The Code That Never Ran
Modeling Attacks on Speculative Evaluation

Craig Disselkoen
University of California San Diego
Mozilla Research Internship
cdisselk@cs.ucsd.edu

Radha Jagadeesan
DePaul University
rjagadeesan@cs.depaul.edu

Alan Jeffrey
Mozilla Research
ajeffrey@mozilla.com

James Riely
DePaul University
jriely@cs.depaul.edu

***Abstract*—This paper studies information flow caused by speculation mechanisms in hardware and software. The Spectre attack shows that there are practical information flow attacks which use an interaction of dynamic security checks, speculative evaluation and cache timing. Previous formal models of program execution have not been designed to model speculative evaluation, and so do not capture attacks such as Spectre. In this paper, we propose a model based on pomsets which is designed to model speculative evaluation. The model provides a compositional semantics for a simple shared-memory concurrent language, which captures features such as data and control dependencies, relaxed memory and transactions. We provide models for existing information flow attacks based on speculative evaluation and transactions. We also model new information flow attacks based on compiler optimizations. The new attacks are experimentally validated against gcc and clang. We develop a simple temporal logic that supports invariant reasoning.**

I. Introduction

This paper studies information flow caused by speculation mechanisms in hardware and software.

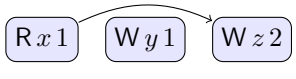
Information flow provides a formal foundation for end-to-end security. Informally, a program is secure if there is no observable dependency of low-security outputs on high-security inputs. The precise formalization of this intuitive idea has been the topic of extensive research Sabelfeld and Myers (2006), encompassing a variety of language features such as nondeterminism Wittbold and Johnson (1990), concurrency Smith and Volpano (1998), reactivity O'Neill et al. (2006), and probability Gray (1992). The static and dynamic enforcement of these definitions in general purpose languages Myers (1999) has influenced language design and implementation.

A key parameter in defining information flow is the *observational power* of the attacker model. Whereas the classical input-output behavior is often an adequate foundation, it has long been known Lampson (1973); Biswas et al. (2017) that side-channels that leak information arise from other observables such as execution time and power consumption. Recently, the Spectre family of attacks Kocher et al. (2019) has shown that side-channels arise from speculative evaluation.

There are several sources of speculative evaluation. Each of these is designed so that failed speculation does not affect the input-output behavior of the program, but may affect other observable behavior, opening an opportunity for a side-channel attack:

- To facilitate pipelining, microprocessors use heuristics to predict the outcome of instructions. Execution proceeds speculatively until these predictions can be validated, at which point instructions are either committed or aborted. The Spectre family of attacks Kocher et al. (2019) exploit the timing differences between successful and unsuccessful predictions. This means, for example, that a single execution of $(\texttt{if}\,(M)\,\{\,C\,\}\,\texttt{else}\,\{\,D\,\})$ may depend on both C and D. This differs from the standard semantics of the conditional, in which executions of C and D are disjoint.
- Some modern microprocessors also support transactional memory Chong et al. (2018), where aborted transactions are meant to be unobservable. Transactions may abort due to cache conflicts, however, and this mechanism can be exploited to improve the efficacy of Spectre-like attacks Disselkoen et al. (2017).
- Relaxed memory models Manson et al. (2005); Boehm and Adve (2008); Zhao et al. (2012) allow for the observation of control and data dependencies. This creates an opportunity for information flows caused by optimizing compilers, whose behavior is driven by dependency analysis. For example, $(\texttt{if}\,(r)\,\{\,x\texttt{:=}1\,\}\,\texttt{else}\,\{\,x\texttt{:=}1\,\})$ can be optimized to $(x\texttt{:=}1)$, whereas $(\texttt{if}\,(r)\,\{\,x\texttt{:=}1\,\}\,\texttt{else}\,\{\,x\texttt{:=}2\,\})$ cannot be so optimized.

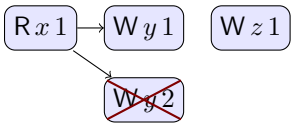
We develop a model to capture such attacks. Our model is based on *partially ordered multisets* Gischer (1988); Plotkin and Pratt (1997) ("pomsets"), whose labels are given by read and write actions. These can be visualized as a graph where the edges indicate dependencies, for example $(r\texttt{:=}x\texttt{;}\ y\texttt{:=}1\texttt{;}\ z\texttt{:=}r+1)$ has an execution modeled by the pomset:



The edge from $(\mathsf{R}\,x\,1)$ to $(\mathsf{W}\,z\,2)$ indicates a data dependency. The novel aspect of the model is that events have *preconditions* which may be false. These are used in giving the semantics of conditionals, for example

Identify applicable funding agency here. If none, delete this.

$(\texttt{if}\,(x)\,\{\,y\texttt{:=}1\texttt{;}\ z\texttt{:=}1\,\}\,\texttt{else}\,\{\,y\texttt{:=}2\texttt{;}\ z\texttt{:=}1\,\})$ has an execution:



The edges from $(\mathsf{R}\,x\,1)$ to $(\mathsf{W}\,y\,1)$ and $(\mathsf{W}\,y\,2)$ indicate control dependencies. The presence of a crossed out $(\mathsf{W}\,y\,2)$ indicates an event with an unsatisfiable precondition.

The novel contributions of this paper are:

- a model of program execution that includes speculation (§II),
- examples showing how the model can be applied, including existing information flow attacks on hardware and transactional memory, and new attacks on optimizing compilers (§III),
- experimental evidence about how practical it is to mount the new class of attacks (§IV), and
- a temporal logic which supports compositional proof (§V).

Readers who wish to focus on the impact of the model can skip past §II on first reading, and refer back to it when needed.

Speculation in Information flow: The information flow literature has largely, with the exceptions noted below, not addressed matters of speculation and the side channels that arise from the observations of efficiency of executions such as execution time and power consumption. For example, the well-known Smith-Volpano type system (which guarantees noninterference) will allow the Prime-Abort attack as formalized in the paper (and Spectre attack as well, if we were to add cache sets to the semantics of a "touch" primitive that we study).

Zhang et al. (2012) models hardware (and thus the microarchitecture) with timing attacks, and explore explicit and static annotations to address timing side channels for hardware description languages. However, this paper does not address speculative execution.

This paper's primary focus is not weak memory. However, hardware relaxed memory (such as Total and Partial Store ordering SPARC (1994)) supports differing views of the memory between threads, that can be viewed as a form of thread-specific speculation. So, the research into the impact of hardware relaxed memory on information flow in programs Mantel et al. (2014); Vaughan and Millstein (2012) is relevant. These papers show that the information flow exhibited by a program depends crucially on the particular model of relaxed memory; for example, they demonstrate progams that have no information flow when executed in the in usual *sequentially consistent* memory, and yet exhibit information flow when executed in the TSO model.

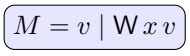
In contrast to the hardware relaxed memory models addressed in these papers, software relaxed memory models (such as the JMM Manson et al. (2005) and C11 Boehm and Adve (2008)) also incorporate speculation on conditionals.

Our model captures enough detail enough to reveal and analyze the presence of side channels revealed by speculative executions that are implicit in the literature on hardware and software relaxed memory models. Thus, we are able to demonstrate example attacks that show that ordinary compiler optimizations can violate some intuitively expected informal information flow guarantees. Furthermore, as far as we know, the observation that compiler optimizations can result in information flows that can be observed without timers, is new to this paper.

Self-composition Barthe et al. (2004) reduces the problem of secure information flow of a program to a *safety* property of a program derived by composing the program with a renaming of itself. Our work shows that the traditional conditional serves as a self-composition operator in the presence of speculation.

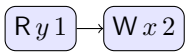
II. Model

The model used in this paper is one of sets of pomsets with event labels of the form $(\phi \mid a)$, where ϕ is the event's precondition (such as $M = v$) and a is the event's action (such as $\mathsf{W}\,x\,v$). For example the semantics of the program $(x := M)$ includes the case where M is v, which is written to x, and is captured by the one-event pomset:

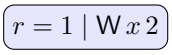


We make few requirements of the logic of preconditions, save that it has includes equalities between expressions, is closed under substitution, and supports a notion of implication.

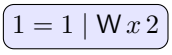
For example, the set of pomsets $[\![r := y;\ x := r{+}1]\!]$ contains:



The semantics is defined compositionally. First, $[\![x := r + 1]\!]$ contains the pomset:

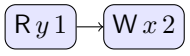


Next, we perform the substitution of r with 1 in every precondition, to get that $[\![x := r{+}1]\!][1/r]$ contains the pomset:

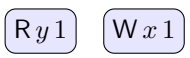


and since $(1 = 1)$ is a tautology, we elide it:

This substitution is performed in defining $[\![r := y;\ x := r + 1]\!]$, which contains the pomset:



as required. There is an ordering $(\mathsf{R}\,y\,1) < (\mathsf{W}\,x\,2)$ because the precondition $(r = 1)$ depends on r. If the precondition was independent of r then there would be no ordering, for example $[\![r := y;\ x := r + 1 - r]\!]$ contains the pomset:



since the precondition $(r + 1 - r = 1)$ is independent of r.

The main novelty of our semantics, since it is designed to model speculative evaluation, is in modeling conditionals. In most sets-of-pomsets semantics, a pomset in $[\![\texttt{if}\,(M)\,\{\,C\,\}\,\texttt{else}\,\{\,D\,\}]\!]$ would either be given by a pomset in $[\![C]\!]$ or a pomset in $[\![D]\!]$. To model speculative evaluation, we need to allow a pomset in $[\![\texttt{if}\,(M)\,\{\,C\,\}\,\texttt{else}\,\{\,D\,\}]\!]$ to be given by both a pomset in $[\![C]\!]$ *and* a pomset in $[\![D]\!]$. For example, $[\![\texttt{if}\,(M)\,\{\,x\texttt{:=}1\,\}\,\texttt{else}\,\{\,x\texttt{:=}2\,\}]\!]$ contains the pomset:

$$\boxed{M \neq 0 \mid \mathsf{W}\,x\,1} \quad \boxed{M = 0 \mid \mathsf{W}\,x\,2}$$

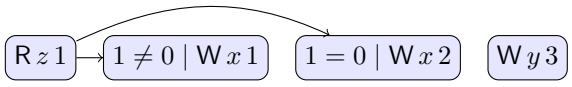
that is we have recorded behavior from both branches of execution. Moreover, an action which is performed on both sides of the conditional can be merged, producing only one event in the resulting pomset. The precondition of the merged event is the disjunction of the preconditions of the original events. For example $[\![\texttt{if}\,(M)\,\{\,x\texttt{:=}1\texttt{;}\ y\texttt{:=}3\,\}\,\texttt{else}\,\{\,x\texttt{:=}2\texttt{;}\ y\texttt{:=}3\,\}]\!]$ contains the pomset:

$$\boxed{M \neq 0 \mid \mathsf{W}\,x\,1} \quad \boxed{M = 0 \mid \mathsf{W}\,x\,2} \quad \boxed{(M \neq 0) \vee (M = 0) \mid \mathsf{W}\,y\,3}$$

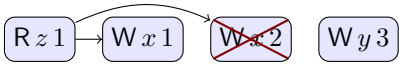
and since $(M \neq 0) \vee (M = 0)$ is a tautology, this is:

$$\boxed{M \neq 0 \mid \mathsf{W}\,x\,1} \quad \boxed{M = 0 \mid \mathsf{W}\,x\,2} \quad \boxed{\mathsf{W}\,y\,3}$$

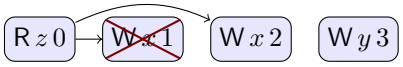
Combining this model of conditionals with the model of memory using substitutions gives that $[\![\texttt{if}\,(z)\,\{\,x\texttt{:=}1\texttt{;}\ y\texttt{:=}3\,\}\,\texttt{else}\,\{\,x\texttt{:=}2\texttt{;}\ y\texttt{:=}3\,\}]\!]$ contains the pomset:



and since $(1 \neq 0)$ is a tautology and $(1 = 0)$ is unsatisfiable, this is:



Similarly, $[\![\texttt{if}\,(z)\,\{\,x\texttt{:=}1\texttt{;}\ y\texttt{:=}3\,\}\,\texttt{else}\,\{\,x\texttt{:=}2\texttt{;}\ y\texttt{:=}3\,\}]\!]$ contains the pomset:



Note that this semantics captures control dependencies such as $(\mathsf{R}\,z\,0) < (\mathsf{W}\,x\,1)$, independencies such as $(\mathsf{R}\,z\,0) \not< (\mathsf{W}\,y\,3)$, and failed speculations such as the crossed out $(\mathsf{W}\,x\,1)$.

In summary, the features we need of the underlying data model are:

- *actions*, which may read or write to memory locations, and
- *preconditions*, which form a logic closed under substitution,

from which we can define the operations used in defining the semantics of programs, which include:

- *prefixing* $a \to \mathcal{P}$, which adds an event with precondition ϕ and action a to pomsets in $\mathcal{P}$,
- *substitution* $\mathcal{P}[M/x]$, which performs a substitution on every precondition in $\mathcal{P}$, and
- *concurrency* $\mathcal{P}_1 \parallel \mathcal{P}_2$, which unions pomsets from $\mathcal{P}_1$ and $\mathcal{P}_2$, allowing events to be merged.

We make data models precise in §II-A, define pomsets in §II-B, and operations on sets of pomsets in §II-C, which are used to give a compositional semantics for a simple imperative language.

