Speculative Taint Tracking (STT): A Formal Analysis

Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, Christopher W. Fletcher

Contents

1 Overview

Speculative Taint Tracking (STT) [4] is a comprehensive hardware protection, which safely executes and selectively forwards the results of speculative instructions that read potential secrets. STT ensures that the forwarded results never reach potential covert channels. In this document, we study security guarantees that STT provides under the assumption of the *Spectre attack model*, i.e., when the results of speculative instructions no longer require protection once all older control-flow instructions have resolved. We formally prove that STT in the Spectre attack model enforces a novel notion of non-interference [1] appropriate for speculative execution attacks. (The formal proof for the futuristic model is ongoing work.)

We formally model processors as state machines. We define an STT machine that is a detailed model of a speculative out-of-order processor with STT. In addition to registers and memory, its state includes hardware structures such as branch predictors, reorder buffer (ROB), load-store queue (LSQ), etc. Its state also includes taint bits for registers. A *processor logic* defines how the state changes at every cycle. In each cycle, it performs *events* that modify the machine's state. These events model microarchitectural events such as instruction fetch, execution by a functional unit, squashes, retirement, tainting/untainting, etc. A subset of these events models observable microarchitectural events, e.g., transmitters in [4]. The STT machine models the protections described in [4], e.g., delaying execution of tainted transmit instructions.

We show that the STT machine provides the following non-interference security guarantee: at each step of its execution, the value of a *doomed* register, that is, written to by a speculative access instruction that is bound to squash, does not influence the future of the execution. The key challenge is that when an instruction executes, we do not know whether it is going to squash or not. We address this by considering a simple in-order processor model, which we use to verify the STT machine's branch predictions against their true outcome, obtained from the in-order processor. Specifically, at each step of the STT machine's execution of a program, in our formal analysis we maintain an auxiliary bit of state, $\mathsf{mispredicted}$, that is only set to true if the prediction of one of the preceding branches differs from the outcome of the corresponding instruction in the in-order execution of the program.

With the way to identify whether the STT machine misspeculates at each point of the execution, we are able to distinguish doomed registers: a register r is doomed in a state σ with a given $\mathsf{mispredicted}$ flag, if it gets tainted in the shadow of what the in-order processor identifies as a misprediction, i.e., while $\mathsf{mispredicted} = \mathsf{true}$. For each register, we also maintain auxiliary state indicating whether it is doomed. We refer to the STT machine state coupled with its auxiliary state as an *extended state*. Given two extended STT states, κ_1 and κ_2, we say that $\kappa_1 \approx \kappa_2$ holds if κ_1 and κ_2 only differ by values of doomed registers.

We prove the following theorem, which states that values of doomed registers neither influence which events the STT machine executes at each cycle nor get leaked into the rest of the state by those events. We parameterize the theorem by an *observability function* view, which models the adversary's view [3], i.e., it projects event traces onto the parts the adversary can observe. The theorem holds in particular for a strong adversary that observes the instructions fetched, when and which functional units are busy (i.e., resource usage and port contention), and the target address of every cache/memory access.

Theorem 1. *At any cycle t, given two extended states κ_1 and κ_2 such that $\kappa_1 \approx \kappa_2$ holds, if $\mathcal{T}_1$ and $\mathcal{T}_2$ are the sequences of instruction events the STT processor logic performs at the cycle t from κ_1 and κ_2 respectively, then the following holds:*

(a) $\mathsf{view}(\mathcal{T}_1) = \mathsf{view}(\mathcal{T}_2)$ *holds, and*

(b) for the extended states κ_1' and κ_2' resulting from executing $\mathcal{T}_1$ and $\mathcal{T}_2$ respectively, $\kappa_1' \approx \kappa_2'$ holds.

The theorem proves that $\kappa_1 \approx \kappa_2$ is an invariant preserved by each cycle of the machine execution. Since this invariant is inductive, we obtain the following corollary: at any cycle t in any extended state κ_1, changes to doomed registers do not influence the future of the execution of the machine. In particular, they never influence the program counter's value, which is sufficient to eliminate traditional implicit covert channels.

This document is structured as follows. In Section 2, we introduce basic definitions of the instruction set and semantics of instruction execution on the in-order processor. In Section 3, we present a detailed model of a speculative out-of-order processor with STT. In Section 4, we formally analyze its security, to which end we formulate and prove the non-interference property of the STT processor.

2 Preliminaries

2.1 Instruction Set

We support a simple RISC-style instruction set architecture (ISA). Instructions manipulate a finite set of logical register identifiers RegID, ranged over by r (we also use r_a, r_b, r_c, r_d and r_v throughout the document). We assume the set of instructions Instr, ranged over by θ, which are tuples consisting of the instruction name and a sequence of its arguments given either as register identifiers or as integer values. We assume only five kinds of instructions in Instr:

- $\langle \textbf{immed}\ r_d, k\rangle$ assigns integer value k to register r_d
- $\langle \textbf{op}\ r_d, r_a, r_b\rangle$ is a binary arithmetic operation, with r_a and r_b as inputs, r_d as output
- $\langle \textbf{branch}\ r_c, r_d\rangle$ is a branch instruction, which sets program counter to the value of register r_d if the value of register r_c is true
- $\langle \textbf{load}\ r_d, r_a\rangle$ loads value from memory address value of register r_a to register r_d
- $\langle \textbf{store}\ r_a, r_v\rangle$ stores the value of register r_v to memory address value of register r_a

A program $P \in \mathsf{Instr}^*$ is a finite sequence of instructions. We refer to i-th instruction in the sequence as $P[i]$. We refer to operands of instructions as *input* and *output* registers. Thus, r_d is the output register of **immed**, **op** and **load** in the description above, and the rest of operands of each instruction are input registers.

2.2 The In-Order Processor Model

State. The in-order processor state $\Sigma = (\mathsf{pc}, \mathsf{mem}, \mathsf{reg}) \in \mathsf{State}_{\mathsf{InO}}$ is a tuple consisting of the following three components:

1. A program counter $\mathsf{pc} \in \mathbb{N}$, which is an integer corresponding to the address of the next instruction to execute.
2. A memory $\mathsf{mem} : \mathbb{N} \to \mathbb{Z}$, which maps memory addresses to integer values.
3. A register file $\mathsf{reg} : \mathsf{RegID} \to \mathbb{Z}$, which is a function mapping register identifiers to integer values denoting the content of a given register.

We also assume an initial state Σ_{init}, which is any state with the program counter of 0.

Semantics of Instructions Execution. Whenever the in-order processor executes an instruction and changes a state, we say that it performs a corresponding *architectural event*[1]: Immediate, Arithmetic, Branch, Load and Store, which altogether form the set $\mathsf{Events}_{\mathsf{InO}}$. We specify semantics of the events with the help of a transition relation $\to \in (\mathsf{State}_{\mathsf{InO}} \times \mathsf{Events}_{\mathsf{InO}} \times \mathsf{State}_{\mathsf{InO}})$ defined by the clauses in Figure 1.

Intuitively, the transition relation defines the in-order execution of the instructions by describing how corresponding events update the state. Even though the state updates are standard, in the following we present them in detail to explain the notation. We will call the list of constraints over the bar in each rule *premises* of the event, and the description of a state update under the bar a *transition* of the event.

We introduce the following bit of notation to describe updates to the register file and memory. Given a function F, we let $F[y \mapsto v]$ be the function such that for every argument x different from y, it returns $F(x)$, and for y it returns v:

$$F[y \mapsto v] \triangleq \lambda x. \begin{cases} F(x), & \text{if } x \neq y \\ v, & \text{otherwise} \end{cases}$$

Thus, for instance, loading a value v into a register r with the current register file reg updates the latter to $\mathsf{reg}[r \mapsto v]$.

The Immediate event occurs whenever the instruction $P[\mathsf{pc}]$ in the program at the program counter is $\langle \textbf{immed}\ r_d, k\rangle$ and simply puts the value k into the output register r_d. To that end, the event changes the state from $(\mathsf{pc}, \mathsf{mem}, \mathsf{reg})$ to $(\mathsf{pc}+1, \mathsf{mem}, \mathsf{reg}[r_d \mapsto k])$, provided that there exist k and r_d such that $P[\mathsf{pc}] = \langle \textbf{immed}\ r_d, k\rangle$ hold.

[1] We separate the notions of an instruction and an event to unify formalisms with those of out-of-order processors, where multiple events correspond to a single instruction

IMMEDIATE:
$$\frac{P[\mathsf{pc}] = \langle \textbf{immed } r_d, k\rangle}{\{\mathsf{pc}, \mathsf{mem}, \mathsf{reg}\} \longrightarrow \{\mathsf{pc}+1, \mathsf{mem}, \mathsf{reg}[r_d \mapsto k]\}}$$

ARITHMETIC:
$$\frac{P[\mathsf{pc}] = \langle \textbf{op } r_d, r_a, r_b\rangle}{\{\mathsf{pc}, \mathsf{mem}, \mathsf{reg}\} \longrightarrow \{\mathsf{pc}+1, \mathsf{mem}, \mathsf{reg}[r_d \mapsto \textbf{op}(\mathsf{reg}(r_a), \mathsf{reg}(r_b))],\}}$$

BRANCH:
$$\frac{P[\mathsf{pc}] = \langle \textbf{branch } r_c, r_d\rangle \qquad \mathsf{pc}' = (\textbf{if } \mathsf{reg}(r_c) \textbf{ then } \mathsf{reg}(r_d) \textbf{ else } \mathsf{pc}+1)}{\{\mathsf{pc}, \mathsf{mem}, \mathsf{reg}\} \longrightarrow \{\mathsf{pc}', \mathsf{mem}, \mathsf{reg}\}}$$

LOAD:
$$\frac{P[\mathsf{pc}] = \langle \textbf{load } r_d, r_a\rangle}{\{\mathsf{pc}, \mathsf{mem}, \mathsf{reg}\} \longrightarrow \{\mathsf{pc}+1, \mathsf{mem}, \mathsf{reg}[r_d \mapsto \mathsf{mem}(\mathsf{reg}(r_a))]\}}$$

STORE:
$$\frac{P[\mathsf{pc}] = \langle \textbf{store } r_a, r_v\rangle}{\{\mathsf{pc}, \mathsf{mem}, \mathsf{reg}\} \longrightarrow \{\mathsf{pc}+1, \mathsf{mem}[\mathsf{reg}(r_a) \mapsto \mathsf{reg}(r_v)], \mathsf{reg}\}}$$

Figure 1: Transition rules of the in-order execution of architectural events. For convenience, we put the name of each event over the corresponding rule.

Algorithm 1: Processor algorithm

```
1 Function InO_Processor(P):
2     Σ <- Σ_init; // initial state
3     t <- 0; // initial cycle counter
4     halt <- false;
5     while ¬halt do
6         (Σ, halt) <- InO_Logic(P, Σ, t);
7         t <- t + 1;
8     end
```

The ARITHMETIC event occurs whenever the instruction $P[\mathsf{pc}]$ in the program at the program counter is $\langle \textbf{op } r_d, r_a, r_b\rangle$. It puts the result of the arithmetic operation $\textbf{op}(\mathsf{reg}(r_a), \mathsf{reg}(r_b))$ into the output register r_d. To that end, the event changes the state from $(\mathsf{pc}, \mathsf{mem}, \mathsf{reg})$ to $(\mathsf{pc}+1, \mathsf{mem}, \mathsf{reg}[r_d \mapsto op(\mathsf{reg}(r_a), \mathsf{reg}(r_b))])$, provided that there exist r_a, r_b and r_d such that $P[\mathsf{pc}] = \langle \textbf{op } r_d, r_a, r_b\rangle$ holds.

The BRANCH event occurs whenever the instruction $P[\mathsf{pc}]$ in the program at the program counter is $\langle \textbf{branch } r_c, r_d\rangle$, and either jumps to an instruction with a program counter $\mathsf{reg}(r_d)$ or to the next instruction, depending on whether $\mathsf{reg}(r_c)$ holds. To that end, the event changes the state from $(\mathsf{pc}, \mathsf{mem}, \mathsf{reg})$ to $(\mathsf{pc}', \mathsf{mem}, \mathsf{reg})$, provided that the instruction at the program counter pc is $P[\mathsf{pc}] = \langle \textbf{branch } r_c, r_d\rangle$, and pc' is either $\mathsf{reg}(r_d)$, if the condition $\mathsf{reg}(r_c)$ holds, or $\mathsf{pc}+1$, otherwise.

The LOAD event occurs whenever the instruction $P[\mathsf{pc}]$ in the program at the program counter is $\langle \textbf{load } r_d, r_a\rangle$, and simply loads the value $\mathsf{mem}(\mathsf{reg}(r_a))$ stored in memory by the address $\mathsf{reg}(r_a)$ into the output register r_d. To that end, the event changes the state from $(\mathsf{pc}, \mathsf{mem}, \mathsf{reg})$ to $(\mathsf{pc}+1, \mathsf{mem}, \mathsf{reg}[r_d \mapsto \mathsf{mem}(\mathsf{reg}(r_a))])$, provided that there exist r_d and r_a such that $P[\mathsf{pc}] = \langle \textbf{load } r_d, r_a\rangle$ hold.

The STORE event occurs whenever the instruction $P[\mathsf{pc}]$ in the program at the program counter is $\langle \textbf{store } r_a, r_v\rangle$, and stores the value $\mathsf{reg}(r_v)$ by the address $\mathsf{reg}(r_a)$. To that end, the event changes the state from $(\mathsf{pc}, \mathsf{mem}, \mathsf{reg})$ to $(\mathsf{pc}+1, \mathsf{mem}[\mathsf{reg}(r_a) \mapsto \mathsf{reg}(r_v)], \mathsf{reg})$, provided that there exist r_a and r_v such that $P[\mathsf{pc}] = \langle \textbf{store } r_a, r_v\rangle$ holds.

Given an instruction θ, we let $\mathsf{matching\_event}(\theta)$ denote the event e corresponding to the type of the instruction θ. We also let a function $\mathsf{perform}(\Sigma, e)$ return Σ' corresponding to applying the transition rule for executing the event e from the state Σ, i.e., such that $\{\Sigma\} \xrightarrow{e} \{\Sigma'\}$ holds.

Processor Model. We model processors as state machines represented by $\mathsf{InO\_Processor}(P)$ in Algorithm 1. A processor takes a program P as an input, a state Σ, a cycle counter t and a termination flag halt. The processor starts with an initial state, a cycle counter initialized to 0 and an unset termination flag. At each further step, the processor updates the aforementioned structures with the help of *processor logic*, $\mathsf{InO\_Logic}$ from Algorithm 2, which

Algorithm 2: Processor logic algorithm of the in-order processor

```
1 Function InO_Logic(P, Σ, t):
2     if (P[Σ.pc] = ⊥) then
3         return (Σ, true);
4     e <- matching_event(P[Σ.pc])
5     Σ' <- perform(Σ, e, t)
6     return (Σ', false);
```

includes all the combinational logic that the in-order processor has to evaluate at every stage for all instruction types. Specifically, the in-order processor performs exactly one event per cycle, which is the instruction with the current program counter. The processor logic updates the state as prescribed by the transition rules for the events, which corresponds to executing instructions on an unpipelined processor.

