MICROARCHITECTURAL ATTACKS

VITALY SHMATIKOV

Credit for many slides: Stanford CS 155

Isolation in Modern Systems

- Process cannot read memory of another process
- User-level code cannot read memory of the OS kernel
- JavaScript cannot read memory of the Web browser outside its sandbox
- Virtual machine cannot read memory of another virtual machine
- Code outside an SGX enclave cannot read memory protected by SGX

Performance in Modern CPUs

Clock speed maxed out

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

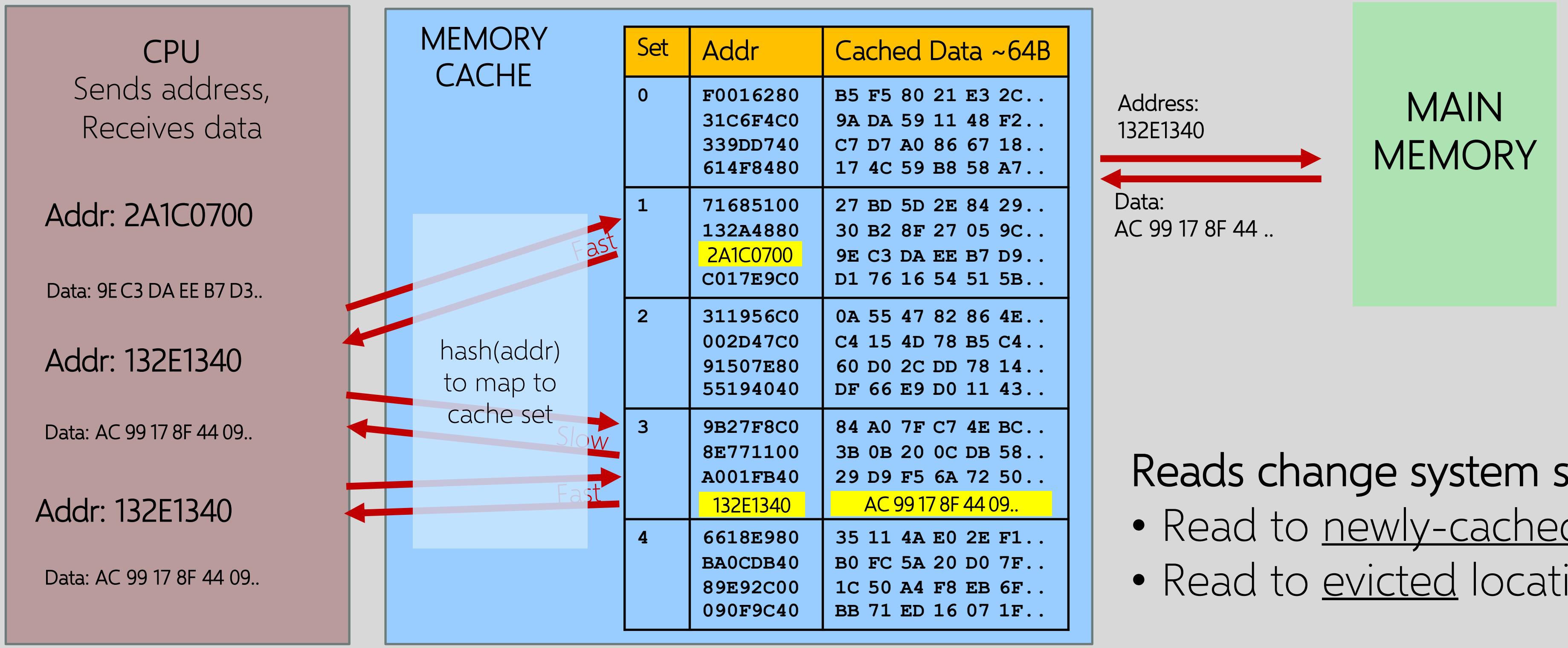
To gain performance, need to do more per cycle…

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

While waiting to determine the value of a condition, guess which branch to take, execute, throw out results if guess is wrong