A. Data models

A *data model* consists of:

- a set of *memory locations* $\mathcal{X}$, ranged over by x and y,
- a set of *registers* $\mathcal{R}$, ranged over by r and s,
- a set of *values* $\mathcal{V}$, ranged over by v and w,
- a set of *expressions* $\mathcal{E}$, ranged over by M and N,
- a set of *logical formulae* Φ, ranged over by ϕ and ψ, and
- a set of *actions* $\mathcal{A}$, ranged over by a and b,

such that:

- values include at least the constants 0 and 1,
- expressions include at least registers and values,
- expressions are closed under substitutions of the form $M[N/r]$,
- formulae include at least true, false, and equalities of the form $(M = N)$ and $(x = N)$,
- formulae are closed under negation, conjunction, disjunction,
- formulae are closed under substitutions of the form $\phi[x/r]$ or $\phi[N/x]$,
- there is a relation $\vDash$ between formulae, and
- there are partial functions R and $\mathsf{W} : \mathcal{A} \to (\mathcal{X} \times \mathcal{V})$.

We shall say a *reads* v *from* x whenever $\mathsf{R}(a) = (x, v)$, and a *writes* v *to* x whenever $\mathsf{W}(a) = (x, v)$. We shall say ϕ *implies* ψ whenever $\phi \vDash \psi$, ϕ is a *tautology* whenever $\mathsf{true} \vDash \phi$, ϕ is *unsatisfiable* whenever $\phi \vDash \mathsf{false}$, and ϕ is *independent of* x whenever $\phi \vDash \phi[v/x] \vDash \phi$ for every v. In examples, the actions are of the form $(\mathsf{R}\,x\,v)$, which reads v from x, and $(\mathsf{W}\,x\,v)$, which writes v to x. Going forward, we assume a given data model, though some examples in §III make use of particular actions.

B. 3-valued pomsets

Recall the definition of a pomset from Gischer (1988):

Definition II.1. A *pomset* $(E, \leq, \lambda)$ with alphabet Σ is a partial order $(E, \leq)$ together with $\lambda : E \to \Sigma$.

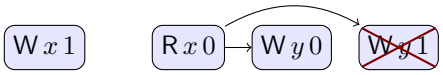
Going forward, we fix the alphabet $\Sigma = (\Phi \times \mathcal{A})$. We will write $(\phi \mid a)$ for the pair (ϕ, a), elide ϕ when ϕ is a tautology, and write a crossed-out ($\not{a}$) when ϕ is unsatisfiable. We lift terminology from logical formulae and actions to events, for example if $\lambda(e) = (\phi \mid a)$ then we say e is unsatisfiable whenever ϕ is unsatisfiable, e writes v to x whenever a writes v to x, and so forth. We visualize a pomset as a graph where the nodes are drawn from E, each node e is labelled with

$\lambda(e)$, and an edge $d \to e$ corresponds to an ordering $d \leq e$. For example:

is a visualization of the pomset where:

$$0 \leq 1 \quad 0 \leq 2 \quad \lambda(0) = (\mathsf{true}, \mathsf{R}\,x\,1) \quad \lambda(1) = (\mathsf{false}, \mathsf{W}\,y\,0) \quad \lambda(2) = (\mathsf{true}, \mathsf{W}\,y\,1)$$

We are building a compositional semantics of shared memory concurrency, which means we require a notion of when a read has a matching write. This is a property we require of closed programs, but *not* of open programs. For example a program whose semantics includes:



may be put in parallel with another program which writes 0 to x. If the program is closed with respect to x though, such an execution cannot exist, so we need each read of x to have a matching write. This is captured by defining when e *reads* x *from* d Alglave et al. (2014). A preliminary definition (which, as we shall see, needs to be strengthened) is:

- $d < e$,
- if e is satisfiable, then d is a tautology,
- d writes v to x, and e reads v from x, and
- there is no $d < c < e$ such that c writes to x.

Unfortunately by itself, this is not enough. The problem is the final clause saying that there does not exist an x-*blocking* event c between d and e. Unfortunately, concurrency can turn events that were not x-blockers into an x-blocker, *even if the new thread does not mention* x. We give an example to show this in §III-E.

There are a number of ways this can be addressed; for example, in models such as Batty et al. (2011) the reads-from relation is taken as a primitive. In this paper, we propose *3-valued pomsets* as a solution. These are pomsets in which, in addition to positive statements $(d < e)$ (interpreted as e depends on d), we also have negative statements $(d \not< e)$ (interpreted as e cannot depend on d).

Definition II.2. A *3-valued poset* $(E, \leq, \not<)$ is a poset $(E, \leq)$ together with $\not< \subseteq (E \times E)$ such that:

- if $d \leq e$ then $e \not< d$,
- if $d \leq e$ and $d \not< e$ then $d = e$,
- if $c \geq d \not< e$ or $c \not< d \geq e$ then $c \not< e$.

Definition II.3. A *3-valued pomset* $(E, \leq, \not<, \lambda)$ is a 3-valued poset $(E, \leq, \not<)$ and a pomset $(E, \leq, \lambda)$.

In diagrams, we visualize $(e \not< d)$ as a dashed arrow from d to e (note the change of direction). We refer to edges introduced by $(d < e)$ as *strong edges* and by $(e \not< d)$ as *weak edges*. For readability, we often highlight the reads-from edges as well. For example:

Structures similar to 3-valued pomsets have come up in many guises, for example rough sets Pawlak (1982) or ultrametrics over $\{0, 1/2, 1\}$. They correspond to axioms A1–A3 of Lamport's *system executions* Lamport (1986). They are the notion of pomset given by interpreting $d \leq e$ in a 3-valued logic Urquhart (1986).

We strengthen the definition of reads-from to require not just that a blocker exists, but that any candidate blocker must either have $d \not< c$ or $c \not< e$. This ensures that any further concurrency cannot turn a non-blocker into a blocker.

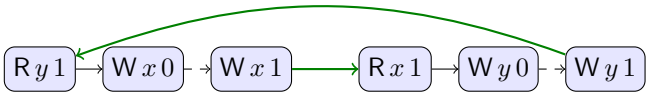
Definition II.4. In a 3-valued pomset, e *can read* x *from* d whenever:

- $d < e$,
- if e is satisfiable, then d is a tautology,
- d writes v to x, and e reads v from x, and
- if c writes to x then either $d \not< c$ or $c \not< e$.

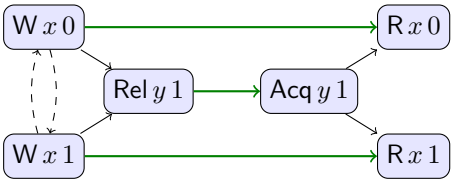
Definition II.5. A 3-valued pomset is x-closed if, for every $e \in E$:

- e is independent of x, and
- if e reads from x, then there is a d such that e can read x from d.

The definitions as they stand allow cycles in weak edges. This is necessary for examples such as $(x := y - 1;\ x := 1 \ ||\ y := x - 1;\ y := 1)$ which has execution:



However, in order to model release/acquire fencing in §III-L, we need to ban executions such as:



The problem here is the weak cycle between $(\mathsf{W}\,x\,0)$ and $(\mathsf{W}\,x\,1)$, which according to Definition II.4, allows both $(\mathsf{R}\,x\,0)$ and $(\mathsf{R}\,x\,1)$, even though one of them must be a stale value. This can be addressed by requiring $\not<$ to form a *per-location* partial order. This is a form of partial coherence, and can be strengthened to total coherence by requiring $\not<$ to be a per-location total order.

Definition II.6. A 3-valued pomset is *partially* (resp. *totally*) x-*coherent* if, when restricted to events which write to x, $\not<$ forms a partial (resp. total) order.

C. Sets of 3-valued pomsets

Our model of programs is going to be sets of 3-valued pomsets. In this section we define the operations on pomsets which are used in giving the semantics. These are operations such as prefixing, parallel composition, restriction, and so on;

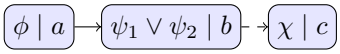
they are familiar from models of concurrency such as Brookes et al. (1984), but adapted here to the setting of speculative evaluation.

Definition II.7. Let $P_0 \in (\mathcal{P}_1 \parallel \mathcal{P}_2)$ whenever there are $P_1 \in \mathcal{P}_1$ and $P_2 \in \mathcal{P}_2$ such that:

- $E_0 = E_1 \cup E_2$,
- if $e \leq_1 d$ or $e \leq_2 d$ then $e \leq_0 d$,
- if $e \not\leq_1 d$ or $e \not\leq_2 d$ then $e \not\leq_0 d$,
- if $\lambda_0(e) = (\phi_0 \mid a)$ then either:
  - $\lambda_1(e) = (\phi_1 \mid a)$ and $\lambda_2(e) = (\phi_2 \mid a)$ and ϕ_0 implies $\phi_1 \vee \phi_2$,
  - $\lambda_1(e) = (\phi_1 \mid a)$ and $e \notin E_2$ and ϕ_0 implies ϕ_1, or
  - $\lambda_2(e) = (\phi_2 \mid a)$ and $e \notin E_1$ and ϕ_0 implies ϕ_2.

We use $\mathcal{P}_1 \parallel \mathcal{P}_2$ in defining the semantics of conditionals and concurrency. It contains the union of pomsets from $\mathcal{P}_1$ and $\mathcal{P}_2$, allowing overlap as long as they agree on actions. For example, if $\mathcal{P}_1$ and $\mathcal{P}_2$ contain:

then $\mathcal{P}_1 \parallel \mathcal{P}_2$ contains:

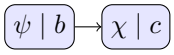


Definition II.8. Let $a \to \mathcal{P}$ be the set $\mathcal{P}'$ where $P' \in \mathcal{P}'$ whenever there is $P \in \mathcal{P}$ such that:

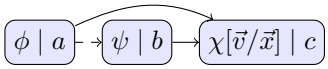
- $E' = E \cup \{c\}$,
- if $d \leq e$ then $d \leq' e$,
- if $d \not\leq e$ then $d \not\leq' e$,
- $\lambda'(c) = (\phi, a)$, and
- if $\lambda(e) = (\psi \mid b)$ then:
  - $\lambda'(e) = (\psi' \mid b)$,
  - ψ' implies $\begin{cases} \psi[v/x] & \text{if } a \text{ reads } v \text{ from } x \text{ and } c \leq e \quad \text{[DEPENDENT READ]} \\ \psi[v/x] \text{ and } \psi & \text{if } a \text{ reads } v \text{ from } x \quad \text{[INDEPENDENT READ]} \\ \psi & \text{otherwise} \quad \text{[NON-READ]} \end{cases}$
  - if a and b both write to y, then $c \not\leq' e$.

Prefixing is used to define the semantics of reads and writes, and adds a new event c with action a. As in the definition of parallel composition, the definition allows the new event to overlap with events in $\mathcal{P}$ as long as they agree on the action.

The tricky parts of the definition are the named cases, which place requirements on read dependencies. If a reads v from x, we have to decide whether e depends on c for some e with old precondition ψ and new precondition ψ'. The first case [DEPENDENT READ] is that the dependency exists, in which case ψ' just has to imply $\psi[v/x]$. The more interesting case is [INDEPENDENT READ], in which case ψ' has to imply $\psi[v/x]$ and ψ. This corresponds to a case where e can be performed with or without c. In particular, if ψ is independent of x then we can pick ψ' to be ψ, and the independent read case will apply. For example, if a and b write to the same location, a reads v from x, ψ is independent of x, and $\mathcal{P}$ contains:



then $a \to \mathcal{P}$ contains:



Definition II.9. Let $\mathcal{P}[M/x]$ be the set $\mathcal{P}'$ where $P' \in \mathcal{P}'$ whenever there is $P \in \mathcal{P}$ such that:

- $E' = E$,
- if $d \leq e$ then $d \leq' e$, and
- if $d \not\leq e$ then $d \not\leq' e$, and
- if $\lambda(e) = (\psi \mid a)$ then $\lambda'(e) = (\psi[M/x] \mid a)$.

and similarly for $\mathcal{P}[x/r]$.

Definition II.10. Let $(\phi \rhd \mathcal{P})$ be the subset of $\mathcal{P}$ such that $P \in \mathcal{P}$ whenever:

- if $\lambda(e) = (\psi \mid a)$ then ϕ implies ψ.

Definition II.11. Let $(\nu x \,.\, \mathcal{P})$ be the subset of $\mathcal{P}$ such that $P \in \mathcal{P}$ whenever P is x-closed and partially x-coherent.

We can use the operations defined above to give the semantics of a simple concurrent imperative programming language.

Definition II.12.

$$
\begin{aligned}
[\![\texttt{skip}]\!] &= \{\emptyset\} \\
[\![x := M\texttt{;}\ C]\!] &= \textstyle\bigcup_v \left((M = v) \rhd (\mathsf{W}\, x\, v) \to [\![C]\!][M/x]\right) \\
[\![r := x\texttt{;}\ C]\!] &= [\![C]\!][x/r] \cup \textstyle\bigcup_v (\mathsf{R}\, x\, v) \to [\![C]\!][x/r] \\
[\![\texttt{if}\,(M)\,\{\,C\,\}\,\texttt{else}\,\{\,D\,\}]\!] &= \left((M \neq 0) \rhd [\![C]\!]\right) \parallel \left((M = 0) \rhd [\![D]\!]\right) \\
[\![C \mid\mid D]\!] &= [\![C]\!] \parallel [\![D]\!] \\
[\![\texttt{var}\, x\texttt{;}\ C]\!] &= \nu x \,.\, [\![C]\!]
\end{aligned}
$$

A write generates a write event that may be visible to other threads. A read may see a thread-local value, or it may generate a read event that must be justified by another thread. In the latter case, occurrences of r are replaced with x (rather than v) to ensure that dependencies are tracked properly. The subsequent substitution of v for x occurs in Definition II.8 of prefixing.

