What Spectre means for Language Implementers

Experience from securing the Web

Ben L. Titzer, WebAssembly TLM
Google Munich
Abstraction

Instruction Set Architecture

leaq 1145(%rip), %rcx
incl (%rcx,%rax,4)
haddpd %xmm1, %xmm1
cmpl %edx, %eax
je 148 <_output+0xEA>
jmp 169 <_output+0x104>
nopl (%rax,%rax)
movsd %xmm0, -56(%rbp)
movq %r14, %rax
orq $1, %rax
leaq 1389(%rip), %rcx
movq (%rcx,%rax,8), %rbx
subq (%rcx,%r14,8), %rbx
cmpq %r15, %rbx
cmovbq %rbx, %r15
cmpq %r13, %rbx
cmovaq %rbx, %r13
xorl %eax, %eax
leaq 806(%rip), %rdi
movq %rbx, %rsi
callq 692
...

...
Abstraction - Above

Source Language

double sample_sum = 0;
for (i = 0; i < NUM_SAMPLES * 2; i += 2) {
    uint64_t s = samples[i + 1] - samples[i];
    if (s < min) min = s;
    if (s > max) max = s;
    sum += s;
    sample_sum += s;
    // find the count for this sample.
    for (j = 0; j < c; j++) {
        if (s == common_vals[j]) {
            common_count[j]++;
            break;
        }
    }
    if (j == c) {
        common_vals[c] = s;
        common_count[c] = 1;
        c++;
    }
}

Instruction Set Architecture

leaq 1145(%rip), %rcx
incl (%rcx,%rax,4)
haddpd %xmm1, %xmm1
cmpl %edx, %eax
je 148 <_output+0xEA>
jmp 169 <_output+0x104>
nopl (%rax,%rax)
movsd %xmm8, -56(%rbp)
movq %r14, %rax
orq $1, %rax
leaq 1389(%rip), %rcx
movq (%rcx,%rax,8), %rbx
subq (%rcx,%r14,8), %rbx
cmpq %r15, %rbx
cmovbq %rbx, %r15
cmpq %r13, %rbx
cmovaq %rbx, %r13
xorl %eax, %eax
leaq 806(%rip), %rdi
movq %rbx, %rsi
callq 692
...
double sample_sum = 0;
for (i = 0; i < NUM_SAMPLES * 2; i += 2) {
    uint64_t s = samples[i + 1] - samples[i];
    if (s < min) min = s;
    if (s > max) max = s;
    sum += s;
    sample_sum += s;
    // find the count for this sample.
    for (j = 0; j < c; j++) {
        if (s == common_vals[j]) {
            common_count[j]++;
            break;
        }
    }
    if (j == c) {
        common_vals[c] = s;
        common_count[c] = 1;
        c++;
    }
}

Instruction Set Architecture

leaq 1145(%rip), %rcx
incl (%rcx,%rax,4)
haddpd %xmm1, %xmm1
cmpl %edx, %eax
je 148 _output+0xEA>
jmp 169 _output+0x104>
nopl (%rax,%rax)
movsd %xmm0, -56(%rbp)
movq %r14, %rax
orq $1, %rax
leaq 1389(%rip), %rcx
movq (%rcx,%rax,8), %rbx
subq (%rcx,%r14,8), %rbx
cmpq %r15, %rbx
cmovbq %rbx, %r15
cmpq %r13, %rbx
cmovaq %rbx, %r13
xorl %eax, %eax
leaq 886(%rip), %rdi
movq %rbx, %rsi
callq 692
...
**Source Language**

```c
double sample_sum = 0;
for (i = 0; i < NUM_SAMPLES * 2; i += 2) {
    uint64_t s = samples[i + 1] - samples[i];
    if (s < min) min = s;
    if (s > max) max = s;
    sum += s;
    sample_sum += s;
    // find the count for this sample.
    for (j = 0; j < c; j++) {
        if (s == common_vals[j]) {
            common_count[j]++;
            break;
        }
    }
    if (j == c) {
        common_vals[c] = s;
        common_count[c] = 1;
        c++;
    }
}
```

