\text{LD}\,\mathsf{n}} v}{(s,t,\mathsf{ps}) \xrightarrow{\text{LU}\,\mathbf{b}\,\mathsf{l}:\text{LD}\,\mathsf{n}::\text{LU}\,\mathbf{stk}\,\mathsf{m}:\text{LD}\,\mathsf{n}} (u,v,\mathsf{ps}')}$$

POISON-STORE-STKUNSAFE

$$\frac{P\,s = \mathbf{a}[b] := c_{\rightarrow\mathsf{sc}} \quad \Psi\,v\,d = (\mathbf{stk},\mathsf{m}) \quad \mathsf{ps}\,b = \mathsf{h} \quad \mathsf{ps}' = \mathsf{ps}[d,(\mathbf{a},\mathsf{l}) \mapsto \mathsf{p}] \quad s \xrightarrow{\text{SU}\,\mathbf{a}\,\mathsf{l}:\text{ST}\,\mathsf{n}} u \quad t \xrightarrow{\text{SU}\,\mathbf{stk}\,\mathsf{m}:\text{ST}\,\mathsf{n}} v}{(s,t,\mathsf{ps}) \xrightarrow{\text{SU}\,\mathbf{a}\,\mathsf{l}:\text{ST}\,\mathsf{n}::\text{SU}\,\mathbf{stk}\,\mathsf{m}:\text{ST}\,\mathsf{n}} (u,v,\mathsf{ps}')}$$

Rules 6.2 Loads

$P\,s = a := \mathbf{a}[b]_{\rightarrow\mathsf{sc}}$

POISON-LOAD-SAFE

$$\frac{\mathsf{ps}\,b = \mathsf{wp} \quad s \xrightarrow{\bullet:\text{LD}\,\mathsf{k}} u \quad t \xrightarrow{\bullet:\text{LD}\,0} v}{(s,t,\mathsf{ps}) \xrightarrow{\bullet:\text{LD}\,\mathsf{k}::\bullet:\text{LD}\,0} (u,v,\mathsf{ps}[a \mapsto \mathsf{p}])}$$

POISON-LOAD-UNSAFE

$$\frac{\mathsf{ps}\,b = \mathsf{wp} \quad s \xrightarrow{\text{LU}\,\mathbf{a}\,\mathsf{l}:\text{LD}\,\mathsf{k}} u \quad t \xrightarrow{\bullet:\text{LD}\,0} v}{(s,t,\mathsf{ps}) \xrightarrow{\text{LU}\,\mathbf{a}\,\mathsf{l}:\text{LD}\,\mathsf{k}::\bullet:\text{LD}\,0} (u,v,\mathsf{ps}[a \mapsto \mathsf{p}])}$$

HEALTHY-LOAD-SAFE

$$\frac{\mathsf{ps}\,b = \mathsf{h} \quad s \xrightarrow{\bullet:\text{LD}\,\mathsf{n}} u \quad t \xrightarrow{\bullet:\text{LD}\,\mathsf{n}} v}{(s,t,\mathsf{ps}) \xrightarrow{\bullet:\text{LD}\,\mathsf{n}::\bullet:\text{LD}\,\mathsf{n}} (u,v,\mathsf{ps}[a \mapsto \mathsf{ps}\,\mathbf{a}\,\mathsf{n}])}$$

HEALTHY-LOAD-UNSAFE

$$\frac{\mathsf{ps}\,b = \mathsf{h} \quad \mathsf{ps}' = \mathsf{ps}[a \mapsto \mathsf{ps}\,\mathbf{b}\,\mathsf{m}] \quad s \xrightarrow{\text{LU}\,\mathbf{b}\,\mathsf{m}:\text{LD}\,\mathsf{n}} u \quad t \xrightarrow{\text{LU}\,\mathbf{b}\,\mathsf{m}:\text{LD}\,\mathsf{n}} v \quad \mathbf{b} \neq \mathbf{stk}}{(s,t,\mathsf{ps}) \xrightarrow{\text{LU}\,\mathbf{b}\,\mathsf{m}:\text{LD}\,\mathsf{n}::\text{LU}\,\mathbf{b}\,\mathsf{m}:\text{LD}\,\mathsf{n}} (u,v,\mathsf{ps}')}$$

Rules 6.3 Stores

$P\,s = \mathbf{a}[b] := c_{\rightarrow\mathsf{sc}}$

POISON-STORE-SAFE

$$\frac{\mathsf{ps}' = \mathsf{ps}[(\mathbf{a},\mathsf{k}),(\mathbf{a},0) \mapsto \mathsf{p}] \quad \mathsf{ps}\,b = \mathsf{wp} \quad s \xrightarrow{\bullet:\text{ST}\,\mathsf{k}} u \quad t \xrightarrow{\bullet:\text{ST}\,0} v}{(s,t,\mathsf{ps}) \xrightarrow{\bullet:\text{ST}\,\mathsf{k}::\bullet:\text{ST}\,0} (u,v,\mathsf{ps}')}$$

POISON-STORE-UNSAFE

$$\frac{\mathsf{ps}\,b = \mathsf{wp} \quad s \xrightarrow{\text{SU}\,\mathbf{a}\,\mathsf{l}:\text{ST}\,\mathsf{k}} u \quad t \xrightarrow{\bullet:\text{ST}\,0} v}{(s,t,\mathsf{ps}) \xrightarrow{\text{SU}\,\mathbf{a}\,\mathsf{l}:\text{ST}\,\mathsf{k}::\bullet:\text{ST}\,0} (u,v,\mathsf{ps}[(\mathbf{a},\mathsf{l}),(\mathbf{a},0) \mapsto \mathsf{ps}\,c])}$$

HEALTHY-STORE-SAFE

$$\frac{\mathsf{ps}' = \mathsf{ps}[(\mathbf{a},\mathsf{n}) \mapsto \mathsf{ps}\,c] \quad \mathsf{ps}\,b = \mathsf{h} \quad s \xrightarrow{\bullet:\text{ST}\,\mathsf{n}} u \quad t \xrightarrow{\bullet:\text{ST}\,\mathsf{n}} v}{(s,t,\mathsf{ps}) \xrightarrow{\bullet:\text{ST}\,\mathsf{n}::\bullet:\text{ST}\,\mathsf{n}} (u,v,\mathsf{ps}')}$$

HEALTHY-STORE-UNSAFE

$$\frac{\mathsf{ps}\,b = \mathsf{h} \quad s \xrightarrow{\text{SU}\,\mathbf{b}\,\mathsf{m}:\text{ST}\,\mathsf{n}} u \quad t \xrightarrow{\text{SU}\,\mathbf{b}\,\mathsf{m}:\text{ST}\,\mathsf{n}} v \quad \mathbf{b} \neq \mathbf{stk}}{(s,t,\mathsf{ps}) \xrightarrow{\text{SU}\,\mathbf{b}\,\mathsf{m}:\text{ST}\,\mathsf{n}::\text{SU}\,\mathbf{b}\,\mathsf{m}:\text{ST}\,\mathsf{n}} (u,v,\mathsf{ps}[(\mathbf{b},\mathsf{m}) \mapsto \mathsf{ps}\,c])}$$

Rules 6.4: Shuffles

POISON-FILL

$$\frac{P\,t = a' := \mathbf{stk}[\mathsf{l}]_{\rightarrow\mathsf{sc}} \quad t \xrightarrow{\bullet:\text{LD}\,\mathsf{l}} v}{(s,t,\mathsf{ps}) \xrightarrow{\varepsilon:\varepsilon::\bullet:\text{LD}\,\mathsf{l}} (s,v,\mathsf{ps})}$$

POISON-SPILL

$$\frac{P\,t = \mathbf{stk}[\mathsf{l}] := b'_{\rightarrow\mathsf{sc}} \quad t \xrightarrow{\bullet:\text{ST}\,\mathsf{l}} v}{(s,t,\mathsf{ps}) \xrightarrow{\varepsilon:\varepsilon::\bullet:\text{ST}\,\mathsf{l}} (s,v,\mathsf{ps})}$$

POISON-MOVE

$$\frac{P\,t = a' := b'_{\rightarrow\mathsf{sc}} \quad t \xrightarrow{\bullet:\bullet} v}{(s,t,\mathsf{ps}) \xrightarrow{\varepsilon:\varepsilon::\bullet:\bullet} (s,v,\mathsf{ps})}$$

POISON-SHUFFLE-SFENCE

$$\frac{P\,t = \mathsf{sfence}_{\rightarrow\mathsf{sc}} \quad t \xrightarrow{\bullet:\bullet} v}{(s,t,\mathsf{ps}) \dashrightarrow (s,v,\mathsf{ps})}$$

POISON-SHUFFLE-SLH

$$\frac{P\,T = \mathsf{slh}\,a'_{\rightarrow\mathsf{sc}} \quad \Psi\,T\,a = a' \quad T \xrightarrow{\bullet:\bullet} V \quad \mathsf{pv} = |\mathsf{P}| = 0\ ?\ \mathsf{ps}\,a : \mathsf{wp}}{(S,T,\mathsf{P.ps}) \dashrightarrow (S,V,\mathsf{P.ps}[a \mapsto \mathsf{pv}])}$$

Rules 6.5: Speculation Sensitive

POISON-STEP

$$\frac{(s,t,\mathsf{ps}) \xrightarrow{\delta:\lambda::\gamma:\kappa} (u,v,\mathsf{ps}')}{(S.s,T.t,\mathsf{P.ps}) \dashrightarrow (S.u,T.v,\mathsf{P.ps}')}$$

POISON-ROLLBACK

$$\frac{|S| = |T| = |\mathsf{P}| \geq 1}{(S.s,T.t,\mathsf{P.ps}) \dashrightarrow (S,T,\mathsf{P})}$$

HEALTHY-SPEC

$$\frac{P\,S = \mathsf{br}\,b_{\rightarrow\mathsf{sc_t},\mathsf{sc_f}} \quad \mathsf{ps}\,b = \mathsf{h} \quad S \xrightarrow{\text{SP:BR}\,\mathsf{b}} S.u \quad T \xrightarrow{\text{SP:BR}\,\mathsf{b}} T.v}{(S,T,\mathsf{P.ps}) \dashrightarrow (S.u,T.v,\mathsf{P.ps.ps})}$$

POISON-SFENCE

$$\frac{P\,s = \mathsf{sfence}_{\rightarrow\mathsf{sc}} \quad s \xrightarrow{\bullet:\bullet} u \quad t \xrightarrow{\bullet:\bullet} v}{(s,t,\mathsf{ps}) \dashrightarrow (u,v,\mathsf{ps})}$$

POISON-SLH

$$\frac{P\,S = \mathsf{slh}\,a_{\rightarrow\mathsf{sc}} \quad S \dashrightarrow U \quad T \dashrightarrow V \quad \mathsf{pv} = |\mathsf{P}| = 0\ ?\ \mathsf{ps}\,a : \mathsf{h}}{(S,T,\mathsf{P.ps}) \dashrightarrow (U,V,\mathsf{P.ps}[a \mapsto \mathsf{pv}])}$$

Poison product. The poison product $P :: [P]^{\mathsf{ra}}$ tracks how poison types are updated when executing P and $[P]^{\mathsf{ra}}$ side by side. We design the side by side execution so that transitions in $[P]^{\mathsf{ra}}$ are replayed by P. We later want to craft a simulation from the product, so every transition of $[P]^{\mathsf{ra}}$ must be included in $P :: [P]^{\mathsf{ra}}$. However, when a transition of $[P]^{\mathsf{ra}}$ can be replayed by multiple transitions in P, which happens when P performs an unsafe memory access, we choose a memory location for the access so that the transition poisons as few registers and memory locations as possible. The states of $P :: [P]^{\mathsf{ra}}$ take the shape (S, T, P) with $S \in \mathit{SState}_P$, $T \in \mathit{SState}_{[P]^{\mathsf{ra}}}$, $\mathsf{P} \in \mathit{Ptype}^*$, and $S\ \Psi^{\mathsf{P}}\ T$ (which already implies $|S| = |T| = |\mathsf{P}|$). There are two types of states: Instruction-matched states and shuffling states. Instruction-matched states satisfy $S\ \Phi\ T$. Shuffling states have T at a shuffle sequence after which the states would be instruction-matched again. That is, $S = S'.s$ and $T = T'.t$, so that $[P]^{\mathsf{ra}}\ t = \mathit{sh}_{\mathsf{sc}}$ with $\Phi\, s = \mathsf{sc}$. Each type of state has transitions in $P :: [P]^{\mathsf{ra}}$:

$$(S, T, \mathsf{P}) \xrightarrow{\gamma:\kappa :: \delta:\lambda} (U, V, \mathsf{Q}) \qquad\qquad (S, T, \mathsf{P}) \xrightarrow{\varepsilon:\varepsilon :: \delta:\lambda} (S, V, \mathsf{Q})$$

The first type of transition is enabled only for instruction-matched $S\ \Phi\ T$. It executes a transition simultaneously in both programs via $S \xrightarrow{\gamma:\kappa} U$ and $T \xrightarrow{\delta:\lambda} V$. The second type is enabled only for shuffling states. In that case, S stutters while T progresses through the shuffle sequence. In both cases, the transition needs to update the poison values depending on the transition rules taken in P and $[P]^{\mathsf{ra}}$. We present how $P :: [P]^{\mathsf{ra}}$ updates poison values in Rules 6.1 to 6.5. We mirror the approach in the semantics and provide the updates in two steps. In a first step, we define the updates for instruction-matched, speculation-free states in Rules 6.1 to 6.3. They represent the transitions for the speculation-free semantics (Rules 2.1). In the second step we lift the speculation-free updates with Rules 6.5 to define the transitions of the product on the speculative semantics (Rules 2.2). Rules 6.4 defines the transitions for shuffling states. The instruction-matched transitions for `nop`, $a := b \oplus c$ and `br` b can be found in the appendix. We explain the transitions in detail.

The transitions POISON-LOAD-STKUNSAFE and POISON-STORE-STKUNSAFE are the source of poison values. They represent the situation from above, where we argued that values between P and $[P]^{\mathsf{ra}}$ can differ: In POISON-LOAD-STKUNSAFE a target program's speculating unsafe load (recognizable from the directive LU **stk** m) loads from a spilled register's stack location. The source program cannot perform that load and thus has to perform any other unsafe load. This leads to different loaded values and poisons the register loaded to. While we could choose the location of the unsafe access in P, the register is poisoned in either case, so we allow arbitrary unsafe loads in P. Similarly, in POISON-STORE-STKUNSAFE a speculating unsafe store poisons the source program's register d by overwriting it on the stack. Again, the source program cannot replay that store as d is a register and needs to store somewhere else. In this case, we always let the source program store to **a** instead, because this reduces the number of poisoned memory locations for static analysis. The overwritten stack-allocated register d and the memory location in **a** are both poisoned. Notice how both rules require the source program to access memory unsafely, as well. This is because the register holding the offset address must be healthy, $\mathrm{ps}\, b = \mathsf{h}$, and thus has the same value in s and t. The address must be healthy, because a poisonous address could be unsafe to leak. A weakly poisoned address, on the other hand, would be 0 in $[P]^{\mathsf{ra}}$ which is always a safe access. This is because we chose `slh` a to wipe a register to 0. Any other constant would work the same way, but the number of rules in the product would increase.

Rules 6.2 and 6.3 for instruction-matched transitions just propagate already poisoned registers and memory locations. For example, HEALTHY-LOAD-SAFE considers the case where the addressing register is healthy. That means that both source and target program load from the same memory location. We can recognize from the premise that the memory access is safe: Source and target program execute the load with the directive •, so they execute with LOAD. The rule then propagates the poison value from the memory location loaded from to the register loaded to. As before,

the product transition can only be taken when the leaked content is not poisoned as that could represent an unsafe leakage. The case of a weakly poisoned address is separately handled by POISON-LOAD-SAFE and POISON-LOAD-UNSAFE. Rules 6.3 propagate poison values for stores.

Rules 6.5 provides the poison updates for instruction-matched states that execute speculation sensitive instructions. Rule POISON-STEP lifts the speculation-free transitions to the speculative setting. HEALTHY-SPEC forbids weakly poisoned branching conditions even though they are safe to leak. This is to avoid that P and $[P]^{\mathsf{ra}}$ arrive at different program points (up to Φ).[10] POISON-SLH creates healthy values when the product is speculating, because source and target program both wipe the register to 0. If not speculating, the poison values are untouched.

Rules 6.4 provides the second type of transitions, for shuffling states, in $P :: [P]^{\mathsf{ra}}$. The speculation insensitive shuffle instructions are again brought to speculating states with POISON-STEP. The insensitive shuffle instructions only move values and don't modify them. There is no need to update the poison type because the relocation is already included in Ψ with shuffle conformity. Thus, the poison type is untouched. Only POISON-SHUFFLE-SLH can be a new source of poisoned values. As discussed earlier, when speculatively executed, it wipes the register's value to 0, which makes it weakly poisoned. On speculation-free states, `sfence` and `slh` a' do not update the poison type.

