last time (1)

side channel idea:

- unintended information leakage
- example: time taken to check password → matching character count

in the cache: PRIME+PROBE strategy

- timing difference indicates what's in cache
- evictions reveal index bits of cache accesses

speculative execution and cache accesses

- OOO processors still run cache accesses on branch misprediction
- problem: branches do things like bounds check
- way of reading out-of-bounds data

last time (2)

Meltdown

some Intel CPUs: speculative page table permissions check

```
if (false) { access array[*kernel_memory * factor] }
```

idea: array access adds to cache (even though undone)

detect what was evicted, learn `*kernel_memory` value

Spectre

```
if (x < size) { access array2[array1[x] * factor] }
```

if statement mispredicted, so array2 access modifies cache

...can detect which cache index accessed

pattern appears naturally in system calls, etc.

learn `array1[x]` value, even though out of bounds

review: PRIME+PROBE

```
char *array;
// PRIME
posix_memalign(&array, CACHE_SIZE, CACHE_SIZE);
AccessAllOf(array);

// (some code we don't control)
other_array[mystery * BLOCK_SIZE] += 1;

// PROBE
for (int i = 0; i < CACHE_SIZE; i += BLOCK_SIZE) {
    if (CheckIfSlowToAccess(&array[i])) {
    ...
    }
}
```

exercise

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

with 64KB direct-mapped cache with 64B blocks

suppose we find out that `array[0x200]` is slow to access

and `other_array` starts at some multiple of cache size

What was mystery?

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

- NSETS = CACHE_SIZE/BLOCK_SIZE = 64KB/64B = 1K = 2^{10}
- And this affected `array[0x200]`
  - Which had cache index 0x200/BLOCK_SIZE = 512/64 = 8
  - Or 0b 0010 0000 0000
- `other_array[mystery] = other_array + mystery` (because these are char array)
- If we know the base address of `other_array` is 0x20000, we need to index(0x20000 + mystery) = 8
- 0b 0010 0000 0000 0000 0000 //other_array
- +0b ???? ???? ???? ???? ???? //mystery
- =0b ???? 0000 0010 00?? ????
- So we get a couple bits in the low-order byte of mystery and the next byte

extracting low-order bits

```
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 64KB direct-mapped cache with 64B blocks

suppose we find out that `array[0x700]` is slow to access

and `other_array` starts at some multiple of cache size

What was mystery?

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

- NSETS = CACHE_SIZE/BLOCK_SIZE = 64KB/64B = 1K = 2^{10}
- And this affected `array[0x700] //cache-aligned`
  - Which had cache index 0x700/BLOCK_SIZE = 1792/64 = 28
  - Or 0b 0111 0000 0000
- `other_array[mystery] = other_array + mystery` (because these are char array)
- If we know the base address of `other_array` is 0x20000, we need index(0x20000 + mystery) = 28
- 0b 0010 0000 0000 0000 0000 //other_array
- +0b ???? ???? ???? ???? ???? //mystery
- =0b ???? 0000 0111 00?? ????
- Now we find the low order byte of mystery, which is 0b 0001 1100 = 28
- In either case, we extract log(NSETS) bits, at the positions that align with the index bits

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

- NSETS = CACHE_SIZE/BLOCK_SIZE = 64KB/64B = 1K = 2^{10}
- And this affected `array[0x700]`
  - Which had cache index 0x700/BLOCK_SIZE = 1792/64 = 28
  - Or 0b 0111 0000 0000
- `other_array[mystery] = other_array + mystery` (because these are char array)
- If we know the base address of `other_array` is 0x20440, we need to index(0x20440 + mystery) = 28
- 0b 0010 0000 0100 0100 0000 //other_array
- +0b ???? 0000 0010 11?? ???? //mystery
- =0b ???? 0000 0111 00?? ????
- Now we find the actual value of mystery, which is 0b 0000 1011 = 11

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

- NSETS = CACHE_SIZE/BLOCK_SIZE = 64KB/64B = 1K
- Each value of `mystery` touches a different cache line
  - So we touched cache index `mystery % NSETS`
  - But base address might be offset
- And this affected `array[0x700]`
  - Which had cache index 0x700/BLOCK_SIZE = 1792/64 = 28
- And `&other_array` starts at 0x20440, which has cache index (0x20440/BLOCK_SIZE)%NSETS = 17
- So IDX(mystery) + IDX(&other_array) = 28
- So IDX(mystery) = 28 -17 = 11
- So mystery = 11 or (11+1024) or …
  - If we know mystery is a char, then we know it’s between 0-255, so in this case mystery = 11
- It’s the same math!!!

