CS 5435

Microarchitectural Attacks

Vitaly Shmatikov

Credit for many slides: Stanford CS 155

Performance in Modern CPUs

Clock speed maxed out:

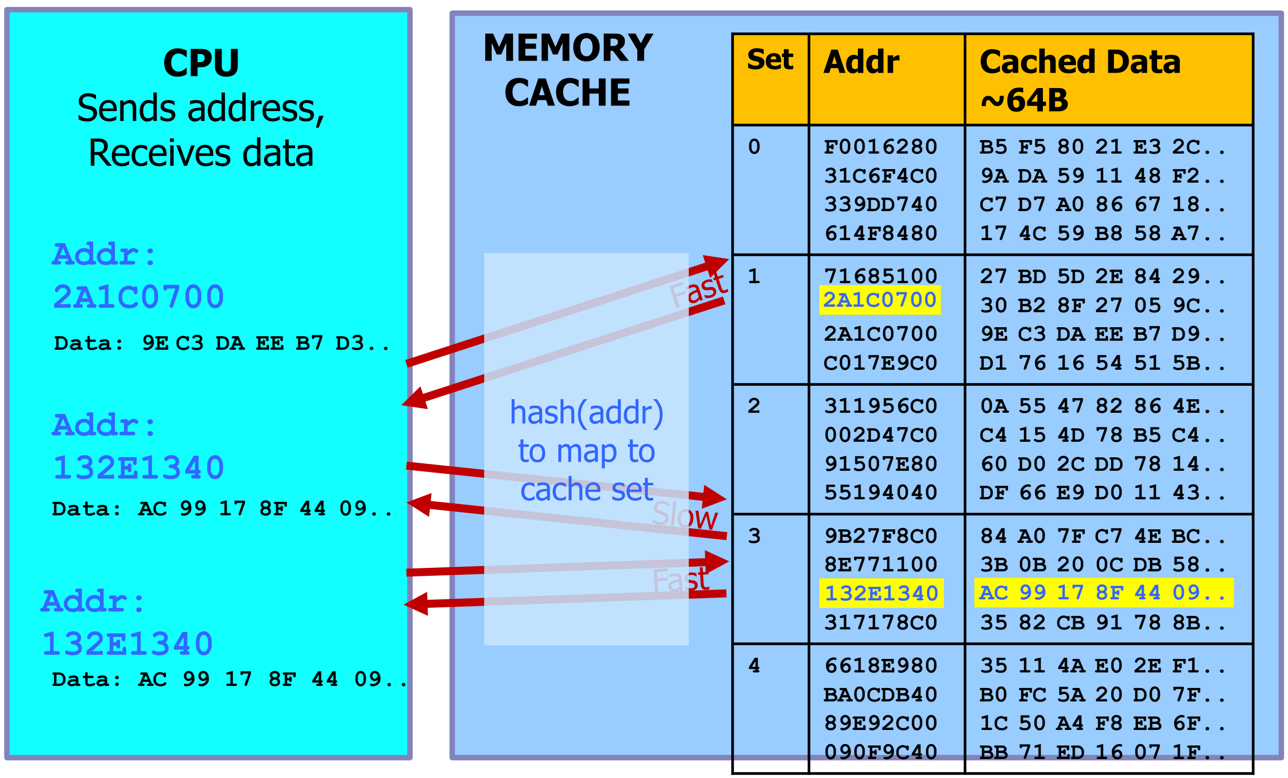
- Pentium 4 reached 3.8 GHz in 2004
- Memory latency is slow and not improving much

To gain performance, need to do more per cycle!

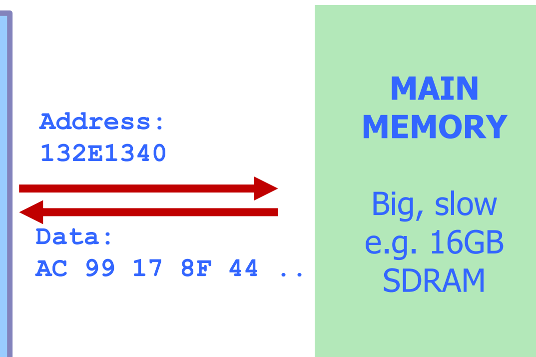
- Reduce memory delays: caches
- Work during delays: speculative execution

Memory Caches

Caches hold local (fast) copy of recently accessed 64-byte chunks of memory



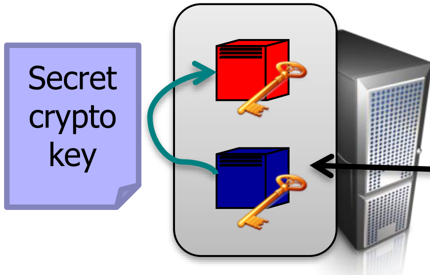
Set	Addr	Cached Data ~64B
0	F0016280	B5 F5 80 21 E3 2C..
	31C6F4C0	9A DA 59 11 48 F2..
	339DD740	C7 D7 A0 86 67 18..
	614F8480	17 4C 59 B8 58 A7..
1	71685100	27 BD 5D 2E 84 29..
	2A1C0700	30 B2 8F 27 05 9C..
	2A1C0700	9E C3 DA EE B7 D9..
	C017E9C0	D1 76 16 54 51 5B..
2	311956C0	0A 55 47 82 86 4E..
	002D47C0	C4 15 4D 78 B5 C4..
	91507E80	60 D0 2C DD 78 14..
	55194040	DF 66 E9 D0 11 43..
3	9B27F8C0	84 A0 7F C7 4E BC..
	8E771100	3B 0B 20 0C DB 58..
	132E1340	AC 99 17 8F 44 09..
	317178C0	35 82 CB 91 78 8B..
4	6618E980	35 11 4A E0 2E F1..
	BA0CDB40	B0 FC 5A 20 D0 7F..
	89E92C00	1C 50 A4 F8 EB 6F..
	090F9C40	BB 71 ED 16 07 1F..



