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

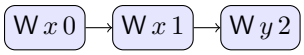
James Riely
DePaul University
jriely@cs.depaul.edu

***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 are designed to capture computer architecture, rather than micro-architecture, 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 is abstract with respect to specific micro-architectural features, such as caches and pipelines, yet is powerful enough to express known attacks such as Spectre and PRIME+ABORT. The model also allows for the prediction of new information flow attacks. We derive two such attacks, which exploit compiler optimizations, and validate these experimentally against gcc and clang.**

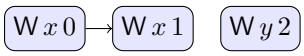
I. INTRODUCTION

This paper is about some of the lies we tell when we talk about programs.

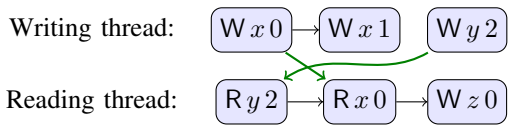
An example lie (or to be more formal, a "leaky abstraction") is the order of reads and writes in a program. We like to pretend that these happen in the order specified by the program text, for example we can think of the program $(x{:=}\,0; x{:=}\,1; y{:=}\,2)$ as having three sequentially ordered write events:



However, due to optimizations in hardware or compilers, instructions may be reordered, resulting in executions where the accesses of x and y are independent, indicating that hardware or the compiler may reorder them:



Instruction reordering optimizations are not problematic as long as they are not visible to user code, that is if programs are sequentially consistent. Unfortunately, multi-threaded programs can often observe instruction reorderings. For example running the above writing thread concurrently with an observing thread $(\mathsf{if}\,(y)\,\{\,z{:=}\,x\,\})$ can produce a sequentially inconsistent execution (where we highlight the justifying write event for each read event):

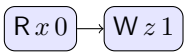


This leaky abstraction has resulted in a literature of *relaxed memory models* [**?**], which try to state precisely the memory guarantees a compiler is expected to provide, without requiring the use of expensive memory barriers to ensure sequential consistency.

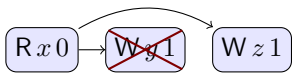
Relaxed memory is an example of how simple models can become complex. Instruction reordering was originally intended only to be visible to the microarchitecture, or inside a compiler, and was not meant to be visible to the architecture or to the behaviour of user code. Reordering optimizations are so important to the performance of modern systems that hardware and programming language designers have now accepted the complexity of relaxed memory models as the price that has to be paid for acceptable performance.

This paper looks at another leaky abstraction: *speculative evaluation*. This is similar to reordering, in that it is an optimization that was intended to be visible only to the microarchitecture, but the arrival of Spectre [22] and similar attacks [**?**] shows that not only is speculation visible, it has serious security implications.

The simplest example of speculative evaluation is branch prediction. The expected observable behavior of a conditional such as $(\mathsf{if}\,(x)\,\{\,y{:=}\,1\,\}\,\mathsf{else}\,\{\,z{:=}\,1\,\})$ is that just one branch will execute, for example:



To improve instruction throughput, hardware will often evaluate branches speculatively, and roll back any failed speculations. For example, harware might incorrectly speculate that x is nonzero, speculatively execute a write to y, but then roll it back and execute a write to z:



Speculation is intended only to be visible at the microarchtectural level, but as Spectre shows, this abstraction is leaky, and in a way that allows side-channel attacks to be mounted.

Instruction reordering and speculative evaluation are similar leaky abstractions. Both were intended originally not to be visible to user code, but both abstractions have leaked. This opens some possible areas of investigation:

- *Using ideas from relaxed memory for speculation*. There is a significant literature showing how to build models of relaxed memory, for use in validating compilers, or proving correctness of programs. Less formally, they provide programmers with a way to visualize and communicate the behavior of their systems. Inspired by these models, we give a compositional model of program execution that includes speculation (§III and §IV) and show how it can be used to model known attacks (§V) on branch prediction [22] and transactional memory [9, 11].
- *Mounting attacks against speculation on relaxed memory*. Spectre shows how a leaky abstraction allows for the construction of side-channels which bypass dynamic security checks. Since defences against buffer overflows are often dynamic checks, these attacks allow all memory in a process to be read. Inspired by these attacks, we show how to mount information flow attacks against compiler optimizations, both in theory (§VI) and in practice (§VII).

The new attacks on optimizing compilers (§VII were discovered as a consequence of building the model. A natural question is whether these attacks are an artifact of the model, or if they can be mounted in practice? We mounted the attacks on gcc and clang, where they succeeded in leaking a SECRET as long as the secret was a constant known at compile time. By itself this is not too worrying, since secrets are not normally static constants. If the same attacks could be mounted against Just-In-Time (JIT) compilers, this is potentially significant, as secrets are often known at JIT-compile time.

Fortunately, our attempts to mount the attacks against SpiderMonkey, V8 and HotSpot did not succeed. We speculate that this is because JIT compilers do not optimize as agressively, and not because of any mitigations against information flows. With the addition of shared-memory concurrency to JavaScript [13, §24.2], the attacks described in this paper might become feasible. We hope that compiler designers become as aware of information flow attacks against optimizations as their hardware designing colleagues.

Readers who wish to focus on the impact of the model can skip to §V on first reading, referring to prior sections as needed.

II. Related work

Chien [8] argues that Spectre-like attacks "extend the functional specification of the architecture to include its detailed performance" and thus "making strong assurances of application security on a computing system requires detailed performance information." This approach has been pursued in the information flow literature, by enriching language semantics with observables such as execution time and power consumption [40, 14]. This approach has also been pursued to develop model-checking techniques for Spectre-like attacks [35].

In this paper, we adopt the opposite approach, attempting to understand Spectre-like attacks as *abstractly* as possible and thus to reveal the "essence" of Spectre. We develop a novel model of speculative evaluation and show that it is sufficient to both capture known attacks and predict new attacks. Our model is defined at the *language* level, rather than the hardware level; thus, we do not model micro-architectural details such as caches or timing.

Relaxed memory models [34, 26, 6, 41, 17, 19] allow speculative execution to varying degrees. Relaxed execution is known to affect the validity of information flow analyses [27, 37]. In these models, a valid execution is defined with reference to other possible executions of the program. These models are not, however, designed for modeling Spectre-style attacks on speculation. For example all of these models will consider the straight-line code:

$$r{:=}\,x;\, s{:=}\,\mathsf{SECRET};\, a[r]{:=}\,1$$

to be the same as the conditional code:

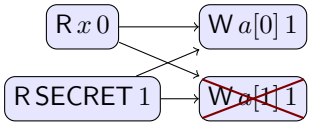
$$r{:=}\,x;\, s{:=}\,\mathsf{SECRET};$$
$$\mathsf{if}\,(r == s)\,\{\,a[s]{:=}\,1\,\}\,\mathsf{else}\,\{\,a[r]{:=}\,1\,\}$$

and indeed an optimizing compiler might choose to rewrite either of these programs to be the other.

An attacker can mount a Spectre-style attack on the conditional code, for example by setting x to be 0, flushing the cache, executing the program, then using timing effects to determine if $a[1]$ is in the cache. If it is, then SECRET must have been 1. This attack is not possible against the straight-line code, and so any model trying to capture Spectre must distinguish them.

Most definitions of non-interference will say that in both programs, there is no observable dependency of the low-security outputs (a) on the high-security inputs (SECRET) and so both programs are safe. The only existing models of non-interference which capture this information flow are ones such as [40] which model micro-architectural features such as caching and timing.

In our model, the straight-line and conditional programs are not equated, since the conditional code has the execution:



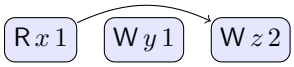
which is not matched in the straight-line code. Indeed, from an information-flow perspective, this refined treatment of dependencies in conditionals identifies a novel distinguishing feature of our model, namely that the traditional conditional is a self-composition operator in the sense of [3].

