A model of speculative evaluation

CRAIG DISSELKOEN, University of California San Diego and Mozilla Research Internship
RADHA JAGADEESAN, DePaul University
ALAN JEFFREY, Mozilla Research
JAMES RIELY, DePaul University

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

1 INTRODUCTION

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

Information flow from high (hi) to low (lo) provides a formal foundation for end-to-end security. Informally, a program is secure if there is no observable dependency of lo-observables on hi-inputs. The precise formalization of this intutive idea has been the topic of extensive research (e.g., see [**?**] for a detailed survey till 2006), and can be generalized to account for a variety of language features and observables such as non-determinisim [**?**], concurrency [**?**], reactivity [**?**], and probability [**?**]. The static and dynamic enforcement of these definitions in general purpose languages [**?**] has also been studied extensively and has influenced language design and implementation.

A key parameter in the definitions cited above is the notion of the *observational power* of the attacker model. Whereas the classical input-output behavior is often an adequate foundation, it has long been known [**?**] that side-channels that leak information arise from other observables such as execution time and power consumption.

The focus of this paper is the development of a compositional semantic model of executions of programs to explicate side channel attacks that arise from speculation. In particular, we model conditionals such that it is possible for the behaviour of a conditional ($\texttt{if}\,(M)\,\{\,C\,\}\,\texttt{else}\,\{\,D\,\}$) to depend on the behaviour of D even when the condition M is true.

Our study addresses several sources of speculation in the concurrent execution of programs.

- Pipelined microprocessors use heuristics to predict the path of a program, for example by guessing memory dependencies, or the outcome of conditionals. Execution proceeds along

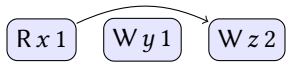
Authors' addresses: Craig Disselkoen, University of California San Diego, Mozilla Research Internship, cdisselk@cs.ucsd.edu; Radha Jagadeesan, DePaul University, rjagadeesan@cs.depaul.edu; Alan Jeffrey, Mozilla Research, ajeffrey@mozilla.com; James Riely, DePaul University, jriely@cs.depaul.edu.

XXXX-XXXX/2018/7-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

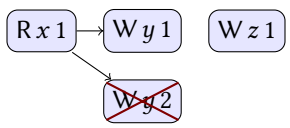
the predicted path until it is possible to validate the prediction; the execution is committed if the prediction is correct; otherwise it is rolled back. The speculation does not affect the observable input-output behavior of the program, thus ensuring correctness with respect to the usual intended semantics of the program. However, since there is clear timing differences between the cases where the prediction is and is not successful, it raises the concern that the predictive mechanisms themselves could cause information leaks via timing side-channels; a fear that is realized with devastating impact by the Spectre family of attacks [**?**].

- Several modern microprocessors [**?**] support transactions to aid in the design of correct and efficient concurrent programs. Transactions are executed optimistically; they are committed if there are no conflicts, and aborted otherwise. All memory effects of an aborted transaction are rolled back; so there is no way for a concurrent observer to detect an aborted transaction. However, the thread of the aborted transaction, via abort-handler code, gets notified of the cancellation of the transaction.
  Thus, when a transaction of a thread aborts, the thread can deduce plausible information about specific memory accesses of a concurrently executing transaction. In combination with the techniques used in Spectre, this potential has been exploited recently to accelerate and scale up the efficacy of the attacker in the Spectre family of attacks [**?**].
- Compiler optimizations may depend on both branches of a conditional, for example one that replaces $(\texttt{if}\,(M)\,\{\,C\,\}\,\texttt{else}\,\{\,C\,\})$ by C. In particular, $(\texttt{if}\,(r)\,\{\,x:=1\,\}\,\texttt{else}\,\{\,x:=1\,\})$ can be optimized to $(x:=1)$, which does not have a control dependency from r to the assignment to x. In contrast, the program $(\texttt{if}\,(r)\,\{\,x:=1\,\}\,\texttt{else}\,\{\,x:=2\,\})$ cannot be optimized, so the control dependency cannot be removed.
  This would be fine if there were no program constructs which could observe control dependencies, but unfortunately relaxed memory models such as JMM [**?**], C++ [**?**] or LLVM [**?**] allow for such observations. This means there is the possibility for information flows caused by optimizing compilers, which we investigate in this paper.
  Whereas information flows caused by speculation in hardware, or by transactional memory are known [**? ?**], these attacks on compiler optimizations, and the relaxed memory models that justify them, is new. In this paper we provide both a formal model for such attacks, and experimental evidence about their practicality.

Information flow attacks motivate the main technical development of this paper. Our model is based on *partially ordered multisets* [**? ?**] ("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 modelled by the pomset:



The edge from $(\mathrm{R}\,x\,1)$ to $(\mathrm{W}\,z\,2)$ indicates a data dependency. The novel aspect of the model is that events have *preconditions* which may be false. These are used in giving the semantics of conditionals, for example $(\texttt{if}\,(x)\,\{\,y:=1;\ z:=1\,\}\,\texttt{else}\,\{\,y:=2;\ z:=1\,\})$ has an execution:



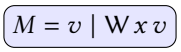
Th edges from $(\mathrm{R}\,x\,1)$ to $(\mathrm{W}\,y\,1)$ and $(\mathrm{W}\,y\,2)$ indicate control dependencies. The presence of a crossed out $(\mathrm{W}\,y\,2)$ indicates an event with an unsatisfiable precondition,

The novel contributions of this paper are:

- a model of program execution that includes speculation (§2),
- examples showing how the model can be applied, including information flow attacks on hardware, optimizing compilers, and transactional memory (§3), and
- experimental evidence about how practical it is to mount the new class of attacks (§4).

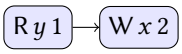
2 MODEL

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

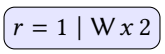


We make few requirements of the logic of preconditions, save that it has includes equalities between expressions, is closed under substitution, and supports a notion of implication.

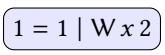
For example, the set of pomsets $[\![r := y;\ x := r + 1]\!]$ contains:



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

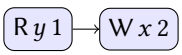


Next, we perform the substitution of r with 1 in every precondition, to get that $[\![x := r + 1]\!][1/r]$ contains the pomset:

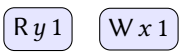


and since $(1 = 1)$ is a tautology, we elide it:

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

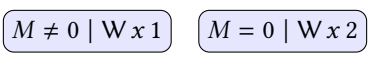


as required. There is an ordering $(\mathrm{R}\, y\, 1) < (\mathrm{W}\, x\, 2)$ because the precondition $(r = 1)$ depends on r. If the preconditon was independent of r then there would be no ordering, for example $[\![r := y;\ x := r + 1 - r]\!]$ contains the pomset:



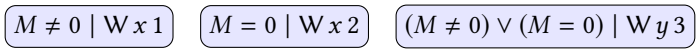
since the precondition $(r + 1 - r = 1)$ is independent of r.