3 The Out-of-Order Processor with STT Protection

In this section, we present a model of an out-of-order single-core processor with STT protection. In the model, changes to the baseline out-of-order processor to support STT are shown in blue. We first introduce register renaming, typically performed by out-of-order processors upon fetching instructions. We then give an overview of the key idea of the STT protection, i.e., taint tracking. We then explain in detail the structure of the state of the out-of-order processor, a processor model as a state machine and the rules describing how the processor updates the state.

3.1 Register Renaming

The ISA specifies a finite set of registers available to programmers. Out-of-order machines typically treat those as only *logical* registers, and instead maintain a large number of *physical* registers to enable more reordering in the execution of instructions. To model this behavior, we assume that the STT processor manipulates physical registers PRegID and maintains a renaming table $\mathsf{rt} : \mathsf{RegID} \to \mathsf{PRegID}$ from logical to physical registers as a part of its state. To simplify technical development, we assume that the number of physical registers in PRegID is infinite. We discuss the impact of this assumption in Section 5.

Analogously to the instruction set architecture, we define the set of *dynamic* instructions $\mathsf{DynInstr}$, ranged over by Θ. A dynamic instruction is the unit of the processor's execution. The STT processor fetches instructions according to the program's control flow and may execute the same static instruction in a program multiple times (for instance, if it is in a loop). In order to distinguish different executions of each instruction, the STT processor creates a dynamic instruction upon fetching, and stores it in a special *reorder buffer*. A dynamic instruction has semantics identical to the non-dynamic one, and simply has physical registers as its parameters.

We define a function $\mathsf{rename} : \mathsf{State}_{\mathsf{STT}} \times \mathsf{Instr} \to (\mathsf{RegID} \to \mathsf{PRegID}) \times \mathsf{DynInstr}$ that captures the effect of register renaming. It takes two arguments, a state σ and a static instruction θ, and returns $\mathsf{rename}(\sigma, \theta) = (\mathsf{rt}, \Theta)$ such that Θ is a corresponding dynamic instruction and rt is updated register renaming table. The function rename is defined as follows:

$$\mathsf{rename}(\sigma,\theta) \triangleq \begin{cases} (\sigma.\mathsf{rt}[r_d \mapsto x_d], \langle \textbf{immed}\ x_d, k\rangle), & \text{if } \theta = \langle \textbf{immed}\ r_d, k\rangle \text{ and } x_d = \mathsf{fresh}(\sigma) \\ (\sigma.\mathsf{rt}[r_d \mapsto x_d], \langle \textbf{op}\ x_d, \sigma.\mathsf{rt}(r_a), \sigma.\mathsf{rt}(r_b)\rangle), & \text{if } \theta = \langle \textbf{op}\ r_d, r_a, r_b\rangle \text{ and } x_d = \mathsf{fresh}(\sigma) \\ (\sigma.\mathsf{rt}, \langle \textbf{branch}\ \sigma.\mathsf{rt}(r_c), \sigma.\mathsf{rt}(r_d)\rangle), & \text{if } \theta = \langle \textbf{branch}\ r_c, r_d\rangle \\ (\sigma.\mathsf{rt}[r_d \mapsto x_d], \langle \textbf{load}\ x_d, \sigma.\mathsf{rt}(r_a)\rangle), & \text{if } \theta = \langle \textbf{load}\ r_d, r_a\rangle \text{ and } x_d = \mathsf{fresh}(\sigma) \\ (\sigma.\mathsf{rt}, \langle \textbf{store}\ \sigma.\mathsf{rt}(r_a), \sigma.\mathsf{rt}(r_v)\rangle) & \text{if } \theta = \langle \textbf{store}\ r_a, r_v\rangle \end{cases}$$

Thus, rename simply renames logical input registers of instructions according to existing renaming in $\sigma.\mathsf{rt}$, and each logical output register is mapped to a new physical one. To that end, we use a function $\mathsf{fresh}(\sigma)$ that, for a given STT state σ, returns a physical register x that is *fresh* in σ, meaning that it has not been previously mapped to by rt (note that all such mappings are logged in dynamic instructions in the reorder buffer and thus can be recovered from σ).

3.2 STT Processor State

The STT processor state $\sigma \in \mathsf{State}_{\mathsf{STT}}$ consists of the components that we introduce below and further explain one by one in detail:

- A program counter $\mathsf{pc} \in \mathbb{N}$, which is an integer corresponding to the address of the next instruction to fetch.
- A memory $\mathsf{mem} : \mathbb{N} \to \mathbb{Z}$, which maps memory addresses to integer values. We treat all addresses as physical addresses, and discuss this in Section 5.
- A register file $\mathsf{reg} : \mathsf{PRegID} \to \mathbb{Z}$, which is a function mapping physical register identifiers to integer values denoting the content of a given register.
- A register status table $\mathsf{ready} : \mathsf{PRegID} \to \mathbb{B}$, which is a function mapping physical register identifiers to boolean flags denoting whether the instruction using a given register for its output has already written into it.
- A register renaming table $\mathsf{rt} : \mathsf{RegID} \to \mathsf{PRegID}$, which is a function that for a given logical register identifier r returns a corresponding physical register identifier x.
- A reorder buffer $\mathsf{rob} \in (\mathbb{N} \times \mathsf{DynInstr} \times (\mathbb{B} \cup \{\bot\}))^* \times \mathbb{N}$, which is a pair $(\mathsf{rob}_{\mathsf{seq}}, \mathsf{rob}_{\mathsf{head}})$ of an unbounded sequence $\mathsf{rob}_{\mathsf{seq}}$ of reorder buffer entries and an index $\mathsf{rob}_{\mathsf{head}}$ of the first uncommitted entry of the reorder buffer. The reorder buffer entries are triples of the form (pc, Θ, b) in which:
  - pc is a program counter,
  - Θ is a dynamic instruction,
  - b is a flag recording the branch prediction, provided that Θ is a branch instruction, or $\bot$ otherwise.
- A load queue $\mathsf{lq} \in (\mathbb{N} \times \mathbb{B} \times (\mathbb{Z} \cup \{\bot\}))^*$, which is a sequence of triples (i, f, t_{end}), where:
  - i is the index of a load instruction in the reorder buffer;
  - f is a boolean flag that is set to true only if the load $\mathsf{rob}[i]$ has completed the execution stage corresponding to loading a value from memory;
  - t_{end} is the cycle at which the load's execution ends.
- A store queue $\mathsf{sq} \in \mathbb{N}^*$, which is a sequence of integers denoting indexes of store instructions in the reorder buffer.
- A branch predictor $\mathsf{bp} \in \mathsf{BrPr}$, which is modeled as an abstract data type with update and $\mathsf{predict}$ functions defined in the following.
- A branch checkpoint list $\mathsf{ckpt} \in (\mathbb{N} \times \mathbb{N} \times \mathcal{P}(\mathsf{LRegID} \to \mathsf{PRegID}))^*$, which is a sequence of tuples $(i, \mathsf{pc}, \mathsf{rt})$ where:
  - i is the index of a branch instruction $\mathsf{rob}[i]$ triggering a checkpoint;
  - pc is the program counter at that checkpoint; and
  - rt is the register renaming table at the checkpoint.
- The cache state $C \in \mathbb{N}^*$, which we represent as the sequence of memory addresses accessed during the execution of a program.
- Taint metadata $\tau \in \mathsf{Taint}$, which is the extension to the out-of-order processor that enables determining which parts of the state should be considered secret.

Given state $\sigma = (\mathsf{pc}, \mathsf{mem}, \mathsf{reg}, \mathsf{ready}, \mathsf{rt}, \mathsf{rob}, \mathsf{lq}, \mathsf{sq}, \mathsf{bp}, \mathsf{ckpt}, C, \tau)$, we use a notation such as $\sigma.\mathsf{pc}$, $\sigma.\mathsf{mem}$ and $\sigma.\mathsf{reg}$ as shorthands for accessing pc, mem and reg respectively (and the other parts of the state, analogously). We also consider the initial state σ_{init}, where all status flags are set to false, and the reorder buffer rob, the load queue lq, the store queue sq, the branch-checkpoints list ckpt, as well as the cache C are all empty sequences.

We introduce the following shorthands for sequences. Firstly, we write $\_$ in formulas for variables whose names are irrelevant in that context. Secondly, given sequences xs, ys of tuples of n elements:

- We let $\mathsf{xs} +\!\!+ \mathsf{ys}$ denote the result of concatenating xs and ys. We overload the notation for appending to sequences: given x that is a tuple of n elements, we let $\mathsf{xs} +\!\!+ x$ denote the result of appending x to xs.
- We let $\mathsf{xs}[i]$ denote i-th entry in xs. We index sequences starting from 0, so $\mathsf{xs}[0]$ is the first element in a sequence.
- We let $\mathsf{xs}|_{<i}$ denote the subsequence of xs which omits all elements $(x, \_, ..., \_)$ such that $x \geq i$. We also let $\mathsf{xs}|_{\neq i}$ denote the subsequence of xs which omits all elements $(x, \_, ..., \_)$ such that $x = i$. Finally, we let $\mathsf{xs}|_i$ denote a prefix of xs that contains only its first i elements.
- We let $|\mathsf{xs}|$ denote the number of entries in the sequence xs.

Reorder buffer rob The reorder buffer $\mathsf{rob} = (\mathsf{rob}_{\mathsf{seq}}, \mathsf{rob}_{\mathsf{head}})$ is a pair of a sequence $\mathsf{rob}_{\mathsf{seq}}$ of reorder buffer entries, which are dynamic instructions of the program fetched w.r.t. its program order, and an index $\mathsf{rob}_{\mathsf{head}}$ of the first uncommitted entry. We assume that the reorder buffer is unbounded (the impact of this assumption is discussed in Section 5). Instructions are added into it upon fetching, and may be committed if the index $\mathsf{rob}_{\mathsf{head}}$ ever points to them. To simplify presentation, we never remove committed instructions from the reorder buffer, and simply increment $\mathsf{rob}_{\mathsf{head}}$ upon committing one. Instructions may also be squashed, at which point a corresponding suffix of the reorder buffer is discarded.

Given a reorder buffer $\mathsf{rob} = (\mathsf{rob}_{\mathsf{seq}}, \mathsf{rob}_{\mathsf{head}})$, we refer to i-th entry in $\mathsf{rob}_{\mathsf{seq}}$ by $\mathsf{rob}[i]$. We also refer to $|\mathsf{rob}_{\mathsf{seq}}|$ as $\mathsf{rob}_{\mathsf{tail}}$, when context is unambiguous. Thus, $\mathsf{rob}[0]$ is the first ROB entry, $\mathsf{rob}[\mathsf{rob}_{\mathsf{head}}]$ is the first uncommitted entry, and $\mathsf{rob}[\mathsf{rob}_{\mathsf{tail}}]$ is the one after the last. We use $+\!\!+$ to append elements to sequences as well as concatenate to sequences. We also let $\mathsf{rob} +\!\!+ (\mathsf{pc}, \Theta, b)$ be a shorthand for $(\mathsf{rob}_{\mathsf{seq}} +\!\!+ (\mathsf{pc}, \Theta, b), \mathsf{rob}_{\mathsf{head}})$.

Load queue lq **and store queue** sq The STT processor state includes a load and a store queue in order to support load-store forwarding. Each time a load instruction $\mathsf{rob}[i]$ is fetched, a new entry (i, f, t_{end}) is appended to lq with $f = \mathsf{false}$ and $t_{\mathsf{end}} = \bot$, which get updated during the load's execution (see Section 3.4). Each time a store instruction $\mathsf{rob}[i]$ is fetched, a new entry i is appended to sq.When either kind of the instruction commits, the corresponding entry is removed from lq or sq, respectively.

Branch predictor Modern branch predictors usually include multiple subcomponents, which are updated at different points within an branch instruction's lifetime. The most common branch predictors (gshare, TAGE, etc.) all have branch history registers, which are updated right after a branch is predicted. Later when the branch is resolved, correct branch outcome is used to train the branch predictor for better prediction in the future. Our branch predictor model bp follows this design decision.

We model the branch predictor abstractly to avoid specifying implementation details of it and stay compatible with common branch predictor designs. In our abstract treatment of it, we consider branch predictor to be a state transition system, the state and transition rules of which are not specified. To this end, we assume a set BrPr, ranged over by bp, of all possible configurations in which the branch predictor can be, and we refer to bp as a branch predictor in the following.

Our branch predictor models both direct and indirect branch prediction. The function predict takes a program counter pc_{br} identifying the current branch instruction and returns predicted direction and target address. Thus, when $\mathsf{bp}.\mathrm{predict}(\mathsf{pc}_{br}) = (\mathsf{spc}, b)$, for a given program counter pc_{br}, the function returns spc, which is the predicted target address, and b, which is a boolean value indicating whether the branch is predicted taken or not: the branch is taken when $b = \mathsf{true}$, in which case the processor changes the program counter to spc, or otherwise simply increments it, if $b = \mathsf{false}$.

Our model of a branch predictor has two update functions returning the updated branch predictor. Following common implementations, the two functions correspond to the two phases in which the branch predictor structures can be updated:

1. The first update $\mathsf{bp}.\mathrm{update}(b)$ happens right after the prediction is made. The history register (global or local) is updated with the predicted direction b.
2. The second update $\mathsf{bp}.\mathrm{update}(\mathsf{pc}, \mathsf{cond}, \mathsf{target})$ happens after the branch instruction is executed by the branch execution unit. The prediction is verified to be correct or incorrect, and the target address is also determined. Branch predictor is updated with both correct direction cond and correct target address target.

Thus, although our branch predictor model is inspired by common branch predictor designs, it does not specify any concrete structure. Therefore, any branch predictor design with the above interface can be modeled by bp.

Branch checkpoints ckpt Branch mispredictions necessitate squashing effects of uncommitted instructions. To model this behavior, we record snapshots of relevant parts of the state in the list of branch checkpoints, ckpt. Upon fetching a branch instruction and making a branch prediction, we extend ckpt with a triple $(i, \mathsf{pc}', \mathsf{rt}')$, in which i is the instruction's index in the reorder buffer, pc' is the current program counter value, and rt' is the current register naming mapping.

If upon executing an instruction $\mathsf{rob}[i] = (\mathsf{pc}', \langle \textbf{branch}\ x_c, x_d \rangle, b)$, the branch prediction b is revealed to be correct, then this entry is removed from ckpt, meaning that we update ckpt to $\mathsf{ckpt}|_{\neq i}$. If the prediction is incorrect, the STT processor (a) uses the checkpoint $(i, \mathsf{pc}', \mathsf{rt}')$ to roll back the program counter and the renaming mapping to pc' and rt', respectively; (b) discards all subsequent checkpoints by trimming the list ckpt of branch predictions to $\mathsf{ckpt}|_{<i}$; (c) discards all entries in lq and sq added after $\mathsf{rob}[i]$ by trimming them to $\mathsf{lq}|_{<i}$ and $\mathsf{sq}|_{<i}$, respectively; and, (d) discards all entries in the reorder buffer after $\mathsf{rob}[i]$. Note that we do not explicitly roll-back the values in register structures reg and ready: a combination of unrolling rt to the state when squashed registers are not accessible and initializing registers upon renaming ensures that the squashed values are never re-used.