We have completed the formal definition of our model of speculative evaluation, and now turn to examples of this model in use.

III. Examples

In this section, we shall start off by giving some basic examples, and then show how three different information flow attacks can be modeled. We cover Spectre in §III-H, new attacks on compiler optimizations in §III-J–III-K, and attacks on transactions in §III-M.

A. Sequential memory accesses

In the semantics of memory, there are two very different ways memory can be accessed: sequentially or concurrently. These are modeled differently, since hardware and compilers give very different guarantees about their behavior. In this section, we discuss the sequential semantics, and leave the concurrent semantics to §III-B.

Consider the program $(x := 0;\ y := x + 1)$. One execution of this program is where the write to y uses the sequential value of x, which is 0:

$$\boxed{\mathsf{W}\,x\,0} \quad \boxed{\mathsf{W}\,y\,1}$$

To see how this execution is modeled, we first expand out the syntax sugar to get the program $(x := 0;\ r := x;\ y := r + 1;\ \texttt{skip})$ Now $[\![\texttt{skip}]\!]$ is just $\{\emptyset\}$, and $[\![y := r + 1;\ \texttt{skip}]\!]$ includes:

$$(r + 1 = 1) \rhd (\mathsf{W}\,y\,1) \to [\![\texttt{skip}]\!][1/y]$$

which contains the pomset:

$$\boxed{r + 1 = 1 \mid \mathsf{W}\,y\,1}$$

expressing that this program can write 1 to y, as long as the precondition $(r + 1 = 1)$ is satisfied. Now $[\![r := x;\ y := r + 1;\ \texttt{skip}]\!]$ has two cases, the sequential case (which does not introduce a read action) and the concurrent case (which does). For the moment, we are interested in the sequential case, which is:

$$[\![y := r + 1;\ \texttt{skip}]\!][x/r]$$

which contains the pomset:

$$\boxed{x + 1 = 1 \mid \mathsf{W}\,y\,1}$$

In this pomset, the precondition is $(x + 1 = 1)$, which specifies a property of the thread-local value of x. Finally $[\![x := 0;\ r := x;\ y := r + 1;\ \texttt{skip}]\!]$ includes:

$$(0 = 0) \rhd (\mathsf{W}\,x\,0) \to [\![r := x;\ y := r + 1;\ \texttt{skip}]\!][0/x]$$

which contains the pomset:

$$\boxed{0 = 0 \mid \mathsf{W}\,x\,0} \quad \boxed{0 = 0 \wedge 0 + 1 = 1 \mid \mathsf{W}\,y\,1}$$

all of whose preconditions are tautologies, so this has the expected behavior:

$$\boxed{\mathsf{W}\,x\,0} \quad \boxed{\mathsf{W}\,y\,1}$$

There is no dependency between $(\mathsf{W}\,x\,0)$ and $(\mathsf{W}\,y\,1)$, since $(0 = 0 \wedge 0 + 1 = 1)$ is independent of x.

This example demonstrates how preconditions capture the sequential semantics of memory. In an execution containing an event with label $(\phi \mid a)$, one way the precondition ϕ can be discharged is by an assignment $x := M$, which performs a substitution $[M/x]$. This is a variant of the Hoare semantics of assignment Hoare (1969), where if C has precondition ϕ then $x := M;\ C$ has precondition $\phi[M/x]$.

B. Concurrent memory accesses

We now turn to the case of concurrent accesses to memory. Consider the program $(x := 1 \ ||\ y := x + 1)$. In executions of this program, it is possible for the second thread to perform a concurrent read of x:

$$\boxed{\mathsf{W}\,x\,1} \longrightarrow \boxed{\mathsf{R}\,x\,1} \to \boxed{\mathsf{W}\,y\,2}$$

To see how this execution is modeled, we first expand out the syntax sugar to get the program $(x := 1;\ \texttt{skip} \ ||\ r := x;\ y := r + 1;\ \texttt{skip})$. As before, $[\![y := r + 1;\ \texttt{skip}]\!]$ includes:

$$(r + 1 = 2) \rhd (\mathsf{W}\,y\,2) \to [\![\texttt{skip}]\!][2/y]$$

which contains the pomset:

$$\boxed{r + 1 = 2 \mid \mathsf{W}\,y\,2}$$

As before, $[\![r := x;\ y := r + 1;\ \texttt{skip}]\!]$ has two cases. We are now interested in the concurrent case, which includes:

$$(\mathsf{R}\,x\,1) \to [\![y := r + 1;\ \texttt{skip}]\!][x/r]$$

which contains the pomset:

$$\boxed{\mathsf{R}\,x\,1} \to \boxed{\mathsf{W}\,y\,2}$$

Note that $(\mathsf{R}\,x\,1)$ reads 1 from x, and while $(x + 1 = 2)[1/x]$ is a tautology, $(x + 1 = 2)$ is not, and so there is a dependency $(\mathsf{R}\,x\,1) < (\mathsf{W}\,y\,2)$.

Now, $[\![x := 1;\ \texttt{skip}]\!]$ includes the pomset:

$$\boxed{\mathsf{W}\,x\,1}$$

and so $[\![x := 1;\ \texttt{skip} \ ||\ r := x;\ y := r + 1;\ \texttt{skip}]\!]$ includes:

$$\boxed{\mathsf{W}\,x\,1} \longrightarrow \boxed{\mathsf{R}\,x\,1} \to \boxed{\mathsf{W}\,y\,2}$$

as expected, including a reads-from dependency $(\mathsf{W}\,x\,1) < (\mathsf{R}\,x\,1)$.

This example demonstrates how read and write events capture the concurrent semantics of memory. In an execution containing an event with label $(\mathsf{R}\,x\,v)$, if the execution is x-closed, then there must be an event it reads from, for example one labelled $(\mathsf{W}\,x\,v)$.

C. Independent writes

Consider an example with two independent writes $(x := 1;\ y := 2)$. This has semantics including:

$$(\mathsf{W}\,x\,1) \to (\mathsf{W}\,y\,2) \to \{\emptyset\}$$

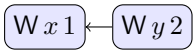
One of the executions this contains is:

$$\boxed{\mathsf{W}\,x\,1} \to \boxed{\mathsf{W}\,y\,2}$$

but it also contains:

$$\boxed{\mathsf{W}\,x\,1} \quad \boxed{\mathsf{W}\,y\,2}$$

and:

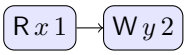


since there is no requirement that $(\mathsf{W}\,x\,1) < (\mathsf{W}\,y\,2)$.

Thus, the semantics of $(x := 1;\ y := 2)$ is the same as the semantics of $(y := 2;\ x := 1)$.

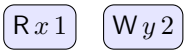
D. Independent reads and writes

Whereas write prefixing introduces weak dependencies on events which write to the same location, read prefixing introduces strong dependencies on preconditions which depend on the location being read. For example in §III-B we saw that the program $(y := x + 1)$ includes the pomset:



but since $(x + 1 = 2)$ depends on x, we have the requirement that $(\mathsf{R}\,x\,1) \leq (\mathsf{W}\,y\,2)$.

This is in contrast to the program $(r := x;\ y := r + 2 - r)$. Since $(x + 2 - x = 2)$ is independent of x (at least for integer arithmetic) this contains:



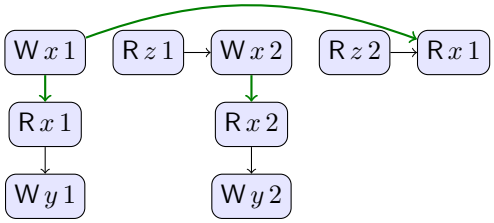
and so the semantics of $(r := x;\ y := r + 2 - r)$ is the same as the semantics of $(y := 2;\ r := x)$.

This example shows that our notion of dependency is based on logical implication, and is therefore semantic rather than syntactic. In syntactic dependency, which is common in hardware models of memory, since r occurs free in $(y := r + 2 - r)$, there would be a dependency between $(r := x)$ and $(y := r + 2 - r)$.

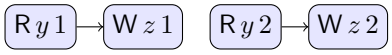
E. Blockers

Recall the preliminary definition of reads-from in §II-B, which defined an x-blocker to be and event c that writes to x such that $d < c < e$. Were we to adopt this definition, then concurrent threads could turn events that were not x-blockers into an x-blocker, even if the new thread does not mention x.

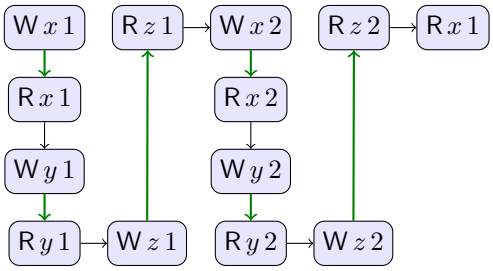
To see this, consider the program $(x := 1;\ y := x \ ||\ x := z + 1;\ y := x \ ||\ \texttt{if}\,(z = 2)\,\{\,r := x\,\})$ with execution:



and the program $(z := y;\ z := y)$ with execution:

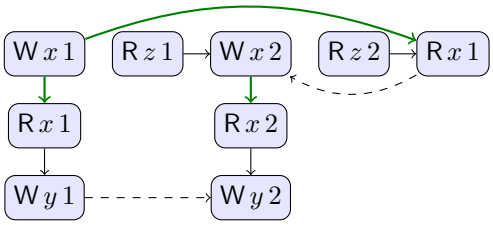


If these are placed in parallel, then a possible execution is:

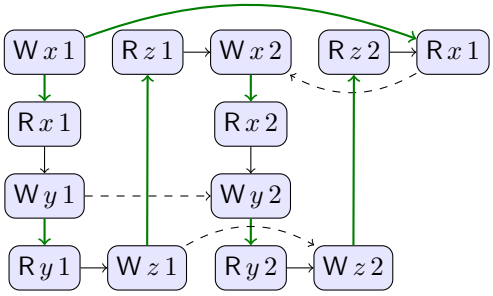


and now the $(\mathsf{W}\,x\,2)$ event is an x-blocker, so $(\mathsf{R}\,x\,1)$ cannot read from $(\mathsf{W}\,x\,1)$.

In the final definition of reads-from in §II-B we ruled out x-blockers by requring that any event c that writes to x has either $d \not< c$ or $c \not< e$. With this definition, in order for $(\mathsf{R}\,x\,1)$ to read from $(\mathsf{W}\,x\,1)$, we either need $(\mathsf{W}\,x\,1) \not< (\mathsf{W}\,x\,2)$ or $(\mathsf{W}\,x\,2) \not< (\mathsf{R}\,x\,1)$, for example:



then putting this in parallel as before results in:



but this is *not* a valid 3-valued pomset, since $(\mathsf{W}\,x\,2) < (\mathsf{R}\,x\,1)$ but also $(\mathsf{W}\,x\,2) \not< (\mathsf{R}\,x\,1)$, which is a contradiction.

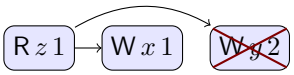
F. Control dependencies

Conditionals introduce control dependencies, for example consider the program:

$$r := z;\ \texttt{if}\,(r)\,\{\,x := 1\,\}\,\texttt{else}\,\{\,y := 2\,\}$$

This includes executions in which the false branch is taken:

and ones where the true branch is taken:



In both cases, we record the actions in the branch that was not taken. This is a novel feature of this model, and is intended

to capture speculative evaluation. In §III-H we will show how this model captures Spectre-like information flow attacks, once the attacker is provided with the ability to observe such speculations.

To see how these executions are modeled, consider the semantics of $[\![x := 1;\ \texttt{skip}]\!]$, which contains any pomset of the form:

$$\boxed{\phi \mid \mathsf{W}\, x\, 1}$$

in particular it contains:

$$\boxed{r \neq 0 \mid \mathsf{W}\, x\, 1}$$

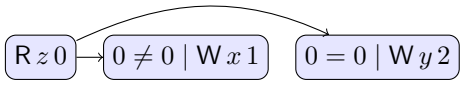
Similarly $[\![y := 2;\ \texttt{skip}]\!]$ contains:

$$\boxed{r = 0 \mid \mathsf{W}\, y\, 2}$$

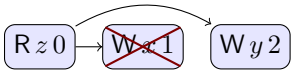
and so $[\![\texttt{if}\,(r)\,\{\,x := 1;\ \texttt{skip}\,\}\,\texttt{else}\,\{\,y := 2;\ \texttt{skip}\,\}]\!]$ contains:

$$\boxed{r \neq 0 \mid \mathsf{W}\, x\, 1} \quad \boxed{r = 0 \mid \mathsf{W}\, y\, 2}$$

Now, the semantics of concurrent read performs substitutions, for example:



which gives the required pomset:



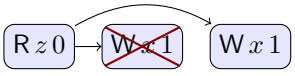
Note that the precondition $r = 0$ is dependent on r, and so there is a dependency $(\mathsf{R}\, z\, 0) < (\mathsf{W}\, y\, 2)$, modeling the control dependency introduced by the conditional.