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

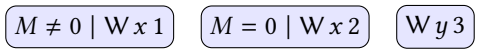


that is we have recorded behaviour from both branches of execution. Moreover, an action which is performed on both sides of the conditional can be merged, producing only one event in the resulting

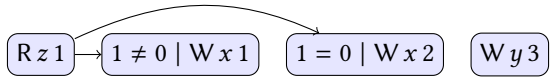
pomset. The precondition of the merged event is the disjunction of the preconditions of the original events. For example $[\![\texttt{if}\,(M)\,\{\,x\texttt{:=}1\texttt{;}\ y\texttt{:=}3\,\}\,\texttt{else}\,\{\,x\texttt{:=}2\texttt{;}\ y\texttt{:=}3\,\}]\!]$ contains the pomset:



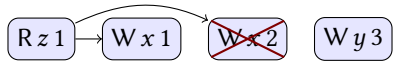
and since $(M \neq 0) \vee (M = 0)$ is a tautology, this is:



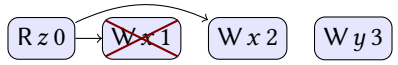
Combining this model of conditionals with the model of memory using substitutions gives that $[\![\texttt{if}\,(z)\,\{\,x\texttt{:=}1\texttt{;}\ y\texttt{:=}3\,\}\,\texttt{else}\,\{\,x\texttt{:=}2\texttt{;}\ y\texttt{:=}3\,\}]\!]$ contains the pomset:



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



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



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

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

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

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

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

We make data models precise in §2.1, define pomsets in §2.2, and operations on sets of pomsets in §2.3, which are used to give a compositional semantics for a simple imperative language.

2.1 Data models

A *data model* consists of:

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

such that:

- values include at least the constants 0 and 1,
- expressions include at least registers and values,
- expressions are closed under substitutions of the form $M[N/r]$,
- formulae include at least true, false, and equalities of the form $(M = N)$ and $(x = N)$,

- formulae are closed under negation, conjunction, disjunction,
- formulae are closed under substitutions of the form $\phi[x/r]$ or $\phi[N/x]$,
- there is a relation $\vDash$ between formulae, and
- there are partial functions R and W : $\mathcal{A} \to (\mathcal{X} \times \mathcal{V})$,

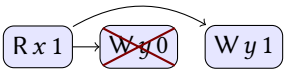
We shall say *a reads v from x* whenever $\mathsf{R}(a) = (x, v)$, and *a writes v to x* whenever $\mathsf{W}(a) = (x, v)$, and We shall say ϕ *implies* ψ whenever $\phi \vDash \psi$, ϕ is a *tautology* whenever $\mathsf{true} \vDash \phi$, ϕ is *unsatisfiable* whenever $\phi \vDash \mathsf{false}$, and ϕ is *independent of x* whenever $\phi \vDash \phi[v/x] \vDash \phi$ for every v. In examples, the actions are of the form $(\mathsf{R}\, x\, v)$, which reads v from x, and $(\mathsf{W}\, x\, v)$, which writes v to x. Going forward, we assume a given data model, though some examples in §3 make use of particular $\mathcal{A}$.

2.2 3-valued pomsets

Recall the definition of a pomset from [**?**]:

Definition 2.1. A *pomset* $(E, \leq, \lambda)$ with alphabet Σ is a partial order $(E, \leq)$ together with $\lambda : E \to \Sigma$.

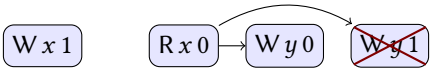
Going forward, we fix the alphabet $\Sigma = (\Phi \times \mathcal{A})$. We will write $(\phi \mid a)$ for the pair (ϕ, a), elide ϕ when ϕ is a tautology, and write ↯ when ϕ is unsatisfiable. We lift terminology from logical formulae and actions to events, for example if $\lambda(e) = (\phi \mid a)$ and then we say e is unsatisfiable whenever ϕ is unsatisfiable, e writes v to x whenever a writes v to x, and so forth. We visualize a pomset as a graph where the nodes are drawn from E, each node e is labelled with $\lambda(e)$, and an edge $d \to e$ corresponds to an ordering $d \leq e$. For example:



is a visualization of the pomset where:

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

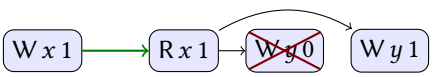
We are building a compositional semantics of shared memory concurrency, which means we require a notion of when a read has a matching write. This is a property we require of closed programs, but *not* of open programs. For example a program whose semantics includes:



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

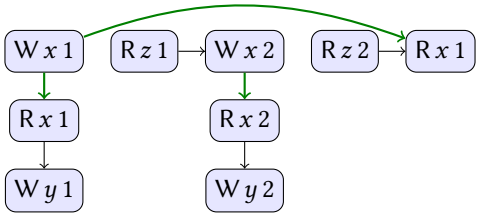
- $d < e$,
- if e is satisfiable, then d is a tautology,
- d writes v to x, and e reads v from x, and
- there is no $d < c < e$ such that c writes to x.

In diagrams, for readability we often highlight the reads-from edges, for example:

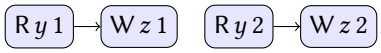


Unfortunately by itself, this is not enough. The problem is the final clause saying that there does not exist an *x-blocking* event c between d and e. Unfortunately, concurrency can turn events that were

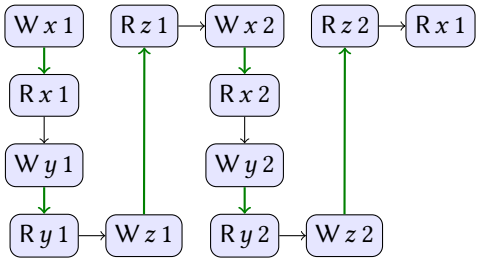
not x-blockers into an x-blocker, *even if the new thread does not mention x*. To see this, consider a program with execution:



If this is placed in parallel with:



then a possible execution is:



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

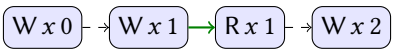
There are a number of ways this can be addressed, for example in models such as [**?**] the reads-from relation is taken as a primitive. In this paper, we propose *3-valued pomsets* as a solution. These are pomsets in which, in addition to positive statements ($d < e$) (interpreted as e depends on d),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

Definition 2.2. A *3-valued poset* $(E, \leq, \not<)$ is a poset $(E, \leq)$ together with $\not< \subseteq (E \times E)$ such that:

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

Definition 2.3. A *3-valued pomset* $(E, \leq, \not<, \lambda)$ is a 3-valued poset $(E, \leq, \not<)$ and a pomset $(E, \leq, \lambda)$.