Using the branch checkpoints list, we define the following predicate $\mathsf{underSpec}(\mathsf{ckpt}, i)$ that holds whenever the index i in the reorder buffer corresponds to an instruction performed under speculation[2]:

$$\mathsf{underSpec}(\mathsf{ckpt}, i) \triangleq (\exists j.\ (j, \_, \_) \in \mathsf{ckpt} \wedge j < i)$$

We use $\mathsf{underSpec}(\sigma)$ as a shorthand for $\mathsf{underSpec}(\sigma.\mathsf{ckpt}, \sigma.\mathsf{rob}_{\mathsf{tail}})$.

Cache C Instead of modeling a specific cache configuration, we model the cache as the sequence C of addresses that loads and stores access during the execution of the program, which is what determines the cache state in practice. This model further allows us to consider a strong adversary that can observe *every* cache/memory access. Additionally, we allow loads to span across multiple cycle, which we refer to as load latency. To model load latency, we assume a function $\mathsf{LoadLat} : \mathbb{N}^* \times \mathbb{N} \to \mathbb{N}$ that for a given sequence of memory addresses already accessed and the next address to access returns the number of cycles it takes to load the value by that address.

Taint metadata τ Taint metadata $\tau \in \mathsf{Taint}$ is the extension to the state of the out-of-order processor that enables determining which parts of the state are secret. We assume three auxiliary functions, taint, $\mathsf{noTaintedInputs}$ and $\mathsf{untaint}$, which we treat as parameters in our presentation. In this document, we describe two instantiations of taint metadata, corresponding to two taint tracking mechanisms possible with the STT processor. On the abstract level, the taint-metadata functions fulfill the following purposes:

- The function $\mathsf{taint} : \mathsf{State}_{\mathsf{STT}} \times \mathsf{Instr} \to \mathsf{Taint}$ prescribes how to taint registers. Intuitively, when $\mathsf{taint}(\sigma, \theta) = \tau'$, τ' is updated taint metadata in which the destination physical register of θ is tainted.
- The function $\mathsf{noTaintedInputs} : \mathsf{State}_{\mathsf{STT}} \times \mathbb{N} \to \mathbb{B}$ prescribes how to check taint of input registers of in-flight instructions. Intuitively, when $\mathsf{noTaintedInputs}(\sigma, i)$ holds, dependencies of $\sigma.\mathsf{rob}[i]$ are not tainted.
- The function $\mathsf{untaint} : \mathsf{State}_{\mathsf{STT}} \to \mathsf{State}_{\mathsf{STT}}$ prescribes how to untaint registers and propagate the changes through the instruction dependencies.

Definition 2. *A physical register $x \in \mathsf{PRegID}$ is* tainted *in a state σ, written $\mathsf{tainted}(\sigma, x)$, if there is an index $i \geq \sigma.\mathsf{rob}_{\mathsf{head}}$ and an instruction $\sigma.\mathsf{rob}[i]$ in the reorder buffer such that one of the following is true:*

- $\sigma.\mathsf{rob}[i] = (\_, \langle \textbf{load}\ x, \_ \rangle, \_)$ *and* $\mathsf{underSpec}(\sigma.\mathsf{ckpt}, i)$ *holds; or*
- *x is an output register for $\sigma.\mathsf{rob}[i]$ and $\mathsf{noTaintedInputs}(\sigma, i)$ does not hold.*

Intuitively, the two cases correspond to the two possible reasons for a register x to be tainted: the former is due to a load instruction using x as an output for data accessed under speculation, and the latter is due to propagation of this taint via instruction dependencies.

We require the following properties from $\mathsf{noTaintedInputs}$.

[2]In practice, to determine which instructions stop being speculative, the STT processor implementation re-uses the logic provided by prior work on InvisiSpec [2]

Algorithm 3: untaint in the basic taint tracking scheme

```
1 Function untaint(σ):
2 |   σ' <- σ;
3 |   for i from σ.rob_head to σ.rob_tail − 1 do
4 |   |   σ'.τ <- untaintInstr(σ', i)
5 |   end
6 |   return σ';
```

Property 3. *If σ is a state, i is an index such that $\sigma.\mathsf{rob}_{\mathsf{head}} \leq i < \sigma.\mathsf{rob}_{\mathsf{tail}}$ holds, and* $\mathsf{noTaintedInputs}(\sigma, i)$ *holds too, then for every input register x of the instruction $\sigma.\mathsf{rob}[i]$,* $\mathsf{tainted}(\sigma, x)$ *does not hold.*

Property 4. *If σ_1 and σ_2 are two states different only by register values, i is an index such that $\sigma_1.\mathsf{rob}_{\mathsf{head}} \leq i < \sigma_1.\mathsf{rob}_{\mathsf{tail}}$ holds, then* $\mathsf{noTaintedInputs}(\sigma_1, i)$ *holds if and only if* $\mathsf{noTaintedInputs}(\sigma_2, i)$ *does.*

3.3 A Basic Taint Tracking Scheme

STT tracks information flow from load instructions to younger in-flight instructions. STT *taints* the output register of a speculative access instruction and propagates taint using standard taint tracking rules, namely that an instruction's output register is tainted if any of its input registers are tainted. Unlike conventional dynamic information flow tracking, STT *automatically* untaints data. When a load instruction stops being speculative, its output register is *untainted*. Untaint information is also propagated, so that when all the data dependencies of an instruction become untainted, the instruction's output is untainted. We explain how taint tracking schemes are integrated into the STT processor model in Section 3.6.

At a high level, taint propagation is piggybacked on the existing register renaming logic in a modern out-of-order core. As an instruction enters the front-end and its registers are renamed, the instruction's output register is tainted if (1) it is an access instruction or (2) any of its input (physical) registers are tainted. Propagating untaint is non-trivial, because dependency chains can be long and each instruction can have many data dependencies whose taint status needs to be tracked. STT addresses these challenges with a novel fast untaint algorithm (see Section 7 in [4]). We defer the detailed scheme of STT's taint/untaint tracking implementation to Section 3.6. In the following, we outline a simplistic scheme providing taint/untaint information.

In order to support the basic taint tracking scheme, we instantiate taint metadata as a register taint table $\tau \in \mathsf{Taint} = (\mathsf{PRegID} \to \mathbb{B})$, which stores a taint bit for every physical register. In the initial state σ_{init}, the taint table stores false for all of the registers.

At each cycle, fetching an instruction θ triggers tainting registers as specified by the following function $\mathsf{taint}(\sigma, \theta)$:

$$\mathsf{taint}(\sigma, \theta) \triangleq \begin{cases} \sigma.\tau[x_d \mapsto (\sigma.\tau(x_a) \vee \sigma.\tau(x_b))], & \text{if } \mathsf{rename}(\sigma, \theta) = (\langle \mathbf{op}\ x_d, x_a, x_b \rangle, \_) \\ \sigma.\tau[x_d \mapsto \mathsf{underSpec}(\sigma)], & \text{if } \mathsf{rename}(\sigma, \theta) = (\langle \mathbf{load}\ x_d, x_a \rangle, \_) \\ \sigma.\tau, & \text{otherwise} \end{cases}$$

The predicate $\mathsf{noTaintedInputs}(\sigma, i)$ holds whenever dependencies of the dynamic instruction stored in $\sigma.\mathsf{rob}[i]$ with the given index i are not tainted, i.e.:

$$\mathsf{noTaintedInputs}(\sigma, i) \triangleq \begin{cases} \mathsf{true}, & \text{if } \sigma.\mathsf{rob}[i] = \langle \mathbf{immed}\ x_d, k \rangle \\ \neg\tau(x_a) \wedge \neg\tau(x_b), & \text{if } \sigma.\mathsf{rob}[i] = \langle \mathbf{op}\ x_d, x_a, x_b \rangle \\ \neg\tau(x_a), & \text{if } \sigma.\mathsf{rob}[i] = \langle \mathbf{load}\ x_d, x_a \rangle \\ \neg\tau(x_c) \wedge \neg\tau(x_d), & \text{if } \sigma.\mathsf{rob}[i] = \langle \mathbf{branch}\ x_c, x_d \rangle \\ \neg\tau(x_a) \wedge \neg\tau(x_v), & \text{if } \sigma.\mathsf{rob}[i] = \langle \mathbf{store}\ x_a, x_v \rangle \end{cases}$$

The function $\mathsf{untaint}(\sigma)$ (Algorithm 3) untaints load instructions that are no longer speculative and propagates untaint information to their dependent instructions. It uses the following $\mathsf{untaintInstr}(\sigma, i)$ function to untaint the output of a dynamic instruction stored in ROB entry i:

$$\mathsf{untaintInstr}(\sigma, i) \triangleq \begin{cases} \sigma.\tau[x_d \mapsto \mathsf{false}], & \text{if } \sigma.\mathsf{rob}[i] = \langle \mathbf{op}\ x_d, x_a, x_b \rangle \text{ and } \mathsf{noTaintedInputs}(\sigma, i) \\ \sigma.\tau[x_d \mapsto \mathsf{false}], & \text{if } \sigma.\mathsf{rob}[i] = \langle \mathbf{load}\ x_d, x_a \rangle \wedge \neg\mathsf{underSpec}(\sigma.\mathsf{ckpt}, i) \\ \sigma.\tau, & \text{otherwise} \end{cases}$$

Events class	**Events** (for all $i \in \mathbb{N}$)
FetchEvents	(Fetch-Immediate, $\bot$) (Fetch-Arithmetic, $\bot$) (Fetch-Branch, $\bot$) (Fetch-Load, $\bot$) (Fetch-Store, $\bot$)
ExecuteEvents	(Execute-Immediate, i) (Execute-Arithmetic, i) (Execute-Branch-Success, i) (Execute-Branch-Fail, i) (Execute-Load-BeginGetS, i) (Execute-Load-EndGetS, i) (Execute-Load-Complete, i)
CommitEvents	(Commit-Immediate, $\bot$) (Commit-Arithmetic, $\bot$) (Commit-Branch, $\bot$) (Commit-Load, $\bot$) (Commit-Store, $\bot$)

Table 1: Classification of μ-events in the set $\mathsf{STT\_Events} = \mathsf{FetchEvents} \cup \mathsf{ExecuteEvents} \cup \mathsf{CommitEvents}$. Events are pairs of a μ-event name and either $\bot$ or $i \in \mathbb{N}$, where i is an index in the reorder buffer of an instruction corresponding to the event (we use $\bot$ in the μ-events which identify the instruction by the program counter).

3.4 Semantics of Instructions Execution

Microarchitectural Events At each cycle, the STT processor logic performs *microarchitectural events* (μ-events) that correspond to one of four logical steps for instructions: getting fetched, executed, untainted or committed. We formalize the semantics of each μ-event with the help of a transition relation $\longrightarrow \subseteq (\mathsf{State_{STT}} \times \mathsf{Events_{STT}} \times \mathsf{State_{STT}})$. We say that $\{\sigma\} \xrightarrow{e,t} \{\sigma'\}$ holds, if at the cycle t, a transition from a state σ to σ' is possible and it corresponds to the μ-event e. Thus, the transition relation describes the changes to the state due to performing μ-events at a particular cycle (the latter is used as a parameter in modelling instruction latency).

We list all μ-events of the STT processor in Table 1. Every event $e \in \mathsf{Events_{STT}}$ is a pair of a μ-event name and either $\bot$ or an index $i \in \mathbb{N}$ of a corresponding instruction in the reorder buffer. Table 1 defines three classes of μ-events in accordance with the stages instructions go through. We let *μ-event trace* be a sequence of μ-events.

We now present transition rules defining semantics of fetching, executing and committing instructions. Although each transition $\langle\sigma\rangle \xrightarrow{e,t} \langle\sigma'\rangle$ between two states σ and σ' is parametrized by a μ-event e and a cycle t, for simplicity of presentation, we omit the μ-events from the notation when we present the transition rules, but put them before each transition. Like in Section 2.2, we call the list of contraints over the bar in each rule *premises* of the μ-event, and the description of a state update under the bar a *transition* of the μ-event.

Fetching instructions Upon fetching, the STT processor renames logical registers into physical registers with the help of the function rename, and appends the resulting dynamic instruction to the reorder buffer.

(Fetch-Immediate, $\bot$):

$$\frac{P[\mathsf{pc}] = \langle \mathbf{immed}\ r_d, k\rangle \qquad \mathsf{rename}(\sigma, \theta) = (\mathsf{rt}', \Theta) \qquad x_d = \mathsf{rt}'(r_d)}{\sigma = \left\{ \begin{array}{l} \mathsf{pc}, \mathsf{mem}, \mathsf{reg}, \mathsf{ready}, \mathsf{rt}, \\ \mathsf{rob}, \mathsf{lq}, \mathsf{sq}, \mathsf{bp}, \mathsf{ckpt}, C, \tau \end{array} \right\} \xrightarrow{t} \left\{ \begin{array}{l} \mathsf{pc}+1, \mathsf{mem}, \mathsf{reg}, \mathsf{ready}[x_d \mapsto \mathsf{false}], \mathsf{rt}', \\ \mathsf{rob} +\!\!+ [(\mathsf{pc}, \Theta, \bot)], \mathsf{lq}, \mathsf{sq}, \mathsf{bp}, \mathsf{ckpt}, C, \tau \end{array} \right\}}$$

The Fetch-Immediate event changes the state by incrementing the program counter, updating the register status table from ready to $\mathsf{ready}[x_d \mapsto \mathsf{false}]$, the register table rt to rt′ to account for renaming r_d into x_d, and also by appending a new entry $(\mathsf{pc}, \Theta, \bot)$ to the reorder buffer rob.

(Fetch-Arithmetic, $\bot$):

$$\frac{P[\mathsf{pc}] = \langle \mathbf{op}\ r_d, r_a, r_b\rangle \qquad \mathsf{rename}(\sigma, \theta) = (\mathsf{rt}', \Theta) \qquad x_d = \mathsf{rt}'(r_d)}{\sigma = \left\{ \begin{array}{l} \mathsf{pc}, \mathsf{mem}, \mathsf{reg}, \mathsf{ready}, \mathsf{rt}, \\ \mathsf{rob}, \mathsf{lq}, \mathsf{sq}, \mathsf{bp}, \mathsf{ckpt}, C, \tau \end{array} \right\} \xrightarrow{t} \left\{ \begin{array}{l} \mathsf{pc}+1, \mathsf{mem}, \mathsf{reg}, \mathsf{ready}[x_d \mapsto \mathsf{false}], \mathsf{rt}', \\ \mathsf{rob} +\!\!+ [(\mathsf{pc}, \Theta, \bot)], \mathsf{lq}, \mathsf{sq}, \mathsf{bp}, \mathsf{ckpt}, C, \tau \end{array} \right\}}$$

The Fetch-Arithmetic event changes the state by incrementing the program counter, updating the register status table from ready to $\mathsf{ready}[x_d \mapsto \mathsf{false}]$, the register table rt to rt′ to account for renaming r_d into x_d, and also by appending a new entry $(\mathsf{pc}, \Theta, \bot)$ to the reorder buffer rob.

