changelog

3 Dec 2025: add some extra exercises re: variations in cache layout and how it affects what information learned from eviction information

3 Dec 2025: add slides on EVICT+RELOAD

5 Dec 2025: “supplying own attack code?”: be more explicit that code is running partially

check_passphrase

```
int check_passphrase(const char *versus) {
    int i = 0;
    while (passphrase[i] == versus[i] &&
           passphrase[i]) {
        i += 1;
    }
    return (passphrase[i] == versus[i]);
}
```

number of iterations = number matching characters

leaks information about passphrase, oops!

exploiting check_passphrase (1)

guess	measured time
aaaa	100 ± 5
baaa	103 ± 4
caaa	102 ± 6
daaa	111 ± 5
eaaa	99 ± 6
faaa	101 ± 7
gaaa	104 ± 4
…	…

exploiting check_passphrase (2)

guess	measured time
daaa	102 ± 5
dbaa	99 ± 4
dcaa	104 ± 4
ddaa	100 ± 6
deaa	102 ± 4
dfaa	109 ± 7
dgaa	103 ± 4
...	...

timing and cryptography

lots of asymmetric cryptography uses big-integer math

example: multiplying 500+ bit numbers together

how do you implement that?

big integer multiplcation

say we have two 64-bit integers x, y

and want to 128-bit product, but our multiply instruction only does 64-bit products

one way to multiply:

divide x, y into 32-bit parts: $x = x_1 \cdot 2^{32} + x_0$ and $y = y_1 \cdot 2^{32} + y_0$

then $xy = x_1 y_1 2^{64} + x_1 y_0 \cdot 2^{32} + x_0 y_1 \cdot 2^{32} + x_0 y_0$

big integer multiplcation

say we have two 64-bit integers x, y

and want to 128-bit product, but our multiply instruction only does 64-bit products

one way to multiply:

divide x, y into 32-bit parts: $x = x_1 \cdot 2^{32} + x_0$ and $y = y_1 \cdot 2^{32} + y_0$

then $xy = x_1 y_1 2^{64} + x_1 y_0 \cdot 2^{32} + x_0 y_1 \cdot 2^{32} + x_0 y_0$

can extend this idea to arbitrarily large numbers

number of smaller multiplies depends on size of numbers!

big integers and cryptography

naive multiplication idea:

- number of steps depends on size of numbers

problem: sometimes the value of the number is a secret

- e.g. part of the private key

oops! revealed through timing

big integer timing attacks in practice (1)

early versions of OpenSSL (TLS implementation)had timing attack

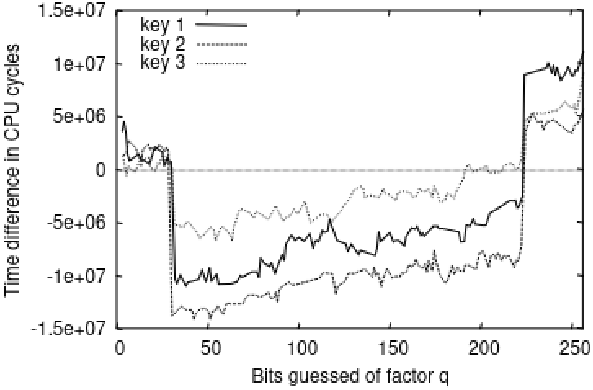
- Brumley and Boneh, "Remote Timing Attacks are Practical" (Usenix Security '03)

attacker could figure out bits of private key from timing

why? variable-time mulitplication and modulus operations

- got faster/slower depending on how input was related to private key

big integer timing attacks in practice (2)



(a) The zero-one gap $T_g - T_{g_{hi}}$ indicates that we can distinguish between bits that are 0 and 1 of the RSA factor q for 3 different randomly-generated keys. For clarity, bits of q that are 1 are omitted, as the x-axis can be used for reference for this case.

Figure 3a from Brumley and Boneh, "Remote Timing Attacks are Practical"

browsers and website leakage

web browsers run code from untrusted webpages

one goal: can't tell what other webpages you visit

some webpage leakage (1)

...as you can see here, here, and here ...

convenient feature 1: browser marks visited links

```
<script>
var the_color = window.getComputedStyle(
    document.querySelector('a[href=~"foo.com"]')
).color
if (the_color == ...) { ... }
</script>
```

convenient feature 2: scripts can query current color of something

some webpage leakage (1)

…as you can see here, here, and here …

convenient feature 1: browser marks visited links

```
<script>
var the_color = window.getComputedStyle(
    document.querySelector('a[href=~"foo.com"]')
).color
if (the_color == ...) { ... }
</script>
```

