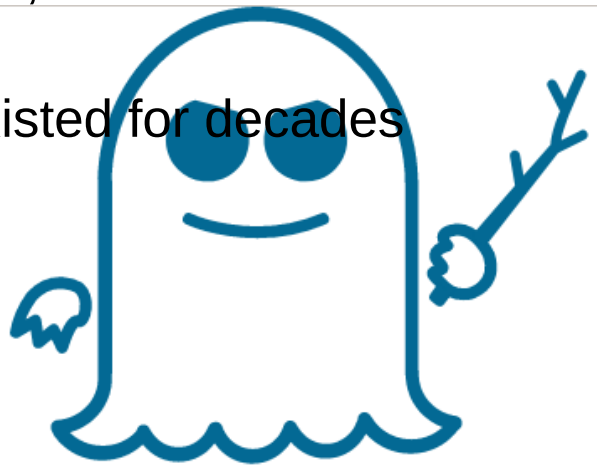
Meltdown/Spectre

Hardware bugs: context

- The *other* famous CPU bug is the Intel floating-point division bug
  - Occurs in the FDIV opcode of Pentium processors (1993-1994)
- Some divisions gave wrong results
  - Divide 4,195,835 by 3,145,727
  - The correct answer is 1.3338204491362410 02
  - The buggy chip would give 1.333739068902037589
- The problem was a typo in a lookup table. Once discovered, it was easy to fix, but required a new CPU (which Intel offered to provide to all affected users)
- This was highly embarrassing and expensive for Intel, but had surprisingly little impact on real applications
- Unlike the FDIV bug, Meltdown/Spectre is a fundamental design issue that has no easy solution. Also unlike FDIV bug, it affects many processors; has existed for a long time; and presents important real-world security vulnerabilities

Meltdown/Spectre

- These are *hardware bugs*
  - They are caused by incorrect designs of modern processors
  - Unlike software bugs, they are difficult to fix
  - They are security vulnerabilities: they allow programs to access private data that they shouldn't (such as passwords)
  - They can be exploited by any malicious program running on your computer, including JavaScript programs that are embedded within web pages
  - They arise due to complex edge cases with caches, speculative execution, and virtual memory
  - They were discovered in 2018, mostly at Google, and affect most modern processors
  - Although discovered recently, the bugs have existed for decades



Hardware bugs: context

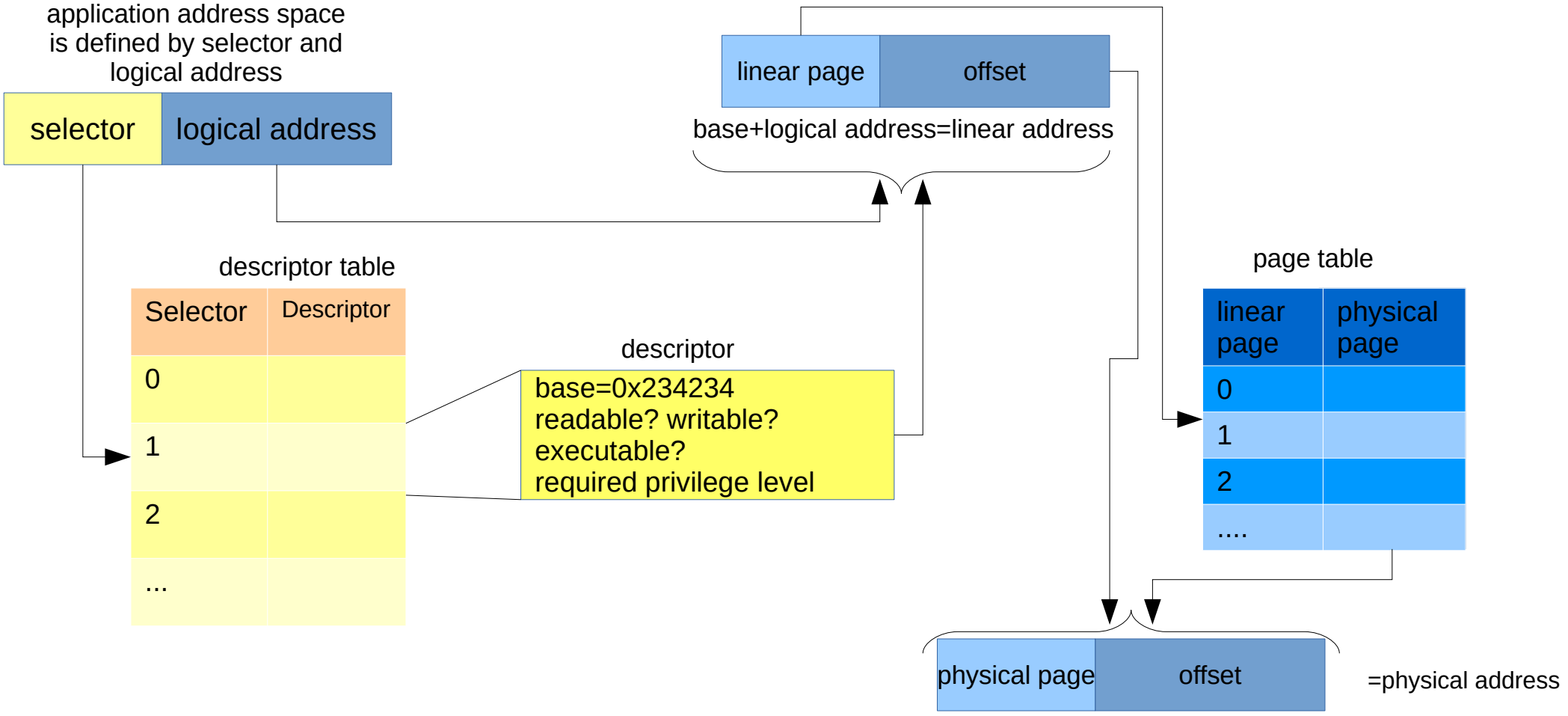
- The *other* famous CPU bug is the Intel floating-point division bug
  - Occurs in the FDIV opcode of Pentium processors (1993-1994)
- Some divisions gave wrong results
  - Divide 4,195,835 by 3,145,727