Reads change system state:

- Read to newly-cached location is fast
- Read to evicted location is slow

Cross-VM Side Channel



Target is 4096-bit ElGamal secret key e

Modular Exponentiation (x, e, N):

let e_n … e_1 be the bits of e

y ← 1

for e_i in {e_n …e_1}

    y ← **Square**(y)   **(S)**

    y ← **Reduce**(y, N)   **(R)**

    if e_i = 1 then

        y ← **Multi**(y, x)   **(M)**

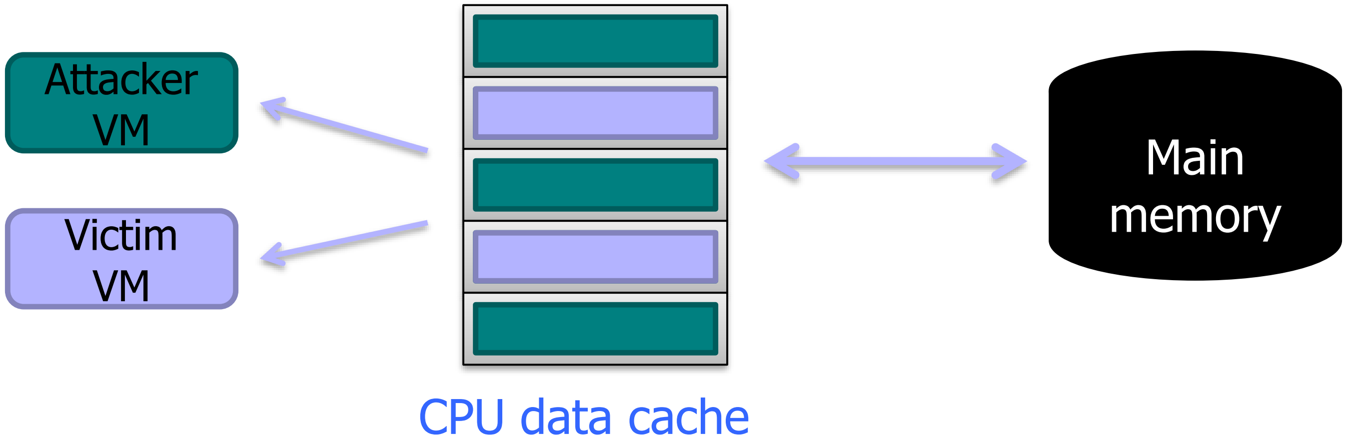
        y ← **Reduce**(y, N)   **(R)**

return y   // $y = x^e \bmod N$

e_i = 1 → “SRMR”
e_i = 0 → “SR”