G. Control independencies

In most models of control dependencies, the dependency relation is syntactic, based on whether the action occurs inside syntactically inside a conditional. In contrast, the notion in this model is semantic: if an action can occur on both sides of a conditional, there is no control dependency. Consider a variant of the example from §III-F:

$$r := z;\ \texttt{if}\,(r)\,\{\,x := 1\,\}\,\texttt{else}\,\{\,x := 1\,\}$$

This has the expected execution in which the control dependencies exist:



but it also has an execution in which the two writes of 1 to x are merged, resulting in no dependency:

$$\boxed{\mathsf{R}\, z\, 0} \quad \boxed{\mathsf{W}\, x\, 1}$$

To see how this arises, consider the definition of $[\![\texttt{if}\,(r)\,\{\,x := 1;\ \texttt{skip}\,\}\,\texttt{else}\,\{\,x := 1;\ \texttt{skip}\,\}]\!]$:

$\mathcal{P}_1 \parallel \mathcal{P}_2$ where $\mathcal{P}_1 = (r \neq 0) \triangleright [\![x := 1;\ \texttt{skip}]\!]$ and $\mathcal{P}_2 = (r = 0) \triangleright$

Now, one pomset in $\mathcal{P}_1$ is:

$$\boxed{r \neq 0 \mid \mathsf{W}\, x\, 1}$$

that is P_1 where:

$$E_1 = \{e\} \quad \lambda_1(e) = (r \neq 0, \mathsf{W}\, x\, 1)$$

and similarly, one pomset in $\mathcal{P}_2$ is:

$$\boxed{r = 0 \mid \mathsf{W}\, x\, 1}$$

that is P_2 where:

$$E_2 = \{e\} \quad \lambda_2(e) = (r = 0, \mathsf{W}\, x\, 1)$$

Crucially, in the definition of $\mathcal{P}_1 \parallel \mathcal{P}_2$ there is *no* requirement that E_1 and E_2 are disjoint, and in this case they overlap at e. As a result, one pomset in $\mathcal{P}_1 \parallel \mathcal{P}_2$ is P_0 where:

$$E_0 = \{e\} \quad \lambda_0(e) = (r \neq 0 \vee r = 0, \mathsf{W}\, x\, 1)$$

that is:

$$\boxed{\mathsf{W}\, x\, 1}$$

Note that this pomset has no precondition dependent on r, since $(r \neq 0 \vee r = 0)$ does not depend on r, which is why we end up with an execution without a control dependency:

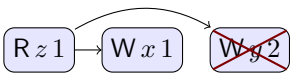
$$\boxed{\mathsf{R}\, z\, 0} \quad \boxed{\mathsf{W}\, x\, 1}$$

This semantics captures compiler optimizations which may, for example, merge code executed on both branches of a conditional, or hoist constant assignments out of loops.

We can now see the counterintuitive behavior of conditionals in the presence of control dependencies. There are programs such as $(\texttt{if}\,(z)\,\{\,x := 1\,\}\,\texttt{else}\,\{\,x := 1\,\})$ with executions in which $(\mathsf{W}\, x\, 1)$ is independent of $(\mathsf{R}\, z\, 1)$:

$$\boxed{\mathsf{R}\, z\, 1} \quad \boxed{\mathsf{W}\, x\, 1}$$

while programs such as $(\texttt{if}\,(z)\,\{\,x := 1\,\}\,\texttt{else}\,\{\,y := 2\,\})$ only have executions in which $(\mathsf{W}\, x\, 1)$ is dependent on $(\mathsf{R}\, z\, 1)$:



These programs have executions with different dependency relations, depending only on conditional branches that were *not* taken. In §III-J we shall see that this has security implications, since relaxed memory can observe dependency. The attack is similar to Spectre, so we shall take a detour to see how Spectre can be modeled in this setting.

H. Spectre

We give a simplified model of Spectre attacks, ignoring the details of timing. In this model, we extend programs with the ability to tell whether a memory location has been touched (in practice this is implemented using timing attacks on the cache). For example, we can model Spectre by:

$$\begin{array}{l}\texttt{var}\,a\texttt{;}\ \texttt{if}\,(\texttt{canRead}(\texttt{SECRET}))\,\{\,a[\texttt{SECRET}]\texttt{:=}1\,\}\\ \texttt{else}\,\texttt{if}\,(\texttt{touched}\,a[0])\,\{\,x\texttt{:=}0\,\}\\ \texttt{else}\,\texttt{if}\,(\texttt{touched}\,a[1])\,\{\,x\texttt{:=}1\,\}\end{array}$$

This is a low-security program, which is attempting to discover the value of a high-security variable SECRET. The low-security program is allowed to attempt to escalate its privileges by checking that it is allowed to read a high-security variable:

$$\texttt{if}\,(\texttt{canRead}(\texttt{SECRET}))\,\{\cdots\text{code allowed to read SECRET}\cdots\}\,\texttt{else}\,\{\cdots\}$$

In this case, canRead(SECRET) is false, so the fallback code is executed. Unfortunately, the escalated code is speculatively evaluated, which allows information to leak by testing for which memory locations have been touched.

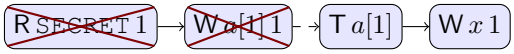
We model the touched test by introducing a new action $(\mathsf{T}\,x)$ and defining:

$$[\![\texttt{if}\,(\texttt{touched}\,x)\,\{\,C\,\}\,\texttt{else}\,\{\,D\,\}]\!] \quad = \quad ((\mathsf{T}\,x) \to [\![C]\!]) \cup [\![D]\!]$$

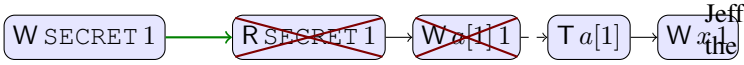
The additional requirement we need to add for x-closure is:

- if $\lambda(e) = (\phi \mid \mathsf{T}\,x)$ then there is $d \not> e$ where d reads or writes x.

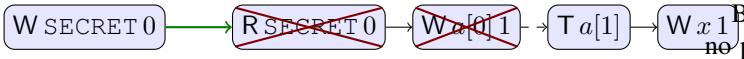
Note that there is no requirement that d be satisfiable, and indeed Spectre has the execution:



Putting this in parallel with a high-security write to SECRET gives:



but due the requirement of a-closure we do *not* have:



Thus, the attacker has managed to leak the value of a high-security location to a low-security one: if $(\mathsf{W}\,x\,1)$ is observed, the SECRET must have been 1.

This shows how our model of speculation can express (very abstract, untimed) Spectre attacks.

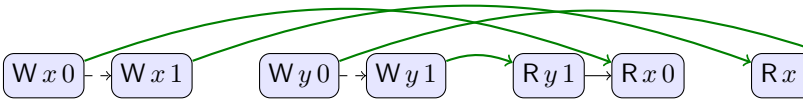
I. Relaxed memory

In §III-J we present an information flow attack on relaxed memory, similar to Spectre in that it relies on speculative evaluation. Unlike Spectre it does not depend on timing attacks, but instead is based on the sensitivity of relaxed memory to data dependencies.

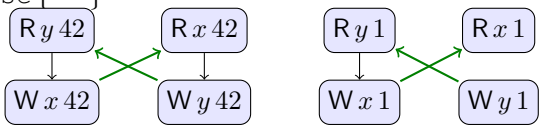
Our model includes concurrent memory accesses, which can introduce concurrent reads-from. Since we are allowing events to be partially ordered, this gives a simple model of relaxed memory. For example an independent read independent write (IRIW) example is:

$$x\texttt{:=}0\texttt{;}\ x\texttt{:=}x{+}1\ \texttt{||}\ y\texttt{:=}0\texttt{;}\ y\texttt{:=}y{+}1\ \texttt{||}\ \texttt{if}\,(x)\,\{\,r\texttt{:=}y\,\}\ \texttt{||}\ \texttt{if}\,(y)\,\{\,s\texttt{:=}$$

which includes the execution:



This model does not introduce thin-air reads (TAR). For example the TAR pit $(x\texttt{:=}y\ \texttt{||}\ y\texttt{:=}x)$ fails to produce a value for x from thin air since this produces a cycle in $\leq$, as shown on the left below:



This cycle can be broken by removing a dependency. For example $(x\texttt{:=}y\ \texttt{||}\ r\texttt{:=}x\texttt{;}\ y\texttt{:=}r+1-r)$ has the execution on the right above. Note that $(\mathsf{R}\,x\,1) \not\leq (\mathsf{W}\,y\,1)$, so this does not introduce a cycle.

Although it is not the primary focus of this paper, our model may be an attractive model of relaxed memory. Many prior models either permit thin-air executions that our model forbids or forbid desirable executions that our model permits. In §V, we develop a logic which allows us to prove that our semantics forbids thin air examples that are permitted by prior speculative models Manson et al. (2005); Jagadeesan et al. (2010a); Kang et al. (2017a).

Our model passes all of the causality test cases Pugh (2004). Significantly, this includes test case 9, which is forbidden by Jeffrey and Riely (2016), one of the few models that disallows the thin air example from §V. We present this test case in the appendix, where we also discuss the thread inlining examples from Manson et al. (2005).

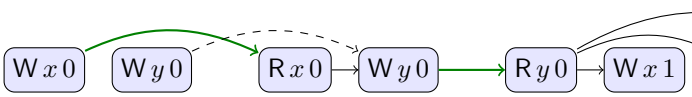
Batty et al. (2015) showed that the thin-air problem has no per-candidate-execution solution for C++. This result does not apply to our model, which has a different notion of dependency.

J. Information flow attacks on relaxed memory

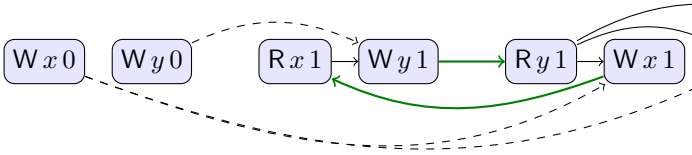
Consider an attacker program, again using security checks to try to learn a SECRET. Whereas SPECTRE uses hardware capabilities, which have to be modeled by adding extra capabilities to the language, this new attacker works by exploiting relaxed memory which can result in unexpected information flows. The attacker program is:

$$\begin{array}{l}\texttt{var}\,x\texttt{:=}0\texttt{;}\ \texttt{var}\,y\texttt{:=}0\texttt{;}\\ \quad y\texttt{:=}x\ \texttt{||}\ \texttt{if}\,(y\texttt{==}0)\,\{\,x\texttt{:=}1\,\}\\ \qquad\qquad \texttt{else}\,\texttt{if}\,(\texttt{canRead}(\texttt{SECRET}))\,\{\,x\texttt{:=}\texttt{SECRET}\,\}\\ \qquad\qquad \texttt{else}\,\{\,x\texttt{:=}1\texttt{;}\ z\texttt{:=}1\,\}\end{array}$$

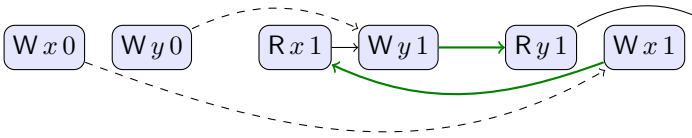
In the case where SECRET is 2, this has many executions, one of which is:



but there are no executions which exhibit $(\mathsf{W}\, z\, 1)$, since any attempt to do so produces a cycle:



In the case where SECRET is 1, there is an execution:



Note that in this case, there is no dependency from $(\mathsf{R}\, y\, 1)$ to $(\mathsf{W}\, x\, 1)$. This lack of dependency makes the execution possible. Thus, if the attacker sees an execution with $(\mathsf{W}\, z\, 1)$, they can conclude that SECRET is 1, which is an information flow attack.

This attack is not just an artifact of the model, since the same behavior can be exhibited by compiler optimizations. Consider the program fragment:

$$\texttt{if}\,(y = 0)\,\{\,x := 1\,\}\,\texttt{else}\,\texttt{if}\,(\texttt{canRead(SECRET)})\,\{\,x := \texttt{SECRET}\,\}\,\texttt{else}\,\{\,x := 1;\, z := 1\,\}$$

In the case where SECRET is a constant 1, the compiler can inline it:

$$\texttt{if}\,(y = 0)\,\{\,x := 1\,\}\,\texttt{else}\,\texttt{if}\,(\texttt{canRead(SECRET)})\,\{\,x := 1\,\}\,\texttt{else}\,\{\,x := 1;\, z := 1\,\}$$

and lift the assignment to x out of the if statement:

$$x := 1;\,\texttt{if}\,(y = 0)\,\{\,\}\,\texttt{else}\,\texttt{if}\,(\texttt{canRead(SECRET)})\,\{\,\}\,\texttt{else}\,\{\,z := 1\,\}$$

After these optimizations, a sequentially consistent execution exhibits $(\mathsf{W}\, z\, 1)$. We discuss the practicality of this attack further in §IV.

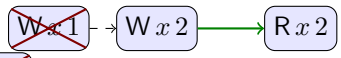
This approach can be generalized to detect information flows in arbitrary code. If we replace the code fragment above with:

$$\texttt{if}\,(y = 0)\,\{\,x := P(0)\,\}\,\texttt{else}\,\{\,x := P(1);\, z := 1\,\}$$

then $(\mathsf{W}\, z\, 1)$ is possible only if P is independent of its input. Thus, the conditional is able to capture multiple executions, as in Barthe et al. (2004).