Static analyses such as the Smith-Volpano type system [33] will reject the conditional program, due to $a[s]{:=}\,1$, in which a low-security assignment depends on a high-security variable. We show how to circumvent such analyses in §V-A.

III. Model

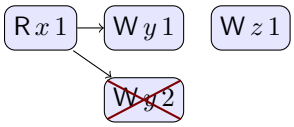
Our model is based on *partially ordered multisets* [15, 31] ("pomsets"), whose labels are given by read and write actions. These can be visualized as a graph where the edges indicate

dependencies, for example $(r{:=}\,x; y{:=}\,1; z{:=}\,r+1)$ has an execution modeled by the pomset:



The edge from $(\mathsf{R}\,x\,1)$ to $(\mathsf{W}\,z\,2)$ indicates a data dependency. Since there is no dependency between $(\mathsf{W}\,y\,1)$ and $(\mathsf{W}\,z\,2)$, the write actions may take place in either order. Such reorderings may arise in hardware (for example, caching) or in the compiler (for example, instruction reordering).

The novel aspect of the model is that events have *preconditions* which may be false. These are used in giving the semantics of conditionals and transactions, modeling failed branch prediction and aborted transactions. For example the program $(\mathsf{if}\,(x)\,\{\,y{:=}\,1; z{:=}\,1\,\}\,\mathsf{else}\,\{\,y{:=}\,2; z{:=}\,1\,\})$ 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, modeling an unsuccessful speculation. Since the $(\mathsf{W}\,z\,1)$ action is performed on both branches of the conditional, there is no control dependency from $(\mathsf{R}\,x\,1)$.

We give the semantics of a program as a set 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:

$$M = v \mid \mathsf{W}\,x\,v$$

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

The semantics is defined compositionally. As an example, we show how to construct one of the pomsets in $[\![r{:=}\,y; x{:=}\,r+1]\!]$. First, $[\![x{:=}\,r+1]\!]$ contains the pomset:

$$r = 1 \mid \mathsf{W}\,x\,2$$

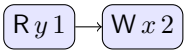
Next, we perform the substitution of r with 1 in every precondition, to get that $[\![x{:=}\,r+1]\!][1/r]$ contains the pomset:

$$1 = 1 \mid \mathsf{W}\,x\,2$$

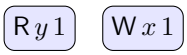
and since $(1 = 1)$ is a tautology, we elide it:

$$\mathsf{W}\,x\,2$$

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

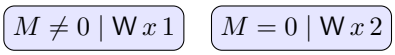


There is an ordering $(\mathsf{R}\,y\,1) < (\mathsf{W}\,x\,2)$ (represented pictorially as an arrow) because the precondition $(r = 1)$ depends on r. If the precondition was independent of r then there would be no ordering, for example $[\![r{:=}\,y; x{:=}\,r+1-r]\!]$ contains the pomset:



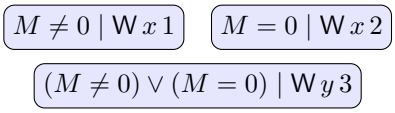
since the precondition $(r + 1 - r = 1)$ is independent of r.

The main novelty of our semantics is the use of preconditions, which allow us to provide an unusual model of conditionals. In most models, an execution of $[\![\mathsf{if}\,(M)\,\{\,C\,\}\,\mathsf{else}\,\{\,D\,\}]\!]$ would either be given by an execution from $[\![C]\!]$ or from $[\![D]\!]$, but not both. In our semantics, a pomset in $[\![\mathsf{if}\,(M)\,\{\,C\,\}\,\mathsf{else}\,\{\,D\,\}]\!]$ may include both a pomset from $[\![C]\!]$ *and* a pomset from $[\![D]\!]$. For example, $[\![\mathsf{if}\,(M)\,\{\,x{:=}\,1\,\}\,\mathsf{else}\,\{\,x{:=}\,2\,\}]\!]$ contains:

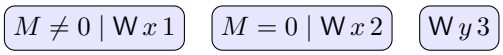


that is we have behavior from both branches of execution.

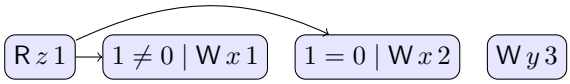
Moreover, two events representing the same action on both sides of a conditional can be merged, producing a single event. The precondition of the merged event is the disjunction of the preconditions of the original events. For example $[\![\mathsf{if}\,(M)\,\{\,x{:=}\,1; y{:=}\,3\,\}\,\mathsf{else}\,\{\,x{:=}\,2; y{:=}\,3\,\}]\!]$ contains:



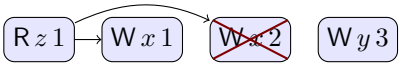
and since $(M \neq 0) \vee (M = 0)$ is a tautology, this is:



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



and we visualize unsatisfiable preconditions as crossed out:



Note that this semantics captures control dependencies such as $(\mathsf{R}\,z\,1) < (\mathsf{W}\,x\,1)$, independencies such as $(\mathsf{R}\,z\,1) \not< (\mathsf{W}\,y\,3)$, and failed speculations such as the crossed out $(\mathsf{W}\,x\,2)$.

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

- *actions*, which may read or write memory locations, and
- *preconditions*, which are closed under substitution.

In rest of this section we make data models precise and define pomsets. In the next section we give the semantics of a simple imperative language as sets of pomsets.

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 Rd and $\mathsf{Wr} : \mathcal{A} \to (\mathcal{X} \times \mathcal{V})$.

We shall say a *reads* v *from* x whenever $\mathsf{Rd}(a) = (x, v)$, and a *writes* v *to* x whenever $\mathsf{Wr}(a) = (x, v)$. 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.

B. 3-valued pomsets

Recall the definition of a pomset from [15]:

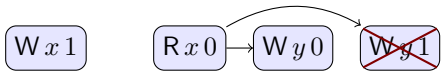
Definition III.1. A *pomset* $(E, \leq, \lambda)$ with alphabet Σ is a partial order $(E, \leq)$ together with $\lambda : E \to \Sigma$.

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:

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

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 [2]. A preliminary definition (which, as we shall see, needs to be strengthened) is:

- $d < e$,
- e implies d,
- 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 Appendix B. This is a problem in that it means the preliminary model violates *scope extrusion* [28], in that we can find programs C and D such that $[\![(\mathsf{var}\, x; (C \,||\, D))]\!]$ is not the same as $[\![(\mathsf{var}\, x; C) \,||\, D]\!]$, even if D does not mention x.

There are a number of ways this can be addressed; for example, in models such as [4] 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 III.2. A *3-valued pomset* $(E, \leq, \not<, \lambda)$ is a *pomset* $(E, \leq, \lambda)$ 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 $c \not< e$.

Structures similar to 3-valued pomsets have come up in many guises, for example rough sets [29] or ultrametrics over $\{0, 1/2, 1\}$. They correspond to axioms A1–A3 of Lamport's *system executions* [24]. They are the notion of pomset given by interpreting $d \leq e$ in a 3-valued logic [36].

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 one execution of $(x{:=}\,0; x{:=}\,1) \,||\, (x{:=}\,x + 1)$ is:

We strengthen the definition of reads-from to require not just that no 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 III.3. In a 3-valued pomset, e *can read* x *from* d whenever:

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

One of the requirements of closed programs is that every read event reads from a write event.

In the remainder of the paper, we drop the prefix "3-valued", referring to 3-valued pomsets simply as *pomsets*.

IV. Semantics of programs

In Figure 1, we give the semantics of a simple shared-memory concurrent language as sets of pomsets. Each pomset $P \in [\![C]\!]$ represents a single execution of C. We do not expect $[\![C]\!]$ to be prefixed closed; thus, one may view each $P \in [\![C]\!]$ as a *completed* execution. However, the sets of pomsets given by our semantics *are* closed with respect to augmentation, which may create additional order and strengthening preconditions:

Definition IV.1. P' is an augmentation of P if $E' = E$, $e \leq d$ implies $e \leq' d$, $e \not< d$ implies $e \not<' d$, and if $\lambda(e) = (\psi \mid b)$ then $\lambda'(e) = (\psi' \mid b)$ where ψ' implies ψ.