Hardware bugs: context

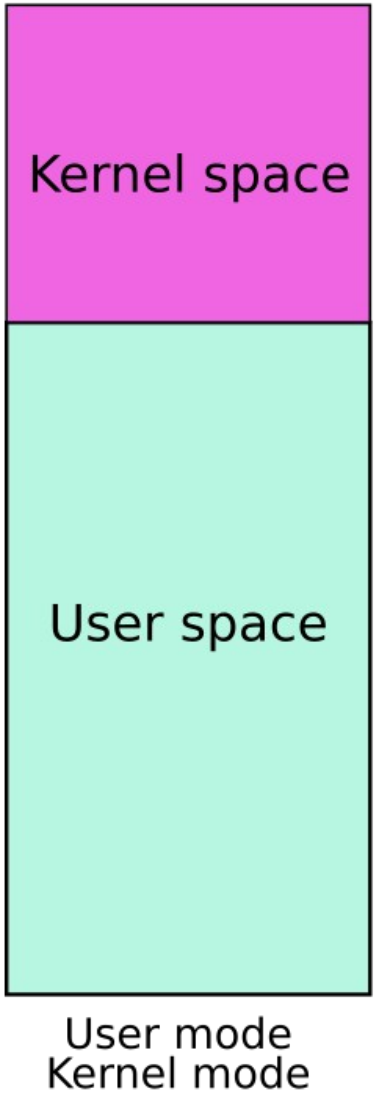
- The *other* famous CPU bug is the Intel floating-point division bug
  - Occurs in the FDIV opcode of Pentium processors (1993-1994)
- Some divisions gave wrong results
  - Divide 4,195,835 by 3,145,727
  - The correct answer is 1.3338204491362410 02
  - The buggy chip would give 1.333739068902037589
- The problem was a typo in a lookup table. Once discovered, it was easy to fix, but required a new CPU (which Intel offered to provide to all affected users)
- This was highly embarrassing and expensive for Intel, but had surprisingly little impact on real applications
- Unlike the FDIV bug, Meltdown/Spectre is a fundamental design issue that has no easy solution. Also unlike FDIV bug, it affects many processors; has existed for a long time; and presents important real-world security vulnerabilities

Meltdown: background info

- Segmentation
  - Intel architecture has an elaborate mechanism for determining memory access permissions and translating addresses



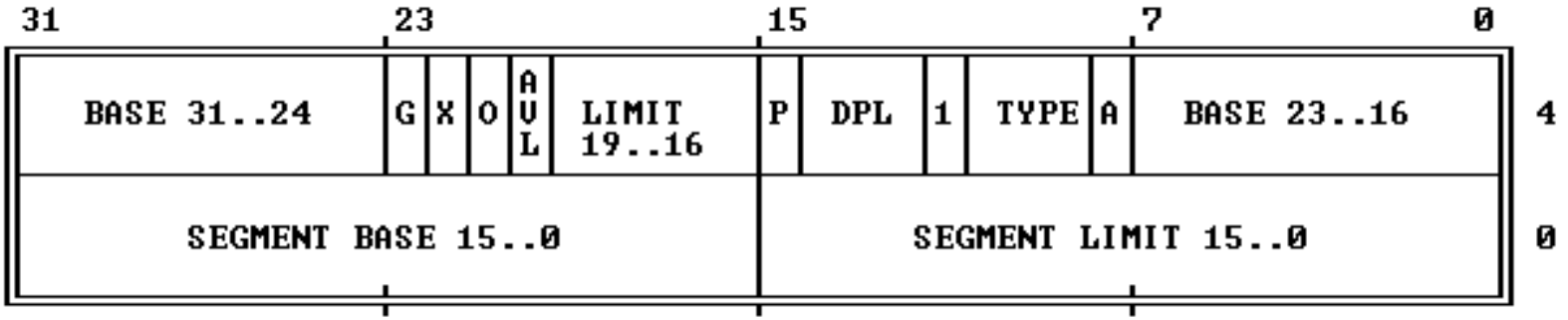
Typical address space layout



OS and application share a single address space. The kernel area is reserved for OS use.

