# 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

The model used in this paper is one of pomsets with event labels of the form $(\phi \mid a)$, where $\phi$ 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:

$$\boxed{M = v \mid \mathrm{W}\, x\, v}$$

For this reason, the model is parameterized by a logic, used to express the preconditions of actions. We make few requirements of this logic, save that it has includes equalities between expressions, is closed under substitution, and supports a notion of implication (and in particular a notion of when a formula is a tautology).

The semantics is defined compositionally. For example, the program $(x := y + 1)$ is shorthand for $(r := y\,;\, x := r + 1)$, which contains the pomset:

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

This pomset is build compositonally. First, $[\![x := r + 1]\!]$ contains:

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

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

$$\boxed{1 = 1 \mid \mathrm{W}\, x\, 2}$$

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

$$\boxed{\mathrm{W}\, x\, 2}$$

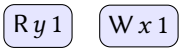
This substitution is performed in defining $[\![r := y\,;\, x := r + 1]\!]$, which contains the pomset:

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

Authors' addresses: Craig Disselkoen, University of California San Diego, 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/2018/6-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

as required. There is an ordering $(\mathsf{R}\, y\, 1) < (\mathsf{R}\, x\, 2)$ 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 $r$.

### 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* $\Phi$, ranged over by $\phi$ and $\psi$, 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 $\ell$ and $k$.

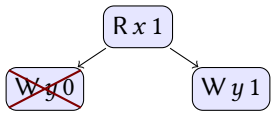
We shall say *a reads $v$ from $x$* whenever $(a, x, v) \in \mathsf{R}$, and *a writes $v$ to $x$* whenever $(a, x, v) \in \mathsf{W}$. In examples, the actions are of the form $(\mathsf{R}\, x\, v)$, which reads $v$ from $x$, and $(\mathsf{W}\, x\, v)$, which writes $v$ to $x$.

### 2.2 Pomsets

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

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

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




is a visualization of the pomset where:

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

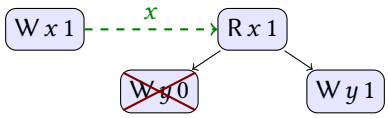
As we shall see, this is a possible execution of the program:

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

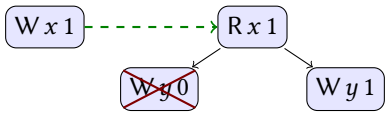
*Definition 2.2.* An *rf-pomset* is a pomset together with a $\mathsf{RF} \subseteq (E \times \mathcal{X} \times E)$ such that for any $(d, x, e) \in \mathsf{RF}$:

- $d < e$,
- $\lambda(d)$ 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 $\lambda(e) = (\phi \mid a)$:

- $\phi$ 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 $\phi_0$ implies $\phi_1 \vee \phi_2$,
  - $\lambda_1(e) = (\phi_1 \mid a)$ and $e \notin E_2$ and $\phi_0$ implies $\phi_1$, or
  - $\lambda_2(e) = (\phi_2 \mid a)$ and $e \notin E_1$ and $\phi_0$ implies $\phi_2$.

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

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

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 $\phi$ implies $\psi$.

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:

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

## 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:

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

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

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

which includes the case where $v$ is 1:

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

which contains the pomset:

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

expressing that this program can write 1 to $y$, as long as the precondition $(r + 1 = 1)$ is satisfied. Now $[\![r := x; y := r + 1; \texttt{skip}]\!]$ is:

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

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:

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

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

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

which includes the case where $v$ is 0:

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

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:

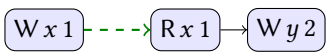
$$\boxed{\mathrm{W}\, x\, 0} \quad \boxed{\mathrm{W}\, y\, 1}$$

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 $\phi$ 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 $\phi$ 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; || 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; skip || r := x; y := r+1; skip)`. As before, $[\![y := r + 1;\ \texttt{skip}]\!]$ is:

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

which includes the case where $v$ is 2:

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

which contains the pomset:

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

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

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

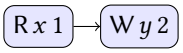
This has two cases, the sequential case (which does not introduce a read action) and the concurrent case (which does). We are now interested in the concurrent case, which is:

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

which includes the case where $v$ is 1:

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

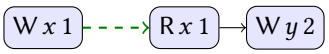
which contains the pomset:




Note that $(\mathrm{R}\, x\, 1)$ reads 1 from $x$, and while true implies $(x + 1 = 2)[1/x]$ (since $1 + 1 = 2$ is a tautology) it does *not* imply $(x + 1 = 2)$ (which is not true for every value of $x$), and so there is an ordering $(\mathrm{R}\, x\, 1) < (\mathrm{W}\, y\, 2)$ modelling the data dependency of the write of $y$ on the read of $x$.