We give the semantics using combinators over sets of pomsets, defined in Appendix A. Using $\mathcal{P}$ to range over sets of pomsets, these are:

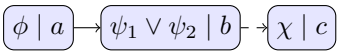
- *restriction* $\nu x\,.\mathcal{P}$, which filters $\mathcal{P}$ to include only pomsets where every event e that reads from x *can read* from some d, following Definition III.3, and where no precondition can depend on x,
- *guarding* $\phi \rhd \mathcal{P}$, which filters $\mathcal{P}$, keeping pomsets whose events have preconditions that imply ϕ,
- *substitution* $\mathcal{P}[M/x]$, which replaces x with M in every precondition of $\mathcal{P}$,
- *composition* $\mathcal{P}_1 \parallel \mathcal{P}_2$, which unions pomsets from $\mathcal{P}_1$ and $\mathcal{P}_2$, allowing events to be merged, and
- *prefixing* $a \to \mathcal{P}$, which adds an event with action a to pomsets in $\mathcal{P}$, ordering a before any e whose predicate depends on the value read by a.

These operations are similar to those from models of concurrency such as [7], but adapted here to the setting of speculative evaluation.

Restriction and guarding filter the set of pomsets; we have $(\nu x\,.\,\mathcal{P}) \subseteq \mathcal{P}$ and $(\phi \rhd \mathcal{P}) \subseteq \mathcal{P}$. Substitution updates the preconditions in a pomset, thus we expect the number of pomsets to be unchanged; in addition, the number of events in each of the pomsets is unchanged. The most interesting operators are composition and prefixing, which create larger pomsets from smaller ones.

Composition is used in giving the semantics for conditionals and concurrency. $\mathcal{P}_1 \parallel \mathcal{P}_2$ 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:

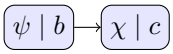
then $\mathcal{P}_1 \parallel \mathcal{P}_2$ contains:



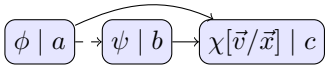
Prefixing is used in giving the semantics of reads and writes. $a \to \mathcal{P}$ adds a new event c with action a to each pomset in $\mathcal{P}$. As in the definition of parallel composition, the definition allows the new event to overlap with events in $\mathcal{P}$ as long as they agree on the action.

If c writes to a location that is also written by e in $\mathcal{P}$, then prefixing introduces weak order between them: $c \not< e$. This ensures that these writes cannot be given the reverse order in an augmentation.

If c reads from a location that occurs in the predicate of e, then prefixing introduces order from c to any e whose predicate depends on x. 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:

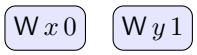


In the remainder of this section, we give examples to explain the semantics, concentrating on reads and conditionals. Security-relevant examples begin in §V.

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 the semantics of $[\![r{:=}x; C]\!]$, given in Figure 1, these are found on left and right sides of the union operation. In this section, we discuss the sequential semantics, $[\![C]\!][x/r]$, leaving the concurrent semantics to §IV-B.

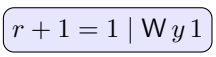
Consider the program $(x{:=}0; y{:=}x+1)$. One execution of this program is where the write to y uses the sequential value of x, which is 0:



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

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

which contains the pomset:



expressing that this program can write 1 to y, as long as the precondition $(r+1=1)$ is satisfied. Now $[\![r{:=}x; y{:=}r+1; \mathsf{skip}]\!]$ has two cases, the sequential case (which does not

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

Fig. 1. Semantics of a concurrent shared-memory language

introduce a read action) and the concurrent case (which does). For the moment, we are interested in the sequential case:

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

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 $[\![x{:=}\,0; r{:=}\,x; y{:=}\,r+1; \mathsf{skip}]\!]$ includes:

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

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:

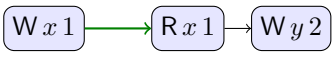
$$\boxed{\mathsf{W}\,x\,0} \quad \boxed{\mathsf{W}\,y\,1}$$

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 [16], 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:



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

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

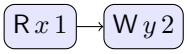
which contains the pomset:

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

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

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

which contains the pomset:

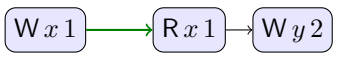


Note that $(\mathsf{R}\,x\,1)$ reads 1 from x, and while $(x+1=2)[1/x]$ is a tautology, $(x+1=2)$ is not, and so there is a dependency $(\mathsf{R}\,x\,1) < (\mathsf{W}\,y\,2)$.

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

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

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



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

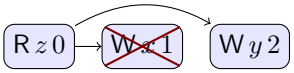
This example demonstrates how read and write events capture the concurrent semantics of memory. In an execution containing an event with label $(\mathsf{R}\,x\,v)$, if the execution is x-closed, then there must be an event it reads from, for example one labelled $(\mathsf{W}\,x\,v)$.

C. Control dependencies

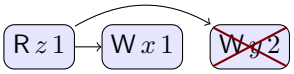
Conditionals introduce control dependencies, for example consider the program:

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

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 §V-A 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 $[\![x{:=}\,1; \mathsf{skip}]\!]$, which contains any pomset of the form:

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

in particular it contains:

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

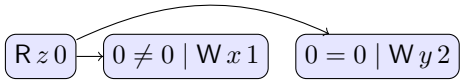
Similarly $[\![y{:=}\,2; \mathsf{skip}]\!]$ contains:

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

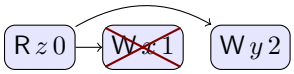
and so $[\![\mathsf{if}\,(r)\,\{\,x{:=}\,1; \mathsf{skip}\,\}\,\mathsf{else}\,\{\,y{:=}\,2; \mathsf{skip}\,\}]\!]$ contains:

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

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.

D. Control independencies

In most models of control dependencies, the dependency relation is syntactic, based on whether the action occurs 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 §IV-C:

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

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 $[\![\mathsf{if}\,(r)\,\{\,x{:=}\,1; \mathsf{skip}\,\}\,\mathsf{else}\,\{\,x{:=}\,1; \mathsf{skip}\,\}]\!]$:

$$\mathcal{P}_1 \parallel \mathcal{P}_2 \quad \text{where} \quad \begin{array}{l} \mathcal{P}_1 = (r \neq 0) \rhd [\![x{:=}\,1; \mathsf{skip}]\!] \\ \mathcal{P}_2 = (r = 0) \rhd [\![x{:=}\,1; \mathsf{skip}]\!] \end{array}$$

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:

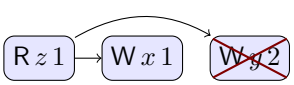
$$\boxed{\mathsf{R}\,z\,0} \quad \boxed{\mathsf{W}\,x\,1}$$

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 $(\mathsf{if}\,(z)\,\{\,x{:=}\,1\,\}\,\mathsf{else}\,\{\,x{:=}\,1\,\})$ 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 $(\mathsf{if}\,(z)\,\{\,x{:=}\,1\,\}\,\mathsf{else}\,\{\,y{:=}\,2\,\})$ 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 §VI-A 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.

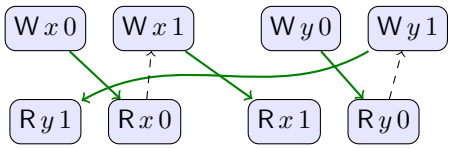
E. Relaxed memory

In §VI-A 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.

