

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

This paper studies information flow caused by speculation mechanisms in hardware and software in the execution of programming languages.

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

A key parameter in the definitions alluded to above is the notion of observable that delimits the attacker model. Whereas the classical input-output behavior is an adequate foundation for compiler transformations, it has long been known that side-channels that leak information arise from other observables such as execution time and power consumption.

The focus of this paper is the development of a semantic model of execution in programming languages to explicate side channel attacks that arise from speculation. Let program order (po) stand for the total order on the actions of each thread. An execution is speculative when it does not respect the program order(po), i.e., there are events $e_1 \leq_{po} e_2$ such that e_2 appears to be executed before e_1 .

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

- Pipelined microprocessors use predictive schemes to utilize resources more efficiently and improve performance. Informally, these schemes use the prior history to predict the path of a program, e.g. by guessing memory dependencies, or the outcome of conditionals. Execution proceeds along the predicted path until it is possible to validate the prediction; the execution is committed if the prediction is correct; otherwise, the execution is rolled back. The speculation does not affect the observable input-output behavior of the program, thus ensuring correctness with respect to the usual intended semantics of the program. However, since there is clear timing differences between the cases where the prediction is and is not successful, it raises the concern that the predictive mechanisms themselves could cause information-leaks via

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.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

© 2018 Copyright held by the owner/author(s).

XXXX-XXXX/2018/6-ART

<https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>

timing side-channels; a fear that is realized with devastating impact by the Spectre family of attacks [?].

- Several modern microprocessors [?] support transactions to aid in the design of correct and efficient concurrent programs. Transactions are executed optimistically; they are committed if there are no conflicts, and aborted otherwise. All memory effects of an aborted transaction are rolled back; so there is no way for a concurrent observer to detect an aborted transaction. However, the thread of the aborted transaction, via abort-handler code, gets notified of the cancellation of the transaction.

Thus, when a transaction of a thread aborts, the thread can deduce plausible information about specific memory accesses of a concurrently executing transaction. In combination with the techniques used in Spectre, this potential has been exploited recently to accelerate and scale up the efficacy of the attacker in the Spectre family of attacks [?].

- A hardware relaxed memory model prescribes a recipe to generate *preserved program order* (ppo) from po [?]; ppo is a suborder of po that removes order between actions that commute according to the architecture. For example, all hardware models (e.g., [?]) permit the commutation of writes to independent locations that are adjacent in po.
- In the memory models of Java and C++[?], the programmer is able to rely on po only when the program is race free. If the program has races, the JMM forces the programmer to account for speculative execution by allowing a concurrent observer to detect violations of po.

The Spectre examples motivate the main technical development of this paper, namely a semantic execution models of programming languages that capture enough detail to reveal and analyze the presence of side channels revealed by speculative execution. In these cases, information leaks are via an (unintended!) side-channel.

In contrast, relaxed memory – by design – results in information leaks that can be observed in the input-output behavior of the program. As a happy, and arguably suprising consequence, we demonstrate – hitherto unknown – information leaks in other uses of speculative execution, namely relaxed memory.

We illustrate with a simple example whose reasoning is validated by our model. In the parlance of information flow, the humble conditional suffices to construct a composition operator to detect information flow [?] in the presence of speculation. Let $P(h_i)$ be a function with domain $\{0, \dots, n\}$ and codomain $\{1, \dots, n\}$. Our aim is to write a program that tests whether there is information flow from input h_i to the return value lo . Consider:

[Initial value of m , flow is 0.]

T1: $r = z$; $m = r$;

T2: $s = m$;

switch(s):

case 0: $z = P(0)$;

case 1: $z = P(1)$; flow = 1;

...

case n : $z = P(n)$; flow = 1;

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

```

T2: z = k;
s = m;
switch(s):
case 0: z = P(0);
case 1: flow = 1;
...
case n: flow = 1;

```

In this case, the program has an execution, validated by our model, that sets flow to 1.