The transitions of $P :: [P]^{\mathsf{ra}}$ are well-defined in the sense that executing a transition leads to a state that again satisfies the constraints of being a state (Lemma 4). Also, the speculation-free states never become poisoned, because POISON-LOAD-STKUNSAFE and POISON-STORE-STKUNSAFE cannot happen: P is memory safe under speculation-free semantics. Further, the only other rule to introduce poison values is POISON-SHUFFLE-SLH, but it introduces no poison value in speculation-free states.

LEMMA 4. *Given* (S, T, P) *and a transition* $(S, T, \mathsf{P}) \xrightarrow{Y:\kappa::\delta:\lambda} (U, V, \mathsf{Q})$, *then* $U\ \Psi^{\mathsf{Q}}\ V$.

LEMMA 5. *Speculation-free states are never poisoned. I.e.* $(s, t, \mathsf{h}) \xrightarrow{e:k::d:l}{}^{*} (u, v, \mathsf{ps})$ *implies* $\mathsf{ps} = \mathsf{h}$.

Example 10. Consider the execution of $P :: [P]^{\mathsf{ra}}$ in Figure 3. It depicts the side by side execution from Example 9 in $P :: [P]^{\mathsf{ra}}$. Each state of $P :: [P]^{\mathsf{ra}}$ is depicted as a small table listing the program counters of P and $[P]^{\mathsf{ra}}$, the values of the registers, and the values of relevant memory locations (we use t, f for Boolean typed values). The right column contains the poison type associated with the pair of states. The first transition simultaneously executes the instruction-matched assignment at `1` and `a`. The second transition executes the shuffle instruction that spills `bytes` to `stk`[0] at `b`, while P waits. The third transition speculates. Notice that HEALTHY-SPEC only admits this transition because `a` is h. The next transition is the unsafe store to `stk` in $[P]^{\mathsf{ra}}$, overwriting the spilled `bytes`. Notice that POISON-STORE-STKUNSAFE has changed the poison value for `bytes` to p, because the memory location written to was Ψ `e bytes` = (`stk`, 0). The rule also poisons the memory location where P writes to. The last transition is the shuffle instruction that relocates the poisoned `bytes` to `a`. The product is now stuck: HEALTHY-SPEC is disabled because the poison value for `bytes` is p and thus not safe to leak in $[P]^{\mathsf{ra}}$. Indeed, Ψ `f bytes` = `a` holds the secret value 42.

6.4 Static Poison Analysis of $P :: [P]^{\mathsf{ra}}$

The poison product finds weaknesses: Whenever a poisoned register's value would be leaked in $[P]^{\mathsf{ra}}$, the product cannot execute the transition. To find the program points where a poisoned register can be leaked, we design a static analysis that over-approximates the poison values any execution could produce. The analysis constructs, for each program point pair $(\mathsf{pc}, \mathsf{pc}')$, an approximate poison type ps. This poison type is approximate in that a statically h-typed register is always healthy in any execution that reaches $(\mathsf{pc}, \mathsf{pc}')$ in a speculating state. Similarly, if it is statically typed wp it is

[10] Branching on weakly poisoned registers could be supported but further complicates the product definition.

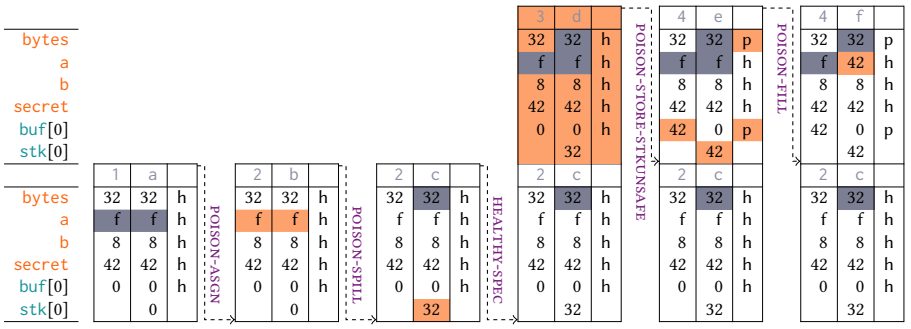

Fig. 3. An execution of $P :: [P]^{\mathsf{ra}}$ on Code 4. Updated values are highlighted. Dead registers are gray.

always weakly poisoned in a speculating state. However, if it is statically typed p it might also be healthy or weakly poisoned. Formally, the analysis constructs a function $\mathcal{X} : PC_{P::[P]^{\mathsf{ra}}} \to \mathit{Ptype}$, where $PC_{P::[P]^{\mathsf{ra}}}$ are the program points of $P :: [P]^{\mathsf{ra}}$,

$$\begin{aligned} PC_{P::[P]^{\mathsf{ra}}} &\triangleq \{(\mathsf{pc}, \mathsf{pc}') \mid \exists (S.s, T.t, \mathsf{P.ps}).\ \mathsf{pc} \equiv s \wedge \mathsf{pc}' \equiv t\} \\ &= \{(\mathsf{pc}, \mathsf{pc}'), (\mathsf{pc}, \mathsf{r}') \mid \Phi\, \mathsf{pc} = \mathsf{pc}' \wedge [P]^{\mathsf{ra}}\, \mathsf{r}' = \mathit{sh}_{\rightarrow \mathsf{pc}'}\}\,. \end{aligned}$$

The pairs $(\mathsf{pc}, \mathsf{pc}')$ are instruction-matched program counters and $(\mathsf{pc}, \mathsf{r}')$ are from shuffling states.

We design our analysis as a forward flow analysis (Equation (fwd)). For that, we need to define a flow lattice L, transfer functions, and the initial flow value $\mathcal{X}_0$. The flow analysis then yields $\mathcal{X}$ as a solution. The lattice is constructed on *Ptype*. We create an ordering $\mathsf{h} < \mathsf{p}$ and $\mathsf{wp} < \mathsf{p}$ on *Poison*. In order to arrive at a lattice, we further extend it by an artificial least element $\bot \in \mathit{Poison}$.[11] We then lift the ordering point-wise to poison types for the flow lattice $L = (\mathit{Ptype}, \leq)$. The ordering is chosen with the intention that when $\mathsf{ps} \leq \mathsf{ps}'$, then $s\ \Psi^{\mathsf{ps}}\ t$ implies $s\ \Psi^{\mathsf{ps}'}\ t$ (which would not be the case if we had set $\mathsf{h} < \mathsf{wp}$). The initial value is set to healthy, $\mathcal{X}_0 = \mathsf{h}$, because the initial states of P and $[P]^{\mathsf{ra}}$ are fully equal up to Ψ. The transfer functions $F_{(\mathsf{pc},\mathsf{pc}')} : \mathit{Ptype} \to \mathit{Ptype}$ need to approximate the poison types in $P :: [P]^{\mathsf{ra}}$. Conceptually, they update the current poison type by simultaneously executing all updates that $P :: [P]^{\mathsf{ra}}$ could do. For instruction-matched $\Phi\, \mathsf{pc} = \mathsf{pc}'$ that means to look at the instruction $i = P\, \mathsf{pc}$ which is the same as $[P]^{\mathsf{ra}}\, \mathsf{pc}'$ up to Ψ. Then, we poison all registers and memory locations that a rule for i from Rules 6.1 to 6.3 and 6.5 could poison. This means the transfer function is solely dependent on the instruction $i = P\, \mathsf{pc}$, and we define it via $F_{(\mathsf{pc},\mathsf{pc}')} = F_i$ below. For a shuffling product state, $P :: [P]^{\mathsf{ra}}$ offers for each instruction $i' = [P]^{\mathsf{ra}}\, \mathsf{r}'$ only one update which we find in Rules 6.4. We again define transfer solely dependent on i' via $F_{(\mathsf{pc},\mathsf{r}')} = G_{i'}$. For a shuffling $i' = \mathtt{slh}\ a'$ we assume that the source register for a' is a, i.e. $\Psi\, \mathsf{r}'\, a = a'$. We only present the interesting cases of F_i and $G_{i'}$. The initial poison type for $(\mathsf{init}, \mathsf{init}')$ is healthy.

$$\begin{aligned} F_{a \coloneqq \mathbf{a}[b]_{,\mathsf{sc}}}\, \mathsf{ps} &= \mathsf{ps}[a \mapsto \mathsf{p}] \\ F_{\mathtt{slh}\, a_{,\mathsf{sc}}}\, \mathsf{ps} &= \mathsf{ps}[a \mapsto \mathsf{h}] \\ G_{\mathtt{slh}\, a'_{,\mathsf{sc}}}\, \mathsf{ps} &= \mathsf{ps}[a \mapsto \mathsf{wp}] \\ G_{\mathtt{sfence}_{,\mathsf{sc}}}\, \mathsf{ps} &= \mathsf{h} \end{aligned} \qquad F_{a \coloneqq b \oplus c_{,\mathsf{sc}}}\, \mathsf{ps} = \begin{cases} \mathsf{ps}[a \mapsto \mathsf{h}] & \mathsf{ps}\, b = \mathsf{ps}\, c = \mathsf{h} \\ \mathsf{ps}[a \mapsto \mathsf{p}] & \text{otherwise} \end{cases}$$

$$F_{\mathbf{a}[b] \coloneqq c_{,\mathsf{sc}}}\, \mathsf{ps} = \mathsf{ps}[\mathit{Mem} \overset{\sqcup}{\mapsto} \mathsf{ps}\, c][\mathit{Reg}, \mathbf{a} \mapsto \mathsf{p}]$$

All transfer functions are monotonic. All but one transfer function are easily defined in order to approximate the rules of $P :: [P]^{\mathsf{ra}}$. The exception is for $\mathbf{a}[b] \coloneqq c$ instructions which we explain. The first substitution $\mathsf{ps}[\mathit{Mem} \overset{\sqcup}{\mapsto} \mathsf{ps}\, c]$ sets all memory locations $(\mathbf{b}, \mathsf{n})$ to $\mathsf{ps}\, (\mathbf{b}, \mathsf{n}) \sqcup \mathsf{ps}\, c$. This approximates HEALTHY-LOAD-SAFE and HEALTHY-LOAD-UNSAFE, because both P and $[P]^{\mathsf{ra}}$ store to

[11] This is a standard construction. In the remainder, we assume that transfer functions preserve $\bot$.

the same location which will have poison value ps c. All other locations maintain their poison value. To approximate this behavior statically, we take the join on the two poison values. The second substitution sets all of **a** and all registers *Reg* to p.[12] This approximates POISON-STORE-STKUNSAFE: *Reg* needs to be poisoned because a store to the stack overwrites the contents of a register in the source program. At the same time, the rule overwrites **a** in the source program P, which leads to **a** being poisoned as well. Additionally, POISON-STORE-SAFE and POISON-STORE-UNSAFE write to **a** only, meaning the poisoning of **a** already approximates them as well.

In order to formally express how a solution $\mathcal{X}$ approximates the poison values of all reachable states in $P :: [P]^{\mathsf{ra}}$, we introduce shorthand notations for sequences of poison types fully consisting of poison types from $\mathcal{X}$. Fix a solution $\mathcal{X}$ to the flow equations. We define:

$$\mathcal{X}_{s,t} = \mathsf{h} \qquad \mathcal{X}_{S.s,T.t} = \mathcal{X}_{S,T}\mathcal{X}\,(s,t)$$

$$S\ \Psi_{\mathcal{X}}\ T \Leftrightarrow S\ \Psi^{\mathcal{X}_{S,T}}\ T \qquad S.s\ \Psi^{\mathsf{ps}}_{\mathcal{X}}\ T.t \Leftrightarrow S.s\ \Psi^{\mathcal{X}_{S,T}.\mathsf{ps}}\ T.t \qquad (S.s, T.t, \mathsf{ps}) = (S.s, T.t, \mathcal{X}_{S,T}.\mathsf{ps})$$

The notation $\mathcal{X}_{S,T}$, where S, T are speculating states, stands for a sequence of poison types of length $|\mathcal{X}_{S,T}| = |S| = |T|$. Intuitively, $\mathcal{X}_{S,T}$ consists of the poison types of $\mathcal{X}$, applied to the program counters of S and T at every level. The only exception is the lowest level in the speculating states. They are always set to h, because we know from Lemma 5 that those states are reachable speculation-free and can never have poisoned registers or memory locations. The notation $S\ \Psi_{\mathcal{X}}\ T$, $S\ \Psi^{\mathsf{ps}}_{\mathcal{X}}\ T$, and $(S.s, T.t, \mathsf{ps})$ are shorthand notations to avoid a lot of repeating $\mathcal{X}_{S,T}$.

Lemma 6. *Whenever* $(S, T, \mathcal{X}_{S,T}) \xrightarrow{Y:\kappa::\delta:\lambda}{}^{*} (U, V, \mathrm{Q})$ *then* $\mathrm{Q} \leq \mathcal{X}_{U,V}$.

6.5 Fixing Register Allocation

We can use our static analysis solution $\mathcal{X}$ to identify whether $[P]^{\mathsf{ra}}$ has weaknesses: If $\mathcal{X}$ guarantees that leakages are never poisonous, then $[P]^{\mathsf{ra}}$ has no register allocation induced weaknesses.

Definition 8. *A register allocation* (Φ, Ψ) *between* P *and* $[P]^{\mathsf{ra}}$ *is poison-typable, if there is a solution* $\mathcal{X}$ *to* (fwd) *which for every* $(\mathsf{pc}, \mathsf{pc}')$ *with* $\Phi\,\mathsf{pc} = \mathsf{pc}'$ *satisfies the additional constraints*

$$\mathsf{ps}\,b = \mathsf{wp} \vee \mathsf{ps}\,b = \mathsf{h} \qquad \text{if} \qquad P\,\mathsf{pc} = a := \mathbf{a}[b]_{\rangle \mathsf{sc}} \vee P\,\mathsf{pc} = \mathbf{a}[b] := c_{\rangle \mathsf{sc}}\ ,$$

$$\mathsf{ps}\,b = \mathsf{h} \qquad \text{if} \qquad P\,\mathsf{pc} = \mathtt{br}\ b_{\rangle \mathsf{sc}_\mathsf{t}, \mathsf{sc}_\mathsf{f}}\ .$$

Our fix is applicable to any register allocation (Φ, Ψ) between P and $[P]^{\mathsf{ra}}$: We check if (Φ, Ψ) is poison-typable. For that, we solve the flow analysis (e.g. [Kildall 1973]) to obtain a static poison assignment $\mathcal{X}$ and check the additional constraints. If it is not poison-typable, then an additional constraint is violated for some program point $(\mathsf{pc}, \mathsf{pc}')$ and register b' of $P :: [P]^{\mathsf{ra}}$. We insert an `sfence` or `slh` b' instruction into $[P]^{\mathsf{ra}}$ at the end of the shuffle sequence right before pc' and obtain a new register allocation where that additional constraint is now satisfied. We then repeat the process until we obtain a poison-typable register allocation.

6.6 Poison-typable Register Allocation is SNiP

We now prove that making register allocation poison-typable already makes it SNiP. With our proof method from Theorem 2, this reduces to crafting a snippy simulation. Again, the crafted simulation is parametric, so that it works for all poison-typable register allocations.

Theorem 11. *If* (Φ, Ψ) *is a poison-typeable register allocation between* P *and* $[P]^{\mathsf{ra}}$*, then there exists a snippy simulation* $(\prec, \lhd)$ *between* $[P]^{\mathsf{ra}}$ *and* P.

For the remainder of the section, fix a poison-typable register allocation (Ψ, Φ) between P and $[P]^{\mathsf{ra}}$, and the static poison assignment $\mathcal{X}$. Further, let init be the entry point for P and init′ for $[P]^{\mathsf{ra}}$.

[12] We could be more precise and poison only those registers spilled at the current program point.

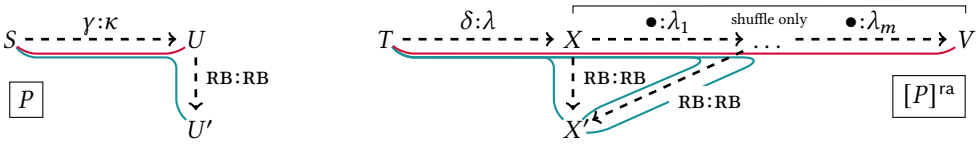

Fig. 4. The shape of Φ-intervals. Teal and red paths form separate intervals.