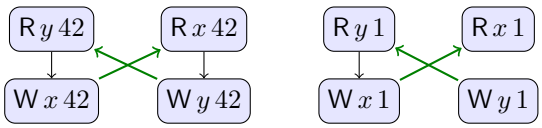
Our model includes concurrent memory accesses, which can introduce concurrent reads-from. Since we are allowing events to be partially ordered, this gives a simple model of relaxed memory. For example an independent read independent write (IRIW) example is:

$$x{:=}\,0; x{:=}\,x+1 \,||\, y{:=}\,0; y{:=}\,y+1 \\ ||\, r_1{:=}\,x; r_2{:=}\,y \,||\, s_1{:=}\,y; s_2{:=}\,x$$

which includes the execution:



This model does not introduce thin-air reads (TAR). For example the TAR pit $(x{:=}\,y \,||\, y{:=}\,x)$ 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 $(x{:=}\,y \,||\, r{:=}\,x; y{:=}\,r+1-r)$ 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.

Pugh [32] developed a set of twenty causality test cases in the process of revising the Java Memory Model (JMM) [26]. 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. [5, §4] and all but one in Ševčík [38, §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. [26, §8], and thus fails to validate thread inlining. In Appendix D, we discuss three of the causality test cases and the thread inlining example from [26].

V. Attacks on speculative execution

In this section, we show how known attacks on speculative execution can be modeled. In §V-A, we discuss Spectre In §V-B, we describe *speculation barriers* for defense against Spectre. In §V-C, we discuss attacks on transactions.

In each attack, there is a high-security variable SECRET, and the goal of the attacker is to learn one bit of information from SECRET. The Spectre and PRIME+ABORT attacks exploit optimizations in hardware, and so can be mounted against a dynamic SECRET.

A. Spectre

We give a simplified model of Spectre attacks, ignoring the details of cache 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:

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

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:

$$\begin{array}{l}\mathsf{if}\,(\mathsf{canRead}(\mathsf{SECRET}))\,\{\,\text{code allowed to read SECRET}\,\} \\ \mathsf{else}\,\{\,\text{code not allowed to read SECRET}\,\}\end{array}$$

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.

Attacks may realize the abstract notions in various ways. For example, in variant 1 of Spectre, the dynamic security check is implemented as an array bounds check.

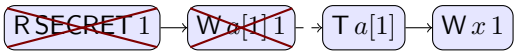
We model the touched test by introducing a new action $(\mathsf{T}\,x)$, and defining:

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

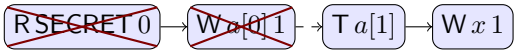
Implementations of touched use cache timing, but their success can be modeled without needing to be precise about such microarchitectural details:

- 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:



but (assuming a successful implementation of touched) *not*:



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 the way in which Spectre-like attacks bypass dynamic security checks, without giving a treatment of microarchitecture.

B. Speculation barriers

The ability to model Spectre is useful, but really we would like to model defences against such attacks, and provide some confidence in the correctness of the defence. One such defence, which fits naturally in our model is *speculation barriers*, which is code that cannot be speculatively executed. For example, we could introduce such a barrier, and require that a barrier is introduced on each security check:

$$\mathsf{if}\,(\mathsf{canRead}(\mathsf{SECRET}))\,\{\,\mathsf{barrier};\,\cdots\,\}\,\mathsf{else}\,\{\,\cdots\,\}$$

To model barriers, we introduce a new action SB and define:

$$[\![\mathsf{barrier}; C]\!] = \{\emptyset\} \cup ((\mathsf{SB}) \to [\![C]\!])$$

Implementations of barrier make use of hardware barriers which halt speculative execution until all instructions up to the barrier have been retired. Such barriers are successful when:

- if $\lambda(e) = (\phi \mid \mathsf{SB})$ then ϕ is satisfiable.

For example, a successful implementation of barriers disallows the execution of Spectre:

One might expect that this is a successful (albeit expensive) defence against Spectre, but it is not, unless the compiler is aware that barrier cannot be lifted out of a conditional. An unaware compiler might perform common subexpression elimination on barriers, allowing the attacker to introduce a barrier to fool a compiler into optimizing the safe:

$$\mathsf{if}\,(\mathsf{canRead}(\mathsf{SECRET}))\,\{\,\mathsf{barrier};\,\cdots\,\}\,\mathsf{else}\,\{\,\mathsf{barrier};\,\cdots\,\}$$

into the unsafe:

$$\mathsf{barrier};\mathsf{if}\,(\mathsf{canRead}(\mathsf{SECRET}))\,\{\,\cdots\,\}\,\mathsf{else}\,\{\,\cdots\,\}$$

To model the requirement that barriers are not moved past conditionals, we make them *unmergeable*: SB events on different arms of a conditional cannot be merged. With this requirement, we can show that barriers act as a defence against Spectre by first showing that $[\![D]\!]$ has the same successful executions as:

$$[\![\mathsf{if}\,(\mathsf{canRead}(\mathsf{SECRET}))\mathsf{barrier}; C\,\}\,\mathsf{else}\,\{\,D\,\}]\!]$$

and then showing that the semantics which only looks at successful executions is *compositional*: if $[\![D]\!]$ has the same successful executions as $[\![D']\!]$ then $[\![C[D]]\!]$ has the same successful executions as $[\![C[D']]\!]$ for any "program with a hole" $C[\bullet]$. Compositional reasoning is what fails when SB is mergeable, as shown by the attack against a compiler which blindly performs common subexpression elimination.

To realize a speculation barrier in microarchitecture, it is likely sufficient for the barrier to stop any further speculation until the barrier is known to succeed. There is experimental evidence that IntelâĂŹs mfence instruction has the effect of a speculation barrier in some contexts [35, §VII-*D*].

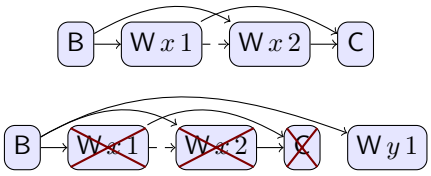
C. Transactions

We present a model of transactional memory [25] that is sufficient to capture PRIME+ABORT attacks [11]. We make several simplifying assumptions: transactions are serializable, strongly isolated, and only abort due to cache conflicts.

The action $(\mathsf{B}\,v)$ represents the begin of a transaction with id v, and $(\mathsf{C}\,v)$ 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):

$$\begin{aligned}&[\![\mathsf{begin}\,v; E; \mathsf{onabort}\,v\,\{\,D\,\}]\!]\\ &\quad = (\mathsf{B}\,v) \to \big([\![E]\!] \cup \big((\mathsf{false} \rhd [\![E]\!]) \parallel [\![D]\!]\big)\big)\\ &[\![\mathsf{commit}\,v; D]\!]\\ &\quad = (\mathsf{C}\,v) \to [\![D]\!]\end{aligned}$$

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 $(\mathsf{begin}; x{:=}1; x{:=}2; \mathsf{commit}; \mathsf{onabort}\,\{y{:=}1\,\})$ are:



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

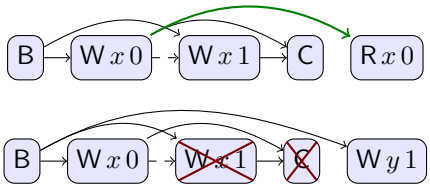
Definition V.1. We say that event c *matches* b if $\lambda(c) = (\mathsf{C}\,v)$ and $\lambda(b) = (\mathsf{B}\,v)$. We say that begin event b *begins* e if $b \le e$ and there is no intervening matching commit; in this case e *belongs to* b. We say that commit event c *commits* e if $e \le c$ and there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

Definition V.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 [12]. 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 §V-A.

Definition V.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 the same location.

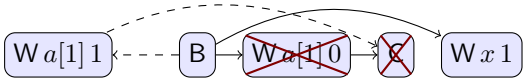
The attack requires an honest agent whose access pattern depends upon a secret. Such an honest agent is:

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

Then the attacker program

$$\mathsf{begin}; a[1] := 0; r := \mathsf{commit}; \mathsf{onabort}\,\{x := 1\}$$

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.

The attacker code here must have write access to the high security variable a. Such a "write up" is allowed by secrecy analyses such as the Smith-Volpano type system [33], which is meant to guarantee noninterference.