```
char array[CACHE_SIZE] // not aligned
AccessAllOf(array); // PRIME
other_array[mystery * BLOCK_SIZE] += 1;
for (int i = 0; i < CACHE_SIZE; i += BLOCK_SIZE) {
    if (CheckIfSlowToAccess(&array[i])) // PROBE
    {...}
}
```

- NSETS = CACHE_SIZE/BLOCK_SIZE = 64KB/64B = 1K
- Each value of `mystery` touches a different cache line
  - So we touched cache index `mystery % NSETS`
  - But base address might be offset
- And this affected `array[0x8280]`
  - Whose base address might also be offset, say 0x48480
  - What cache index is `array[0x8280]`?
  - IDX(&array + 0x8280) = ((0x48480 + 0x8280)/BLOCK_SIZE)%NSETS = 28
- And `&other_array` starts at 0x20440, which has cache index (0x20440/BLOCK_SIZE)%NSETS = 17
- So IDX(mystery) + IDX(&other_array) = 28
- So IDX(mystery) = 28 -17 = 11
- So mystery = 11 or (11+1024) or …
  - If we know mystery is a char, then we know it's between 0-255, so in this case mystery = 11

What about associative caches?

```
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 64KB 2-way cache with 64B blocks

suppose we find out that `array[0x800]` is slow to access

and `other_array` starts at some multiple of cache size

What was mystery?

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

- NSETS = CACHE_SIZE/BLOCK_SIZE/ASSOC = 64KB/64B/2 = 512 *(not 1024)*
- Each value of `mystery` touches a different cache line
  - So we touched cache index `mystery % NSETS`
- And this affected `array[0x800]`
  - Which had cache index 0x800/BLOCK_SIZE
- So mystery % N_SETS = 0x800/BLOCK_SIZE
- So mystery = 0x800/BLOCK_SIZE + k * N_SETS
- So mystery = 32 or (32+512) or …
- Can also do the bitwise approach (1 fewer index bit) and should get the same answer

exercise

```
char *array;
// PRIME
posix_memalign(&array, CACHE_SIZE, CACHE_SIZE);
AccessAllOf(array);
// (some code we don't control)
other_array[mystery * BLOCK_SIZE] += 1;
// PROBE
for (int i = 0; i < CACHE_SIZE; i += BLOCK_SIZE) {
    if (CheckIfSlowToAccess(&array[i])) {
    ...
    }
}
```

64KB (2^{16}B) direct-mapped cache with 64B blocks

array[0x800] slow to access;

other_array at `0x4000000`

value of `mystery`?

exercise solution (1)

NUM_SETS $=$ 64KB/64B $=$ 1K (1024) sets

array[0x800] has cache index `0x800`/BLOCK_SIZE mod NUM_SETS
$=$ cache index 32

know `other_array[mystery * BLOCK_SIZE]` had same index

`other_array[0]` at cache index 0
(0x4000000 / BLOCK_SIZE) mod NUM_SETS $=$ 0

exercise solution (2)

recall have found:

- `other_array[0]` at index 0;
- `other_array[mystery*BLOCK_SIZE]` has index 32 (same as `array[0x800]`)

`other_array[X]` at cache index (0 + X/BLOCK_SIZE mod NUM_SETS)

- advanced by X/BLOCK_SIZE blocks
- wrapping around after NUM_SETS blocks

X = mystery * BLOCK_SIZE

32 = 0 + mystery mod NUM_SETS

mystery = 32 or 32 $\pm$ 1024 or 32 $\pm$ 1024 $\times$ 2 or etc.

