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   %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 and Below

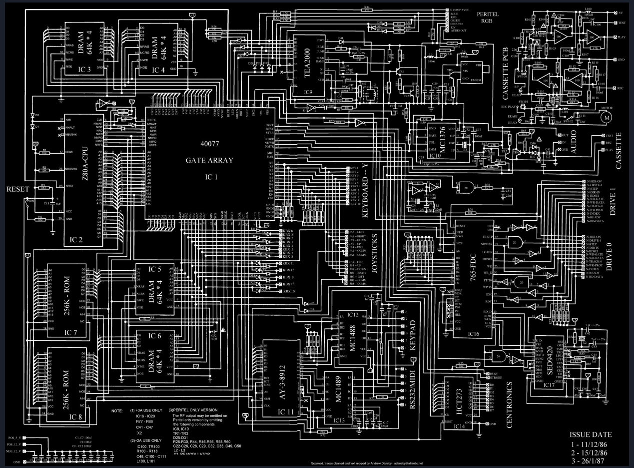
Source Language

```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++;
  }
}
```

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
. . .
```

Hardware



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++;
  }
}
```

- 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)

Abstraction - Above

Source Language

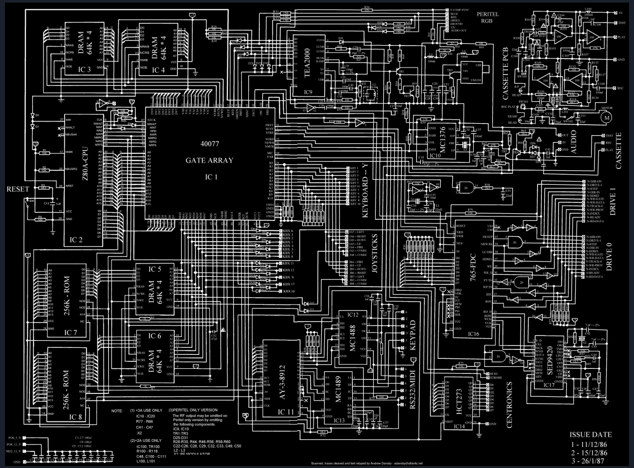
```
double sample_sum = 0;
for (i = 0; i < NUM_SAMPLES * 2; i += 2) {
  uint64_t s = samples[i + 1] - samples[i];
  if (s < min) min = s;
  if (s > max) max = s;
  sum += s;
  sample_sum += s;
  // find the count for this sample.
  for (j = 0; j < c; j++) {
    if (s == common_vals[j]) {
      common_count[j]++;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