If we require that the attacker and honest agent access disjoint locations in memory, then we must include a bit of microarchitecture to model the attack. Suppose that the set of locations $\mathcal{X}$ is partitioned into *cache sets* and update Definition V.3 so that the commit event aborts due to memory conflict if e and d touch locations *in the same cache set*.

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

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

The attack relies on discovery of some y which belongs to the cache-set of $a[1]$. Then the attack can be written as:

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

As before, if the attacker knows that commits only abort due to memory conflicts, then there is an information flow, since the memory conflict only happens when the SECRET is 1.

This style of attack can be thwarted by requiring that the honest agent and attack code access disjoint cache sets. This approach is pursued in [21].

Another defense is require a speculation barrier at the beginning of each transaction. This would have the effect, however, of undermining any optimistic execution strategy for transactions: the transaction would only be able to begin when it is known that its commit will succeed.

VI. ATTACKS AGAINST COMPILER OPTIMIZATIONS

In this section, we model two attacks on compiler optimizations. The first attack exploits reordering allowed by relaxed memory models (§VI-A). The second exploit dead store elimination (§VI-B).

As in the previous section, the goal of the attacker is to learn one bit of information from the high-security SECRET. The attacks on compiler optimizations require the SECRET to be known to the compiler, for example a static SECRET or a JIT compiler.

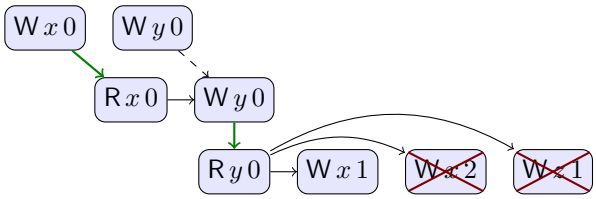
To defend against these attacks it is sufficient to require a traditional memory fence after each security check: compilers do not reorder instructions over fences.

A. Relaxed memory orders

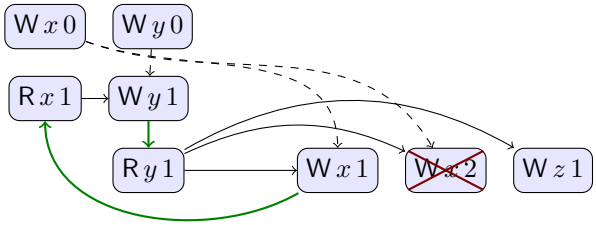
Consider an attacker program, again using dynamic security checks to try to learn a SECRET. Whereas SPECTRE uses hardware capabilities, which have to be modeled by adding extra capabilities to the language, this new attacker works by exploiting relaxed memory which can result in unexpected information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

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

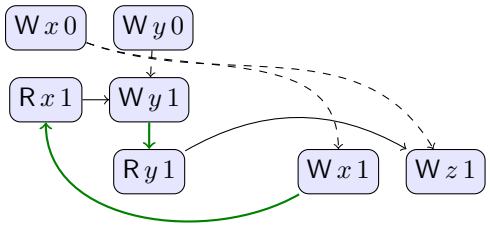
In the case where SECRET is 2, this has many executions, one of which is:



but there are no executions which exhibit $(\mathsf{W}\, z\, 1)$, since any attempt to do so produces a cycle, since the value written to x has a control dependency on the value read from y:



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:

$$\begin{array}{l}\mathsf{if}\,(y=0)\,\{\,x{:=}\,1\,\}\\ \mathsf{else\ if}\,(\mathsf{canRead}(\mathsf{SECRET}))\,\{\,x{:=}\,\mathsf{SECRET}\,\}\\ \mathsf{else}\,\{\,x{:=}\,1;\,z{:=}\,1\,\}\end{array}$$

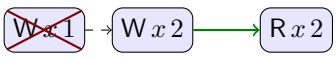
In the case where SECRET is a constant 1, the compiler can inline it and lift the assignment to x out of the if statement:

$$\begin{array}{l}x{:=}\,1;\,\mathsf{if}\,(y=0)\,\{\ \}\\ \mathsf{else\ if}\,(\mathsf{canRead}(\mathsf{SECRET}))\,\{\ \}\\ \mathsf{else}\,\{\,z{:=}\,1\,\}\end{array}$$

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

B. 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:



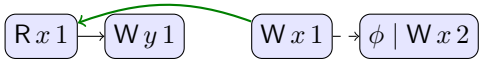
Recall that for any satisfiable e, if e reads x from y then d is satisfiable. This means that, although we can eliminate $(\mathsf{W}\,x\,1)$ we cannot eliminate $(\mathsf{W}\,x\,2)$.

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 a write event d is eliminable if there is a tautology $e \not\leftarrow d$ which writes to the same location. A model of dead store elimination is one where, in every pomset, every eliminable event is unsatisfiable. This model includes the example above.

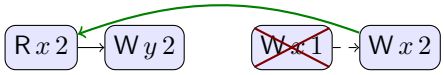
Note that if dead store elimination is *always* performed, then there is an information flow attack similar to the one in §VI-A. Consider the program:

$$\begin{array}{l}y{:=}\,x\,||\,x{:=}\,1;\\ \quad\mathsf{if}\,(\mathsf{canRead}(\mathsf{SECRET}))\,\{\,\mathsf{if}\,(\mathsf{SECRET})\,\{\,x{:=}\,2\,\}\,\}\\ \quad\mathsf{else}\,\{\,x{:=}\,2\,\}\end{array}$$

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



where ϕ is $(\neg\mathsf{canRead}(\mathsf{SECRET}))$, 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 §VII.

VII. 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. §VI-A and §VI-B 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 [23, 39].

In this section we present implementations of the attacks described in §VI-A and §VI-B, 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

As explained in Section V, our model expresses a variety of attacks with differing attacker models. The Spectre (§V-A) and Prime+Abort (§V-C) attacks exploit optimizations in hardware, and so can be mounted against a dynamic SECRET. Our model captures this appropriately. In contrast, the attacks from §VI-A and §VI-B leverage compiler optimizations and require the SECRETto be known to the compiler, for example a static SECRETor a JIT compiler. As our experimental section is devoted to these latter (novel) attacks, we discuss the attacker model for these attacks in more detail.

In the attacker model for the compiler-optimization attacks, 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 §VI-A 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 §VI-A 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:

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

Figure 2 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.

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. 2. 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.

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

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

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

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. 3. 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.

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 3 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 §VI-B based on dead store elimination. We show that in our attacker model (given in §VII-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 §VI-B, 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 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 §VII-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 §VII-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.

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. 4. 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 3.

Stall amount	10	100	500
Redundancy 1	2.54 million 98.15%	1.54 million 99.996%	584 thousand 99.998%
Redundancy 2	1.24 million 99.73%	774 thousand 100.0%	295 thousand 100.0%
Redundancy 3	841 thousand 99.94%	521 thousand 100.0%	201 thousand 100.0%
Redundancy 4	620 thousand 99.992%	387 thousand 100.0%	145 thousand 100.0%

Fig. 5. 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.

Performance results for the dead store elimination attack against clang are given in Figure 4, and against gcc are given in Figure 5. Both attacks are faster than the load-store-reordering attack from §VII-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.

VIII. 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 [22, 11], 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.

We have tried where possible to abstract away from the micro-architectural details that enable attackers to exploit speculation, while still trying to capture the "essence" of Spectre. There are trade-offs with any such abstraction, as higher-level abstractions make program behavior easier to understand and reason about, but at the cost of ignoring potential attacks. One software developer's useful abstraction is another's ignoring the difficult issues.

As a concrete instance, one feature of Spectre we have glossed over is the ability of the attacker to influence speculation, for example by training the branch predictor or influencing the contents of caches. We expect that such attacker influence could be modelled using a mechanism similar to the speculation barriers of §V-B, but under the control of the attacker rather than the honest agents.