In diagrams, we visualize ($e \not< d$) as a dashed arrow from d to e (note the flip of direction). We wrefer to edges introduced by ($d < e$) as *strong edges* and by ($e \not< d$) as *weak edges*, for example:



Structures similar to 3-valued pomsets have come up in many guises, for example rough sets [**?**] or ultrametrics over $\{0, {}^1\!/_2, 1\}$. They correspond to axioms A1–A3 of Lamport's *system executions* [**?**]. They are the notion of pomset given by interpreting $d \leq e$ in a 3-valued logic [**?**].

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

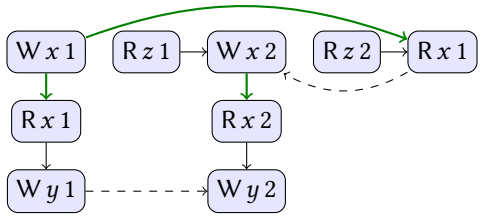
- $d < e$,

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

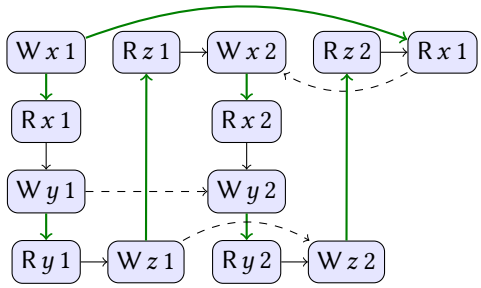
Definition 2.5. A 3-valued pomset is x-closed if for every $e \in E$:

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

In our previous example, in order for $(\mathrm{R}\, x\, 1)$ to read from $(\mathrm{W}\, x\, 1)$, we either need $(\mathrm{W}\, x\, 1) \not< (\mathrm{W}\, x\, 2)$ or $(\mathrm{W}\, x\, 2) \not< (\mathrm{W}\, x\, 1)$, for example:



then putting this in parallel as before results in:



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

2.3 Sets of 3-valued pomsets

Our model of programs is going to be sets of 3-valued pomsets. In this section we define the operations on pomsets which are used in giving the semantics. These are operations such as prefixing, parallel composition, restriction, and so on; they are familiar from models of concurrency such as [**?**], but adapted here to the setting of speculative evaluation.

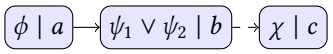
Definition 2.6. Let $P_0 \in (\mathcal{P}_1 \parallel \mathcal{P}_2)$ whenever there are $P_1 \in \mathcal{P}_1$ and $P_2 \in \mathcal{P}_2$ such that:

- $E_0 = E_1 \cup E_2$,
- 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 $\lambda_0(e) = (\phi_0 \mid a)$ then either:
  - $\lambda_1(e) = (\phi_1 \mid a)$ and $\lambda_2(e) = (\phi_2 \mid a)$ and ϕ_0 implies $\phi_1 \vee \phi_2$,
  - $\lambda_1(e) = (\phi_1 \mid a)$ and $e \notin E_2$ and ϕ_0 implies ϕ_1, or
  - $\lambda_2(e) = (\phi_2 \mid a)$ and $e \notin E_1$ and ϕ_0 implies ϕ_2.

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

and $\mathcal{P}_2$ contain:

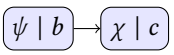
then $\mathcal{P}_1 \parallel \mathcal{P}_2$ contains:



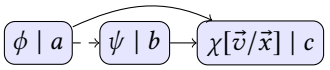
Definition 2.7. Let $(\phi \mid a) \rightarrow \mathcal{P}$ be the set $\mathcal{P}'$ where $P' \in \mathcal{P}'$ whenever there is $P \in \mathcal{P}$ such that:

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

Prefixing is used to define the semantics of reads and writes, and adds a new event c with action a and a precondition which implies ϕ. It performs a substitution on the preconditions of any subsequent events. The tricky part of the definition is the last two clauses, which places requirements on $c < e$ and $c \not\leq e$. In normal prefixing, these would always be required, but we only require it when either the precondition of e depends on a variable read by a, or if c and e write to the same variable. For example, if $\mathcal{P}$ contains:



where a reads v from x and writes w to y, b writes to y, and ψ is independent of x, then $(\phi \mid a) \rightarrow \mathcal{P}$ contains:



Definition 2.8. Let $\mathcal{P}[M/x]$ be the set $\mathcal{P}'$ where $P' \in \mathcal{P}'$ whenever there is $P \in \mathcal{P}$ such that:

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

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

Definition 2.9. Let $(\phi \mid \mathcal{P})$ be the subset of $\mathcal{P}$ such that $P \in \mathcal{P}$ whenever:

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

Definition 2.10. Let $(\nu x \,.\, \mathcal{P})$ be the subset of $\mathcal{P}$ such that $P \in \mathcal{P}$ whenever P is x-closed.

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

Definition 2.11.

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

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

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

3 EXAMPLES

3.1 Sequential memory accesses

In the semantics of memory, there are two very different ways memory can be accessed: sequentially or concurrently. These are modelled differently, since hardware and compilers give very different guarantees about their behaviour. In this section, we discuss the sequential semantics, and leave the concurrent semantics to §3.2.

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

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

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

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

which contains the pomset:

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

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

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

which contains the pomset:

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

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

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

which contains the pomset:

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

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

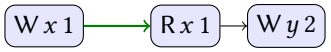
$$\boxed{\mathrm{W}\, x\, 0} \quad \boxed{\mathrm{W}\, y\, 1}$$

Note that there is no requirement of order between $(\mathrm{W}\, x\, 0)$ and $(\mathrm{W}\, y\, 1)$.

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 for assignment, where if C has preconditon ϕ then $x := M\,;\, C$ has precondition $\phi[M/x]$.

3.2 Concurrent memory accesses

We now turn to the case of concurrent accesses to memory. Consider a concurrent version of the program from §3.1: $(x := 1 \mid\mid y := x + 1)$. One execution of this program is where the write to y performs a concurrent read of x:



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

$$(r + 1 = 2 \mid \mathrm{W}\, y\, 2) \longrightarrow [\![\mathtt{skip}]\!][2/y]$$

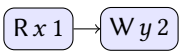
which contains the pomset:

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

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

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

which contains the pomset:

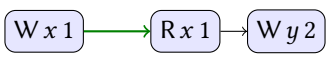


Note that $(\mathrm{R}\, x\, 1)$ reads 1 from x, and while $(x + 1 = 2)[1/x]$ is a tautlogy, $(x + 1 = 2)$ is not, and so there is a dependency $(\mathrm{R}\, x\, 1) < (\mathrm{W}\, y\, 2)$.