Figure 5-3. General Segment-Descriptor Format

DESCRIPTORS USED FOR APPLICATIONS CODE AND DATA SEGMENTS



Base Address
32 bit starting memory address of the segment
Segment Limit
20 bit length of the segment. (More specifically, the address of the last accessible data, so the length is one more that the value stored here.) How exactly this should be interpreted depends on other bits of the segment descriptor.
G=Granularity
If clear, the limit is in units of bytes, with a maximum of 220 bytes. If set, the limit is in units of 4096-byte pages, for a maximum of 232 bytes.
D=Default operand size
If clear, this is a 16-bit code segment; if set, this is a 32-bit segment.
B=Big
If set, the maximum offset size for a data segment is increased to 32-bit 0xffffffff. Otherwise it's the 16-bit max 0x0000ffff. Essentially the same meaning as "D".
L=Long
If set, this is a 64-bit segment (and D must be zero), and code in this segment uses the 64-bit instruction encoding. "L" cannot be set at the same time as "D" aka "B".
AVL=Available
For software use, not used by hardware
P=Present
If clear, a "segment not present" exception is generated on any reference to this segment
DPL=Descriptor privilege level
Privilege level (ring) required to access this descriptor
Type
If set, this is a code segment descriptor. If clear, this is a data/stack segment descriptor, which has "D" replaced by "B", "C" replaced by "E"and "R" replaced by "W". This is in fact a special case of the 2-bit type field, where the preceding bit 12 cleared as "0" refers to more internal system descriptors, for LDT, LSS, and gates.
C=Conforming
Code in this segment may be called from less-privileged levels.
E=Expand-Down
If clear, the segment expands from base address up to base+limit. If set, it expands from maximum offset down to limit, a behavior usually used for stacks.
R=Readable
If clear, the segment may be executed but not read from.
W=Writable
If clear, the data segment may be read but not written to.
A=Accessed
This bit is set to 1 by hardware when the segment is accessed, and cleared by software.

Meltdown: background info

- Descriptors
  - carry privilege information about memory regions
    - is the data readable? for the given process
    - is the data writable? for the given process
    - is the data executable? for the given process
  - provide protection of address spaces, including the OS
- Every memory access
  - is translated from logical address to linear address (via descriptor) to physical address (via page table)
  - Requires a privilege check on the given memory region

```
int main() {
    int *x = (int *)0x12345678;
    int y = *x; // requires a privilege check
    cout << y;
}
```

If the current process fails a privilege check, it causes a *protection fault*, which usually means the OS will kill the program. You see this as a **`Segmentation fault`**

Meltdown: background info

- We talked about *branch prediction*:

```
jeq $1, $0, foo
movi $2, 2
foo: movi $3, 3
```

- Branch prediction is form of *speculative execution*:
  - If you don't know the outcome of a condition, expect the most common outcome
  - If the outcome is different, you have to "unroll" the executed completed code
- Intel uses speculative execution for every memory access
  - We assume that the memory access is okay
  - If the privilege check fails, we unroll the subsequently executed code

```
int main() {
    int *x = (int *)0x12345678;
    int z = 0;
    int y = *x; // requires a privilege check
    z = 1;      // we might have to unroll this
}
```

Meltdown: background info

- In the case of a protection fault, the application is usually killed by the OS
- However, the application can handle the fault instead, allowing it to survive a protection fault
- Unroll occurs in any case

```
void signal_handler(int signo) {
    cout << "I am not allowed to read that address";
}
int main() {
    signal(SIGSEGV, signal_handler);          // register handler
    int *x = (int *)0x12345678;
    int y = *x;                               // allowed?
    cout << "I am allowed to read that address";
}
```

In effect, this means an application can *try* to access a memory location, and on the basis of trial-and-error, determine which addresses are accessible.

Meltdown: background info

- Intel architecture usually has a three-level cache, similar to what we've discussed
- Applications can detect whether a particular block is in the cache using a *timing attack*

```
int main() {
    int array[50]; // Is this in the cache?
    int before_time = get_current_cycle_count();
    int y = array[25];
    int after_time = get_current_cycle_count();
    if (after_time - before_time > THRESHOLD)
        cout << "array[25] was probably NOT cached";
    else
        cout << "array[25] was probably cached";
}
```

Let's assume we have a function `is_cached` that will tell us whether a particular variable is in the cache.

Meltdown: background info

- We now understand the three components that go into Meltdown:
  - virtual memory
  - speculative execution
  - caching

Meltdown: goals

- A malicious program can use Meltdown to read memory that it shouldn't be allowed to
- Some OS memory regions are mapped into application address space, but marked unreadable in the descriptor table
- With Meltdown, a normal application can read private OS data structures
- The program below demonstrates what a malicious program may *try* to do. By exploiting Meltdown, it becomes possible.
  This program reads only one byte. Realistically, a malicious program would probably read as much memory as it could, by invoking the exploitation repeatedly.

```
int main() {
    char *pointer_to_secret = (char *)0x12345678;
    char secret = *pointer_to_secret;
    cout << "Now I know your secret is " << secret;
    // This won't work, but it shows what we are trying to do
}
```

Meltdown

- Consider this code:

```
#define BLOCKSIZE 8
int array[256 * BLOCKSIZE];

int handler(int signo) {
    char secret;
    for (int i=0; i<256; i++)
        if (is_cached(array[i*BLOCKSIZE]))
            secret = i;
    cout << "Now I know your secret is " << secret;
}

int main() {
    char *pointer_to_secret = (char *)0x12345678;
    int dummy;
    signal(SIGSEGV, handler);
    dummy = array[(*pointer_to_secret) * BLOCKSIZE]; // cause fault
}
```