The paper's primary focus is not weak memory, and the model of relaxed memory used in this paper is deliberately simplified, compared for example to C11 [6, 4]. Nonetheless, we believe that the model developed in the paper has promise as a semantics for relaxed memory. Our model appears to be the first in the literature that both validates all of the JMM causality test cases and also forbids thin air behavior; the most prominent existing models are either too permissive [26, 17, 19] or too conservative [18]. In separate work, we are exploring the usual properties of weak memory, such as comparisons with sequentially consistent models, optimization soundness, or compilation soundness. While our model of transactions shows the flexibility of our model, in this future work, we will include known features of hardware, including locks, fences, and read-modify-write instructions. This development is not core to the basic findings of this paper.

The design space for transactions is very rich [12]. 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 [31] can be extended to 3-valued pomsets.

One interesting feature of this model is that (in the language of [30]) 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 [17, 20] in which "alternative futures" are explored, in a style reminiscent of Abramsky's bisimulation as a testing equivalence [1]. Models of information flow are similar, in that they require comparing different runs to test for the presence of dependencies [10]. 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 [3], the humble conditional suffices to construct a composition operator to detect information flow in the presence of speculation.

References

[1] Samson Abramsky. Observation equivalence as a testing equivalence. *Theoretical Computer Science*, 53(2):225 – 241, 1987. ISSN 0304-3975. doi: https://doi.org/10.1016/0304-3975(87)90065-X. URL http://www.sciencedirect.com/science/article/pii/030439758790065X.

[2] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.*, 36(2):7:1–7:74, July 2014. ISSN 0164-0925. doi: 10.1145/2627752. URL http://doi.acm.org/10.1145/2627752.

[3] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. In *Proceedings of the 17th IEEE Workshop on Computer Security Foundations*, CSFW '04, pages 100–114, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2169-X. doi: 10.1109/CSFW.2004.17. URL https://doi.org/10.1109/CSFW.2004.17.

[4] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing C++ concurrency. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 55–66, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0490-0. doi: 10.1145/1926385.1926394. URL http://doi.acm.org/10.1145/1926385.1926394.

[5] Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. The problem of programming language concurrency semantics. In *Proc. European Symp. on Programming*, pages 283–307, 2015.

[6] Hans-J. Boehm and Sarita V. Adve. Foundations of the c++ concurrency memory model. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 68–78, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-860-2. doi: 10.1145/1375581.1375591. URL http://doi.acm.org/10.1145/1375581.1375591.

[7] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31(3):560–599, June 1984. ISSN 0004-5411. doi: 10.1145/828.833. URL http://doi.acm.org/10.1145/828.833.

[8] Andrew A. Chien. Computer architecture: Disruption from above. *Commun. ACM*, 61(9):5–5, August 2018. ISSN 0001-0782. doi: 10.1145/3243136. URL http://doi.acm.org/10.1145/3243136.

[9] Nathan Chong, Tyler Sorensen, and John Wickerson. 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*, pages 211–225, 2018. doi: 10.1145/3192366.3192373. URL http://doi.acm.org/10.1145/3192366.3192373.

[10] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *J. Comput. Secur.*, 18(6):1157–1210, September 2010. ISSN 0926-227X. URL http://dl.acm.org/citation.cfm?id=1891823.1891830.

[11] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean M. Tullsen. Prime+abort: A timer-free high-precision L3 cache attack using intel TSX. In Engin Kirda and Thomas Ristenpart, editors, *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017.*, pages 51–67. USENIX Association, 2017. URL https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/disselkoen.

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

[13] ECMA TC39. ECMAScript 2017 language specification. https:

//www.ecma-international.org/ecma-262/8.0/, 2017.

[14] Andrew Ferraiuolo, Mark Zhao, Andrew C. Myers, and G. Edward Suh. Hyperflow: A processor architecture for non-malleable, timing-safe information-flow security. In *25th ACM Conf. on Computer and Communications Security (CCS)*, October 2018. URL http://www.cs.cornell.edu/andru/papers/hyperflow.

[15] Jay L. Gischer. The equational theory of pomsets. *Theoretical Computer Science*, 61(2):199–224, 1988. ISSN 0304-3975. doi: 10.1016/0304-3975(88)90124-7. URL http://www.sciencedirect.com/science/article/pii/0304397588901247.

[16] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969. ISSN 0001-0782. doi: 10.1145/363235.363259. URL http://doi.acm.org/10.1145/363235.363259.

[17] Radha Jagadeesan, Corin Pitcher, and James Riely. Generative operational semantics for relaxed memory models. In *Proceedings of the 19th European Conference on Programming Languages and Systems*, ESOP'10, pages 307–326, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-11956-5, 978-3-642-11956-9. doi: 10.1007/978-3-642-11957-6_17. URL http://dx.doi.org/10.1007/978-3-642-11957-6_17.

[18] A. Jeffrey and J. Riely. On thin air reads towards an event structures model of relaxed memory. In M. Grohe, E. Koskinen, and N. Shankar, editors, *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*, pages 759–767. ACM, 2016. ISBN 978-1-4503-4391-6. doi: 10.1145/2933575.2934536. URL http://doi.acm.org/10.1145/2933575.2934536.

[19] J. Kang, C.-K. Hur, O. Lahav, V. Vafeiadis, and D. Dreyer. A promising semantics for relaxed-memory concurrency. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 175–189. ACM, 2017. ISBN 978-1-4503-4660-3. doi: 10.1145/3009837. URL http://dl.acm.org/citation.cfm?id=3009850.

[20] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. A promising semantics for relaxed-memory concurrency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 175–189, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4660-3. doi: 10.1145/3009837.3009850. URL http://doi.acm.org/10.1145/3009837.3009850.

[21] Vladimir Kiriansky, Ilia A. Lebedev, Saman P. Amarasinghe, Srinivas Devadas, and Joel Emer. DAWG: A defense against cache timing attacks in speculative execution processors. *IACR Cryptology ePrint Archive*, 2018:418, 2018. URL https://eprint.iacr.org/2018/418.

[22] 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. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[23] David Kohlbrenner and Hovav Shacham. Trusted browsers for uncertain times. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 463–480, Austin, TX, 2016. USENIX Association. ISBN 978-1-931971-32-4. URL https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/kohlbrenner.

[24] Leslie Lamport. On interprocess communication. part I: basic formalism. *Distributed Computing*, 1(2):77–85, 1986. doi: 10.1007/BF01786227. URL https://doi.org/10.1007/BF01786227.

[25] Jim Larus and Ravi Rajwar. *Transactional Memory (Synthesis Lectures on Computer Architecture)*. Morgan & Claypool Publishers, 2007. ISBN 1598291246.

[26] Jeremy Manson, William Pugh, and Sarita V. Adve. The java memory model. *SIGPLAN Not.*, 40(1):378–391, January 2005. ISSN 0362-1340. doi: 10.1145/1047659.1040336. URL http://doi.acm.org/10.1145/1047659.1040336.

[27] H. Mantel, M. Perner, and J. Sauer. Noninterference under weak memory models. In *2014 IEEE 27th Computer Security Foundations Symposium*, pages 80–94, July 2014. doi: 10.1109/CSF.2014.14.

[28] Robin Milner. *Communicating and Mobile Systems: The π-calculus*. Cambridge University Press, New York, NY, USA, 1999. ISBN 0-521-65869-1.

[29] Zdzisław Pawlak. Rough sets. *International Journal of Computer & Information Sciences*, 11(5):341–356, Oct 1982. ISSN 1573-7640. doi: 10.1007/BF01001956. URL https://doi.org/10.1007/BF01001956.

[30] Jean Pichon-Pharabod and Peter Sewell. 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, pages 622–633, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3549-2. doi: 10.1145/2837614.2837616. URL http://doi.acm.org/10.1145/2837614.2837616.