(Fetch-Branch, $\bot$):

$$\frac{\mathsf{bp.predict}(\mathsf{pc}) = (\mathsf{spc}, b) \qquad P[\mathsf{pc}] = \langle \textbf{branch } r_c, r_d \rangle \qquad \mathsf{rename}(\sigma, \theta) = (\mathsf{rt}', \Theta) \qquad \mathsf{bp}' = \mathsf{bp.update}(b) \qquad \mathsf{pc}' = (\textbf{if } b \textbf{ then } \mathsf{spc} \textbf{ else } \mathsf{pc} + 1) \qquad \mathsf{newCkpt} = (\mathsf{rob}_{\mathsf{tail}}, \mathsf{pc}, \mathsf{rt})}{\sigma = \left\{ \begin{array}{l} \mathsf{pc}, \mathsf{mem}, \mathsf{reg}, \mathsf{ready}, \mathsf{rt}, \\ \mathsf{rob}, \mathsf{lq}, \mathsf{sq}, \mathsf{bp}, \mathsf{ckpt}, C, \tau \end{array} \right\} \xrightarrow{t} \left\{ \begin{array}{l} \mathsf{pc}', \mathsf{mem}, \mathsf{reg}, \mathsf{ready}, \mathsf{rt}', \\ \mathsf{rob} +\!\!+ [(\mathsf{pc}, \Theta, (\mathsf{spc}, b))], \mathsf{lq}, \mathsf{sq}, \mathsf{bp}', \mathsf{ckpt} +\!\!+ [\mathsf{newCkpt}], C, \tau \end{array} \right\}$$

The Fetch-Branch event changes the state by updating the program counter, appending a new entry $(\mathsf{pc}, \Theta, (\mathsf{spc}, b))$ to the reorder buffer rob, updating the state of the branch predictor to a new value bp' and also by appending a new checkpoint $\mathsf{newCkpt} = (\mathsf{rob}_{\mathsf{tail}}, \mathsf{pc}, \mathsf{rt})$ to the list of branch checkpoints ckpt.

(Fetch-Load, $\bot$):

$$\frac{P[\mathsf{pc}] = \langle \textbf{load } r_d, r_a \rangle \qquad \mathsf{rename}(\sigma, \theta) = (\mathsf{rt}', \Theta) \qquad x_d = \mathsf{rt}'(r_d)}{\sigma = \left\{ \begin{array}{l} \mathsf{pc}, \mathsf{mem}, \mathsf{reg}, \mathsf{ready}, \mathsf{rt}, \\ \mathsf{rob}, \mathsf{lq}, \mathsf{sq}, \mathsf{bp}, \mathsf{ckpt}, C, \tau \end{array} \right\} \xrightarrow{t} \left\{ \begin{array}{l} \mathsf{pc} + 1, \mathsf{mem}, \mathsf{reg}, \mathsf{ready}[x_d \mapsto \mathsf{false}], \mathsf{rt}', \\ \mathsf{rob} +\!\!+ [(\mathsf{pc}, \Theta, \bot)], \mathsf{lq}, \mathsf{sq}, \mathsf{bp}, \mathsf{ckpt}, C, \tau \end{array} \right\}$$

The Fetch-Load event changes the state by incrementing the program counter, updating the register status table from ready to $\mathsf{ready}[x_d \mapsto \mathsf{false}]$, the register table rt to rt' to account for renaming r_d into x_d, and also by appending a new entry $(\mathsf{pc}, \Theta, \bot)$ to the reorder buffer rob.

(Fetch-Store, $\bot$):

$$\frac{P[\mathsf{pc}] = \langle \textbf{store } r_a, r_v \rangle \qquad \mathsf{rename}(\sigma, \theta) = (\mathsf{rt}', \Theta)}{\sigma = \left\{ \begin{array}{l} \mathsf{pc}, \mathsf{mem}, \mathsf{reg}, \mathsf{ready}, \mathsf{rt}, \\ \mathsf{rob}, \mathsf{lq}, \mathsf{sq}, \mathsf{bp}, \mathsf{ckpt}, C, \tau \end{array} \right\} \xrightarrow{t} \left\{ \begin{array}{l} \mathsf{pc} + 1, \mathsf{mem}, \mathsf{reg}, \mathsf{ready}, \mathsf{rt}', \\ \mathsf{rob} +\!\!+ (\mathsf{pc}, \Theta, \bot), \mathsf{lq}, \mathsf{sq} +\!\!+ [\mathsf{rob}_{\mathsf{tail}}], \mathsf{bp}, \mathsf{ckpt}, C, \tau \end{array} \right\}$$

The Fetch-Store event changes the state by incrementing the program appending a new entry $(\mathsf{pc}, \Theta, \bot)$ to the reorder buffer rob and also the index of the new rob entry to the store queue.

Executing instructions Upon performing the Execute-Immediate event, we simply put a value into the destination register:

(Execute-Immediate, i):

$$\frac{\mathsf{rob}[i] = (\_; \langle \textbf{immed } x_d, k \rangle; \bot) \qquad \neg \mathsf{ready}(x_d)}{\left\{ \begin{array}{l} \mathsf{pc}, \mathsf{mem}, \mathsf{reg}, \mathsf{ready}, \mathsf{rt}, \\ \mathsf{rob}, \mathsf{lq}, \mathsf{sq}, \mathsf{bp}, \mathsf{ckpt}, C, \tau \end{array} \right\} \xrightarrow{t} \left\{ \begin{array}{l} \mathsf{pc}, \mathsf{mem}, \mathsf{reg}[x_d \mapsto k], \mathsf{ready}[x_d \mapsto \mathsf{true}], \mathsf{rt}, \\ \mathsf{rob}, \mathsf{lq}, \mathsf{sq}, \mathsf{bp}, \mathsf{ckpt}, C, \tau \end{array} \right\}$$

Upon performing the Execute-Arithmetic event for an arithmetic instruction in $\mathsf{rob}[i]$, we ensure that the inputs of the instruction have been computed, and then store the result of the instruction into the destination register:

(Execute-Arithmetic, i):

$$\frac{\mathsf{rob}[i] = (\_; \langle \textbf{op } x_d, x_a, x_b \rangle; \bot) \qquad \mathsf{ready}(x_a) \qquad \mathsf{ready}(x_b) \qquad \neg \mathsf{ready}(x_d)}{\left\{ \begin{array}{l} \mathsf{pc}, \mathsf{mem}, \mathsf{reg}, \mathsf{ready}, \mathsf{rt}, \\ \mathsf{rob}, \mathsf{lq}, \mathsf{sq}, \mathsf{bp}, \mathsf{ckpt}, C, \tau \end{array} \right\} \xrightarrow{t} \left\{ \begin{array}{l} \mathsf{pc}, \mathsf{mem}, \mathsf{reg}[x_d \mapsto \textbf{op}(\mathsf{reg}(x_a), \mathsf{reg}(x_b))], \mathsf{ready}[x_d \mapsto \mathsf{true}], \mathsf{rt}, \\ \mathsf{rob}, \mathsf{lq}, \mathsf{sq}, \mathsf{bp}, \mathsf{ckpt}, C, \tau \end{array} \right\}$$

Upon executing a branch instruction in $\mathsf{rob}[i]$, we perform one of the following two μ-events, Execute-Branch-Success or Execute-Branch-Fail, depending on whether the branch prediction was correct:

(Execute-Branch-Success, i):

$$\frac{\mathsf{ready}(x_c) \qquad \mathsf{ready}(x_d) \qquad \mathsf{rob}[i] = (\mathsf{pc}'; \langle \textbf{branch } x_c, x_d \rangle; (\mathsf{spc}, b)) \qquad b = \mathsf{reg}(x_c) \qquad \mathsf{spc} = \mathsf{reg}(x_d) \qquad \mathsf{bp}' = \mathsf{bp.update}(\mathsf{pc}', \mathsf{reg}(x_c), \mathsf{reg}(x_d)),}{\left\{ \begin{array}{l} \mathsf{pc}, \mathsf{mem}, \mathsf{reg}, \mathsf{ready}, \mathsf{rt}, \\ \mathsf{rob}, \mathsf{lq}, \mathsf{sq}, \mathsf{bp}, \mathsf{ckpt}, C, \tau \end{array} \right\} \xrightarrow{t} \left\{ \begin{array}{l} \mathsf{pc}, \mathsf{mem}, \mathsf{reg}, \mathsf{ready}, \mathsf{rt}, \\ \mathsf{rob}, \mathsf{lq}, \mathsf{sq}, \mathsf{bp}', \mathsf{ckpt}|_{\neq i}, C, \tau \end{array} \right\}$$

(Execute-Branch-Fail, i):

$$\frac{\begin{array}{l} \mathsf{rob}[i] = (\mathsf{pc}'; \langle \textbf{branch } x_c, x_d \rangle; (\mathsf{spc}, b)) \qquad \mathsf{ready}(x_c) \qquad \mathsf{ready}(x_d) \qquad (b \neq \mathsf{reg}(x_c)) \vee (\mathsf{spc} \neq \mathsf{reg}(x_d)) \\ (i, \mathsf{pc}', \mathsf{rt}') \in \mathsf{ckpt} \qquad \mathsf{pc}'' = (\textbf{if } \mathsf{reg}(x_c) \textbf{ then } \mathsf{reg}(x_d) \textbf{ else } \mathsf{pc}' + 1) \qquad \mathsf{bp}' = \mathsf{bp.update}(\mathsf{pc}', \mathsf{reg}(x_c), \mathsf{reg}(x_d)) \end{array}}{\left\{ \begin{array}{l} \mathsf{pc}, \mathsf{mem}, \mathsf{reg}, \mathsf{ready}, \mathsf{rt}, \\ \mathsf{rob}, \mathsf{lq}, \mathsf{sq}, \mathsf{bp}, \mathsf{ckpt}, C, \tau \end{array} \right\} \xrightarrow{t} \left\{ \begin{array}{l} \mathsf{pc}'', \mathsf{mem}, \mathsf{reg}, \mathsf{ready}, \mathsf{rt}', \\ \mathsf{rob}|_i, \mathsf{lq}|_{<i}, \mathsf{sq}|_{<i}, \mathsf{bp}', \mathsf{ckpt}|_{<i}, C, \tau \end{array} \right\}$$

If the prediction made upon fetching is revealed to be correct, Execute-Branch-Success occurs and removes a corresponding checkpoint from the list. Otherwise, Execute-Branch-Fail unrolls parts of the state to a

previously made checkpoint. It is important to note that Algorithm 5 ensures that transitions for both μ-events occur only if the instruction's operands are not tainted.

Execution of a load instruction in $\mathsf{rob}[i]$ goes through three steps corresponding to the following three μ-events:

$$\frac{\mathsf{rob}[i] = (\mathsf{pc}_i; \langle \textbf{load}\ x_d, x_a \rangle; \bot) \qquad \mathsf{ready}(x_a) \qquad \neg\mathsf{ready}(x_d) \qquad \mathsf{storesAreReady}(i) \qquad t_{\mathsf{end}} = t + \mathsf{LoadLat}(C, \mathsf{reg}(x_a))}{\left\{\begin{array}{l}\mathsf{pc}, \mathsf{mem}, \mathsf{reg}, \mathsf{ready}, \mathsf{rt},\\ \mathsf{rob}, \mathsf{lq} +\!\!+ (i, \mathsf{false}, \bot) +\!\!+ \mathsf{lq}', \mathsf{sq}, \mathsf{bp}, \mathsf{ckpt}, C, \tau\end{array}\right\} \xrightarrow{t} \left\{\begin{array}{l}\mathsf{pc}, \mathsf{mem}, \mathsf{reg}, \mathsf{ready}, \mathsf{rt},\\ \mathsf{rob}, \mathsf{lq} +\!\!+ (i, \mathsf{false}, t_{\mathsf{end}}) +\!\!+ \mathsf{lq}', \mathsf{sq}, \mathsf{bp}, \mathsf{ckpt}, C +\!\!+ \mathsf{reg}(x_a), \tau\end{array}\right\}}$$

(Execute-Load-EndGetS, i):

$$\frac{\mathsf{rob}[i] = (\mathsf{pc}_i; \langle \textbf{load}\ x_d, x_a \rangle; \bot) \qquad t \geq t_{\mathsf{end}} \qquad t_{\mathsf{end}} \neq \bot}{\left\{\begin{array}{l}\mathsf{pc}, \mathsf{mem}, \mathsf{reg}, \mathsf{ready}, \mathsf{rt},\\ \mathsf{rob}, \mathsf{lq} +\!\!+ (i, \mathsf{false}, t_{\mathsf{end}}) +\!\!+ \mathsf{lq}', \mathsf{sq}, \mathsf{mem}, \mathsf{bp}, \mathsf{ckpt}, C, \tau\end{array}\right\} \xrightarrow{t} \left\{\begin{array}{l}\mathsf{pc}, \mathsf{mem}, \mathsf{reg}[x_d \mapsto \mathsf{mem}(\mathsf{reg}(x_a))], \mathsf{ready}, \mathsf{rt},\\ \mathsf{rob}, \mathsf{lq} +\!\!+ (i, \mathsf{true}, t_{\mathsf{end}}) +\!\!+ \mathsf{lq}', \mathsf{sq}, \mathsf{bp}, \mathsf{ckpt}, C, \tau\end{array}\right\}}$$

(Execute-Load-Complete, i):

$$\frac{\mathsf{rob}[i] = (\mathsf{pc}_i; \langle \textbf{load}\ x_d, x_a \rangle; \bot) \qquad (i, \mathsf{true}, \_) \in \mathsf{lq} \qquad \mathsf{res} = \mathsf{loadResult}(i, x_a, \mathsf{reg}(x_d))}{\left\{\begin{array}{l}\mathsf{pc}, \mathsf{mem}, \mathsf{reg}, \mathsf{ready}, \mathsf{rt},\\ \mathsf{rob}, \mathsf{lq}, \mathsf{sq}, \mathsf{bp}, \mathsf{ckpt}, C, \tau\end{array}\right\} \xrightarrow{t} \left\{\begin{array}{l}\mathsf{pc}, \mathsf{mem}, \mathsf{reg}[x_d \mapsto \mathsf{res}], \mathsf{ready}[x_d \mapsto \mathsf{true}], \mathsf{rt},\\ \mathsf{rob}, \mathsf{lq}, \mathsf{sq}, \mathsf{bp}, \mathsf{ckpt}, C, \tau\end{array}\right\}}$$

Execution of the load instruction is predicated on several important conditions. Firstly, Algorithm 5 ensures that transitions for both μ-events are only possible if the load address is not tainted. Secondly, the μ-event only occurs if all store instructions in the store queue are ready for execution, i.e., if their input registers have received their data. We formalize the latter as the following predicate:

$$\mathsf{StoresAreReady}(i) \triangleq \forall j', \mathsf{pc}', x'_b, x'_v.\ j' \in \mathsf{sq} \wedge j' < i \wedge \mathsf{rob}[j'] = (\mathsf{pc}'; \langle \textbf{store}\ x'_b, x'_v \rangle; \bot; \_) \implies \mathsf{ready}(x'_b)$$

The loads span across multiple cycles, which we model in transition rules by having separate μ-events for starting and finishing the memory access performed as a part of executing a load. Initially, the Execute-Load-BeginGetS event is performed, and after the cycle counter increases past the load latency, Execute-Load-EndGetS occurs.