K. Dead store elimination

A common compiler optimization is *dead store elimination*, in which writes are omitted if they will be overwritten by a subsequent write later in the same thread. We can model eliminated writes by ones with an unsatisfiable precondition. For example, one execution of $(x := 1;\, x := 2) \;||\; (r := x)$ is:



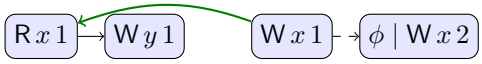
Recall that for any satisfiable e, if e reads x from y then d is a tautology. This means that, although we can eliminate $(\mathsf{W}\, x\, 1)$ we cannot eliminate $(\mathsf{W}\, x\, 2)$.

One heuristic that a compiler might adopt is to only eliminate writes that are guaranteed to be followed by another write to the same variable. This can be formalized by saying that d is eliminable if there is a $e \not\leq d$ such that e is a tautology and d writes to every location e writes to. A model of dead store elimination is one where, in every pomset, every eliminable event is unsatisfiable. This simple model includes the examples above.

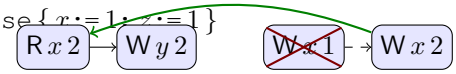
Note that if dead store elimination is *always* performed, then there is an information flow attack similar to the one in §III-J. Consider the program:

$$y := x \;||\; x := 1;\; \texttt{if}\,(\texttt{canRead(SECRET)})\,\{\,\texttt{if}\,(\texttt{SECRET})\,\{\,x := 2\,\}\,\}\;\texttt{else}\,\{\,x := 2\,\}$$

In the case that SECRET is 0, there is an execution:



where ϕ is $(\neg\texttt{canRead(SECRET)})$, which is not a tautology, and so the $(\mathsf{W}\, x\, 1)$ event is not eliminated. In the case that SECRET is not 0, the matching execution is:



Now the $(\mathsf{W}\, x\, 2)$ event is a guaranteed write, so the $(\mathsf{W}\, x\, 1)$ is eliminated, and so cannot be read. In the case that the attacker can rely on dead store elimination taking place, this is an information flow: if the attacker observes x to be 1, then they know SECRET is 0. We return to this attack in §IV.

L. Release/acquire synchronization

In relaxed memory models, synchronization actions act as memory fences: that is, they are a barrier to reordering memory accesses. In this section, we present a simple model of release/acquire fencing. In §III-M, we show that this can be scaled up to a model of transactional memory.

We assume there are sets Rel and Acq $\subseteq \mathcal{A}$. We say that a is a *release action* if $a \in$ Rel and a is an *acquire action* if $a \in$ Acq. In a pomset, a release event is one labelled with a release action, and an acquire event is one labelled by an acquire action. To give the semantics of fences, we add extra constraints to Definition II.8 of prefixing (recalling that c is the event being introduced):

- $c \leq e$ whenever c is an acquire event or e is a release event, and
- if c is an acquire event then e is independent of x, for every x.

The first constraint ensures that events are ordered before a release and after an acquire. The second constraint ensures that thread-local reads do not cross acquire fences.

In examples, we will use releasing writes and acquiring reads:

- $(\mathsf{Rel}\, x\, v)$, a release action that writes v to x, and
- $(\mathsf{Acq}\, x\, v)$, an acquire action that reads v from x.

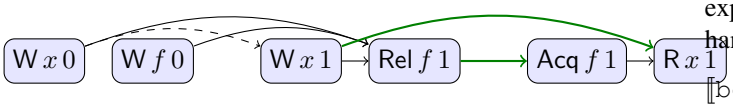
The semantics of programs with releasing write and acquiring read are similar to regular write and read, with $\mathsf{Rel}\, x\, v$ replacing $\mathsf{W}\, x\, v$ and $\mathsf{Acq}\, x\, v$ replacing $\mathsf{R}\, x\, v$:

$$\begin{aligned}[\![\mathsf{rel}\, x := M;\ C]\!] &= \bigcup_v \left((M = v) \rhd (\mathsf{Rel}\, x\, v) \to [\![C]\!][M/x]\right)\\ [\![\mathsf{acq}\, r := x;\ C]\!] &= \bigcup_v (\mathsf{Acq}\, x\, v) \to [\![C]\!][x/r]\end{aligned}$$

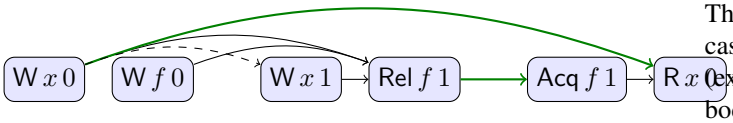
To see the need for the first constraint on prefixing, consider the program:

$$\mathtt{var}\, x := 0;\ \mathtt{var}\, f := 0;\ (x := 1;\ \mathsf{rel}\, f := 1 \ ||\ \mathsf{acq}\, r := f; s := x)$$

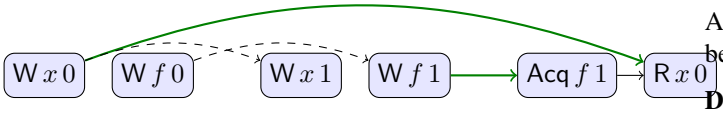
This has an execution:



but *not*:



since $(\mathsf{W}\, x\, 0) \not< (\mathsf{W}\, x\, 1) < (\mathsf{R}\, x\, 0)$, so this pomset does not satisfy the requirements to be x-closed. If we replace the release with a plain write, then the outcome $(\mathsf{Acq}\, f\, 1)$ and $(\mathsf{R}\, x\, 0)$ is possible:

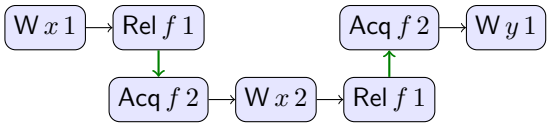


since no order is required between $(\mathsf{W}\, x\, 1)$ and $(\mathsf{W}\, f\, 1)$. Symmetrically, if we replace the acquire of the original program with a plain read, then the outcome $(\mathsf{R}\, f\, 1)$ and $(\mathsf{R}\, x\, 0)$ is possible.

To see the need for the second constraint on prefixing, consider the program:

$$(x := 1;\ \mathsf{rel}\, f := 1;\ \mathsf{acq}\, r := f;\ y := x) \ ||\ (\mathsf{acq}\, s := f;\ x := 2;\ \mathsf{rel}\, f := 2;)$$

whose semantics includes execution:



This execution exists because $[\![y := x]\!]$ includes $(x = 1 \mid \mathsf{W}\, y\, 1)$ and the precondition $x = 1$ is fulfilled by the preceding write $x := 1$. In implementation term, this execution is reading 1 from x in a "stale cache." The alternative execution that attempts to read 1 from the x in "main memory," has an explicit $(\mathsf{R}\, x\, 1)$ between $(\mathsf{Acq}\, f\, 2)$ and $(\mathsf{W}\, y\, 1)$, and thus will fail to be x-closed.

To prevent thread-local writes from crossing release/acquire pairs, we require that pomsets in the semantics of acquire have no free locations. This corresponds to the idea that acquires flush the read cache, and therefore reads must reload values from main memory after an acquire.

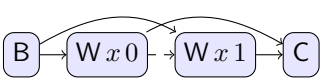
M. Transactions

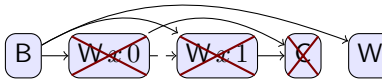
We present a model of transactional memory Larus and Rajwar (2007) that is sufficient to capture PRIME+ABORT attacks Disselkoen et al. (2017). We make several simplifying assumptions: transactions are serializable, strongly isolated, and only abort due to cache conflicts. To model the latter, we assume that the set of locations $\mathcal{X}$ is partitioned into *cache sets*.

The action $(\mathsf{B}\, v) \in \mathsf{Acq}$ represents the begin of a transaction with id v and $(\mathsf{C}\, v) \in \mathsf{Rel}$ represents the corresponding commit. We model a language in which transactions have explicit identifiers (which we elide in examples) and abort handlers (which we elide when they are empty):

$$\begin{aligned}[\![\mathtt{begin}\, v;\ C;\ \mathtt{onabort}\, v\, \{\, D\,\}]\!] &= (\mathsf{B}\, v) \to \left([\![C]\!] \cup \left((\mathsf{false} \rhd [\![C]\!]) \right.\right.\right.\|\\ [\![\mathtt{commit}\, v;\ D]\!] &= (\mathsf{C}\, v) \to [\![D]\!]\end{aligned}$$

The semantics of a transaction has two cases: a committed case (executing only the transaction body) and an aborted case (executing both the body and the recovery code, where the body is marked unsatisfiable). For example, two executions of $(\mathtt{begin};\ x := 1;\ x := 2;\ \mathtt{commit};\ \mathtt{onabort}\, \{y := 1\})$ are:





At top level, we require that pomsets be *serializable*, as defined below.

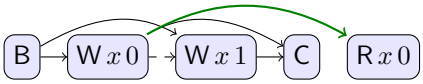
Definition III.1. We say that event c *matches* b if $\lambda(c) = (\mathsf{C}\, v)$ and $\lambda(b) = (\mathsf{B}\, v)$. We say that begin event b *begins* e if $b \le e$ and there is no intervening matching commit; in this case e *belongs to* b. We say that commit event c *commits* e if $e \le c$ and there is no intervening matching begin.

Definition III.2. A pomset is *serializable* if:

1) no two begins have the same id,
2) every commit follows the matching begin,
3) $\le$ totally orders tautological begins and commits,
4) if b begins e, but not d, and $d \le e$ then $d \le b$,
5) if c ends e, but not d, and $e \le d$ then $c \le d$,
6) if e and d belong to b and read the same location, then both read the same value, and
7) if e belongs to b, then e implies some matching c that ends e.

Conditions 1-5 ensure serializability of committed transactions. Conditions 4-6 also ensure strong isolation for non-transactional events Dongol et al. (2018). Condition 7 ensures that all events in aborted transactions are unsatisfiable. For

example Conditions 5 and 7 rule out executions (which violate strong isolation and atomicity):



In order to model PRIME+ABORT, we need a mechanism for modeling *why* a transaction aborts, as this can be used as a back channel. We model a simple form of concurrent transaction, which aborts when it encounters a memory conflict—this is similar to the treatment of `touched` in §III-H.

Definition III.3. A commit event c matching b *aborts due to memory conflict* if there is some e ended by c, and some tautologous $b \not\succ d \not\succ c$ that does not belong to b such that e and d touch locations in the same cache set.

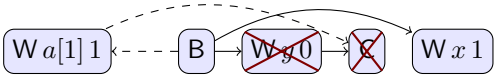
PRIME+ABORT requires an honest agent whose cache-set access depends upon a secret. If $a[0]$ and $a[1]$ belong to separate cache sets, then such an honest agent is:

$$a[\texttt{SECRET}] := 1$$

The attack relies on discovery of some y which belongs to the cache-set of $a[1]$. Then the program

$$\texttt{begin}; y := 0; r := \texttt{commit}; \texttt{onabort}\,\{x := 1\}$$

can write 1 to x if the `SECRET` is 1, in which case the following execution is possible.



If the attacker knows that commits only abort due to memory conflicts, then this attack is an information flow, since the memory conflict only happens when the `SECRET` is 1.

IV. EXPERIMENTS

One theme of this paper is that optimizations not typically part of formal abstractions can result in information flow leaks. This is typified by the Spectre attack, which leverages speculative execution, a hardware optimization. §III-J and §III-K presented other attacks along the same line, which leverage compiler optimizations. These attacks also, unlike Spectre, do not rely on timing side channels, or indeed timers of any kind, bypassing many common Spectre mitigations Kohlbrenner and Shacham (2016); Wagner (2018).

In this section we present implementations of the attacks described in §III-J and §III-K, in both cases exploiting compiler optimizations to construct an information flow attack. We demonstrate the efficacy of our proof-of-concept attacks against the `clang` and `gcc` C compilers. All of our experiments are performed on a Debian 9 machine with an Intel i7-6500U processor and 8 GB RAM; we test against `gcc` 6.3.0 and `clang` 3.8.

A. Attacker model

In our attacker model, we assume that there is a `SECRET` hardcoded into an application; for instance, `SECRET` may be an API key. This `SECRET` is known at compile time, but may not be accessed except behind a security check. Since the attacker is running with low security privileges, the security check always fails, so the attacker can only access the `SECRET` in dead code. The attacker's goal is to learn the value of the `SECRET`.

As a running hypothetical example, suppose there is a library that contains a hardcoded `SECRET`:

```
static const uint SECRET = 0x1234;
static volatile bool canReadSecret = false;
```

The attacker is not allowed to write to `canReadSecret` or read from `SECRET` except after performing an `if(canReadSecret)` check.

This is not necessarily a realistic attacker model, since in most cases secrets are only known at run time rather than compile time, which means that the attacks presented in this section are more proof-of-concepts rather than immediately exploitable vulnerabilities. However, the mechanisms we use are novel and could potentially be applied in other contexts. For instance, many real-world contexts allow untrusted or third-party entities to write code in a scripting language which is then compiled alongside and integrated into a larger application, often using a just-in-time (JIT) compiler. JavaScript code from third-party websites running in a browser is a common example of this. Although we consider only attacks using C code against a C compiler, one could imagine a similar attack using JavaScript against browser JIT compilers, where the compiler may have access to interesting secrets such as the browser's cookie store, and may be able to optimize based on those secrets. We plan to explore JavaScript attacks of this type as future work.

B. Load-store reordering attack

We begin by examining the attack in §III-J in more detail. We show that by exploiting compiler optimizations which perform load-store reordering, an attacker can learn the value of a compile-time `SECRET` despite only being allowed to use it inside dead code. We verified that this attack succeeds against `gcc` version 6.3.0.

