A model of speculative evaluation

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

1 INTRODUCTION

2 MODEL

2.1 Preliminaries

We assume:

- 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 substitution, written $M[N/r]$,
- 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 substitution, written $\phi[N/\ell]$, and
- there are relations R and W $\subseteq (\mathcal{A} \times \mathcal{X} \times \mathcal{V})$,

where:

- the set of *lvalues* is $\mathcal{L} = (\mathcal{X} \cup \mathcal{R})$, ranged over by ℓ and k.

We shall say *a reads v from x* whenever $(a, x, v) \in$ R, and *a writes v to x* whenever $(a, x, v) \in$ W. 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.

2.2 Pomsets

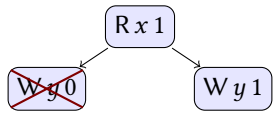
Definition 2.1. A *pomset* $(E, \leq, \lambda)$ with alphabet Σ is a partial order $(E, \leq)$ together with a function $\lambda : E \to \Sigma$.

Authors' addresses: Craig Disselkoen, University of California San Diego, cdisselk@eng.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/2018/5-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Going forward, we fix the alphabet $\Sigma = (\Phi \times \mathcal{A})$. We will write $(\phi \mid a)$ for the pair (ϕ, a), a for (true, a) and $\not{a}$ for (false, a).

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 $d \longrightarrow e$ corresponds to an ordering $d \leq e$. For example:



is a visualization of the pomset where:

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

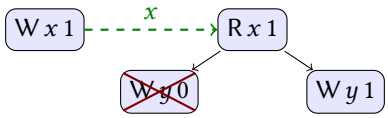
As we shall see, this is a possible execution of the program:

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

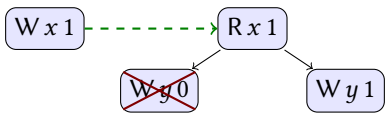
Definition 2.2. An *rf-pomset* is a pomset together with a $\mathsf{RF} \subseteq (E \times \mathcal{X} \times E)$ such that for any $(d, x, e) \in \mathsf{RF}$:

- $d < e$,
- $\lambda(d)$ writes v to x, and $\lambda(e)$ reads v from x, and
- there is no $d < c < e$ such that $\lambda(c)$ writes w to x.

We visualize rf-pomsets by drawing a dashed edge between nodes in RF, labelled with the memory location, for example:



In most cases, the memory location is obvious from context, so we elide it:



As we shall see, this is a possible execution of the program:

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

Definition 2.3. An rf-pomset is x-closed if for $e \in E$ with $\lambda(e) = (\phi \mid a)$:

- ϕ is independent of x, and
- if a reads v from x, then there is a d with $(d, x, e) \in \mathsf{RF}$.

2.3 Sets of pomsets

Let $\mathcal{P}_1 \sqcup \mathcal{P}_2$ be the set $\mathcal{P}_0$ where $P_0 \in \mathcal{P}_0$ whenever there are $P_1 \in \mathcal{P}_1$ and $P_2 \in \mathcal{P}_2$ such that:

- $E_0 = E_1 \cup E_2$,
- $\mathsf{RF}_0 = \mathsf{RF}_1 \cup \mathsf{RF}_2$,
- if $e \leq_1 d$ or $e \leq_2 d$ then $e \leq_0 d$,
- if $\lambda_0(e) = (\phi_0 \mid a)$ then either:
  - $\lambda_1(e) = (\phi_1 \mid a)$ and $\lambda_2(e) = (\phi_2 \mid a)$ and ϕ_0 implies $\phi_1 \vee \phi_2$,
  - $\lambda_1(e) = (\phi_1 \mid a)$ and $e \notin E_2$ and ϕ_0 implies ϕ_1, or
  - $\lambda_2(e) = (\phi_2 \mid a)$ and $e \notin E_1$ and ϕ_0 implies ϕ_2.

Let $\mathcal{P}_1 \parallel \mathcal{P}_2$ be defined the same as $\mathcal{P}_1 \sqcup \mathcal{P}_2$ except that:

- $\mathsf{RF}_0 \supseteq \mathsf{RF}_1 \cup \mathsf{RF}_2$, and for any $d, e \in E_i$, if $(d, e) \in \mathsf{RF}_0$ then $(d, e) \in \mathsf{RF}_i$.