Finally, the Execute-Load-Complete event checks whether the previously loaded value can be returned by the load instruction. To this end, the μ-event checks whether there is a possibility of store-to-load forwarding, in which case that is how the result of the load is decided, or otherwise the result is the previously loaded value. We encode this behavior with a predicate $\mathsf{CanForward}$ and a function $\mathsf{loadResult}$ defined as follows:

$$\begin{aligned}
&\mathsf{CanForward}(i, x_a, j, x_v) \triangleq j \in \mathsf{sq} \wedge j < i \wedge \\
&\exists x_b.\ \mathsf{rob}[j] = (_; \langle \textbf{store}\ x_b, x_v \rangle; \bot; _) \wedge \mathsf{ready}(x_b) \wedge \mathsf{ready}(x_v) \wedge \mathsf{reg}(x_a) = \mathsf{reg}(x_b) \\
&\quad \wedge (\forall j', x'_b.\ j < j' < i \wedge \mathsf{rob}[j'] = (_; \langle \textbf{store}\ x'_b, _ \rangle; \bot; _) \implies \mathsf{reg}(x_a) \neq \mathsf{reg}(x'_b)) \\
&\mathsf{loadResult}(i, x_a, \mathsf{oldRes}) \triangleq \begin{cases} \mathsf{reg}(x_v) & \text{if there exists } j \text{ such that } \mathsf{CanForward}(i, x_a, j, x_v) \\ \mathsf{oldRes} & \text{otherwise} \end{cases}
\end{aligned}$$

Commiting instructions The transition rules for commiting instructions are standard.

(Commit-Immediate, $\bot$):

$$\frac{\mathsf{rob}_{\mathsf{seq}}[\mathsf{rob}_{\mathsf{head}}] = (\_; \langle \textbf{immed}\ x_d, \_ \rangle; \bot) \qquad \mathsf{ready}(x_d)}{\left\{\begin{array}{l}\mathsf{pc}, \mathsf{mem}, \mathsf{reg}, \mathsf{ready}, \mathsf{rt},\\ (\mathsf{rob}_{\mathsf{seq}}, \mathsf{rob}_{\mathsf{head}}), \mathsf{lq}, \mathsf{sq}, \mathsf{bp}, \mathsf{ckpt}, C, \tau\end{array}\right\} \xrightarrow{t} \left\{\begin{array}{l}\mathsf{pc}, \mathsf{mem}, \mathsf{reg}, \mathsf{ready}, \mathsf{rt},\\ (\mathsf{rob}_{\mathsf{seq}}, \mathsf{rob}_{\mathsf{head}} + 1), \mathsf{lq}, \mathsf{sq}, \mathsf{bp}, \mathsf{ckpt}, C, \tau\end{array}\right\}}$$

(Commit-Arithmetic, $\bot$):

$$\frac{\mathsf{rob}_{\mathsf{seq}}[\mathsf{rob}_{\mathsf{head}}] = (\_; \langle \textbf{op}\ x_d, x_a, x_b \rangle; \bot) \qquad \mathsf{ready}(x_d)}{\left\{\begin{array}{l}\mathsf{pc}, \mathsf{mem}, \mathsf{reg}, \mathsf{ready}, \mathsf{rt},\\ (\mathsf{rob}_{\mathsf{seq}}, \mathsf{rob}_{\mathsf{head}}), \mathsf{lq}, \mathsf{sq}, \mathsf{bp}, \mathsf{ckpt}, C, \tau\end{array}\right\} \xrightarrow{t} \left\{\begin{array}{l}\mathsf{pc}, \mathsf{mem}, \mathsf{reg}, \mathsf{ready}, \mathsf{rt},\\ (\mathsf{rob}_{\mathsf{seq}}, \mathsf{rob}_{\mathsf{head}} + 1), \mathsf{lq}, \mathsf{sq}, \mathsf{bp}, \mathsf{ckpt}, C, \tau\end{array}\right\}}$$

(Commit-Branch, $\bot$):

$$\frac{\mathsf{rob}_{\mathsf{seq}}[\mathsf{rob}_{\mathsf{head}}] = (\_; \langle \textbf{branch}\ x_c, x_d \rangle; b) \qquad (\mathsf{rob}_{\mathsf{head}}, \mathsf{pc}', \mathsf{rt}') \notin \mathsf{ckpt}}{\left\{\begin{array}{l}\mathsf{pc}, \mathsf{mem}, \mathsf{reg}, \mathsf{ready}, \mathsf{rt},\\ (\mathsf{rob}_{\mathsf{seq}}, \mathsf{rob}_{\mathsf{head}}), \mathsf{lq}, \mathsf{sq}, \mathsf{bp}, \mathsf{ckpt}, C, \tau\end{array}\right\} \xrightarrow{t} \left\{\begin{array}{l}\mathsf{pc}, \mathsf{mem}, \mathsf{reg}, \mathsf{ready}, \mathsf{rt},\\ (\mathsf{rob}_{\mathsf{seq}}, \mathsf{rob}_{\mathsf{head}} + 1), \mathsf{lq}, \mathsf{sq}, \mathsf{bp}, \mathsf{ckpt}, C, \tau\end{array}\right\}}$$

(COMMIT-LOAD, $\bot$):

$$\frac{\mathsf{rob}[\mathsf{rob}_{\mathsf{head}}] = (\_; \langle \textbf{load}\ x_d, x_a\rangle; \bot; \_) \qquad \mathsf{ready}(x_d)}{\left\{\begin{array}{l}\mathsf{pc}, \mathsf{mem}, \mathsf{reg}, \mathsf{ready}, \mathsf{rt},\\ (\mathsf{rob}_{\mathsf{seq}}, \mathsf{rob}_{\mathsf{head}}), \mathsf{lq}, \mathsf{sq}, \mathsf{bp}, \mathsf{ckpt}, C, \tau\end{array}\right\} \xrightarrow{t} \left\{\begin{array}{l}\mathsf{pc}, \mathsf{mem}, \mathsf{reg}, \mathsf{ready}, \mathsf{rt},\\ (\mathsf{rob}_{\mathsf{seq}}, \mathsf{rob}_{\mathsf{head}}+1), \mathsf{lq}|_{\neq \mathsf{rob}_{\mathsf{head}}}, \mathsf{sq}, \mathsf{bp}, \mathsf{ckpt}, C, \tau\end{array}\right\}}$$

(COMMIT-STORE, $\bot$):

$$\frac{\mathsf{rob}[\mathsf{rob}_{\mathsf{head}}] = (\_; \langle \textbf{store}\ x_a, x_v\rangle; \bot; \_) \qquad \mathsf{ready}(x_a) \qquad \mathsf{ready}(x_v)}{\left\{\begin{array}{l}\mathsf{pc}, \mathsf{mem}, \mathsf{reg}, \mathsf{ready}, \mathsf{rt},\\ (\mathsf{rob}_{\mathsf{seq}}, \mathsf{rob}_{\mathsf{head}}), \mathsf{lq}, \mathsf{sq}, \mathsf{bp}, \mathsf{ckpt}, C, \tau\end{array}\right\} \xrightarrow{t} \left\{\begin{array}{l}\mathsf{pc}, \mathsf{mem}[\mathsf{reg}(x_a) : \mathsf{reg}(x_v)], , \mathsf{reg}, \mathsf{ready}, \mathsf{rt}\\ (\mathsf{rob}_{\mathsf{seq}}, \mathsf{rob}_{\mathsf{head}}+1), \mathsf{lq}, \mathsf{sq}|_{\neq \mathsf{rob}_{\mathsf{head}}}, \mathsf{bp}, \mathsf{ckpt}, C +\!\!+ \mathsf{reg}(x_a), \tau\end{array}\right\}}$$

3.5 The STT Processor Model

Analogously to the in-order processor from Algorithm 1, we model the STT processor as a state machine represented by $\mathsf{STT\_Processor}(P)$ in Algorithm 4. A processor takes a program P as an input, a state σ, a cycle counter t and a termination flag `halt`. The processor starts with an initial state, a cycle counter initialized by 0 and unset termination flag.

We introduce the STT processor logic with the help of the following auxiliary notation for identifying μ-events corresponding to advancing a particular instruction through the pipeline stages:

- The function $\mathsf{fetch\_event}(\theta)$ for a given instruction θ return a corresponding μ-event from $\mathsf{FetchEvents}$.
- The function $\mathsf{execute\_event}(\sigma, i)$ for a given state σ and an index i in the reorder buffer, returns a corresponding μ-event from $\mathsf{ExecuteEvents}$.
- We let the function $\mathsf{commit\_event}(\theta)$ for a given instruction θ return a corresponding μ-event from $\mathsf{CommitEvents}$.

We also introduce the following functions for a given state σ, a μ-event e and a cycle t. We let a function $\mathsf{enabled}(\sigma, e, t)$ return whether e is possible to perform:

$$\mathsf{enabled}(\sigma, e, t) \triangleq (\exists \sigma'.\ \{\sigma\} \xrightarrow{e,t} \{\sigma'\})$$

We let a function $\mathsf{perform}(\sigma, e, t)$ return the state resulting from perfoming the μ-event, when it is possible to perform.

$$\mathsf{perform}(\sigma, e, t) \triangleq \begin{cases} \sigma', & \text{if } \mathsf{enabled}(\sigma, e, t) \text{ holds, and } \sigma' \text{ is such that } \{\sigma\} \xrightarrow{e,t} \{\sigma'\} \\ \text{undefined}, & \text{if } \mathsf{enabled}(\sigma, e, t) \text{ does not hold} \end{cases}$$

Finally, for a μ-event $e \in$ EXECUTEEVENTS, we let $\mathsf{ready}(\sigma, e, t)$ be a boolean function returning true if input registers of the instruction corresponding to performing e all have their ready flags set.

The STT processor performs new μ-events at each cycle with the help of *processor logic*, $\mathsf{STT\_Logic}$, presented in Algorithm 5. We first explain it without the STT protection (lines highlighted in blue), which corresponds to the processor logic of an out-of-order processor. The processor logic takes the current processor state σ and the current cycle count t, and performs μ-events for instructions at each pipeline stage. It starts by taking a snapshot σ_0 of the current state at line 2, which it later uses to check that instructions only get executed if their dependencies were ready at the beginning of the cycle. At lines 4–11, the processor logic attempts to perform commit events for a fixed number COMMIT_WIDTH of instructions, one-by-one starting from the head of the reorder buffer. The committing phase of the processsor logic stops if an instruction is not ready to commit, which is how we ensure that instructions commit in program order. At lines 12–19, the processor logic attempts to fetch a fixed number FETCH_WIDTH of the next instructions in the program. The fetching stops once the first branch instruction is fetched, to account for how in a realistic processor the branch predictor unit is looked up once per cycle. At lines 20–27, the STT processor logic proceeds to performing execute events, ignoring the newly fetched instructions. For each i-th entry in the reorder buffer, we perform a corresponding execute event, provided that its input dependencies had their ready flags set at the beginning of the cycle (verified by checking the snapshot-state σ_0 recorded at the beginning of the cycle). In this way, we disallow execute events to enable other execute events within one cycle. It is also important to note that the execution phase of the STT processor logic assumes that

Algorithm 4: The STT processor algorithm

```
1 Function STT_Processor(P):
2     σ <- σ_init; // initial state
3     t <- 0; // initial cycle counter
4     halt <- false;
5     while ¬halt do
6         (σ, halt) <- STT_Logic(P, σ, t);
7         t <- t + 1;
8     end
```

there is an infinite amount of functional units. We explain why resource contention does not affect the security guarantees of the STT processor in Section 5. The condition for terminating processor execution is simply, the program is fetched completely ($P[\sigma'.\mathsf{pc}] = \bot$) and all existing instructions have committed ($\sigma'.\mathsf{rob}_{\mathsf{head}} = \sigma'.\mathsf{rob}_{\mathsf{tail}}$). (For illustration purposes, at lines 3, 8, 16 and 24 we form a list $\mathcal{T}$ of all events performed at a given cycle.)

We now explain the STT protection in Algorithm 5. Upon fetching a new instruction $P[\sigma.\mathsf{pc}]$ in the current state σ, the processor logic updates taint metadata by calling $\mathsf{taint}(\sigma, P[\sigma.\mathsf{pc}])$ at line 14. As the STT protection classifies loads as *access instructions*, i.e., instructions capable of reading secrets under speculative execution [4], this step corresponds to both ensuring that output registers of loads get tainted and propagating that taint through instruction dependencies. Later on, prior to performing an execute event for an instruction $\sigma.\mathsf{rob}[i]$, the STT processor checks with the help of taint metadata at line 22 whether the execution needs to be delayed. We define this condition with the help of the following predicates:

$$
\begin{aligned}
\mathsf{isExplicitBranch}(e) &\triangleq (e = (\textsc{Execute-Branch-Success}, \_) \vee e = (\textsc{Execute-Branch-Fail}, \_)) \\
\mathsf{isTransmitter}(e) &\triangleq (e = (\textsc{Execute-Load-BeginGetS}, \_)) \\
\mathsf{delayed}(\sigma, e, t) &\triangleq \exists i.\, e = (\_, i) \wedge (\mathsf{isExplicitBranch}(e) \vee \mathsf{isTransmitter}(e)) \wedge \neg \mathsf{noTaintedInputs}(\sigma, i)
\end{aligned}
$$

Thus, an instruction's execute event gets delayed if it is an event either for an explicit branch, as defined by the isExplicitBranch predicate, or for a *transmit* instruction, i.e. if its execution leaks its argument through an explicit covert channel. The STT protection identifies loads also as transmit instructions, so we define the isTransmitter predicate accordingly. For these two kinds of events, the delayed predicate holds if the instruction's inputs are tainted. Lastly, as instructions get executed and committed, the initially tainted output registers of speculative loads might be possible to untaint. To this end, at line 28 the processor logic calls the untaint function, which untaints output registers, where possible, and propagates that information through instruction dependencies (note that some taint tracking schemes, such as the one in Section 3.6, untaint registers automatically). In this analysis, we do not consider implicit branches, which otherwise would also contribute to the definition delayed.

Algorithm 5: The STT processor logic algorithm

```
1  Function STT_Logic(P, σ, t):
2      σ_0 <- σ;
3      // T <- empty trace;
4      for i from 1 to COMMIT_WIDTH do
5          e <- commit_event(σ.rob[σ.rob_head]);
6          if enabled(σ, e, t) then
7              σ <- perform(σ, e, t);
8              // T <- T ++ e;
9          else
10             break;
11     end
12     for i from 1 to FETCH_WIDTH do
13         e <- fetch_event(P[σ.pc]);
14         σ.τ <- taint(σ, P[σ.pc]);
15         σ <- perform(σ, e, t);
16         // T <- T ++ e;
17         if e = (FETCH-BRANCH, ⊥) then
18             break;
19     end
20     for i from σ.rob_head to σ_0.rob_tail − 1 do
21         e <- execute_event(σ, i);
22         if enabled(σ, e, t) and ready(σ_0, e, t) and (not delayed(σ_0, e, t)) then
23             σ <- perform(σ, e, t);
24             // T <- T ++ e;
25             if e = (EXECUTE-BRANCH-FAIL, i) then
26                 break;
27     end
28     σ <- untaint(σ);
29     halt <- (P[σ.pc] = ⊥) ∧ (σ.rob_head = σ.rob_tail)
30     return (σ, halt);
```

3.6 The YRoT-Based Taint Tracking Scheme

We now present the taint tacking scheme employed by the STT processor implementation (see Section 7 in [4]). The key challenge that the implementation overcomes is how to implement the untaint operation efficiently. In the following, we present a scheme that implements taint and untaint, and is realizable in hardware.