variation: different starting location

other_array starts at 0x4001440

then other_array[0] at cache index

$0x4001440 / \text{BLOCK\_SIZE} \bmod \text{NUM\_SETS} = 51$

$(51 + \text{mystery} * \text{BLOCK\_SIZE} / \text{BLOCK\_SIZE}) \bmod \text{NUM\_SETS} = 32$

mystery = -19 or 1005 or 2029 or …

variation: associative cache

```
char *array;
// PRIME
posix_memalign(&array, CACHE_SIZE, CACHE_SIZE);
AccessAllOf(array);
// (some code we don't control)
other_array[mystery * BLOCK_SIZE] += 1;
// PROBE
for (int i = 0; i < CACHE_SIZE; i += BLOCK_SIZE) {
    if (CheckIfSlowToAccess(&array[i])) { ...  }
}
```

suppose 2-way 64KB cache instead of direct-mapped

NUM_SETS $=$ 64KB$/$2$/$64B $=$ 512 sets

array[0x800] still has cache index 32 (still)

but now mystery can be 32 or $32 + 512$ or $32 + 512 \cdot 2$ or ...

variation: associative cache (2)

```
char *array;
// PRIME
posix_memalign(&array, CACHE_SIZE, CACHE_SIZE);
AccessAllOf(array);
// (some code we don't control)
other_array[mystery * BLOCK_SIZE] += 1;
// PROBE
for (int i = 0; i < CACHE_SIZE; i += BLOCK_SIZE) {
    if (CheckIfSlowToAccess(&array[i])) { ...  }
}
```

suppose 2-way 64KB cache w/ 64B and `array[0x8800]` is slow

0x8800/BLOCK_SIZE $= 544 = 512 + 32$

since 512 sets total, still set index 32

mystery still 32 or $32 + 512$ or $32 + 512 \cdot 2$ or ...

exercise

if 4-way 64KB cache w/64B blocks and something from cache set 32 evicted,
then where could slow access be?

recall: 2-way cache: i=0x800, i=0x8800

A. i=0x400, i=0x800, i=0x8400, i=0x8800

B. i=0x800, i=0x8800, i=0x10800, i=0x18800

C. i=0x800, i=0x4800, i=0x8800, i=0xc800

D. i=0x800, i=0x4800, i=0x8800, i=0x10800

E. something else

not just BLOCK_SIZE

```
char *array;
// PRIME
posix_memalign(&array, CACHE_SIZE, CACHE_SIZE);
AccessAllOf(array);
// (some code we don't control)
other_array[mystery * N] += 1;  // previously: * BLOCK_SIZE
// PROBE
for (int i = 0; i < CACHE_SIZE; i += BLOCK_SIZE) {
    if (CheckIfSlowToAccess(&array[i])) {
    ...
    }
}
```

64KB (2^{16}B) direct-mapped cache with 64B blocks

array[0x800] slow to access?

other_array at `0x4000000` (index 0, offset 0)

value of `mystery` if N $=$ 1? N $=$ 32 * 64?

solution (N=1)

$$\lfloor \text{mystery} * N/\text{BLOCK\_SIZE} \rfloor \mod 1024 = 32$$
$$\lfloor \text{mystery} * N/\text{BLOCK\_SIZE} \rfloor = 32 + 1024K$$

let offset be some number in [0,BLOCK_SIZE):

$$\text{mystery} * N = \text{BLOCK\_SIZE} \times (32 + 1024K) + \text{offset}$$
$$\text{mystery} = \text{BLOCK\_SIZE} \times (32 + 1024K) + N \times \text{offset}$$
$$\text{mystery} = 64 \times (32 + 1024K) + N \times \text{offset}$$

N=1: mystery $= 2048$, 2049, 2050, ..., $2048 + 63$, $64 \cdot 1024 + 2048$, $64 \cdot 1024 + 2048 + 1$, ...

exercise (N=32*64)

what if N = 32*64

recall: other_array[0] is set 0, offset 0

other_array[mystery * N] is set 32

possible values of mystery?

$$\begin{aligned}
\text{mystery} \cdot 32 \cdot 64 &= 64(32 + 1024K) + \text{offset} \\
&= 64 \cdot 32 + 65536K + \text{offset} \\
\text{mystery} &= 1 + \frac{65536}{64 \cdot 32}K + \frac{\text{offset}}{64 \cdot 32} = 1 + 32K
\end{aligned}$$

alternate view

learn index bits of mystery * N

this example: bits 6–15

N = 1, bits 6–15 of mystery

N = 64, bits 0–9 of mystery

N = 32*64 (2^{11}), bits 0–4 of mystery

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 = ker
    // %rbx = ar                              viction
    xor %rax, %ra
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
```

space out accesses by 4096
ensure separate cache sets and
avoid triggering prefetcher

Meltdown

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