Memory Caches

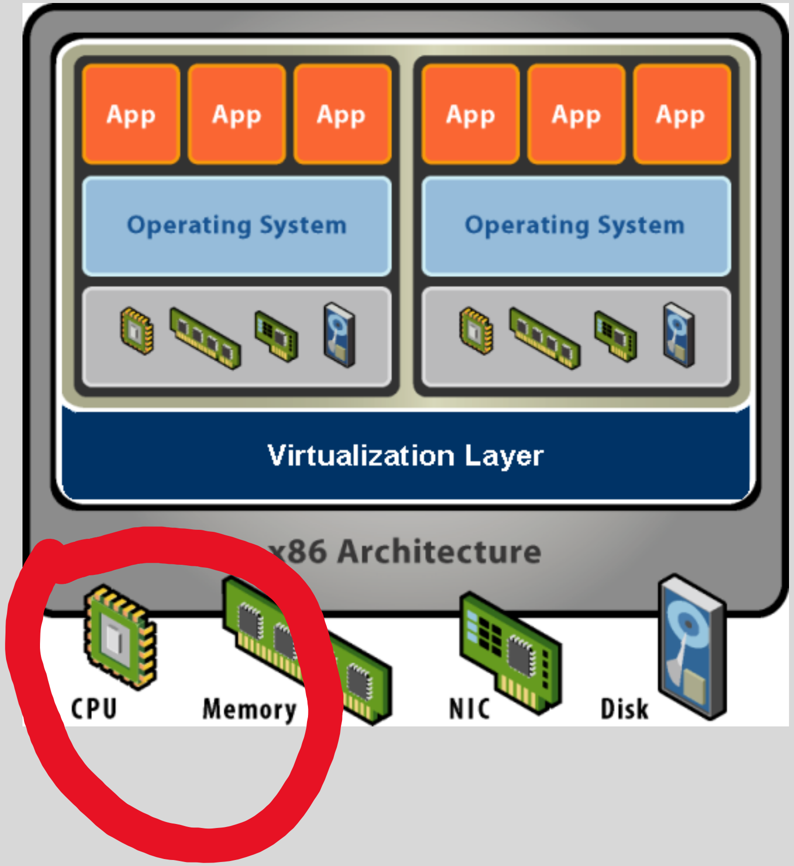
Caches hold local (fast) copy of recently accessed 64-byte chunks of memory



Big, slow
(e.g. 16GB SDRAM)

Reads change system state:
- Read to newly-cached location is fast
- Read to evicted location is slow

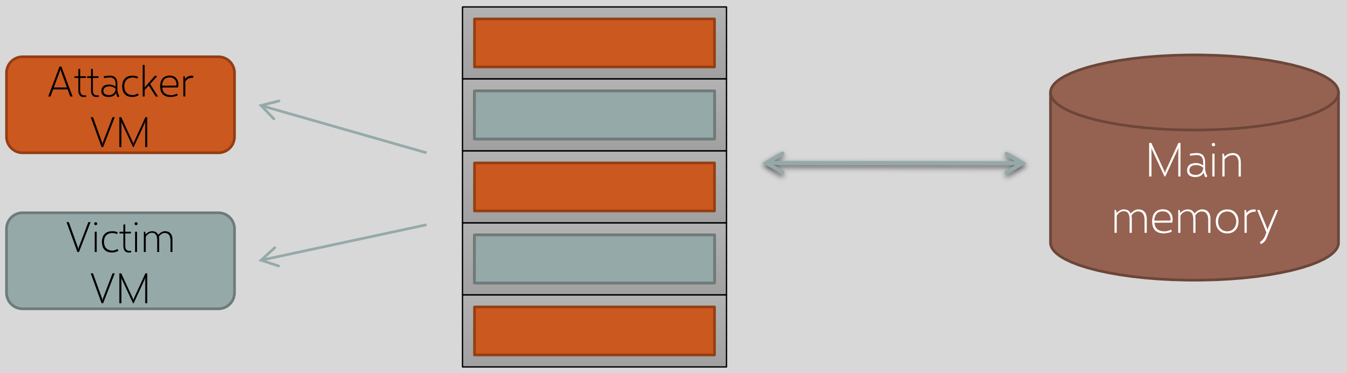
Virtual Machine



- Software abstraction
  - Behaves like hardware
  - Encapsulates all OS and application state
- Virtualization layer
  - Extra level of indirection
  - Decouples hardware, OS
  - Enforces isolation
  - Multiplexes physical hardware across VMs

Cache Contention

Same hardware cache used
by victim and attacker



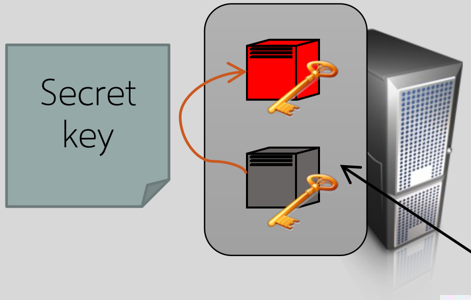
CPU data cache

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 (load measurement)