2 MODEL

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

$$M = v \mid W x v$$

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 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:

$$R y 1 \rightarrow W x 2$$

This pomset is build compositionally. First, $\llbracket x := r + 1 \rrbracket$ contains the pomset:

$$r = 1 \mid W x 2$$

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

$$1 = 1 \mid W x 2$$

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

$$W x 2$$

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

$$R y 1 \rightarrow W x 2$$

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

$$R y 1 \quad W x 1$$

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

The main novelty of our semantics, since it is designed to model speculative evaluation, is in modelling conditionals. In most sets-of-pomsets semantics, a pomset in $\llbracket \text{if } (M) \{ C \} \text{ else } \{ D \} \rrbracket$ would either be given by a pomset in $\llbracket C \rrbracket$ or a pomset in $\llbracket D \rrbracket$. To model speculative evaluation,

we need to allow a pomset in $\llbracket \text{if } (M) \{ C \} \text{ else } \{ D \} \rrbracket$ to be given by both a pomset in $\llbracket C \rrbracket$ and a pomset in $\llbracket D \rrbracket$. For example, $\llbracket \text{if } (M) \{ x := 1 \} \text{ else } \{ x := 2 \} \rrbracket$ contains the pomset:

$$M \neq 0 \mid W x 1 \quad M = 0 \mid W x 2$$

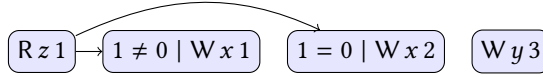
that is we have recorded behaviour from both branches of execution. Moreover, an action which is performed on both sides of the conditional can be merged, producing only one event in the resulting pomset. The precondition of the merged event is the disjunction of the preconditions of the original events.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

$$M \neq 0 \mid W x 1 \quad M = 0 \mid W x 2 \quad (M \neq 0) \vee (M = 0) \mid W y 3$$

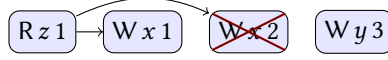
and since $(M \neq 0) \vee (M = 0)$ is a tautology, this is:

$$M \neq 0 \mid W x 1 \quad M = 0 \mid W x 2 \quad W y 3$$

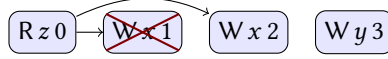
Combining this model of conditionals with the model of memory using substitutions gives that $\llbracket \text{if } (z) \{ x := 1; y := 3 \} \text{ else } \{ x := 2; y := 3 \} \rrbracket$ contains the pomset:



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



Similarly, $\llbracket \text{if } (z) \{ x := 1; y := 3 \} \text{ else } \{ x := 2; y := 3 \} \rrbracket$ contains the pomset:



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

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

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

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

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

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

2.1 Data models

A *data model* consists of:

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

- a set of *actions* \mathcal{A} , ranged over by a and b ,

such that:

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

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

2.2 Pomsets

1

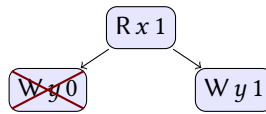
Recall the definition of a pomset from [?]:

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

Going forward, we fix the alphabet $\Sigma = (\Phi \times \mathcal{A})$. We will write $(\phi \mid a)$ for the pair (ϕ, a) , elide ϕ when ϕ is a tautology, and write \bot when ϕ is unsatisfiable. We lift terminology from logical formulae and actions to events, for example if $\lambda(e) = (\phi \mid a)$ and $\lambda(d) = (\psi \mid b)$ then we say:

- e implies d whenever ϕ implies ψ ,
- e is a tautology whenever ϕ is a tautology,
- e is unsatisfiable whenever ϕ is unsatisfiable,
- e is independent of x whenever ϕ is independent of x ,
- e writes v to x whenever a writes v to x , and
- e reads v from x whenever a reads v from x .

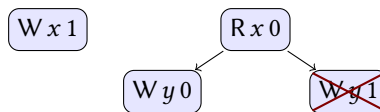
We visualize a pomset as a graph where the nodes are drawn from E , each node e is labelled with $\lambda(e)$, and an edge $d \rightarrow e$ corresponds to an ordering $d \leq e$. For example:



is a visualization of the pomset where:

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

Pomsets by themselves do not capture which read events have a matching write. This is necessary for a compositional concurrent semantics, for example a program whose semantics includes:

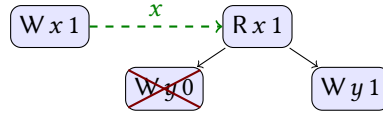


may be put in parallel with another program which writes 0 to x . If x is scoped, though, such a program cannot exist, so we need each read of x to have a matching write. This is captured by the *reads-from* relation [?], which we adapt to our setting.