- 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)**

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

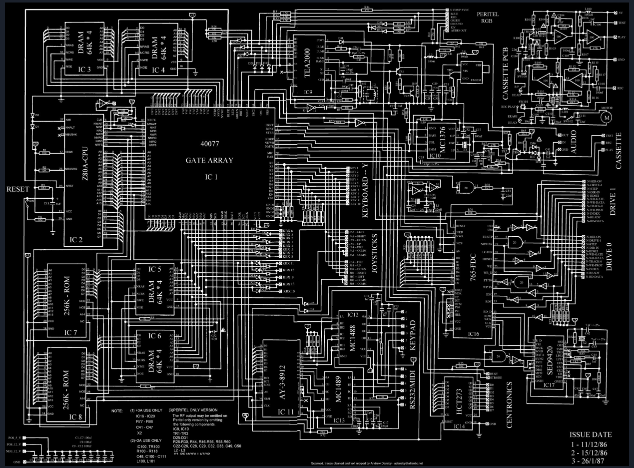
Hardware



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**

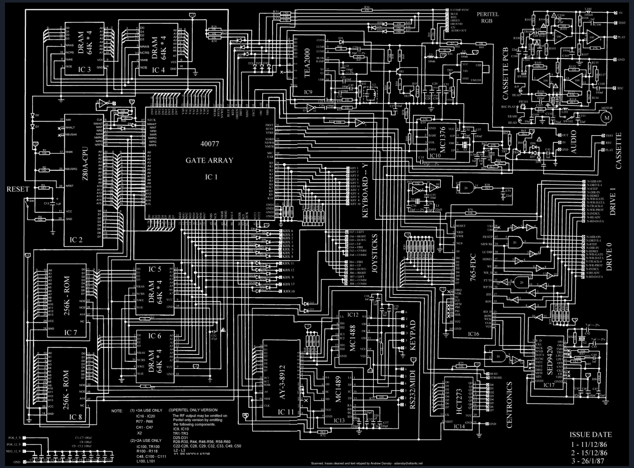
Hardware



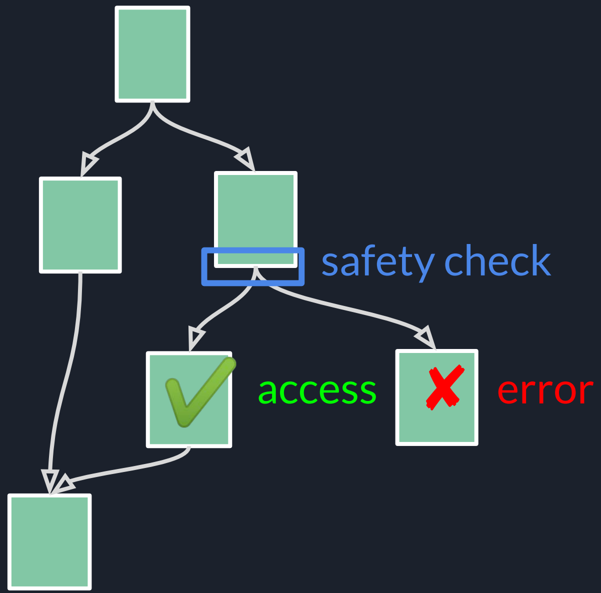
Abstraction - Below

- 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**

Hardware

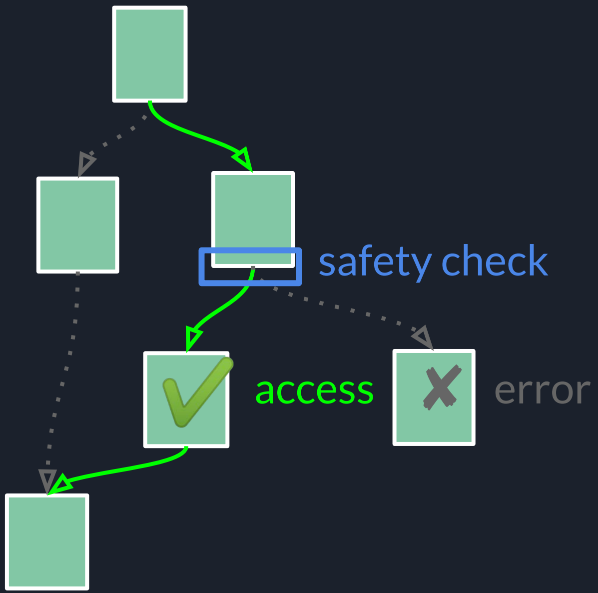


Branch Prediction - Variant 1



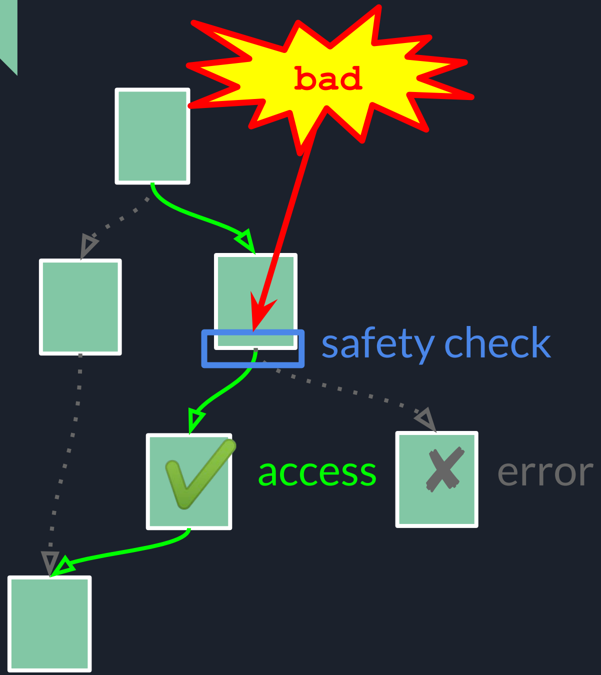
Example CFG

Branch Prediction - Variant 1



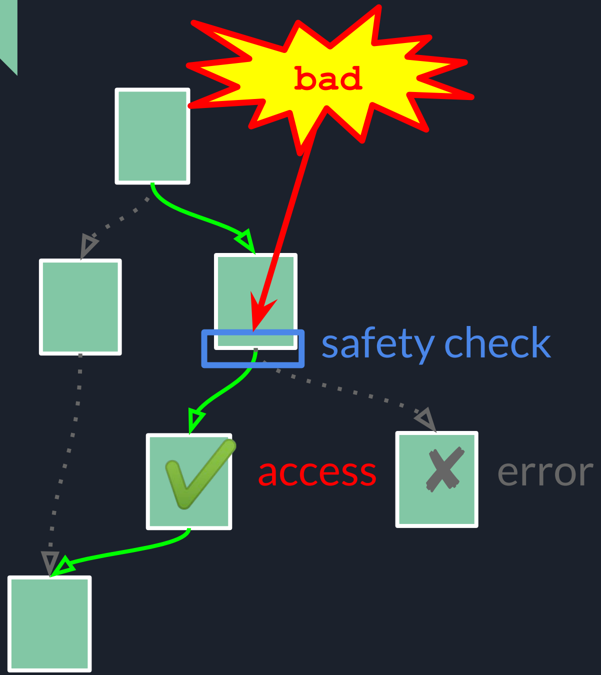
After training

Branch Prediction - Variant 1



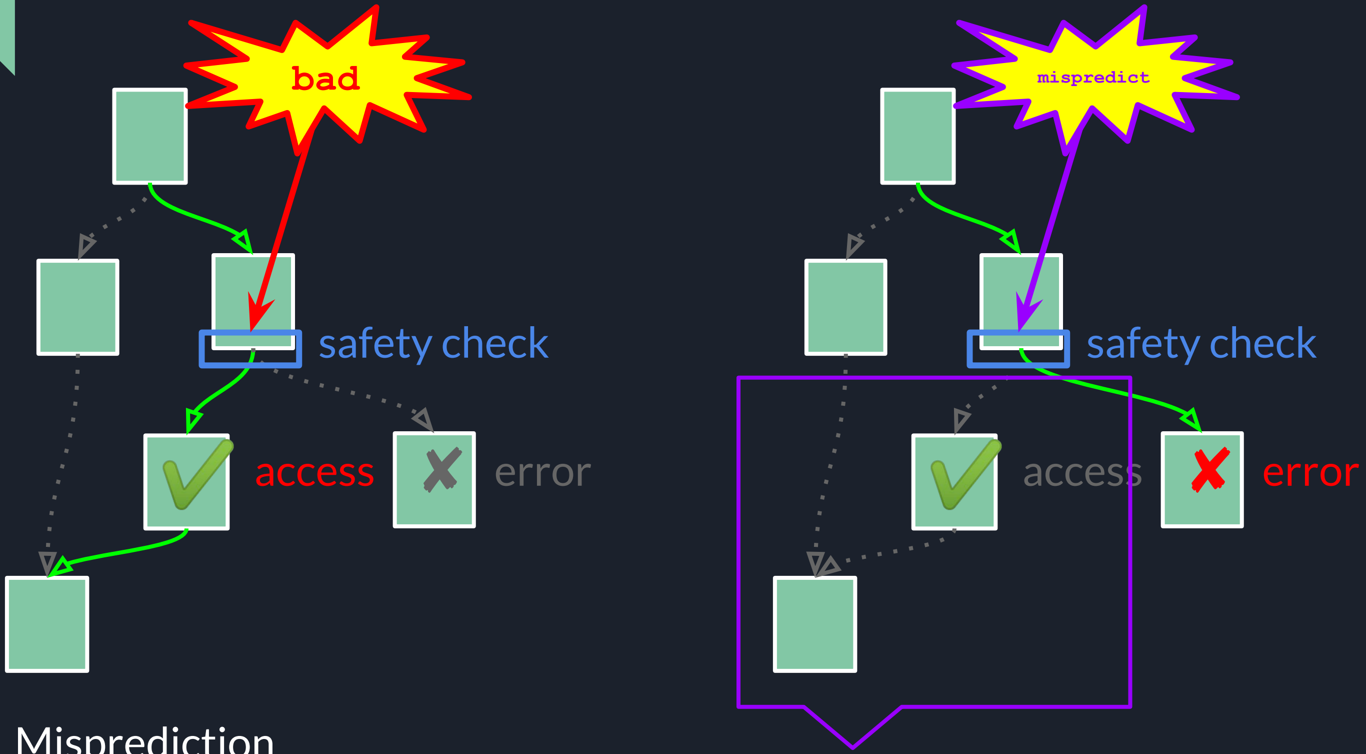
Misprediction

Branch Prediction - Variant 1



Misprediction

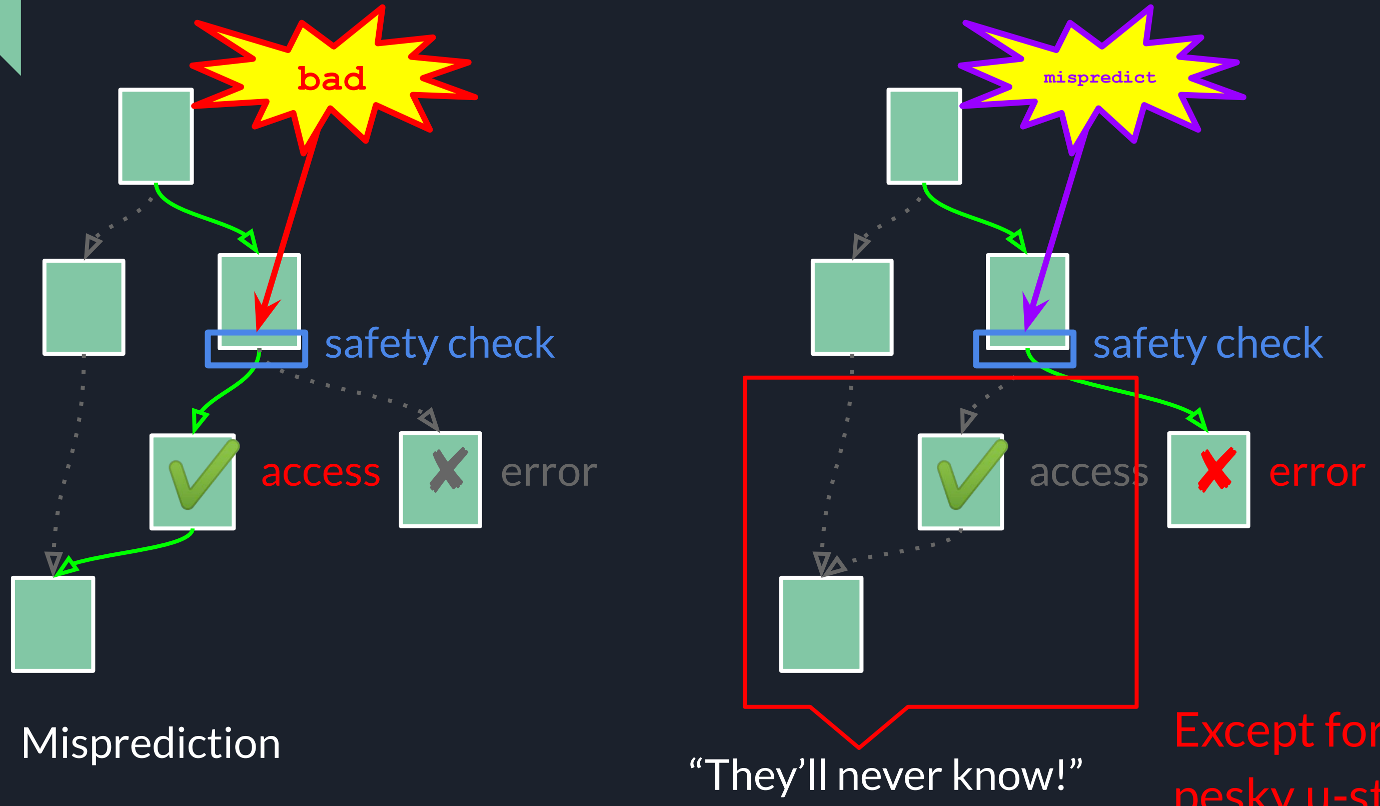
Branch Prediction - Variant 1



Misprediction