- **Source Language establishes an abstraction**
  - Closer to human language
  - Allows reasoning about source-level concepts
  - Hides implementation details
  - Portable across hardware, simulators
  - Safety properties for good SW engineering
  - Can run *untrusted* code if a virtual machine provides memory safety (a sandbox)
double sample_sum = 0;
for (i = 0; i < NUM_SAMPLES * 2; i += 2) {
    uint64_t s = samples[i + 1] - samples[i];
    if (s < min) min = s;
    if (s > max) max = s;
    sum += s;
    sample_sum += s;
    // find the count for this sample.
    for (j = 0; j < c; j++) {
        if (s == common_vals[j]) {
            common_count[j]++;
            break;
        }
    }
    if (j == c) {
        common_vals[c] = s;
        common_count[c] = 1;
        c++;
    }
}
Abstraction - Below

- Hardware is also an *abstraction*
  - Closer to physics
  - Realized with physical circuits
  - Provides backwards compatibility
  - Implementation freedom
  - Evolution of microarchitecture over time
  - Optimizations are not observable
Abstraction - Below

- Hardware is also an *abstraction*
  - Closer to physics
  - Realized with physical circuits
  - Provides backwards compatibility
  - Implementation freedom
  - Evolution of microarchitecture over time
  - **Optimizations are not observable**
Abstraction - Below

- Hardware is also an *abstraction*
  - Closer to physics
  - Realized with physical circuits
  - Provides backwards compatibility
  - Implementation freedom
  - Evolution of microarchitecture over time
  - Optimizations are not observable
  - Optimizations do not leak information
Branch Prediction - Variant 1

Example CFG
Branch Prediction - Variant 1

After training

- Safety check
- Access
- Error
Branch Prediction - Variant 1

- Misprediction safety check
- bad error
- access
- Misprediction
Branch Prediction - Variant 1

- Misprediction
- Safety check
- Access
- Error

"They'll never know!"
Branch Prediction - Variant 1

Misprediction

They’ll never know!

Except for that pesky μ-state!
Hidden μ-state in Computer Systems

- Page tables
- DRAM line buffer
- Instruction and data caches: L1i, L1d, L2, L3 + prefetchers
- Translation lookaside buffer: L1 and L2
- Branch prediction state (BHB, BTB)
- Return stack buffer
- Micro-op cache
- Model-specific registers

- Store buffer
- Memory disambiguator
- Frequency scaling (power save mode, TurboBoost)
- Execution port occupancy
- Lazy FPU state
- Bus and cache fill line occupancy
The ideal leakage mechanism!

- Page tables
- DRAM line buffer
- Instruction and data caches: L1i, L1d, L2, L3 + prefetchers
- Translation lookaside buffer: L1 and L2
- Branch prediction state (BHB, BTB)
- Return stack buffer
- Micro-op cache
- Model-specific registers

Store buffer
- Memory disambiguator
- Frequency scaling (power save mode, TurboBoost)
- Execution port occupancy
- Lazy FPU state
- Bus and cache fill line occupancy

Perfectly-designed side-channel for leaking information
Mounting an attack

```java
if (index < array.length) {
    int value = array[index];
    int unused = timing[(value & 1) * 64];
}
```

Out-of-bounds when mispredicted

Boundscheck bypass vulnerability

Either `timing[0]` or `timing[64]` is now faster
It’s not just bounds checks in JavaScript!

```javascript
if (condition) {
   /* potentially unsafe code */
}
```

Safetycheck bypass vulnerability

- Number check
- Object shape (map) check
- Function arity
- Global variable check
- Runtime checks

Any high-level language safety feature which an implementation enforces with branches is potentially vulnerable.
The Universal Read Gadget

- Using speculative safety check bypasses, it is possible to write a well-typed procedure:

```c
byte read_memory(uint64 address);
```

that uses side-channels to read from any memory location in the entire address space.
A (much) closer look at timing channels