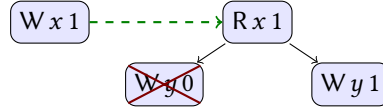
Definition 2.2. An *rf-pomset* is a pomset together with a $\text{RF} \subseteq (E \times \mathcal{X} \times E)$ such that for any $(d, x, e) \in \text{RF}$:

- $d < e$,
- e implies d ,
- d writes v to x , and e reads v from x , and
- there is no $d < c < e$ such that e implies c and 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:



In the case where x is scoped, every read of x , must have a matching write, and there can be no preconditions dependent on x :

Definition 2.3. An rf-pomset is *x-closed* if for every $e \in E$:

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

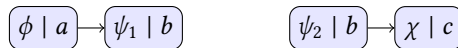
2.3 Sets of pomsets

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

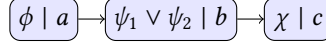
Definition 2.4. Let $P_0 \in (\mathcal{P}_1 \sqcup \mathcal{P}_2)$ whenever there are $P_1 \in \mathcal{P}_1$ and $P_2 \in \mathcal{P}_2$ such that:

- $E_0 = E_1 \cup E_2$,
- $\text{RF}_0 = \text{RF}_1 \cup \text{RF}_2$,
- 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 .

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



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



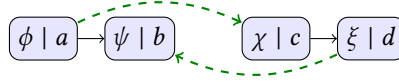
Definition 2.5. Let $P_0 \in (\mathcal{P}_1 \parallel \mathcal{P}_2)$ whenever there are $P_1 \in \mathcal{P}_1$ and $P_2 \in \mathcal{P}_2$ such that:

- $E_0 = E_1 \cup E_2$, for disjoint E_1 and E_2 ,
- $\text{RF}_0 \supseteq \text{RF}_1 \cup \text{RF}_2$, and for any $d, e \in E_i$, if $(d, e) \in \text{RF}_0$ then $(d, e) \in \text{RF}_i$.
- if $e \leq_1 d$ or $e \leq_2 d$ then $e \leq_0 d$,
- if $\lambda_0(e) = (\phi_0 | a)$ then either:
 - $\lambda_1(e) = (\phi_1 | a)$ and $e \notin E_2$ and ϕ_0 implies ϕ_1 , or
 - $\lambda_2(e) = (\phi_2 | a)$ and $e \notin E_1$ and ϕ_0 implies ϕ_2 .

Parallel composition is very similar to join, with two important differences: it can add rf edges between the two sides, and there is no merging of events. For example, if \mathcal{P}_1 and \mathcal{P}_2 contain:



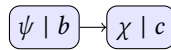
and a writes v to x , which c reads, and d writes w to y , which b reads, then $\mathcal{P}_1 \sqcup \mathcal{P}_2$ contains:



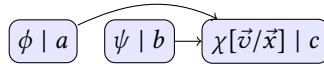
Definition 2.6. Let $(\phi | a) \rightarrow \mathcal{P}$ be the set \mathcal{P}' where $P' \in \mathcal{P}'$ whenever there is $P \in \mathcal{P}$ such that:

- $E' = E \cup \{c\}$,
- $\text{RF}' = \text{RF}$,
- if $d \leq e$ then $d \leq' e$,
- $\lambda'(c) = (\psi, a)$, where ψ implies ϕ , and
- if $\lambda(e) = (\psi' | b)$ then:
 - $\lambda'(e) = (\psi' | b)$,
 - ψ' implies $\psi[\vec{v}/\vec{x}]$, where a reads \vec{v} from \vec{x} , and
 - if ψ is dependent on \vec{x} or there is some location that both a and b write to, then $c \leq' e$.

Prefixing is used to define the semantics of reads and writes, and adds a new event c with action a and a precondition which implies ϕ . It performs a substitution on the preconditions of any subsequent events. The tricky part of the definition is the last clause, which places requirements on $c \leq e$. In normal prefixing, this would always be required, but we only require it when either the precondition of e depends on one of the variables read by a , or if c and e write to the same variable (this condition is called *per-location program order* in [?], and simplifies the treatment of some of the examples). For example, if \mathcal{P} contains:



where a reads \vec{v} from \vec{x} , b does not write to a 's write locations, and ψ is independent of \vec{x} , then $(\phi | a) \rightarrow \mathcal{P}$ contains:



Definition 2.7. Let $\mathcal{P}[M/x]$ be the set \mathcal{P}' where $P' \in \mathcal{P}'$ whenever there is $P \in \mathcal{P}$ such that:

- $E' = E$,
- $\text{RF}' = \text{RF}$,
- if $d \leq e$ then $d \leq' e$, and

- if $\lambda(e) = (\psi \mid a)$ then $\lambda'(e) = (\psi[M/x] \mid a)$.

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

Definition 2.8. Let $(\phi \mid \mathcal{P})$ be the subset of \mathcal{P} such that $P \in \mathcal{P}$ whenever:

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

Definition 2.9. Let $(\nu x . \mathcal{P})$ be the subset of \mathcal{P} such that $P \in \mathcal{P}$ whenever P is x -closed.