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

- $E' = E \cup \{0\}$,
- $\mathsf{RF}' = \mathsf{RF}$,
- if $d \leq e$ then $d \leq' e$,
- $\lambda'(0) = (\phi, a)$, where ψ implies ϕ, and
- if $\lambda(e) = (\psi \mid b)$ then:
  - $\lambda'(e) = (\psi' \mid b)$,
  - ψ' implies $\psi[\vec{v}/\vec{x}]$, where a reads $\vec{v}$ from $\vec{x}$, and
  - $0 \leq' e$ or ψ' implies ψ.

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

- $E' = E$,
- $\mathsf{RF}' = \mathsf{RF}$,
- if $d \leq e$ then $d \leq' e$, and
- if $\lambda(e) = (\psi \mid a)$ then $\lambda'(e) = (\psi[M/\ell] \mid a)$.

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.

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

2.4 Semantics of programs

Define:

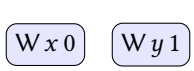
$$
\begin{aligned}
[\![\texttt{skip}]\!] &= \{\emptyset\} \\
[\![x := M; C]\!] &= \bigcup_v (M = v \mid \mathsf{W}\, x\, v) \to [\![C]\!][M/x] \\
[\![r := x; C]\!] &= [\![C]\!][x/r] \cup \bigcup_v (\mathsf{true} \mid \mathsf{R}\, x\, v) \to [\![C]\!][x/r] \\
[\![\texttt{if}\, M\, \texttt{then}\, C\,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

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:



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; skip)`. Now $[\![\texttt{skip}]\!]$ is just $\{\emptyset\}$, and $[\![y := r + 1; \texttt{skip}]\!]$ is:

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

which includes the case where v is 1:

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

which contains the pomset:

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

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

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

This 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:

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

Finally $[\![x := 0; r := x; y := r + 1; \texttt{skip}]\!]$ is:

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

which includes the case where v is 0:

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

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:

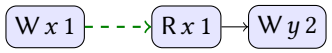
$$\boxed{\mathrm{W}\, x\, 0} \quad \boxed{\mathrm{W}\, y\, 1}$$

Note that $(\mathrm{W}\, x\, 0)$ does not read anything, and so 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 a write $x := M$, which performs a substitution $[M/x]$. This is a variant of the usual 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 the program `x := 1; || y := x+1;`. One execution of this program is where the write to y performs a concurrent read of x:



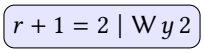
To see how this execution is modelled, we first expand out the syntax sugar to get the program `(x := 0; skip || r := x; y := r+1; skip)`. As before, $[\![y := r + 1;\ \texttt{skip}]\!]$ is:

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

which includes the case where v is 2:

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

which contains the pomset:



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

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

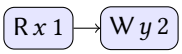
This has two cases, the sequential case (which does not introduce a read action) and the concurrent case (which does). We are now interested in the concurrent case, which is:

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

which includes the case where v is 1:

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

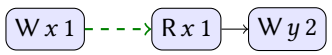
which contains the pomset:



Note that $(\mathrm{R}\, x\, 1)$ reads 1 from x, and while true implies $(x + 1 = 2)[1/2]$ it does *not* imply $(x + 1 = 2)$, and so there is an ordering $(\mathrm{R}\, x\, 1) < (\mathrm{W}\, y\, 2)$.

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

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



as expected.

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 the semantics:

$$\bigcup_v (1 = v \mid \mathrm{W}\, x\, v) \to \left(\bigcup_w (2 = w \mid \mathrm{W}\, y\, v) \to (\{\emptyset\})\,[2/y] \right)[1/x]$$

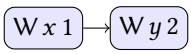
which is the same as:

$$\bigcup_v (1 = v \mid \mathrm{W}\, x\, v) \to \bigcup_w (2 = w \mid \mathrm{W}\, y\, v) \to \{\emptyset\}$$

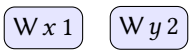
which includes the case where $v = 1$ and $w = 2$:

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

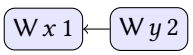
One of the executions this contains is:



but it also contains:



and:

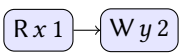


since there is no requirement that $(\mathrm{W}\, x\, 1) \leq (\mathrm{W}\, y\, 2)$.