Sequence of function calls reveals secret key

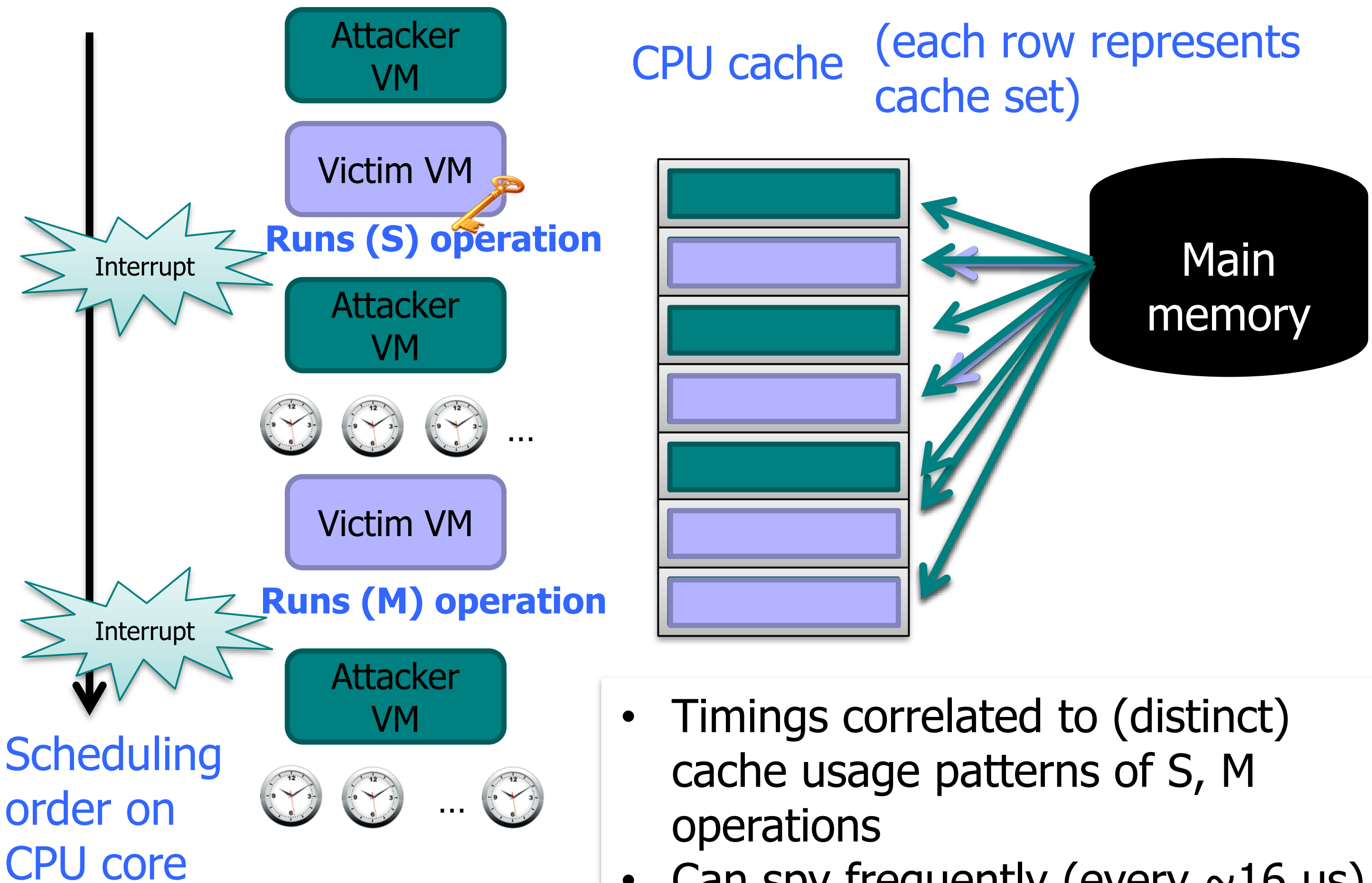
Cache Contention



1) Read in a large array (fill CPU cache with attacker data)
2) Busy loop (allow victim to run)
3) Measure time to read large array (the load measurement)

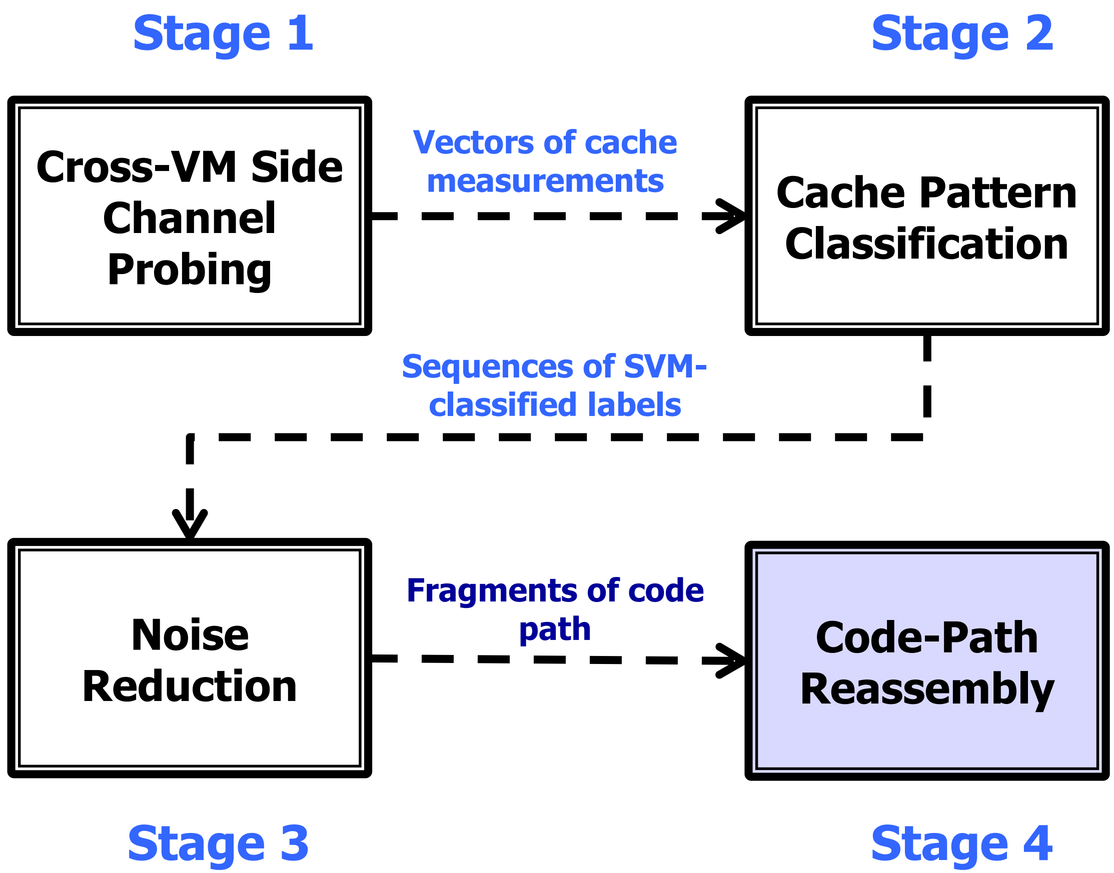
Locations in cache occupied by victim will take longer to load ➔ Information about victim's use of cache revealed to attacker