“They’ll never know!”

Branch Prediction - Variant 1



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!

Perfectly-designed side-channel for leaking information

- 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

Mounting an attack

Out-of-bounds
when mispredicted

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

Boundscheck bypass vulnerability

Either `timing[0]`
or `timing[64]`
is now ***faster***

It's not just bounds checks in 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.

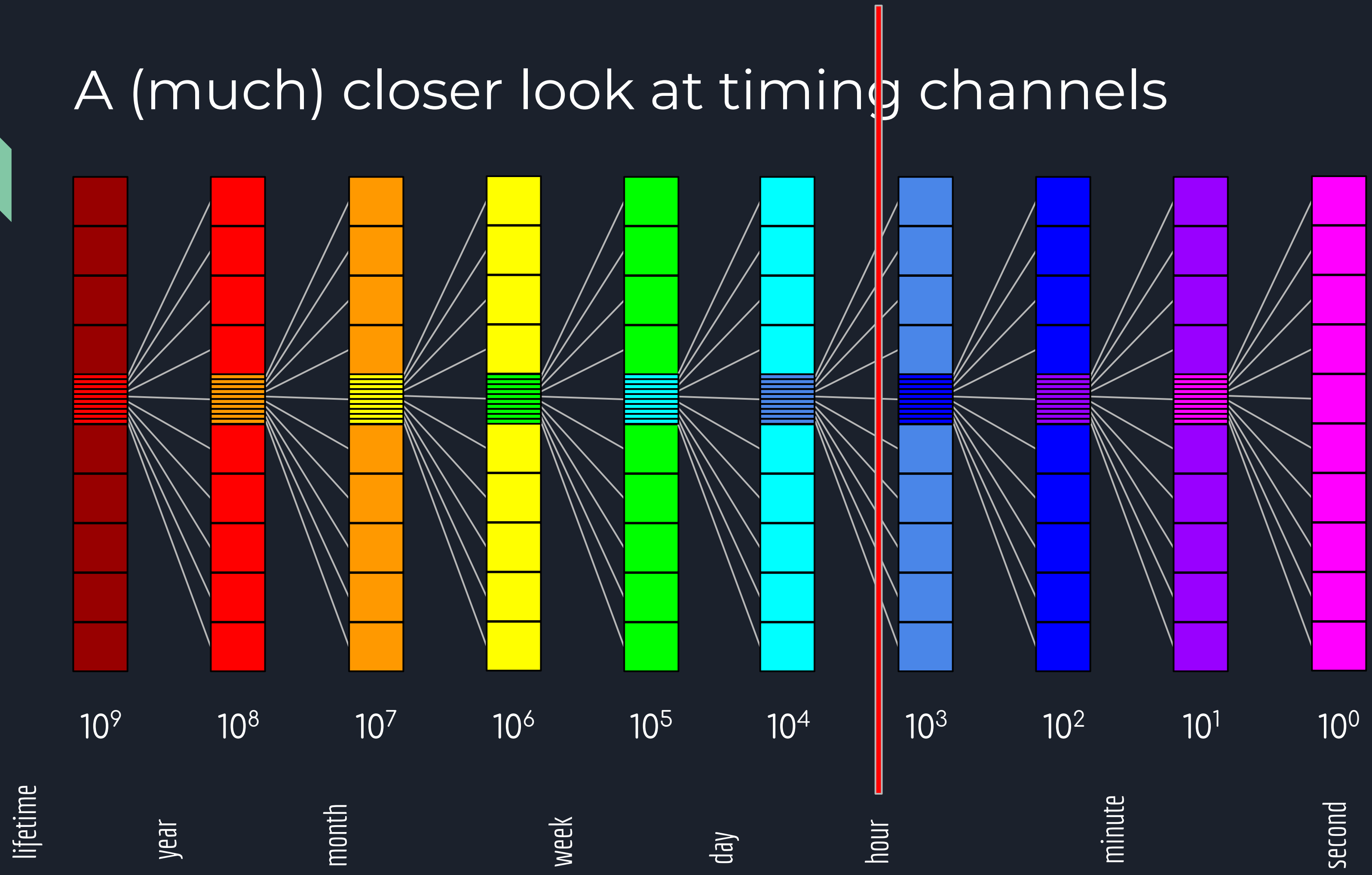
The Universal Read Gadget

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

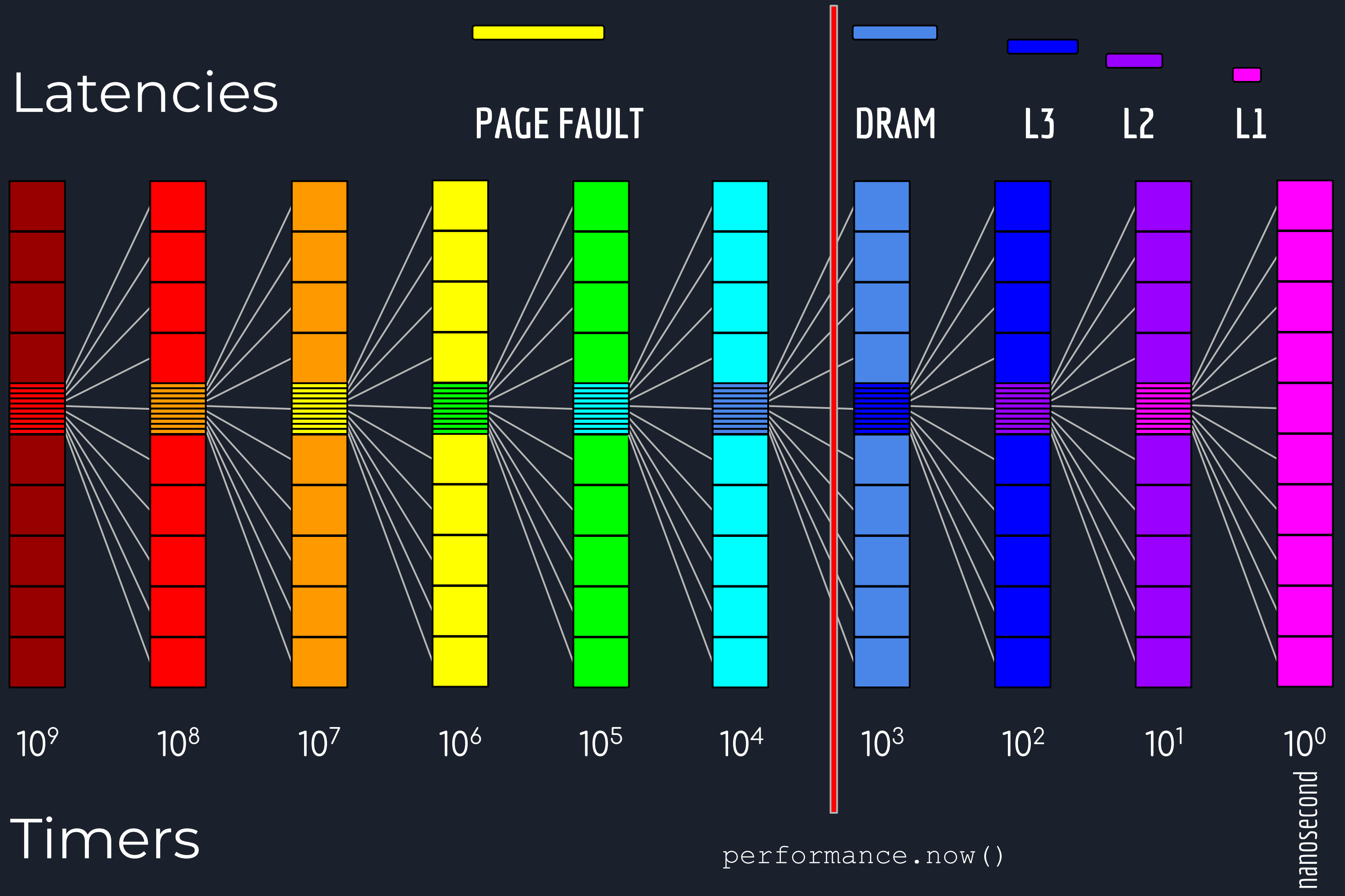
```
byte read_memory(uint64 address);
```

that uses side-channels to read from any memory location in the entire address space

A (much) closer look at timing channels



Latencies



Timers

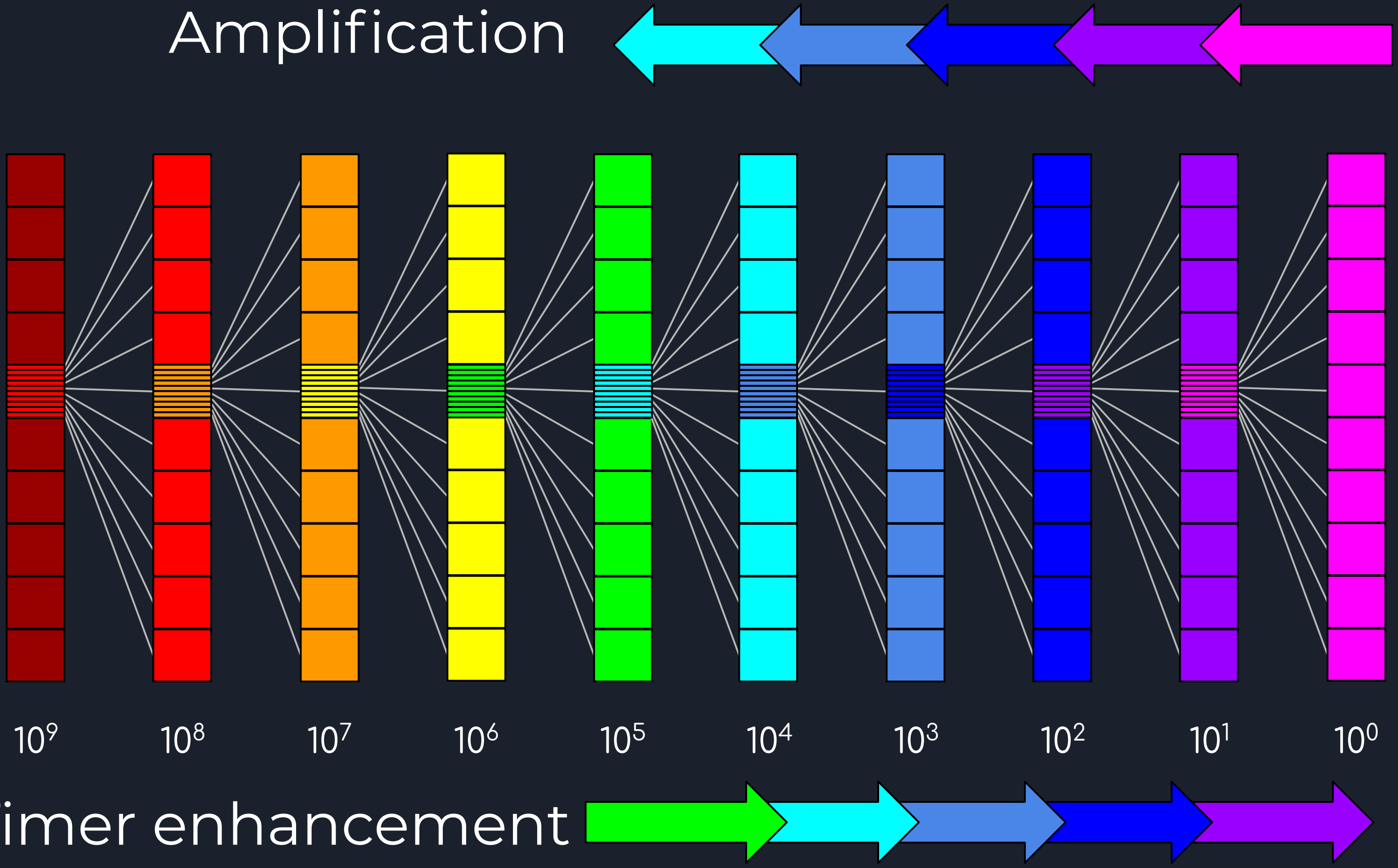