Defining $(\prec, \triangleleft)$. We define $\prec \subseteq SState_{[P]^{\text{ra}}} \times SState_P$ similar to the simulation for register allocation without speculative semantics: Source state S and target state T are instruction-matched and coincide in values up to relocation by Ψ. The difference is that the states do not need to coincide on registers and memory locations poisoned by $\mathcal{X}$:

$$T \prec S \quad \textit{if and only if} \quad S\ \Phi\ T \wedge S\ \Psi_{\mathcal{X}}\ T\,.$$

For the directive transformation $\triangleleft_{T \prec S}$, we rely on $P :: [P]^{\text{ra}}$'s transitions. We say that a transition sequence $(S, T, \text{ps}) \xrightarrow{e:k::d:l}{}^{*} (U, V, \text{ps}')$ is a Φ-interval if $S\ \Phi\ T$, $U\ \Phi\ V$, and no intermediary state pairs are instruction matched. The intent is that Φ-intervals are precisely the simulation intervals once we have proven that our defined relation is a simulation. Figure 4 depicts the shape of Φ-intervals: The product first executes the instruction-matched instruction on both S and T. Then, the target program can perform any number of shuffle steps, until either the shuffle sequence is fully executed (the pair of red paths), or a rollback happened before that (any pair of teal paths). In that case, $P :: [P]^{\text{ra}}$ rolls back on U as well. We use the Φ-intervals as directive transformations,

$$\triangleleft_{T \prec S} \quad \triangleq \quad \{(e, d) \mid (S, T, \text{ps}) \xrightarrow{e:k::d:l}{}^{*} (U, V, \text{ps}') \text{ is a } \Phi\text{-interval}\}\,.$$

Lemma 7. $(\prec, \triangleleft)$ *is a snippy simulation between* $[P]^{\text{ra}}$ *and* P.

7 Related Work

We already discussed the closely related work in the context of compiler correctness. Here, we give a broader picture and elaborate on methods for proving non-interference for single programs. Note the difference: when reasoning about compiler passes, we reason over all programs. For a broader overview, we defer the reader to a recent survey [Cauligi, Disselkoen, Moghimi, et al. 2022].

Speculation Sources. Since its discovery in 2018, Spectre attacks have been rediscovered in multiple variants. The main difference between the variants lies in the hardware feature that is trained in order to trigger a misspeculation. We call the respective feature the source of speculation. The first version of the attack trains the Prediction History Table (PHT) of the processor, in order to inflict a mispredicted branching speculation [Kocher et al. 2019]. Other variants train the Branch Target Buffer (BTB) to mispredict indirect branching instructions [Kocher et al. 2019], the Return Stack Buffer (RSB) to mispredict return points [Koruyeh et al. 2018], both of which highjack the speculative control flow to execute leaking gadgets speculatively. They can be mitigated in software with a `retpoline` gadget [Turner 2018]. The Speculative Store Bypass (SSB) mechanism, also called Store-To-Load Forwarding (STL), reads from memory even though pending stores have unresolved addresses [Horn 2018], and the Predictive Store Forwarding (PSF) mechanism forwards pending stores to loads with unresolved address [Guanciale et al. 2020]. These mechanisms speculatively load values that either should have been overwritten in the meantime, or should never arrive in memory at the loaded address, creating further potential for unwanted information-flow. Memory speculation sources can be disabled in hardware with mediocre performance penalty. The Meltdown attack introduces speculation through an out-of-order read from elevated-permission memory regions, racing against the MMU to detect the violation before the read memory can be leaked [Lipp

Table 3. Tools that check a program against speculative non-interference: Pitchfork [Cauligi, Disselkoen, Gleissenthall, et al. 2020], Spectector [Guarnieri, Köpf, Morales, et al. 2020], RelSE [Daniel et al. 2021], Blade [Vassena et al. 2021], Jasmin SCT [Barthe, Cauligi, et al. 2021], Typing V1 [Shivakumar et al. 2023].

	Pitchfork	Spectector	RelSE	Blade	Jasmin SCT	Typing V1
Source	PHT, SSB	PHT	PHT, SSB	PHT	PHT	PHT
Property	TS	NI	NI	TS	TS	TS
Method	EX	SE	SE	SA	SA	SA
Speculation	SW	SW	SW / SB	US	US	US
Directives	Y	Y (Oracle)	N	Y	Y	Y
Memory Safety	U	U	U	U	SS	S
Bug / Proof	Bug	Bug	Bug	Proof	Proof	Proof

et al. 2018]. While the speculation sources are varying, all attack variants leak the secrets through the side-channels covered by the constant-time leakage model [Guarnieri, Köpf, Reineke, et al. 2021]. In this paper, we consider only the PHT speculation source. We believe that our notion of simulation also holds for other speculation sources, but we are less sure about what compiler passes satisfy SNiP when they are considered.

Properties. The constant time programming guideline requires no influence of sensitive data towards the leakage observable by the attacker. Formally, *non-interference* [Goguen and Meseguer 1982] (Definition 1) expresses this property. Non-interference is a hyper-property [Clarkson and Schneider 2010]: It requires to reason about two executions of the program. Hyper-properties tend to be harder to verify than single trace properties due to synchronization issues with the traces compared. Non-interference for side-channel leakage evades these issues: The constant time leakage model exposes the program counter to the attacker. Thus, when two traces from attacker-indistinguishable initial states do not coincide in their control flow, the program can immediately be rejected as insecure. Such a high degree of synchronization allows for a sound approximation we call taint safety [A. C. Myers 1999; Sabelfeld and A. Myers 2003].

Tools. Existing tools check single implementations for non-interference. Our work on compiler transformations complements this line of work. It is now possible to check source programs for non-interference, and rely on the compiler that guarantees to preserve non-interference to the executed program for *any source program*. The tools deal with two main challenges: Non-determinism from speculation and the two executions required for non-interference. Table 3 presents a list of tools and classifies their approaches. The **Source** lists the considered speculation sources. **Property** is the formal property checked: NI is non-interference under speculative semantics. TS is taint safety. **Method** is either SA for static analysis such as flow- or type-systems, SE for symbolic execution, or EX for state space exploration. **Speculation** lists how the tool models and copes with speculation based non-determinism. SW means that the semantics have a bounded speculation window, i.e. a bound to the number of speculatively executed instructions. SB means that the semantics have a bounded store buffer that limits the number of speculatively executed store instructions. US means unbounded speculation, i.e. the semantics speculate arbitrarily long. **Mem Safety** describes the memory model and memory requirements on the source program. U stands for unstructured memory, S for structured memory and memory safety under speculation-free semantics, and SS is memory safety even under speculative semantics.

Jasmin SCT (Table 3) comes along with the Jasmin compiler (Table 1), that is proven to preserve Ni under speculation-free semantics. In [Barthe, Cauligi, et al. 2021], the authors suggest an additional

requirement to cope with speculative semantics: Source programs need to be memory safe even when executed under speculative semantics (SS). However, there is no proof that the compiler preserves non-interference under this requirement. SS is also a performance breaking requirement: They report that SS-implementations are ~20% slower than the insecure reference implementations; in contrast to ~1% overhead reported for recently protected implementations without SS.

Similarities and differences to a simultaneously developed proof technique. Another proof method for preserving speculative side-channel security through compilers has been developed independently of our work [Arranz Olmos et al. 2025].[13] There are subtle differences in the definition of semantics, security property, and simulation: First, their semantics comes without rollbacks. The idea is that a difference in leakage can always be obtained by an execution that contains no rollback. This result has been obtained previously [Barthe, Cauligi, et al. 2021]. Second, their security property, SCT, is the same as SNi from this work. However, the authors employ a slightly modified semantics which allows them to phrase the property differently. Finally, our simulation constraint is a constant-time cube (Figure 2). The constraint formulated by Arranz Olmos et al. [2025] instead requires the existence of two functions: One back-translates directives similar to how our simulation finds a sequence of source directives to replay the target directives. The other forward-translates leakage from source leakage to target leakage. The existence of both functions already guarantees that our constant-time cube is satisfied. The bigger difference is in the application of our proof methods. Our work spots a weakness in register allocation, and we develop a static analysis to protect against security-threatening spills. Arranz Olmos et al. [2025] target the Jasmin compiler with their work. Jasmin's source language already requires the programmer to tag variables as register or stack variable. Therefore, the Jasmin compiler performs no spilling, avoiding the weakness by demanding the programmer to appropriately choose register-allocated variables. Their work instead focuses on proving nine other passes secure by extending preservation proofs from leakage semantics to speculative semantics.

8 Conclusion

We have developed a method for proving that compiler transformations preserve non-interference from source to target programs *under speculative semantics*. When experimenting with our method, we found that it worked well on simple transformations like dead code elimination, but we had trouble applying it to register allocation. As it turned out, the fault was not on our side but register allocation is actually insecure. Our method led us to discover a new vulnerability introduced by register allocation. We have confirmed the existence of this vulnerability in the mainstream compiler `LLVM` on code from `libsodium`, a modern cryptographic library. Interestingly, our proof method also guided us towards a fix: We have presented a new static analysis to identify weaknesses introduced by register allocation, and an automated procedure to fix them. With these additions, we have been able to prove that register allocation preserves non-interference.

As future work, we would like to integrate our proof method with certified compilers, investigate transformations that we left out so far, and consider further sources of speculation in the semantics.

References

John McCarthy and James Painter. 1967. "Correctness of a Compiler for Arithmetic Expressions." *Proc. Sympos. Appl. Math., Vol. XIX*. Amer. Math. Soc., Providence, RI, 33–41.

Robin Milner. 1971. "An Algebraic Definition of Simulation between Programs." *IJCAI*. William Kaufmann, 481–489.

Gary A. Kildall. 1973. "A Unified Approach to Global Program Optimization." *POPL*. ACM, 194–206. https://doi.org/10.1145/512927.512945.

[13] To be appearing in the same conference.

Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. 1981. "Register Allocation via Coloring." *Computer Languages*, 47–57. https://doi.org/10.1016/0096-0551(81)90048-5.

Joseph A. Goguen and José Meseguer. 1982. "Security Policies and Security Models." *S&P*, 11–11. https://doi.org/10.1109/SP.1982.10014.

Omri Traub, Glenn Holloway, and Michael D. Smith. 1998. "Quality and Speed in Linear-Scan Register Allocation." *PLDI*. ACM, 142–151. https://doi.org/10.1145/277650.277714.

Andrew C. Myers. 1999. "JFlow: Practical Mostly-Static Information Flow Control." *POPL*. ACM, 228–241. https://doi.org/10.1145/292540.292561.

Massimiliano Poletto and Vivek Sarkar. 1999. "Linear Scan Register Allocation." *ACM Trans. Program. Lang. Syst.*, 895–913. https://doi.org/10.1145/330249.330250.

A. Sabelfeld and A.C. Myers. 2003. "Language-Based Information-Flow Security." *IEEE Journal on Selected Areas in Communications*, 5–19. https://doi.org/10.1109/JSAC.2002.806121.

David Brumley and Dan Boneh. 2005. "Remote Timing Attacks Are Practical." *Computer Networks*, 701–716. https://doi.org/10.1016/j.comnet.2005.01.010.

David Molnar, Matt Piotrowski, David Schultz, and David Wagner. 2006. "The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks." *ICISC*. Springer, 156–168. https://doi.org/10.1007/11734727_14.

Xavier Leroy. 2009. "A Formally Verified Compiler Back-end." *Journal of Automated Reasoning*, 363–446. https://doi.org/10.1007/s10817-009-9155-4.

Michael R. Clarkson and Fred B. Schneider. 2010. "Hyperproperties." *Journal of Computer Security*, 1157–1210. https://doi.org/10.3233/JCS-2009-0393.

Nadhem J. Al Fardan and Kenneth G. Paterson. 2013. "Lucky Thirteen: Breaking the TLS and DTLS Record Protocols." *S&P*, 526–540. https://doi.org/10.1109/SP.2013.42.

Gilles Barthe, Gustavo Betarte, Juan Campo, Carlos Luna, and David Pichardie. 2014. "System-Level Non-interference for Constant-time Cryptography." *CCS*. ACM, 1267–1279. https://doi.org/10.1145/2660267.2660283.

Yuval Yarom and Katrina Falkner. 2014. "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack." *USENIX*, 719–732.

Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. "Last-Level Cache Side-Channel Attacks Are Practical." *S&P*, 605–622. https://doi.org/10.1109/SP.2015.43.

David Costanzo, Zhong Shao, and Ronghui Gu. 2016. "End-to-End Verification of Information-Flow Security for C and Assembly Programs." *PLDI*. ACM, 648–664. https://doi.org/10.1145/2908080.2908100.

Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. 2018. "Secure Compilation of Side-Channel Countermeasures: The Case of Cryptographic "Constant-Time"." *CSF*, 328–343. https://doi.org/10.1109/CSF.2018.00031.

Jann Horn. 2018. *Speculative Execution, Variant 4: Speculative Store Bypass*. Retrieved July 11, 2024 from https://bugs.chromium.org/p/project-zero/issues/detail?id=1528.

Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2018. "Spectre Returns! Speculation Attacks Using the Return Stack Buffer." *USENIX*.

Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. "Meltdown: Reading Kernel Memory from User Space." *USENIX*, 973–990.

Laurent Simon, David Chisnall, and Ross Anderson. 2018. "What You Get Is What You C: Controlling Side Effects in Mainstream C Compilers." *EuroS&P*, 1–15. https://doi.org/10.1109/EuroSP.2018.00009.

Paul Turner. 2018. *Retpoline: A Software Construct for Preventing Branch-Target-Injection*. Retrieved Oct. 24, 2024 from https://support.google.com/faqs/answer/7625886.

Gilles Barthe, Sandrine Blazy, Benjamin Grégoire, Rémi Hutin, Vincent Laporte, David Pichardie, and Alix Trieu. 2019. "Formal Verification of a Constant-Time Preserving C Compiler." *POPL*, 7:1–7:30. https://doi.org/10.1145/3371075.

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." *USENIX*, 249–266.

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." *S&P*, 1–19. https://doi.org/10.1109/SP.2019.00002.

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." *PLDI*. ACM, 913–926. https://doi.org/10.1145/3385412.3385970.

Roberto Guanciale, Musard Balliu, and Mads Dam. 2020. "InSpectre: Breaking and Fixing Microarchitectural Vulnerabilities by Formal Analysis." *CCS*. ACM, 1853–1869. https://doi.org/10.1145/3372297.3417246.

Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. 2020. "Spectector: Principled Detection of Speculative Information Flows." *S&P*, 1–19. https://doi.org/10.1109/SP40000.2020.00011.

Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. 2021. "SoK: Computer-Aided Cryptography." *S&P*, 777–795. https://doi.org/10.1109/SP40001.2021.00008.

Gilles Barthe, Sandrine Blazy, Rémi Hutin, and David Pichardie. 2021. "Secure Compilation of Constant-Resource Programs." *CSF*, 1–12. https://doi.org/10.1109/CSF51468.2021.00020.

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." *S&P*, 1884–1901. https://doi.org/10.1109/SP40001.2021.00046.

Gilles Barthe, Benjamin Grégoire, Vincent Laporte, and Swarn Priya. 2021. "Structured Leakage and Applications to Cryptographic Constant-Time and Cost." *CCS*. ACM, 462–476. https://doi.org/10.1145/3460120.3484761.

Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. 2021. "Hunting the Haunter — Efficient Relational Symbolic Execution for Spectre with Haunted RelSE." *NDSS*. Internet Society. https://doi.org/10.14722/ndss.2021.24286.

Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. 2021. "Hardware-Software Contracts for Secure Speculation." *S&P*, 1868–1883. https://doi.org/10.1109/SP40001.2021.00036.

Marco Patrignani and Marco Guarnieri. 2021. "Exorcising Spectres with Secure Compilers." *CCS*. ACM, 445–461. https://doi.org/10.1145/3460120.3484534.

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." *POPL*, 49:1–49:30. https://doi.org/10.1145/3434330.

Son Tuan Vu, Albert Cohen, Arnaud De Grandmaison, Christophe Guillon, and Karine Heydemann. 2021. "Reconciling Optimization with Secure Compilation." *OOPSLA*, 142:1–142:30. https://doi.org/10.1145/3485519.

Sunjay Cauligi, Craig Disselkoen, Daniel Moghimi, Gilles Barthe, and Deian Stefan. 2022. "SoK: Practical Foundations for Software Spectre Defenses." *S&P*, 666–680. https://doi.org/10.1109/SP46214.2022.9833707.

Florent Bouchez Tichadou and Fabrice Rastello. 2022. "Register Allocation." *SSA-based Compiler Design*. Springer International Publishing, 303–328. https://doi.org/10.1007/978-3-030-80515-9_22.

Minki Cho, Youngju Song, Dongjae Lee, Lennard Gäher, and Derek Dreyer. 2023. "Stuttering for Free." *OOPSLA*, 281:1677–281:1704. https://doi.org/10.1145/3622857.

Basavesh Ammanaghatta Shivakumar, Gilles Barthe, Benjamin Gregoire, Vincent Laporte, Tiago Oliveira, Swarn Priya, Peter Schwabe, and Lucas Tabary-Maujean. 2023. "Typing High-Speed Cryptography against Spectre V1." *S&P*, 1094–1111. https://doi.org/10.1109/SP46215.2023.10179418.