Prime + Probe



- Timings correlated to (distinct) cache usage patterns of S, M operations
- Can spy frequently (every ~16 μs) by exploiting scheduling

Attack Stages



Prime + Probe Feasibility

Setup for in-lab experimentation:

- Intel Yorkfield processor (4 cores, 32KB L1 instruction cache)
- Xen + Linux + GnuPG + libgcrypt

Best result:

- 300,000,000 prime-probe results (6 hours)
- Over 300 key fragments
- Brute force the secret key in ~9800 guesses

Not practical in deployment settings

Microarchitectural Side Channels

Lots of research

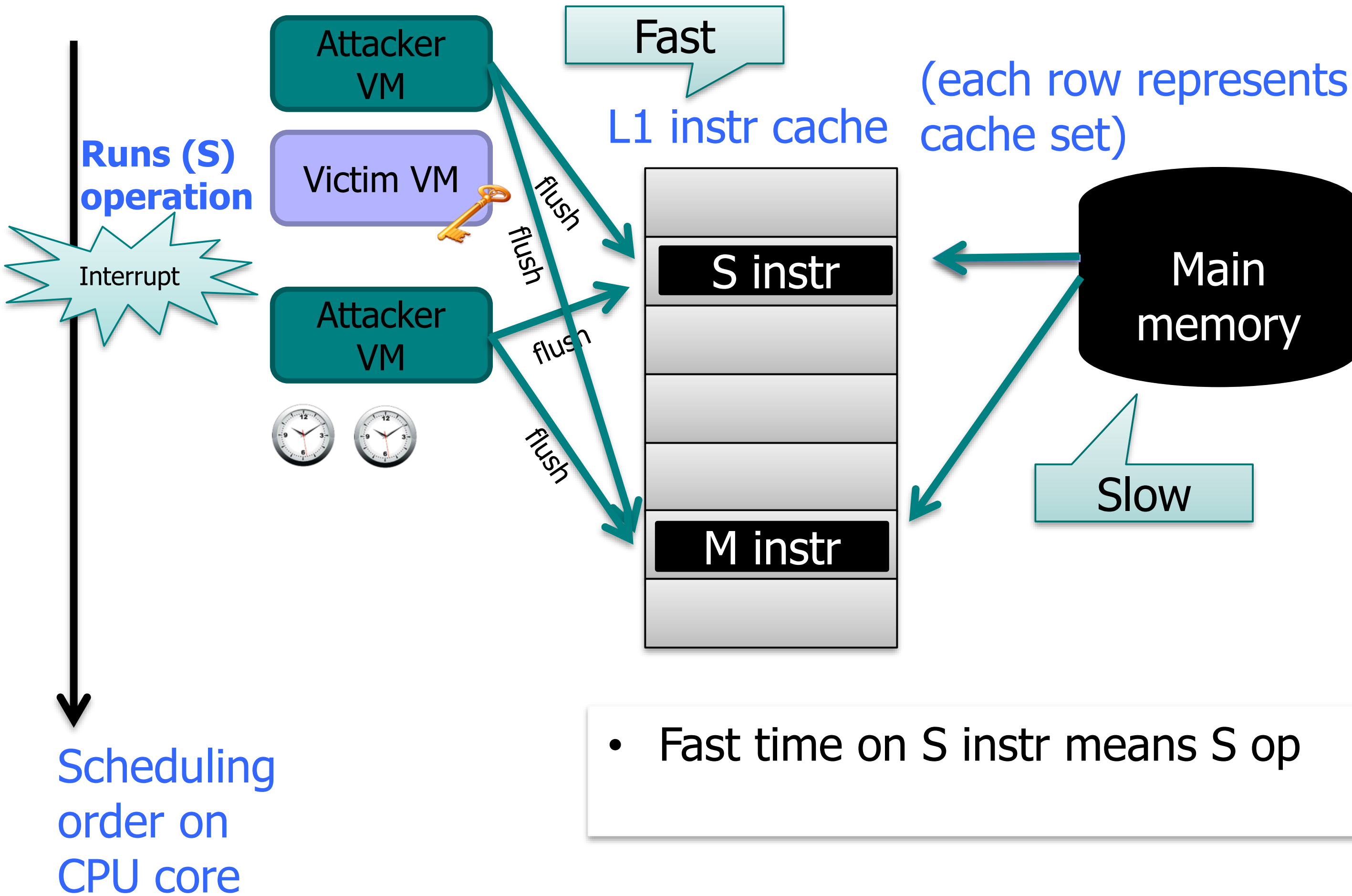
State-of-the-art Prime+Probe attacks

- Sinan Inci et al. 2016 “Cache Attacks Enable Bulk Key Recovery on the Cloud”

Flush+Reload more robust side channel in shared memory settings [Yarom, Falkner 2013]