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

Definition 2.10.

$$\begin{aligned}
 \llbracket \text{skip} \rrbracket &= \{\emptyset\} \\
 \llbracket x := M; C \rrbracket &= \llbracket C \rrbracket[M/x] \cup \bigcup_v (M = v \mid W x v \rightarrow \llbracket C \rrbracket[M/x]) \\
 \llbracket r := x; C \rrbracket &= \llbracket C \rrbracket[x/r] \cup \bigcup_v (R x v \rightarrow \llbracket C \rrbracket[x/r]) \\
 \llbracket \text{if } (M) \{ C \} \text{ else } \{ D \} \rrbracket &= (M \neq 0 \mid \llbracket C \rrbracket) \sqcup (M = 0 \mid \llbracket D \rrbracket) \\
 \llbracket C \mid \mid D \rrbracket &= \llbracket C \rrbracket \parallel \llbracket D \rrbracket \\
 \llbracket \text{var } x; C \rrbracket &= \nu x . \llbracket C \rrbracket
 \end{aligned}$$

At this point we have completed the formal definition of our model of speculative evaluation, and turn to examples of this model in use.

3 EXAMPLES

3.1 Sequential memory accesses

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

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

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

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

$$\bigcup_v (r + 1 = v \mid W y v \rightarrow \llbracket \text{skip} \rrbracket[v/y])$$

which includes the case where v is 1:

$$(r + 1 = 1 \mid W y 1) \rightarrow \llbracket \text{skip} \rrbracket[1/y]$$

which contains the pomset:

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

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

$$\llbracket y := r+1; \text{skip} \rrbracket[x/r] \cup \bigcup_v (R x v \rightarrow \llbracket y := r+1; \text{skip} \rrbracket[x/r])$$

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:

$$\llbracket y := r + 1; \text{skip} \rrbracket[x/r]$$

which contains the pomset:

$$x + 1 = 1 \mid W y 1$$

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

$$\bigcup_v (0 = v \mid W x v) \rightarrow \llbracket r := x; y := r + 1; \text{skip} \rrbracket[v/x]$$

which includes the case where v is 0:

$$(0 = 0 \mid W x 0) \rightarrow \llbracket r := x; y := r + 1; \text{skip} \rrbracket[0/x]$$

which contains the pomset:

$$0 = 0 \mid W x 0$$

$$0 + 1 = 1 \mid W y 1$$

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

$$W x 0$$

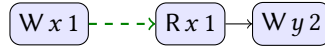
$$W y 1$$

Note that $(W x 0)$ does not read anything, and so there is no requirement of order between $(W x 0)$ and $(W y 1)$.

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 precondition ϕ then $x := M; C$ has precondition $\phi[M/x]$.

3.2 Concurrent memory accesses

We now turn to the case of concurrent accesses to memory. Consider a concurrent version of the program from §3.1: $(x := 1; \mid \mid y := x+1; \text{skip})$. 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 $(x := 1; \text{skip} \mid \mid r := x; y := r+1; \text{skip})$. As before, $\llbracket y := r + 1; \text{skip} \rrbracket$ is:

$$\bigcup_v (r + 1 = v \mid W y v) \rightarrow \llbracket \text{skip} \rrbracket[v/y]$$

which includes the case where v is 2:

$$(r + 1 = 2 \mid W y 2) \rightarrow \llbracket \text{skip} \rrbracket[2/y]$$

which contains the pomset:

$$r + 1 = 2 \mid W y 2$$

Now $\llbracket r := x; y := r + 1; \text{skip} \rrbracket$ is:

$$\llbracket y := r + 1; \text{skip} \rrbracket[x/r] \cup \bigcup_v (R x v) \rightarrow \llbracket y := r + 1; \text{skip} \rrbracket[x/r]$$

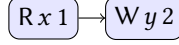
This has two cases, the sequential case (which does not introduce a read action) and the concurrent case (which does). We are now interested in the concurrent case, which is:

$$\bigcup_v (R x v) \rightarrow \llbracket y := r + 1; \text{skip} \rrbracket [x/r]$$

which includes the case where v is 1:

$$(R x 1) \rightarrow \llbracket y := r + 1; \text{skip} \rrbracket [x/r]$$

which contains the pomset:

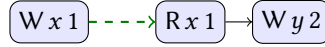


Note that $(R x 1)$ reads 1 from x , and while true implies $(x + 1 = 2)[1/x]$ (since $1 + 1 = 2$ is a tautology) it does *not* imply $(x + 1 = 2)$ (which is not true for every value of x), and so there is an ordering $(R x 1) < (W y 2)$ modelling the data dependency of the write of y on the read of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

Now, $\llbracket x := 1; \text{skip} \rrbracket$ includes the pomset:



and so $\llbracket x := 1; \text{skip} \rrbracket \parallel \llbracket r := x; y := r + 1; \text{skip} \rrbracket$ 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 $(R x v)$, if the execution is x -closed, then there must be an event it reads from, for example one labelled $(W x v)$.