```
    // %rcx
    // %rbx                                          on
    xor %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
```

repeat access if zero

apparently value of zero speculatively read

when real value not yet available

Meltdown

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

```
    // %rcx =
    // %rbx =                                              ion
    xor %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
```

access cache to allow measurement later in paper with FLUSH+RELOAD instead of PRIME+PROBE technique

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)

```
// 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
```

EVICT+RELOAD

PRIME+PROBE: fill cache, detect eviction

alternate idea EVICT+RELOAD:

```
unsigned char *probe_array;
posix_memalign(&probe_array, CACHE_SIZE, CACHE_SIZE);
access OTHER things to evict all of probe_array
if (something false) {
    read probe_array[mystery * BLOCK_SIZE];
}
check which value from probe_array is faster
```

requires code to access something you can access

but often easier to setup/more reliable than PRIME+PROBE

EVICT+RELOAD

PRIME+PROBE: fill cache, detect eviction

alternate idea EVICT+RELOAD:

```
unsigned char *probe_array;
posix_memalign(&probe_array, CACHE_SIZE, CACHE_SIZE);
access OTHER things to evict all of probe_array
if (something false) {
    read probe_array[mystery * BLOCK_SIZE];
}
check which value from probe_array is faster
```

requires code to access something you can access

but often easier to setup/more reliable than PRIME+PROBE

EVICT+RELOAD

PRIME+PROBE: fill cache, detect eviction

alternate idea EVICT+RELOAD:

```
unsigned char *probe_array;
posix_memalign(&probe_array, CACHE_SIZE, CACHE_SIZE);
access OTHER things to evict all of probe_array
if (something false) {
    read probe_array[mystery * BLOCK_SIZE];
}
check which value from probe_array is faster
```

requires code to access something you can access

but often easier to setup/more reliable than PRIME+PROBE

into exploit: Meltdown

```
uint8_t* probe_array = new uint8_t[256 * 4096];
// ... Make sure probe_array is not cached
uint8_t kernel_memory_val = *(uint8_t*)(kernel_address);
uint64_t final_kernel_memory = kernel_memory_val * 4096;
uint8_t dummy = probe_array[final_kernel_memory];
// ... catch page fault
// ... in signal handler, determine which of 256 slots in prol
```

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 `0x1000000` 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)

```
unsigned 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)

```
unsigned 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 130

another exercise

```
char array1[…];
…
int secret;
…
y = array2[array1[x] * 4096];
```

- Suppose our cache has 64B blocks and 1K sets, and array2[0] is in set 0
- Suppose our prime+probe lets us see that something in cache set 256 or our probe array (array2) is evicted
- What do we know about array1[x]?

```
char array1[…];
…
int secret;
…
y = array2[array1[x] * 4096];
```

- Suppose our cache has 64B blocks and 1K sets, and array2[0] is in set 0
  - So array2[64] is in set 1, array2[128] is in set 2, etc.
- Suppose our prime+probe lets us see that something in cache set 256 of our probe array (array2) is evicted,
  - So CACHE_SET(array1[x]*4096) = 256
- What do we know about array1[x]?

- array1[x] * 4K = 64 * target_set + some multiple of number of sets
- array1[x] * 4K = 64 * 256 + …
- So array1[x] = (64*256)/4K = 16K/4K = 4 + …

```
char array1[…];
…
int secret;
…
y = array2[array1[x] * 4096];
```

- Suppose our cache has 64B blocks and 32K sets, and array2[0] is in set 0
  - So array2[64] is in set 1, array2[128] is in set 2, etc.
- Suppose our prime+probe lets us see that something in cache set 256 of our probe array is evicted, so CACHE_SET(array1[x]*4096) = 256
- What do we know about array1[x]?

- array1[x] * 4K = 64 * target_set + some multiple of number of sets
- array1[x] * 4K = 64 * 256 + n*32K*64
- So array1[x] = (64*256 + n*32K*64)/4K = 16K/4K + (n*32K*64)/4K
  - So array1[x] = 4 or 4+512 or…
  - But it's a char, so it can only be 4

