A classic locked-room mystery. Eve was in the false branch of a conditional the whole time,
how could she do it?
A day out at the Tate Modern
Introduction

Spectre

Optimizations

Simplified Spectre

Results

Experiments

Conclusions
OMG a timing back channel based on speculative evaluation and caching. Three issues that are often brushed under the carpet by formal models.
Attacks bypass run-time security checks.

Can bypass array bounds checks, and read whole process memory.

Can be exploited from JS, so evil.ad.com can read your bank.com data.

Attacks *speculative evaluation* hardware optimization.
A lie we tell programmers:
“computers execute instructions one after the other.”

\[ x := x + 1; y := 1 \]

has execution:

\[ R x \ 1 \rightarrow W x \ 2 \rightarrow W y \ 1 \]
Optimizations in hardware

A lie we tell programmers:
“computers execute instructions one after the other.”

\[ x := x + 1; y := 1 \]

has execution where \( W \ y \ 1 \) might happen first:

\[ R \ x \ 1 \rightarrow W \ x \ 2 \quad W \ y \ 1 \]
Optimizations in hardware

A lie we tell programmers:
“computers execute instructions one after the other.”

\[ x := x + 1; y := 1 \]

has execution:

\[ \text{R } x \text{ 1} \rightarrow \text{W } x \text{ 2} \quad \text{W } y \text{ 1} \]

Shared-memory concurrency leaks the abstraction
Optimizations in hardware

A lie we tell programmers:
“computers execute instructions one after the other.”

\[ x := x + 1; y := 1 \]

has execution:

\[ R x \rightarrow W x 2 \rightarrow W y 1 \]

Shared-memory concurrency leaks the abstraction

Resulted in entire research area: weak memory models (e.g. Pugh et al.; C11)
Another lie we tell programmers:
“only one branch of an if is executed.”

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

has execution:

\[
R \times 1 \rightarrow W \ y 1 \rightarrow W \ z 1
\]
Optimizations in hardware

Another lie we tell programmers: “only one branch of an if is executed.”

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

has execution where \( Wz1 \) might happen before \( Wy1 \):

\[
\text{Rx1} \rightarrow \text{Wy1} \rightarrow \text{Wz1}
\]
Optimizations in hardware

Another lie we tell programmers: “only one branch of an if is executed.”

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

has execution where \( Wy2 \) might happen, then get rolled back:
Optimizations in hardware and compilers

Another lie we tell programmers:
“only one branch of an if is executed.”

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

has execution where Wz1 might happen first:
Optimizations in hardware and compilers

Another lie we tell programmers:
“only one branch of an if is executed.”

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

has execution:

No language-level model for this!

As weak memory models are to OOO, so what is to speculation?
Imagine a SECRET, protected by a run-time security check:

```plaintext
if canRead(SECRET) { ... use SECRET ... } else { ... }
```

For attacker code canRead(SECRET) is always false.
Simplified Spectre

Imagine a SECRET, protected by a run-time security check:

```
if canRead(SECRET) { ... use SECRET ... } else { ... }
```

For attacker code canRead(SECRET) is always false, e.g.

```
R y 1 → W x 2
```

```
R SECRET 1 → W x 1
```

is an execution of if y { if canRead(SECRET) { x := SECRET } else { x := 2 } }.
Simplified Spectre

Imagine a SECRET, protected by a run-time security check:

\[
\text{if canRead(SECRET) \{ \ldots use SECRET \ldots \} else \{ \ldots \}}
\]

For attacker code canRead(SECRET) is always false, e.g.

\[
\begin{array}{c}
R y 1 \rightarrow W x 2 \\
\text{R SECRET 1} \rightarrow W x 1
\end{array}
\]

is an execution of if \( y \) \{ if canRead(SECRET) \{ x := \text{SECRET} \} else \{ x := 2 \} \}.

Attacker goal: learn if SECRET is 0 or 1.
A very simplified Spectre attack:

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

with execution

```
R SECRET 1 W a[1] 1 magic! W x 1
```

Information flow from SECRET to $x$, *if* there's an implementation of "magic".
Simplified Spectre

A very simplified Spectre attack:

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

with execution

![Execution Diagram]

Information flow from SECRET to x, *if* there’s an implementation of “magic”.

*Narrator:* there was one.
Results

Formalization of pretty pictures as *partially ordered multisets* (Gisher, 1988).

Compositional semantics based on weak memory models (e.g. C11).

Examples modeling Spectre, Spectre mitigations, PRIME+ABORT attack on transactional memory...
Results

Formalization of pretty pictures as *partially ordered multisets* (Gisher, 1988).

Compositional semantics based on weak memory models (e.g. C11).

Examples modeling Spectre, Spectre mitigations, PRIME+ABORT attack on transactional memory... and a new family of attacks on compiler optimizations.
Modeling an attack on compiler optimizations

An attacker running two threads (initially $x = y = 0$):

$$y := x \quad || \quad \text{if } (y == 0) \{ x := 1 \} \quad \text{else if} \ (\text{canRead}(\text{SECRET})) \{ x := \text{SECRET} \} \quad \text{else} \{ x := 1; z := 1 \}$$

If SECRET is 1, there is an execution:

- $R_x 1 \quad W_y 1 \quad R_y 1 \quad W_z 1$
- $R_x 1 \quad W_y 1 \quad W_x 1 \quad W_z 1$

If SECRET is 2, there is no execution (due to cyclic dependency):

- $R_x 1 \quad W_x 1 \quad R_y 1 \quad W_z 1$
Implementing attacks on compiler optimizations

Spectre and Prime+Abort are implemented.