Zhiyuan Zhang, Gilles Barthe, Chitchanok Chuengsatiansup, Peter Schwabe, and Yuval Yarom. 2023. "Ultimate SLH: Taking Speculative Load Hardening to the Next Level." *USENIX*, 7125–7142.

Santiago Arranz Olmos, Gilles Barthe, Lionel Blatter, Benjamin Grégoire, and Vincent Laporte. 2025. "Preservation of Speculative Constant-time by Compilation." *POPL*, To appear. https://doi.org/10.1145/3704880.

Chandler Carruth. 2024. *Speculative Load Hardening — LLVM*. Retrieved July 12, 2024 from https://llvm.org/docs/SpeculativeLoadHardening.html.

A Missing proofs from Section 2

Proof of Lemma 1. Let $S_1 = S_1'.(\mathrm{pc}, \rho_1, \mu_1)$, $S_2 = S_2'.(\mathrm{pc}, \rho_2, \mu_2)$, We do the proof by induction on the first transition $S_1 \xrightarrow{\delta:\lambda} T_1$ and $S_2 \xrightarrow{\delta:\lambda} T_1$. We do case distinction.

- $\delta = \textsc{rb} = \lambda$, then $|S_1| = |S_2| > 1$. Further, $T_1 = S_1'$ and $T_2 = S_2'$. Thus, $T_1 \equiv T_2$ follows immediately from $S_1 \equiv S_2$
- $\delta = \textsc{sp}$ and $\lambda = \textsc{br}\,\mathrm{b}$ Then, $S_1 \xrightarrow{\textsc{sp}:\textsc{br}\,\mathrm{b}} S_1.(\mathrm{sc}_{\neg\mathrm{b}}, \rho_1, \mu_1)$ and $S_2 \xrightarrow{\textsc{sp}:\textsc{br}\,\mathrm{b}} S_2.(\mathrm{sc}_{\neg\mathrm{b}}, \rho_2, \mu_2)$. Clearly, $T_2 = S_2.(\mathrm{sc}_{\neg\mathrm{b}}, \rho_2, \mu_2) \equiv S_1.(\mathrm{sc}_{\neg\mathrm{b}}, \rho_1, \mu_1)$. The case of $\delta = \textsc{br}$ and $\lambda = \textsc{br}\,\mathrm{b}$ is similar.
- All other cases Because P pc has only a single successor sc, and only modifies the executing state, it is clear from $S_1' \equiv S_2'$ and both executing states being in sc, that $T_1 \equiv T_2$. □

B Missing Proofs from Section 4

Lemma 3

Proof. The $\supseteq$ direction for terminating behavior is immediate: $\xrightarrow{\cdot}_{\mathrm{t}} \subseteq \xrightarrow{\cdot}^*$. For diverging behavior, consider any $(d{:}l)$ with $T \xrightarrow{d:l}_{\mathrm{t}} \infty$. By definition, there is $T \xrightarrow{d:l}_{\mathrm{t}} V \xrightarrow{e:k}_{\mathrm{t}} \infty$. Notably, $\xrightarrow{d:l}_{\mathrm{t}}$ cannot do stuttering transitions, thus there are transitions $T \xrightarrow{\delta_0:\lambda_0} \ldots \xrightarrow{\delta_n:\lambda_n} V$ and $\lambda_0.\ldots.\lambda_n = l$, $\delta_0.\ldots.\delta_n = d$. Further, if there is $T \xrightarrow{d:l}_{\mathrm{t}} V$ then also $(S, T) \xrightarrow{e:k \blacktriangleleft d:l} (U, V)$. Thus, $S \xrightarrow{e:k}_{\mathrm{s}} U$ and $V \prec U$. By coinduction, $V \xrightarrow{e:k} \infty$, which completes this direction.

For $\subseteq$, be given the proof that justifies $T \prec S$. Show that when $T \xrightarrow{d:l} \infty$ ($T \xrightarrow{d:l}{}^* X$, X final), then (either $T = X$ are final, or) there is $T \xrightarrow{f:m}_{\mathrm{t}} V \prec U$, with $V \xrightarrow{e:k}_{\mathrm{t}} \infty$ ($V \xrightarrow{e:k}_{\mathrm{t}}{}^* X$) so that $l = m.k$ and $d = f.e$ ($l = m.k$ and $d = f.e$).

To do so, we first show the following: For every $T \prec S$ with $T \xrightarrow{d:l}{}^* X$ (note that X need not be final), either $T \xrightarrow{f:m}{}^* V \xrightarrow{e:k}{}^* X$ so that $T \xrightarrow{f:m}_{\mathrm{t}} V$ or there is a proof node $\langle \boxed{\prec}, \blacktriangleleft_{T \prec S} \vdash X \prec_{\mathrm{t}} S : d \rangle$. We do so by induction on the structure of $T \xrightarrow{d:l}{}^* X$. For the base case ($T \xrightarrow{\varepsilon:\varepsilon}{}^* T$), we already know that the proof has a node $\langle \boxed{\prec}, \blacktriangleleft_{T \prec S} \vdash T \prec_{\mathrm{t}} S : d \rangle$ for justification of $T \prec S$. For the inductive case, consider $T \xrightarrow{d:l}{}^* X' \xrightarrow{\delta:\lambda} X$. Then, by induction, either $T \xrightarrow{f:m}_{\mathrm{t}} V \xrightarrow{e:k}{}^* X'$ or there is a proof node $\langle \boxed{\prec}, \blacktriangleleft_{T \prec S} \vdash X' \prec_{\mathrm{t}} S : d \rangle$. In the first case, we are done. In the second case, we do case distinction by the rule that derives $\langle \boxed{\prec}, \blacktriangleleft_{T \prec S} \vdash X' \prec_{\mathrm{t}} S : d \rangle$. In case of rule DIRECT-TF, we get $T \xrightarrow{d:l}_{\mathrm{t}} X'$ (because the proof also derives a source sequence). Otherwise, the case is rule TGT. In that case, $\langle \boxed{\prec}, \blacktriangleleft_{T \prec S} \vdash X \prec_{\mathrm{t}} S : d.\delta \rangle \overset{\delta}{\Longrightarrow} \langle \boxed{\prec}, \blacktriangleleft_{T \prec S} \vdash X' \prec_{\mathrm{t}} S : d \rangle$ as desired.

Then, we utilize this to first prove the slightly different statement: For every $T \prec S$ with $T \xrightarrow{d:l} \infty$ ($T \xrightarrow{d:l}{}^* X$, X final), (either $T = X$ is final, or) there is $T \xrightarrow{f:m}_{\mathrm{t}} V \prec U$, with $V \xrightarrow{e:k} \infty$ ($V \xrightarrow{e:k}{}^* X$) so that $l = m.k$ and $d = f.e$ ($l = m.k$ and $d = f.e$). Towards contradiction, assume it is not the case. Then, the previous fact yields us an infinite chain of nodes in the proof tree, a contradiction to the well-foundedness of the proof: First, due to $T \xrightarrow{\varepsilon:\varepsilon}{}^* T$, $\langle \boxed{\prec}, \blacktriangleleft_{T \prec S} \vdash T \prec_{\mathrm{t}} S : \varepsilon \rangle$ is part of the proof. And when by induction the transitions $T \xrightarrow{f:m}{}^* V' \xrightarrow{\delta:\lambda} V \xrightarrow{e:k} \infty$, $f = \delta_0.\ldots.\delta_n$ imply that the proof contains $\langle \boxed{\prec}, \blacktriangleleft_{T \prec S} \vdash T \prec_{\mathrm{t}} S : \varepsilon \rangle \overset{\delta_0}{\Longleftarrow} \ldots \overset{\delta_n}{\Longleftarrow} \langle \boxed{\prec}, \blacktriangleleft_{T \prec S} \vdash V' \prec_{\mathrm{t}} S : f \rangle$, then it cannot be proven by rule DIRECT-TF as that would create $T \xrightarrow{f:m}_{\mathrm{t}} V$. Thus, it must be proven via rule TGT which further requires $\langle \boxed{\prec}, \blacktriangleleft_{T \prec S} \vdash V' \prec_{\mathrm{t}} S : f \rangle \overset{\delta}{\Longleftarrow} \langle \boxed{\prec}, \blacktriangleleft_{T \prec S} \vdash V \prec_{\mathrm{t}} S : f.\delta \rangle$.

Finally, we derive the result by coinduction, which yields $V \xrightarrow{e:k}_{\mathrm{t}} \infty$. With $T \xrightarrow{f:m}_{\mathrm{t}} V$, we compose $T \xrightarrow{f.e:m.k}_{\mathrm{t}} \infty$. □

C Missing proofs from section 5 and Transfer for Liveness Analysis

We define the transfer functions for Liveness analysis on instructions. Write F_i for F_{pc} with $P\,\mathrm{pc} = i$.

$$F_{a \coloneqq \mathbf{a}[x]_{\rangle \mathrm{sc}}}\, l = \begin{cases} l & a \notin l \\ (l \setminus \{a\}) \cup \{(\mathbf{a}, \mathsf{n})\} & x = \mathsf{n} \\ (l \setminus \{a\}) \cup \mathit{Mem}) & x = b \end{cases} \qquad F_{\mathsf{nop}_{\rangle \mathrm{sc}}}\, l = l$$

$$F_{\mathbf{a}[a] \coloneqq x_{\rangle \mathrm{sc}}}\, l = \begin{cases} l & x = \mathsf{n} \in |\mathbf{a}|, (\mathbf{a}, \mathsf{n}) \notin l \\ (l \setminus \{(\mathbf{a}, \mathsf{n})\}) & x = \mathsf{n} \in |\mathbf{a}|, (\mathbf{a}, \mathsf{n}) \in \mathrm{lm} \\ l \cup \{b\} & x = b \end{cases} \qquad F_{a \coloneqq b \oplus c_{\rangle \mathrm{sc}}}\, l = \begin{cases} (l \setminus \{a\}) \cup \{b, c\} & a \in l \\ l & a \notin \mathrm{lr} \end{cases}$$

$$F_{\mathsf{br}\ a_{\rangle \mathrm{sc_t}, \mathrm{sc_f}}}\, l = l \cup \{a\} \qquad F_{\mathsf{ret}}\, l = \mathcal{X}_0 = \mathit{Mem}$$

Remaining cases for Theorem 7

- $\langle a \coloneqq b \oplus c_{\rangle \mathrm{sc}}, \mathsf{nop}_{\rangle \mathrm{sc}} \rangle$ For the first part, we have $t_1 \xrightarrow{\bullet:\bullet} v_1$ and need to show $s_1 \xrightarrow{\bullet:\bullet} u_1$ and $v_1 \prec u_1$. The former is immediate from semantics. For the latter, we know that $a \notin \mathcal{X}\,\mathrm{pc}$ because the instruction was replaced by nop. The transfer of $\mathcal{X}\,\mathrm{pc}$ along i is then $F_{\mathrm{pc}}\,(\mathcal{X}\,\mathrm{pc}) = \mathcal{X}\,\mathrm{pc}$ because an assignment to a non-live register does not make any registers live. Together with (bwd), we get $F_{\mathrm{pc}}\,(\mathcal{X}\,\mathrm{pc}) = \mathcal{X}\,\mathrm{pc} \supseteq F_{\mathrm{sc}}\,(\mathcal{X}\,\mathrm{sc})$. Because registers and memory of t_1 and v_1 are equal and s_1 and u_1 only differ on a, $t_1 \prec s_1$ implies $v_1 \prec u_1$ because $a \notin \mathcal{X}\,\mathrm{pc} \supseteq F_{\mathrm{sc}}\,(\mathcal{X}\,\mathrm{sc})$. For the second part, further assume $s_2 \xrightarrow{\bullet:\bullet} u_2$. We need to show that $t_2 \xrightarrow{\bullet:\bullet} v_2$ and $v_2 \prec u_2$. The former is again immediate from semantics. For the latter, we have the same arguments as for the first part: Memory and register contents of t_2 and v_2 are equal and s_2 and u_2 only differ in a.
- $\mathbf{a}[b] \coloneqq c_{\rangle \mathrm{sc}}, \mathbf{a}[b] \coloneqq c_{\rangle \mathrm{sc}}$ There are two subcases - either safe or unsafe store.
  For safe store we get $t_1 \xrightarrow{\bullet:\mathrm{ST}\,\mathsf{n}} v_1$ and since $b \in F_{\mathrm{pc}}\,(\mathcal{X}\,\mathrm{pc})$, $\rho_{t_1}\, b = \rho_{s_1}\, b$, so $s_1 \xrightarrow{\bullet:\mathrm{ST}\,\mathsf{n}} u_1$. Also, $F_{\mathrm{pc}}\,(\mathcal{X}\,\mathrm{pc}) \cup \{(\mathbf{a}, \mathsf{n})\} \geq \mathcal{X}\,\mathrm{pc} \geq F_{\mathrm{sc}}\,(\mathcal{X}\,\mathrm{sc})$ and $\mu_{t_1} = \mu_{v_1}$ and $\mu_{s_1} = \mu_{u_1}$ on all slots except $(\mathbf{a}, \mathsf{n})$. But due to $c \in F_{\mathrm{pc}}\,(\mathcal{X}\,\mathrm{pc})$, we also have $\mu_{t_1}\, \mathbf{a}\, \mathsf{n} = \mu_{v_1}\, \mathbf{a}\, \mathsf{n}$, so $v_1 \prec u_1$. Further $t_2 \xrightarrow{\bullet:\mathrm{ST}\,\mathsf{n}} v_2$ and $\bullet \lhd_{t_2 \prec s_2} \bullet$ are a given.
  For unsafe store we have an analogue proof, except both directives are $\mathrm{SU}\,\mathbf{b}\,\mathsf{m}$.
- $\mathbf{a}[b] \coloneqq c_{\rangle \mathrm{sc}}, \mathsf{nop}_{\rangle \mathrm{sc}}$ Analogue to the previous case, except $\kappa = \bullet$ and the argument for $\mu_{t_1}\, \mathbf{a}\, \mathsf{n} = \mu_{v_1}\, \mathbf{a}\, \mathsf{n}$ does not hold anymore. However, since $(\mathbf{a}, \mathsf{n}) \notin \mathcal{X}\,\mathrm{pc} \geq F_{\mathrm{sc}}\,(\mathcal{X}\,\mathrm{sc})$, we don't need it for $v_1 \prec u_1$.
- $a \coloneqq \mathbf{a}[b]_{\rangle \mathrm{sc}}, a \coloneqq \mathbf{a}[b]_{\rangle \mathrm{sc}}$ and $a \coloneqq \mathbf{a}[b]_{\rangle \mathrm{sc}}, \mathsf{nop}_{\rangle \mathrm{sc}}$ The argumentation is analogue to the previous two cases.
- $\mathsf{nop}_{\rangle \mathrm{sc}}, \mathsf{nop}_{\rangle \mathrm{sc}}$ We get $t_1 \xrightarrow{\bullet:\bullet} v_1$ and $s_1 \xrightarrow{\bullet:\bullet} u_1$. Also, $F_{\mathrm{pc}}\,(\mathcal{X}\,\mathrm{pc}) = \mathcal{X}\,\mathrm{pc} \geq F_{\mathrm{sc}}\,(\mathcal{X}\,\mathrm{sc})$ and $\rho_{t_1} = \rho_{v_1}$ and $\rho_{s_1} = \rho_{u_1}$, so $v_1 \prec u_1$. Further $t_2 \xrightarrow{\bullet:\bullet} v_2$ and $\bullet \lhd_{t_2 \prec s_2} \bullet$ are a given.
- $a \coloneqq b \oplus c_{\rangle \mathrm{sc}}, a \coloneqq b \oplus c_{\rangle \mathrm{sc}}$ We get $t_1 \xrightarrow{\bullet:\bullet} v_1$ and $s_1 \xrightarrow{\bullet:\bullet} u_1$. Also, $F_{\mathrm{pc}}\,(\mathcal{X}\,\mathrm{pc}) \cup \{a\} \geq \mathcal{X}\,\mathrm{pc} \geq F_{\mathrm{sc}}\,(\mathcal{X}\,\mathrm{sc})$ and $\rho_{t_1} = \rho_{v_1}$ and $\rho_{s_1} = \rho_{u_1}$ on all registers except a. But due to $b, c \in F_{\mathrm{pc}}\,(\mathcal{X}\,\mathrm{pc})$, we also have $\rho_{t_1}\, a = \rho_{v_1}\, a$, so $v_1 \prec u_1$. Further $t_2 \xrightarrow{\bullet:\bullet} v_2$ and $\bullet \lhd_{t_2 \prec s_2} \bullet$ are a given.
- ret No transition can be made by t_1, the case does not exist.
- $\mathsf{br}\ a_{\rangle \mathrm{sc_t}, \mathrm{sc_f}}, \mathsf{br}\ a_{\rangle \mathrm{sc_t}, \mathrm{sc_f}}$ We get $T_1.t_1 \xrightarrow{\mathrm{SP}\neg\mathrm{b}:\mathrm{BR}\,\neg\mathrm{b}} T_1.v_{1,\mathrm{b}}.v_{1,\neg\mathrm{b}} = V_1$ where $\mathrm{b} = (\rho_{t_1}\, a = 0)$ and by $a \in F_{\mathrm{pc}}\,(\mathcal{X}\,\mathrm{pc})$, $\rho_{t_1}\, a = \rho_{s_1}\, a$ so $S_1.s_1 \xrightarrow{\mathrm{SP}\neg\mathrm{b}:\mathrm{BR}\,\mathrm{b}} S_1.u_{1,\mathrm{b}}.u_{1,\neg\mathrm{b}} = U_1$. Further, $F_{\mathrm{pc}}\,(\mathcal{X}\,\mathrm{pc}) \geq \mathcal{X}\,\mathrm{pc} \geq F_{\mathrm{sc_b}}\,(\mathcal{X}\,\mathrm{sc_b})$. Thus, $v_{1,\mathrm{b}} \prec u_{1,\mathrm{b}}$ and $v_{1,\neg\mathrm{b}} \prec u_{1,\neg\mathrm{b}}$. Further, if $S_2 \xrightarrow{\mathrm{SP}\neg\mathrm{b}:\mathrm{BR}\,\neg\mathrm{b}} U_2$, by same arguments, we have $T_2 \xrightarrow{\mathrm{SP}\neg\mathrm{b}:\mathrm{BR}\,\neg\mathrm{b}} V_2$ and $\mathrm{SP}\neg\mathrm{b} \lhd_{t_2 \prec s_2} \mathrm{SP}\neg\mathrm{b}$.