Locations in cache occupied by victim will take longer to load

Information about victim's use of cache revealed to attacker

Cross-VM Side Channel



Secret 4096-bit encryption key

<u>Modular Exponentiation (x, e, N):</u>

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

$y \leftarrow 1$

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

    $y \leftarrow$ **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 // $y = x^e \bmod N$

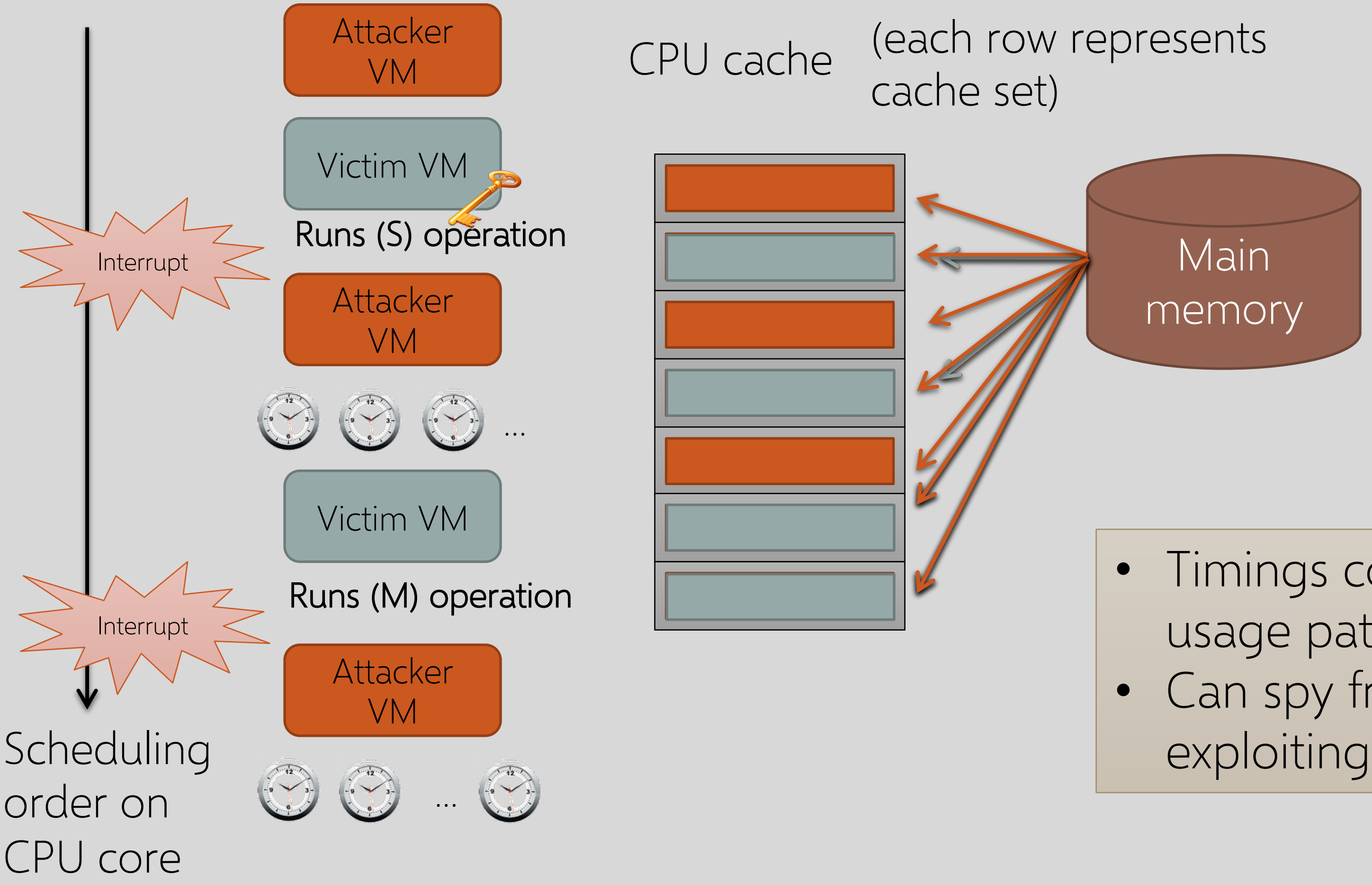
ElGamal encryption algorithm

If $e_i = 1$, execute "SRMR"

If $e_i = 0$, execute "SR"