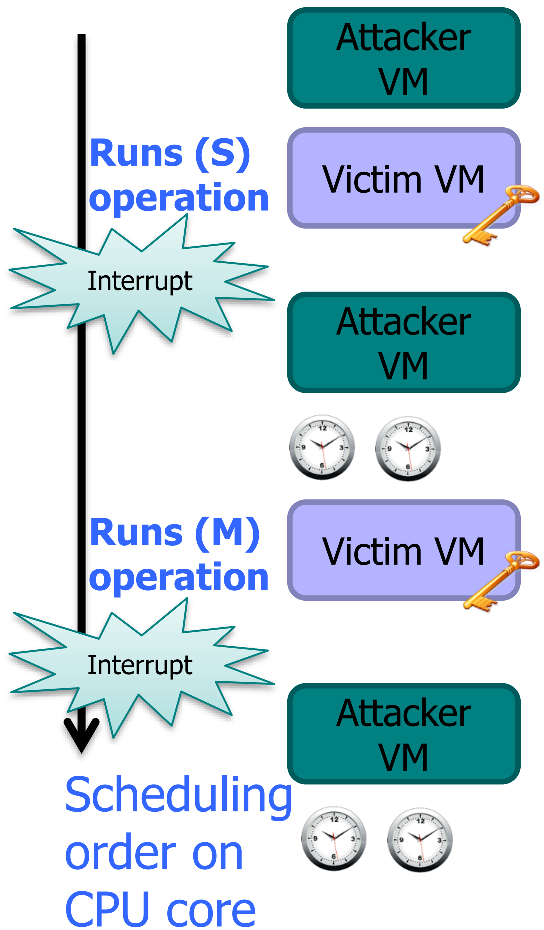
- Spy process flushes memory shared with victim from caches
- Idles
- Times how long it takes to read shared memory