3.3 Independent writes

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

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

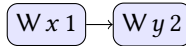
which is the same as:

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

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

$$(1 = 1 \mid W x 1) \rightarrow (2 = 2 \mid W y 2) \rightarrow \{\emptyset\}$$

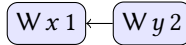
One of the executions this contains is:



but it also contains:



and:

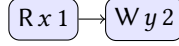


since there is no requirement that $(W x 1) \leq (W y 2)$.

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

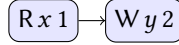
3.4 Independent reads and writes

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:

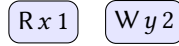


but since true does not imply $(x + 1 = 2)$, we have the requirement that $(R x 1) \leq (W y 2)$.

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

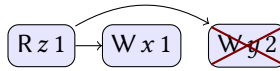
Conditionals introduce control dependencies, for example consider the program:

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

This includes executions in which the false branch is taken:



and ones where the true branch is taken:

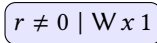


In both cases, we record the actions in the branch that was not taken. This is a novel feature of this model, and is intended to capture speculative evaluation. In §3.7 we will show how this model captures Spectre-like information flow attacks, once the attacker is provided with the ability to observe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

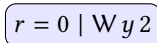
To see how these executions are modelled, consider the semantics of $\llbracket x := 1; \text{ skip} \rrbracket$, which contains any pomset of the form:



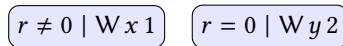
in particular it contains:



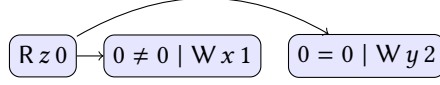
Similarly $\llbracket y := 2; \text{ skip} \rrbracket$ contains:



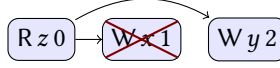
and so $\llbracket \text{if } (r) \{ x := 1; \text{ skip} \} \text{ else } \{ y := 2; \text{ skip} \} \rrbracket$ contains:



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 $(R z 0) < (W y 2)$, 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:



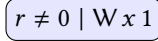
but it also has an execution in which the two writes of 1 to x are merged, resulting in no dependency:



To see how this arises in the model, consider the definition of $\llbracket \text{if } (r) \{x:=1; \text{skip}\} \} \text{else } \{ \{x:=1; \text{skip}\} \rrbracket$:

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

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



that is P_1 where:

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

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



that is P_2 where:

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

Crucially, 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:

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

that is:

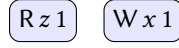


Note that this pomset has no precondition dependent on r , since $(r \neq 0 \vee r = 0)$ does not depend on r , which is why we end up with an execution without a control dependency:

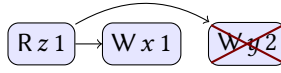


This semantics captures compiler optimizations which may, for example merge code executed on both branches of a conditional, or hoist constant assignments out of loops.

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



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



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 $\text{canRead}(\text{SECRET})$ 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.

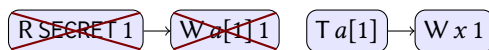
We model the touched test by introducing a new action $(T x)$ and defining:

$$\llbracket \text{if touched } x \{ C \} \text{ else } \{ D \} \rrbracket = ((Tx) \rightarrow \llbracket C \rrbracket) \cup \llbracket D \rrbracket$$

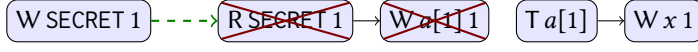
The additional requirement we need to add for x -closure is:

- if $\lambda(e) = (\phi \mid Tx)$ then there is $d \not\approx e$ with $\lambda(d) = (\psi \mid a)$ where a reads or writes x .

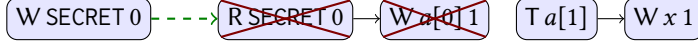
For example, one execution of SPECTRE is:



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.