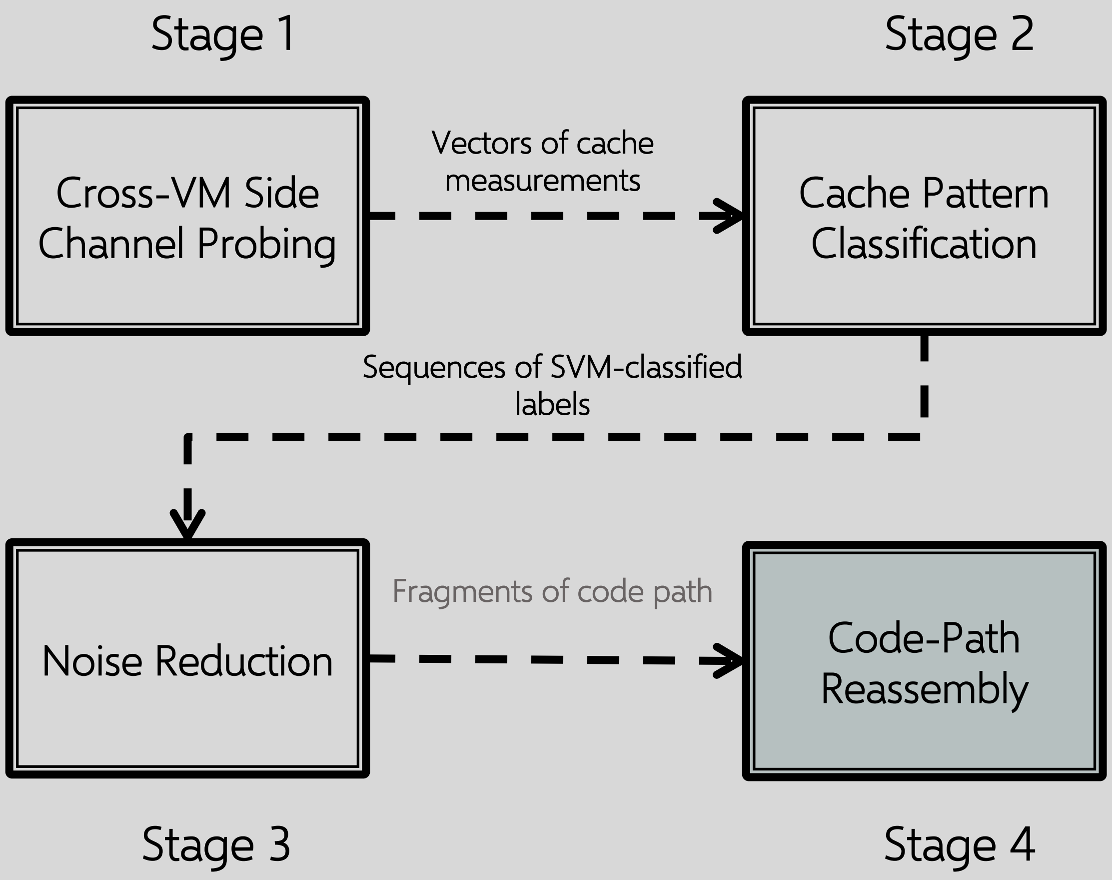
Sequence of function calls reveals secret key