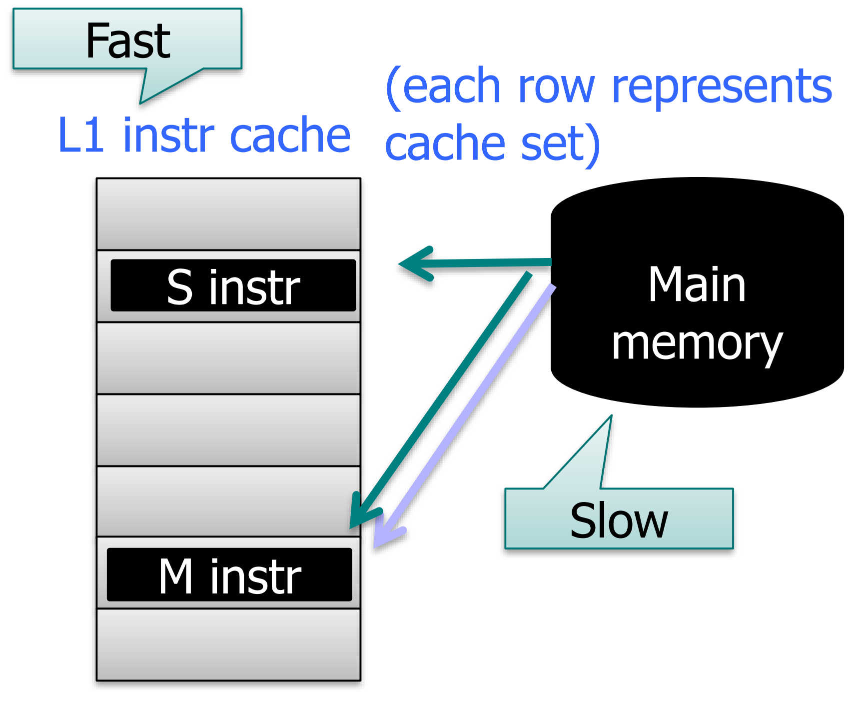
Flush + Reload



- Fast time on S instr means S op

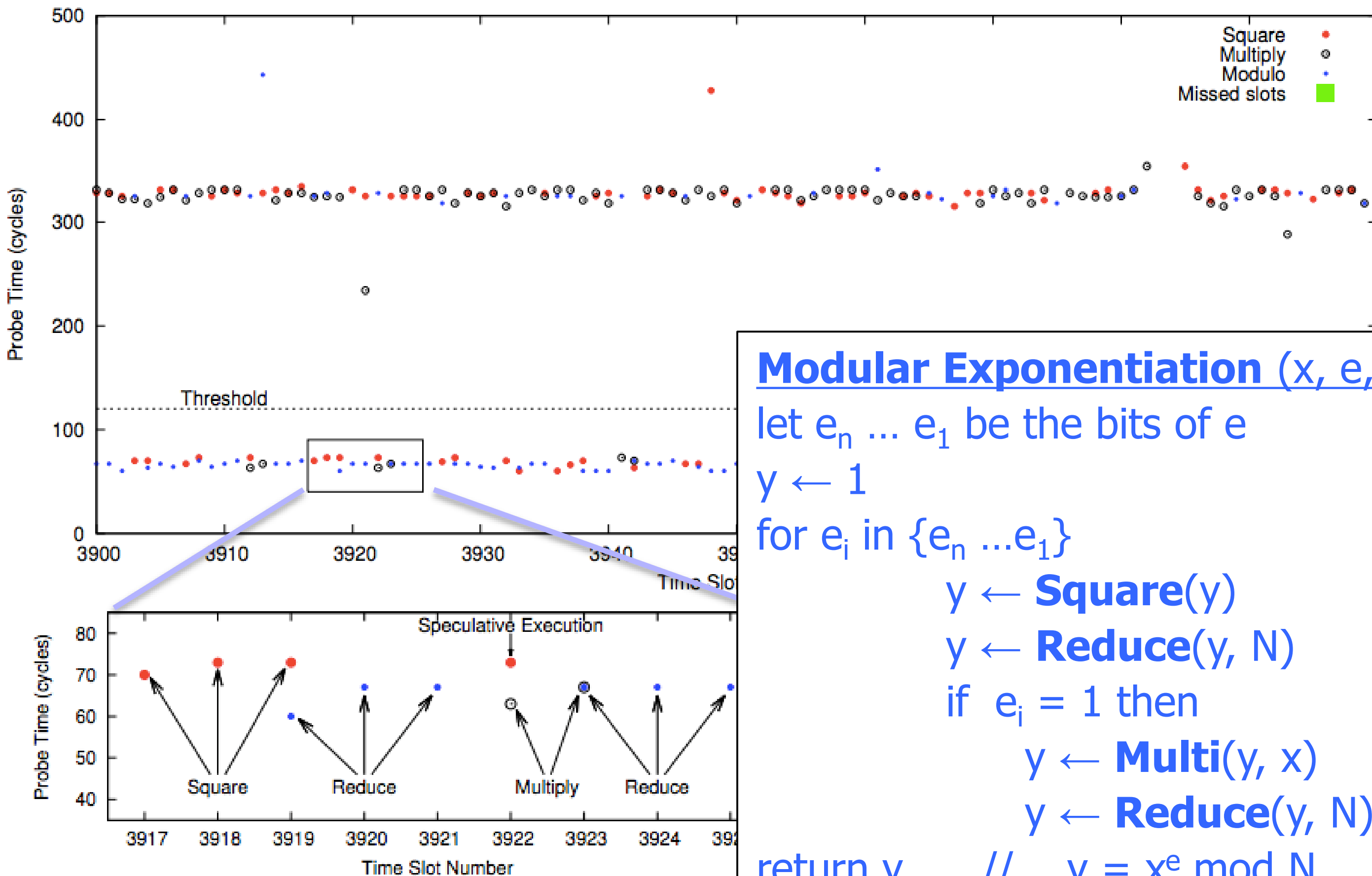
Flush + Reload





- Fast time on S instr means S op
- Fast time on M instr means M op

Attacking Square-and-Multiply



Modular Exponentiation (x, e, N):

let $e_n \dots e_1$ be the bits of e

$y \leftarrow 1$

for e_i in $\{e_n \dots e_1\}$

  $y \leftarrow$ **Square**(y) **(S)**

  $y \leftarrow$ **Reduce**(y, N) **(R)**

  if $e_i = 1$ then

    $y \leftarrow$ **Multi**(y, x) **(M)**

    $y \leftarrow$ **Reduce**(y, N) **(R)**

return y // $y = x^e \bmod N$

Speculative Execution

CPUs can guess likely program path and do <u>speculative execution</u>

- Example:

```
if  (uncached_value == 1)     // load from memory
      a = compute(b)
```

- Branch predictor guesses if() is 'true' (based on prior history)
- Starts executing compute(b) speculatively

- When value arrives from memory, check if guess was correct:
  - Correct: Save speculative work ⇒ performance gain
  - Incorrect: Discard speculative work ⇒ no harm (?)

Problem: Side Effects

Architectural Guarantee

Register values eventually match the result of in-order execution

Speculative Execution

CPU regularly performs incorrect calculations, then deletes mistakes

Is making + discarding mistakes the same as in-order execution?

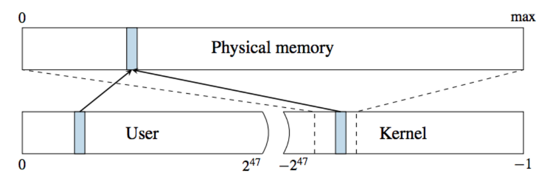
The processor executed instructions that were not supposed to run !!

The problem: instructions can have observable side-effects

Spectre and Meltdown

Speculative execution bugs in Intel x86, ARM, IBM processors + cache-based side-channels (F+R)

- Allows reading kernel (or hypervisor, other VM) memory
- Similar attacks on SGX, etc.



Intel didn't warn US government about CPU security flaws until they were public

Meltdown and Spectre were kept secret

Researchers find malware samples that exploit Meltdown and Spectre

As of Feb. 1, antivirus testing firm AV-TEST had found 139 malware samples that exploit Meltdown and Spectre. Most are not very functional, but that could change.

Conditional Branch (Var 1) Attack

```
if (x < array1_size)
   y = array2[array1[x]*4096];
```

Suppose `unsigned int x` comes from untrusted caller

Execution without speculation is safe:

`array2[array1[x]*4096]` not eval unless `x < array1_size`

What about with speculative execution?

Conditional Branch (Var 1) Attack

```
if (x < array1_size)
    y = array2[array1[x]*4096];
```

Before attack:

- Train branch predictor to expect if() is true (e.g. call with `x < array1_size`)
- Evict `array1_size` and `array2[]` from cache

Memory & Cache Status

`array1_size = 00000008`

Memory at `array1` base:
8 bytes of data (value doesn't matter)
Memory at `array1` base+1000:
`09 F1 98 CC 90...` (something secret)

```
array2[ 0*4096]
array2[ 1*4096]
array2[ 2*4096]
array2[ 3*4096]
array2[ 4*4096]
array2[ 5*4096]
array2[ 6*4096]
array2[ 7*4096]
array2[ 8*4096]
array2[ 9*4096]
array2[10*4096]
array2[11*4096]
...
```

Contents don't matter
only care about cache **status**

Uncached Cached

Conditional Branch (Var 1) Attack

```
if (x < array1_size)
   y = array2[array1[x]*4096];
```