[31] Gordon Plotkin and Vaughan Pratt. Teams can see pomsets (preliminary version). In *Proceedings of the DIMACS Workshop on Partial Order Methods in Verification*, POMIV '96, pages 117–128, New York, NY, USA, 1997. AMS Press, Inc. ISBN 0-8218-0579-7. URL http://dl.acm.org/citation.cfm?id=266557.266600.

[32] W. Pugh. Causality test cases. http://www.cs.umd.edu/~pugh/java/memoryModel/CausalityTestCases.html, 2004.

[33] Geoffrey Smith and Dennis Volpano. 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, pages 355–364, New York, NY, USA, 1998. ACM. ISBN 0-89791-979-3. doi: 10.1145/268946.268975. URL http://doi.acm.org/10.1145/268946.268975.

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

[35] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. Checkmate: Automated synthesis of hardware exploits and security litmus tests. In *51st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2018, Fukuoka, Japan, October 20-24, 2018*, pages 947–960. IEEE, 2018. ISBN 978-1-5386-6240-3. doi: 10.1109/MICRO.2018.00081. URL https://doi.org/10.1109/MICRO.2018.00081.

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

[37] Jeffrey A. Vaughan and Todd Millstein. Secure information flow for concurrent programs under total store order. In *Proceedings of the 2012 IEEE 25th Computer Security Foundations Symposium*, CSF '12, pages 19–29, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4718-3. doi: 10.1109/CSF.2012.20. URL http://dx.doi.org/10.1109/CSF.2012.20.

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

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

[40] Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. Language-based control and mitigation of timing channels. *SIGPLAN Not.*, 47(6):99–110, June 2012. ISSN 0362-1340. doi: 10.1145/2345156.2254078. URL http://doi.acm.org/10.1145/

2345156.2254078.

[41] Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. In John Field and Michael Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 427–440. ACM, 2012. ISBN 978-1-4503-1083-3. doi: 10.1145/2103656.2103709. URL http://doi.acm.org/10.1145/2103656.2103709.

APPENDIX

A. Operations on sets of pomsets

Here we give the formal definitions for the operations described at the beginning of §IV.

In order to model speculation barriers in §V-B, we partition the actions into *mergeable* and *unmergeable*.

In transactional memory, begin and commit actions are memory fences: that is, they are a barrier to reordering memory accesses. To capture this (and other memory barriers), we identify sets Rel and $\mathsf{Acq} \subseteq \mathcal{A}$. For transactions, we have $(\mathsf{B}\, v) \in \mathsf{Acq}$ for begins, $(\mathsf{C}\, v) \in \mathsf{Rel}$ for commits. We say that a is a *release* if $a \in \mathsf{Rel}$ and a is an *acquire* if $a \in \mathsf{Acq}$.

Definition A.1. 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 $d \not\leq' e$,
- if $c \in E$ then c is mergeable,
- $\lambda'(c) = (\phi, a)$, and
- if $\lambda(e) = (\psi \mid b)$ then $\lambda'(e) = (\psi' \mid b)$, where:
  - $c <' e$ whenever a is an acquire or b is a release,
  - if a is an acquire then ψ is independent of every y,
  - if a and b both touch the same location and one is a write, then $c \not> e$, and
  - ψ' implies
    - $\psi[v/x]$ if a reads v from x and $c <' e$ [DEPENDENT READ]
    - $\psi[v/x]$ and ψ if a reads v from x [INDEPENDENT READ]
    - ψ otherwise [NON-READ]

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.

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.

Definition A.2. 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 $e \not<_1 d$ or $e \not<_2 d$ then $e \not<_0 d$,
- if $e \in E_1 \cap E_2$ then e is mergeable,
- 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.

Definition A.3. 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 $d \not< e$ then $d \not<' e$, and
- if $\lambda(e) = (\psi \mid a)$ then $\lambda'(e) = (\psi[M/x] \mid a)$.

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

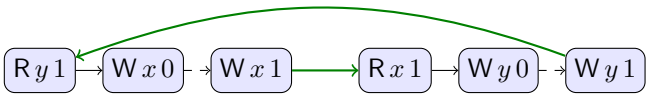
Definition A.4. Let $(\phi \rhd \mathcal{P})$ be the subset of $\mathcal{P}$ such that $P \in \mathcal{P}$ whenever:

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

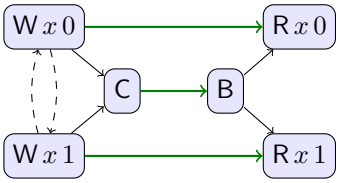
Definition A.5. A 3-valued pomset is x-closed if, for every $e \in E$:

- 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 transactions, 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 III.3, 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.

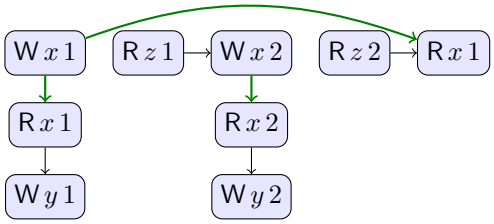
Definition A.6. A 3-valued pomset is *partially* (resp. *totally*) x-*coherent* if, when restricted to events which touch x, $\not<$ forms a partial (resp. total) order.

Definition A.7. 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.

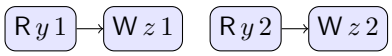
B. Blockers

Recall the preliminary definition of reads-from in §III-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.

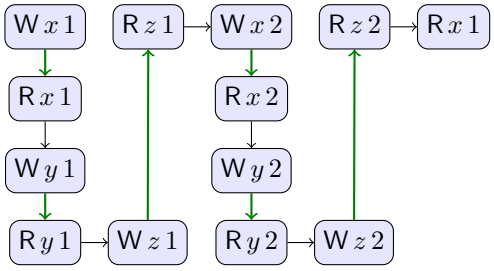
To see this, consider the program $(x{:=}\,1; y{:=}\,x \,||\, x{:=}\,z+1; y{:=}\,x \,||\, \mathsf{if}\,(z=2)\,\{\,r{:=}\,x\,\})$ with execution:



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

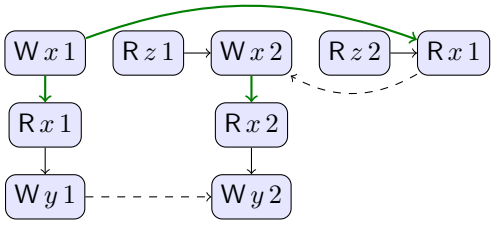


If these are placed in parallel, then a possible execution is:

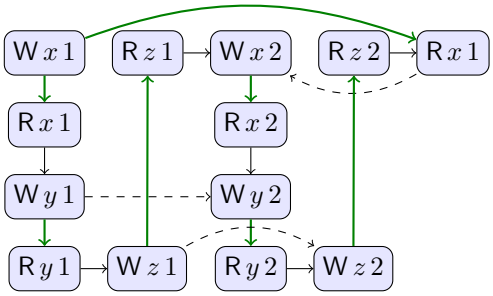


and now the $(\mathsf{W}\,x\,2)$ event is an x-blocker, so $(\mathsf{R}\,x\,1)$ cannot read from $(\mathsf{W}\,x\,1)$.

In the final definition of reads-from in §III-B we ruled out x-blockers by requiring 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.

C. Release/acquire synchronization

We can develop a simple model of release/acquire synchronization using the following actions:

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

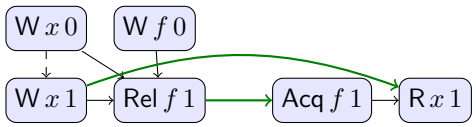
The semantics of programs with releasing write and acquiring read are similar to regular write and read, with $\mathsf{Rel}\,x\,v$ replacing $\mathsf{W}\,x\,v$ and $\mathsf{Acq}\,x\,v$ replacing $\mathsf{R}\,x\,v$:

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

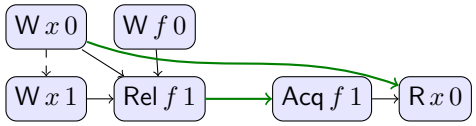
To see the need for the first constraint on prefixing, consider the program:

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

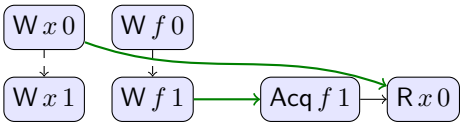
This has an execution:



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.