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

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

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



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

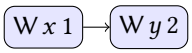
This example demonstrates how read and write events capture the concurrent semantics of memory. In an execution containing an event with label $(\mathrm{R}\, x\, v)$, if the execution is x-closed, then there must be an event it reads from, for example one labelled $(\mathrm{W}\, x\, v)$.

3.3 Independent writes

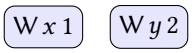
Consider an example with two independent writes $(x := 1\,;\, y := 2)$. This has semantics including:

$$(1 = 1 \mid \mathrm{W}\,x\,1) \longrightarrow (2 = 2 \mid \mathrm{W}\,y\,2) \longrightarrow \{\emptyset\}$$

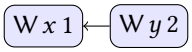
One of the executions this contains is:



but it also contains:



and:

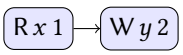


since there is no requirement that $(\mathrm{W}\,x\,1) < (\mathrm{W}\,y\,2)$.

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

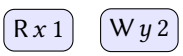
3.4 Independent reads and writes

Whereas write prefixing introduces weak dependencies on events which write to the same location, read prefixing introduces strong dependencies on preconditions which depend on the location being read. For example in §3.2 we saw that the program $(y := x + 1)$ includes the pomset:



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

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



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

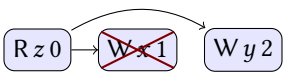
Note this this example shows that we are not just dealing with a syntactic notion of dependency, which is common in hardware models of memory. In syntactic dependency, since r occurs free in $(y := r + 2 - r)$, there would be a dependency between $(r := x)$ and $(y := r + 2 - r)$. In contrast, this model is based on logical implication, which can be interpreted semantically.

3.5 Control dependencies

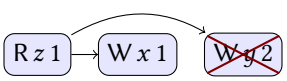
Conditionals introduce control dependencies, for example consider the program:

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

This includes executions in which the false branch is taken:



and ones where the true branch is taken:



In both cases, we record the actions in the branch that was not taken. This is a novel feature of this model, and is intended to capture speculative evaluation. In §3.7 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 modelled, consider the semantics of $[\![x := 1;\ \texttt{skip}]\!]$, which contains any pomset of the form:

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

in particular it contains:

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

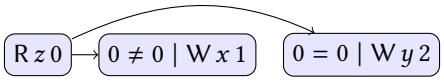
Similarly $[\![y := 2;\ \texttt{skip}]\!]$ contains:

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

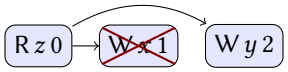
and so $[\![\texttt{if}\,(r)\,\{\,x := 1;\ \texttt{skip}\,\}\,\texttt{else}\,\{\,y := 2;\ \texttt{skip}\,\}]\!]$ contains:

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

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



which gives the required pomset:



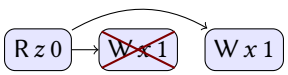
Note that the precondition $r = 0$ is dependent on r, and so there is a dependency $(\mathrm{R}\, z\, 0) < (\mathrm{W}\, y\, 2)$, modelling the control dependency introduced by the conditional.

3.6 Control independencies

In most models of control dependencies, the dependency relation is syntactic, based on whether the action occurs inside syntactically inside a conditional. In contrast, the notion in this model is semantic: if an action can occur on both sides of a conditional, there is no control dependency. Consider a variant of the example from §3.5:

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

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



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

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

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

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

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

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

that is P_1 where:

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

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

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

that is P_2 where:

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

Crucuially, 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 $\mathcal{P}_1 \sqcup \mathcal{P}_2$ is P_0 where:

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

that is:

$$\mathrm{W}\, x\, 1$$

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:

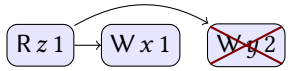
$$\mathrm{R}\, z\, 0 \qquad \mathrm{W}\, x\, 1$$

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

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

$$\mathrm{R}\, z\, 1 \qquad \mathrm{W}\, x\, 1$$

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



so these programs have different dependency relations, depending on conditional branches that were not taken. In §3.9 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.

3.7 Spectre

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

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

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:

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

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

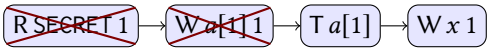
We model the touched test by introducing a new action $(\mathsf{T}\,x)$ and defining:

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

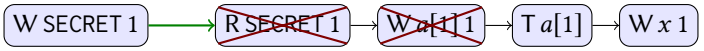
The additional requirement we need to add for x-closure is:

- if $\lambda(e) = (\phi \mid \mathsf{T}\,x)$ then there is $d < e$ where d reads or writes x.

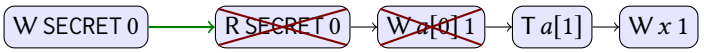
Note that there is no requirement that d be satisfiable, and indeed Spectre has the execution:



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



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



Thus, the attacker has managed to leak the value of a high-security location to a low-security one.

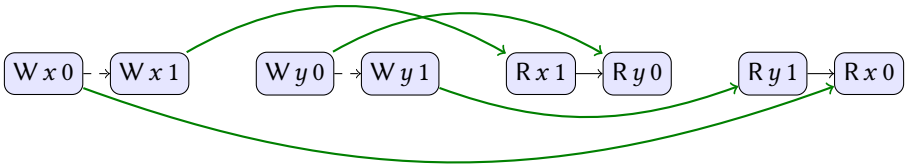
This shows how our model of speculation can express (very abstract, untimed) Spectre attacks.

3.8 Relaxed memory

In §3.9 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. For this reason, we present a simple model of relaxed memory, which is strong enough to capture this attack. The 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; || if (x) { r := y; } || if (y) { s := x; }
```

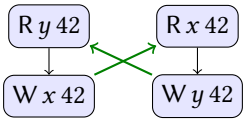
which includes the execution:



This model does not introduce thin-air reads (TAR), for example the TAR pit program is:

```
x := y; || y := x;
```

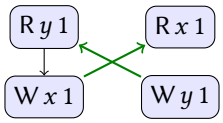
but an attempt to produce a value from thin air fails, for the usual reason of producing a cycle in $\leq$:



This cycle can be broken if one of the writes does not depend on the read, for example:

```
x := y; || r := x; y := r+1-r;
```

has execution:



Note that $(\mathrm{R}\, x\, 1) \not\leq (\mathrm{W}\, y\, 1)$, so this does not introduce a cycle.

{We get around the no-per-candidate result of [?] by modeling dependency more precisely.}

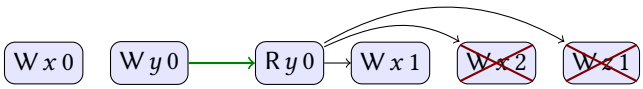
{We handle all of the causality test cases [?] that are expressible in our language (TC15-16 require loops). This includes TC9, which fails for [?]. See appendix.}

3.9 Information flow attacks on relaxed memory

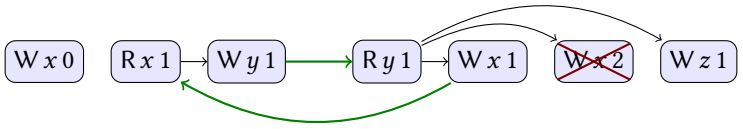
Consider an attacker program, again using security checks to try to learn a SECRET. Whereas SPECTRE uses hardware capabilities, which have to be modeled by adding extra capabilities to the language, this new attacker works by exploiting relaxed memory which can result in unexpected information flows.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

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

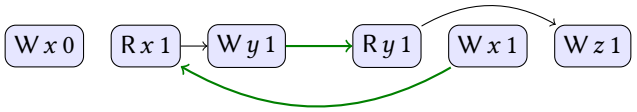
In the case where SECRET is 2, this has many executions, one of which is:



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



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



Note that in this case, there is no dependency from $(\mathrm{R}\, y\, 1)$ to $(\mathrm{W}\, x\, 1)$, which is what makes this execution possible. Thus, if the attacker sees an execution with $(\mathrm{W}\, z\, 1)$, 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:

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

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

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

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

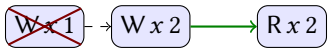
```
x := 1;
if (y == 0) { }
else if (canRead(SECRET)) { }
else { z := 1; }
```

After these optimizations, a sequentially consistent execution exhibits $(\mathrm{W}\, z\, 1)$. We discuss the practicality of this attack further in §4.

{Cite work on noninterference with two executions. Perhaps [?].}

3.10 Dead store elimination

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



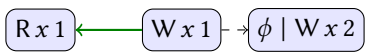
Recall that for any satisfiable e, if e reads x from y then d is a tautology. This means that, although we can eliminate $(\mathrm{W}\, x\, 1)$ we cannot eliminate $(\mathrm{W}\, x\, 2)$.

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

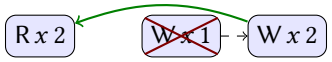
Note that if dead store elimination is *always* performed, then there is an information flow attack similar to the one in §3.9. Consider the program:

```
(
  r := x;
) || (
  x := 1;
  if (canRead(SECRET)) { if (SECRET) { x := 2; } }
  else { x := 2; }
)
```

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



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

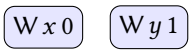


Now the (W x 2) event is a guaranteed write, so the (W x 1) is eliminated, and so cannot be read. In the case that the attacker can rely on dead store elimination taking place, this is an information flow: if the attacker observes x to be 1, then they know `SECRET` is 0. We return to this attack in §4.

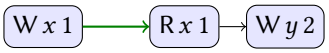
3.11 Thread inlining

One property one could ask of a model of shared memory is thread inlining: any execution of $[\![P\,;\,Q]\!]$ is an execution of $[\![P \mid\mid Q]\!]$. 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 [**?**], 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 §3.1 that the program `(x := 0; y := x+1;)` has execution:



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

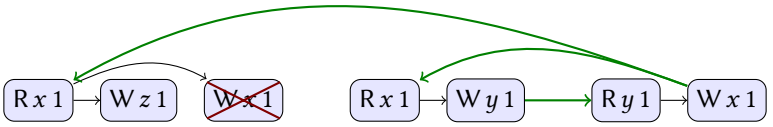


That is, in the sequential case there is no dependency from the write of x to the write of y, but in the concurrent case there is such a dependency.

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

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

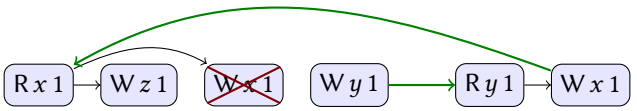
This has no execution containing (W z 1). Any attempt to build such an execution results in a cycle:



Inlining the thread `(y := x)` gives [**?** , Ex 12]:

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

with execution:



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

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

Removing the syntax sugar, this is:

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

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

$$\boxed{r_1 = 1 \mid \mathrm{W}\, z\, 1} \quad \boxed{r_1 = x = 1 \mid \mathrm{W}\, y\, 1}$$

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

$$\boxed{r_1 \neq 1 \mid \mathrm{W}\, x\, 1} \quad \boxed{r_1 \neq 1 \mid \mathrm{W}\, y\, 1}$$

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

$$\boxed{r_1 = 1 \mid \mathrm{W}\, z\, 1} \quad \boxed{r_1 \neq 1 \mid \mathrm{W}\, x\, 1} \quad \boxed{(r_1 = x = 1) \vee (r_1 \neq 1) \mid \mathrm{W}\, y\, 1}$$

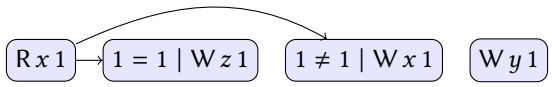
which means $[\![\texttt{if}\,(r_1 = 1)\,\{\, z := 1; r_2 := x; y := r_2; \texttt{skip}\,\}\,\texttt{else}\,\{\, x := 1; r_3 := x; y := r_3; \texttt{skip}\,\}]\!][x/r_1]$ includes:

$$\boxed{x = 1 \mid \mathrm{W}\, z\, 1} \quad \boxed{x \neq 1 \mid \mathrm{W}\, x\, 1} \quad \boxed{(x = x = 1) \vee (x \neq 1)) \mid \mathrm{W}\, y\, 1}$$

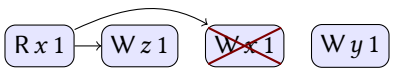
Now $(x = x = 1) \vee (x \neq 1)$ is a tautology, so this is just:

$$\boxed{x = 1 \mid \mathrm{W}\, z\, 1} \quad \boxed{x \neq 1 \mid \mathrm{W}\, x\, 1} \quad \boxed{\mathrm{W}\, y\, 1}$$

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



which simplifies to:



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

3.12 Release/acquire synchronization

In relaxed memory models, synchronization actions act as memory fences: that is, they are a barrier to reordering memory accesses. In this section, we present a simple model of release/acquire fencing. In §3.13, we show that this can be scaled up to a model of transactional memory.

We assume there are sets Rel and Acq $\subseteq \mathcal{A}$. We say that a is a *release action* if $a \in$ Rel and a is an *acquire action* if $a \in$ Rel. In a pomset, a release event is one labelled with a release action, and an acquire event is one labelled by an acquire action. The semantics of fences are given by adding an extra constraint to the definition of $(\phi \mid a) \longrightarrow \mathcal{P}$ (recalling that c is the a-labelled event being introduced):

- $c \leq e$ whenever c is an acquire event or e is a release event.

This constraint ensures that events are ordered before a release and after an acquire.

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

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

The semantics of programs with releasing write and acquiring read are the same as for regular write and read and, but with $\mathsf{Rel}\,x\,v$ replacing $\mathsf{W}\,x\,v$ and $\mathsf{Acq}\,x\,v$ replacing $\mathsf{R}\,x\,v$.

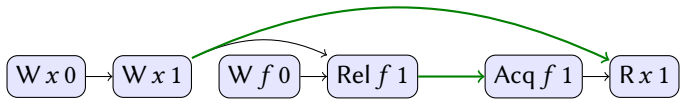
$$[\![\mathsf{rel}\,x := M;\,C]\!] \;=\; \bigcup_v (M = v \mid \mathsf{Rel}\,x\,v) \to [\![C]\!][M/x]$$

$$[\![\mathsf{acq}\,r := x;\,C]\!] \;=\; [\![C]\!][x/r] \cup \bigcup_v (\mathsf{true} \mid \mathsf{Acq}\,x\,v) \to [\![C]\!][x/r]$$

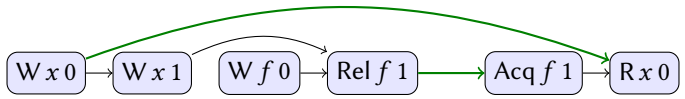
For example, consider the program:

```
(x := 0; f := 0; x:= 1; rel f := 1;) || (acq r := f; s := x;)
```

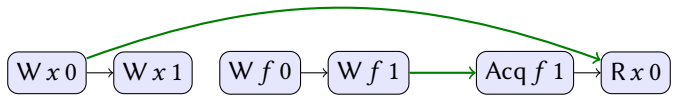
This has an execution:



but *not*:



since $(\mathsf{W}\,x\,0) < (\mathsf{W}\,x\,1) < (\mathsf{R}\,x\,0)$, so this pomset does not satisfy the requirements to be a 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.

3.13 Transactions

We present a simple model of serializable transactions that is sufficient to capture Prime+Abort attacks [?]. We assume that the set of locations $\mathcal{X}$ is partitioned into *cache sets*.

The action $(\mathsf{B}\,v) \in \mathsf{Acq}$ represents the begin of a transaction with id v and $(\mathsf{C}\,v) \in \mathsf{Rel}$ represents the corresponding commit, with semantics:

$$[\![r := \mathtt{begin}\,v;\,D]\!] \;=\; (\mathsf{true} \mid \mathsf{B}\,v) \to [\![C]\!][v/r]$$

$$[\![r := \mathtt{commit}\,v;\,D]\!] \;=\; (\mathsf{true} \mid \mathsf{C}\,v) \to [\![D]\!][1/r] \cup (\mathsf{false} \mid \mathsf{C}\,v) \to [\![D]\!][0/r]$$

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

Definition 3.1. We say that event c *matches* b if $\lambda(c) = (\mathsf{C}\,v)$ and $\lambda(b) = (\mathsf{B}\,v)$. We say that a begin event *aborts* if every matching commit is unsatisfiable. 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. 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, then $d \le e$ implies $d \le b$ and $e \not\le d$ implies $b \not\le d$
(5) if c ends e, but not d, then $e \le d$ implies $c \le d$ and $d \not\le e$ implies $d \not\le c$,

(6) if e and d belong to b and read the same location, then both read the same value,
(7) if e belongs to b, then e implies some matching c that ends e, and
(8) if b aborts then there must be some e that belongs to b and some tautologous $d \leq e$ that does not belong to b such that e and d touch locations in the same cache set.

In discussion, we identify transactions by their unique begin event. A transaction that does not abort is *successful*. Conditions 1-5 ensure serializability of committed transactions. Conditions 4-6 also ensure strong isolation for non-transactional events [**?**]. Condition 7 ensures that all events in aborted transactions are unsatisfiable. Condition 8 requires that aborts only occur due to caching conflicts—this is similar to the treatment of the touch operation in §3.7.

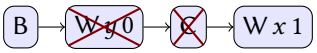
To model Prime+Abort, we require an honest agent whose cache-set access depends upon a secret. If $a[0]$ and $a[1]$ belong to separate cache sets, then such an honest agent is:

```
a[SECRET] := 1
```

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

```
begin; y:=0; r:=commit; x:=r+1;
```

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



4 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. Sections 3.9 and 3.10 presented other attacks along this same theme, which leverage relaxed memory models and dead store elimination respectively. In particular, the latter attack (and, to a degree, the former attack), result not from hardware optimizations, but from common *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 [**?**].

In this section we present concrete implementations of the attacks outlined in Sections 3.9 and 3.10, in both cases leveraging compiler optimizations to construct an information-flow attack. The attacker model for these attacks (detailed in Section 4.1) is currently unrealistic in a real-world sense, rendering these attacks proof-of-concepts rather than immediately exploitable vulnerabilities. However, we believe the novelty of their general mechanisms may lead to interesting discussion; and with much more development, these attacks may evolve into genuine threats against real-world targets such as JIT compilers. We demonstrate the efficacy of both of our concrete proof-of-concept attacks against modern compilers and hardware (under our attacker model), including both the `clang` and `gcc` C compilers.

All of our experiments are performed on a **{describe machine}** with `clang` version **{clang version}** and `gcc` version **{gcc version}**.

4.1 Attacker model

In our attacker model, we assume that there is a SECRET which an attacker wishes to learn; for instance, SECRET may be a cryptographic key hardcoded into the application. This SECRET is known to the compiler at compile time, but may not be accessed except behind a security check. We assume that the security check always evaluates to `false` at runtime for the attacker, but that the attacker is allowed to write (compile and execute) arbitrary code subject to the above restrictions. The attacker has no capabilities other than writing and executing code — in particular the attacker may not disassemble the compiler or libraries to learn the SECRET directly; may not examine the internal state of the compiler; may not access timers of any kind; and may not leverage hardware

side channels. The attacker's goal is to learn the value of the SECRET despite only being allowed to use it inside dead code, that is, code that can never be executed at runtime.

As a hypothetical concrete example, suppose there is a library which contains a hardcoded SECRET but does not allow library users unrestricted access to this SECRET. Instead, library users must call an (arbitrarily complex) library function as a guard before using the SECRET in code. In this hypothetical example, a static compiler pass ensures that the attacker's code only accesses the SECRET inside an `if` statement after calling the library's guard function; that is, the static pass ensures that the SECRET is only accessed at program points where it can be proven that the guard function returned `true`.

As noted above, this is not necessarily a realistic attacker model in the real-world sense, which means that the attacks presented in this section are more of theoretical interest than practical concern; few environments allow attackers to write nearly-arbitrary C code that touches secrets in any way. However, the mechanism of the attack is novel and could potentially be applied in other contexts. For instance, many real-world contexts allow "attackers" (untrusted or third-party entities) to write code in a scripting language which is then compiled alongside and integrated into a larger application, often in a just-in-time (JIT) manner. JavaScript code from third-party websites running in a browser is a common example of this. Our attack gives an attacker similar "scripting" capabilities against a compiler, except it considers the simpler setting of 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 in the browser itself, and may be more liberal with what it considers a "constant". We plan to explore JavaScript attacks of this type as future work.

4.2 Load-store reordering attack

We begin by examining the attack in Section 3.9 in more detail, subject to the attacker model given above. In particular, 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, that is, code that can never be executed at runtime. This attack was tested and works against gcc version **{gcc version}**.

The form of the attack presented in Section 3.9 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, we did find a related pattern in which gcc will emit a different ordering of the read of y and the write of x depending on the value of a SECRET:

```
y := 0;
(
  y := x;
) || (
  x := 1;
  if (canRead(SECRET)) { x := SECRET; }
  if (y) { return 0; }
  else { return 1; }
)
```

In the case that SECRET is 1, gcc will remove the `if` statement entirely, and move the read of y above the write of x. However, if SECRET is not 1, the `if` statement must remain intact, and gcc will not move the read of y. This means that if SECRET is 1, the second thread will always read `y == 0` and always return 1. However, if SECRET is not 1, 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 return `0`. In this way, we leverage compiler load-store reordering to learn the value of a compile-time `SECRET`.

We extend this attack to leak a secret consisting of an arbitrary number `N` of bits.

To do this, we simply compile `N` copies of the test function, each performing a boolean test on a single bit of the secret. The function used for eading the `k`th bit is as follows (for `N <= 64`):

```
(
  y := x;
) || (
  x := 1;
  if (canRead(SECRET)) { x := (SECRET & (1 << k)) ? 1 : 0; }
  if (y) { return 0; }
  else { return 1; }
)
```

Following the same analysis as above, this function will always return `1` if the appropriate bit of `SECRET` is `1`, but may return `0` if the appropriate bit of `SECRET` is `0`. The extension of the attack to the general case with truly arbitrary `N` is straightforward; `SECRET` becomes an array of 64-bit values, and we use `k / 64` and `1 << (k & 63)` as the array index and bitmask respectively.

We make three additional tweaks to improve the reliability so that the attacker can confidently infer the value of `SECRET` based on the observed return values of the function. First, rather than performing `y := x` only once in the first thread, we perform `y := x` continuously in a loop. This maximizes the probability that, once `x := 1` occurs in the second thread, `y` will be immediately assigned `1` by the first thread and the second 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 second thread (in the case where the appropriate bit of `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 the appropriate bit of `SECRET` is `1`. We do this by inserting many copies of the line

```
if (canRead(SECRET)) { x := (SECRET & (1 << k)) ? 1 : 0; }
```

instead of just one. In the case where the appropriate bit of `SECRET` is `0`, this results in many calls to `canRead(SECRET)` and many conditional jumps, which in practice creates a timing window for the first thread to perform `y := x`. However, in the case where the appropriate bit of `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 return value of the function in each case. We note that if *any* of the redundant runs produces a return value of `0` for a particular bit position, we can be certain that the corresponding bit of `SECRET` *must* be `0`, as it implies the read of `y` was not reordered upwards in that particular function. On the other hand, the more runs that produce a return value of `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 the appropriate bit of `SECRET` is `1`.

Figure 1 gives the performance results for this attack against `gcc` version **{ver}**. 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. Note that this bandwidth assumes that all copies of the attack function are already compiled; the cost of compilation is not included here.

Redundancy	Bandwidth (bits/s)	Bitwise Acc	Per-run Acc
1	3.17 million	90.13%	0.0%
2	1.62 million	96.77%	0.7%
3	1.07 million	98.84%	3.9%
4	812 thousand	99.55%	13.5%
5	652 thousand	99.83%	34.0%
7	466 thousand	99.97%	71.8%
10	322 thousand	99.998%	96.6%
15	216 thousand	100.00%	100.0%

Fig. 1. 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. **{Note: numbers are not final (collected on Craig's machine inside a VM), but give an idea of where we stand.}**

4.3 Dead store elimination attack

In this section we return to the attack in Section 3.10 based on dead store elimination. We show that in our attacker model (given in Section 4.1), 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, that is, code that can never be executed at runtime. 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 Section 3.10, and extend it to leak a secret consisting of an arbitrary number `N` of bits. As we did in the load-store reordering attack, we again compile `N` copies of the test function, each performing a boolean test on a single bit of the secret. The function used for reading the `k`th bit is as follows (for `N <= 64`):

```
(
  r := x;
) || (
  x := 1;
  if (canRead(SECRET)) {
    if (SECRET & (1 << k)) { x := 2; }
  } else {
    x := 2;
  }
)
```

Then, we test each function in turn, each time noting the value of `r` observed by the 'listening' thread. If the appropriate bit of `SECRET` is 1, the `x := 2` assignment is guaranteed to happen, so the compiler can eliminate the `x := 1` assignment as a dead store and we will observe `r == 2`; however, if the appropriate bit of `SECRET` is 0, the `x := 1` assignment cannot be eliminated, and we will observe `r == 1` with some probability. The extension of the attack to the general case with truly arbitrary `N` is straightforward and proceeds exactly as it did for the attack on load-store reordering.

We make three additional tweaks to improve the reliability so that the attacker can confidently infer the value of SECRET based on the observed values of r. These three tweaks strongly resemble the reliability tweaks we made to the load-store reordering attack and differ only in a few details.

First, rather than simply observing x with r := x in the 'listening' thread, we continuously load x in a loop until a nonzero value is observed — i.e., we perform