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


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




as expected, including an rf-dependency between the concurrent write of $x$ and the matching read.

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:

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

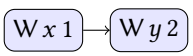
which is the same as:

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

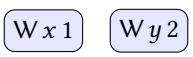
which includes the case where $v = 1$ and $w = 2$:

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

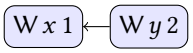
One of the executions this contains is:




but it also contains:




and:

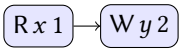



since there is no requirement that $(\mathrm{W}\, x\, 1) \leq (\mathrm{W}\, y\, 2)$.

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

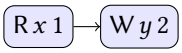
### 3.4 Independent reads and writes

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:

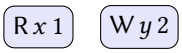



but since true does not imply $(x + 1 = 2)$, we have the requirement that $(\mathrm{R}\, x\, 1) \leq (\mathrm{W}\, y\, 2)$.

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;)`.

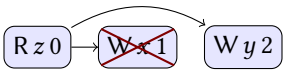
Note this this example shows that we are not just dealing with a syntactic notion of dependency, which is common in hardware models of memory. In syntactic dependency, since $r$ occurs free in $(y := r + 2 - r)$, there would be a dependency between $(r := x)$ and $(y := r + 2 - r)$. In contrast, this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

### 3.5 Control dependencies

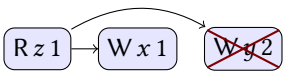
Conditionals introduce control dependencies, for example consider the program:

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

This includes executions in which the false branch is taken:

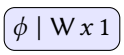



and ones where the true branch is taken:

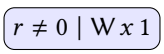



In both cases, we record the actions in the branch that was not taken. This is a novel feature of this model, and is intended to capture speculative evaluation. In §3.7 we will show how this model captures Spectre-like information flow attacks, once the attacker is provided with the ability to observe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

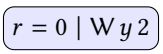
To see how these executions are modelled, consider the semantics of $[\![x := 1;\ \mathtt{skip}]\!]$, which contains any pomset of the form:



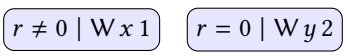

in particular it contains:



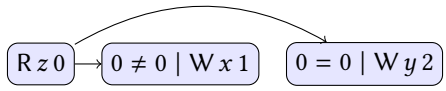

Similarly $[\![y := 2;\ \mathtt{skip}]\!]$ contains:



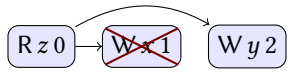

and so $[\![\mathtt{if}\,(r)\{x := 1;\ \mathtt{skip}\}\,\mathtt{else}\,\{y := 2;\ \mathtt{skip}\}]\!]$ contains:



Now, the semantics of concurrent read performs substitutions, for example a read of 0 from $z$ into $r$ results in:




which gives the required pomset:



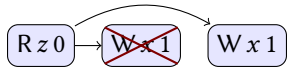

Note that the precondition $r = 0$ is dependent on $r$, and so there is a dependency $(\mathrm{R}\, z\, 0) < (\mathrm{W}\, y\, 2)$, modelling the control dependency introduced by the conditional.

### 3.6 Control independencies

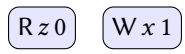
In most models of control dependencies, the dependency relation is syntactic, based on whether the action occurs inside syntactically inside a conditional. In contrast, the notion in this model is semantic: if an action can occur on both sides of a conditional, there is no control dependency. Consider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



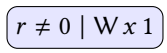

but it also has an execution in which the two writes of 1 to $x$ are merged, resulting in no dependency:




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

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

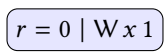
Now, one pomset in $\mathcal{P}_1$ is:




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:

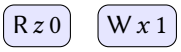
$$E_2 = \{e\} \quad \lambda_2(e) = (r = 0, \mathrm{W}\, x\, 1)$$

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:

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

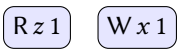
that is:

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:

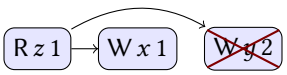



This semantics captures compiler optimizations which may, for example merge code executed on both branches of a conditional, or hoist constant assignments out of loops.