Prime + Probe



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

Attack Stages

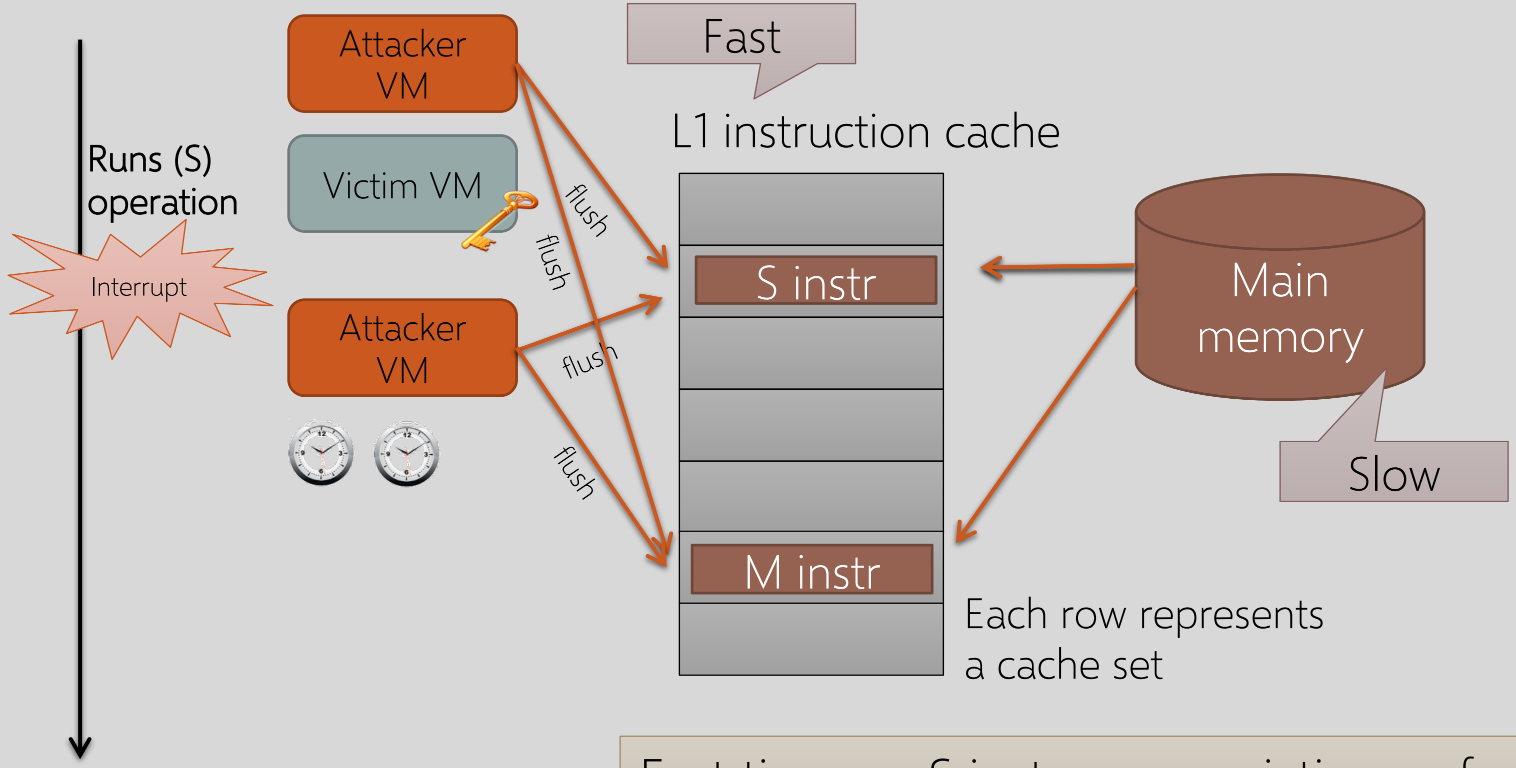


Prime + Probe Feasibility

- 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

State-of-the-art Prime+Probe attacks:
Sinan Inci et al. 2016 "Cache Attacks Enable Bulk Key Recovery on the Cloud"