- Lifetime: $10^9$
- Year: $10^8$
- Month: $10^7$
- Week: $10^6$
- Day: $10^5$
- Hour: $10^4$
- Minute: $10^3$
- Second: $10^2$, $10^1$, $10^0$
Timers in JavaScript

<table>
<thead>
<tr>
<th></th>
<th>Firefox 51</th>
<th>Chrome 53</th>
<th>Edge 38</th>
<th>Tor 6.0.4</th>
<th>Firefox</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Free-running</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>performance.now</code></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CSS animations</td>
<td>✓</td>
<td>16 ms</td>
<td>16 ms</td>
<td>16 ms</td>
<td>16 ms</td>
</tr>
<tr>
<td><code>setTimeout</code></td>
<td>✓</td>
<td>4 ms</td>
<td>4 ms</td>
<td>2 ms</td>
<td>4 ms</td>
</tr>
<tr>
<td><code>setImmediate</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>postMessage</code></td>
<td>✓</td>
<td>45 μs</td>
<td>35 μs</td>
<td>40 μs</td>
<td>40 μs</td>
</tr>
<tr>
<td><code>Sub worker</code></td>
<td>✓</td>
<td>20 μs</td>
<td>–</td>
<td>50 μs</td>
<td>15 μs</td>
</tr>
<tr>
<td>Broadcast Channel</td>
<td>✓</td>
<td>145 μs</td>
<td>–</td>
<td>–</td>
<td>55 μs</td>
</tr>
<tr>
<td>MessageChannel</td>
<td></td>
<td>12 μs</td>
<td>55 μs</td>
<td>20 μs</td>
<td>20 μs</td>
</tr>
<tr>
<td>MessageChannel (W)</td>
<td>✓</td>
<td>75 μs</td>
<td>100 μs</td>
<td>20 μs</td>
<td>30 μs</td>
</tr>
<tr>
<td>SharedArrayBuffer</td>
<td>✓</td>
<td>2 ns&lt;sup&gt;3&lt;/sup&gt;</td>
<td>15 ns&lt;sup&gt;4&lt;/sup&gt;</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td><strong>Interpolation&lt;sup&gt;1&lt;/sup&gt;</strong></td>
<td></td>
<td>500 ns</td>
<td>500 ns</td>
<td>350 ns</td>
<td>15 μs</td>
</tr>
<tr>
<td><strong>Edge thresholding&lt;sup&gt;2&lt;/sup&gt;</strong></td>
<td></td>
<td>2 ns</td>
<td>15 ns</td>
<td>10 ns</td>
<td>2 ns</td>
</tr>
</tbody>
</table>
Timer Mitigations

● Upon disclosure of Spectre, Browser vendors immediately reduced resolution of available timers in JavaScript
  ○ `performance.now()` 5μs → 100μs
  ○ Disabled `SharedArrayBuffer`, from which a high-resolution timer can be constructed. (This delayed the introduction of WebAssembly threads)
  ○ Introduced uniform-random `jitter` to defeat thresholding and interpolation

● Part of defense-in-depth, inadequate on their own
Amplification

Timer enhancement
Shared memory Timer Construction

```c
volatile uint64_t time = 0;
```

Global mutable variable

```c
for (; ;) { time++; }
```

Dedicated timer thread

```c
uint64_t before = time;
workload();
uint64_t delta = time - before;
```

Measuring workload
Amplification

- Small timing differences can be amplified to large timing differences
  - L1 cache hit vs main memory access ~ 1ns vs 100ns
  - Timer resolution of 1µs is “safe”, right?
  - No, can exploit timing difference repeatedly to amplify difference up to 600µs

```plaintext
if (secret_bit)
  lines A_0…A_n
  lines B_0…B_n
disclose

Limited by cache capacity
```
Arbitrary Amplification