Timers in JavaScript

	Free-running	Firefox 51	Chrome 53	Edge 38	Tor 6.0.4	Fuzzyfox
performance.now	✓	5 µs	5 µs	1 µs	100 ms	100 ms
CSS animations	✓	16 ms	16 ms	16 ms	16 ms	125 ms
setTimeout		4 ms	4 ms	2 ms	4 ms	100 ms
setImmediate		–	–	50 µs	–	–
postMessage		45 µs	35 µs	40 µs	40 µs	47 ms
Sub worker	✓	20 µs	–[2]	50 µs	15 µs	–
Broadcast Channel	✓	145 µs	–	–	55 µs	760 µs
MessageChannel		12 µs	55 µs	20 µs	20 µs	45 ms
MessageChannel (W)	✓	75 µs	100 µs	20 µs	30 µs	1120 µs
SharedArrayBuffer	✓	2 ns[3]	15 ns[4]	–	–	2 ns
Interpolation[1]		500 ns	500 ns	350 ns	15 µs	–
Edge thresholding[1]		2 ns	15 ns	10 ns	2 ns	–

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

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

Global mutable variable

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

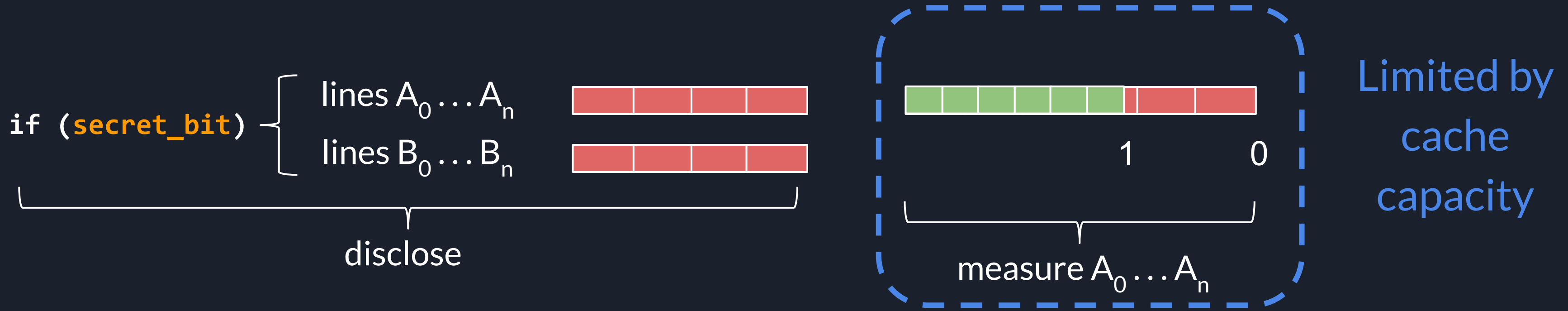
Dedicated timer thread