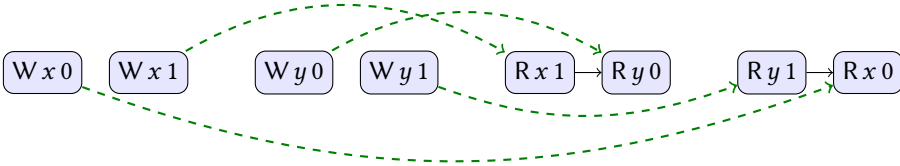
This shows how our model of speculative evaluation can express (very abstract, untimed) Spectre attacks.

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 evaluation. Unlike Spectre it does not depend on timing attacks, but instead is based on the sensitivity of relaxed memory to data dependencies. For this reason, we present a simple model of relaxed memory, which is strong enough to capture this attack. The model includes concurrent memory accesses, which can introduce concurrent reads-from. Since we are allowing events to be partially ordered, this gives a simple model of relaxed memory, for example an independent read independent write (IRIW) example is:

$x := 0; x := x+1; \parallel y := 0; y := y+1; \parallel \text{if } (x) \{ r := y; \} \parallel \text{if } (y) \{ s := x; \}$

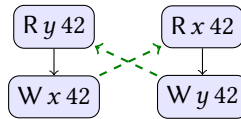
which includes the execution:



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

$x := y; \parallel y := x;$

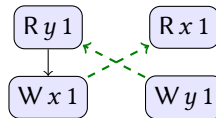
but an attempt to produce a value from thin air fails, for the usual reason of producing a cycle in \leq :



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

$x := y; \parallel r := x; y := r+1-r;$

has execution:



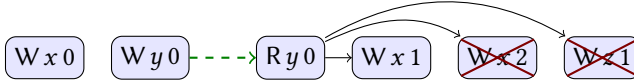
Note that $(R x 1) \not\leq (W y 1)$, so this does not introduce a cycle.

3.9 Information flow attacks on relaxed memory

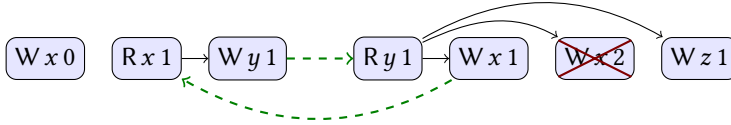
Consider an attacker program, again using security checks to try to learn a SECRET. Whereas SPECTRE uses hardware capabilities, which have to be modeled by adding extra capabilities to the language, this new attacker works by exploiting relaxed memory which can result in unexpected information flows.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

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

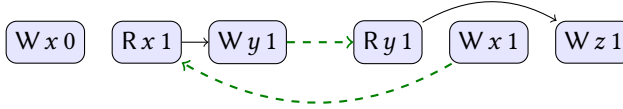
In the case where SECRET is 2, this has many executions, one of which is:



but there are no executions which exhibit (Wz1), 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 (Ry1) to (Wx1), which is what makes this execution possible. Thus, if the attacker sees an execution with (Wz1), 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 (Wz1). 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:

$Wx2$

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

$Wx1 \quad r \mid Wx2$

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 $(\phi \mid a) \rightarrow \mathcal{P}$, replace the requirement:

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

by:

- either:
 - $\lambda'(0) = (\psi, a)$, where ψ implies ϕ , or
 - $\lambda(0) = (\psi, b)$, where ϕ implies ψ , 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:

$Rx1 \xleftarrow{\text{dashed}} Wx1 \quad \phi \mid Wx2$

where ϕ is $(\neg \text{canRead}(\text{SECRET}))$, which is not a tautology, and so the $(Wx1)$ event is not omitted. In the case that SECRET is not 0, the matching execution is:

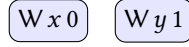
$Rx2 \xleftarrow{\text{dashed}} Wx2$

Now the $(Wx2)$ event is a guaranteed write, so the $(Wx1)$ 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.

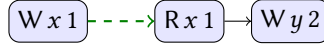
3.11 Thread inlining

One property one could ask of a model of shared memory is thread inlining: any execution of $\llbracket P; Q \rrbracket$ is an execution of $\llbracket P \parallel Q \rrbracket$. 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 $(x := 1; \parallel y := x+1;)$ has:

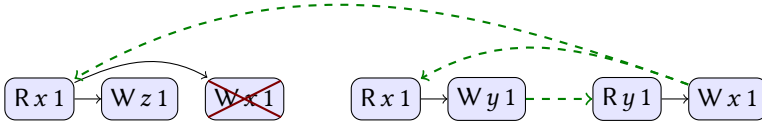


That is in the sequential case there is no dependency from the write of x to the write of y , but in the concurrent case there is such a dependency.