D Additional Material and Missing proofs for Section 6

Rules D.1: Shuffle Semantics

MOVE

$$\frac{P\,\mathsf{pc} = a := b_{\rangle \mathsf{sc}}}{(\mathsf{pc}, \rho, \mu) \xrightarrow{\bullet:\bullet} (\mathsf{sc}, \rho[a \mapsto \rho\, b], \mu)}$$

FILL

$$\frac{P\,\mathsf{pc} = a := \mathbf{stk}[\mathsf{l}]_{\rangle \mathsf{sc}}}{(\mathsf{pc}, \rho, \mu) \xrightarrow{\bullet:\mathrm{LD}\,\mathsf{l}} (\mathsf{sc}, \rho[a \mapsto \mu\, \mathbf{stk}\, \mathsf{l}], \mu)}$$

SPILL

$$\frac{P\,\mathsf{pc} = \mathbf{stk}[\mathsf{l}] := a_{\rangle \mathsf{sc}}}{(\mathsf{pc}, \rho, \mu) \xrightarrow{\bullet:\mathrm{ST}\,\mathsf{l}} (\mathsf{sc}, \rho, \mu[(\mathbf{stk}, \mathsf{l}) \mapsto \rho\, a])}$$

Missing rules for $P :: [P]^{\mathrm{ra}}$

Rules D.2: Basic

POISON-NOP

$$\frac{P\,s = \mathsf{nop}_{\rangle \mathsf{sc}} \quad s \xrightarrow{\bullet:\bullet} u \quad t \xrightarrow{\bullet:\bullet} v}{(s, t, \mathrm{ps}) \xrightarrow{\bullet:\bullet::\bullet:\bullet} (u, v, \mathrm{ps})}$$

HEALTHY-BRANCH

$$\frac{P\,s = \mathsf{br}\ b_{\rangle \mathsf{sc}_\mathsf{t}, \mathsf{sc}_\mathsf{f}} \quad \mathrm{ps}\, b = \mathsf{h} \quad s \xrightarrow{\mathrm{BR:BR}\,b} u \quad t \xrightarrow{\mathrm{BR:BR}\,b} v}{(s, t, \mathrm{ps}) \xrightarrow{\mathrm{BR:BR}\,b::\mathrm{BR:BR}\,b} (u, v, \mathrm{ps})}$$

POISON-ASGN

$$\frac{P\,s = a := b \oplus c_{\rangle \mathsf{sc}} \quad s \xrightarrow{\bullet:\bullet} u \quad t \xrightarrow{\bullet:\bullet} v \quad \mathrm{pv} = (\mathrm{ps}\, b = \mathrm{ps}\, c = \mathsf{h})\ ?\ \mathsf{h} : \mathsf{p}}{(s, t, \mathrm{ps}) \xrightarrow{\bullet:\bullet::\bullet:\bullet} (u, v, \mathrm{ps}[a \mapsto \mathrm{pv}])}$$

Rule POISON-ASGN propagates poison values for assignments. Notice that weakly poisoned values become poisoned because the operator's result value is most likely a non-zero value. Thus, only if both arguments are healthy, we propagate h to the assigned register and p otherwise. HEALTHY-BRANCH forbids weakly poisoned branching conditions even though they are safe to leak. This is to avoid that P and $[P]^{\mathrm{ra}}$ arrive at different program points (up to Φ). Branching with weakly poisoned registers could be supported by letting one state speculate while the other does regular branching to keep them at the same program point. However, this would violate the condition $|S| = |T|$ and leads to a less intuitive product definition.

Lemma 4

Proof. We do a case distinction on the transition rule for $(S, T, \mathsf{P}) \xrightarrow{\gamma:\kappa::\delta:\lambda} (U, V, \mathsf{Q})$. To that end, let $\mathsf{P} = \mathsf{P}'.\mathrm{ps}$ and $\mathsf{Q} = \mathsf{Q}'.\mathrm{ps}'$, $U = U'.u$ and $V = V'.v$, and $S = S'.s$ and $T = T'.t$. Let further be $i = P\,s$ and $i' = [P]^{\mathrm{ra}}\,t$. For all cases except POISON-ROLLBACK and HEALTHY-SPEC it suffices to show that $u\ \Psi^{\mathrm{ps}'}\ v$. All those rules only modify the top states and poison values, so $\mathsf{Q}' = \mathsf{Q}'$. The definition of Ψ^{Q} is then satisfied from $S'\ \Psi^{\mathsf{Q}'}\ T'$.

We first do the separate two cases who change the size of the speculating states.

- HEALTHY-SPEC Then, $i = \mathsf{br}\ b$, $\mathrm{ps}\, b = \mathsf{h}$, and $\mathsf{Q} = \mathsf{P}'.\mathrm{ps}.\mathrm{ps}$. Further, u and s as well as t and v are fully equal except for the program counter. Because $s\ \Psi^{\mathrm{ps}}\ t$ and Ψ^{ps} is indifferent to the program counter, $u\ \Psi^{\mathrm{ps}}\ v$.
- POISON-ROLLBACK Then, $U = S'\ \Psi^{\mathsf{P}'}\ T' = V$.

Now to the other cases.

- POISON-FILL, POISON-SPILL, POISON-MOVE, POISON-SHUFFLE-SFENCE, POISON-SFENCE, POISON-SLH, POISON-NOP, and HEALTHY-BRANCH $\mathrm{ps}' = \mathrm{ps}$. Further, u and s as well as t and v are fully equal except for the program counter. Because $s\ \Psi^{\mathrm{ps}}\ t$ and Ψ^{ps} is indifferent to the program counter, $u\ \Psi^{\mathrm{ps}}\ v$.

- POISON-LOAD-STKUNSAFE, POISON-LOAD-SAFE, and POISON-LOAD-UNSAFE Then, $i = a := \mathbf{a}[b]$ and $\mathsf{ps}' = \mathsf{ps}[a \mapsto \mathsf{ps}]$. Further, u and s as well as t and v are equal except for the program counter and a as well as $a' = \Psi\, v\, a$. But a is poisoned in ps', so $\Psi^{\mathsf{ps}'}$ is indifferent to its value, $u\ \Psi^{\mathsf{ps}'}\ v$.
- HEALTHY-LOAD-SAFE Then, $i = a := \mathbf{a}[b]$ and $\mathsf{n} = \rho_s\, b$ and $\mathsf{ps}' = \mathsf{ps}[a \mapsto \mathsf{ps}\,\mathbf{a}\,\mathsf{n}] \leq \mathsf{ps}[a \mapsto \mathsf{ps}]$. Further, u and s as well as t and v are equal except for the program counter and a as well as $a' = \Psi\, v\, a$. The value of a is $\mu_s\,\mathbf{a}\,\mathsf{n}$ and of a' is $\mu_t\,\mathbf{a}\,\mathsf{n}$. But a has the poison value of ps for $(\mathbf{a}, \mathsf{n})$ in ps', so $u\ \Psi^{\mathsf{ps}'}\ v$ follows from $s\ \Psi^{\mathsf{ps}}\ t$.
- HEALTHY-LOAD-UNSAFE Analogue to the previous case, but $(\mathbf{a}, \mathsf{n})$ swapped to $(\mathbf{b}, \mathsf{m})$.
- POISON-STORE-STKUNSAFE Then, $i = \mathbf{a}[b] := c$ and $\gamma = \mathrm{su}\,\mathbf{a}\,\mathsf{l}$ and $\delta = \mathrm{su}\,\mathbf{stk}\,\mathsf{m}$ and $\Psi\, v\, d = (\mathbf{stk}, \mathsf{m})$. We get $\mathsf{ps}' = \mathsf{ps}[d, (\mathbf{stk}, \mathsf{m}) \mapsto \mathsf{p}]$. Further, values of u and s are equal except for d and values of t and v are equal except $\Psi\, v\, d = (\mathbf{stk}, \mathsf{m})$. Again, those are poisoned in ps', so $u\ \Psi^{\mathsf{ps}'}\ v$ follows from $s\ \Psi^{\mathsf{ps}}\ t$.
- POISON-STORE-SAFE Then, $i = \mathbf{a}[b] := c$ and $\rho_s\, b = \mathsf{k}$ and $\rho_t\ (\Psi\, t\, b = 0$ due to weak poisonedness. We get $\mathsf{ps}' = \mathsf{ps}[(\mathbf{a}, \mathsf{k}), (\mathbf{a}, 0) \mapsto \mathsf{p}]$. Further, values of u and s are equal except for $(\mathbf{a}, \mathsf{k})$ and values of t and v are equal except $(\mathbf{a}, 0)$. Again, those are poisoned in ps', so $u\ \Psi^{\mathsf{ps}'}\ v$ follows from $s\ \Psi^{\mathsf{ps}}\ t$.
- POISON-STORE-UNSAFE Analogue to the previous case, but k swapped to l.
- HEALTHY-STORE-SAFE Then, $i = \mathbf{a}[b] := c$ and $\rho_s\, b = \rho_t\ (\Psi\, t\, b) = \mathsf{n} \in |\mathbf{a}|$ due to healthiness. Let further $\Psi\, t\, c = c'$. We get $\mathsf{ps}' = \mathsf{ps}[(\mathbf{a}, \mathsf{n}) \mapsto \mathsf{ps}\, c]$. Further, values of u and s are equal except for $(\mathbf{a}, \mathsf{k})$ and values of t and v are equal except $(\mathbf{a}, 0)$. The value of $(\mathbf{a}, \mathsf{n})$ in u is $\rho_s\, c$ and in v is $\rho_t\, c'$. But $(\mathbf{a}, \mathsf{n})$ has the poison value of ps for c in ps', so $u\ \Psi^{\mathsf{ps}'}\ v$ follows from $s\ \Psi^{\mathsf{ps}}\ t$.
- HEALTHY-STORE-UNSAFE Analogue, but $(\mathbf{a}, \mathsf{n})$ swapped for $(\mathbf{b}, \mathsf{m})$.
- POISON-SHUFFLE-SLH Then, $i' = \mathtt{slh}\, a'$, where $\Psi\, s\, a = \Psi\, t\, a = a'$. We get $\mathsf{ps}' = \mathsf{ps}[a \mapsto \mathsf{wp}]$. t is equal to v and if S is speculating, then ps' is set to wp and $\rho_v\, a' = 0$. Otherwise, t is equal to v in values. In both cases, $u\ \Psi^{\mathsf{ps}'}\ v$ follows from $s\ \Psi^{\mathsf{ps}}\ t$. □

Remaining transfer functions for $P :: [P]^{\mathsf{ra}}$

$$F_{\mathsf{nop},\mathsf{sc}}\ \mathsf{ps} = G_{a' := b',\mathsf{sc}}\ \mathsf{ps} = G_{a' := \mathbf{stk}[\mathsf{l}],\mathsf{sc}}\ \mathsf{ps} = G_{\mathbf{stk}[\mathsf{l}] := b',\mathsf{sc}}\ \mathsf{ps} = \mathsf{ps}$$

$$F_{\mathtt{sfence},\mathsf{sc}}\ \mathsf{ps} = \mathsf{h}$$

$$F_{\mathtt{br}\ b,\mathsf{sc_t},\mathsf{sc_f}}\ \mathsf{ps} = \begin{cases} \mathsf{ps} & \mathsf{pr}\, b = \mathsf{h} \\ \mathsf{p} & \mathsf{pr}\, b \geq \mathsf{wp} \end{cases}$$

Lemma 6

Proof. By induction, all transitions update poison types monotonically. For the induction step, consider $S = S'.s$ and $T = T'.t$. We do a case distinction on the transition rule for $(S, T, \mathcal{X}_{S,T}) \xrightarrow{Y:\kappa::\delta:\lambda} (U, V, \mathsf{Q})$. To that end, let $\mathcal{X}_{S,T} = \mathsf{P}'.\mathsf{ps}$ and $\mathsf{Q} = \mathsf{Q}'.\mathsf{ps}'$, $U = U'.u$ and $V = V'.v$. Let further be $i = P\, s$ and $i' = [P]^{\mathsf{ra}}\, t$. We need to show for all cases except POISON-ROLLBACK and HEALTHY-SPEC that $\mathsf{ps}' \leq F_i\, \mathsf{ps}$ (respectively $\mathsf{ps}' \leq G_{i'}\, \mathsf{ps}$ for shuffling states). Then, by equation (fwd), $\mathsf{ps}' \leq F_i\, \mathsf{ps} \leq \mathcal{X}\,(u, v)$ (respectively $\mathsf{ps}' \leq G_{i'}\, \mathsf{ps} \leq \mathcal{X}\,(u, v)$). All those rules only modify the top states and poison values, so $\mathsf{P}' = \mathsf{Q}' = \mathcal{X}_{U',V'}$. Together, we get $\mathsf{Q} \leq \mathcal{X}_{U,V}$.

We first do the separate two cases who change the size of the speculating states.

- HEALTHY-SPEC Then, $i = \mathtt{br}\ b$, $\mathsf{ps}\, b = \mathsf{h}$, and $\mathsf{Q} = \mathsf{P}.\mathsf{ps}$. We have $\mathsf{ps} = F_i\, \mathsf{ps} \leq \mathcal{X}\,(s, t)$. And because $U = S.u$ and $V = T.v$, $\mathsf{P} \leq \mathsf{P}$, we have $\mathsf{P}.\mathsf{ps} \leq \mathsf{P}.\mathcal{X}\,(s, t) = \mathcal{X}_{U,V}$.
- POISON-ROLLBACK Then, $U = S'$ and $V = T'$, and since $\mathcal{X}_{S,T} = \mathcal{X}_{S',T'}.\mathsf{ps}$, $\mathsf{Q}' = \mathcal{X}_{U,V}$.

Now to the other cases.