- Small timing differences can be amplified to arbitrarily large timing differences
  - Arbitrary amplification technique: disclose and immediately read the secret bit
  - Bit is encoded as N misses vs 2N misses

```plaintext
if (secret_bit)
  read A
  read A
  disclose
  read
loop N times
```

Measure:

- 1
- 0
if (index < array.length) {
    int value = array[index];
    leak(value);
}
Variant 1 - Mitigations

```java
if (index < array.length) {
    LFENCE();
    int value = array[index];
    leak(value);
}
```

Early Intel recommendation

Boundscheck bypass vulnerability
Variant 1 - Mitigations

```java
if (index < array.length) {
    LFENCE();
    int value = array[index];
    leak(value);
}
```

LFENCE mitigation

2.8X
if (index < 0x10000) { // memory size
    int value = memory_base + index & 0xFFFF;
}
Variant 1 - Pervasive poisoning

Potentially vulnerable condition in attacker code

```javascript
if (condition) {
    var x = load(...);
    leak(x);
}
```

Poison operations inserted by V8 optimizing compiler

- Prologue
- Branches
- Loads
Variant 2 - Branch Target Injection

```javascript
var f = abc(...);
var x = f();
```

Potentially vulnerable indirect call in code

```javascript
var g = xyz(...);
var x = g();
```

Aliasing indirect call in attacker code
Variant 2 - Function prologue poisoning

```javascript
var f = abc(...);
var x = f(f);

function t(tn) {
  var poison = tn == t? -1 : 0;
  ...
}
```

Potential vulnerability: indirect call

Poison operations inserted by V8 optimizing compiler:
- Prologue
- Branches
- Loads
- Interpreter

```javascript
var f = abc(...);
var x = f(f);
```
Variant 4 - Speculative Store Bypass

- Page tables
- DRAM line buffer
- Instruction and data caches: L1i, L1d, L2, L3 + prefetchers
- Translation lookaside buffer: L1 and L2
- Branch prediction state (BHB, BTB)
- Return stack buffer
- Micro-op cache
- Model-specific registers
- Store buffer
- Memory disambiguator
- Frequency scaling (power save mode, TurboBoost)
- Execution port occupancy
- Lazy FPU state
- Bus and cache fill line occupancy
Out of Order Execution: now observable!
Out of Order Execution

The fetch pc is based completely on the predicted path; the branch instructions are still waiting to execute.
Out of Order Execution

Instructions in the reorder buffer execute based on dataflow (register) dependency graph.
Out of Order Execution

- Retired instructions
- Commit (~5 μ-ops/cycle)
- Reorder buffer (~200 μ-ops)
- Fetch (~5 ins/cycle)

Branch predictions are verified at instruction commit.
Out of Order Execution

- Retired instructions
- Commit (~5 μ-ops/cycle)
- Reorder buffer (~200 μ-ops)
- Fetch (~5 ins/cycle)

Branch mispredictions cause a flush of the reorder buffer upon commit.
Out of Order Execution

**reorder buffer (~200 μ-ops)**

**fetch (~5 ins/cycle)**

**commit (~5 μ-ops/cycle)**

**retired instructions**

Inside the reorder buffer, dataflow dependencies are tracked through registers, but **not control or memory dependencies**.
Observing Out-of-Order Execution

- CPUs try to execute loads and stores out-of-order
- Memory disambiguator predicts when this is profitable
- Mispredictions (unexpected aliases) cause re-execution of loads and dependents

\[
\begin{align*}
\text{mov } [x], \ #0 \\
\text{mov } rax, \ [y]
\end{align*}
\]

Any order OK if \( x \neq y \)
Speculative Store Bypass Vulnerability

Original code

...  
mov [x], #0  
mov rax, [y]  
...  