We make a key observation that helps implement the new taint tracking scheme. Since instructions get committed in program order, the arguments for an instruction Θ remain tainted until the youngest instruction that is causing the taint for Θ stops being speculative. We call this instruction the *youngest root of taint* (YRoT) of Θ.

With this approach, we track the position of the YRoT for each dynamic instruction in the reorder buffer. To this end, in our model the taint metadata includes a function yrot mapping reorder buffer indices onto corresponding instructions' youngest roots of taint. Metadata also includes extensions to the renaming table, $\mathsf{rt}_{\mathsf{YRoT}}$ and $\mathsf{rt}_{\mathsf{LL}}$ (Section 7 in [4] names analogous extensions `AccessInstrIdx` and `YRoT` respectively), which are used in adding new entries to yrot. More formally, we instantiate taint metadata as a combination of several structures. We let $\tau = (\mathsf{yrot}, \mathsf{rt}_{\mathsf{YRoT}}, \mathsf{rt}_{\mathsf{LL}}) \in \mathsf{Taint} = (\mathbb{N} \to (\mathbb{N} \cup \{-1\})) \times (\mathsf{RegID} \to (\mathbb{N} \cup \{-1\})) \times (\mathsf{RegID} \to (\mathbb{N} \cup \{\bot\}))$, where:

- yrot is an extension to the reorder buffer given by a function mapping indexes of instructions in the reorder buffer onto indexes of their youngest roots of taint. Thus, for an instruction $\mathsf{rob}[i]$, $\mathsf{yrot}(i)$ is the youngest root of taint (or -1, if $\mathsf{rob}[i]$ does not have tainted inputs).
- $\mathsf{rt}_{\mathsf{YRoT}}$ is an extension to the renaming table given by a function mapping logical registers onto indexes in the reorder buffer. For a logical register r, $\mathsf{rt}_{\mathsf{YRoT}}(r)$ is the youngest root of taint of the last fetched instruction whose output is register r.
- $\mathsf{rt}_{\mathsf{LL}}$ is an extension to the renaming table given by a function mapping logical registers onto indexes in the reorder buffer. For a logical register r, $\mathsf{rt}_{\mathsf{LL}}(r)$ is the index in the reorder buffer of the last fetched instruction whose output is register r, if it is an access instruction such as a load, or $\bot$ otherwise.

At each cycle, fetching an instruction θ triggers tainting registers as specified by the function $\mathsf{taint}(\sigma, \theta)$ that we further define. To this end, we first introduce two auxiliary ones. Given a register r, we identify the instruction responsible for the taint of r as either $\mathsf{rt}_{\mathsf{LL}}(r)$ (when it is defined) or $\mathsf{rt}_{\mathsf{YRoT}}(r)$:

$$\mathsf{taintingInstr}(r) \triangleq \text{if } \mathsf{rt}_{\mathsf{LL}}(r) = \bot \text{ then } \mathsf{rt}_{\mathsf{YRoT}}(r) \text{ else } \mathsf{rt}_{\mathsf{LL}}(r)$$

Algorithm 6: untaint in the YRoT-based taint tracking scheme

```
1 Function untaint(σ):
2 |   return σ;
```

We also identify the youngest root of taint for a newly fetched instruction θ with the help of a function $\mathsf{newYRoT}(\theta)$ defined as follows:

$$\mathsf{newYRoT}(\theta) \triangleq \begin{cases} \bot, & \text{if } \theta = \langle \textbf{immed } r_d, k\rangle \\ \max(\mathsf{taintingInstr}(r_a), \mathsf{taintingInstr}(r_b)), & \text{if } \theta = \langle \textbf{op } r_d, r_a, r_b\rangle \\ \mathsf{taintingInstr}(r_a), & \text{if } \theta = \langle \textbf{load } r_d, r_a\rangle \\ \max(\mathsf{taintingInstr}(r_c), \mathsf{taintingInstr}(r_d)), & \text{if } \theta = \langle \textbf{branch } r_c, r_d\rangle \\ \max(\mathsf{taintingInstr}(r_a), \mathsf{taintingInstr}(r_v)), & \text{if } \theta = \langle \textbf{store } r_a, r_v\rangle \end{cases}$$

We let $\mathsf{taint}(\sigma, \theta)$ return $(\mathsf{yrot}', \mathsf{rt}'_{\mathsf{YRoT}}, \mathsf{rt}'_{\mathsf{LL}})$, provided that $\sigma.\tau = (\mathsf{yrot}, \mathsf{rt}_{\mathsf{YRoT}}, \mathsf{rt}_{\mathsf{LL}})$ and the following is the case:

- $\mathsf{yrot}' = \mathsf{yrot}[\sigma.\mathsf{rob}_{\mathsf{tail}} \mapsto \mathsf{newYRoT}(\theta)]$;
- When r_d is the output register of θ (i.e., when θ is either $\langle \textbf{load } r_d, \_\rangle$ or $\langle \textbf{op } r_d, \_, \_\rangle$):
  - $\mathsf{rt}'_{\mathsf{YRoT}} = \mathsf{rt}_{\mathsf{YRoT}}[r_d \mapsto \mathsf{newYRoT}(\theta)]$
  - $\mathsf{rt}'_{\mathsf{LL}} = \mathsf{rt}_{\mathsf{LL}}[r_d \mapsto (\text{if } \theta = \langle \textbf{load } r_d, \_\rangle \text{ then } \sigma.\mathsf{rob}_{\mathsf{tail}} \text{ else } \bot)]$.

At each cycle, the execution of a reorder buffer entry $\sigma.\mathsf{rob}[i]$ is predicated upon the condition $\mathsf{noTaintedInputs}(\sigma, i)$, which is a predicate that holds whenever dependencies of an instruction $\sigma.\mathsf{rob}[i]$ with the given index i are not tainted. To this end, the YRoT-based taint tracking scheme checks whether $\sigma.\mathsf{rob}[i]$ has no root of taint or whether its YRoT is no longer speculative:

$$\mathsf{noTaintedInputs}(\sigma, i) \triangleq (\sigma.\tau.\mathsf{yrot}(i) = \bot) \vee \neg\mathsf{underSpec}(\sigma.\mathsf{ckpt}, \sigma.\tau.\mathsf{yrot}(i))$$

The function $\mathsf{untaint}(\sigma)$ of the YRoT-based taint tracking scheme is simply an identity function, as presented in Algorithm 6. The key advantage of the YRoT-based taint tracking scheme is that it makes untainting automatic. Indeed, as the check of taint in noTaintedInputs makes use only of metadata that gets updated through the normal execution of instructions, there is no need to explicitly modify any taint metadata in untaint: $\mathsf{noTaintedInputs}(\sigma, i)$ starts holding once the youngest root of taint of $\sigma.\mathsf{rob}[i]$ stops being speculative. We note that in the STT processor implementation logic indicating which instructions reach the point when they are no longer under speculation is provided by prior work on InvisiSpec [2].

4 Security analysis

In this section, we formalize our assumptions about correctness of the STT processor, and build our security analysis on top of them. In Section 4.1, we formulate correctness of the STT processor as a correspondence between its committed state and the state of the in-order processor. In Section 4.2, we show that the STT processor provides the following non-interference security guarantee: at each step of its execution, the value of a register tainted by a transient (bound to squash) access instruction does not influence the future of the execution. To this end, we formalize which instructions are transient by leveraging the correspondence between the STT processor and the in-order processor, and then we prove that the STT processor provides non-interference.

4.1 Correctness assumptions for the STT processor

We formalize correctness of the STT processor by requiring that changes to the state performed by committed instructions simulate the in-order processor. Thus, the in-order processor serves as a specification for the STT processor.

Definition 5. *An in-order processor state $\Sigma \in \mathsf{State}_{\mathsf{InO}}$ is the* committed state *for a STT processor state $\sigma \in \mathsf{State}_{\mathsf{STT}}$, written $\sigma \sim \Sigma$, if Σ is the result of executing a sequence of instructions corresponding to entries in the reorder buffer $\sigma.\mathsf{rob}[0], \sigma.\mathsf{rob}[1], \ldots, \sigma.\mathsf{rob}[\mathsf{rob}_{\mathsf{head}} - 1]$ from the initial state Σ_{init}.*

Definition 6. *Given a program P, we say that the STT processor running P from the state σ simulates the in-order processor running P from the state Σ, written* $\mathsf{sim}(P, \sigma, \Sigma)$*, if the following is the case:*

(a) $\sigma \sim \Sigma$*,*

(b) if the execution of P terminated on the STT processor in σ, then so did the in-order processor in Σ, and otherwise

(c) for every STT processor state σ' and for every STT transition that advances the state from σ to σ', there exists an in-order processor state Σ' (possibly equal to Σ) and zero or more transitions of the in-order processor advancing Σ to Σ' so that $\mathsf{sim}(P, \sigma', \Sigma')$*.*

We also say that the STT processor simulates the in-order processor, if for every program P and the initial states σ_{init} and Σ_{init}, $\mathsf{sim}(P, \sigma_{\mathsf{init}}, \Sigma_{\mathsf{init}})$ *holds.*

Intuitively, one can understand the simulation above as the STT processor and the in-order processor running synchronously, with the latter executing instructions entirely whenever the STT processor commits them. In the rest of our analysis, we assume that the STT processor simulates the in-order processor.

4.2 The Non-Interference Theorem

In our formal analysis of the STT processor's security, we overcome the challenge of identifying doomed registers with the help of the in-order processor. We extend our model of computation by considering the STT processor and the in-order processor running simultaneously. We synchronize their steps as follows: whenever the STT processor fetches an instruction, the in-order processor executes it; and if the instruction is a branch resulting in an outcome different from the STT processor's prediction, the in-order processor remains idle until the STT processor squashes due to this branch (in other words, the in-order processor waits until the STT processor resolves the corresponding branch instruction). We refer to the resulting state machine as an *extended STT processor*. Note that the extension is only a part of the non-interference analysis, and does not require any changes to the implementation of the STT processor.

Knowing that the STT processor simulates the in-order processor, we are able to identify in the extended STT processor which branch predictions are correct based on the branch outcomes in the in-order processor execution. We leverage that knowledge to identify transient instructions. Intuitively, a register r is doomed if it becomes considered tainted after what the in-order processor identifies as a misprediction.

More formally, we consider the extended processor, whose execution is characterized by tuples $\kappa = (\sigma, \Sigma, \mathsf{mispredicted}, \mathsf{doomed})$ which we call *extended states*. They consist of: the STT processor state σ, the in-order processor state Σ, as well as two additional forms of *auxiliary state* (that is, the state that is used to refer to the past of the execution and only exists as a part of a proof): a boolean flag $\mathsf{mispredicted} \in \mathbb{B}$, which indicates whether the STT processor has an in-flight branch with a mispredicted direction, and a map $\mathsf{doomed} : \mathsf{PRegID} \to \mathbb{B}$, which indicates whether a given physical register is doomed.

The extended processor maintains auxiliary state as follows. For the initial extended state κ_{init}, we let $\mathsf{mispredicted}$ be false and doomed map every register to false. To update the extended state, we lift the STT processor transition rules. We overload the predicate $\mathsf{enabled}$ (Section 3.5) by letting event e be enabled in κ at a cycle t, if it is enabled in the corresponding STT state (recall that we use $\_$ as a placeholder for variables whose names are irrelevant in the context):

$$\mathsf{enabled}(\kappa, e, t) \triangleq (\exists \sigma.\, \kappa = (\sigma, \_, \_, \_) \wedge \mathsf{enabled}(\sigma, e, t))$$

We lift ready and $\mathsf{delayed}$ to extended states analogously. It is easy to see that extended processor performs the same events as the STT processor. It also updates the STT state in the same way, but also updates the rest of the extended state as we further specify. To this end, we overload the function $\mathsf{perform}$. Given $\kappa = (\sigma, \Sigma, \mathsf{mispredicted}, \mathsf{doomed})$, we let $\mathsf{perform}(\kappa, e, t)$ return $\kappa' = (\sigma', \Sigma', \mathsf{mispredicted}', \mathsf{doomed}')$ constrained as follows:

- $\sigma' = \mathsf{perform}(\sigma, e, t)$.
- Σ' is either $\mathsf{perform}(\Sigma, \mathsf{matching\_event}(P[\Sigma.\mathsf{pc}]), t)$, if $e \in \mathsf{FetchEvents}$ and $\mathsf{mispredicted} = \mathsf{false}$ both hold, or remains unchanged from Σ, otherwise.
- $\mathsf{mispredicted}'$ is (a) set to true, if $e = (\textsc{Fetch-Branch}, i)$ holds and the STT processor mispredicts the branch direction, or (b) set to false, if $e = (\textsc{Execute-Branch-Fail}, i)$ holds and $\sigma.\mathsf{rob}[i]$ corresponds to the branch the in-order processor is idle at, or (c) unchanged from $\mathsf{mispredicted}$, otherwise.

- doomed' is either (a) a function $\mathsf{doomed}[x \mapsto \mathsf{true}]$, if $\mathsf{mispredicted}$ holds, $e \in \mathsf{FetchEvents}$ holds, x is the physical output register of the newly fetched instruction $P[\sigma.\mathsf{pc}]$, and $\mathsf{taint}(\sigma, P[\sigma.\mathsf{pc}])$ results in $\mathsf{tainted}(\sigma, x)$ holding, or (b) a function that maps every physical register to false, if $e = (\textsc{Execute-Branch-Fail}, i)$ holds and $\sigma.\mathsf{rob}[i]$ corresponds to the branch the in-order processor is idle at, or (c) unchanged from doomed, otherwise.

When the extended processor updates the state according to the rules above, it ensures the following property formalizing the intuitive definition of doomed:

Definition 7. *Given a physical register $x \in \mathsf{PRegID}$, we say that $\mathsf{doomed}(x)$ holds if and only if there is a reorder buffer index i such that $\sigma.\mathsf{rob}_{\mathsf{head}} \leq i < \sigma.\mathsf{rob}_{\mathsf{tail}}$, x is an output register of $\sigma.\mathsf{rob}[i]$, $\mathsf{tainted}(\sigma, x)$, $\mathsf{underSpec}(\sigma.\mathsf{ckpt}, i)$ and $\mathsf{mispredicted}$ all hold.*

Definition 8. *Given two extended states $\kappa = (\sigma, \Sigma, \mathsf{mispredicted}, \mathsf{doomed})$ and $\kappa' = (\sigma', \Sigma', \mathsf{mispredicted}', \mathsf{doomed}')$, we say that κ and κ' are low-equivalent, written $\kappa \approx \kappa'$, if σ_1 and σ_2 only differ by the values of doomed registers, $\Sigma = \Sigma'$, $\mathsf{mispredicted} = \mathsf{mispredicted}'$ and $\mathsf{doomed} = \mathsf{doomed}'$.*

We can now re-state formally our non-interference theorem. (Recall from Section 1 that we parameterize the theorem by an observability function view, which models the adversary's view [3]: that is, which projects μ-event traces onto the parts the adversary can observe.)