- POISON-FILL, POISON-SPILL, POISON-MOVE, and POISON-SHUFFLE-SFENCE $\mathsf{ps}' = \mathsf{ps} = G_{i'}\,\mathsf{ps}$.
- POISON-SFENCE, POISON-SLH, POISON-NOP, and HEALTHY-BRANCH $\mathsf{ps}' = \mathsf{ps} = F_i\,\mathsf{ps}$.
- POISON-LOAD-STKUNSAFE, POISON-LOAD-SAFE, and POISON-LOAD-UNSAFE Then, $i = a := \mathbf{a}[b]$ and $\mathsf{ps}' = \mathsf{ps}[a \mapsto \mathsf{ps}] = F_i\,\mathsf{ps}$.
- HEALTHY-LOAD-SAFE Then, $i = a := \mathbf{a}[b]$ and $\mathsf{n} = \rho_s\, b$ and $\mathsf{ps}' = \mathsf{ps}[a \mapsto \mathsf{ps}\,\mathbf{a}\,\mathsf{n}] \leq \mathsf{ps}[a \mapsto \mathsf{ps}] = F_i\,\mathsf{ps}$.
- HEALTHY-LOAD-UNSAFE Then, $i = a := \mathbf{a}[b]$ and $\gamma = \textsc{lu}\,\mathbf{b}\,\mathsf{m}$ and $\mathsf{ps}' = \mathsf{ps}[a \mapsto \mathsf{ps}\,\mathbf{b}\,\mathsf{m}] \leq \mathsf{ps}[a \mapsto \mathsf{ps}] = F_i\,\mathsf{ps}$.
- POISON-STORE-STKUNSAFE Then, $i = \mathbf{a}[b] := c$ and $\gamma = \textsc{su}\,\mathbf{a}\,\mathsf{l}$ and $\delta = \textsc{su}\,\mathbf{stk}\,\mathsf{m}$ and $\Psi\,t\,d = (\mathbf{stk}, \mathsf{m})$. We get $\mathsf{ps}' = \mathsf{ps}[d, (\mathbf{stk}, \mathsf{m}) \mapsto \mathsf{p}] \leq \mathsf{ps}[\mathit{Mem} \overset{\sqcup}{\mapsto} \mathsf{ps}\,c][\mathit{Reg}, \mathbf{a} \mapsto \mathsf{p}] = F_i\,\mathsf{ps}$.
- POISON-STORE-SAFE Then, $i = \mathbf{a}[b] := c$ and $\rho_s\, b = \mathsf{k}$ and $\rho_t\,(\Psi\,t\,b = 0$ due to weak poisonedness. We get $\mathsf{ps}' = \mathsf{ps}[(\mathbf{a}, \mathsf{k}), (\mathbf{a}, 0) \mapsto \mathsf{p}] \leq \mathsf{ps}[\mathit{Mem} \overset{\sqcup}{\mapsto} \mathsf{ps}\,c][\mathit{Reg}, \mathbf{a} \mapsto \mathsf{p}] = F_i\,\mathsf{ps}$.
- POISON-STORE-UNSAFE Then, $i = \mathbf{a}[b] := c$ and $\gamma = \textsc{su}\,\mathbf{a}\,\mathsf{l}$ and $\rho_t\,(\Psi\,t\,b = 0$ due to weak poisonedness. We get $\mathsf{ps}' = \mathsf{ps}[(\mathbf{a}, \mathsf{l}), (\mathbf{a}, 0) \mapsto \mathsf{p}] \leq \mathsf{ps}[\mathit{Mem} \overset{\sqcup}{\mapsto} \mathsf{ps}\,c][\mathit{Reg}, \mathbf{a} \mapsto \mathsf{p}] = F_i\,\mathsf{ps}$.
- HEALTHY-STORE-SAFE Then, $i = \mathbf{a}[b] := c$ and $\rho_s\, b = \rho_t\,(\Psi\,t\,b = \mathsf{n} \in |\mathbf{a}|$ due to healthiness. We get $\mathsf{ps}' = \mathsf{ps}[(\mathbf{a}, \mathsf{n}) \mapsto \mathsf{ps}\,c] \leq \mathsf{ps}[\mathit{Mem} \overset{\sqcup}{\mapsto} \mathsf{ps}\,c][\mathit{Reg}, \mathbf{a} \mapsto \mathsf{p}] = F_i\,\mathsf{ps}$.
- HEALTHY-STORE-UNSAFE Then, $i = \mathbf{a}[b] := c$ and $\delta = \gamma = \textsc{su}\,\mathbf{b}\,\mathsf{m}$. We get $\mathsf{ps}' = \mathsf{ps}[(\mathbf{b}, \mathsf{m}) \mapsto \mathsf{ps}\,c] \leq \mathsf{ps}[\mathit{Mem} \mapsto \mathsf{ps}\,c][\mathit{Reg}, \mathbf{a} \overset{\sqcup}{\mapsto} \mathsf{p}] = F_i\,\mathsf{ps}$.
- POISON-SHUFFLE-SLH Then, $i' = \mathtt{slh}\,a'$, where $\Psi\,s\,a = \Psi\,t\,a = a'$. We get $\mathsf{ps}' = \mathsf{ps}[a \mapsto \mathsf{wp}] = G_{i'}\,\mathsf{ps}$. □

Full proof for Lemma 7

Like for dead code elimination in Section 5, we need to prove that $(\prec, \lhd)$ **(i)** is a simulation (Definition 4), **(ii)** respects sec (Definition 2), and **(iii)** is snippy (Definition 6).

The intermediary lemmas are proved subsequently in their own sections.

PROOF THAT $(\prec, \lhd)$ IS SEC-RESPECTING. We required $\mathsf{sec}\,\mathbf{stk} = \mathsf{L}$ so when $t_1 =_{\mathsf{sec}} t_2$, and $s_1\ \Psi^{\mathsf{h}}\ t_1$ as well as $s_2\ \Psi^{\mathsf{h}}\ t_2$, then s_1 and t_1 equal on all values in memory other than **stk** (remember that ps is not defined on **stk**). Similarly, s_2 and t_2 equal on memory. So, $s_1 =_{\mathsf{sec}} s_2$ if and only if $t_1 =_{\mathsf{sec}} t_2$. □

PROOF THAT $(\prec, \lhd)$ IS A SIMULATION. In order to prove that $\prec$ is a simulation (Definition 4), we first need to show that for all initial t of $[P]^{\mathsf{ra}}$ there is an initial s for P with $t \prec s$. Secondly, for all pairs of states $T \prec S$, we need to derive $\langle \boxed{\prec}, \lhd_{T \prec S} \vdash T \prec_{\mathsf{t}} S : \varepsilon \rangle$ in Rules 4.1. For the initial states, let $t = (\mathsf{init}', \rho', \mu')$. We construct $s = (\mathsf{init}, \rho, \mu)$. First, we know from instruction matching, that $\Phi\,\mathsf{init} = \mathsf{init}'$, thus $s\ \Phi\ t$. We can then choose $\mu = \mu'$ and $\rho\,a = [\![\Psi\,\mathsf{init}'\,a]\!]_t$.

For the second part, consider $T \prec S$. We need to provide a proof for $\langle \boxed{\prec}, \lhd_{T \prec S} \vdash T \prec_{\mathsf{t}} S : \varepsilon \rangle$. The case where T is final is trivial, so consider non-final T and S. We need two small helping-lemmas that state that Φ-injected instructions and shuffle instructions of $[P]^{\mathsf{ra}}$ are preserved to the product. The proof of these we skip, but they rely on the fact that (Φ, Ψ) is poison-typable with $\mathcal{X}$.

LEMMA 8. *Let $T \prec S$, $T \xrightarrow{\delta:\lambda} X$, and $\mathsf{P.ps} = \mathcal{X}_{S,T}$. Then, $(S, T, \mathsf{ps}) \xrightarrow{\gamma:\kappa::\delta:\lambda} (U, X, \mathsf{ps}')$ with $U\ \Psi_{\mathcal{X}}^{\mathsf{ps}'}\ X$.*

LEMMA 9. *Let $U\ \Psi_{\mathcal{X}}^{\mathsf{ps}}\ X$ and $X \xrightarrow{d:l}{}^{*} V$ be shuffle only in $[P]^{\mathsf{ra}}$. Then, $P :: [P]^{\mathsf{ra}}$ has a unique execution $(U, X, \mathsf{ps}) \xrightarrow{\varepsilon:\varepsilon::d:l}{}^{*} (U, V, \mathsf{ps}')$ with $U\ \Psi_{\mathcal{X}}^{\mathsf{ps}'}\ V$.*

Let $T = T'.t$ and $S = S'.s$, and $\mathcal{X}\,(s, t) = \mathsf{ps}$. We sketch how to derive $\langle \boxed{\prec}, \lhd_{T \prec S} \vdash T \prec_{\mathsf{t}} S : \varepsilon \rangle$. We will first explore executions from T in $[P]^{\mathsf{ra}}$ with TGT, perform a directive transformation with DIRECT-TF, replay the execution from S in P, and finally prove that the reached states belong to $\prec$ to end the proof with COIND. First, we explore executions from T in $[P]^{\mathsf{ra}}$ with TGT. Every execution in $[P]^{\mathsf{ra}}$ from T eventually enters a Φ-injected state. So we explore executions $T \xrightarrow{\delta:\lambda} X \xrightarrow{d:l}{}^{*} V$,

where V is again Φ-injected but no intermediary state is. The explored executions have the shape of the teal and red paths of $[P]^{\mathrm{ra}}$ in Figure 4. We thus need to prove $\langle \boxed{\prec}, \triangleleft_{T\prec S} \vdash V \prec_{\mathrm{t}} S : \delta.d \rangle$. The next step is to perform directive transformation with $\triangleleft_{T\prec S}$. But we defined the transformation on Φ-intervals. So we find the appropriate transitions for $T \xrightarrow{\delta:\lambda} X \xrightarrow{d:l}{}^{*} V$ in $P :: [P]^{\mathrm{ra}}$. For the first transition, Lemma 8 yields $(S, T, \mathrm{ps}) \xrightarrow{\gamma:\kappa::\delta:\lambda} (U, X, \mathrm{ps}')$ so that $U\ \Psi_X^{\mathrm{ps}'}\ X$. For the remaining transitions, there are two cases:

► *No* RB *occurs in* d. Then, $X \xrightarrow{d:l}{}^{*} V$ is shuffle-only, and we can apply Lemma 9 for transitions $(U, X, \mathrm{ps}') \xrightarrow{\varepsilon:\varepsilon::d:l}{}^{*} (U, V, \mathrm{ps}'')$ with $U\ \Psi_X^{\mathrm{ps}''}\ V$. Together with the first transition, $P :: [P]^{\mathrm{ra}}$ can execute the transitions $(S, T, \mathrm{ps}) \xrightarrow{\gamma:\kappa::\delta:\lambda} (U, X, \mathrm{ps}') \xrightarrow{\varepsilon:\varepsilon::d:l}{}^{*} (U, V, \mathrm{ps}'')$. This is precisely the red Φ-interval, so $\gamma \triangleleft_{T\prec S} \delta.d$. We can thus use DIRECT-TF and are left proving $\langle \boxed{\prec}, \triangleleft_{T\prec S} \vdash V \prec_{\mathrm{s}} S : \gamma \rangle$. However, $(S, T, \mathrm{ps}) \xrightarrow{\gamma:\kappa::\delta:\lambda} (U, X, \mathrm{ps}')$ already provides us the replay $S \xrightarrow{\gamma:\kappa} U$ in P. We can use it with SRC, and it remains to show $\langle \boxed{\prec}, \triangleleft_{T\prec S} \vdash V \prec_{\mathrm{s}} U : \varepsilon \rangle$. In order to use COIND, we need to establish $V \prec U$, i.e. $U\ \Phi\ V$ and $U\ \Psi_X\ V$. The condition $U\ \Phi\ V$ holds because it has been the condition for terminating exploration in $[P]^{\mathrm{ra}}$. For the latter condition, we have already established $U\ \Psi_X^{\mathrm{ps}''}\ V$. Thanks to Lemma 6, we also have $\mathcal{X}_{U',V'}.\mathrm{ps}'' \leq \mathcal{X}_{U,V}$, where $U = U'.u$ and $V = V'.v$. We designed the lattice so that Ψ^{ps} is monotonic in ps, so they yield $U\ \Psi_X\ V$. Thus, we get $V \prec U$ and can use COIND.

► *Otherwise, let* RB *occur in* d. Then, the explored execution from X is $X \xrightarrow{d':l'}{}^{*} Y \xrightarrow{\mathrm{RB}:\mathrm{RB}} V$ with $d = d'.\mathrm{RB}$. Indeed, d must end with RB, because V is already Φ-injected: When $X = X'.x$ and $U = U'.u$, then $V = X'$. Further, X' and U' are already contained in T and S, respectively. Because $S\ \Phi\ T$ we also have $U'\ \Phi\ X' = V$. So V is Φ-injected and exploration stopped upon discovering it. Again, by Lemma 9 there is $(U, X, \mathrm{ps}') \xrightarrow{\varepsilon:\varepsilon::d:l}{}^{*} (U, Y, \mathrm{ps}'')$ with $U\ \Psi_X^{\mathrm{ps}''}\ Y$. By definition of $P :: [P]^{\mathrm{ra}}$, $(U, Y, \mathrm{ps}') \xrightarrow{\mathrm{RB}:\mathrm{RB}::\mathrm{RB}:\mathrm{RB}} (U', X', \mathcal{X}_{U',X'})$. This is one of the teal Φ-intervals, so $\gamma.\mathrm{RB} \triangleleft_{T\prec S} \delta.d'.\mathrm{RB}$. The remaining arguments for SRC and COIND are similar to the previous case. □

Corollary 1. *For $T \prec S$, $(S, T, \mathcal{X}_{S,T}) \xrightarrow{e:k::d:l}{}^{*} (U, V, \mathrm{Q})$ is a Φ-interval iff $(S, T) \xrightarrow{e:k\triangleleft d:l} (U, V)$.*

Proof that $(\prec, \triangleleft)$ is snippy. In order to prove the simulation snippy, we are given the black parts of Figure 2. So, consider $T_1 \prec S_1$, $T_2 \prec S_2$, $T_1 \equiv T_2$, and $S_1 \equiv S_2$. Further, consider a simulation interval $(S_1, T_1) \xrightarrow{e:k\triangleleft d:l} (U_1, V_1)$ and $S_2 \xrightarrow{e:k} U_2$. We need to prove $(S_2, T_2) \xrightarrow{e:k\triangleleft d:l} (U_2, V_2)$. As simulation intervals are the same as Φ-intervals, we have the Φ-interval $(S_1, T_1, \mathcal{X}_{S_1,T_1}) \xrightarrow{e:k::d:l}{}^{*} (U_1, V_1, \mathrm{ps}')$, and want to prove that $(S_2, T_2, \mathcal{X}_{S_2,T_2}) \xrightarrow{e:k::d:l} (U_2, V_2, \mathrm{ps}')$. We again split the interval into the instruction-matched transition and a shuffle sequence: $(S_1, T_1, \mathcal{X}_{S_1,S_1}) \xrightarrow{\gamma:\kappa::\delta:\lambda} (W_1, X_1, \mathrm{ps}'') \xrightarrow{e':k'::d':l'}{}^{*} (U_1, V_1, \mathrm{ps}')$. The difficult part is to reproduce the instruction-matched transition from $(S_2, T_2, \mathcal{X}_{S_2,T_2})$. Because $\mathcal{X}_{S_1,T_1}$ is only dependent on program counters, $\mathcal{X}_{S_1,T_1} = \mathcal{X}_{S_2,T_2}$.

Lemma 10. *There is a transition $(S_2, T_2, \mathcal{X}_{S_2,T_2}) \xrightarrow{\gamma:\kappa::\delta:\lambda} (U_2, V_2, \mathrm{ps}')$ in $P :: [P]^{\mathrm{ra}}$.*

We reproduce the second part of the Φ-interval, $(W_1, X_1, \mathrm{ps}''_1) \xrightarrow{e':k'::d':l'}{}^{*} (U_1, V_1, \mathrm{ps}'_1)$. Once more, we perform case distinction on whether d' and e' follow the red or teal paths in Figure 4.

► *No* RB *occurs in* d'. In this case, d' is shuffle-only and $e' = \varepsilon$. Because shuffle semantics are deterministic and all have the same directive $\bullet$, the following lemma is straightforward to show. Together with Lemma 9, the shuffle sequence is reproduced from (W_2, X_2, ps').

Lemma 11. *If $V_1 \equiv V_2$ and $P\ V_1 = sh_{,\mathrm{sc}'}$ with shuffle-only $V_1 \xrightarrow{d:l}{}^{*} X_1$, then $V_2 \xrightarrow{d:l}{}^{*} X_2$.*

Indeed, we complete reproduction in this case: With $(S_2, T_2, \mathrm{ps}) \xrightarrow{\gamma:\kappa::\delta:\lambda} (W_2, X_2, \mathrm{ps}'')$ from Lemma 10 and $(W_2, X_2, \mathrm{ps}''_2) \xrightarrow{\varepsilon:\varepsilon::d':l'}{}^{*} (U_2, V_2, \mathrm{ps}')$ from Lemmas 9 and 11, we constructed $(S_2, T_2, \mathcal{X}_{S_2,T_2}) \xrightarrow{e:k::d:l} (U_2, V_2, \mathrm{ps}')$.