The form of the attack presented in §III-J works in theory, but in practice, just because a compiler is *allowed* to perform a load-store reordering doesn't mean that it *will*. We found that `gcc` and `clang` chose to read y into a register first (before writing to x), regardless of the value of `SECRET`. However, using a similar program we were able to coax `gcc` to emit a different ordering of the read of y and the write of x depending on the value of a `SECRET`:

```
var x := 0; var y := 0;
  y := x || x := 1;
          if (canReadSecret) { x := SECRET }
          if (y > 0) { z := 0 } else { z := 1 }
```

SECRET == 0	SECRET == 1
mov s(%rip), %eax	mov s(%rip), %eax
mov $1, x(%rip)	mov y(%rip), %eax
test %eax, %eax	mov $1, x(%rip)
je label1	test %eax, %eax
mov $0, x(%rip)	sete %eax
label1:	
mov y(%rip), %eax	
test %eax, %eax	
sete %eax	

Fig. 1. Simplified x86 assembly output from `gcc` for the main thread of the load-store reordering attack. In particular, note that the order between (`mov $1, x(%rip)`) and (`mov y(%rip), %eax`) is different in the two cases. References to the `canReadSecret` variable have been shortened to `s` for the figure.

Redundancy	Bandwidth (bits/s)	Bitwise Acc	Per-run Acc
1	3.14 million	90.89%	1.9%
2	1.56 million	96.04%	8.1%
3	1.04 million	98.09%	10.0%
4	783 thousand	98.98%	24.3%
5	626 thousand	99.71%	50.2%
7	447 thousand	99.91%	70.6%
10	314 thousand	99.991%	93.8%
15	208 thousand	99.994%	95.5%
20	157 thousand	99.9995%	99.2%
30	105 thousand	99.99995%	99.9%

Fig. 2. Performance results for the load-store reordering attack when leaking a 2048-bit secret. 'Redundancy' is the number of redundant runs performed for error correction; more redundant runs improves accuracy but reduces bandwidth. 'Bandwidth' is the number of bits leaked per second after accounting for any error correction. 'Bitwise Accuracy' is the percentage of bits that were correct, while 'Per-run Accuracy' is the percentage of full 2048-bit secrets that were correct in all bit positions.

Figure 1 shows the assembly output of `gcc` on this program in the cases where `SECRET` is 0 and 1 respectively. In the case that `SECRET` is 1, `gcc` removes the `if` statement entirely, and moves the read of y above the write of x. However, when `SECRET` is 0, the `if` statement must remain intact, and `gcc` does not move the read of y. This means that if `SECRET` is 1, the second thread will always read $y == 0$ and always assign $z := 1$. However, if `SECRET` is 0, it is possible that the first thread may observe $x == 1$ and write $y := 1$ in time for the second thread to observe $y == 1$ and thus assign $z := 0$. In this way, we leverage compiler load-store reordering to learn the value of a compile-time `SECRET`.

We extend this attack to leak a secret consisting of an arbitrary number `N` of bits. To do this, we compile `N` copies of the test function, each performing a boolean test on a single bit of `SECRET`. So that the bit value is constant at compile time, we must compile a separate function for each bit, rather than execute the same code repeatedly in a loop.

We make three additional tweaks to improve the reliability, so that the attacker can confidently infer the value of `SECRET` based on the observed value of z. First, rather than performing $y := x$ only once in the forwarding thread, we perform $y := x$ continuously in a loop. This maximizes the probability that, once $x := 1$ occurs in the main thread, y will be immediately assigned 1 by the forwarding thread and the main thread will be able to read $y == 1$.

Second, we wish to lengthen the timing window between $x := 1$ and the read of y in the main thread (in the case where `SECRET` is 0 and the read of y remains below $x := 1$). However, we wish to do this in a way that does not block the reordering of the read of y upwards in the case where `SECRET` is 1. We do this by inserting many copies of the line

$$\texttt{if}\,(\texttt{canReadSecret})\,\{\,x := \texttt{SECRET}\,\}$$

instead of just one. In the case where `SECRET` is 0, this results in many reads of `canReadSecret` and many conditional jumps, which in practice creates a timing window for the forwarding thread to perform $y := x$. However, in the case where `SECRET` is 1, all of these inserted lines can be removed just as a single copy could be. In practice, we found that inserting too many copies of the line prevents `gcc` from reordering the read of y above the write to x as desired; inserting 30 copies was sufficient to create a timing window while still allowing the desired reordering.

Finally, we redundantly execute the entire attack several times, noting the value of z in each case. We note that if *any* of the redundant runs produces a value of $z == 0$ for a particular bit position, then we can be certain that the corresponding bit of `SECRET` *must* be 0, as it implies the read of y was not reordered upwards in that particular function. On the other hand, the more runs that produce a value of $z == 1$ for a particular bit position, the more certain we can be that the read of y was reordered above the $x := 1$ assignment, and `SECRET` is 1.

Figure 2 gives the performance results for this attack against `gcc` version 6.3.0. The attack can sustain hundreds of thousands of bits per second leaked with near-perfect accuracy, or millions of bits per second with error rates of a few percent. This means that an attacker can leak a 2048-bit secret with near-perfect accuracy in under 10 ms. Note that this bandwidth assumes that all copies of the attack function are already compiled; the cost of compilation is not included here.

C. Dead store elimination attack

In this section we return to the attack in §III-K based on dead store elimination. We show that in our attacker model (given in §IV-A), the attacker is able to exploit dead store elimination to again learn the value of a compile-time `SECRET` despite only being allowed to use it inside dead code. This attack is even more efficient than the attack on load-store reordering, and further, we were able to demonstrate its effectiveness against both `gcc` and `clang`.

We start from the simple form of the attack presented in §III-K, and extend it to leak a secret consisting of an arbitrary number of bits, in the same way that we extended the load-store reordering attack. We make three additional tweaks to improve the reliability so that the attacker can

Redundancy	Bandwidth (bits/s)	Bitwise Acc	Per-run Acc
1	1.19 million	99.991%	95.6%
2	597 thousand	99.99986%	99.7%
3	397 thousand	100.0%	100.0%

Fig. 3. Performance results for the dead store elimination attack on `clang` when leaking a 2048-bit secret. Terms are the same as defined in the caption for Figure 2.

Stall amount	10	20	50	100	200	500
Redundancy 1	2.54 million 98.15%	2.36 million 99.80%	1.95 million 99.987%	1.54 million 99.996%	1.12 million 99.995%	584 thousand 99.998%
Redundancy 2	1.24 million 99.73%	1.17 million 99.993%	989 thousand 100.0%	774 thousand 100.0%	555 thousand 100.0%	295 thousand 100.0%
Redundancy 3	841 thousand 99.94%	784 thousand 100.0%	666 thousand 100.0%	521 thousand 100.0%	370 thousand 100.0%	201 thousand 100.0%
Redundancy 4	620 thousand 99.992%	585 thousand 100.0%	499 thousand 100.0%	387 thousand 100.0%	285 thousand 100.0%	145 thousand 100.0%

Fig. 4. Performance results for the dead store elimination attack on `gcc` when leaking a 2048-bit secret. Rows give different values of 'redundancy' (as defined in previous figures), while columns give amounts of stall time immediately following the $x := 1$ write (as measured in loop iterations). Each table cell gives the leak bandwidth in bits/sec, followed by the bitwise accuracy.

confidently infer the value of `SECRET`. Two of them follow exactly the same pattern as the reliability tweaks for the load-store reordering attack in §IV-B — continuously forwarding x to y in the forwarding thread, and running the entire attack multiple times. The remaining tweak is again motivated by increasing the timing window in which the forwarding can happen, but differs in some details from the implementation in §IV-B.

To increase the timing window, we insert additional time-consuming computation immediately following the $x := 1$ operation in the main thread. This increases the likelihood that the listening thread will be able to observe $x == 1$ (unless the $x := 1$ write was eliminated). Inserting this computation should be done without interfering with the dead store elimination process itself, so that the compiler will continue to eliminate the $x := 1$ write if and only if `SECRET` was 1. For `gcc`, we have a fair amount of freedom with the time-consuming computation — for instance, we can use an arbitrarily long loop. In fact, we can perform a further optimization by monitoring the value of the variable y (written to by the listening thread) and breaking out of the loop early if we see that the listening thread has already observed $x == 1$. However, with `clang`, we cannot use a loop at all — the time-consuming computation must be branch-free and, furthermore, must not consist of too many instructions. Nonetheless, we find that even with these restrictions, we are able to construct a reliable and fast attack against both `clang` and `gcc`.

Performance results for the dead store elimination attack against `clang` are given in Figure 3, and against `gcc` are given in Figure 4. Both attacks are faster than the load-store-reordering attack from §IV-B when comparing settings which give the same accuracy. In particular, the attack on `gcc` can leak a 2048-bit cryptographic key with perfect accuracy (in our tests) in about 2 ms.

V. Logic

In this section, we develop sufficient logical infrastructure to prove that our semantics disallows thin air executions. We present a variant of the TAR-pit example from §III-I which poses difficulties under many speculative semantics.

We adapt past linear temporal logic (PLTL) Lichtenstein et al. (1985) to pomsets by dropping the previous instant operator and adopting strict versions of the temporal operators. The atoms of our logic are write and read events. Given an pomset P and event e, define:

$$\begin{aligned}
P, e &\models \mathsf{W}\, x\, v, \text{ if } \lambda(e) = (\mathsf{true}, \mathsf{W}\, x\, v)\\
P, e &\models \mathsf{R}\, x\, v, \text{ if } \lambda(e) = (\mathsf{true}, \mathsf{R}\, x\, v)\\
P, e &\models \phi \wedge \psi, \text{ if } P, e \models \phi \text{ and } P, e \models \psi\\
P, e &\models \mathsf{true}\\
P, e &\models \neg\phi, \text{ if } P, e \not\models \phi\\
P, e &\models \Box^{-1}\phi, \text{ if } (\forall d \le e,\, d \neq e)\ P, d \models \phi
\end{aligned}$$

Define $P \models \phi$ if $(\forall e \in E)\ P, e \models \phi$ and $\mathcal{P} \models \phi$ if $(\forall P \in \mathcal{P})\ P \models \phi$.

Let $\Diamond^{-1}\phi$ be defined as $\neg(\Box^{-1}\neg\phi)$. In addition, let false, $\vee$ and $\Rightarrow$ be defined in the standard way.

The past operators do not include the current instant, and thus they do *not* satisfy the rule $\Box^{-1}\phi \Rightarrow \Diamond^{-1}\phi$. However, they do satisfy:

$$(\phi \Rightarrow \Diamond^{-1}\phi) \Rightarrow \neg\phi \qquad \text{(Coinduction)}$$

$$(\Box^{-1}\phi \Rightarrow \phi) \Rightarrow \phi \qquad \text{(Induction)}$$

Note that $P \models \phi \wedge \Box^{-1}\phi$ whenever $P \models \phi$.

We now present two proof rules. The first rule captures the semantics of local variables. Define $\mathsf{closed}(x) = (\mathsf{R}\, x\, v \Rightarrow \Diamond^{-1}\mathsf{W}\, x\, v)$. Although this definition does not mention intervening writes, it is sufficient for our example. It is straightforward to establish that following rule is sound:

$$\frac{\phi \text{ is independent of } x \qquad P \models \mathsf{closed}(x) \Rightarrow \phi}{\nu x\,.\, P \models \phi} \qquad \text{(Closing } x\text{)}$$

The second rule describes composition, in the style of Abadi and Lamport Abadi and Lamport (1993). To simplify the presentation, we consider the special case with a single invariant. In order to state the theorem, we generalize the satisfaction relation to include environment assumptions. Let $\mathsf{Models}(\phi) = \{\mathcal{P} \mid \mathcal{P} \models \phi\}$ be the set of pomsets that satisfy ϕ. We say that ϕ is prefix closed if $\mathsf{Models}(\phi)$ is prefix-closed[1]. Define $\phi, \mathcal{P} \models \psi$ if $\mathsf{Models}(\phi) \parallel \mathcal{P} \models \psi$.

Proposition V.1. *Let ϕ be prefix-closed. Let $\mathcal{P}_1, \mathcal{P}_2$ be augmentation-closed[2]. Then:*

$$\frac{\phi, \mathcal{P}_1 \models \phi \qquad \phi, \mathcal{P}_2 \models \phi}{\mathcal{P}_1 \parallel \mathcal{P}_2 \models \phi} \qquad \text{(Composition)}$$

[1] P' is a prefix of P if $E' \subseteq E$, $e \in E'$ and $d \le e$ imply $d \in E'$, and $(\lambda', \le', \not<') $ coincide with $(\lambda, \le, \not<)$ for elements of E'.

[2] P' is an augmentation of P if $E' = E$, $e \le d$ implies $e \le' d$, $e \not< d$ implies $e \not<' d$, and if $\lambda(e) = (\psi \mid b)$ then $\lambda'(e) = (\psi' \mid b)$ where ψ' implies ψ.

Proof sketch. We will show that all prefixes in the prefix closures of $\mathcal{P}_1 \parallel \mathcal{P}_2$ satisfy the required property. Proof proceeds by induction on prefixes of $P \in \mathcal{P}_1 \parallel \mathcal{P}_2$.

The case for empty prefix follows from assumption that ϕ is prefix closed.