```
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**



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

Variant 1 - Mitigations

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

Boundscheck bypass vulnerability

Disable speculation altogether?

Variant 1 - Mitigations

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

Early Intel recommendation

Boundscheck bypass vulnerability

Variant 1 - Mitigations

2.8X

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

LFENCE mitigation

Variant 1 - WebAssembly Mitigation

~15%

```
if (index < 0x10000) { // memory size
    int value = memory_base + index & 0xFFFF;
}
```

WebAssembly memory masking mitigation

Variant 1 - Pervasive poisoning

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

Potentially vulnerable condition in attacker code

```
var poison = -1;
if (condition) {
    poison &= -condition;
    var x = load(...);
    x = x & poison;
    leak(x);
}
```

Poison operations inserted by V8 optimizing compiler

- Prologue
- Branches
- Loads

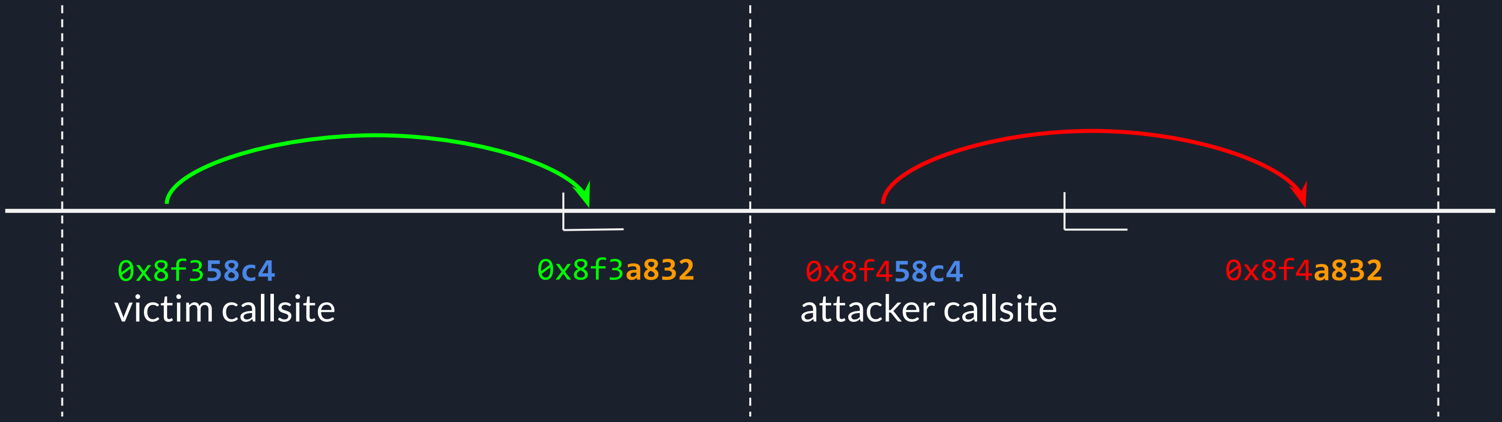
Variant 2 - Branch Target Injection

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

Potentially vulnerable indirect call in code

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

Aliasing indirect call in attacker code



Variant 2 - Function prologue poisoning

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

Potentially vulnerable indirect call

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

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

Poison operations inserted by V8 optimizing compiler

- Prologue
- Branches
- Loads
- Interpreter

Variant 4 - Speculative Store Bypass

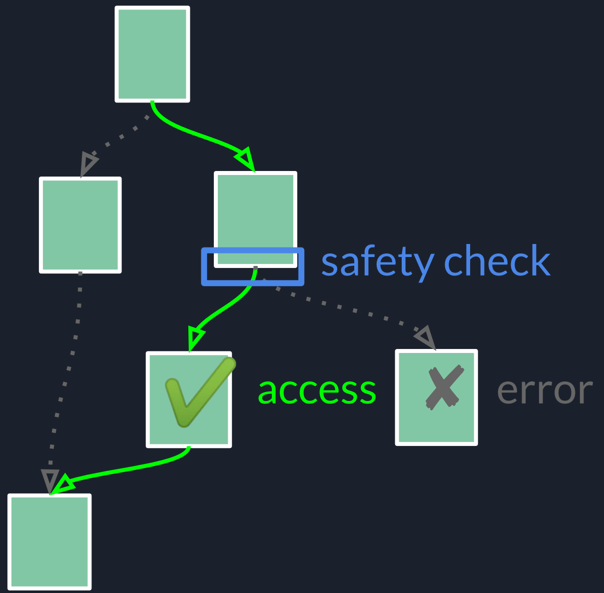
- Page tables
- DRAM line buffer
- Instruction and data caches: L1i, L1d, L2, L3 + prefetchers
- Translation lookaside buffer: L1 and L2
- Branch prediction state (BHB, BTB)
- Return stack buffer
- Micro-op cache
- Model-specific registers