~~convenient feature 2: scripts can query current color of something~~

fix 1: getComputedStyle lies about the color

fix 2: limited styling options for visited links

some webpage leakage (2)

one idea: script in webpage times loop that writes big array

variation in timing depends on *other things running on machine*

some webpage leakage (2)

one idea: script in webpage times loop that writes big array

variation in timing depends on *other things running on machine*

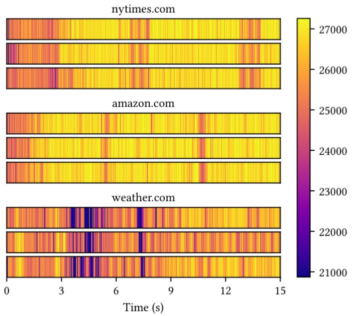


Figure 3: Example loop-counting traces collected over 15 seconds. Darker shades indicate smaller counter values and lower instruction throughput.

turns out, other webpages create distinct "signatures"

Figure from Cook et al, "There's Always a Bigger Fish: Clarifying Analysis of a Machine-Learning-Assisted Side-Channel Attack" (ISCA '22)

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)
- (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 of index bits

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[`0b`*`1 10`*`00 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`: ...0*000 0001 10*00 1000

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

to get index 6, offset 0 array[`0b`*`1 10`*`00 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: ...0*000 1000 00*00 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* ?? ????
`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 tables

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)

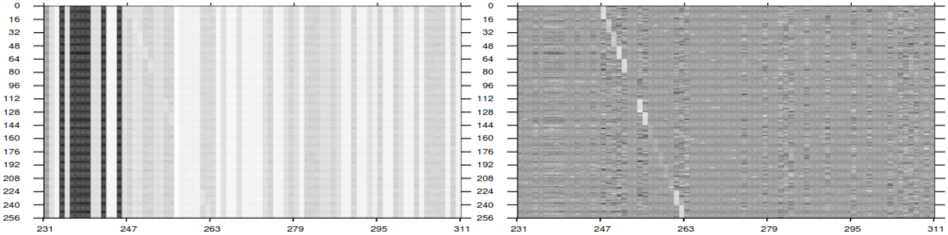


Fig. 5. Prime+Probe attack using 30,000 encryption calls on a 2GHz Athlon 64, attacking Linux 2.6.11 `dm-crypt`. The horizontal axis is the evicted cache set (i.e., $\langle y \rangle$ plus an offset due to the table's location) and the vertical axis is p_0. Left: raw timings (lighter is slower). Right: after subtraction of the average timing of the cache set. The bright diagonal reveals the high nibble of $p_0 = 0\text{x}00$.

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
`movq mystery, %rax`	F	D	R	I	E	E	E	W	C			
`test %rax, %rax`	F	D	R				I	E	W	C		
`jz skip` (mispred.)		F	D	R				I	E	W	C	
`mov pointer, %rax`		F	D	R	I	E	E	E	W			
`mov (%rax), %r8`			F	D	R				I	E	W	
`add $1, %r8`			F	D	R							
`mov %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 (something false) {
    int value = ReadMemoryMarkedNonReadableInPageTable();
    access other_array[value * ...];
}
Probe();
```

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 (something false) {
    int value = ReadMemoryMarkedNonReadableInPageTable();
    access other_array[value * ...];
}
Probe();
```

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 (something false) {
    int value = ReadMemoryMarkedNonReadableInPageTable();
    access other_array[value * ...];
}
Probe();
```

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 (something false) {
    int value = ReadMemoryMarkedNonReadableInPageTable();
    access other_array[value * ...];
}
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 = 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”

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

```
    // %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
```

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
```

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 `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

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

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 ± 1024 or $32 \pm 1024 \times 2$ or etc.

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

not just BLOCK_SIZE

```
char *array, *other_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):

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

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?

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

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

variation: different starting location

other_array starts at 0x4001440

then other_array[0] at cache index

0x4001440 / BLOCK_SIZE mod NUM_SETS = 51

(51 + mystery * BLOCK_SIZE / BLOCK_SIZE) mod 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

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 prob
```

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 for real

…just need branch predictor to fetch it
so it gets partially executed speculatively

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
```

backup slides

Quiz week 14

Question 5 (0 / 4 pt; mean 0.16)

Consider the following C code:

```
struct Files all_files[NUM_FILES];
int GetLastByteOfFile(int index) {
    if (index >= 0 && index < all_files) {
        struct File *file = &all_files[index];
        if (file->type == MEMORY) {
            return file->data[file->size - 1];
        } else if (file->type == DISK) {
            return GetLastByteOfDiskFile(file)
        } else {
            return -1;
        }
    } else {
        return -1;
    }
}
```

If the above function runs in kernel mode, we might be able to use a Spectre-style attack where the cache evictions caused by memory access of `file->data[file.size - 1]` allows us learn about the value of an arbitrary memory location. To perform this attack, the attacker would prefer to choose an out-of-bounds `index` such that ___.

A.	88%	○ the address of `all_files[index].data[file.size - 1]` is the memory address whose value they want to learn about
B.	2%	○ the address of `all_files[index].type` is the memory address they want to learn about
C.	4% (correct)	○ the address of `all_files[index].size` is the memory address they want to learn about
D.		○ the value of `all_files[index].type` is `DISK`

- file = &all_files[index]
- So if we provide an out of bounds index, we can read arbitrary memory
- file->data[], ie all_files[index].data[] is like array2
- file->size, ie all_files[index].size, is like array1