```
do {
  r := x;
} while(r == 0);
```

This remedies the case where r := x could observe a value of x from 'before' either of the two possible writes performed by the other thread. **{do we need to explicitly say that we ensure x is initialized to 0 — and coherently seen as such by both threads — before starting the attack?}**

Second, we insert additional time-consuming computation immediately following the x := 1 operation in the 'main' thread. This lengthens the timing window in which x has the value 1, increasing the likelihood that the 'listening' thread will be able to observe x == 1 (unless the x := 1 write was eliminated, of course). Inserting this computation can 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 the appropriate bit of 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 r (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. This is because clang's dead store elimination pass operates only within basic blocks, and uses a heuristic to stop scanning the basic block early if it is too large. Nonetheless, we find that even with these restrictions, we are able to construct a reliable and fast attack against both clang and gcc.

Finally, we redundantly execute the entire attack several times, noting the final value of r (the first observed nonzero value of x) in each case. We note that if *any* of the redundant runs produces r == 1 for a particular bit position, we can be certain that the corresponding bit of SECRET *must* be 0, as it implies that the x := 1 write was not eliminated in that particular function. On the other hand, the more runs that observe r == 2 in a particular bit position despite our other reliability-increasing measures taken above, the more certain we can be that the x := 1 write was eliminated in that function, and the appropriate bit of SECRET is 1.

Unlike the load-store reordering attack, our implementation of the dead store elimination attack has two important "knobs" which trade off reliability vs. performance, rather than only one. First, we have the length of time which the writing thread attempts to "stall" immediately after the x := 1 write. Second, we have the number of entire redundant runs of the attack that are performed before the attacker reaches her conclusion. Increased reliability can be achieved by adjusting either of these knobs, and they each have (different) effects on the overall performance of the attack. After exploring the parameter space, we found that 3 redundant runs is sufficient to provide near-100% accuracy while allowing us to maximize the speed of the attack. Specifically, on our machine, our attack on gcc reaches speeds of **{exact gcc leak speed}** bits leaked per second (**{exact gcc raw leak speed}** 'raw' bits leaked per second, that is, before error correction) with **{exact gcc accuracy}**, while our attack on clang reaches speeds of **{exact clang leak speed}** bits leaked per second (**{exact clang raw leak speed}** 'raw' bits leaked per second) with **{exact clang accuracy}**. In particular, this means our attack can leak a 2048-bit cryptographic key in under **{exact speed}** ms, on either gcc or clang, with probability **{exact probability}** that there are exactly zero bit

errors in the leaked key, or probability **{exact probability}** that there is at most one bit error in the leaked key.

5 LOGIC

In this section, we develop sufficient logical infrastructure to prove that our semantics disallows thin air executions. We present a variant of the TAR-pit example from §3.8 which poses difficulties under many speculative semantics.

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

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

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

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

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

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

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

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

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

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

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

Proposition 5.1. *Let ϕ be prefix-closed. Let $\mathcal{P}_1, \mathcal{P}_2$ be augmentation-closed[2] Then:*

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

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

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

For the inductive case, consider $P \in P_1 \parallel P_2$ where $P_i \in \mathcal{P}_i$. Since $\mathcal{P}_1$ and $\mathcal{P}_2$ are augmentation closed, we can assume that the restriction of P to the events of P_i coincides with P_i, for $i = 1, 2$.

[1] P' is a prefix of P if $E' \subseteq E$, $e \in E'$ and $d \leq e$ imply $d \in E'$, and $(\lambda', \leq', \not<')$ coincide with $(\lambda, \leq, \not<)$ for elements of E'.
[2] 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.

Consider a prefix P' derived by removing a maximal element e from P. Suppose e comes from P_1 (the other case is symmetric). Since P_2 is a prefix of P' and $P' \models \phi$ by induction hypothesis, we deduce that $P_2 \models \phi$. Since $P_1 \in \mathcal{P}_1$, by assumption $\phi, \mathcal{P}_1 \models \phi$ we deduce that $P \models \phi$. □

We now turn to the example program, which is a variant of [**?** , Figure 8]:

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

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

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

Closing y, we have, $\Diamond^{-1}(\mathrm{R}\, y\, 1) \Rightarrow \Diamond^{-1}(\mathrm{W}\, y\, 1)$ and thus, using the left conjunct, we have:

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

By closing x, we can replace $\Diamond^{-1}(\mathrm{R}\, x\, 1)$ with $\Diamond^{-1}(\mathrm{W}\, x\, 1)$:

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

Applying coinduction to the right conjunct, we have:

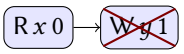
$$\Diamond^{-1}(\mathrm{W}\, a\, 1) \Rightarrow (\Diamond^{-1}(\mathrm{W}\, x\, 1) \wedge \Box^{-1}(\neg \mathrm{W}\, x\, 1))$$

Simplifying, we have, as required:

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

6 CONCLUSIONS AND FUTURE WORK

One oddity of the model is that $[\![r := x;\ y := r;\ \texttt{skip}]\!]$ includes:



where the write action guessed its value incorrectly, and therefore has precondition $0 = 1$. This form of speculative execution does not appear to be used in practice. In order to disallow it, one could change the semantics of `skip` to introduce a tick action denoting successful completion of the thread and only consider executions in which the precondition of every tick action is satisfiable. We leave the elaboration of this idea as future work.

{Comment on the following:}

- coherence = per location total order on $\not<$
- Validation of write removal requires some tricks to ensure that thread does not rf its own write
- Definition of rf can use $\not<$ in first clause, rather than $\leq$. We chose the stronger definition because it makes some of the examples simpler: in particular, the example motivating rf needs a volatile in the other thread if you don't have rf-implies-hb. Old text: The notion rf-pomset is sufficient to capture hardware models and release/acquire access in C++, where reads-from implies happens-before [**?**]. To model C++ relaxed access, it would be necessary to use a more general notion of rf-pomset, where $(d, x, e) \in \mathsf{RF}$ does not necessarily imply $d < e$, instead requiring that $(< \cup \mathsf{RF})$ be acyclic.
- The design space for transactions is very rich [**?**]. We have only presented one option. we pun between abort and false commit

- causality test cases 1, 6, 8, 9, 18, 20 (12?) require that the logic make assertions about the domain of variables

{A question:} Here is an example that [**?**] fails on:

```
r1:=x; y:=r1;   ||  r2:=y; if(r2<2){x:=1;}  || y:=2;
```

This should allow `r1==r2==1`. Hopefully this is allowed by our semantics...

{Integrate this example on information flow?}

Let `P(hi)` be a function with domain $\{0, ..., n\}$ and codomain $\{1, ..., n\}$. Our aim is to write a program that tests whether there is information flow from input `hi` to the return value `lo`. Consider:

```
[Initial value of m is $0$ ,  noflow is  false. ]
T1: r = z; m =r;
T2: s=m; if (s!=0)  z=P(s); noflow=true;  else  z=P(0);
% T2:  s = m;
%      switch(s):
%          case 0: z = P(0);
%          case 1: z = P(1); noflow =  true;
%           ...
%          case n: z = P(n); noflow =  true;
```

When there is a dependency of the output on the input, there are at least two different possible assignments to `z`. In this case, the only possible execution, as validated by our model, of the above program is the SC execution that reads `m` as `0` and leaves `noflow` untouched. When there is no dependency from input to output the value of `z` is the same, say `k` in all cases, non-zero by assumption. In this case, the compiler can hoist the assignment of to `z` outside the conditional and swap with the independent asignment to `hi`, thus rewriting T2 to:

```
T2: z=k; s=m; if (s!=0)  noflow=true;
% T2:  z = k;
%      s= m;
%      switch(s):
%      case 0: z = P(0);
%      case 1: noflow =  true;
%      ...
%      case n: noflow =  true;
```

In this case, the program has an execution, validated by our model, that sets `noflow` to `true`.

In the parlance of information flow, the humble conditional suffices to construct a composition operator to detect information flow in the presence of speculation.