Spectre V4

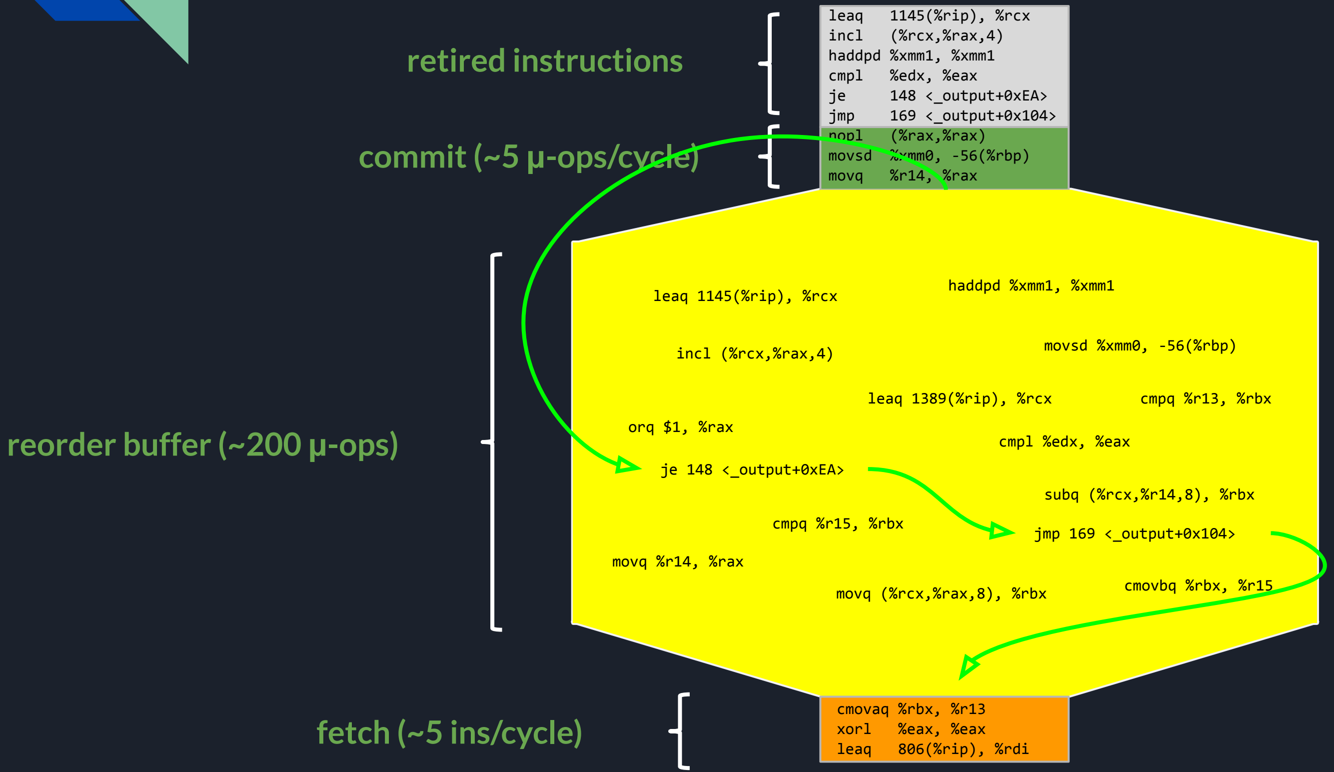
- Store buffer
- Memory disambiguator
- Frequency scaling (power save mode, TurboBoost)
- Execution port occupancy
- Lazy FPU state
- Bus and cache fill line occupancy

Out of Order Execution: now observable!