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

$x := 0; \text{ if } (x == 1) \{ z := 1; \} \text{ else } \{ x := 1; \} \parallel y := x; \parallel x := y;$

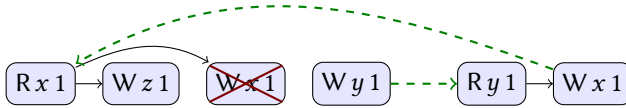
This has no execution containing $(Wz1)$. Any attempt to build such an execution results in a cycle:



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

$x := 0; \text{ if } (x == 1) \{ z := 1; \} \text{ else } \{ x := 1; \} y := x; \parallel x := y;$

with execution:



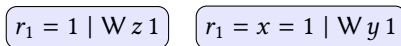
To see why this execution exists, consider the program fragment:

$\text{if } (x == 1) \{ z := 1; \} \text{ else } \{ x := 1; \} y := x;$

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, $\llbracket z := 1; r2 := x; y := r2; skip \rrbracket$ includes pomset:



and $\llbracket x := 1; r3 := x; y := r3; \text{skip} \rrbracket$ includes pomset:

$$r_1 \neq 1 \mid W x 1 \quad r_1 \neq 1 \mid W y 1$$

so $\llbracket \text{if } (r_1 = 1) \{ z := 1; r2 := x; y := r2; \text{skip} \} \text{ else } \{ x := 1; r3 := x; y := r3; \text{skip} \} \rrbracket$ includes:

$$r_1 = 1 \mid W z 1 \quad r_1 = 0 \mid W x 1 \quad (r_1 = x = 1) \vee (r_1 = 0) \mid W y 1$$

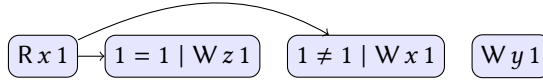
which means $\llbracket \text{if } (r_1 = 1) \{ z := 1; r2 := x; y := r2; \text{skip} \} \text{ else } \{ x := 1; r3 := x; y := r3; \text{skip} \} \rrbracket[x/r_1]$ includes:

$$x = 1 \mid W z 1 \quad x \neq 1 \mid W x 1 \quad (x = x = 1) \vee (x \neq 1) \mid W y 1$$

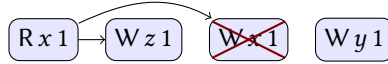
Now $(x = x = 1) \vee (x \neq 1)$ is a tautology, so this is just:

$$x = 1 \mid W z 1 \quad x \neq 1 \mid W x 1 \quad W y 1$$

and so $\llbracket r_1 := x; \text{if } (r_1 = 1) \{ z := 1; r2 := x; y := r2; \text{skip} \} \text{ else } \{ x := 1; r3 := x; y := r3; \text{skip} \} \rrbracket$ 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.

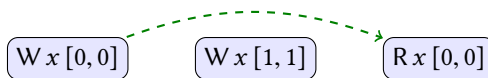
3.12 Word tearing

In §3.14, we shall be considering transactional memory, and in §3.14 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.

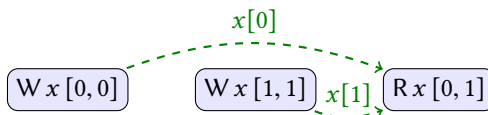
In word tearing, a program contains a write instruction with data larger than the hardware word size, for example copying a byte array, or assigning a 64-bit float on a 32-bit architecture. For example, consider the program:

$(x := [\emptyset, \emptyset];) \parallel (x := [1, 1];) \parallel (r := x;)$

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 Release/acquire synchronization

In relaxed memory models, synchronization actions act as memory fences: that is, they are a barrier to reordering memory accesses. In this section, we present a simple model of release/acquire fencing. In §3.14, we show that this can be scaled up to a model of transactional memory.

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

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

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

- $(W_{\text{rel}} x v)$, a release action that writes v to x , and
- $(R_{\text{acq}} x v)$, an acquire action that reads v from x .