► *Otherwise, let* RB *occur in* d. Then, $(W_1, X_1, \mathrm{ps}'') \xrightarrow{e':k'::d':l'}{}^{*} (U_1, V_1, \mathrm{ps}')$ is actually the teal path $(W_1, X_1, \mathrm{ps}'') \xrightarrow{\varepsilon:\varepsilon::d'':l''}{}^{*} (W'_1, X'_1, \mathrm{ps}''') \xrightarrow{\mathrm{RB}:\mathrm{RB}::\mathrm{RB}:\mathrm{RB}} (U_1, V_1, \mathrm{ps}')$, where d'' is a shuffle

sequence. Same as in the previous case, we can reproduce $(W_2, X_2, \mathrm{ps}'') \xrightarrow{\varepsilon:\varepsilon::d'':l''}{}^{*} (W_2', X_2', \mathrm{ps}''')$. The product can clearly execute $(W_2', X_2', \mathrm{ps}''') \xrightarrow{\mathrm{RB}:\mathrm{RB}::\mathrm{RB}:\mathrm{RB}} (U_2, V_2, \mathrm{ps}')$ because $|W_2'| = |X_2'| = |W_1'| = |X_1'| > 1$. Reproduction of $(S_2, T_2, \mathcal{X}_{S_2,T_2}) \xrightarrow{e:k::d:l} (U_2, V_2, \mathrm{ps}')$ is done. □

Lemma 8

Proof. By construction of the product. Case distinction on $T_1 \xrightarrow{\delta:\lambda} X_1$. Throughout the proof, we assume $\Psi\ T\ a = a'$, $\Psi\ T\ b = b'$, and $\Psi\ T\ c = c'$. Further, we invariantly let $T = T'.t$, $S = S'.s$, $X = X'.x$, $U = U'.u$, Also $P\ T = i'$ and $P\ S = i$. $S\ \Psi_X^{\mathrm{ps}'}\ X$ follows from Lemma 4.

- NOP and ASGN We skip NOP. For ASGN, let $i' = a' := b' \oplus c'_{,\mathrm{sc}'}$. Then $i = a := b \oplus c_{,\mathrm{sc}}$. Clearly there is U with $S \xrightarrow{\bullet:\bullet} U$. Rule POISON-ASGN yields $(S, T, \mathrm{ps}) \xrightarrow{\bullet:\bullet::\bullet:\bullet} (U, X, \mathrm{ps}')$. $U\ \Psi_X^{\mathrm{ps}'}\ X$ follows from construction of Rule POISON-ASGN.
- LOAD For LOAD, let $i' = a' := \mathbf{a}[b']_{,\mathrm{sc}'}$ and $\lambda = \mathrm{LD}\,\mathsf{n}$. We have $\mathrm{ps}\,b \geq \mathrm{wp}$, thus $\rho_T\,b' = 0$ or $\rho_T\,b' = \rho_S\,b$.
  - $\rho_S\,b = \mathsf{m} \in |\mathbf{a}|$ Clearly there is U with $S \xrightarrow{\bullet:\mathrm{LD}\,\mathsf{m}} U$.
    POISON-LOAD-SAFE yields $(S, T, \mathrm{ps}) \xrightarrow{\bullet:\mathrm{LD}\,\mathsf{m}::\bullet:\mathrm{LD}\,\mathsf{n}} (U, X, \mathrm{ps}')$.
  - $\rho_S\,b = \mathsf{m} \notin |\mathbf{a}|$ Clearly there is U with $S \xrightarrow{\mathrm{LU}\,\mathbf{a}\,0:\mathrm{LD}\,\mathsf{m}} U$.
    POISON-LOAD-UNSAFE yields $(S, T, \mathrm{ps}) \xrightarrow{\mathrm{LU}\,\mathbf{a}\,0:\mathrm{LD}\,\mathsf{m}::\bullet:\mathrm{LD}\,\mathsf{n}} (U, X, \mathrm{ps}')$.
- STORE For STORE, let $i' = \mathbf{a}[b'] := c'_{,\mathrm{sc}'}$ and $\lambda = \mathrm{ST}\,\mathsf{n}$. We have $\mathrm{ps}\,b \geq \mathrm{wp}$, thus $\rho_T\,b' = 0$ or $\rho_T\,b' = \rho_S\,b$.
  - $\rho_S\,b = \mathsf{m} \in |\mathbf{a}|$ Clearly there is U with $S \xrightarrow{\bullet:\mathrm{ST}\,\mathsf{m}} U$.
    POISON-STORE-SAFE yields $(S, T, \mathrm{ps}) \xrightarrow{\bullet:\mathrm{ST}\,\mathsf{m}::\bullet:\mathrm{ST}\,\mathsf{n}} (U, X, \mathrm{ps}')$.
  - $\rho_S\,b = \mathsf{m} \notin |\mathbf{a}|$ Clearly there is U with $S \xrightarrow{\mathrm{SU}\,\mathbf{a}\,0:\mathrm{ST}\,\mathsf{m}} U$.
    POISON-STORE-UNSAFE yields $(S, T, \mathrm{ps}) \xrightarrow{\mathrm{SU}\,\mathbf{a}\,0:\mathrm{ST}\,\mathsf{m}::\bullet:\mathrm{ST}\,\mathsf{n}} (U, X, \mathrm{ps}')$.
- LOAD-UNSAFE For LOAD-UNSAFE, let $i' = a' := \mathbf{a}[b']_{,\mathrm{sc}'}$ and $\lambda = \mathrm{LD}\,\mathsf{n}$ and $\delta = \mathrm{LU}\,\mathbf{b}\,\mathsf{m}$. We have $\mathrm{ps}\,b \geq \mathrm{wp}$, thus $\rho_T\,b' = 0$ or $\rho_T\,b' = \rho_S\,b$. Due to $0 \in |\mathbf{a}|$, only $\rho_T\,b' = \rho_S\,b$ needs to be considered. We consider two cases:
  - $\mathbf{b} \neq \mathbf{stk}$ There is U with $S \xrightarrow{\mathrm{LU}\,\mathbf{b}\,\mathsf{m}:\mathrm{LD}\,\mathsf{m}} U$.
    HEALTHY-LOAD-UNSAFE yields $(S, T, \mathrm{ps}) \xrightarrow{\bullet:\mathrm{LD}\,\mathsf{m}::\bullet:\mathrm{LD}\,\mathsf{n}} (U, X, \mathrm{ps}')$.
  - $\mathbf{b} = \mathbf{stk}$ There is U with $S \xrightarrow{\mathrm{LU}\,\mathbf{a}\,0:\mathrm{LD}\,\mathsf{m}} U$.
    POISON-LOAD-STKUNSAFE yields $(S, T, \mathrm{ps}) \xrightarrow{\mathrm{LU}\,\mathbf{a}\,0:\mathrm{LD}\,\mathsf{m}::\bullet:\mathrm{LD}\,\mathsf{n}} (U, X, \mathrm{ps}')$.
- STORE-UNSAFE For STORE-UNSAFE, let $i' = \mathbf{a}[b'] := c'_{,\mathrm{sc}'}$ and $\lambda = \mathrm{ST}\,\mathsf{n}$ and $\delta = \mathrm{SU}\,\mathbf{b}\,\mathsf{m}$. We have $\mathrm{ps}\,b \geq \mathrm{wp}$, thus $\rho_T\,b' = 0$ or $\rho_T\,b' = \rho_S\,b$. Due to $0 \in |\mathbf{a}|$, only $\rho_T\,b' = \rho_S\,b$ needs to be considered. We consider two cases:
  - $\mathbf{b} \neq \mathbf{stk}$ There is U with $S \xrightarrow{\mathrm{SU}\,\mathbf{b}\,\mathsf{m}:\mathrm{ST}\,\mathsf{m}} U$.
    HEALTHY-STORE-UNSAFE yields $(S, T, \mathrm{ps}) \xrightarrow{\bullet:\mathrm{ST}\,\mathsf{m}::\bullet:\mathrm{ST}\,\mathsf{n}} (U, X, \mathrm{ps}')$.
  - $\mathbf{b} = \mathbf{stk}$ There is U with $S \xrightarrow{\mathrm{SU}\,\mathbf{a}\,0:\mathrm{ST}\,\mathsf{m}} U$.
    POISON-STORE-STKUNSAFE yields $(S, T, \mathrm{ps}) \xrightarrow{\mathrm{SU}\,\mathbf{a}\,0:\mathrm{ST}\,\mathsf{m}::\bullet:\mathrm{ST}\,\mathsf{n}} (U, X, \mathrm{ps}')$.
- BRANCH For BRANCH, let $i' = \mathtt{br}\ b'_{,\mathrm{sc}'_t,\mathrm{sc}'_f}$ and $\lambda = \mathrm{BR}\,\mathsf{b}$ and $\delta = \mathrm{BR}$. We have $\mathrm{ps}\,b = \mathsf{h}$, thus $\rho_T\,b' = \rho_S\,b$. Then there is U with $S \xrightarrow{\mathrm{BR}:\mathrm{BR}\,\mathsf{b}} U$. HEALTHY-BRANCH yields $(S, T, \mathrm{ps}) \xrightarrow{\bullet:\mathrm{BR}\,\mathsf{b}::\bullet:\mathrm{BR}\,\mathsf{b}} (U, X, \mathrm{ps}')$.
- SPEC For SPEC, let $i' = \mathtt{br}\ b'_{,\mathrm{sc}'_t,\mathrm{sc}'_f}$ and $\lambda = \mathrm{BR}\,\mathsf{b}$ and $\delta = \mathrm{SP}$. We have $\mathrm{ps}\,b = \mathsf{h}$, thus $\rho_T\,b' = \rho_S\,b$. Then there is U with $S \xrightarrow{\mathrm{SP}:\mathrm{BR}\,\mathsf{b}} U$. HEALTHY-SPEC yields $(S, T, \mathrm{ps}) \xrightarrow{\mathrm{SP}:\mathrm{BR}\,\mathsf{b}::\mathrm{SP}:\mathrm{BR}\,\mathsf{b}} (U, X, \mathrm{ps}')$.
- SFENCE For SFENCE, let $i' = \mathtt{sfence}_{,\mathrm{sc}'}$ and $\lambda = \bullet$ and $\delta = \bullet$. Then $|T| = 1 = |S|$. Thus there is U with $S \xrightarrow{\bullet:\bullet} U$. POISON-SFENCE yields $(S, T, \mathrm{ps}) \xrightarrow{\bullet:\bullet::\bullet:\bullet} (U, X, \mathrm{ps}')$.
- SLH For SLH, let $i' = \mathtt{slh}\ a'_{,\mathrm{sc}'}$ and $\lambda = \bullet$ and $\delta = \bullet$. Thus there is U with $S \xrightarrow{\bullet:\bullet} U$. POISON-SLH yields $(S, T, \mathrm{ps}) \xrightarrow{\bullet:\bullet::\bullet:\bullet} (U, X, \mathrm{ps}')$. □

Lemma 9

Proof. Prove this for a single step, the rest is induction. Let further be $U = U'.u$ and $X = X'.x$. Let further be $\mathsf{ps} = (\mathsf{pr}, \mathsf{pm})$. Let $u = (\mathsf{pc}, \rho, \mu)$, $x = (\mathsf{pc}', \rho', \mu')$, $S\ \Psi_X^{\mathsf{ps}}\ X$ and $[P]^{\mathsf{ra}}\ \mathsf{pc}' = si\mathsf{r}'$. We do case distinction on si.

- $a' := b'_{,\mathsf{r}'}$ Then, $x = (\mathsf{pc}', \rho', \mu') \xrightarrow{\bullet:\bullet} (\mathsf{r}', \rho'', \mu') = v$ with $\rho'[a' \mapsto \rho'\ b'] = \rho''$. By shuffle conformity, we know that there is b with $\Psi\ \mathsf{pc}'\ b = b'$. Further, we know that $\Psi\ \mathsf{r}'$ coincides with $\Psi\ \mathsf{pc}'$ except for b. Also, $\mathsf{ps} = \mathsf{ps}'$. By induction, $S\ \Psi_X^{\mathsf{ps}}\ X$, we have $[\![\Psi\ \mathsf{pc}'\ b]\!]_v = \rho''\ a' = \rho'\ b' = [\![\Psi\ \mathsf{pc}'\ b]\!]_x = \rho\ b$ as required for $S\ \Psi_X^{\mathsf{ps}}\ V$. POISON-MOVE yields the transition.
- $a' := \mathbf{stk}[\mathsf{l}]_{,\mathsf{r}'}$ Then, $x = (\mathsf{pc}', \rho', \mu') \xrightarrow{\bullet:\mathrm{LD}\,\mathsf{l}} (\mathsf{r}', \rho'', \mu') = v$ with $\rho'[a' \mapsto \mu\ \mathbf{stk}\ \mathsf{l}] = \rho''$. By shuffle conformity, we know that there is b with $\Psi\ \mathsf{pc}'\ b = \mathsf{l}$. Further, we know that $\Psi\ \mathsf{r}'$ coincides with $\Psi\ \mathsf{pc}'$ except for b, where $\Psi\ \mathsf{r}'\ b = a'$. Also, $\mathsf{ps} = \mathsf{ps}'$. By induction, $S\ \Psi_X^{\mathsf{ps}}\ X$, we have $[\![\Psi\ \mathsf{pc}'\ b]\!]_v = \rho''\ a' = \mu'\ \mathbf{stk}\ \mathsf{l} = [\![\Psi\ \mathsf{pc}'\ b]\!]_x = \rho\ b$ as required for $S\ \Psi_X^{\mathsf{ps}}\ V$. POISON-FILL yields the transition.
- $\mathbf{stk}[\mathsf{l}] := b'_{,\mathsf{r}'}$ Then, $x = (\mathsf{pc}', \rho', \mu') \xrightarrow{\bullet:\mathrm{ST}\,\mathsf{l}} (\mathsf{r}', \rho', \mu'') = v$ with $\mu'[(\mathbf{stk}, \mathsf{l}) \mapsto \rho'\ b'] = \mu''$. By shuffle conformity, we know that there is b with $\Psi\ \mathsf{pc}'\ b = b'$. Further, we know that $\Psi\ \mathsf{r}'$ coincides with $\Psi\ \mathsf{pc}'$ except for b, where $\Psi\ \mathsf{r}'\ b = \mathsf{l}$. Also, $\mathsf{ps} = \mathsf{ps}'$. By induction, $S\ \Psi_X^{\mathsf{ps}}\ X$, we have $[\![\Psi\ \mathsf{pc}'\ b]\!]_v = \mu''\ \mathbf{stk}\ \mathsf{l} = \rho'\ b' = [\![\Psi\ \mathsf{pc}'\ b]\!]_x = \rho\ b$ as required for $S\ \Psi_X^{\mathsf{ps}}\ V$. POISON-SPILL yields the transition.
- $\mathtt{sfence}_{,\mathsf{r}'}$ Then $|X'| = 0$ and $x = (\mathsf{pc}', \rho', \mu') \xrightarrow{\bullet:\bullet} (\mathsf{r}', \rho', \mu') = v$. Also, $\mathsf{ps} = \mathsf{ps}'$. $s\ \Psi_X^{\mathsf{ps}}\ x$ implies $s\ \Psi_X^{\mathsf{ps}}\ v = V$. POISON-SHUFFLE-SFENCE yields the transition.
- $\mathtt{slh}\ a'_{,\mathsf{r}'}$ If $|X'| = 0$, this case is fully analogue to the previous one. Thus, let $|X'| \geq 1$. Then, $x = (\mathsf{pc}', \rho', \mu') \xrightarrow{\bullet:\bullet} (\mathsf{r}', \rho'', \mu') = v$ with $\rho'[a' \mapsto 0] = \rho''$. By shuffle conformity, we know that $\Psi\ \mathsf{pc}' = \Psi\ \mathsf{r}'$. Also $\mathsf{ps}' = (\mathsf{pr}[a \mapsto \mathsf{wp}], \mathsf{pm})$. Indeed, $S\ \Psi_X^{\mathsf{ps}}\ X$ and $[\![\Psi\ \mathsf{pc}'\ a]\!]_v = 0$ yields $S\ \Psi_X^{\mathsf{ps}}\ V$. POISON-SHUFFLE-SLH yields the transition. □

Lemma 10

Proof. By case distinction on $(S_1, T_1, \mathsf{ps}) \xrightarrow{\gamma:\kappa::\delta:\lambda} (U_1, V_1, \mathsf{ps}')$. For all cases, and $i \in \{1, 2\}$: Let $T_i = T_i'.t_i$, $S_i = S_i'.s_i$, $V_i = V_i'.v_i$, and $U_i = U_i'.u_i$, and $t_1 \equiv \mathsf{pc}' \equiv t_2$ as well as $s_1 \equiv \mathsf{pc} \equiv s_2$. Further, let $[P]^{\mathsf{ra}}\ \mathsf{pc}' = i'$ and $P\ \mathsf{pc} = i$, and assume $\Psi\ \mathsf{pc}'\ a = a'$, $\Psi\ \mathsf{pc}'\ b = b'$, and $\Psi\ \mathsf{pc}'\ c = c'$ if they occur in i and i'. Finally, let $\mathcal{X}_{S_1,T_1} = \mathcal{X}_{S_2,T_2} = \mathsf{P.ps}$. Please note, that ps' is not dependent on values: Whenever the same rule in $P :: [P]^{\mathsf{ra}}$ is executed, then the same ps' is obtained. The arguments for equality of ps' are thus skipped.