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 $(\mathsf{W}\, x\, 1)$ is independent of $(\mathsf{R}\, z\, 1)$:




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




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.

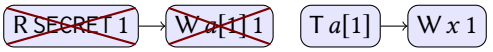
We model the `touched` test by introducing a new action $(\mathsf{T}\, x)$ and defining:

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

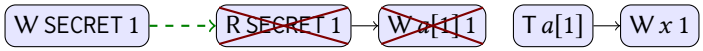
The additional requirement we need to add for $x$-closure is:

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

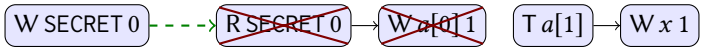
For example, one execution of SPECTRE is:



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.

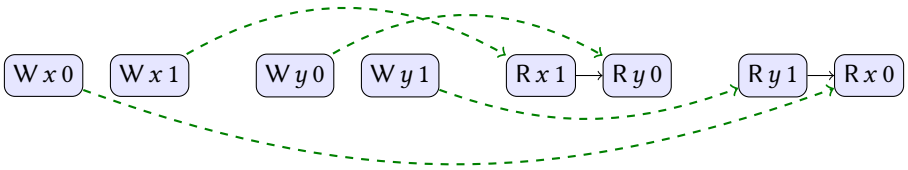
This shows how our model of speculative evaluation can express (very abstract, untimed) Spectre attacks.

### 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 evalutation. 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; }
```

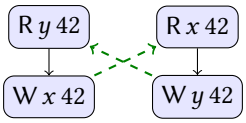
which includes the execution:




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

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

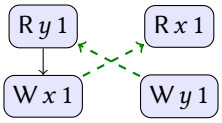
but an attempt to produce a value from thin air fails, for the usual reason of producing a cycle in $\leq$:




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:



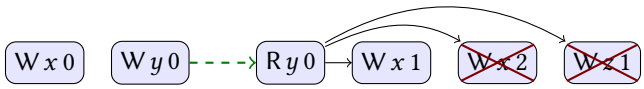

Note that $(\mathsf{R}\, x\, 1) \not\leq (\mathsf{W}\, y\, 1)$, so this does not introduce a cycle.

### 3.9 Information flow attacks on relaxed memory

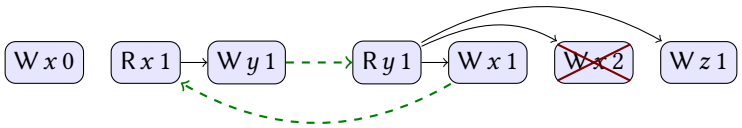
Consider an attacker program, again using security checks to try to learn a SECRET. Whereas SPECTRE uses hardware capabilities, which have to be modeled by adding extra capabilities to the language, this new attacker works by exploiting relaxed memory which can result in unexpected information flows.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

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

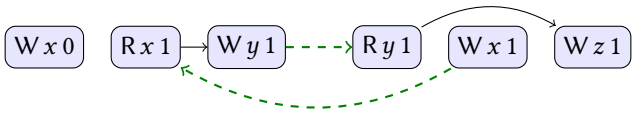
In the case where SECRET is 2, this has many executions, one of which is:




but there are no executions which exhibit (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:

```
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:

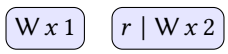
```
x := 1;
if (y == 0) { }
else if (canRead(SECRET)) { }
else { z := 1; }
```

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

### 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. For example, in the program `(x := 1; x := 2)`, the first write to `x` can be eliminated, since the second is guaranteed:


but in the program `(x := 1; if (r) { x := 2; })` the first write cannot be eliminated, since the second might not happen:




A simple model of dead store elimination changes the semantics of write to only introduce a write event if there is no subsequent guaranteed write of the same variable. We can do this by giving a looser interpretation of prefixing: in the definition of $(\phi \mid a) \to \mathcal{P}$, replace the requirement:

- $\lambda'(0) = (\psi, a)$, where $\psi$ implies $\phi$,

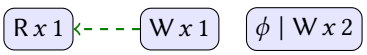
by:

- either:
  - $\lambda'(0) = (\psi, a)$, where $\psi$ implies $\phi$, or
  - $\lambda(0) = (\psi, b)$, where $\phi$ implies $\psi$, and every location $a$ writes to is also written to by $b$.