For the inductive case, consider $P \in P_1 \parallel P_2$ where $P_i \in \mathcal{P}_i$. Since $\mathcal{P}_1$ and $\mathcal{P}_2$ are augmentation closed, we can assume that the restriction of P to the events of P_i coincides with P_i, for $i = 1, 2$. Consider a prefix P' derived by removing a maximal element e from P. Suppose e comes from P_1 (the other case is symmetric). Since P_2 is a prefix of P' and $P' \models \phi$ by induction hypothesis, we deduce that $P_2 \models \phi$. Since $P_1 \in \mathcal{P}_1$, by assumption $\phi, \mathcal{P}_1 \models \phi$ we deduce that $P \models \phi$. □

We now turn the conditional TAR-pit program, which is a variant of (Lochbihler, 2013, Figure 8):

$$\texttt{var}\, x := 0; \texttt{var}\, y := 0; \texttt{var}\, z := 0; (y := x \;||\; \texttt{if}\,(z)\,\{x := 1\}\,\texttt{else}\,\{z := x\}) \;||\; \texttt{if}\,(x)\,\{a := 1\}$$

This program is allowed to write 1 to a under many speculative memory models Manson et al. (2005); Jagadeesan et al. (2010a); Kang et al. (2017a), even though the read of 1 from y in the else branch of the second thread arises out of thin air. In contrast, we prove the formula $\neg\Diamond^{-1}(\mathsf{W}\, a\, 1)$ holds for the models of this program in our semantics. We start with the following invariant, which holds for each of the three threads, and thus, by composition, for the aggregate program:

$$[\Diamond^{-1}(\mathsf{W}\, y\, 1) \Rightarrow \Diamond^{-1}(\mathsf{R}\, x\, 1)] \wedge [\Diamond^{-1}(\mathsf{W}\, a\, 1) \Rightarrow (\Diamond^{-1}(\mathsf{R}\, y\, 1) \wedge \Box^{-1}(\mathsf{W}\, x\, 1 \Rightarrow \Diamond^{-1}(\mathsf{R}\, x\, 1)))]$$

Closing y, we have, $\Diamond^{-1}(\mathsf{R}\, y\, 1) \Rightarrow \Diamond^{-1}(\mathsf{W}\, y\, 1)$ which we substitute into the left conjunct to get:

$$\Diamond^{-1}(\mathsf{R}\, y\, 1) \Rightarrow \Diamond^{-1}(\mathsf{R}\, x\, 1)$$

which in turn we substitute into the right conjunct to get:

$$\Diamond^{-1}(\mathsf{W}\, a\, 1) \Rightarrow (\Diamond^{-1}(\mathsf{R}\, x\, 1) \wedge \Box^{-1}(\mathsf{W}\, x\, 1 \Rightarrow \Diamond^{-1}(\mathsf{R}\, x\, 1)))$$

Closing x, we can replace $\Diamond^{-1}(\mathsf{R}\, x\, 1)$ with $\Diamond^{-1}(\mathsf{W}\, x\, 1)$:

$$\Diamond^{-1}(\mathsf{W}\, a\, 1) \Rightarrow (\Diamond^{-1}(\mathsf{W}\, x\, 1) \wedge \Box^{-1}(\mathsf{W}\, x\, 1 \Rightarrow \Diamond^{-1}(\mathsf{W}\, x\, 1)))$$

Applying coinduction to the right conjunct, we have:

$$\Diamond^{-1}(\mathsf{W}\, a\, 1) \Rightarrow (\Diamond^{-1}(\mathsf{W}\, x\, 1) \wedge \Box^{-1}(\neg\mathsf{W}\, x\, 1))$$

Simplifying, we have, as required:

$$\Diamond^{-1}(\mathsf{W}\, a\, 1) \Rightarrow \mathsf{false}$$

VI. Conclusions and future work

In this paper, we have presented a model of speculative evaluation and shown that it captures non-trivial properties of speculations produced by hardware, compiler optimizations, and transactions. These properties include information flow attacks: in the case of hardware and transactions this is modeling known attacks Kocher et al. (2019); Disselkoen et al. (2017), but in the case of compiler optimizations the attacks are new, and were discovered as a direct result of developing the model. We have experimentally validated that the attacks can be carried out against `gcc` and `clang`, though only against secrets known at compile time.

The model of relaxed memory used in this paper is deliberately simplified, compared for example to C11 Boehm and Adve (2008); Batty et al. (2011). In particular our model of reads-from is strong, and could be weakened by replacing the requirement $d < e$ in Definition II.4 by $e \not< d$. It remains to be seen how this impacts the model, in particular the logical formulation of x-closure in §V as $((\mathsf{R}\, x\, v) \Rightarrow \Diamond^{-1}(\mathsf{W}\, x\, v))$ would no longer be sound.

The design space for transactions is very rich Dongol et al. (2018). We have only presented one design choice, and it remains to be seen how other design choices could be adopted. For example we have chosen not to distinguish commits that are aborted due to transaction failure from commits which are aborted for other reasons, such as failed speculation.

In future work, it would be interesting to see if full-abstraction results for pomsets Plotkin and Pratt (1997) can be extended to 3-valued pomsets.

One interesting feature of this model is that (in the language of Pichon-Pharabod and Sewell (2016)) it is a *per-candidate execution model*, in that the correctness of an execution only requires looking at that one execution, not at others. This is explicit in memory models such as Jagadeesan et al. (2010b); Kang et al. (2017b) in which "alternative futures" are explored, in a style reminiscent of Abramsky's bisimulation as a testing equivalence Abramsky (1987). Models of information flow are similar, in that they require comparing different runs to test for the presence of dependencies Clarkson and Schneider (2010). In contrast, the model presented here explicitly captures dependency in the pomset order, and models multiple runs by giving the semantics of `if` in terms of a concurrent semantics of both branches. In the parlance of information flow Barthe et al. (2004), the humble conditional suffices to construct a composition operator to detect information flow in the presence of speculation.

References

Martín Abadi and Leslie Lamport. 1993. Composing Specifications. *ACM Trans. Program. Lang. Syst.* 15, 1 (Jan. 1993), 73–132. https://doi.org/10.1145/151646.151649

Samson Abramsky. 1987. Observation equivalence as a testing equivalence. *Theoretical Computer Science* 53, 2 (1987), 225 – 241. https://doi.org/10.1016/0304-3975(87)90065-X

Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Trans. Program. Lang. Syst.* 36, 2, Article 7 (July 2014), 74 pages. https://doi.org/10.1145/2627752

Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. 2004. Secure Information Flow by Self-Composition. In *Proceedings of the 17th IEEE Workshop on Computer Security Foundations (CSFW '04)*. IEEE Computer Society, Washington, DC, USA, 100–114. https://doi.org/10.1109/CSFW.2004.17

Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. 2015. The Problem of Programming Language Concurrency Semantics. In *Proc. European Symp. on Programming*. 283–307.

Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ Concurrency. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '11)*. ACM, New York, NY, USA, 55–66. https://doi.org/10.1145/1926385.1926394

Arnab Kumar Biswas, Dipak Ghosal, and Shishir Nagaraja. 2017. A Survey of Timing Channels and Countermeasures. *ACM Comput. Surv.* 50, 1, Article 6 (March 2017), 39 pages. https://doi.org/10.1145/3023872

Hans-J. Boehm and Sarita V. Adve. 2008. Foundations of the C++ Concurrency Memory Model. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. ACM, New York, NY, USA, 68–78. https://doi.org/10.1145/1375581.1375591

S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. 1984. A Theory of Communicating Sequential Processes. *J. ACM* 31, 3 (June 1984), 560–599. https://doi.org/10.1145/828.833

Nathan Chong, Tyler Sorensen, and John Wickerson. 2018. The semantics of transactions and weak memory in x86, Power, ARM, and C++. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*. 211–225. https://doi.org/10.1145/3192366.3192373

Michael R. Clarkson and Fred B. Schneider. 2010. Hyperproperties. *J. Comput. Secur.* 18, 6 (Sept. 2010), 1157–1210. http://dl.acm.org/citation.cfm?id=1891823.1891830

Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean M. Tullsen. 2017. Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017.*, Engin Kirda and Thomas Ristenpart (Eds.). USENIX Association, 51–67. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/disselkoen

Brijesh Dongol, Radha Jagadeesan, and James Riely. 2018. Transactions in relaxed memory architectures. *PACMPL* 2, POPL (2018), 18:1–18:29. https://doi.org/10.1145/3158106

Jay L. Gischer. 1988. The equational theory of pomsets. *Theoretical Computer Science* 61, 2 (1988), 199–224. https://doi.org/10.1016/0304-3975(88)90124-7

James W. Gray, III. 1992. Toward a Mathematical Foundation for Information Flow Security. *J. Comput. Secur.* 1, 3-4 (May 1992), 255–294. http://dl.acm.org/citation.cfm?id=2699806.2699811

C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (Oct. 1969), 576–580. https://doi.org/10.1145/363235.363259

Radha Jagadeesan, Corin Pitcher, and James Riely. 2010a. Generative Operational Semantics for Relaxed Memory Models. In *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings (Lecture Notes in Computer Science)*, Andrew D. Gordon (Ed.), Vol. 6012. Springer, 307–326. https://doi.org/10.1007/978-3-642-11957-6_17

Radha Jagadeesan, Corin Pitcher, and James Riely. 2010b. Generative Operational Semantics for Relaxed Memory Models. In *Proceedings of the 19th European Conference on Programming Languages and Systems (ESOP'10)*. Springer-Verlag, Berlin, Heidelberg, 307–326. https://doi.org/10.1007/978-3-642-11957-6_17

A. Jeffrey and J. Riely. 2016. On Thin Air Reads Towards an Event Structures Model of Relaxed Memory. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*, M. Grohe, E. Koskinen, and N. Shankar (Eds.). ACM, 759–767. https://doi.org/10.1145/2933575.2934536

J. Kang, C.-K. Hur, O. Lahav, V. Vafeiadis, and D. Dreyer. 2017a. A promising semantics for relaxed-memory concurrency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 175–189. https://doi.org/10.1145/3009837

Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017b. A Promising Semantics for Relaxed-memory Concurrency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 175–189. https://doi.org/10.1145/3009837.3009850

Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*.

David Kohlbrenner and Hovav Shacham. 2016. Trusted Browsers for Uncertain Times. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 463–480. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/kohlbrenner

Leslie Lamport. 1986. On Interprocess Communication. Part I: Basic Formalism. *Distributed Computing* 1, 2 (1986), 77–85. https://doi.org/10.1007/BF01786227

Butler W. Lampson. 1973. A Note on the Confinement Problem. *Commun. ACM* 16, 10 (Oct. 1973), 613–615. https://doi.org/10.1145/362375.362389

Jim Larus and Ravi Rajwar. 2007. *Transactional Memory (Synthesis Lectures on Computer Architecture)*. Morgan & Claypool Publishers.

Orna Lichtenstein, Amir Pnueli, and Lenore D. Zuck. 1985. The Glory of the Past. In *Proceedings of the Conference*

on Logic of Programs. Springer-Verlag, London, UK, UK, 196–218. http://dl.acm.org/citation.cfm?id=648065.747612

A. Lochbihler. 2013. Making the Java memory model safe. *ACM Trans. Program. Lang. Syst.* 35, 4 (2013), 12:1–12:65. https://doi.org/10.1145/2518191

Jeremy Manson, William Pugh, and Sarita V. Adve. 2005. The Java Memory Model. *SIGPLAN Not.* 40, 1 (Jan. 2005), 378–391. https://doi.org/10.1145/1047659.1040336

H. Mantel, M. Perner, and J. Sauer. 2014. Noninterference under Weak Memory Models. In *2014 IEEE 27th Computer Security Foundations Symposium*. 80–94. https://doi.org/10.1109/CSF.2014.14

Andrew C. Myers. 1999. JFlow: practical mostly-static information flow control. In *26th ACM Symp. on Principles of Programming Languages (POPL)*. 228241. http://www.cs.cornell.edu/andru/papers/popl99/popl99.pdf

Kevin R. O'Neill, Michael R. Clarkson, and Stephen Chong. 2006. Information-Flow Security for Interactive Programs. In *Proceedings of the 19th IEEE Workshop on Computer Security Foundations (CSFW '06)*. IEEE Computer Society, Washington, DC, USA, 190–201. https://doi.org/10.1109/CSFW.2006.16

Zdzisław Pawlak. 1982. Rough sets. *International Journal of Computer & Information Sciences* 11, 5 (01 Oct 1982), 341–356. https://doi.org/10.1007/BF01001956

Jean Pichon-Pharabod and Peter Sewell. 2016. A Concurrency Semantics for Relaxed Atomics That Permits Optimisation and Avoids Thin-air Executions. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 622–633. https://doi.org/10.1145/2837614.2837616

Gordon Plotkin and Vaughan Pratt. 1997. Teams Can See Pomsets (Preliminary Version). In *Proceedings of the DIMACS Workshop on Partial Order Methods in Verification (POMIV '96)*. AMS Press, Inc., New York, NY, USA, 117–128. http://dl.acm.org/citation.cfm?id=266557.266600

W. Pugh. 2004. Causality Test Cases. http://www.cs.umd.edu/~pugh/java/memoryModel/CausalityTestCases.html.

A. Sabelfeld and A. C. Myers. 2006. Language-based Information-flow Security. *IEEE J.Sel. A. Commun.* 21, 1 (Sept. 2006), 5–19. https://doi.org/10.1109/JSAC.2002.806121

Geoffrey Smith and Dennis Volpano. 1998. Secure Information Flow in a Multi-threaded Imperative Language. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '98)*. ACM, New York, NY, USA, 355–364. https://doi.org/10.1145/268946.268975

Inc. CORPORATE SPARC. 1994. *The SPARC Architecture Manual (version 9)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Alasdair Urquhart. 1986. *Many-valued Logic*. Springer Netherlands, Dordrecht, 71–116. https://doi.org/10.1007/978-94-009-5203-4_2

Jeffrey A. Vaughan and Todd Millstein. 2012. Secure Information Flow for Concurrent Programs Under Total Store Order. In *Proceedings of the 2012 IEEE 25th Computer Security Foundations Symposium (CSF '12)*. IEEE Computer Society, Washington, DC, USA, 19–29. https://doi.org/10.1109/CSF.2012.20

Jaroslav Ševčík. 2008. *Program Transformations in Weak Memory Models*. PhD thesis. Laboratory for Foundations of Computer Science, University of Edinburgh.