Theorem 1. *At any cycle t, given two extended states κ_1 and κ_2 such that $\kappa_1 \approx \kappa_2$ holds, if $\mathcal{T}_1$ and $\mathcal{T}_2$ are the μ-event traces the STT processor logic performs at the cycle t from κ_1 and κ_2 respectively, then the following holds:*

(a) $\mathsf{view}(\mathcal{T}_1) = \mathsf{view}(\mathcal{T}_2)$ holds, and

(b) for the extended states κ_1' and κ_2' resulting from executing $\mathcal{T}_1$ and $\mathcal{T}_2$ respectively, $\kappa_1' \approx \kappa_2'$ holds.

We prove the theorem for the two taint tracking schemes from Section 3.3 and Section 3.6. We encapsulate the key steps of the proof of Theorem 1 in two lemmas. First, Lemma 9 states that the STT processor logic preserves low-equivalence when it executes any μ-event.

Lemma 9. *Consider two runs of the STT processor logic from Algorithm 5 that started at a cycle t from extended states $\kappa_1^{(0)}$ and $\kappa_2^{(0)}$ such that $\kappa_1^{(0)} \approx \kappa_2^{(0)}$. If during the cycle, the STT processor logic executes any μ-event e from any κ_1 and κ_2 such that $\kappa_1 \approx \kappa_2$ holds, resulting in $\kappa_1' = (\mathsf{perform}(\kappa_1, e, t)$ and $\kappa_2' = \mathsf{perform}(\kappa_2, e, t)$ respectively, then $\kappa_1' \approx \kappa_2'$.*

The STT processor never allows values of doomed registers influence the program counter. Intuitively, the branch predictions are never a function of tainted registers, and branch resolution happens only once branch instruction dependencies are untainted. Lemma 9 captures this intuition, as it shows that all non-doomed parts of the state are never influenced by doomed-register values.

The second lemma states that the STT processor logic running from low-equivalent states can execute the same μ-events in two runs.

Lemma 10. *Consider two runs of the STT processor logic from Algorithm 5 that started a cycle t from low-equivalent extended states $\kappa_1^{(0)}$ and $\kappa_2^{(0)}$ ($\kappa_1^{(0)} \approx \kappa_2^{(0)}$ holds). If κ_1 and κ_2 are extended states the STT processor logic reaches during the two runs such that $\kappa_1 \approx \kappa_2$ holds, then the STT processor logic in Algorithm 5 can perform any event e in κ_1 if and only if it can perform the event in κ_2.*

Proof of Theorem 1. We assume the strongest possible observability function $\mathsf{view}(\mathcal{T}) \triangleq \mathcal{T}$. We consider two runs of the STT processor logic from Algorithm 5 at a cycle t from states $\kappa_1^{(0)}$ and $\kappa_2^{(0)}$ such that $\kappa_1^{(0)} \approx \kappa_2^{(0)}$ holds. Let $\mathcal{T}_1$ and $\mathcal{T}_2$ take form of a sequence of commit events followed by fetch events followed by execute events.

We first argue that the same commit events occur in $\mathcal{T}_1$ and $\mathcal{T}_2$. At lines 4–11, the processor logic attempts to perform a fixed number of commit events in a loop, which starts from low-equivalent extended states $\kappa_1^{(0)}$ and $\kappa_2^{(0)}$. As the two states have the same reorder buffer, the same event $\mathsf{commit\_event}(\mathsf{rob}[\mathsf{rob}_{\mathsf{head}}])$ is to be performed in the two runs. By Lemma 10, the STT processor logic either performs the event in both runs or in neither. Moreover, if the event is performed, by Lemma 9, its updates to $\kappa_1^{(0)}$ and $\kappa_2^{(0)}$ preserve low-equivalence. By applying the same reasoning at each iteration of the loop, we conclude that the same events are committed in the two runs in the same order, and therefore that the resulting extended states are low-equivalent.

We then argue that the same fetch events occur in $\mathcal{T}_1$ and $\mathcal{T}_2$. At lines 12–19, the processor logic attempts to perform a fixed number of fetch events in a loop, which starts from two low-equivalent extended states. As the two

states have the same program counter, the same event $\mathsf{fetch\_event}(P[\mathsf{pc}])$ is to be performed in the two runs. By Lemma 10, the STT processor logic either performs the event in both runs or in neither. Moreover, if the event is performed, by Lemma 9, its updates to the extended states preserve low-equivalence. By applying the same reasoning at each iteration of the loop, we conclude that the same events are fetched in the same order in the two runs, and therefore that the resulting extended states are low-equivalent.

We now argue that the same execute events occur in $\mathcal{T}_1$ and $\mathcal{T}_2$. At lines 20–27, the processor logic iterates over a slice of the reorder buffer. As the loop starts from two low-equivalent states, the same slice of the reorder buffer is iterated over and the same event $\mathsf{execute\_event}(\sigma, \mathsf{rob}_{\mathsf{head}})$ is to be performed in the two runs at the first loop iteration. By Lemma 10, the STT processor logic either performs the event in both runs or in neither. Moreover, if the event is performed, by Lemma 9, its updates to the extended states preserve low-equivalence. By applying the same reasoning at each iteration of the loop, we conclude that the same events are executed in the two runs in the same order, and therefore that the resulting extended states are low-equivalent.

Finally, we argue that low-equivalency of the extended states is maintained by the $\mathsf{untaint}()$ performed in line 28. This trivially holds for the YRoT taint tracking scheme, since its $\mathsf{untaint}()$ is the identity function. For the basic taint tracking scheme, $\mathsf{untaint}()$ (Algorithm 3) iterates over a slice of the reorder buffer. As the loop starts from two low-equivalent states, the same slice of the reorder buffer is iterated over. Each iteration invokes $\mathsf{untaintInstr}()$ on the same reorder buffer index. The output of $\mathsf{untaintInstr}$ is determined by the taint metadata τ, the contents of the reorder buffer, and ckpt, all of which are identical in both extended states. Therefore, each iteration of the loop preserves low-equivalence of the extended states.

Overall, we have showed that $\mathcal{T}_1 = \mathcal{T}_2$, and that the resulting states κ_1' and κ_2' are low-equivalent, which concludes the proof of the theorem. □

4.3 Auxiliary proofs

In the rest of this section, we prove Lemma 9 and Lemma 10. We begin with stating several simple invariant properties of the extended processor.

Proposition 11. *For every program P, the following holds of the extended state $\kappa = (\sigma, \Sigma, \mathsf{mispredicted}, \mathsf{doomed})$ at each point in the execution:*

(a) For any reorder buffer indices i and j such that $0 \leq i, j < \sigma.\mathsf{rob}_{\mathsf{tail}}$, if the output register of $\sigma.\mathsf{rob}[i]$ is an input register of $\sigma.\mathsf{rob}[j]$, then $i < j$ holds.

(b) For any reorder buffer index i such that $\sigma.\mathsf{rob}_{\mathsf{head}} \leq i < \sigma.\mathsf{rob}_{\mathsf{tail}}$, if $\mathsf{underSpec}(\sigma.\mathsf{ckpt}, i)$ holds then there is j such that $\sigma.\mathsf{rob}_{\mathsf{head}} \leq j \leq i$ and $\sigma.\mathsf{rob}[j]$ is a dynamic branch instruction.

(c) For every index i in the reorder buffer such that $\sigma.\mathsf{rob}_{\mathsf{head}} \leq i$, if $\sigma.\mathsf{rob}[i]$ has an output register x and any of its input registers is doomed, then x is also doomed.

Proof sketch. Proposition 11 can be shown by induction on the length of the STT processor execution. We only highlight the intuition behind each part of it. The parts (a) and (b) assert simple invariants maintained by the out-of-order processor: (a) states the order of instruction dependencies, and (b) states that an instruction can only be speculative if it follows an in-flight branch instruction. The part (c) states that doomedness of registers propagates through instruction dependencies, which follows from a combination of Definition 7 with (a) and (b). □

Proposition 12. *Consider a cycle of the STT processor logic from Algorithm 5 that starts from an extended state κ_0, and let κ be any state the algorithm reaches during the cycle. Then for any event $e \in \mathsf{ExecuteEvents}$, $\mathsf{delayed}(\kappa, e, t)$ implies $\mathsf{delayed}(\kappa_0, e, t)$.*

Proof sketch. We show that the proposition holds for the two taint tracking schemes from Section 3.3 and Section 3.6. Consider any event $e = (\_, i) \in \mathsf{ExecuteEvents}$. We do a proof by contrapositive: we assume that $\mathsf{delayed}(\kappa_0, e, t)$ does not hold, and show that neither does $\mathsf{delayed}(\kappa, e, t)$.

When $\mathsf{delayed}(\kappa_0, e, t)$ does not hold, we have:

$$(\neg\mathsf{isExplicitBranch}(e) \wedge \neg\mathsf{isTransmitter}(e)) \vee \mathsf{noTaintedInputs}(\kappa_0, i)$$

When neither $\mathsf{isExplicitBranch}(e)$ nor $\mathsf{isTransmitter}(e)$ holds, we immediately get that $\mathsf{delayed}(\kappa, e, t)$ does not hold. In the rest of the proof, we consider the case when one of the two holds, so the following is true:

$$(\mathsf{isExplicitBranch}(e) \vee \mathsf{isTransmitter}(e)) \wedge \mathsf{noTaintedInputs}(\kappa_0, i)$$

We first consider the basic taint tracking scheme. By definition of $\mathsf{noTaintedInputs}(\kappa_0, i)$, dependencies of the instruction $\mathsf{rob}[i]$ are not tainted in taint metadata τ. Algorithm 5 only uses taint metadata through interface functions taint, $\mathsf{noTaintedInputs}$ and $\mathsf{untaint}$, and only fetching an instruction triggers tainting its output register. From transition rules for fetching, we know that an output register is always generated fresh and cannot coincide with physical registers already in use. Hence, we can conclude that dependencies of $\mathsf{rob}[i]$ remain untainted, so $\mathsf{noTaintedInputs}(\kappa, i)$ holds and $\mathsf{delayed}(\kappa, e, t)$ does not hold.

We now consider the YRoT taint tracking scheme. By definition of $\mathsf{noTaintedInputs}(\kappa_0, i)$, either $\kappa_0.\mathsf{yrot}(i) = \bot$ or $\neg\mathsf{underSpec}(\kappa_0.\mathsf{ckpt}, \kappa_0.\mathsf{yrot}(i))$ holds. Algorithm 5 only uses taint metadata through interface functions taint, $\mathsf{noTaintedInputs}$ and $\mathsf{untaint}$. Note that $\mathsf{yrot}(i)$, once assigned, cannot be changed, and that reorder buffer indices are never reused. Therefore, if $\kappa_0.\mathsf{yrot}(i) = \bot$, it will remain such throughout the cycle, and $\kappa.\mathsf{yrot}(i) = \bot$ will hold too. In that case, $\mathsf{noTaintedInputs}(\kappa, i)$ holds and so $\mathsf{delayed}(\kappa, e, t)$ does not hold. It remains to consider the case when $\neg\mathsf{underSpec}(\kappa_0.\mathsf{ckpt}, \kappa_0.\mathsf{yrot}(i))$. However, an instruction not under speculation cannot become speculative again, as new branch instructions are appended to the reorder buffer. In this case, $\mathsf{noTaintedInputs}(\kappa, i)$ holds and $\mathsf{delayed}(\kappa, e, t)$ does not hold, which concludes the proposition. □

Proof of Lemma 9. Let us consider any cycle t and two extended states κ_1 and κ_2 such that $\kappa_1 \approx \kappa_2$ holds. By Definition 8, $\kappa_1 \approx \kappa_2$ implies that the two extended states are different only by values of doomed registers. In particular, κ_1 and κ_2 take the following form: $\kappa_1 = (\sigma_1, \Sigma, \mathsf{mispredicted}, \mathsf{doomed})$ and $\kappa_2 = (\sigma_2, \Sigma, \mathsf{mispredicted}, \mathsf{doomed})$, and σ_1 and σ_2 are different only by values of doomed registers (as the rest of the state is the same, we write rob instead of $\sigma_1.\mathsf{rob}$ and $\sigma_2.\mathsf{rob}$ in the proof). As the same registers are doomed in the two extended states, to conclude $\kappa_1' \approx \kappa_2'$, it suffices to consider each event and show that values of doomed registers are only ever used to update doomed registers, and not the rest of the state.

Firstly, we consider any event $e \in \mathsf{FetchEvents}$, which Algorithm 5 performs unconditionally. None of them use values of registers in their update to the state, so it is easy to see that $\kappa_1' \approx \kappa_2'$ holds.

Secondly, we consider any event $e \in \mathsf{CommitEvents}$. Algorithm 5 performs e provided that it is enabled, so we need to show:

$$\mathsf{enabled}(\kappa_1, e, t) \wedge \mathsf{enabled}(\kappa_2, e, t) \implies (\kappa_1' = \mathsf{perform}(\kappa_1, e, t) = \mathsf{perform}(\kappa_2, e, t) = \kappa_2')$$

All events but $(\textsc{Commit-Store}, \bot)$ do not use values of registers in their update to the state, so $\kappa_1' \approx \kappa_2'$ holds immediately for them. We now consider the case when $e = (\textsc{Commit-Store}, \bot)$ and assume that it is enabled both in κ_1 and κ_2. Note that it is enabled only when the front of the reorder buffer is a dynamic store instruction. By Property 11(b), neither $\mathsf{underSpec}(\sigma_1)$ nor $\mathsf{underSpec}(\sigma_2)$ hold then. By Definition 7, in both κ_1 and κ_2 there are no doomed registers (in particular, the register value written into memory by the event is not doomed). Therefore, $(\textsc{Commit-Store}, \bot)$ does not update the state with values of doomed registers, so $\kappa_1' \approx \kappa_2'$ holds.

It remains to consider events from $\mathsf{ExecuteEvents}$. Algorithm 5 performs $e \in \mathsf{ExecuteEvents}$ provided that it is enabled in the current state, and ready and not delayed in the state at the beginning of the cycle. We omit the readiness requirement and prove a stronger statement:

$$\begin{aligned}\mathsf{enabled}(\kappa_1, e, t) \wedge \neg\mathsf{delayed}(\kappa_1^{(0)}, e, t) \wedge \mathsf{enabled}(\kappa_2, e, t) \wedge \neg\mathsf{delayed}(\kappa_2^{(0)}, e, t)\\ \implies (\kappa_1' = \mathsf{perform}(\kappa_1, e, t) = \mathsf{perform}(\kappa_2, e, t) = \kappa_2')\end{aligned}$$

We assume that the premise of the above holds. By Proposition 12, $\neg\mathsf{delayed}(\kappa_1^{(0)}, e, t)$ implies $\neg\mathsf{delayed}(\kappa_1, e, t)$, and $\neg\mathsf{delayed}(\kappa_2^{(0)}, e, t)$ implies $\neg\mathsf{delayed}(\kappa_2, e, t)$. Hence, we get:

$$\mathsf{enabled}(\kappa_1, e, t) \wedge \neg\mathsf{delayed}(\kappa_1, e, t) \wedge \mathsf{enabled}(\kappa_2, e, t) \wedge \neg\mathsf{delayed}(\kappa_2, e, t),$$

and we prove $(\kappa_1' = \mathsf{perform}(\kappa_1, e, t) = \mathsf{perform}(\kappa_2, e, t) = \kappa_2')$ by considering each possible execute event separately in the following.

