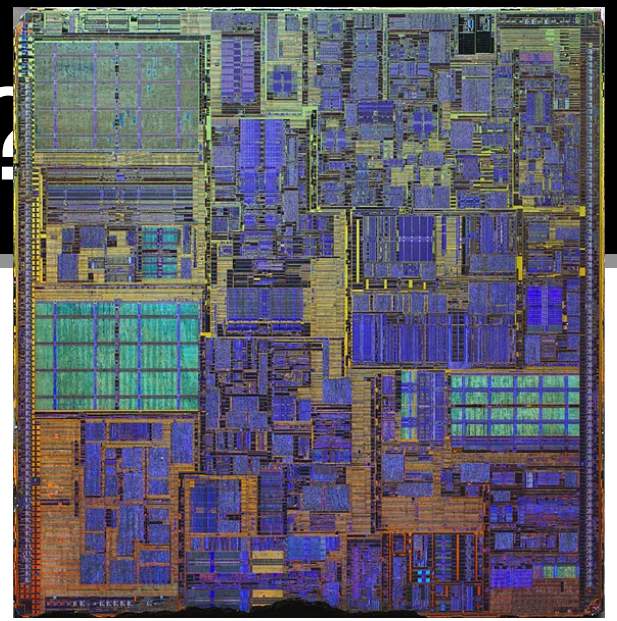
Spectre Attacks: Exploiting Speculative Execution

IEEE Security & Privacy (May 20, 2019)

Paul Kocher[1], Jann Horn[2], Anders Fogh[3], Daniel Genkin[4], Daniel Gruss[5], Werner Haas[6], Mike Hamburg[7], Mortiz Lipp[5], Stefan Mangard[5], Thomas Prescher[6], Michael Schwartz[5], Yuval Yarom[8]

[1] Independent, [2] Google Project Zero, [3] G DATA Advanced Analytics, [4] University of Pennsylvania and University of Maryland,
[5] Graz University of Technology, [6] Cyberus Technology, [7] Rambus, Cryptography Research Division, [8] University of Adelaide & Data61

How to boost CPU performance?



No more easy gains from low-level physics, e.g.:

- Increase clock rates — Mostly maxed out (3.8 GHz Pentium 4 in 2004)
- Improve memory speeds — DRAM latency huge, not improving much

Industry focus on pipelining + boosting average-case performance, e.g.:

- Reducing memory delays → Caches
- Working during delays → Speculative execution

Computer architecture: *n.* The art and science of introducing new side channel vulnerabilities.

Speculative execution

Programs are expressed sequentially
… but fast CPUs leverage HW's parallelism (pipelining…) and speculation

Speculation: Start likely tasks early, then clean up errors.