Meltdown

- Consider th

We register our signal handler, so that when a protection fault occurs, we can handle it, instead of getting killed.

```
#define BLOCKSIZ
int array[256 *

int handler(int
    char secret;
    for (int i=0;
        if (is_cached(a                OCKSIZE]))
            secret = i;
    cout << "Now I kn          r secret is " << secret;
}

int main() {
    char *pointe_to_secret = (char *)0x12345678;
    int dummy;
    signal(SIGSEGV, handler);
    dummy = array[(*pointer_to_secret) * BLOCKSIZE]; // cause fault
}
```

Meltdown

- Consider thi

```
#define BLOCKSIZ
int array[256 *

int handler(int
    char secret;
    for (int i=0;
        if (is_cached(ar                OCKSIZE]))
            secret = i;
    cout << "Now I know          ecret is " << secret;
}

int main() {
    char *pointer_to_se         t = (char *)0x12345678;
    int dummy;
    signal(SIGSEGV, han         ler);
    dummy = array[(*pointer_to_secret) * BLOCKSIZE]; // cause fault
}
```

This dereference causes the protection fault. We aren't allowed to access that memory.

However, thanks to speculative execution, this line will be executed: we will read an element from `array` and store it into `dummy`.

However, the write to dummy will get unrolled.

Meltdown

- Consider this code:

```
#define BLOCKSIZE 8
int array[256 * BLOCKSIZE];

int handler(int signo) {
    char secret;
    for (int i=0; i<256; i++)
        if (is_cached(array[i*BLOC
            secret = i;
    cout << "Now I know your secr
}

int main() {
    char *pointer_to_secret = (char *)0x12345678;
    int dummy;
    signal(SIGSEGV, handler);
    dummy = array[(*pointer_to_secret) * BLOCKSIZE]; // cause fault
}
```

When the protection fault occurs, the CPU unrolls the write to `dummy` and calls our handler.

Melt

- Consider this code:

```
#define BLOCKSIZE 8
int array[256 * BLOCKSIZE];

int handler(int signo) {
    char secret;
    for (int i=0; i<25
        if (is_cached(arra
            secret = i;
    cout << "Now I know your
}

int main() {
    char *pointer_to_secret = (char *)0x12345678;
    int dummy;
    signal(SIGSEGV, handler);
    dummy = array[(*pointer_to_secret) * BLOCKSIZE]; // cause fault
}
```

KEY INSIGHT

The "unrolling," i.e. the reversal of speculatively-executed code takes into account most elements of state that are "visible" to an application, including registers, program counter, and memory.

However, the state of the cache will *not* be unrolled to its pre-fault state.

Therefore, the cache may contain data that was cached in the code that was unrolled.

We can't access cache directly, but we can determine which addresses are cached.

Meltdown

- Consider this code:

```
#define BLOCKSIZE 8
int array[256 * BLOCKSIZE];

int handler(int signo) {
    char secret;
    for (int i=0; i<256; i++)
        if (is_cached(array[i*BLOCKSIZE]))
            secret = i;
    cout << "Now I know your secret is " << secret;
}

int main() {
    char *pointer_to_secret = (char *)0x12345678;
    int dummy;
    signal(SIGSEGV, handler);
    dummy = array[(*pointer_to_secret) * BLOCKSIZE]; // cause fault
}
```

We check which element of `array` is cached. This will tell us which element of the array was accessed during the speculatively-executed code in `main`, before it was unrolled.

Meltdown

- Consider this code:

```
#define BLOCKSIZE 8
int array[256 * BLOCKSIZE];

int handler(int signo) {
    char secret;
    for (int i=0; i<256; i++)
        if (is_cached(array[i*BLOCKSIZE]))
            secret = i;
    cout << "Now I know your secret is " << secret;
}

int main() {
    char *pointer_to_secret = (char *)0x12345678;
    int dummy;
    signal(SIGSEGV, handler);
    dummy = array[(*pointer_to_secret) * BLOCKSIZE]; // cause fault
}
```

Example: if the value we *tried* to read was 5, then **`array[5*BLOCKSIZE]`** will now be in the cache.

We have to multiply by BLOCKSIZE, because the CPU won't cache just one memory cell.