Luke Wagner. 2018. Mitigations landing for a new class of timing attack. https://blog.mozilla.org/security/2018/01/03/mitigations-landing-new-class-timing-attack/.

J. Todd Wittbold and Dale M. Johnson. 1990. Information Flow in Nondeterministic Systems. In *IEEE Symposium on Security and Privacy*.

Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. 2012. Language-based Control and Mitigation of Timing Channels. *SIGPLAN Not.* 47, 6 (June 2012), 99–110. https://doi.org/10.1145/2345156.2254078

Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2012. Formalizing the LLVM intermediate representation for verified program transformations. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, John Field and Michael Hicks (Eds.). ACM, 427–440. https://doi.org/10.1145/2103656.2103709

APPENDIX

Pugh (2004) developed a set of twenty causality test cases in the process of revising the Java Memory Model (JMM) Manson et al. (2005). Using hand calculation, we have confirmed that our model gives the desired result for all twenty cases, unrolling loops as necessary. Our model also gives the desired results for all of the examples in Batty et al. (2015, §4) and all but one in Ševčík (2008, §5.3): redundant-write-after-read-elimination fails for any sensible non-coherent semantics. Our model agrees with the JMM on the "surprising and controversial behaviors" of Manson et al. (2005, §8), and thus fails to validate thread inlining.

In this section, we discuss three of the causality test cases and the thread inlining from Manson et al. (2005). In presenting the examples, we unroll loops, correct typos and simplify the code.

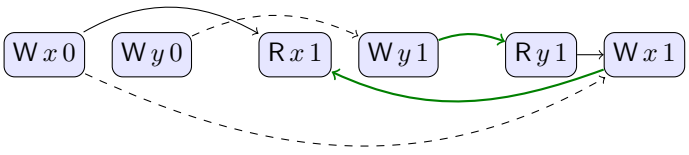
A. Causality test case 8

Test case 8 asks whether:

$$\texttt{var}\, x := 0; \texttt{var}\, y := 0; (\texttt{if}\,(x < 2)\,\{\, y := 1 \,\} \;||\; x := y)$$

may read 1 for both x and y. This behavior is allowed, since "interthread analysis could determine that x and y are always either 0 or 1." This breaks the dependency between the read of x and the write to y in the first thread, allowing the write to be moved earlier.

The semantics of TC8 includes



Where we require $(\mathsf{W}\,x\,0) < (\mathsf{R}\,x\,1)$ but not $(\mathsf{R}\,x\,1) < (\mathsf{W}\,y\,1)$. To see why this execution exists, consider the left thread with syntax sugar removed:

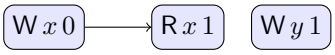
$$r := x; \texttt{if}\,(r < 2)\,\{\, y := 1 \,\}$$

$[\![\texttt{if}\,(r < 2)\,\{\, y := 1 \,\}]\!]$ includes $(r < 2 \mid \mathsf{W}\,y\,1)$. Thus, by Definition II.12, $[\![r := x; \texttt{if}\,(r < 2)\,\{\, y := 1 \,\}]\!]$ includes $(\mathsf{R}\,x\,1) \to (r < 2 \mid \mathsf{W}\,y\,1)[x/r]$ which simplifies to $(\mathsf{R}\,x\,1) \to (x < 2 \mid \mathsf{W}\,y\,1)$, which, by Definition II.8, includes:

$$\boxed{\mathsf{R}\,x\,1} \quad \boxed{x < 2 \mid \mathsf{W}\,y\,1}$$

Here we have used the [NON-ORDERING READ] clause of Definition II.8: "ψ' implies $\psi[v/x] \wedge \psi$, if a reads v from x," where $a = (\mathsf{R}\,x\,1)$, $\psi = \psi' = (x < 2)$. We can use this case since $x < 2$ implies $1 < 2 \wedge x < 2$.

Prefixing with $(\mathsf{W}\,x\,0)$ allows us to discharge the assumption $x < 2$, arriving at:



Here we have used the [ORDERING READ] clause of II.8: "ψ' implies $\psi[v/x]$, if a reads v from x and $c <' e$," where $a = (\mathsf{W}\,x\,0)$, $\psi = (x < 2)$ and $\psi' = \mathsf{true}$. As long as require $(\mathsf{W}\,x\,0) < (\mathsf{R}\,x\,1)$, we can use this case since true implies $0 < 2$.

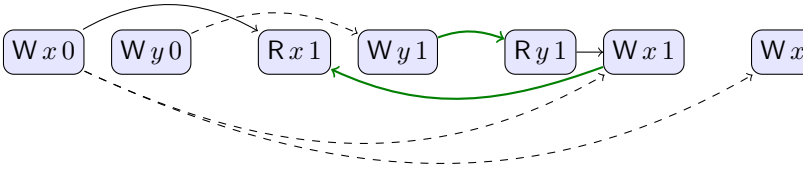
B. Causality test case 9

Test case 9 asks whether:

$$\texttt{var}\, x := 0; \texttt{var}\, y := 0; (\texttt{if}\,(x < 2)\,\{\, y := 1 \,\} \;||\; x := y \;||\; y := 2;)$$

may read 1 for both x and y. This behavior is also allowed. This is "similar to test case 8, except that x is not always 0 or 1. However, a compiler might determine that the read of x by thread 1 will never see the write by thread 3 (perhaps because thread 3 will be scheduled after thread 1)"

Reasoning as for test case 8, the semantics of test case 9 includes:



Thus, with respect to the introduction of new threads, our model appears to be more robust than the event structures semantics of Jeffrey and Riely (2016), which fails on this test case.

C. Causality test case 14

Test case 14 asks whether:

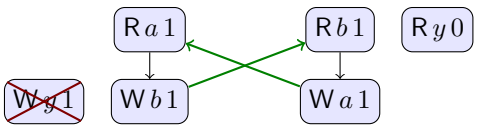
$$\texttt{var}\, a := 0; \texttt{var}\, b := 0; \texttt{var}\, y := 0; (\texttt{if}\,(a)\,\{\, b := 1 \,\}\texttt{else}\,\{\, y := 1 \,\} \;||$$

may read 1 for a and b, yet 0 for y. Here a and b are regular variables and y is volatile, which is equivalent to release/acquire in this example. This behavior is also disallowed, since "in all sequentially consistent executions, [the read of a gets 0] and the program is correctly synchronized. Since the program is correctly synchronized in all SC executions, no non-SC behaviors are allowed."

Unrolling the loop once, we have:

$$\texttt{var}\, a := 0; \texttt{var}\, b := 0; \texttt{var}\, y := 0; (\texttt{if}\,(a)\,\{\, b := 1 \,\}\texttt{else}\,\{\, y := 1 \,\} \;||$$

We argue that any execution with $(\mathsf{R}\,a\,1)$, $(\mathsf{R}\,b\,1)$, and $(\mathsf{R}\,y\,0)$ must be cyclic. The closure requirements require that $(\mathsf{W}\,a\,1) < (\mathsf{R}\,a\,1)$ and $(\mathsf{R}\,b\,1) < (\mathsf{R}\,b\,1)$. Ignoring initialization, least ordered execution that includes all of these actions is:



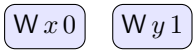
where the read of a is ordering for $(\mathsf{W}\,b\,1)$ but not $(\mathsf{W}\,y\,1)$, and the read of b is ordering for $(\mathsf{W}\,a\,1)$ but the read of y is not. $(\mathsf{W}\,y\,1)$ is crossed out, since its precondition must imply $(\neg a)[1/a]$, which is equivalent to false. To avoid order from $(\mathsf{R}\,y\,0)$ to $(\mathsf{W}\,a\,1)$, we have strengthened the predicate on

$(\mathsf{W}\,a\,1)$ from $(y \vee b)$ to $(y = 0 \wedge b = 1)$. Note that we cannot use this trick symmetrically to remove the order from $(\mathsf{R}\,b\,1)$ to $(\mathsf{W}\,a\,1)$, since $b = 1$ does not follow from the initialization of b.

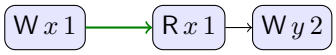
D. Thread inlining

One property one could ask of a model of shared memory is thread inlining: any execution of $[\![P\texttt{;}\ Q]\!]$ is an execution of $[\![P\ \texttt{||}\ Q]\!]$. This is *not* a goal of our model, and indeed is not satisfied, due to the different semantics of concurrent and sequential memory accesses. We demonstrate this by considering an example from the Java Memory Model Manson et al. (2005), which shows that Java does not satisfy thread inlining either.

The lack of thread inlining is related to the different dependency relations introduced by sequential and concurrent access. Recall from §III-A that the program `(x := 0; y := x+1;)` has execution:



but that `(x := 1; || y := x+1;)` has:

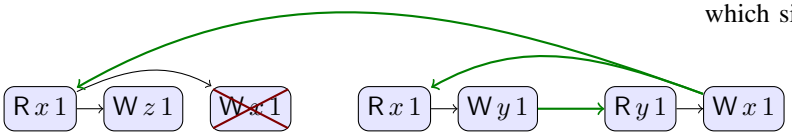


That is, in the sequential case there is no dependency from the write of x to the write of y, but in the concurrent case there is such a dependency.

This can be used to construct a counter-example to thread inlining, based on (Manson et al., 2005, Ex 11):

```
x := 0; if (x == 1) { z := 1; } else { x := 1; } || y := x; || x := y;
```

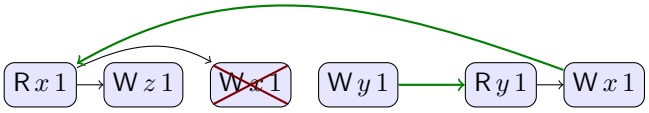
This has no execution containing $(\mathsf{W}\,z\,1)$. Any attempt to build such an execution results in a cycle:



Inlining the thread `(y := x)` gives (Manson et al., 2005, Ex 12):

```
x := 0; if (x == 1) { z := 1; } else { x := 1; } y := x; || x := y;
```

with execution:



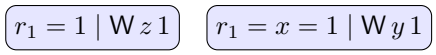
To see why this execution exists, consider the program fragment:

```
if (x == 1) { z := 1; } else { x := 1; } y := x;
```

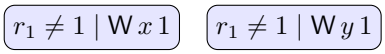
Removing the syntax sugar, this is:

```
r1 := x; if (r1 == 1) {
  z := 1; r2 := x; y := r2; skip
} else {
  x := 1; r3 := x; y := r3; skip
}
```

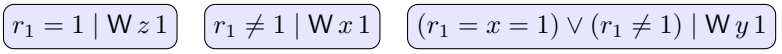
Now, $[\![z := 1;\ r_2 := x;\ y := r_2;\ \texttt{skip}]\!]$ includes pomset:



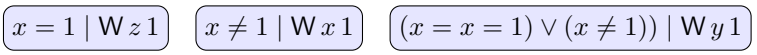
and $[\![x := 1;\ r_3 := x;\ y := r_3;\ \texttt{skip}]\!]$ includes pomset:



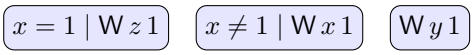
so $[\![\texttt{if}\,(r_1 = 1)\,\{z := 1;\ r_2 := x;\ y := r_2;\ \texttt{skip}\}\,\texttt{else}\,\{x := 1;\ r_3 := x;\ y := r_3;\ \texttt{skip}\}]\!]$ includes:



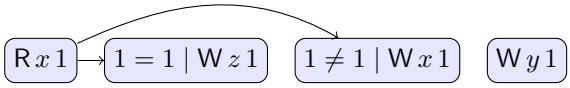
which means $[\![\texttt{if}\,(r_1 = 1)\,\{z := 1;\ r_2 := x;\ y := r_2;\ \texttt{skip}\}\,\texttt{else}\,\{x := 1;\ r_3 := x;\ y := r_3;\ \texttt{skip}\}][x/r_1]$ includes:



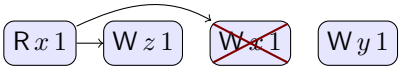
Now $(x = x = 1) \vee (x \neq 1)$ is a tautology, so this is just:



and so $[\![r_1 := x;\ \texttt{if}\,(r_1 = 1)\,\{z := 1;\ r_2 := x;\ y := r_2;\ \texttt{skip}\}\,\texttt{else}\,\{x := 1;\ r_3 := x;\ y := r_3;\ \texttt{skip}\}]\!]$ includes:



which simplifies to:



as required. The rest of the example is straightforward, and shows that our semantics agrees with the JMM in not supporting thread inlining.