Out of Order Execution > Branch Prediction

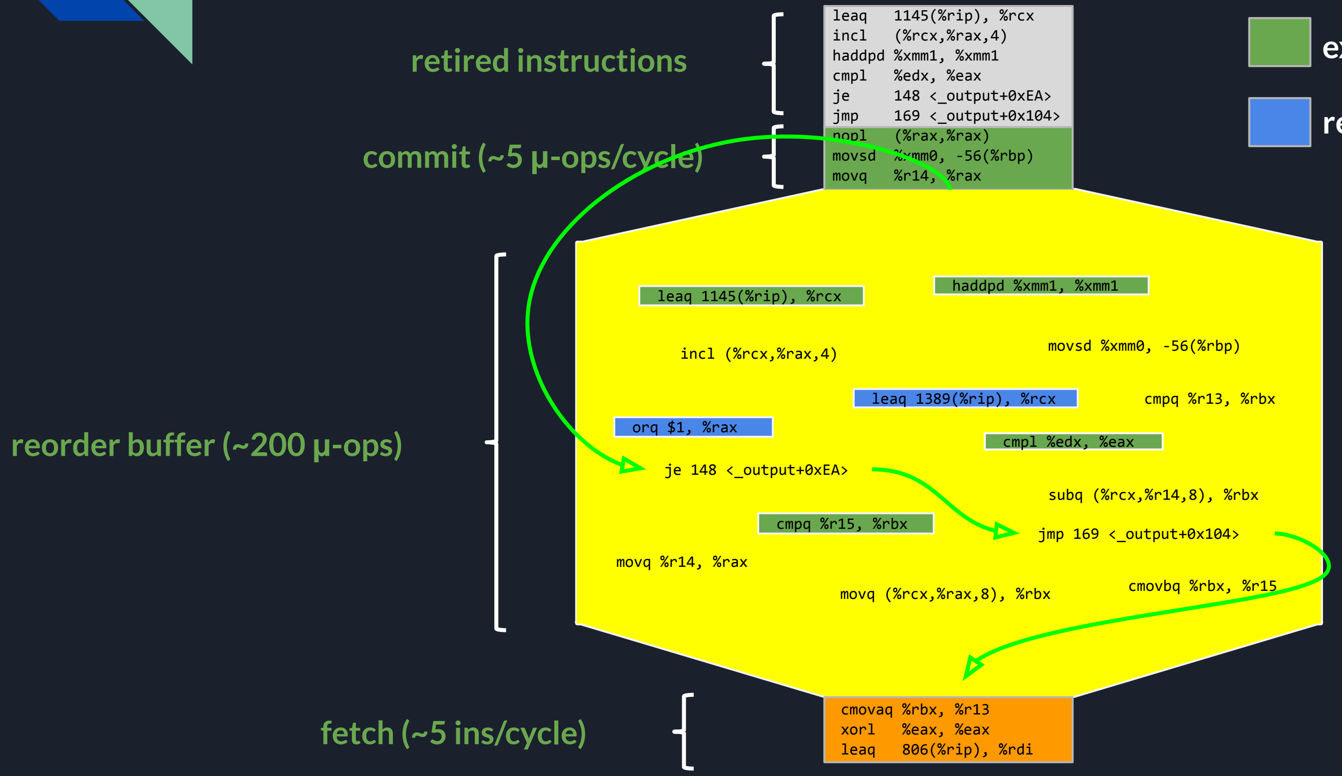


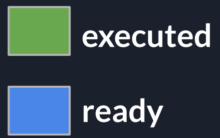
Out of Order Execution



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

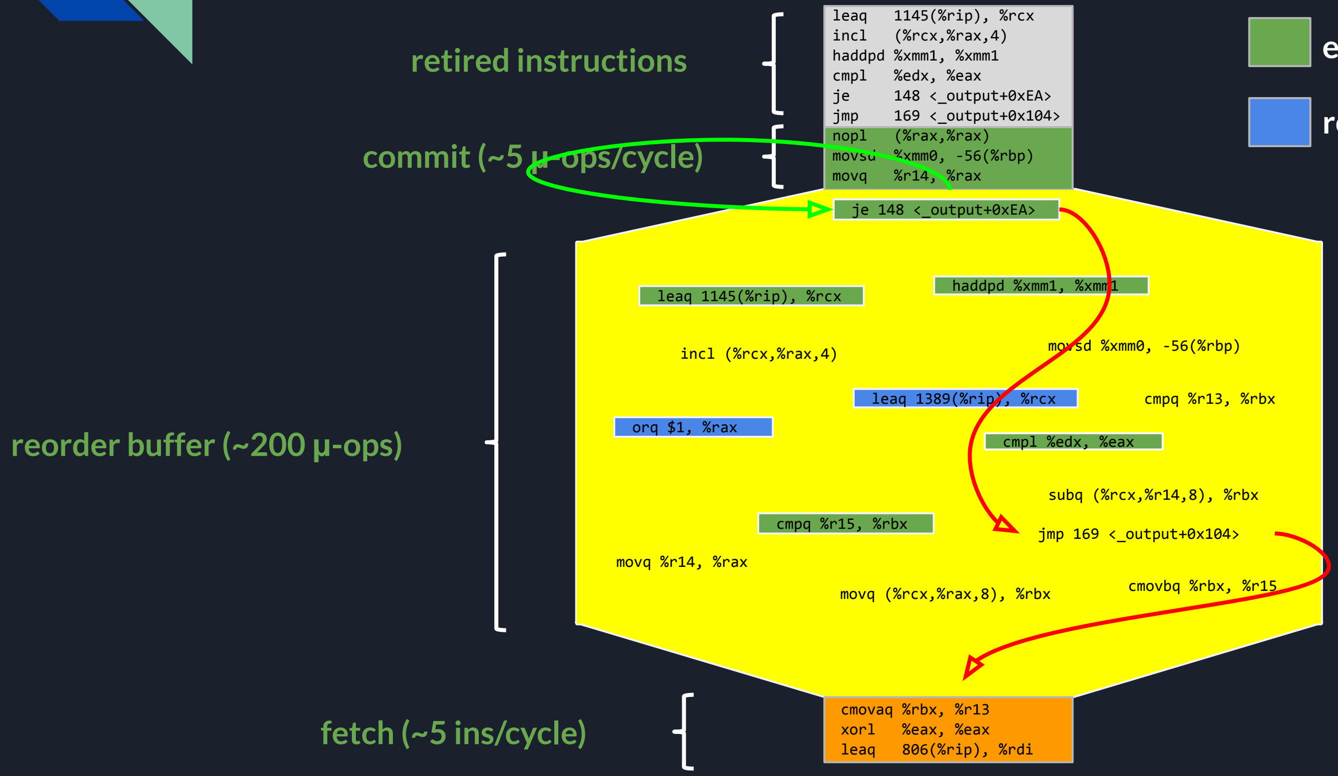
Out of Order Execution

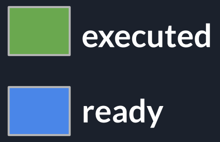




Instructions in the reorder buffer execute based on dataflow (register) dependency graph.