Example:

```
if (x == 1) {
    abc…
} else {
    xyz…
}
```

If **x** is uncached, processor faces a long delay

CPU can guess execution path & proceed speculatively

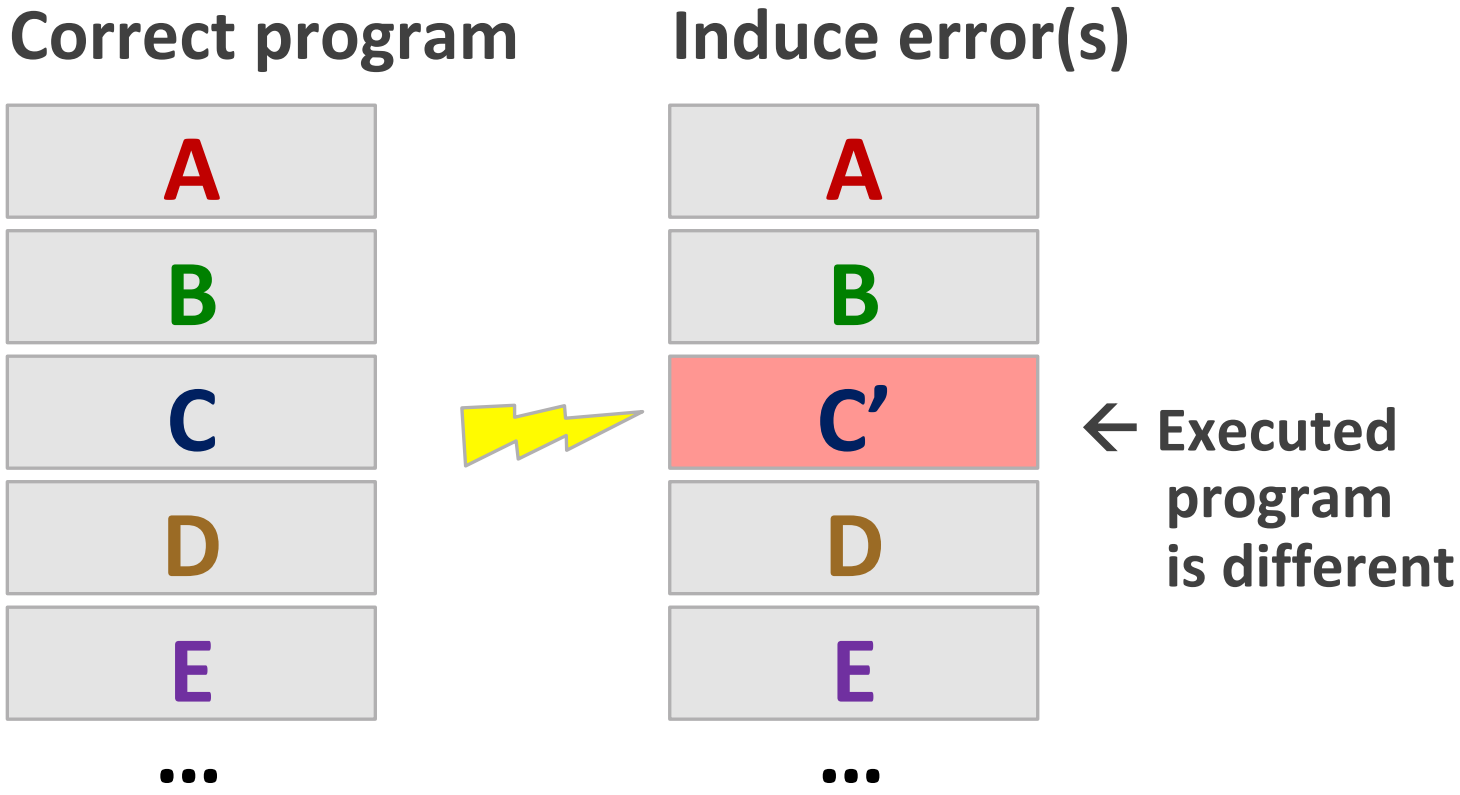
When **x** arrives from DRAM, check if guess was correct

- Correct: commit speculative work = performance gain
- Wrong guess: Discard faulty work

Fault attacks

Secure programs are unsafe if executed erroneously

Example: Induce analog glitches on clock, reset, power/ground…



Almost any kind of error is exploitable

Are there any security implications from speculative execution? *-- Mike Hamburg*

CPU is secretly making errors on its own

≈ fault attack hardware is built-in

Faulty results are discarded, but CPUs are riddled with side/covert channels
(… much simpler than combined fault+differential power analysis)

Conditional branch (Variant 1) attack

```
if (x < array1_size)
    y = array2[array1[x]*512];
```

Attack scenario:

- Code runs in a trusted context
- Adversary wants to read memory and controls unsigned integer `x`
- Branch predictor will expect `if()` to be true (e.g. because prior calls had `x` < `array1_size`)
- `array1_size` and `array2[]` are not in cache

Memory & Cache Status

`array1_size = 00000008`

Memory at `array1` base address:

8 bytes of data (value doesn't matter)

[… lots of memory up to `array1` base+N…]