Meltdown

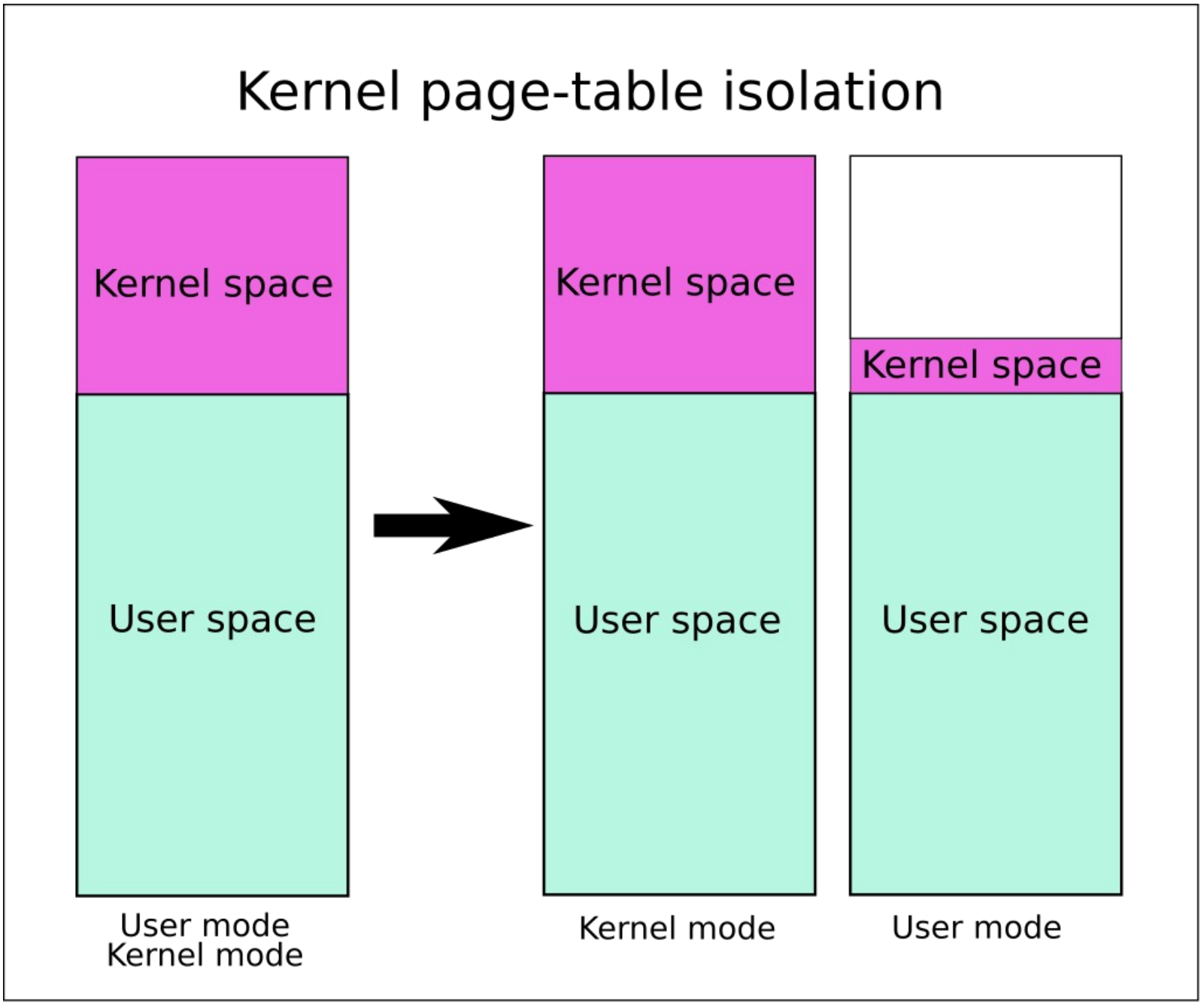
- Consider this code:

```
#define BLOCKSIZE 8
int array[256 * BLOCKSIZE];

int handler(int signo) {
    char secret;
    for (int i=0; i<256; i++)
        if (is_cached(array[i*BLOCKSIZ
            secret = i;
    cout << "Now I know your secret is " << secret;
}

int main() {
    char *pointer_to_secret = (char *)0x12345678;
    int dummy;
    signal(SIGSEGV, handler);
    dummy = array[(*pointer_to_secret) * BLOCKSIZE]; // cause fault
}
```

We have effectively read memory
that the CPU does not allow us to read.

How to fix Meltdown?

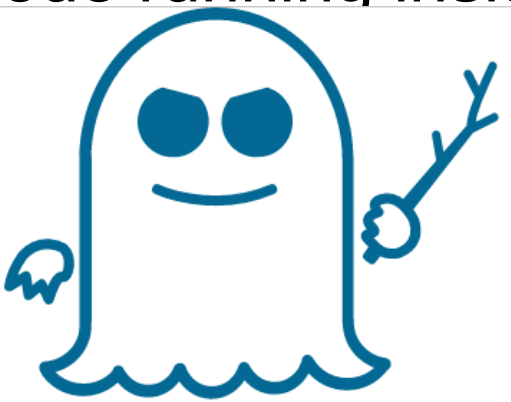
- Option 1: unroll cache state on wrongly-predicted code
  - too slow
  - requires new CPU
- Option 2: avoid speculatively executing memory accesses
  - some performance impact
  - the page table's "supervisor bit" can be used to prevent speculation to privileged regions
  - some processors already do this, e.g. AMD processors
- Option 3: don't map sensitive OS memory into application address space
  - system calls will be slower
  - can be solved with OS upgrade
  - "kernel page-table isolation" (KPTI)



On the left, a single page table serves both application code ("user mode") and OS code.
On the right, separate page table serve the application and the application.