D. Causality test cases

Pugh [32] developed a set of twenty causality test cases in the process of revising the Java Memory Model (JMM) [26]. 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. [5, §4] and all but one in Ševčík [38, §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. [26, §8], and thus fails to validate thread inlining.

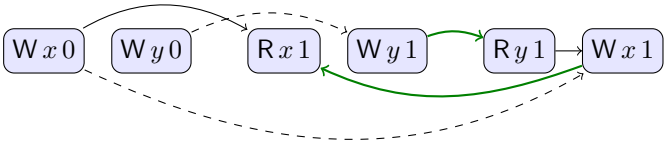
In this section, we discuss three of the causality test cases and the thread inlining from [26]. In presenting the examples, we unroll loops, correct typos and simplify the code.

1) Causality test case 8: Test case 8 asks whether:

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

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:

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

$[\![\mathsf{if}\,(r < 2)\,\{\, y{:=}\,1 \,\}]\!]$ includes $(r < 2 \mid \mathsf{W}\,y\,1)$. Thus, by Figure 1, $[\![r{:=}\,x; \mathsf{if}\,(r < 2)\,\{\, y{:=}\,1 \,\}]\!]$ 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 A.1, 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 A.1: "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:

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

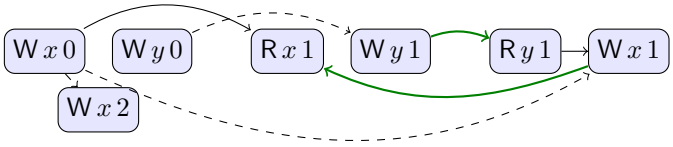
Here we have used the [ORDERING READ] clause of A.1: "ψ' implies $\psi[v/x]$, if a reads v from x and $c <' e$," where $a = (\mathsf{W}\,x\,0)$, $\psi = (x < 2)$ and $\psi' = \mathsf{true}$. As long as require $(\mathsf{W}\,x\,0) < (\mathsf{R}\,x\,1)$, we can use this case since true implies $0 < 2$.

2) Causality test case 9: Test case 9 asks whether:

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

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 [18], which fails on this test case.

3) Causality test case 14: Test case 14 asks whether:

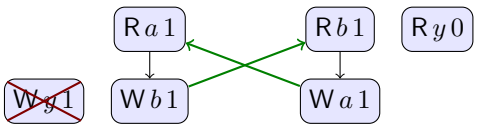
$$\begin{aligned}&\mathsf{var}\, a{:=}\,0; \mathsf{var}\, b{:=}\,0; \mathsf{var}\, y{:=}\,0;\\ &\qquad(\mathsf{if}\,(a)\,\{\, b{:=}\,1 \,\}\,\mathsf{else}\,\{\, y{:=}\,1 \,\} \,||\\ &\qquad\qquad\mathsf{while}\,(y + b == 0)\,\{\, \mathsf{skip} \,\}\; a{:=}\,1)\end{aligned}$$

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:

$$\begin{aligned}&\mathsf{var}\, a{:=}\,0; \mathsf{var}\, b{:=}\,0; \mathsf{var}\, y{:=}\,0;\\ &\qquad(\mathsf{if}\,(a)\,\{\, b{:=}\,1 \,\}\,\mathsf{else}\,\{\, y{:=}\,1 \,\} \,||\\ &\qquad\qquad\mathsf{if}\,(y \vee b)\,\{\, a{:=}\,1 \,\})\end{aligned}$$

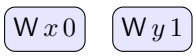
We argue that any execution with $(\mathsf{R}\,a\,1)$, $(\mathsf{R}\,b\,1)$, and $(\mathsf{R}\,y\,0)$ must be cyclic. The closure requirements require that $(\mathsf{W}\,a\,1) < (\mathsf{R}\,a\,1)$ and $(\mathsf{R}\,b\,1) < (\mathsf{R}\,b\,1)$. Ignoring initialization, least ordered execution that includes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



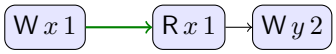
where the read of a is ordering for $(\mathsf{W}\,b\,1)$ but not $(\mathsf{W}\,y\,1)$, and the read of b is ordering for $(\mathsf{W}\,a\,1)$ but the read of y is not. $(\mathsf{W}\,y\,1)$ is crossed out, since its precondition must imply $(\neg a)[1/a]$, which is equivalent to false. To avoid order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

4) Thread inlining: One property one could ask of a model of shared memory is thread inlining: any execution of $[\![P;Q]\!]$ is an execution of $[\![P \,||\, Q]\!]$. 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 [26], 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 §IV-A that the program (x := 0; y := x+1;) has execution:



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

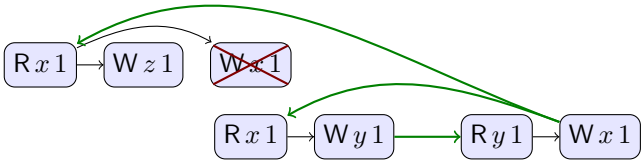


That is, in the sequential case there is no dependency from the write of x to the write of y, but in the concurrent case there is such a dependency.

This can be used to construct a counter-example to thread inlining, based on [26, Ex 11]:

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

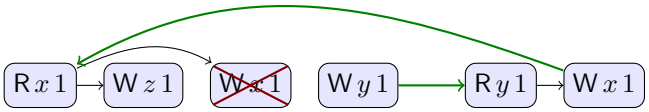
This has no execution containing $(\mathsf{W}\, z\, 1)$. Any attempt to build such an execution results in a cycle:



Inlining the thread (y := x) gives [26, Ex 12]:

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

with execution:



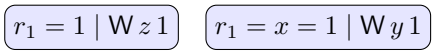
To see why this execution exists, consider the program fragment:

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

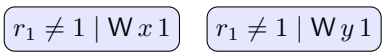
Removing the syntax sugar, this is:

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

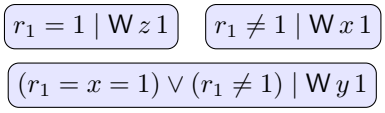
Now, $[\![z := 1; r_2 := x; y := r_2; \mathsf{skip}]\!]$ includes pomset:



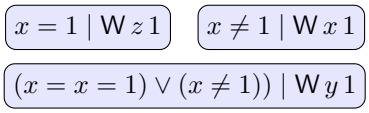
and $[\![x := 1; r_3 := x; y := r_3; \mathsf{skip}]\!]$ includes pomset:



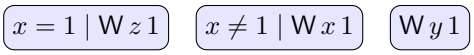
so $[\![\mathsf{if}\,(r_1 = 1)\,\{\, z := 1; r_2 := x; y := r_2; \mathsf{skip}\,\}\,\mathsf{else}\,\{\, x := 1; r_3 := x; y := r_3; \mathsf{skip}\,\}]\!]$ includes:



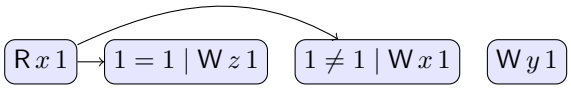
which means $[\![\mathsf{if}\,(r_1 = 1)\,\{\, z := 1; r_2 := x; y := r_2; \mathsf{skip}\,\}\,\mathsf{else}\,\{\, x := 1; r_3 := x; y := r_3; \mathsf{skip}\,\}]\!][x/r_1]$ includes:



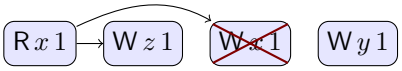
Now $(x = x = 1) \vee (x \neq 1)$ is a tautology, so this is just:



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