09 F1 98 CC 90... (something secret)

```
array2[ 0*512]
array2[ 1*512]
array2[ 2*512]
array2[ 3*512]
array2[ 4*512]
array2[ 5*512]
array2[ 6*512]
array2[ 7*512]
array2[ 8*512]
array2[ 9*512]
array2[10*512]
array2[11*512]
  •••
```

Contents don't matter

only care about cache ***status***

Uncached Cached

Conditional branch (Variant 1) attack

```
if (x < array1_size)
    y = array2[array1[x]*512];
```

Attacker calls victim code with `x`=N (where N > 8)

- Speculative exec while waiting for `array1_size`
  - Predict that if() is true
  - Read address (`array1` base + `x`) w/ out-of-bounds `x`
  - Read returns secret byte = **09** (fast – in cache)

Memory & Cache Status

`array1_size = 00000008`

Memory at `array1` base address:

8 bytes of data (value doesn't matter)

[… lots of memory up to `array1` base+N…]

09 F1 98 CC 90... (something secret)

```
array2[ 0*512]
array2[ 1*512]
array2[ 2*512]
array2[ 3*512]
array2[ 4*512]
array2[ 5*512]
array2[ 6*512]
array2[ 7*512]
array2[ 8*512]
array2[ 9*512]
array2[10*512]
array2[11*512]
  •••
```

Contents don't matter

only care about cache ***status***

Uncached Cached

Conditional branch (Variant 1) attack

```
if (x < array1_size)
    y = array2[array1[x]*512];
```

Attacker calls victim code with `x`=N (where N > 8)

- Speculative exec while waiting for `array1_size`
  - Predict that if() is true
  - Read address (`array1` base + `x`) w/ out-of-bounds `x`
  - Read returns secret byte = **09** (fast – in cache)
  - Request memory at (`array2` base + **09***512)
  - Brings `array2[09*512]` into the cache
  - Realize if() is false: discard speculative work
- Finish operation & return to caller

Attacker times reads from `array2[i*512]`

- Read for `i`=**09** is fast (cached), revealing secret byte

Memory & Cache Status

`array1_size = 00000008`

Memory at `array1` base address:

8 bytes of data (value doesn't matter)

[… lots of memory up to `array1` base+N…]

09 F1 98 CC 90... (something secret)

```
array2[ 0*512]
array2[ 1*512]
array2[ 2*512]
array2[ 3*512]
array2[ 4*512]
array2[ 5*512]
array2[ 6*512]
array2[ 7*512]
array2[ 8*512]
array2[ 9*512]
array2[10*512]
array2[11*512]
  ...
```