Can we implement the attacks on compiler optimizations?
Implementing attacks on compiler optimizations

Spectre and Prime+Abort are implemented.

Can we implement the attacks on compiler optimizations?

Yes
Implementing attacks on compiler optimizations

Spectre and Prime+Abort are implemented.

Can we implement the attacks on compiler optimizations?

Yes, under unrealistic assumptions:

- SECRET is a constant known at compile-time
- canRead(SECRET) is a run-time check
Implementing an attack on load/store reordering

Main attacker thread:  
\[
\begin{align*}
  x &:= 1; \\
  \text{if (canRead(SECRET)) } \{ \\
  x &:= \text{SECRET; } \\
  \} r := y;
\end{align*}
\]
Implementing an attack on load/store reordering

Main attacker thread: \( x := 1; \text{if } (\text{canRead(SECRET)}) \{ x := \text{SECRET}; \} r := y; \)

<table>
<thead>
<tr>
<th>When SECRET ( \neq 1 ), gcc generates:</th>
<th>When SECRET = 1, gcc generates:</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov canReadSecret(%rip), %eax</td>
<td>mov canReadSecret(%rip), %eax</td>
</tr>
<tr>
<td>mov $1, x(%rip)</td>
<td>mov y(%rip), %eax</td>
</tr>
<tr>
<td>test %eax, %eax</td>
<td>mov $1, x(%rip)</td>
</tr>
<tr>
<td>je label1</td>
<td>label1:</td>
</tr>
<tr>
<td>mov $0, x(%rip)</td>
<td>mov y(%rip), %eax</td>
</tr>
<tr>
<td>label1:</td>
<td>mov $0, x(%rip)</td>
</tr>
<tr>
<td>mov y(%rip), %eax</td>
<td></td>
</tr>
</tbody>
</table>
Implementing an attack on load/store reordering

Main attacker thread: \( x := 1; \) if (canRead(SECRET)) \{ \( x := \) SECRET; \} \( r := y; \)

When \( \text{SECRET} \neq 1 \), gcc generates:

\[
\begin{align*}
\text{mov canReadSecret(\%rip), \%eax} \\
\text{mov $1, x(\%rip)} \\
\text{test \%eax, \%eax} \\
\text{je label1} \\
\text{mov $0, x(\%rip)} \\
\text{mov y(\%rip), \%eax} \\
\text{label1:} \\
\text{mov y(\%rip), \%eax}
\end{align*}
\]

Writes \( x \) then reads \( y \)

When \( \text{SECRET} = 1 \), gcc generates:

\[
\begin{align*}
\text{mov canReadSecret(\%rip), \%eax} \\
\text{mov y(\%rip), \%eax} \\
\text{mov $1, x(\%rip)}
\end{align*}
\]
Implementing an attack on load/store reordering

Main attacker thread: \( x := 1; \text{if} (\text{canRead(SECRET)}) \{ x := \text{SECRET}; \} r := y; \)

When \( \text{SECRET} \neq 1 \), gcc generates:

```
  mov canReadSecret(%rip), %eax
  mov $1, x(%rip)
  test %eax, %eax
  je label1
  mov $0, x(%rip)
label1:
  mov y(%rip), %eax
```

\( \text{Writes } x \text{ then reads } y \)

When \( \text{SECRET} = 1 \), gcc generates:

```
  mov canReadSecret(%rip), %eax
  mov y(%rip), %eax
  mov $1, x(%rip)
```

\( \text{Conditional has been eliminated!} \)
\( \text{Reads } y \text{ then writes } x \)
Implementing an attack on load/store reordering

Main attacker thread: \( x := 1; \text{if} \left( \text{canRead(SECRET)} \right) \{ x := \text{SECRET}; \} r := y; \)

When \( \text{SECRET} \neq 1 \), gcc generates:

\[
\begin{align*}
\text{mov} & \quad \text{canReadSecret}(%\text{rip}), \%\text{eax} \\
\text{mov} & \quad 1, x(%\text{rip}) \\
\text{test} & \quad \%\text{eax}, \%\text{eax} \\
\text{je} & \quad \text{label1} \\
\text{mov} & \quad 0, x(%\text{rip}) \\
\text{label1:} & \quad \text{mov} \ y(%\text{rip}), \%\text{eax} \\
\end{align*}
\]

Write \( x \) then reads \( y \)

When \( \text{SECRET} = 1 \), gcc generates:

\[
\begin{align*}
\text{mov} & \quad \text{canReadSecret}(%\text{rip}), \%\text{eax} \\
\text{mov} & \quad y(%\text{rip}), \%\text{eax} \\
\text{mov} & \quad 1, x(%\text{rip}) \\
\end{align*}
\]

Conditional has been eliminated!

Reads \( y \) then writes \( x \)

Forwarding thread \( x := y \) allows attacker to spot the reordering
Implementing an attack on load/store reordering

Small delay between write $x$ and read $y$: increases probability of round trip

gcc will reorder across 30 straight-line instructions

Repeat to leak multiple bits, error correction

Bitwise accuracy 99.99% at 300Kbps
Implementing an attack on dead store elimination

A similar attack targets dead store elimination

Works on clang + gcc

Bitwise accuracy 99.99% at 1.2Mbps
Contributions

A compositional model of program execution that includes speculation.

Examples including existing information flow attacks on branch prediction and transactional memory, and new attacks on optimizing compilers.

Experimental evidence that the new attacks can be carried out, but only against compile-time secrets.

(Phew, we failed to mount attacks on JIT compilers.)

https://github.com/chicago-relaxed-memory/spec-eval