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

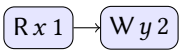
3.4 Independent reads and writes

Whereas write prefixing introduces no new dependencies, read prefixing can. For example in §3.2 we saw that the program (`r := x; y := r+1;`) includes the pomset:

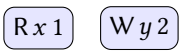


but since true does not imply $(x + 1 = 2)$, we have the requirement that $(\mathrm{R}\, x\, 1) \leq (\mathrm{W}\, y\, 2)$.

This is in contrast to the program (`r := x; y := r+2-r;`). Since true implies $(x+2-x = 2)[1/x]$, this contains:



but also true implies $(x + 2 - x = 2)$ (at least for integer arithmetic) and so this also contains:



Thus, the semantics of (`r := x; y := r+2-r;`) is the same as the semantics of (`y := 2; r := x;`).

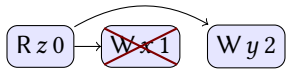
Note this this example shows that we are not just dealing with a syntactic notion of dependency, which is common in hardware models of memory. In syntactic dependency, since r occurs free in $(y := r + 2 - r)$, there would be a dependency between $(r := x)$ and $(y := r + 2 - r)$. In contrast, this model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

3.5 Control dependencies

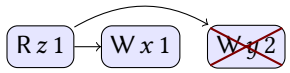
Conditionals introduce control dependencies, for example consider the program:

```
r := z; if (r) { x := 1; } else { y := 2; }
```

This includes executions in which the false branch is taken:

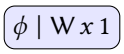


and ones where the true branch is taken:

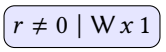


In both cases, we record the actions in the branch that was not taken. This is a novel feature of this model, and is intended to capture speculative evaluation. In §3.7 we will show how this model captures Spectre-like information flow attacks, once the attacker is provided with the ability to observe such speculations.

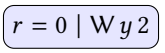
To see how these executions is modelled, consider the semantics of $[\![x := 1;\ \texttt{skip}]\!]$, which contains any pomset of the form:



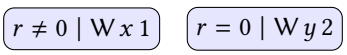
in particular it contains:



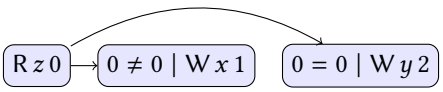
Similarly $[\![y := 2;\ \texttt{skip}]\!]$ contains:



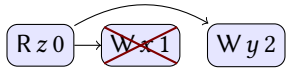
and so $[\![\texttt{if}\,(r)\{x := 1;\ \texttt{skip}\}\,\texttt{else}\,\{y := 2;\ \texttt{skip}\}]\!]$ contains:



Now, the semantics of concurrent read performs substitutions, for example a read of 0 from x results in:



which gives the required pomset:



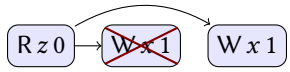
Note that the precondition $r = 0$ is dependent on r, and so there is a dependency $(\mathrm{R}\, z\, 0) < (\mathrm{W}\, y\, 2)$, modelling the control dependency introduced by the conditional.

3.6 Control independencies

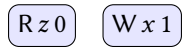
In most models of control dependencies, the dependency relation is syntactic, based on whether the action occurs inside syntactically inside a conditional. In contrast, the notion in this model is semantic: if an action can occur on both sides of a conditional, there is no control dependency. Consider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

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

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



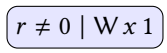
but it also has an execution in which the two writes of 1 to x are merged, resulting in no dependency:



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

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

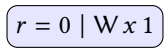
Now, one pomset in $\mathcal{P}_1$ is:



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:



that is P_2 where:

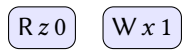
$$E_2 = \{e\} \quad \lambda_2(e) = (r = 0, \mathrm{W}\,x\,1)$$

Crucuially, in the definition of $\mathcal{P}_1 \sqcup \mathcal{P}_2$ 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:

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

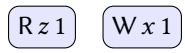
that is:

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:

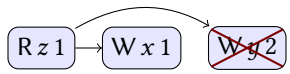


This semantics captures compiler optimizations which may, for example merge code executed on both branches of a conditional, or hoist constant assignments out of loops.