Spectre V1: background info

- Meltdown lets us violate protection between application and OS
- Spectre lets us violate protection between application and application
  - We assume that the applications communicate, allowing one (the "attacker") to manipulate values in the other (the "victim")
  - Common example: JavaScript code running inside a web browser



Spectre V1: background info

Victim

JavaScript interpreter in web browser

```
if (x < array1_size)
  y = array2[ array1[x] * 256 ];
```

Attacker

JavaScript program

```
var arr = Array(50);
var x = 200;
var dummy = arr[x];
```

Spectre V1: background info

Victim

JavaScript interpreter in web browser

```
if (x < array1_size)
  y = array2[ array1[x] * 256 ];
```

Attacker

JavaScript program

```
var arr = Array(50);
var x = 200;
var dummy = arr[x];
```

The attacker and victim communicate via variable x, which we assume is the same in both.

Spectre V1: background info

Victim

JavaScript interpreter in web browser

```
if (x < array1_size)
   y = array2[ array1[x] * 256 ];
```

The victim probably has this kind of *guard*, to prevent invalid memory accesses. It prevents x from being too large.

The body of the **`if`** will be executed *speculatively.*

Attacker

JavaScript program

```
var arr = Array(50);
var x = 200;
var dummy = arr[x];
```

Spectre V1: background info

Victim

JavaScript interpreter in web browser

```
if (x < array1_size)
  y = array2[ array1[x] * 256 ];
```

During speculative execution, **x** may be much larger than the size of **array1**.

In C/C++, array indexing is calculated by simple addition: **array1[x]** means **(&array1)+x**.

Therefore, the value of highlighted expression may be *any arbitrary value in the victim's address space*.

Attacker

JavaScript program

```
var arr = Array(50);
var x = 200;
var dummy = arr[x];
```

Spectre V1: background info

Victim

JavaScript interpreter in web browser

```
if (x < array1_size)
  y = array2[ array1[x] * 256 ];
```

i=array1[x] is the secret we're trying to extract.
Similar to Meltdown, the ***i***th element of **array2** will be cached.
As in Meltdown, the attacker can check which element of **array2** has been cached.
That's the secret.

Attacker

JavaScript program

```
var arr = Array(50);
var x = 200;
var dummy = arr[x];
```

Spectre V1: goals

- A malicious program can use Spectre to read memory that it shouldn't be allowed to
- In the previous example, we showed how a JavaScript program can read arbitrary memory form a web browser
  - such as passwords, emails, grades
  - any data in any tab
- It's harder to apply than Meltdown
  - Spectre requires a communication channel between attacker and victim
  - and it requires that the victim use that channel in certain ways, i.e. as an index to an array
  - Such opportunities are very common
- What software does Spectre affect?
  - Web browsers
  - Emulators (such as for running old games)
  - Interpreted, sandboxed languages such as eBPF

How to fix Spectre V1?

- Option 1: unroll cache state on wrongly-predicted code
  - too slow
  - requires new CPU
- Option 2: avoid speculatively executing memory accesses
  - too slow
  - requires new CPU
- Option 3: prevent precise timing in untrusted code
  - detecting cache status require very precise timing
  - web browsers (and other trusted code) can deny access to precise timing mechanisms
  - this is being done: new web browsers reduce the precision of JS functions like **`performance.now()`**

Spectre V2

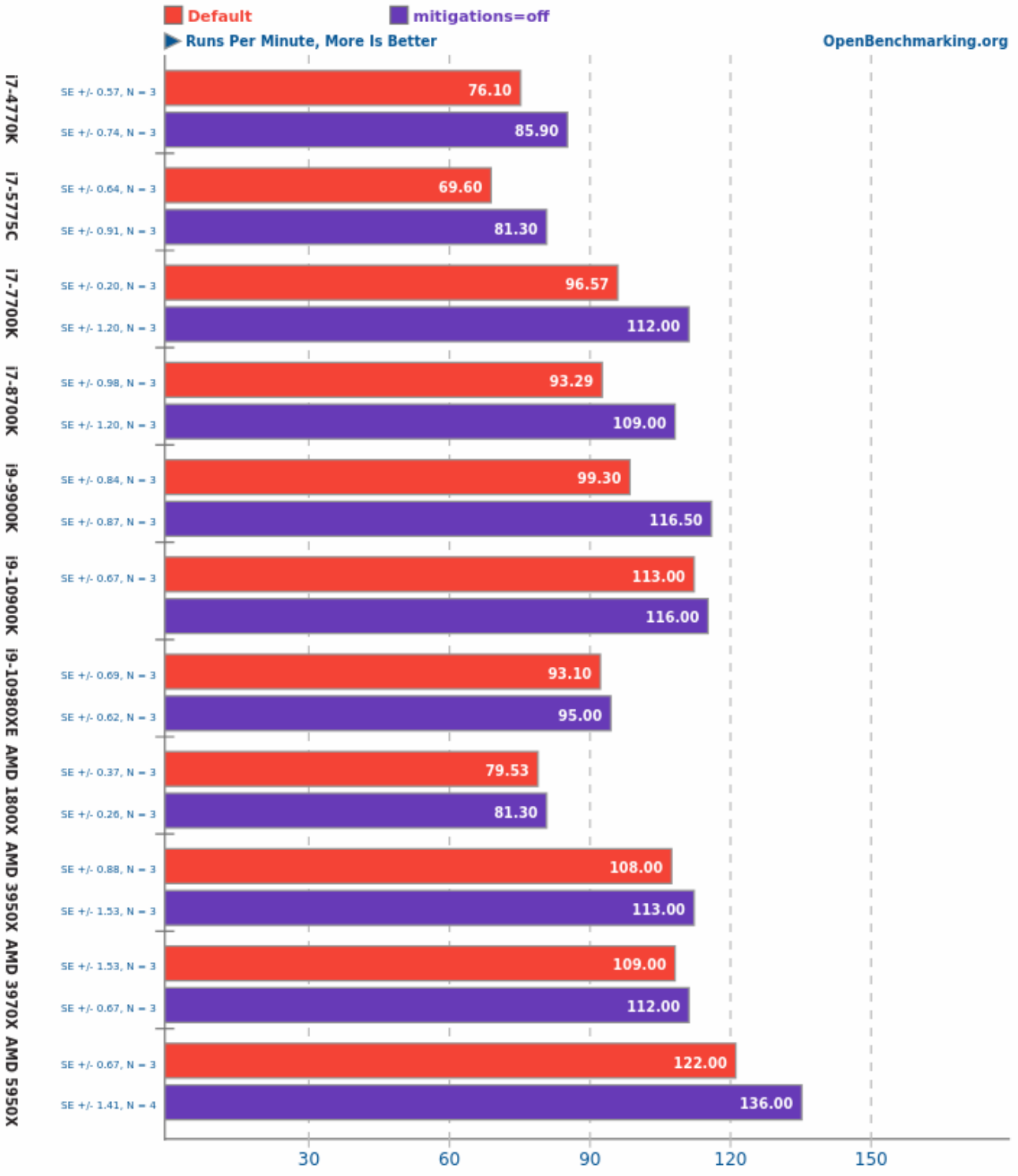
- Another Spectre approach (the so-called Variant 2) is based on indirect jumps
  - Indirect jumps mean that the jump target is determined by a register value
  - In E20, that would be `jr`
    - `jr $7  # jump to the address in register $7`
  - In Intel, we can use `jmp` with a register argument
    - `jmp eax ; jump to address in register eax`
- See also Retpoline