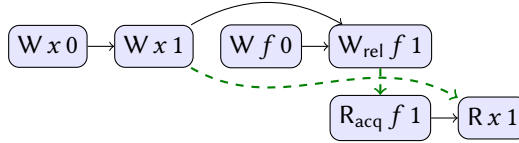
The semantics of programs with releasing write and acquiring read are the same as for regular write and read and, but with $W_{\text{rel}} x v$ replacing $W x v$ and $R_{\text{acq}} x v$ replacing $R x v$.

$$\begin{aligned} \llbracket \text{rel } x := M; C \rrbracket &= \bigcup_v (M = v \mid W_{\text{rel}} x v \rightarrow \llbracket C \rrbracket [M/x]) \\ \llbracket \text{acq } r := x; C \rrbracket &= \llbracket C \rrbracket [x/r] \cup \bigcup_v (\text{true} \mid R_{\text{acq}} x v \rightarrow \llbracket C \rrbracket [x/r]) \end{aligned}$$

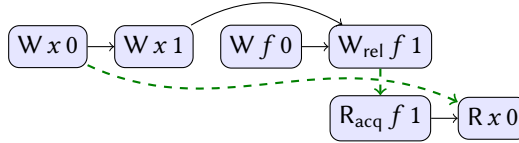
The extra constraint on prefixing ensures that events are ordered before a release and after an acquire. For example, consider the program:

$(x := 0; f := 0; x := 1; \text{rel } f := 1;) \parallel (\text{acq } r := f; s := x;)$

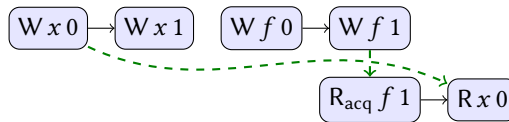
This has an execution:



but *not*:



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



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

The notion rf-pomset is sufficient to capture hardware models and release/acquire access in C++, where happens-before contains reads-from [1]. To model relaxed access, it would be necessary to use a more general notion of rf-pomset, where $(d, x, e) \in \text{RF}$ does not necessarily imply $d < e$, instead requiring that $(< \cup \text{RF})$ be acyclic.

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:

$$\begin{aligned} \llbracket \text{begin}; D \rrbracket &= (\text{true} \mid B) \rightarrow \llbracket D \rrbracket \\ \llbracket \text{if commit } \vec{x} \{ D_1 \} \text{ else } \{ D_2 \} \rrbracket &= \bigcup_{\vec{v}, \phi \text{ implies } \vec{x}=\vec{v}} ((\phi \mid C \vec{x} \vec{v}) \rightarrow (\phi \mid \llbracket D_1 \rrbracket)) \sqcup (\neg\phi \mid \llbracket D_2 \rrbracket) \end{aligned}$$

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 $\lambda(e) = (\phi \mid B)$ and a *commit event* if $\lambda(e) = (\phi \mid C \vec{x} \vec{v})$.

We write ϕ_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}) :

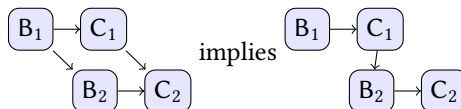
- (1) ϕ_e implies $\bigvee_i \phi_{c_i}$,
- (2) if $d < e$ then $d < b$,
- (3) if $e < d$ and d is compatible with c_i then $c_i < d$,
- (4) if $e' \neq e$ belongs to (b', \vec{c}') but not $(b, _)$ and
 - c'_j is compatible with e , and
 - c_i is compatible with e'
 then either $c'_j < b$ or $c_i <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

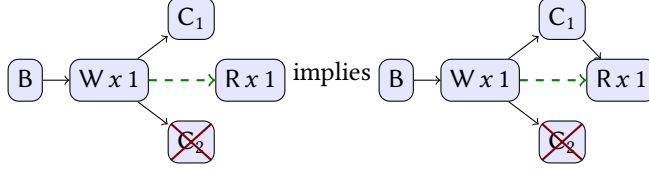
The definition of atomicity guarantees strong isolation. For weak isolation, clauses (5) and (6) are unnecessary, (2) only applies when d is a commit, and (3) only applies when d is a begin.

The definition handles simple examples:

- Single threaded example: $B_1 C_1 B_2 C_2$. Because C_2 is a release, we know that $C_1 < C_2$. Because B_1 is an acquire, we know that $B_1 < B_2$. By lifting, either of these is sufficient to require that $C_1 < B_2$.



- Abort example:

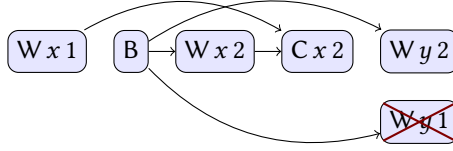


Let “end; D ” be syntax sugar for “if commit \vec{x} { D } else { D }”, where \vec{x} are the free variables of D .

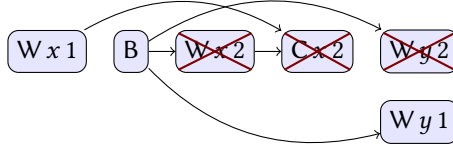
The semantics of

$x:=1$; 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;)
```

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 $N \leq 64$):

```
(
  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

REFERENCES

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