Attacker calls victim with `x`=1000

Speculative exec while waiting for `array1_size`:

- Predict that if() is true
- Read address (`array1` base + `x`) (using out-of-bounds `x`=1000)
- Read returns secret byte = **09** (in cache ⇒ fast)

Memory & Cache Status

`array1_size = 00000008`

Memory at `array1` base:
8 bytes of data (value doesn't matter)

Memory at `array1` base+1000:
09 `F1 98 CC 90...` (something secret)

```
array2[ 0*4096]
array2[ 1*4096]
array2[ 2*4096]
array2[ 3*4096]
array2[ 4*4096]
array2[ 5*4096]
array2[ 6*4096]
array2[ 7*4096]
array2[ 8*4096]
array2[ 9*4096]
array2[10*4096]
array2[11*4096]
...
```

Contents don't matter only care about cache **status**

Uncached Cached

Conditional Branch (Var 1) Attack

```
if (x < array1_size)
   y = array2[array1[x]*4096];
```

Attacker calls victim with `x`=1000

- Request mem at (`array2` base + **09***4096)
- Brings `array2[09*4096]` into the cache
- Realize if() is false, discard speculative work

Finish operation & return to caller

Memory & Cache Status

`array1_size = 00000008`

Memory at `array1` base:
  8 bytes of data (value doesn't matter)
Memory at `array1` base+1000:
  `09 F1 98 CC 90...` (something secret)

```
array2[ 0*4096]
array2[ 1*4096]
array2[ 2*4096]
array2[ 3*4096]
array2[ 4*4096]
array2[ 5*4096]
array2[ 6*4096]
array2[ 7*4096]
array2[ 8*4096]
array2[ 9*4096]
array2[10*4096]
array2[11*4096]
...
```

Contents don't matter only care about cache **status**

Uncached Cached

Conditional Branch (Var 1) Attack

```
if (x < array1_size)
   y = array2[array1[x]*4096];
```

Attacker calls victim with x=1000

- Measures read time for array2[i*4096]
- Read for i=**09** is fast (cached!), reveals secret byte !!
- Repeat with many x (10KB/s)

Memory & Cache Status

array1_size = 00000008

Memory at array1 base:
8 bytes of data (value doesn't matter)

Memory at array1 base+1000:
09 F1 98 CC 90... (something secret)

```
array2[ 0*4096]
array2[ 1*4096]
array2[ 2*4096]
array2[ 3*4096]
array2[ 4*4096]
array2[ 5*4096]
array2[ 6*4096]
array2[ 7*4096]
array2[ 8*4096]
array2[ 9*4096]
array2[10*4096]
array2[11*4096]
...
```

Contents don't matter only care about cache **status**

Uncached | Cached

Violating JavaScript Sandbox

Browsers run JavaScript from untrusted websites

- JIT compiler inserts safety checks, including bounds checks on array accesses

Speculative execution runs through safety checks…

`index` will be in-bounds on training passes, and out-of-bounds on attack passes

JIT thinks this check ensures `index` < `length`, so it omits bounds check in next line. Separate code evicts `length` for attack passes

Do the out-of-bounds read on attack passes!

```
if (index < simpleByteArray.length) {
  index = simpleByteArray[index | 0];
  index = (((index * TABLE1_STRIDE)|0) & (TABLE1_BYTES-1))|0;
  localJunk ^= probeTable[index|0]|0;
}
```

"|0" is a JS optimizer trick (makes result an integer)

4096 bytes = memory page size

Keeps the JIT from adding unwanted bounds checks on the next line

Need to use the result so the operations aren't optimized away

Leak out-of-bounds read result into cache state!

Can evict length/probeTable from JavaScript (easy)

… then use timing to detect newly-cached location in probeTable

Indirect Branches (Var 2)

Indirect branches can go anywhere, e.g., `jmp[rax]`

- If destination is delayed, CPU guesses and proceeds speculatively
- Find an indirect jmp with attacker-controlled register(s), then cause mispredict to a useful 'gadget'

```
y = array2[array1[x]*4096];
```

Attack steps:

- **Mistrain** branch prediction so speculative execution will go to gadget
- **Evict** address [rax] from cache to cause speculative execution
- **Execute** victim so it runs gadget speculatively
- **Detect** change in cache state to determine memory data

Mitigating Spectre

How to prevent Spectre without a huge cost in performance?

Idea 1: fully restore cache state when speculation fails

Insecure! Speculative execution can have observable side effects beyond the cache state

```
if (x < array1_size) {
    y = array1[x];
    do_something_observable(y);
}
```

occupy a bus (detectable from another core, or cause EM radiation

Stopping Speculation

Idea: insert **`LFENCE`** on <u>all</u> vulnerable code paths

Efficient, no impact on benchmark software

```
if (x < array1_size)
    LFENCE          // processor instruction
    y = array2[ array1[x]*4096 ];
```

Insert LFENCEs manually?	Often millions of control flow paths Too confusing - speculation runs 188++ instructions, crosses modules Too risky – miss one and attacker can read entire process memory
Put LFENCES everywhere?	Abysmal performance - LFENCE is <u>very</u> slow Not in binary libraries, compiler-created code patterns
Insert by smart compiler?	Protect only known-bad bad patterns = unsafe • Microsoft Visual C/C++ `/Qspectre` unsafe for 13 of 15 tests ⇒ Protect all potentially-exploitable patterns

Transfer of blame (CPU -> SW): "you should have put an `LFENCE` there"

Remove All Branches?