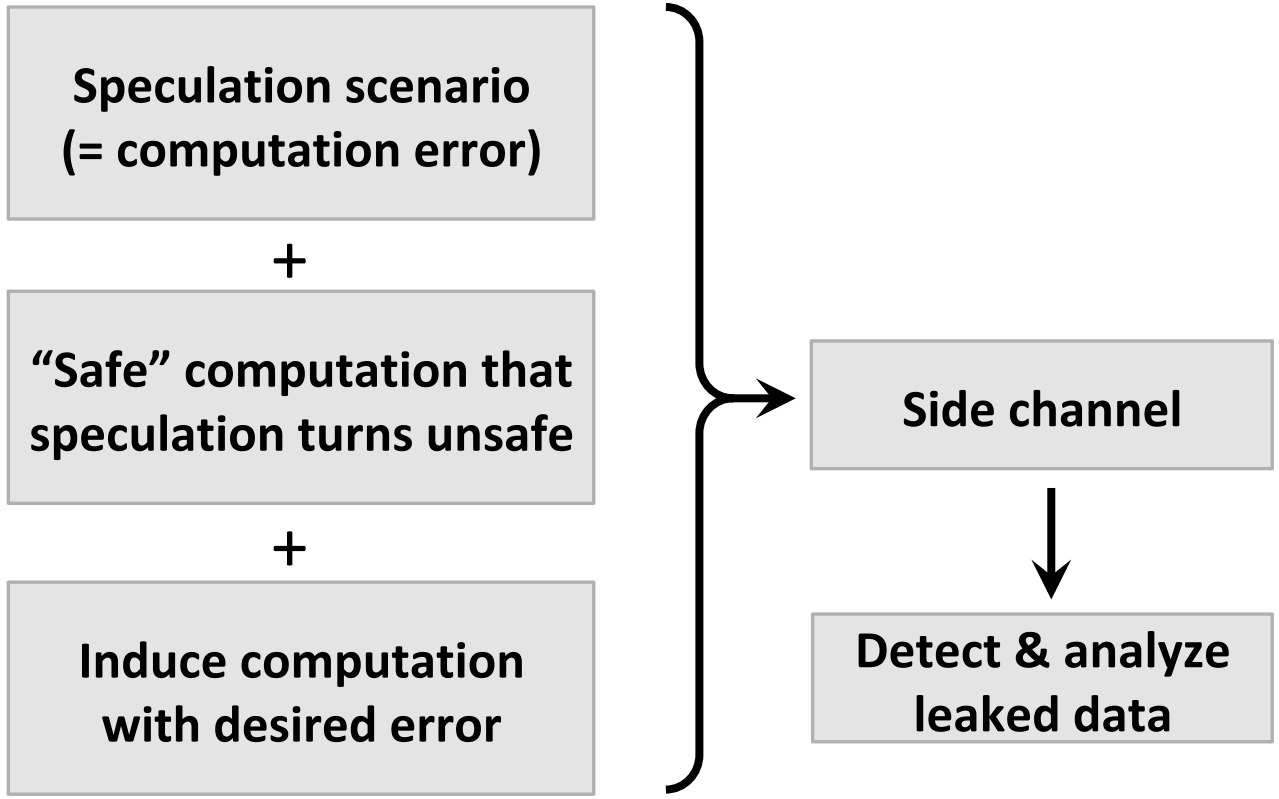
Contents don't matter

only care about cache ***status***

Uncached Cached

Spectre is a messy class of vulnerabilities

Many possible variations



Many related results

- Speculative Store Bypass/Variant 4
- NetSpectre
- Foreshadow
- Spectre1.1
- Spectre-NG
- Rogue System Register Read
- Speculative Store Bypass (SSB)
- LazyFP (Lazy FPU state leak)
- ret2spec
- SpectreRSB

+ more to come

Is Spectre a bug?

Everything complies with the architecture specs

- Branch predictor is learning from history, as expected
- Speculative execution unwinds architectural state correctly
- Reads are fetching data the victim is allowed to read
- Caches are allowed to hold state
- Covert channels & side channels are well known

?!

Spectre is a symptom

Symptom of excessive architectural ambiguity

- Typical architectures' guarantees are insufficient for security

  E.g. no promise to keep *anything* secret from other processes? Across intra-process domains?

- Consequence: software developers to rely on guesses

  Hopeless for developer: even if tested on all chips today, future chips may be different

- Key research topic: What should architectures guarantee?

  Minimum requirement: Sufficient for secure software

  Metric: likelihood final system (HW+SW) will be secure

  … given realistic assumptions about SW+HW development practices

  Challenges: performance, power, legacy compatibility, die area…

Step 1: Tell programmers to add LFENCE instructions wherever something could go wrong (and nowhere else because LFENCE is really slow)

…

Step n: Blame programmer

Spectre is a symptom

History of prioritizing performance, legacy compatibility, … over security

- Scaling issue: As complexity grows, security risks increase faster than benefits
- Balance has shifted for many applications: value of performance gains << insecurity costs
- Latency in changing mindsets: Dominant people and businesses grew up when performance > security

Need to specialize designs for **performance** vs. **security**

- Can co-exist on the same chip
  (analogous to ARM's big.LITTLE for power)
- Security = much less complex TCB (HW+SW),
  not just a different mode (like TrustZone/SGX)

VS

Q&A

If the surgery proves unnecessary, we'll revert your architectural state at no charge.