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:



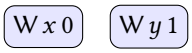

where $\phi$ is (`¬canRead(SECRET)`), which is not a tautology, and so the (W $x$ 1) event is not omitted. In the case that `SECRET` is not 0, the matching execution is:


Now the (W $x$ 2) event is a guaranteed write, so the (W $x$ 1) is omitted. 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.

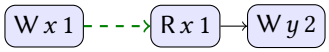
### 3.11 Thread inlining

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:

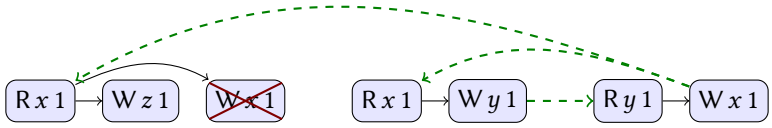



That is in the sequential case there is no depenency from the write of $x$ to the write of $y$, but in the concurrent case there is such a dependecy.

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;
```

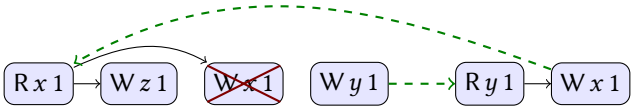
This has no execution containing (W $z$ 1). Any attempt to build such an execution results in a cycle:




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

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

with execution:



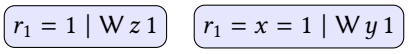

To see why this execution exists, consider the program fragment:

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

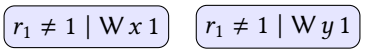
Removing the syntax sugar, this is:

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

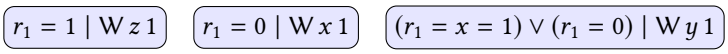
Now, $[\![z := 1; r2 := x; y := r2; \texttt{skip}]\!]$ includes pomset:



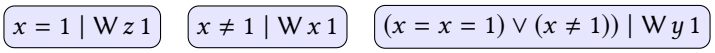
and $[\![x := 1; r3 := x; y := r3; \texttt{skip}]\!]$ includes pomset:



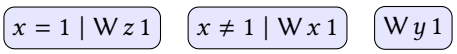

so $[\![\texttt{if}\,(r_1 = 1)\{z := 1; r2 := x; y := r2; \texttt{skip}\}\,\texttt{else}\,\{x := 1; r3 := x; y := r3; \texttt{skip}\}]\!]$ includes:



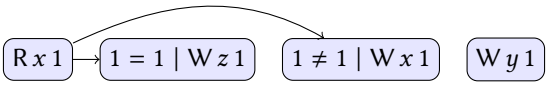

which means $[\![\texttt{if}\,(r_1 = 1)\{z := 1; r2 := x; y := r2; \texttt{skip}\}\,\texttt{else}\,\{x := 1; r3 := x; y := r3; \texttt{skip}\}]\!][x/r_1]$ includes:



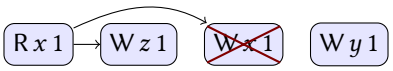

Now $(x = x = 1) \vee (x \neq 1)$ is a tautology, so this is just:




and so $[\![r_1 := x; \texttt{if}\,(r_1 = 1)\{z := 1; r2 := x; y := r2; \texttt{skip}\}\,\texttt{else}\,\{x := 1; r3 := x; y := r3; \texttt{skip}\}]\!]$ 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.

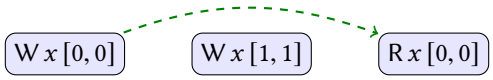
### 3.12 Word tearing

In §**??**, we shall be considering transactional memory, and in §**??** show that we can model a simplified version of an information flow attack on transactions. In order to model transactions, we need to consider actions that can write many memory locations at once, since this is part of the semantics of commitment. To lead up to this, we first consider a simpler scenario of many-location writes and reads, which is word tearing.

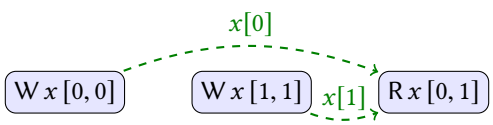
In word tearing, a program contains a write instruction with data larger than the hardware word size, for example copying a byte array, or assigning a 64-bit float on a 32-bit architecture. For example, consider the program:

```
(x := [0, 0];) || (x := [1, 1];) || (r := x;)
```

This has executions in which the read of $x$ only reads from one of the writes, for example:




but also has executions in which the read of $x$ reads from both writes, for example:



Word tearing can occur, for example, in Java extended floating point [**?** ], LLVM 64-bit instructions on 32-bit hardware [**?** ], or in JavaScript SharedArrayBuffers [**?** ].

### 3.13 Fences and release/acquire synchronization

Another feature that transactions have is that they act as memory fences, that is they are a barrier to reordering memory accesses. In this section, we present a simple model of fencing, and show in §**??** that it 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 aquire event is one labelled by an aquire action. In examples, we will use releasing writes and aquiring reads:

- $(\mathrm{W}_{\mathrm{rel}}\, x\, v)$, a release action that writes $v$ to $x$, and
- $(\mathrm{R}_{\mathrm{acq}}\, x\, v)$, an acquire action that reads $v$ from $x$.

The semantics of fences are given by adding extra constraints to the definition of $(\phi \mid a) \rightarrow \mathcal{P}$:

- $0 \leq e$ whenever 0 is an aquire event or $e$ is a release event.

The semantics of programs with acquiring read and releasing write are the same as for regular read and write, but using the appropriate actions:

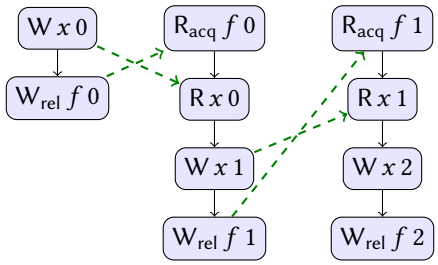
$$[\![x_{\mathrm{rel}} := M; C]\!] = \bigcup_v (M = v \mid \mathrm{W}_{\mathrm{rel}}\, x\, v) \rightarrow [\![C]\!][M/x]$$

$$[\![r_{\mathrm{acq}} := x; C]\!] = [\![C]\!][x/r] \cup \bigcup_v (\mathsf{true} \mid \mathrm{R}_{\mathrm{acq}}\, x\, v) \rightarrow [\![C]\!][x/r]$$

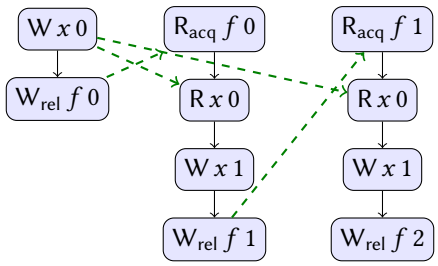
For example, consider the program:

```
x := 0; rel f := 0; ||
acq r := f; if (r == 0) { x := x+1; rel f := 1; } ||
acq s := f; if (r == 1) { x := x+1; rel f := 2; }
```

This has an execution:




but *not*:




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

### 3.14 Transactions

We model transactions by introducing the actions:

- (B), an acquire action, and
- (C $\vec{x}$ $\vec{v}$), a release action that writes $\vec{v}$ to $\vec{x}$,

with semantics:

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

where we require that all pomsets in the semantics of programs be *atomic*.

Before defining atomicity, we provide some auxiliary notation.

We say that $e$ is a *begin event* if $\lambda(e) = (\phi \mid \mathrm{B})$ and a *commit event* if $\lambda(e) = (\phi \mid \mathrm{C}\, \vec{x}\, \vec{v})$.

We write $\phi_e$ for the formula and $a_e$ for the action of $e$; that is, when $\lambda(e) = (\phi_e \mid a_e)$.

We say that $e$ is *compatible with* $d$ when $\phi_e \wedge \phi_d$ is satisfiable.

We say that event $e$ *belongs to* $(b, \vec{c})$ when

- $b$ is a begin event, $b < e$, and there is no commit event $d$ such that $b < d < e$,
- $\vec{c}$ are the commit events $c_i$ such that $e < c_i$ and there is no begin event $d$ such that $e < d < c_i$.

*Definition 3.1.* A pomset is *atomic* when for any $e$ that belongs to $(b, \vec{c})$:

(1) $\phi_e$ implies $\bigvee_i \phi_{c_i}$,
(2) if $d < e$ then $d < b$,
(3) if $e < d$ and $d$ is compatible with $c_i$ then $c_i < d$,
(4) if $e' \neq e$ belongs to $(b', \vec{c}')$ but not $(b, \_)$ and
   - $c'_j$ is compatible with $e$, and
   - $c_i$ is compatible with $e'$

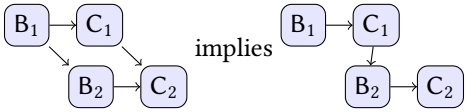
   then either $c'_j < b$ or $c_i < b'$,
(5) if $e$ writes $x$ and $c_i$ writes $\vec{x}$ then $x = x_i$, for some $i$, and
(6) if $e$ reads $x$ and $e' \neq e$ reads $x$, belongs to $(b, \_)$ and is compatible with $e$ then $a_e = a_{e'}$.

Clause (1) requires that the precondition on $e$ is false on an aborted transaction. The *lifting clauses*, (2) and (3), require order come in or out of $e$ is lifted to the corresponding begin or commit event. Clause (4) requires that transactions be totally ordered. Clause (5) requires that all writes be committed. Clause (6) requires that multiple reads of a location in a single transaction must see the same value.

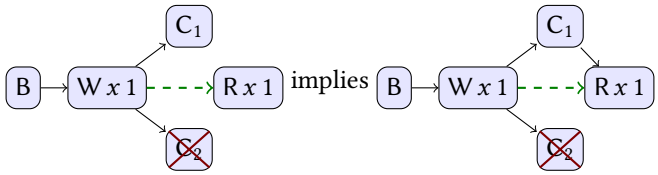
The definition of atomicity guarantees strong isolation. For weak isolation, clauses (5) and (6) are unnecessary, (2) only applies when $d$ is a commit, and (3) only applies when $d$ is a begin.

The definition handles simple examples:

- Single threaded example: $\mathrm{B}_1\mathrm{C}_1\mathrm{B}_2\mathrm{C}_2$. Because $\mathrm{C}_2$ is a release, we know that $\mathrm{C}_1 < \mathrm{C}_2$. Because $\mathrm{B}_1$ is an acquire, we know that $\mathrm{B}_1 < \mathrm{B}_2$ By lifting, either of these is sufficient to require that $\mathrm{C}_1 < \mathrm{B}_2$.



- Abort example:



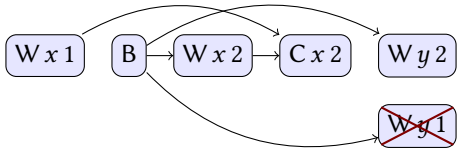

- Clause (??) stops transaction from reading two different values for the same variable from transactions (it is possible with no transactional writes).
- Clause (??) also stops transactional IRIW.

Let "`end; D`" be syntax sugar for "`if commit` $\vec{x}$ `then` $D$ `else` $D$", where $\vec{x}$ are the free variables of $D$.

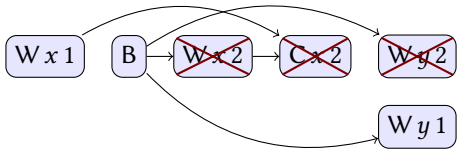
The semantics of

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

includes




and




Publication example:

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

### 3.15 Picky details

The relations R and W and RF should be restricted to elements of $(\mathcal{A} \times \mathcal{X} \times \mathcal{V})$ that are in $(\mathcal{A} \times (\mathcal{X} \to \mathcal{V}))$.

$(\mathsf{T}\, x)$ neither reads nor writes $x$.

## 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 [**?** ]. Here we demonstrate the efficacy of each of these attacks against modern compilers and hardware, including both the `clang` and `gcc` compilers.

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

### 4.1 Relaxed memory attack

### 4.2 Dead store elimination attack

In this section we return to the attack in Section 3.10 based on dead store elimination.

As in Section 4.1, 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 arbitrary code subject to the above restrictions. Despite the attacker's apparent inability to access `SECRET`, we show that the attacker can learn its value using the idea in Section 3.10. This attack was tested and works on both `clang` and `gcc`.

First, 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. To do this, we simply compile `N` copies of the function above, 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 second thread. The extension 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 values of `r`. First, we insert additional time-consuming computation immediately following the `x := 1` operation. This lengthens the timing window in which `x` has the value 1, increasing the likelihood that the other 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), but with `clang`, the computation must be branch-free, and furthermore 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`.

Second, 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.

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.

Our implementation has two important "knobs" which trade off reliability vs. performance. 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 CONCLUSIONS