DOOM with no branches:

one frame every ~7 hours

A branchless DOOM

This directory provides a branchless, mov-only version of the classic DOOM video game.

DOOM, running with only mov instructions.

This is thought to be entirely secure against the Meltdown and Spectre CPU vulnerabilities, which require speculative execution on branch instructions.

Oops! Idea 4: speculative store

More Attacks

Meltdown

Foreshadow

Rogue inflight data load (**RIDL**) and **Fallout**

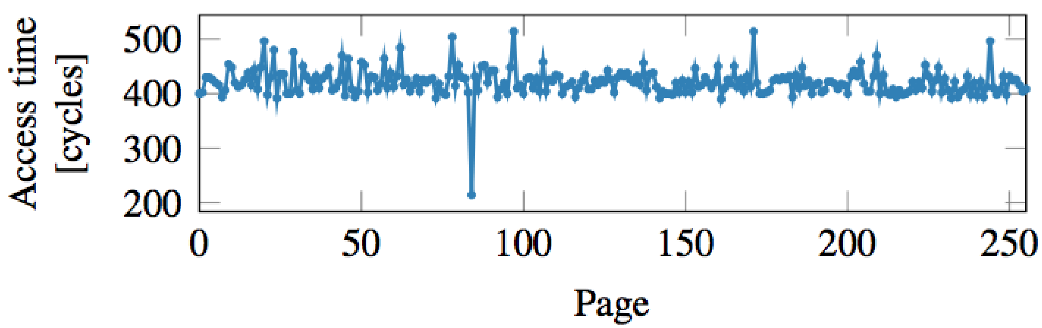
ZombieLoad

Store-to-leak forwarding

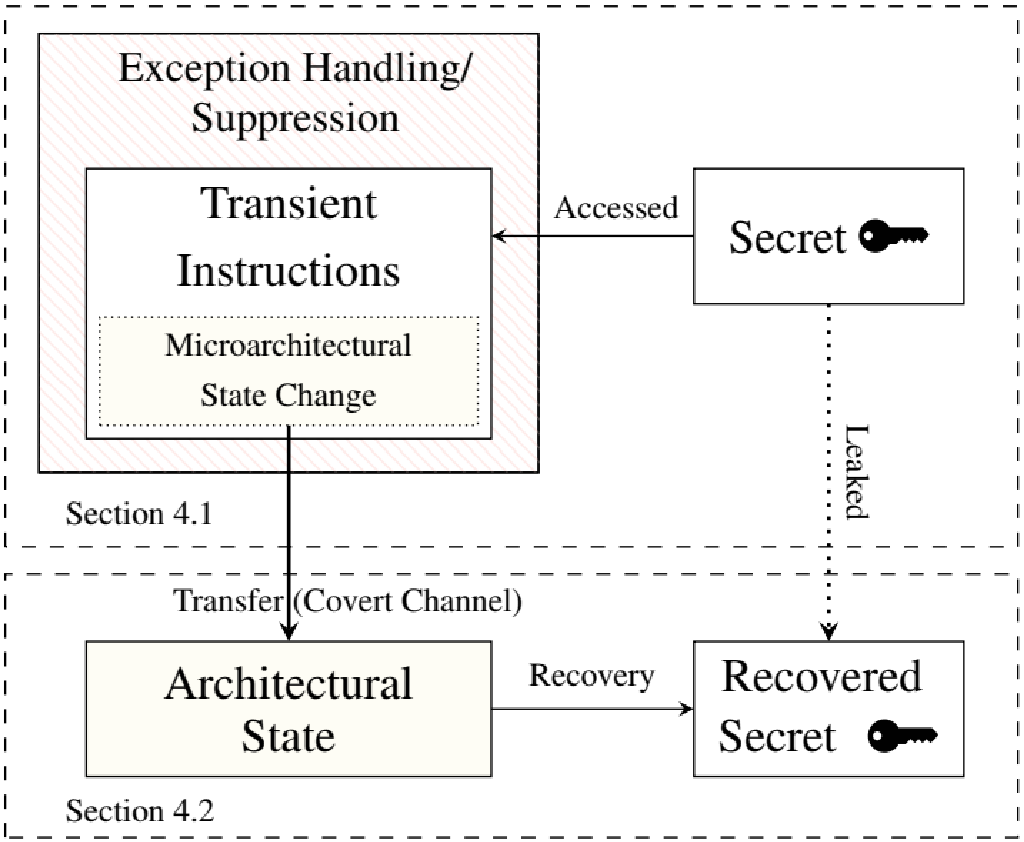
Enable reading unauthorized memory (client, cloud, SGX), mitigating incurs significant performance costs

Meltdown: Intuition

```
raise_exception();
// the line below is never reached
access(probe_array[data * 4096]);
```



Meltdown: Design



Meltdown: Core Spy Code

Retry reading privileged memory

Access privileged memory

Multiply by page size

```
; rcx = kernel address
; rbx = probe array
retry:
mov al, byte [rcx]
shl rax, 0xc
jz retry
mov rbx, qword [rbx + rax]
```

Read from an attacker (unprivileged) array at: (secret value) * 2^{12}

Attacker times accessing [rbx + rax] for different values of rax
When finds one that loads fast, learns sensitive byte