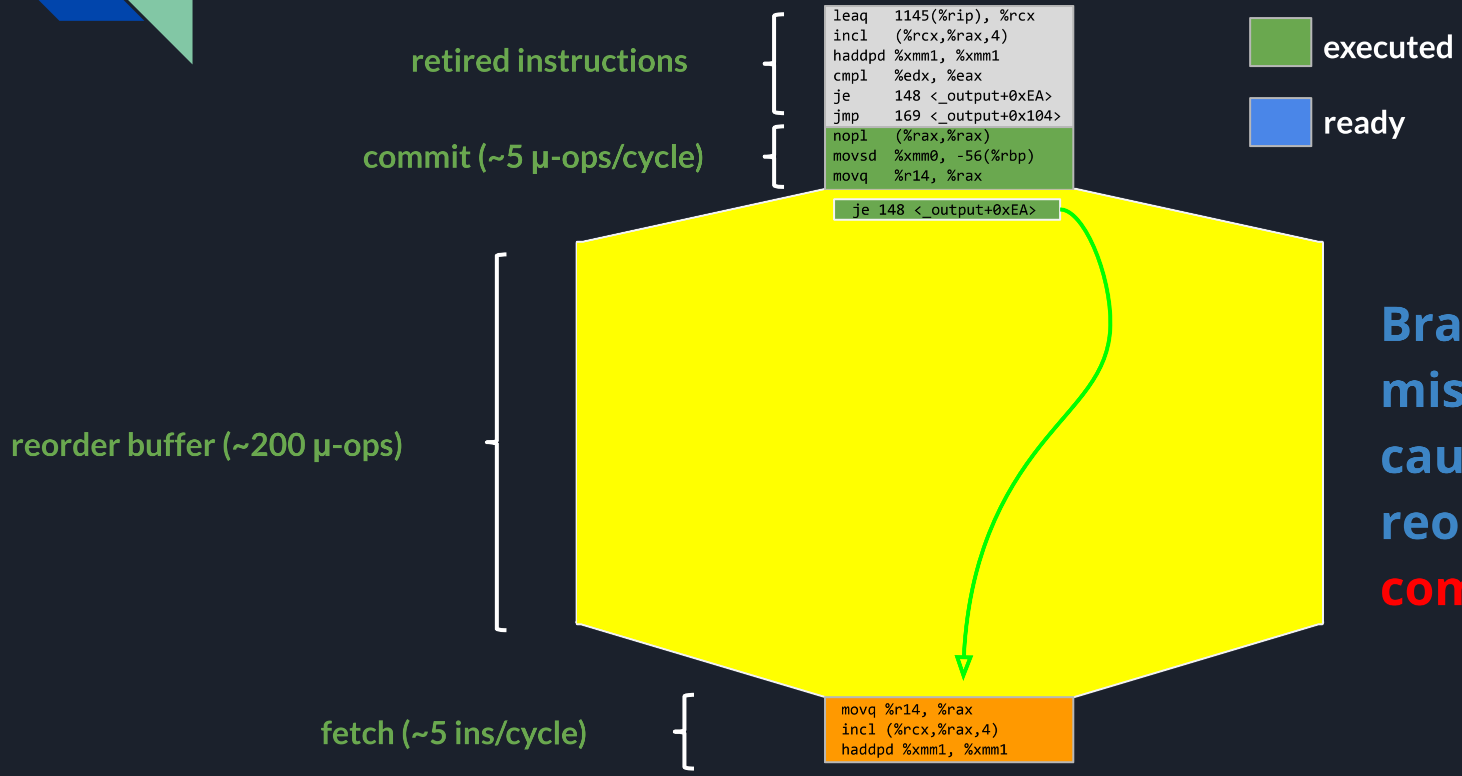
Out of Order Execution





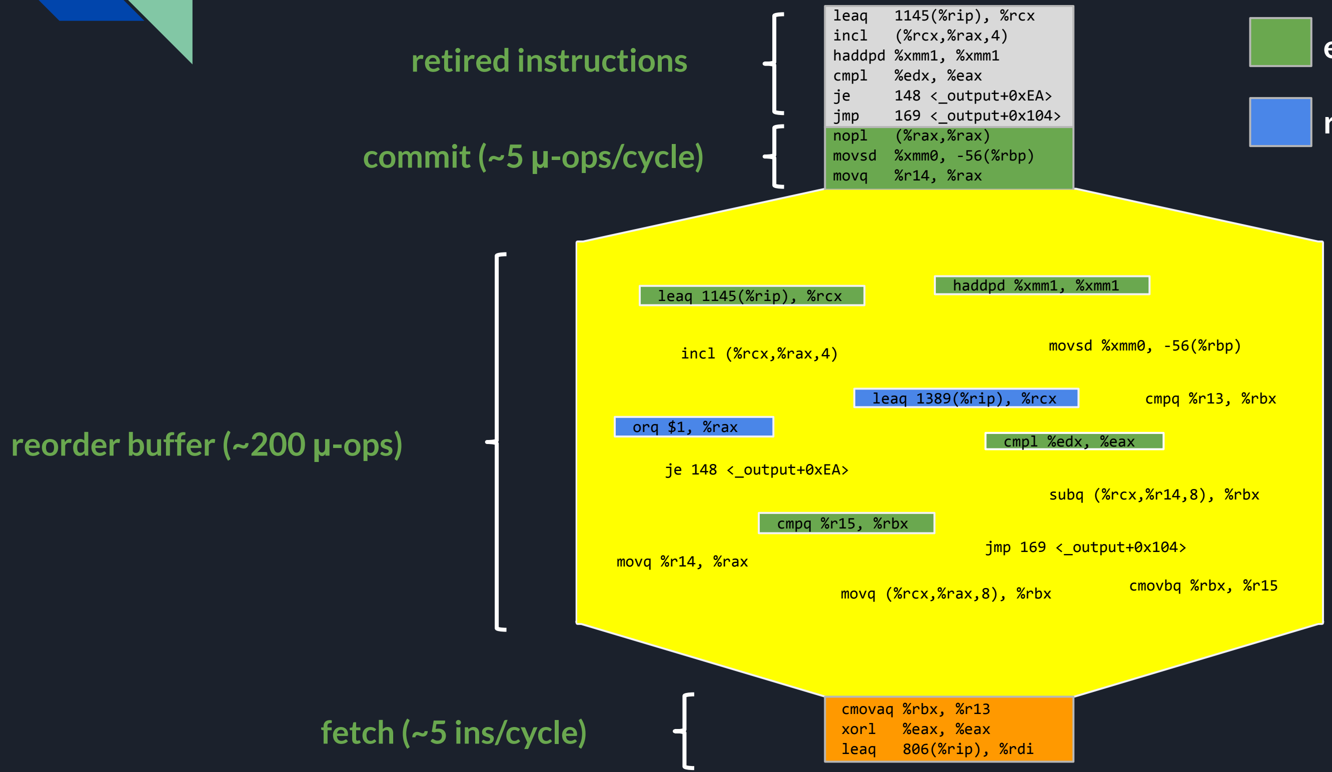
Branch predictions are verified at instruction commit.

Out of Order Execution



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

Out of Order Execution



executed

ready

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

store

```
…
mov [x], #0
mov rax, [y]
…
```

load

✔ Any order OK if `x != y`

Speculative Store Bypass Vulnerability

stale

store

load

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

Original code

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

Out-order-execution

✔ Still OK if `x == y`

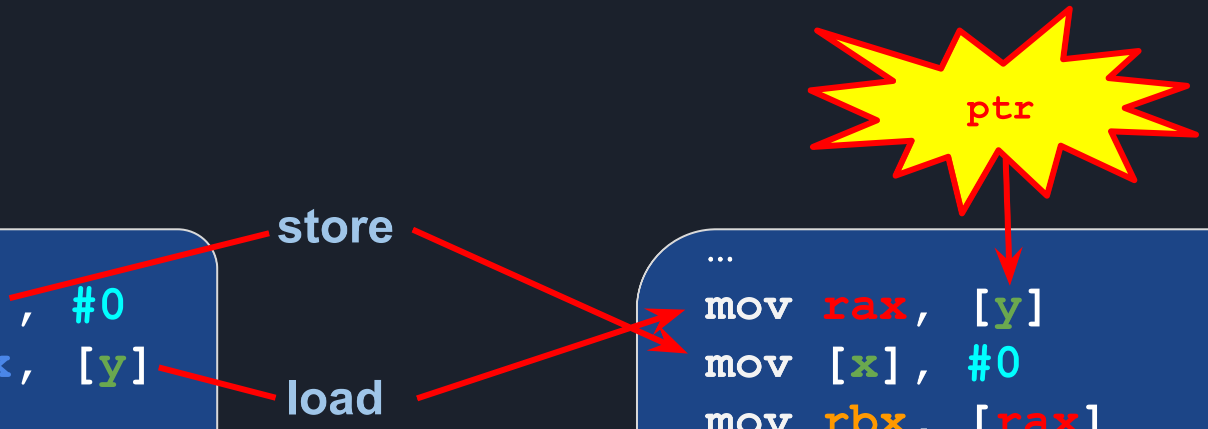
✘ Incorrect if `x == y`

Speculative Store Bypass Vulnerability

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

Original code

✓ Still OK if `x == y`




```
...
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

Spectre L1TF

RowHammer

TLBBleed

MDS

Spectre V4

Spectre V1 & V2

Spectre Returns

PortSmash

Rogue System Register Read

Spectre Lazy FPU

The Universal Read Gadget

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

```
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

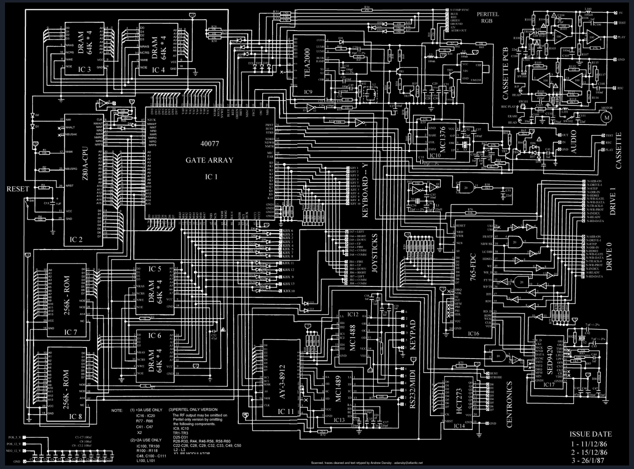
Source Language

```
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
cmpq   %r15, %rbx
cmovbq %rbx, %r15
cmpq   %r13, %rbx
cmovaq %rbx, %r13
xorl   %eax, %eax
leaq   806(%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 run) 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
  ⇒ It's possible to construct the universal read gadget in any language.

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"