The events $(\textsc{Execute-Branch-Success}, i)$ and $(\textsc{Execute-Branch-Fail}, i)$ correspond to executing a dynamic instruction $\mathsf{rob}[i] = (\_, \langle \mathbf{branch}\ x_c, x_d \rangle, \_)$. Recall that these two events are not delayed in κ_1 and κ_2, meaning that $\mathsf{noTaintedInputs}$ holds of them and the instruction inputs x_c and x_d are not tainted. Consequently, by Definition 7, they are not doomed either, which together with low-equivalence of κ_1 and κ_2 gives that $\sigma_1.\mathsf{reg}(x_c) = \sigma_2.\mathsf{reg}(x_c)$ and $\sigma_1.\mathsf{reg}(x_d) = \sigma_2.\mathsf{reg}(x_d)$. The events $(\textsc{Execute-Branch-Success}, i)$ and $(\textsc{Execute-Branch-Fail}, i)$ do not use other registers in their update to the state. Therefore, we conclude that they do not update the state with values of doomed registers, and $\kappa_1' \approx \kappa_2'$ holds.

The event $(\textsc{Execute-Arithmetic}, i)$ corresponds to executing a dynamic instruction $\mathsf{rob}[i] = (\_, \langle \mathbf{op}\ x_d, x_a, x_b \rangle, \_)$. By Proposition 11(c), if x_a or x_b is doomed, so is x_d. The event $(\textsc{Execute-Arithmetic}, i)$

only uses x_a and x_b to store the result of the instruction in x_d as a part of its update the state. Therefore, we conclude that (EXECUTE-ARITHMETIC, i) only uses values of doomed registers to update other doomed registers, so $\kappa_1' \approx \kappa_2'$ holds.

The event (EXECUTE-LOAD-BEGINGETS, i) corresponds to executing a dynamic instruction $\mathsf{rob}[i] = (\_, \langle \textbf{load}\ x_d, x_a \rangle, \_)$. Recall that e is not delayed in κ_1 and κ_2, meaning that $\mathsf{noTaintedInputs}$ holds of them and the instruction input x_a is not tainted. Consequently, by Definition 7, x_a is not doomed either, which together with low-equivalence of κ_1 and κ_2 gives that $\sigma_1.\mathsf{reg}(x_a) = \sigma_2.\mathsf{reg}(x_a)$. The event (EXECUTE-LOAD-BEGINGETS, i) only uses x_a to update the state: (1) it extends the list of cache accesses C with the memory address $\mathsf{reg}(x_a)$, and (2) it updates the load queue entry of the dynamic load with the cycle in which the load's execution ends, $t_{\mathsf{end}} = t + \mathsf{LoadLat}(C, \mathsf{reg}(x_a))$. Therefore, we conclude that (EXECUTE-LOAD-BEGINGETS, i) does not update the state with values of doomed registers, and $\kappa_1' \approx \kappa_2'$ holds.

The event (EXECUTE-LOAD-ENDGETS, i) only happens after (EXECUTE-LOAD-BEGINGETS, i), so the input registers of the load are not doomed, as we have shown before. The event (EXECUTE-LOAD-ENDGETS, i) does not use other registers in its update to the state, so it is easy to see that $\kappa_1' \approx \kappa_2'$ holds.

The event (EXECUTE-LOAD-COMPLETE, i) only happens after (EXECUTE-LOAD-BEGINGETS, i), so the input registers of the load are not doomed, as we have shown before. As the latter was not delayed in κ_1 and κ_2, $\mathsf{noTaintedInputs}$ holds of them and the instruction input registers are not tainted. Let $\mathsf{rob}[i]$ take form of a dynamic instruction $(\_, \langle \textbf{load}\ x_d, x_a \rangle, \_)$. By Definition 7, x_a is not doomed. The event (EXECUTE-LOAD-ENDGETS, i) uses x_a and, possibly, another register in its update to the state in the case of store-to-load forwarding. Let us consider the case when store-to-load forwarding takes place: i.e., when there exists a reorder buffer index j such that $\mathsf{CanForward}(i, x_a, j, x_v)$ holds (here x_v is the register whose value the event uses to update the state). If x_v is not doomed, we can immediately conclude $\kappa_1' \approx \kappa_2'$. We further consider the case when $\mathsf{doomed}(x_v)$ holds. By definition of $\mathsf{CanForward}(i, x_a, j, x_v)$, we get:

$$\begin{aligned} j \in \mathsf{sq} \wedge j < i \wedge \exists x_b.\, \mathsf{rob}[j] = (\_; \langle \textbf{store}\ x_b, x_v \rangle; \bot; \_) \wedge \mathsf{ready}(x_b) \wedge \mathsf{ready}(x_v) \wedge \mathsf{reg}(x_a) = \mathsf{reg}(x_b) \\ \wedge\, (\forall j', x_b'.\, j < j' < i \wedge \mathsf{rob}[j'] = (\_; \langle \textbf{store}\ (x_b'), \_ \rangle; \bot; \_) \implies \mathsf{reg}(x_a) \neq \mathsf{reg}(x_b')) \end{aligned}$$

In particular, x_v is an input register for a store instruction $\mathsf{rob}[j]$, and $j < i$. By Definition 7 of $\mathsf{doomed}(x_v)$, there is an instruction $\mathsf{rob}[j']$ in the reorder buffer with the index j', $\mathsf{underSpec}(\mathsf{ckpt}, j')$ and $\mathsf{mispredicted}$ all hold. Since the output of $\mathsf{rob}[j']$ is an input of $\mathsf{rob}[j]$, Proposition 11(a) gives us that $j' < j < i$. It is easy to see from definition of $\mathsf{underSpec}(\mathsf{ckpt}, j')$ that $\mathsf{underSpec}(\mathsf{ckpt}, i)$ holds as well: indeed, all instructions following j' in the reorder buffer are also under the same misspeculation. Then, by Definition 7, $\mathsf{doomed}(x_d)$ holds. Overall, we have shown that (EXECUTE-LOAD-COMPLETE, i) only uses values of doomed registers to update other doomed registers, so $\kappa_1' \approx \kappa_2'$ holds.

□

Proposition 13. *At any cycle t, for any μ-event e and any two extended states κ_1 and κ_2, if $\kappa_1 \approx \kappa_2$ holds, then any event $e \in \mathsf{ExecuteEvents}$ is ready (delayed) in κ_1 if and only if it is ready (delayed) in κ_2:*

$$\begin{aligned} \forall \kappa_1, \kappa_2, e, t.\, \kappa_1 \approx \kappa_2 \implies (\mathsf{ready}(\kappa_1, e, t) \iff \mathsf{ready}(\kappa_2, e, t)) \\ \wedge\, (\mathsf{delayed}(\kappa_1, e, t) \iff \mathsf{delayed}(\kappa_2, e, t)) \end{aligned}$$

Proof. An execute event $e = (\_, i)$ is ready in $\kappa_1 = (\sigma_1, \Sigma_1, \mathsf{mispredicted}_1, \mathsf{doomed}_1)$ at a cycle t if register status table bits $\sigma_1.\mathsf{ready}$ for dependencies of the instruction $\sigma_1.\mathsf{rob}[i]$ are set to true. By Definition 8 of $\kappa_1 \approx \kappa_2$, the register status tables are the same in κ_1 and κ_2, meaning that $\mathsf{ready}(\kappa_1, e, t)$ holds if and only if $\mathsf{ready}(\kappa_2, e, t)$ does.

An execute event $e = (\_, i)$ is delayed in κ_1 at a cycle t if it is an event of a branch or transmit instruction and $\mathsf{noTaintedInputs}(\sigma_1, i)$ does not hold. By Definition 8 of $\kappa_1 \approx \kappa_2$, all parts of the STT state in κ_1 and κ_2 but the (doomed) register values are the same. Hence, by Property 4, $\mathsf{noTaintedInputs}$ also does not hold of the STT state of κ_2, meaning that $\mathsf{delayed}(\kappa_1, e, t)$ holds if and only if $\mathsf{delayed}(\kappa_2, e, t)$ does. □

Proof of Lemma 10. Let us consider any cycle t and two extended states κ_1 and κ_2 such that $\kappa_1 \approx \kappa_2$ holds. By Definition 8, $\kappa_1 \approx \kappa_2$ implies that the two extended states are different only by values of doomed registers. In particular, κ_1 and κ_2 take the following form: $\kappa_1 = (\sigma_1, \Sigma, \mathsf{mispredicted}, \mathsf{doomed})$ and $\kappa_2 = (\sigma_2, \Sigma, \mathsf{mispredicted}, \mathsf{doomed})$, and σ_1 and σ_2 are different only by values of doomed registers (as the rest of the state is the same, we write rob instead of $\sigma_1.\mathsf{rob}$ and $\sigma_2.\mathsf{rob}$ in the proof). We consider each event and show that values of doomed registers do not affect whether the STT processor logic can perform the event.

Firstly, we consider events from $\mathsf{FetchEvents}$, which Algorithm 5 performs unconditionally. Indeed, each of them is enabled independently from values of any physical register. Moreover, the rest of the state (and, in particular, the program counter) coincides in σ_1 and σ_2. Therefore, the lemma holds of such events.

Secondly, we consider events from CommitEvents, which Algorithm 5 performs only when they are enabled. Therefore, we need to show for each $e \in \mathsf{CommitEvents}$ that $\mathsf{enabled}(\kappa_1, e, t)$ holds if and only if $\mathsf{enabled}(\kappa_2, e, t)$ does. However, it is easy to see that each of these events is enabled independently from values of any physical register, and the rest of the state in σ_1 and σ_2 coincides, so the lemma holds of such events.

It remains to consider events from ExecuteEvents, which Algorithm 5 performs only when they are enabled, provided that they were ready and not delayed at the beginning of the cycle. Note that that readiness of input dependencies for each instruction only makes use of ready, which is the same in the two low-equivalent states. Thus, we need to show for every $e \in \mathsf{ExecuteEvents}$:

$$\mathsf{enabled}(\kappa_1, e, t) \wedge \neg\mathsf{delayed}(\kappa_1^{(0)}, e, t) \iff \mathsf{enabled}(\kappa_2, e, t) \wedge \neg\mathsf{delayed}(\kappa_2^{(0)}, e, t)$$

By Proposition 12, $\neg\mathsf{delayed}(\kappa_1^{(0)}, e, t)$ implies $\neg\mathsf{delayed}(\kappa_1, e, t)$, and $\neg\mathsf{delayed}(\kappa_2^{(0)}, e, t)$ implies $\neg\mathsf{delayed}(\kappa_2, e, t)$. Hence, it is sufficient to show:

$$\mathsf{enabled}(\kappa_1, e, t) \wedge \neg\mathsf{delayed}(\kappa_1, e, t) \iff \mathsf{enabled}(\kappa_2, e, t) \wedge \neg\mathsf{delayed}(\kappa_2, e, t)$$

We prove the above in one direction (the other direction can be shown analogously). We assume that $\mathsf{enabled}(\kappa_1, e, t)$ and $\neg\mathsf{delayed}(\kappa_1, e, t)$ hold. By Proposition 13, $\neg\mathsf{delayed}(\kappa_2, e, t)$ holds too. It remains to show $\mathsf{enabled}(\kappa_2, e, t)$. Hence, to conclude the lemma, it suffices to consider each non-delayed event and show that values of doomed registers do not influence whether it is enabled.

The events (Execute-Arithmetic, i), (Execute-Load-BeginGetS, i), (Execute-Load-EndGetS, i) and (Execute-Load-Complete, i) are enabled independently from values of any physical register, and the rest of the state in σ_1 and σ_2 coincides, so the lemma holds of these events.

The events (Execute-Branch-Success, i) and (Execute-Branch-Fail, i) depend on values of registers x_c and x_d. Knowing that $\neg\mathsf{delayed}(\kappa_2, e, t)$ holds, we get that $\mathsf{noTaintedInputs}(\kappa_2, i)$ holds too. By Proposition 3, x_c and x_d are not tainted and, by Definition 7, not doomed. Since $\kappa_1 \approx \kappa_2$ holds, we know that only doomed registers are different in the two extended states. Therefore, values of doomed registers do not influence whether it is enabled, so $\mathsf{enabled}(\kappa_1, e, t)$ implies $\mathsf{enabled}(\kappa_2, e, t)$, which allows us to conclude the lemma.

□

5 Limitations of the processor model

The STT processor model is an abstraction of an out-of-order speculative processor. To make presentation and technical development more accessible, we use simplified models for multiple aspects of the processor. In our models of the in-order and out-of-order processors, we assume a basic instruction set. Also, in our model of processor state, we treat memory addresses as physical addresses. Supporting more instruction types and considering translation from virtual to physical address space are straightforward extensions to the processor model, which do not affect the non-interference. In the rest of this section, we comment on the non-straightforward extensions to the model more elaborately and argue that they do not fundamentally affect the non-interference proof.

The amount of resources available for a processor is limited in practice, which leads to contention over those. Our model of the STT processor avoids the complexity of certain resource contention by making the following three notable simplifications. Firstly, the STT processor manipulates an unbounded number of physical registers PRegID. To make our model more realistic, we could make the set PRegID finite and, for instance, adjust the STT processor logic to stop fetching instructions unless there are physical registers available for register renaming. Secondly, the STT processor operates the reorder buffer of an infinite length. To make our model more realistic, we could bound the length of the reorder buffer, which would require adjusting the STT processor logic to stop fetching instructions while the buffer remains full. Thirdly, the STT processor manipulates an unbounded number of functional units. To make our model more realistic, we could adjust the STT processor logic to not perform Execute events unless the corresponding functional unit is available. For each of these changes, the proof of Theorem 1 could be carried out identically to our current proof: the key step would be to show that the program counter never gets tainted (as we prove for the simplified model in Section 4.2).

Execution of functional units can take multiple cycles, depending on their inputs or, possibly, independently from those. Many arithmetic units have fixed latency, whereas some others, such as the multiplication units or floating-point units, have input-dependent latency. In our model, we treated all arithmetic operation instructions as if the corresponding functional units execute in one cycle. However, we model variable latency of load instructions, and we could analogously support other instruction taking multiple cycles.

Unlike the STT processor implementation, our model does not support memory-dependency speculation. We leave adding store set predictors into the model for future work.

References

[1] J. Mclean. Security models. In *Encyclopedia of Software Engineering*. Wiley & Sons, 1994.

[2] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. W. Fletcher, and J. Torrellas. InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy. In *MICRO'18*, 2018.

[3] J. Yu, L. Hsiung, M. E. Hajj, and C. W. Fletcher. Data oblivious isa extensions for side channel-resistant and high performance computing. In *NDSS'19*, 2019.

[4] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. W. Fletcher. Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data. In *52nd IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019.