Spectre/Meltdown mitigation cost

https://www.phoronix.com/scan.php?page=article&item=3-years-specmelt&num=2

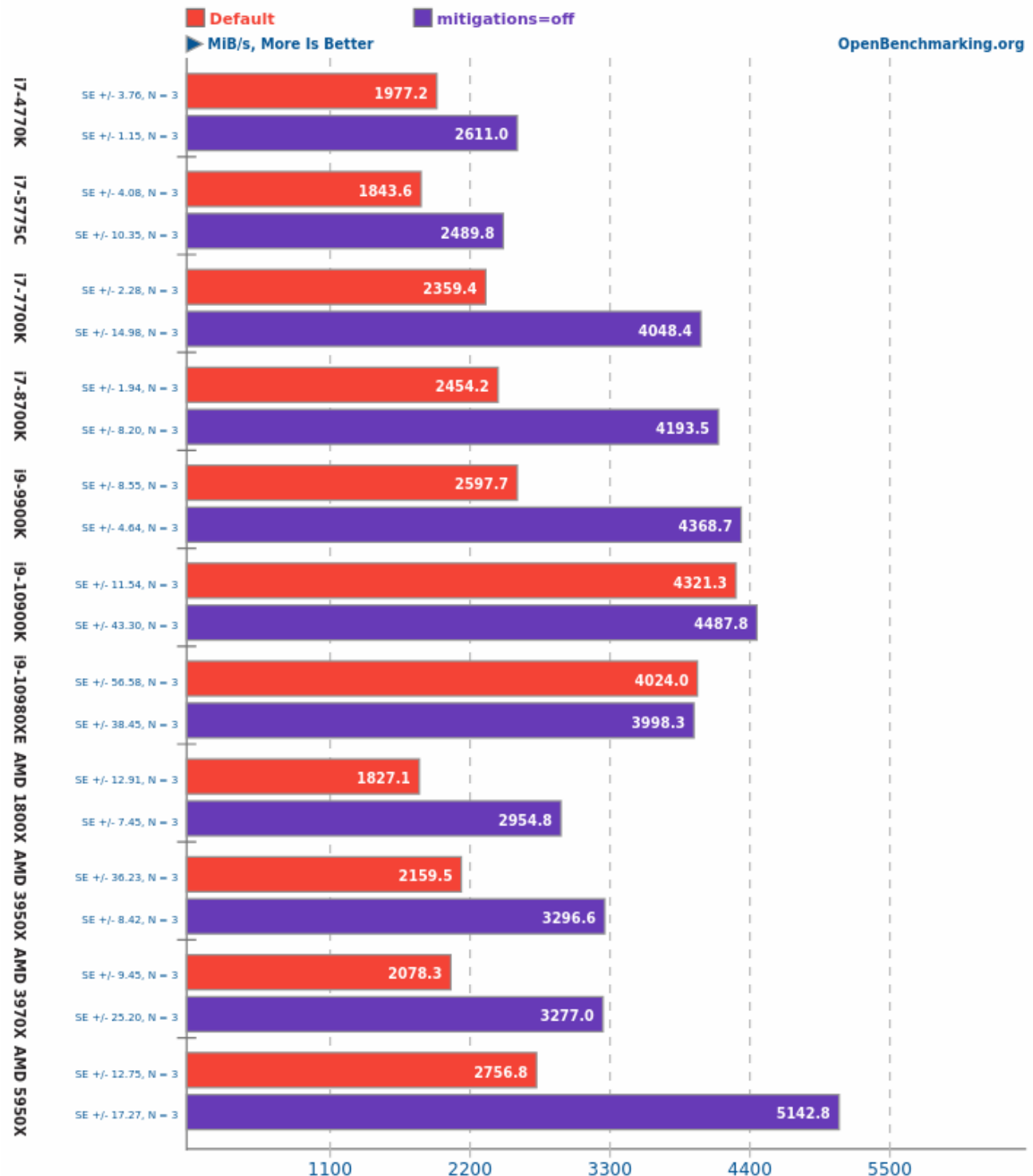
Selenium

Benchmark: Speedometer - Browser: Firefox



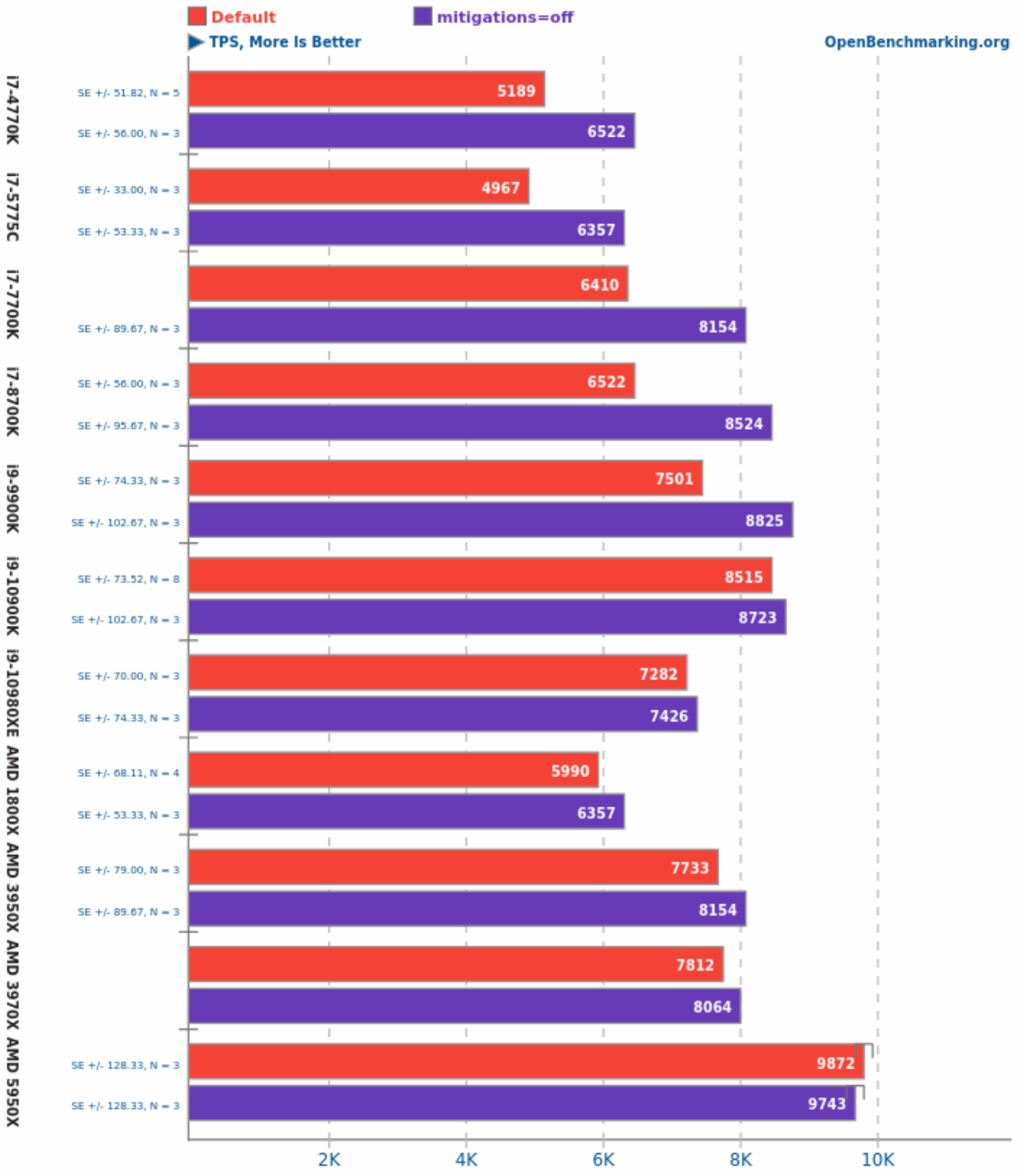
Cryptsetup

AES-XTS 256b Encryption



PostMark 1.51

Disk Transaction Performance



1. (CC) gcc options: -O3

perf-bench

Benchmark: Futex Hash

1. (CC) gcc options: -O6 -ggdb3 -funwind-tables -std=gnu99 -Xlinker -lpthread -lrt -lm -ldl -lelf -lcrypto -lz -lnuma