

A classic locked-room mystery.  
Eve was in the false branch  
of a conditional the whole time,  
*how could she do it?*

3 January 2018



## A day out at the Tate Modern

The Code  
That Never Ran

Craig Disselkoen,  
Radha Jagadeesan,  
Alan Jeffrey,  
James Riely

Introduction

Spectre

Optimizations

Simplified Spectre

Results

Experiments

Conclusions

3 January 2018



## The Code That Never Ran

Craig Disselkoen,  
Radha Jagadeesan,  
Alan Jeffrey,  
James Riely

### Introduction

Spectre

Optimizations

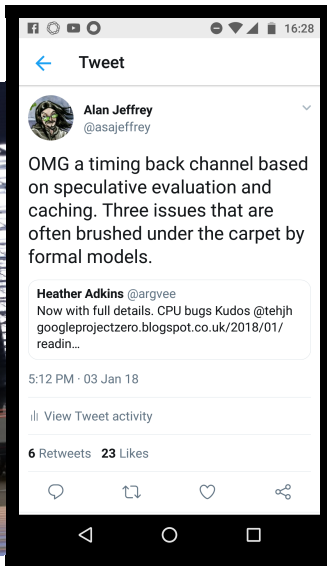
Simplified Spectre

Results

Experiments

Conclusions

3 January 2018



## The Code That Never Ran

Craig Disselkoen,  
Radha Jagadeesan,  
Alan Jeffrey,  
James Riely

### Introduction

### Spectre

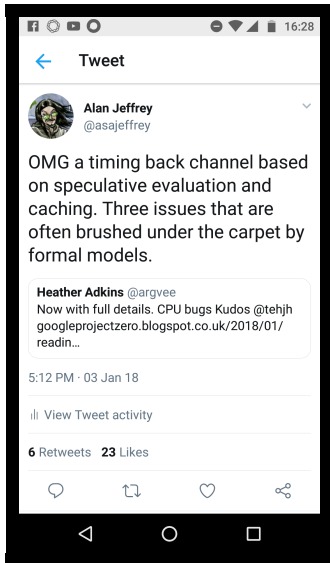
### Optimizations

### Simplified Spectre

### Results

### Experiments

### Conclusions



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.

# Optimizations in hardware

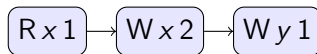
The Code  
That Never Ran

Craig Disselkoen,  
Radha Jagadeesan,  
Alan Jeffrey,  
James Riely

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

$$x := x + 1; y := 1$$

has execution:



Introduction

Spectre

**Optimizations**

Simplified Spectre

Results

Experiments

Conclusions

# Optimizations in hardware

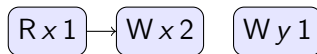
The Code  
That Never Ran

Craig Disselkoen,  
Radha Jagadeesan,  
Alan Jeffrey,  
James Riely

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:



Introduction

Spectre

Optimizations

Simplified Spectre

Results

Experiments

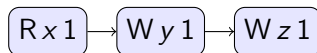
Conclusions

# 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:



The Code  
That Never Ran

Craig Disselkoen,  
Radha Jagadeesan,  
Alan Jeffrey,  
James Riely

Introduction

Spectre

Optimizations

Simplified Spectre

Results

Experiments

Conclusions

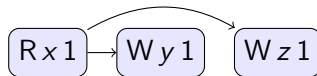


# Optimizations in hardware

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  $W z 1$  might happen before  $W y 1$ :



The Code  
That Never Ran

Craig Disselkoen,  
Radha Jagadeesan,  
Alan Jeffrey,  
James Riely

Introduction

Spectre

Optimizations

Simplified Spectre

Results

Experiments

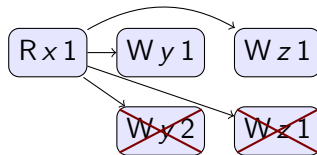
Conclusions

# 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  $W y 2$  might happen, then get rolled back:



The Code  
That Never Ran

Craig Disselkoen,  
Radha Jagadeesan,  
Alan Jeffrey,  
James Riely

Introduction

Spectre

Optimizations

Simplified Spectre

Results

Experiments

Conclusions

# Optimizations in hardware and compilers

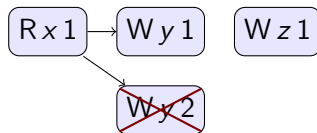
The Code  
That Never Ran

Craig Disselkoen,  
Radha Jagadeesan,  
Alan Jeffrey,  
James Riely

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  $W z 1$  might happen first:



Introduction

Spectre

Optimizations

Simplified Spectre

Results

Experiments

Conclusions

# 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

The Code  
That Never Ran

Craig Disselkoen,  
Radha Jagadeesan,  
Alan Jeffrey,  
James Riely

Introduction

Spectre

Optimizations

**Simplified Spectre**

Results

Experiments

Conclusions

# Simplified Spectre

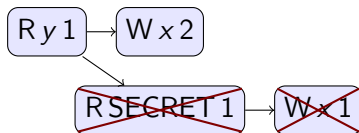
The Code  
That Never Ran

Craig Disselkoen,  
Radha Jagadeesan,  
Alan Jeffrey,  
James Riely

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.



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

Introduction

Spectre

Optimizations

Simplified Spectre

Results

Experiments

Conclusions

# Simplified Spectre

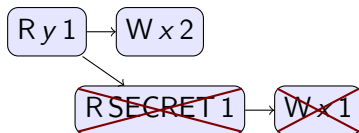
The Code  
That Never Ran

Craig Disselkoen,  
Radha Jagadeesan,  
Alan Jeffrey,  
James Riely

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.