```
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

- kind ~ file->size, ie all_files[index].size, is like array1
  - This is the address we want to learn about by observing its cache behavior
- other_table [] ~ file->data[], ie all_files[index].data[] is like array2
  - Not using the .foo here

Question 5 (0 / 4 pt; mean 0.16)

Consider the following C code:

```
struct Files all_files[NUM_FILES];
int GetLastByteOfFile(int index) {
    if (index >= 0 && index < all_files) {
        struct File *file = &all_files[index];
        if (file->type == MEMORY) {
            return file->data[file->size - 1];
        } else if (file->type == DISK) {
            return GetLastByteOfDiskFile(file)
        } else {
            return -1;
        }
    } else {
        return -1;
    }
}
```

If the above function runs in kernel mode, we might be able to use a Spectre-style attack where the cache evictions caused by memory access of `file->data[file.size - 1]` allows us learn about the value of an arbitrary memory location. To perform this attack, the attacker would prefer to choose an out-of-bounds `index` such that ___.

A.	88%		the address of `all_files[index].data[file.size - 1]` is the memory address whose value they want to learn about
B.	2%		the address of `all_files[index].type` is the memory address they want to learn about
C.	4%	T (correct)	the address of `all_files[index].size` is the memory address they want to learn about
D.			the value of `all_files[index].type` is `DISK`

- Why not A?
- all_files[index].data is like array2
  - Based on which cache set is affected by different index values, we learn what those index values are
  - So we need to set up the index (~array1) to refer to the memory location we're interested in – here file.size, ie all_files[index].size

Q4

Consider the following code:

```
unsigned char check_array[32768];
int mystery = /* unknown */;

int Check(int key) {
    return check_array[(key + mystery) % 32768];
}
```

Suppose that:

- (to simplify the problem) virtual memory is not use
- `check_array` is located at physical address `0x1400000`
- the system has a 2-way 64KB (2 to 16 byte) data cache with 64-byte cache blocks.
- `Check` is compiled to perform exactly three memory accesses:
  - to read the global variable read `mystery`
  - to reads its return address from the stack
  - to read from `check_array`

Suppose we determine that calling `Check` evicts from the data cache sets as follows:

```
key value | evicts values from cache set indexes
------------------------------------------------
0         | 15, 72, 435
32        | 15, 72, 435
48        | 15, 72, 436
128       | 15, 72, 437
8240      | 15, 52, 72
```

Based on this information what is a possible value for `mystery`?

- (0 + mystery) // 64 +/- multiple of 512 (number of cache sets) = 435
- (32 + mystery) // 64 +/- multiple of 512 (number of cache sets) = 435
- (48 + mystery) // 64 +/- multiple of 512 (number of cache sets) = 436
- …

There was a typo originally that we fixed. Originally wrote %16384 instead of %32768, and 4096 instead of 8240 for the last row

Suppose we determine that calling `Check` evicts from the data cache sets as follows:

```
key value | evicts values from cache set indexes
------------------------------------------------
0         | 15, 72, 435
32        | 15, 72, 435
48        | 15, 72, 436
128       | 15, 72, 437
8240      | 15, 52, 72
```

Based on this information what is a possible value for `mystery`?

Answer:

Key: `a value between 27856 and 27871 +/- any multiple of 32768`

originally we erroneously wrote % 16384 instead of % 32768, and 4096 instead of 8240 for the last value

(0 + mystery) // 64 +/- multiple of 512 (number of cache sets) = 435

(32 + mystery) // 64 +/- multiple of 512 (number of cache sets) = 435

(48 + mystery) // 64 +/- multiple of 512 (number of cache sets) = 436

...

let's choose the mulitple of 512 to be 0 for simplicity, for now

mystery // 64 = 435 implies mystery in [435 * 64 = 27840, 27840 + 63 = 27903]

(32 + mystery) // 64 = 435 implies mystery in [435 * 64 - 32 = 27808, 27808 + 63 = 27871]

(48 + mystery) // 64 = 436 implies mystery in [436 * 64 - 48 = 27856, 27856 + 63 = 27919]

overlap here implies mystery in [27856, 27871]

the multiple of 512 offsets this by 512 * 64 = 32768

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?

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
  -