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



while programs such as `r := z; if (r) { x := 1; } 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 write a SPECTRE program as:

```
var a;
if (canRead(SECRET)) { a[SECRET] := 1; }
else if (touched a[0]) { x := 0; }
else if (touched a[1]) { x := 1; }
```

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:

```
if (canRead(x)) { ... code allowed to read x ... }
else { ... fallback code ... }
```

In this case, the `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.

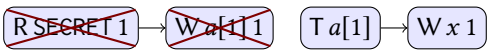
We model the `touched` test by introducing a new action $(\mathsf{T}\,x)$ and defining:

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

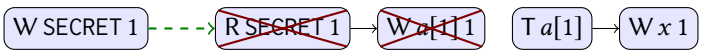
The additional requirement we need to add for x-closure is:

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

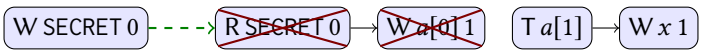
For example, one execution of SPECTRE is:



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 a (very abstract, untimed) model of Spectre attacks using speculative evaluation can be modeled.

3.8 Relaxed memory

The model includes concurrent memory accesses, which can introduce concurrent reads-from. For example, the program:

```
x := 1; || r := x;
```

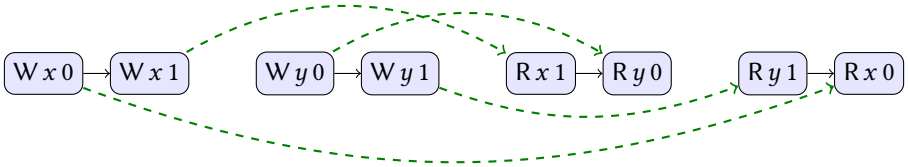
has an execution in which the write to x does not justify the read from x:

but also has an execution in which the write to x does justify the read from x:

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; }
```

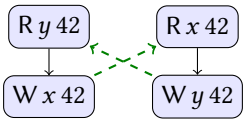
which includes the execution:



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

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

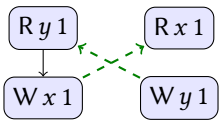
but an attempt to produce a value from thin air fails, for the usual reason of producing a cycle in $\leq$:



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.

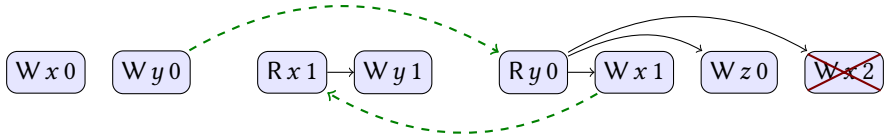
This model of relaxed memory is simple, and does not model many features such as coherence, non-release-acquire access, or memory fences, but it is good enough for the examples in this paper. In particular, we use the fact that relaxed memory is sensitive to data dependency to give a Spectre attack which does not depend on timing in §3.9.

3.9 Information flow attacks on relaxed memory

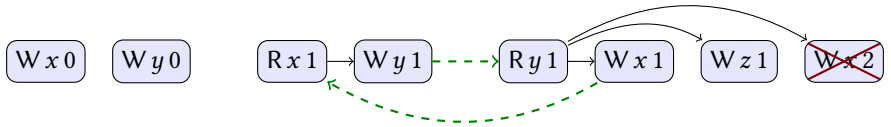
Consider an attacker program, again using security checks to try to learn a SECRET. Whereas SPECTRE uses hardware capabilities, which have to be modeled by adding extra capabilities to the language, this new attacker works by exploiting relaxed memory which can result in unexpected information flows.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

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

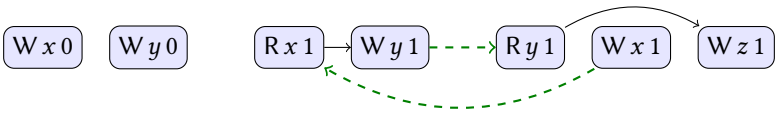
In the case where SECRET is 2, this has many executions, one of which is:



but there are no executions which exhibit (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 (R y 1) to (W x 1), which is what makes this execution possible. Thus, if the attacker sees an execution with (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:

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

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

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

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

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

and then perform independent read/write reordering:

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

After these optimizations, a sequentially consistent execution exhibits (W z 1).

3.10 Fences and release/acquire synchronization

We assume a subsets of release actions and acquire actions. Reads, writes and touches in neither release nor acquire.

Publication example:

```
var x; var f; x:=0; f:=0; (x:=1; f_rel:=1;  ||  r:=f_acq; s:=x;)
```

We disallow the execution where `r==1` and `s!=1`

We model release/acquire synchronization by introducing the following actions:

- $(\mathsf{W}_{\mathsf{rel}}\, x\, v)$ is a release action, which writes x at v.
- $(\mathsf{R}_{\mathsf{acq}}\, x\, v)$ is an acquire action, which reads x at v.

$$\begin{array}{rcl} [\![x_{\mathsf{rel}} := M;\, D]\!] & = & \bigcup_v (M = v \mid \mathsf{W}_{\mathsf{rel}}\, x\, v) \rightarrow [\![D]\!][M/x] \\ [\![r := x_{\mathsf{acq}};\, D]\!] & = & \bigcup_v (\mathsf{true} \mid \mathsf{R}_{\mathsf{acq}}\, x\, v) \rightarrow [\![D]\!][x/r] \end{array}$$

There are no additional requirements for x-closure is.

3.11 Transactions

We present a model of weakly isolated transactions. To get strong isolation, we need to make B record the reads, symmetrically to C; we also need to require in parallel composition that any order in-to/out-of a transactional event be lifted to the corresponding B/C

```
var x; var f; x:=0; f:=0; fence;
   x:=1; (begin; f:=1; f:=2; end;) || (begin; r:=f; end; s:=x;)
```

`end` ; D is sugar for `if commit` $\vec{x}$ `then` D `else` D.

- (B) is an acquire action
- $(\mathsf{C}\, \vec{x}\, \vec{v})$ is a release action which writes $\vec{x}$ at $\vec{v}$

$$\begin{array}{rcl} [\![\texttt{begin}\ ;\, D]\!] & = & \mathit{atomic}((\mathsf{true} \mid \mathsf{B}) \rightarrow [\![D]\!]) \\ [\![\texttt{if commit}\ \vec{x}\ \texttt{then}\ D_1\ \texttt{else}\ D_2]\!] & = & \bigcup_{\vec{v},\, \phi \text{ implies } \vec{x}=\vec{v}} ((\phi \mid \mathsf{C}\, \vec{x}\, \vec{v}) \rightarrow (\phi \mid [\![D_1]\!])) \sqcup (\neg\phi \mid [\![D_2]\!]) \end{array}$$

Pomset is *atomic* if

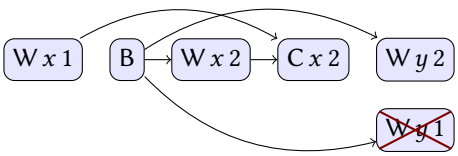
- for every e, where $\lambda(e) = (\phi \mid \_)$,
  if $d < e$ with $\lambda(d) = (\_ \mid \mathsf{B})$ and there is no $d < \mathsf{C} < e$
  then ϕ implies $\bigcup_{(\psi \mid \mathsf{C}) \text{ ends } d} \psi$
- if two reads of the same location see different values occur between begin and commit, then the formula on the commit implies false
- STRONG ISO: order from e to transaction must be lifted to begin
- STRONG ISO: order from transaction to e must be lifted to commit
- WEAK ISO: order from transactional e to transaction must be lifted to begin
- WEAK ISO: order from transaction to transactional e must be lifted to commit

Single threaded example: `B1 C1 B2 C2`, we have `C1<B2` since `C1<C2` and we must lift

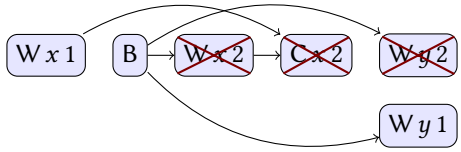
For example, the semantics of

```
x:=1; begin; x:=2; end; y:=x;
```

includes



and



4 EXPERIMENTS

5 CONCLUSIONS