Flush + Reload



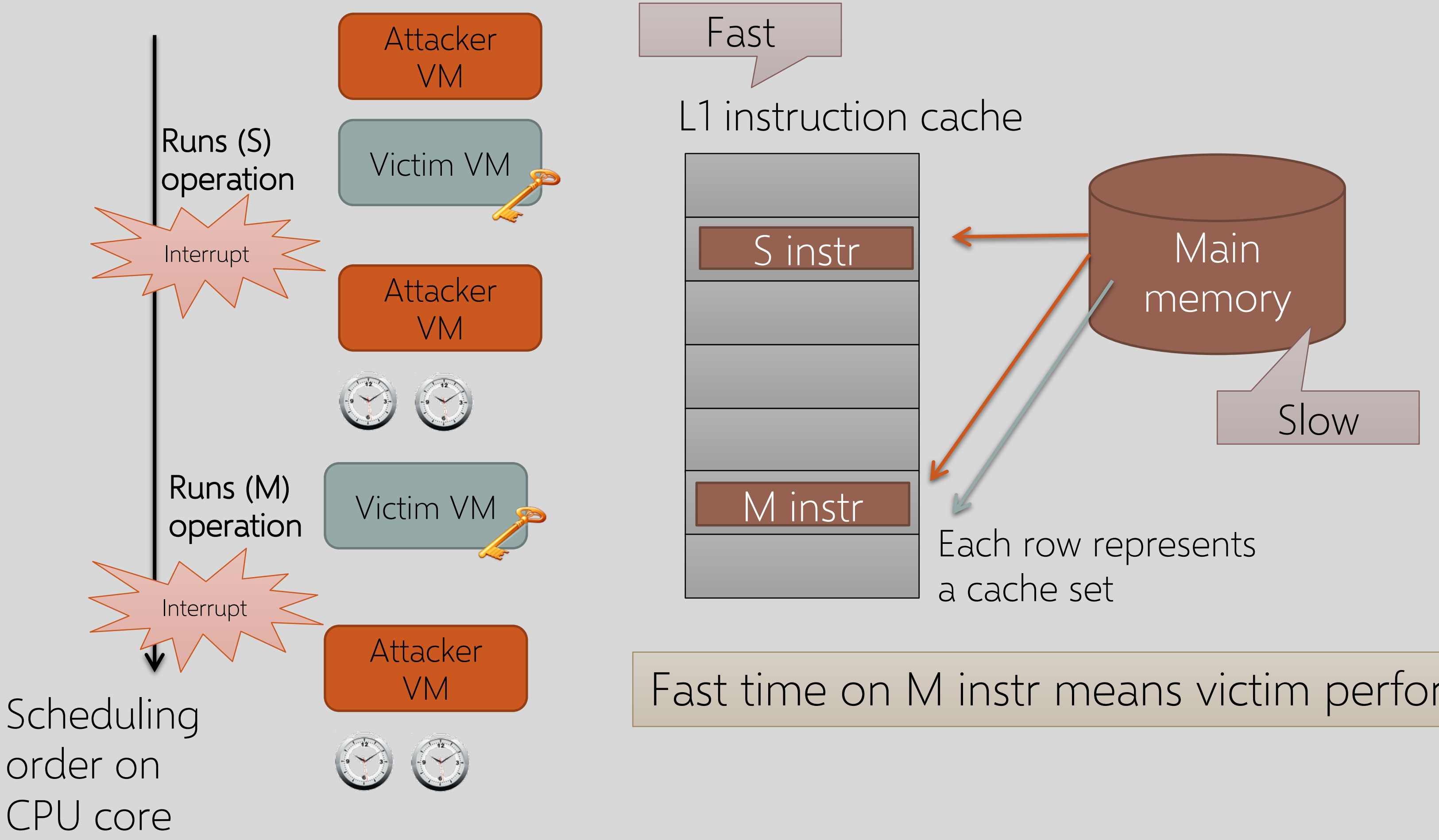
Scheduling order on CPU core

Yuval Yarom

Fast time on S instr means victim performed S op

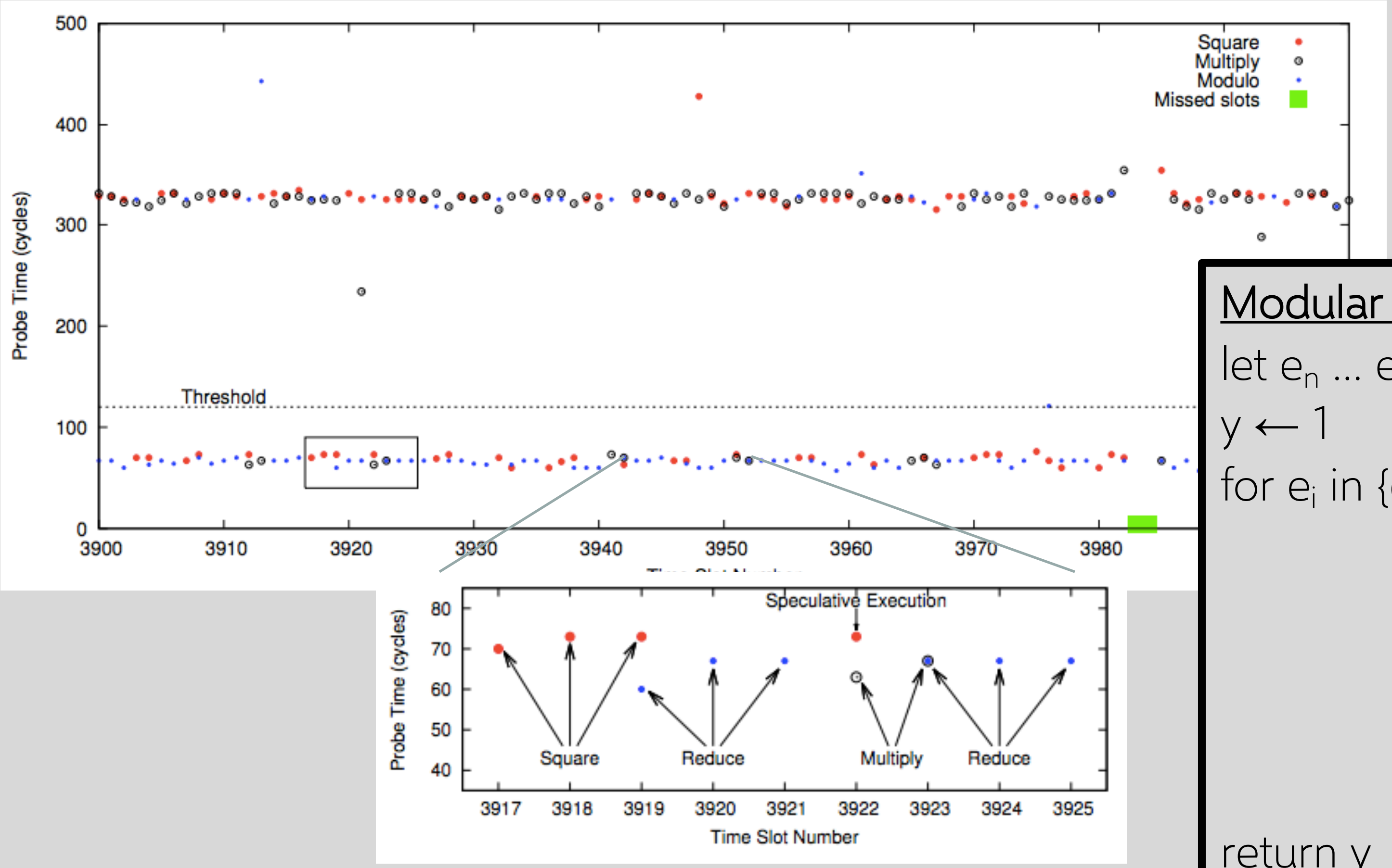
Yarom and Faulkner (2013)