Out-order-execution

...  
mov rax, [y]  
mov [x], #0  
...  

store

load

✔ Still OK if x == y

✘ Incorrect if x == y
Speculative Store Bypass Vulnerability

Original code:
```
... mov [x], #0
mov rax, [y]
...```

Still OK if $x == y$

Universal Read Gadget:
```
... mov rax, [y]
mov [x], #0
mov rbx, [rax]
and rbx, #0xFF
shl rbx, #6
mov rcx, [#time+rbx]
...```

❌ Universal Read Gadget
Variant 4 - Mitigations

- We *failed* :(  

- Particularly challenging: type confusion on the execution stack (spill slots)
Recall all this?

- Page tables
- DRAM line buffer
- Instruction and data caches: L1i, L1d, L2, L3 + prefetchers
- Translation lookaside buffer: L1 and L2
- Branch prediction state (BHB, BTB)
- Return stack buffer
- Micro-op cache
- Model-specific registers

- Store buffer
- Memory disambiguator
- Frequency scaling (power save mode, TurboBoost)
- Execution port occupancy
- Lazy FPU state
- Bus and cache fill line occupancy
Known attacks on μ-state

- Page tables
- DRAM line buffer
- Instruction and data caches: L1i, L1d, L2, L3 + prefetchers
- Translation lookaside buffer: L1 and L2
- Branch prediction state (BHB, BTB)
- Return stack buffer
- Micro-op cache
- Model-specific registers

- Store buffer
- Memory disambiguator
- Frequency scaling (power save mode, TurboBoost)
- Execution port occupancy
- Lazy FPU state
- Bus and cache fill line occupancy
The Universal Read Gadget

- Using many **different mechanisms**, it is possible to write a well-typed procedure:

  ```c
  byte read_memory(uint64 address);
  ```

  in any language, that uses side-channels to read from any memory location in the **entire address space**
Abstraction - Above and Below

Source Language

double sample_sum = 0;
for (i = 0; i < NUM_SAMPLES * 2; i += 2) {
    uint64_t s = samples[i + 1] - samples[i];
    if (s < min) min = s;
    if (s > max) max = s;
    sum += s;
    sample_sum += s;
    // find the count for this sample.
    for (j = 0; j < c; j++) {
        if (s == common_vals[j]) {
            common_count[j]++;
            break;
        }
    }
    if (j == c) {
        common_vals[c] = s;
        common_count[c] = 1;
        c++;
    }
}

Instruction Set Architecture

   leaq 1145(%rip), %rcx
   incl (%rcx,%rax,4)
   haddpd %xmm1, %xmm1
   cmpl %edx, %eax
   je 148 <_output+0xEA>
   jmp 169 <_output+0x104>
   nopl (%rax,%rax)
   movsd %xmm0, -56(%rbp)
   movq %r14, %rax
   orq $1, %rax
   leaq 1389(%rip), %rcx
   movq (%rcx,%rax,8), %rbx
   subq (%rcx,%r14,8), %rbx
   cmpl %r15, %rbx
   cmovbq %rbx, %r15
   cmovaq %rbx, %r13
   xorl %eax, %eax
   leaq 886(%rip), %rdi
   movq %rbx, %rsi
   callq 692
   . . .

Hardware
Retreating to the Process Boundary

- With new variants mounting every month, Chrome pushed ahead with **site isolation** on platforms where it is available
  - Separate renderers (where JavaScript runs) from each other and the browser process, which contains secrets
  - Cross-origin resource blocking
  - **Mitigations** enabled on platforms where SI is infeasible

- Other browsers **working feverishly** to ship site isolation
  - Mitigations enabled, limited isolation via opt-in headers
Conclusion

- Modern CPUs *mispredict, misspeculate*, and execute programs *out of order*, which can be a problem for safety checks inserted by language implementations.
- Micro-architectural details *leak into and out of* programs via timing.
- Programming language implementations *cannot establish confidentiality* on today’s hardware
  \[ \Rightarrow \text{It’s possible to construct the universal read gadget in any language.} \]

**bad**

Don’t run untrusted code in the same process with secrets it could steal.
Questions?

Paper available on ArXiv:
“Spectre is Here to Stay: an Analysis of Side-channels and Speculative Execution”