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

Attacker goal: learn if SECRET is 0 or 1.

Introduction

Spectre

Optimizations

Simplified Spectre

Results

Experiments

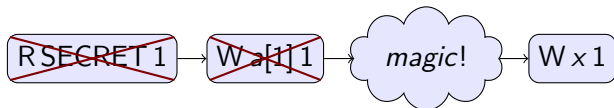
Conclusions

# Simplified Spectre

A very simplified Spectre attack:

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

with execution



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

The Code  
That Never Ran

Craig Disselkoen,  
Radha Jagadeesan,  
Alan Jeffrey,  
James Riely

Introduction

Spectre

Optimizations

Simplified Spectre

Results

Experiments

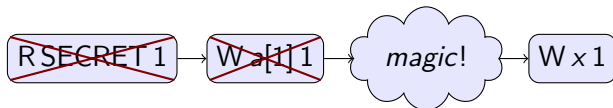
Conclusions

# Simplified Spectre

A very simplified Spectre attack:

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

with execution



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

*Narrator:* there was one.

The Code  
That Never Ran

Craig Disselkoen,  
Radha Jagadeesan,  
Alan Jeffrey,  
James Riely

Introduction

Spectre

Optimizations

Simplified Spectre

Results

Experiments

Conclusions



# 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. . .

The Code  
That Never Ran

Craig Disselkoen,  
Radha Jagadeesan,  
Alan Jeffrey,  
James Riely

Introduction

Spectre

Optimizations

Simplified Spectre

**Results**

Experiments

Conclusions

# 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.

The Code  
That Never Ran

Craig Disselkoen,  
Radha Jagadeesan,  
Alan Jeffrey,  
James Riely

Introduction

Spectre

Optimizations

Simplified Spectre

**Results**

Experiments

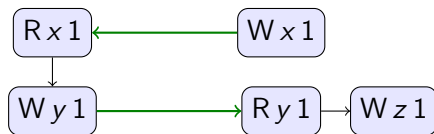
Conclusions

# Modeling an attack on compiler optimizations

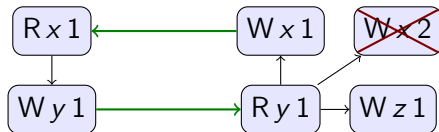
An attacker running two threads (initially  $x = y = 0$ ):

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

If SECRET is 1, there is an execution:



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



# Implementing attacks on compiler optimizations

The Code  
That Never Ran

Craig Disselkoen,  
Radha Jagadeesan,  
Alan Jeffrey,  
James Riely

Spectre and Prime+Abort are implemented.

What about the attacks on compiler optimizations?

Introduction

Spectre

Optimizations

Simplified Spectre

Results

**Experiments**

Conclusions

# Implementing attacks on compiler optimizations

The Code  
That Never Ran

Craig Disselkoen,  
Radha Jagadeesan,  
Alan Jeffrey,  
James Riely

Spectre and Prime+Abort are implemented.

What about the attacks on compiler optimizations?

Yes

Introduction

Spectre

Optimizations

Simplified Spectre

Results

Experiments

Conclusions

# Implementing attacks on compiler optimizations

The Code  
That Never Ran

Craig Disselkoen,  
Radha Jagadeesan,  
Alan Jeffrey,  
James Riely

Introduction

Spectre

Optimizations

Simplified Spectre

Results

Experiments

Conclusions

Spectre and Prime+Abort are implemented.

What about 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

The Code  
That Never Ran

Craig Disselkoen,  
Radha Jagadeesan,  
Alan Jeffrey,  
James Riely

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

If SECRET is not 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
```

Writes  $x$  then reads  $y$

If SECRET is 1, gcc generates:

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

Conditional has been eliminated!

Reads  $y$  then writes  $x$

A forwarding thread copies  $x$  to  $y$ , so the attacker can spot the reordering.

Introduction

Spectre

Optimizations

Simplified Spectre

Results

Experiments

Conclusions

# Implementing an attack on load/store reordering

The Code  
That Never Ran

Craig Disselkoen,  
Radha Jagadeesan,  
Alan Jeffrey,  
James Riely

Introduction

Spectre

Optimizations

Simplified Spectre

Results

Experiments

Conclusions

To improve accuracy, introduce a small delay between write of  $x$  and read of  $y$ , increases probability of round trip.

Experimentally, gcc will reorder load/store across 30 straight-line instructions.

Repeat attack to leak multiple bits, and increase probability of success.

Attack is 99.99% accurate at 300Kbps.



# Implementing an attack on dead store elimination

The Code  
That Never Ran

Craig Disselkoen,  
Radha Jagadeesan,  
Alan Jeffrey,  
James Riely

Introduction

Spectre

Optimizations

Simplified Spectre

Results

Experiments

Conclusions

A similar attack on works on dead store elimination.

Attack works on clang as well as gcc, 99.99% accurate at 1.2Mbps.

# Contributions

The Code  
That Never Ran

Craig Disselkoen,  
Radha Jagadeesan,  
Alan Jeffrey,  
James Riely

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.)

Introduction

Spectre

Optimizations

Simplified Spectre

Results

Experiments

Conclusions

# Thank you!

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

The Code  
That Never Ran

Craig Disselkoen,  
Radha Jagadeesan,  
Alan Jeffrey,  
James Riely

Introduction

Spectre

Optimizations

Simplified Spectre

Results

Experiments

Conclusions