Flush + Reload



Fast time on M instr means victim performed M op

Attacking Square-and-Multiply



<u>Modular Exponentiation</u> <u>(x, e, N):</u>

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

Speculative Execution

Slow

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

Must wait to execute this code until if() is known

Branch predictor guesses if() is true based on past history
CPU speculatively executes compute(b) while value is being loaded

After value arrives from memory...
Guess correct: save speculative work, **performance gain** (!!)
Incorrect: discard speculative work, ho harm (??)

Problem: Side Effects

Architectural Guarantee:

Register values eventually match the result of in-order execution

Speculative execution:

CPU performs incorrect calculations, then deletes mistakes

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

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

The problem: these instructions can have observable side-effects

Spectre and Meltdown

- Speculative execution bugs in Intel x86, ARM, IBM processors + cache-based side channels (Flush+Reload)
- Consequences: break memory protection and isolation in kernels, JavaScript sandboxes, hypervisors, other VMs, trusted execution enclaves (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 Attack (Variant 1)

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

Suppose `unsigned int x` comes from an untrusted caller

Execution without speculation is safe:

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

What about with speculative execution?

Conditional Branch Attack (Variant 1)

```
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 Attack (Variant 1)

```
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 Attack (Variant 1)

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

Attacker calls victim with `x`=1000

...

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

Finish operation and 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 Attack (Variant 1)

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

Attacker calls victim with `x`=1000

...

- Measures read time for `array2[i*4096]` for all `i`
- Read for `i`=**09** is fast (because cached!), reveals the value of the 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 bounds checks on array accesses
- Speculative execution runs through safety checks…

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

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

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

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

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

Evict length/probeTable from JavaScript (easy), then use timing to detect newly-cached location in probeTable

Indirect Branches (Variant 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 infer memory contents

First Fully Weaponized Spectre Exploit Discovered Online

A fully weaponized exploit for the Spectre CPU vulnerability was uploaded on the malware-scanning website VirusTotal last month, marking the first time a working exploit capable of doing actual damage has entered the public domain.

March 1, 2021

Can dump /etc/shadow password file

https://therecord.media/first-fully-weaponized-spectre-exploit-discovered-online/

Mitigating Spectre: Restore Cache State

Idea: 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

Mitigating Spectre: 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.