```
char array1[…];
…
int secret;
…
y = array2[array1[x] * 4096];
```

- Suppose our cache has 64B blocks and 2K sets, and array2[0] is in set 0
  - So array2[64] is in set 1, array2[128] is in set 2, etc.
- Suppose our prime+probe lets us see that something in cache set 256 of our probe array is evicted, so CACHE_SET(array1[x]*4096) = 256
- What do we know about array1[x]?

- array1[x] * 4K = 64 * target_set + some multiple of number of sets
- array1[x] * 4K = 64 * 256 + n*2K*64
- So array1[x] = (64*256 + n*2K*64)/4K = 16K/4K + (n*2K*64)/4K
  - So array1[x] = 4 or 4+32 or 4+64 or…
  - But it's a char, so it can only be 4, 36, 68, 100, 132, 164, or 196
  - … This works better in last-level caches with larger # of sets

```
char array1[…];
…
int secret;
…
y = array2[array1[x]];  // no *4096 this time
```

- Suppose our cache has 64B blocks and 32K sets, and array2[0] is in set 0
  - So array2[64] is in set 1, array2[128] is in set 2, etc.
- Suppose our prime+probe lets us see that something in cache set 3 of our probe array is evicted, so CACHE_SET(array1[x]~~*4096~~) = 3
- What do we know about array1[x]?

- array1[x] ~~* 4K~~ = 64 * target_set + some multiple of number of sets
- array1[x] ~~* 4K~~ = 64 * 3 + n*32K*64
- So array1[x] = 196 + n*32K*64
  - So array1[x] = 196 or some large number
  -

exploit with contrived(?) code

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

```
/* exploiting code */
    /* step 1: mistrain branch predictor */
for (a lot) {
    systemCallHandler(0 /* less than array1_size */);
}
    /* step 2: evict from cache using misprediction */
Prime();
systemCallHandler(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 0x0FFFE

bounds check in kernel

```
if (x < array1_size) {
    y = array2[array1[x]];
}
```

our template

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

actual code

bounds check in kernel

```
if (x < array1_size) {
    y = array2[array1[x]];
}
```

our template

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

actual code

bounds check in kernel

```
if (x < array1_size) {
    y = array2[array1[x]];
}
```

our template

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

actual code

bounds check in kernel

```
if (x < array1_size) {
    y = array2[array1[x]];
}
```

our template

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

actual code

privilege levels?

vulnerable code runs with higher privileges

so far: higher privileges = kernel mode

but other common cases of higher privileges

example: scripts in web browsers

JavaScript

JavaScript: scripts in webpages

not supposed to be able to read arbitrary memory, but…

can access arrays to examine caches

and could take advantage of some browser function being vulnerable

JavaScript

JavaScript: scripts in webpages

not supposed to be able to read arbitrary memory, but…

can access arrays to examine caches

and could take advantage of some browser function being vulnerable

or — doesn't even need browser to supply vulnerable code itself!

just-in-time compilation?

for performance, compiled to machine code, 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

examines what's evicted by probeTable access

supplying own attack code?

JavaScript: could supply own attack code

turns out also possible with kernel mode scenario

trick: don't need to *actually run* code

...just need branch predictor to fetch it!

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 jmps 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
  - …

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 changes to isolate better

without hardware help:

- transform `jmp *(%rax)`, etc. into code that
- will only 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
```

predicting ret: ministack of return addresses

predicting ret — ministack in processor registers

- push on ministack on call; pop on ret

ministack overflows? discard oldest, mispredict it later

baz saved registers
baz return address
bar saved registers
bar return address
foo local variables
foo saved registers
foo return address
foo saved registers

stack in memory

baz return address
bar return address
foo return address

(partial?) stack in CPU registers

4-entry return address stack

4-entry return address stack in CPU

current index

1

idx	saved return addresses
0	0x12345
1	0x44432
2	0x44F92
3	0x22331

next prediction for ret

next saved return address from call

on `call`: increment index, save return address in that slot

on `ret`: read prediction from index, decrement index

backup slides

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?

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`