- HEALTHY-STORE-SAFE In this case, $i = \mathbf{a}[b] := c_{,\mathsf{sc}}$ and $i' = \mathbf{a}[b'] := c'_{,\mathsf{r}'}$. The presumptions for transition $(S_1, T_1, \mathsf{ps}) \xrightarrow{\gamma:\kappa::\delta:\lambda} (U_1, V_1, \mathsf{ps}')$ yield $t_1 \xrightarrow{\bullet:\mathrm{ST}\,\mathsf{n}} v_1$, and $s_1 \xrightarrow{\bullet:\mathrm{ST}\,\mathsf{n}} u_1$, and $\mathsf{ps}\ b = \mathsf{h}$. Our assumption is that $s_2 \xrightarrow{\bullet:\mathrm{ST}\,\mathsf{n}} u_2$ can be executed. The leaked address is $\mathsf{n} = \rho_{s_2}\ b \in |\mathbf{a}|$. Due to $\mathsf{ps}\ b = \mathsf{h}$ and $S_2\ \Psi_X^{\mathsf{ps}}\ T_2$, $\rho_{t_2}\ b' = \rho_{s_2}\ b = \mathsf{n}$. This suffices for $T_2 \xrightarrow{\bullet:\mathrm{ST}\,\mathsf{n}} V_2$, and in turn for $(S_2, T_2, \mathsf{ps}) \xrightarrow{\bullet:\mathrm{ST}\,\mathsf{n}::\bullet:\mathrm{ST}\,\mathsf{n}} (U_2, V_2, \mathsf{ps}')$.
- POISON-LOAD-STKUNSAFE Then, $i = a := \mathbf{a}[b]_{,\mathsf{sc}}$ and $i = a := \mathbf{a}[b]_{,\mathsf{sc}}$. The presumptions yield $t_1 \xrightarrow{\mathrm{LU}\,\mathbf{stk}\,\mathsf{m}:\mathrm{LD}\,\mathsf{n}} v_1$, and $s_1 \xrightarrow{\mathrm{LU}\,\mathbf{b}\,\mathsf{l}:\mathrm{LD}\,\mathsf{n}} u_1$, and $\mathsf{ps}\ b = \mathsf{h}$. Our assumption is $s_2 \xrightarrow{\mathrm{LU}\,\mathbf{b}\,\mathsf{l}:\mathrm{LD}\,\mathsf{n}} u_2$. Due to $\mathsf{ps}\ b = \mathsf{h}$, $\rho_{t_1}\ b' = \rho_{s_1}\ b = \mathsf{n} = \rho_{s_2}\ b = \rho_{t_2}\ b' \notin |\mathbf{a}|$. That justifies $T_2 \xrightarrow{\mathrm{LU}\,\mathbf{stk}\,\mathsf{m}:\mathrm{LD}\,\mathsf{n}} V_2$. That suffices for $(S_2, T_2, \mathsf{ps}) \xrightarrow{\mathrm{LU}\,\mathbf{b}\,\mathsf{l}:\mathrm{LD}\,\mathsf{n}::\mathrm{LU}\,\mathbf{stk}\,\mathsf{m}:\mathrm{LD}\,\mathsf{n}} (U_2, V_2, \mathsf{ps}')$.
- POISON-NOP and POISON-ASGN Trivial.
- POISON-STORE-STKUNSAFE Then, $i = \mathbf{a}[b] := c_{,\mathsf{sc}}$, $t_1 \xrightarrow{\mathrm{SU}\,\mathbf{stk}\,\mathsf{m}:\mathrm{ST}\,\mathsf{n}} v_1$, and $s_1 \xrightarrow{\mathrm{SU}\,\mathbf{b}\,\mathsf{l}:\mathrm{ST}\,\mathsf{n}} u_1$, and $\mathsf{ps}\ b = \mathsf{h}$. Our assumption is $s_2 \xrightarrow{\mathrm{SU}\,\mathbf{b}\,\mathsf{l}:\mathrm{ST}\,\mathsf{n}} u_2$. Due to $\mathsf{ps}\ b = \mathsf{h}$, $\rho_{t_1}\ b' = \rho_{s_1}\ b = \mathsf{n} = \rho_{s_2}\ b = \rho_{t_2}\ b' \notin |\mathbf{a}|$. That justifies $T_2 \xrightarrow{\mathrm{LU}\,\mathbf{stk}\,\mathsf{m}:\mathrm{LD}\,\mathsf{n}} V_2$. That suffices for $(S_2, T_2, \mathsf{ps}) \xrightarrow{\mathrm{SU}\,\mathbf{b}\,\mathsf{l}:\mathrm{ST}\,\mathsf{n}::\mathrm{LU}\,\mathbf{stk}\,\mathsf{m}:\mathrm{LD}\,\mathsf{n}} (U_2, V_2, \mathsf{ps}')$.

- POISON-LOAD-SAFE Then, $i = a := \mathbf{a}[b]_{\rangle sc}$, $t_1 \xrightarrow{\bullet:\text{LD}\,0} v_1$, and $s_1 \xrightarrow{\bullet:\text{LD}\,k} u_1$, and $\text{ps}\,b = \text{wp}$. Further $s_2 \xrightarrow{\bullet:\text{LD}\,k} u_2$. Due to $\text{ps}\,b = \text{wp}$, $\rho_{T_2}\,b' = 0 \in |\mathbf{a}|$ justifies $T_2 \overset{\bullet:\text{LD}\,0}{\dashrightarrow} V_2$. That suffices for $(S_2, T_2, \text{ps}) \overset{\bullet:\text{LD}\,k::\bullet:\text{LD}\,0}{\dashrightarrow} (U_2, V_2, \text{ps}')$.
- POISON-LOAD-UNSAFE Then, $i = a := \mathbf{a}[b]_{\rangle sc}$, $t_1 \xrightarrow{\bullet:\text{LD}\,0} v_1$, and $s_1 \xrightarrow{\text{LU}\,\mathbf{b}\,l:\text{LD}\,k} u_1$, and $\text{ps}\,b = \text{wp}$. Further $s_2 \xrightarrow{\text{LU}\,\mathbf{b}\,l:\text{LD}\,k} u_2$. Due to $\text{ps}\,b = \text{wp}$, $\rho_{T_2}\,b' = 0 \in |\mathbf{a}|$ justifies $T_2 \overset{\bullet:\text{LD}\,0}{\dashrightarrow} V_2$. That suffices for $(S_2, T_2, \text{ps}) \overset{\text{LU}\,\mathbf{b}\,l:\text{LD}\,k::\bullet:\text{LD}\,0}{\dashrightarrow} (U_2, V_2, \text{ps}')$.
- HEALTHY-LOAD-SAFE Then, $i = a := \mathbf{a}[b]_{\rangle sc}$, $t_1 \xrightarrow{\bullet:\text{LD}\,n} v_1$, and $s_1 \xrightarrow{\bullet:\text{LD}\,n} u_1$, and $\text{ps}\,b = \text{h}$. Further $s_2 \xrightarrow{\bullet:\text{LD}\,n} u_2$. Due to $\text{ps}\,b = \text{h}$, $\rho_{T_1}\,b' = \rho_{S_1}\,b = \text{n} = \rho_{S_2}\,b = \rho_{T_2}\,b' \in |\mathbf{a}|$ justifies $T_2 \overset{\bullet:\text{LD}\,n}{\dashrightarrow} V_2$. That suffices for $(S_2, T_2, \text{ps}) \overset{\bullet:\text{LD}\,n::\bullet:\text{LD}\,n}{\dashrightarrow} (U_2, V_2, \text{ps}')$.
- HEALTHY-LOAD-UNSAFE Then, $i = a := \mathbf{a}[b]_{\rangle sc}$, $t_1 \xrightarrow{\text{LU}\,\mathbf{b}\,m:\text{LD}\,n} v_1$, and $s_1 \xrightarrow{\text{LU}\,\mathbf{b}\,m:\text{LD}\,n} u_1$, and $\text{ps}\,b = \text{h}$. Further $s_2 \xrightarrow{\text{LU}\,\mathbf{b}\,m:\text{LD}\,n} u_2$. Due to $\text{ps}\,b = \text{h}$, $\rho_{T_1}\,b' = \rho_{S_1}\,b = \text{n} = \rho_{S_2}\,b = \rho_{T_2}\,b' \notin |\mathbf{a}|$ justifies $T_2 \overset{\text{LU}\,\mathbf{b}\,m:\text{LD}\,n}{\dashrightarrow} V_2$. That suffices for $(S_2, T_2, \text{ps}) \overset{\text{LU}\,\mathbf{b}\,m:\text{LD}\,n::\text{LU}\,\mathbf{b}\,m:\text{LD}\,n}{\dashrightarrow} (U_2, V_2, \text{ps}')$.
- POISON-STORE-SAFE Then, $i = \mathbf{a}[b] := c_{\rangle sc}$, $t_1 \xrightarrow{\bullet:\text{ST}\,0} v_1$, and $s_1 \xrightarrow{\bullet:\text{ST}\,k} u_1$, and $\text{ps}\,b = \text{wp}$. Further $s_2 \xrightarrow{\bullet:\text{ST}\,k} u_2$. Due to $\text{ps}\,b = \text{wp}$, $\rho_{T_2}\,b' = 0 \in |\mathbf{a}|$ justifies $T_2 \overset{\bullet:\text{ST}\,0}{\dashrightarrow} V_2$. That suffices for $(S_2, T_2, \text{ps}) \overset{\bullet:\text{ST}\,k::\bullet:\text{ST}\,0}{\dashrightarrow} (U_2, V_2, \text{ps}')$.
- POISON-STORE-UNSAFE Then, $i = \mathbf{a}[b] := c_{\rangle sc}$, $t_1 \xrightarrow{\bullet:\text{ST}\,0} v_1$, and $s_1 \xrightarrow{\text{SU}\,\mathbf{b}\,l:\text{ST}\,k} u_1$, and $\text{ps}\,b = \text{wp}$. Further $s_2 \xrightarrow{\text{SU}\,\mathbf{b}\,l:\text{ST}\,k} u_2$. Due to $\text{ps}\,b = \text{wp}$, $\rho_{T_2}\,b' = 0 \in |\mathbf{a}|$ justifies $T_2 \overset{\bullet:\text{ST}\,0}{\dashrightarrow} V_2$. That suffices for $(S_2, T_2, \text{ps}) \overset{\text{SU}\,\mathbf{b}\,l:\text{ST}\,k::\bullet:\text{ST}\,0}{\dashrightarrow} (U_2, V_2, \text{ps}')$.
- HEALTHY-STORE-UNSAFE Then, $i = \mathbf{a}[b] := c_{\rangle sc}$, $t_1 \xrightarrow{\text{SU}\,\mathbf{b}\,m:\text{ST}\,n} v_1$, and $s_1 \xrightarrow{\text{SU}\,\mathbf{b}\,m:\text{ST}\,n} u_1$, and $\text{ps}\,b = \text{h}$. Further $s_2 \xrightarrow{\text{SU}\,\mathbf{b}\,m:\text{ST}\,n} u_2$. Due to $\text{ps}\,b = \text{h}$, $\rho_{t_1}\,b' = \rho_{s_1}\,b = \text{n} = \rho_{s_2}\,b = \rho_{t_2}\,b' \notin |\mathbf{a}|$ justifies $T_2 \overset{\text{SU}\,\mathbf{b}\,m:\text{ST}\,n}{\dashrightarrow} V_2$. That suffices for $(S_2, T_2, \text{ps}) \overset{\text{SU}\,\mathbf{b}\,m:\text{ST}\,n::\text{SU}\,\mathbf{b}\,m:\text{ST}\,n}{\dashrightarrow} (U_2, V_2, \text{ps}')$.
- HEALTHY-BRANCH Then, $i = \texttt{br}\ b_{\rangle sc_t, sc_f}$, $t_1 \xrightarrow{\text{BR:BR}\,b} v_1$, and $s_1 \xrightarrow{\text{BR:BR}\,b} u_1$, and $\text{ps}\,b = \text{h}$. Further $s_2 \xrightarrow{\text{BR:BR}\,b} u_2$. Due to $\text{ps}\,b = \text{h}$, $(\rho_{t_1}\,b' = 0) = (\rho_{s_1}\,b = 0) = \text{b} = (\rho_{s_2}\,b = 0) = (\rho_{t_2}\,b' = 0)$ justifies $T_2 \overset{\text{BR:BR}\,b}{\dashrightarrow} V_2$. That suffices for $(S_2, T_2, \text{ps}) \overset{\text{BR:BR}\,b::\text{BR:BR}\,b}{\dashrightarrow} (U_2, V_2, \text{ps}')$.
- HEALTHY-SPEC Then, $i = \texttt{br}\ b_{\rangle sc_t, sc_f}$, $T_1 \overset{\text{SP:BR}\,b}{\dashrightarrow} T_1.v_1$, and $S_1 \overset{\text{SP:BR}\,b}{\dashrightarrow} S_1.u_1$, and $\text{ps}\,b = \text{h}$. Further $S_2 \overset{\text{SP:BR}\,b}{\dashrightarrow} S_2.u_2$. Due to $\text{ps}\,b = \text{h}$, $(\rho_{T_1}\,b' = 0) = (\rho_{S_1}\,b = 0) = \text{b} = (\rho_{S_2}\,b = 0) = (\rho_{T_2}\,b' = 0)$ justifies $T_2 \overset{\text{SP:BR}\,b}{\dashrightarrow} T_2.v_2 = V_2$. That suffices for $(S_2, T_2, \text{ps}) \overset{\text{SP:BR}\,b::\text{SP:BR}\,b}{\dashrightarrow} (U_2, V_2, \text{ps}')$.
- POISON-SFENCE Then, $i = \texttt{sfence}_{\rangle sc}$, $t_1 \overset{\bullet:\bullet}{\dashrightarrow} v_1$, and $s_1 \overset{\bullet:\bullet}{\dashrightarrow} u_1$, and $|T_1| = |S_1| = |S_2| = |T_2| = 1$. Further $s_2 \overset{\bullet:\bullet}{\dashrightarrow} u_2$. $|T_2| = 1$ justifies $t_2 \overset{\bullet:\bullet}{\dashrightarrow} v_2$. That suffices for $(S_2, T_2, \text{ps}) \overset{\bullet:\bullet::\bullet:\bullet}{\dashrightarrow} (U_2, V_2, \text{ps}')$.
- POISON-SLH Then, $i = \texttt{slh}\ a_{\rangle sc}$, $T_1 \overset{\bullet:\bullet}{\dashrightarrow} V_1$, and $S_1 \overset{\bullet:\bullet}{\dashrightarrow} U_1$. Further $S_2 \overset{\bullet:\bullet}{\dashrightarrow} U_2$. Semantics yield an appropriate V_2 with $T_2 \overset{\bullet:\bullet}{\dashrightarrow} V_2$. That suffices for $(S_2, T_2, \text{ps}) \overset{\bullet:\bullet::\bullet:\bullet}{\dashrightarrow} (U_2, V_2, \text{ps}')$.
- POISON-ROLLBACK Then, $T_1 \overset{\text{RB:RB}}{\dashrightarrow} T_1' = V_1$, and $S_1 \overset{\text{RB:RB}}{\dashrightarrow} S_1' = U_1$. Further $S_2 \overset{\text{RB:RB}}{\dashrightarrow} S_2' = U_2$. Semantics yield $T_2 \overset{\text{RB:RB}}{\dashrightarrow} T_2' = V_2$. That suffices for $(S_2, T_2, \text{P.ps}) \overset{\text{RB:RB::RB:RB}}{\dashrightarrow} (U_2, V_2, \text{P})$.
- POISON-FILL, POISON-SPILL, POISON-MOVE, POISON-SHUFFLE-SFENCE, and POISON-SHUFFLE-SLH Not applicable due to definition of $\prec$, $T_1 \prec S_1$ makes it impossible for T_1 to be at a shuffle program counter. □

Lemma 11

Proof. We do the induction step by case distinction on $V_1 \overset{\bullet:\lambda}{\dashrightarrow} X_1$. Let $i = P\ V_1 = P\ V_2$.

- SLH *and* MOVE Then, $\lambda = \bullet$ and there is $V_2 \overset{\bullet:\bullet}{\dashrightarrow} X_2$ by definition of semantics.
- FILL Then, $\lambda = \text{LD}\,l$ and there is $V_2 \overset{\bullet:\text{LD}\,l}{\dashrightarrow} X_2$ by definition of semantics.
- SPILL Then, $\lambda = \text{ST}\,l$ and there is $V_2 \overset{\bullet:\text{ST}\,l}{\dashrightarrow} X_2$ by definition of semantics.
- SFENCE Then, $|V_1| = 1 = |V_2|$. By definition, $V_2 \overset{\bullet:\bullet}{\dashrightarrow} X_2$. □

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