Mitigating Spectre: Stop Speculation

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

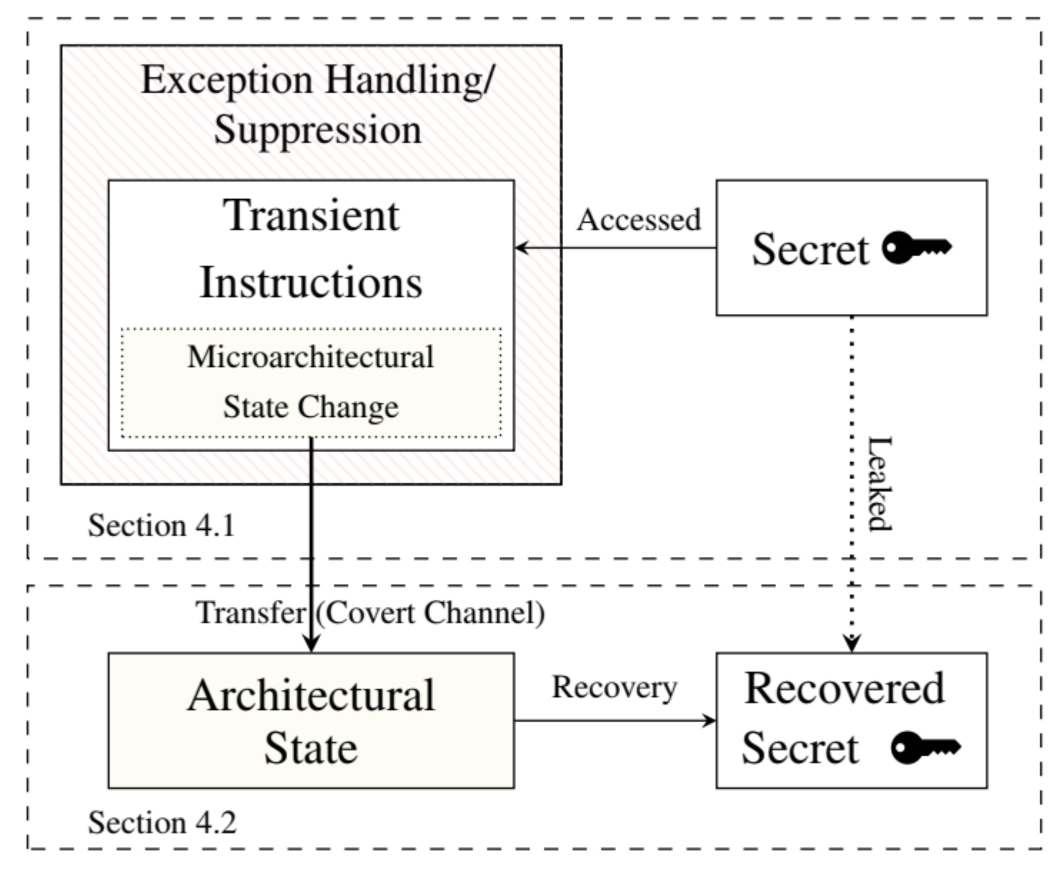
```
if (x < array1_size)
   LFENCE          // processor instruction
   y = array2[ array1[x]*4096 ];
```

Efficient, no impact on benchmark software

Transfers blame from CPU to software: "should of put an LFENCE here!"

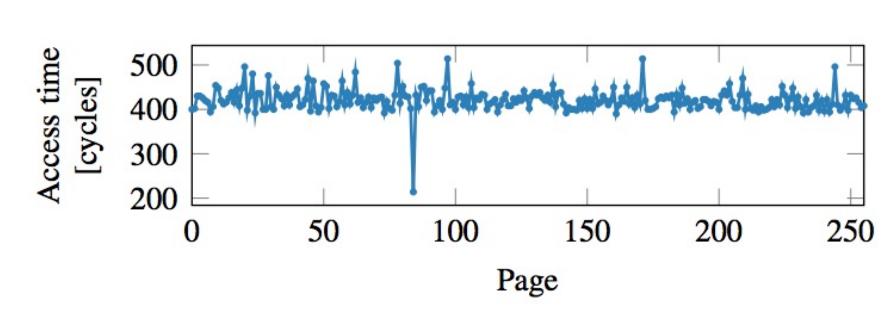
Insert LFENCEs manually?	Often millions of control flow paths Too confusing - speculation runs 188++ instructions, crosses modules Too risky – miss one and attacker can read entire process memory
Put LFENCES everywhere?	Abysmal performance - LFENCE is <u>very</u> slow Not in binary libraries, compiler-created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

Meltdown



Intuition

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



Meltdown: Core Spy Code

Retry reading privileged memory

Access privileged memory

Multiply by page size

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

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

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

Meltdown Mitigation

KAISER/KPTI (kernel page table isolation):
remove kernel memory mapping in user-space processes

- Some performance impact
- Some kernel memory still needs to be mapped

More Attacks

- Foreshadow
- Rogue inflight data load (RIDL) and Fallout
- ZombieLoad
- Store-to-leak forwarding

All enable reading unauthorized memory (client, cloud, SGX)

Mitigating incurs significant performance costs