Fixing hardware faults with software patches

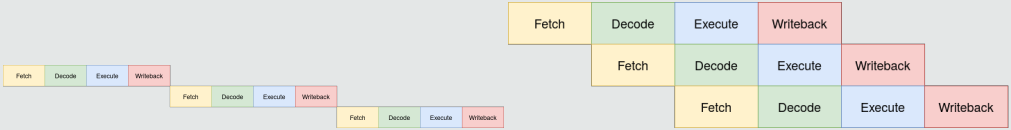
Pierre-Emmanuel Patry

How did we get here ?

Let’s take a look at modern microprocessor history

Problem: Power & memory wall

Solution: Pipelines



Problem: Pipeline limitations

Solution: Deeper pipelines

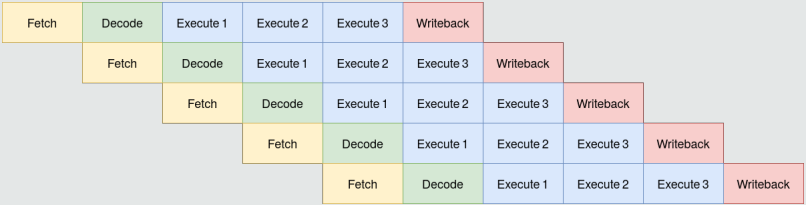


Figure 1: deep pipeline

Problem: Not all pipelines have the same length

Solution: Superscalar processors

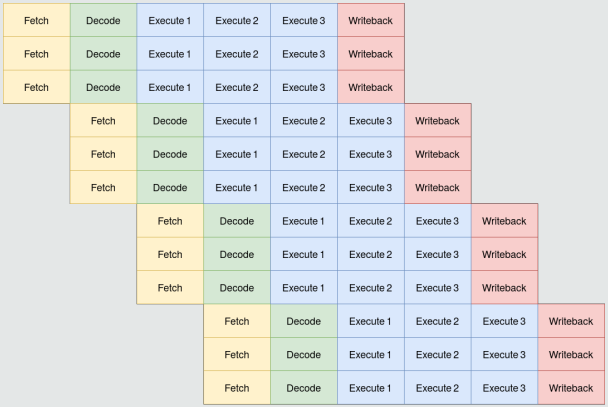


Figure 2: superscalar processor

Problem: Branches and long operations

Solution: Speculative execution

- Branch predictors
  - 1 level branch predictors (history [93.5%])
  - 2 level branch predictors (patterns [~97%])
  - Hybrid branch predictors
  - Perceptron based neural predictor
- Memory dependence prediction

Speculative instructions

- Arithmetical
- Logical
- Memory read
- Memory write (uncommitted)
- Branches

Speculative instructions

- Arithmetical
- Logical
- Memory read (with cache access)
- Memory write (uncommitted)
- Branches

And Spectre?

Context

- Same machine (Container/VMs)
- Mean to measure time

On rollbacks, context is discarded but microarchitectural effects remain

- Cache lines are shared between speculative and real execution.
- Cache lines are shared between multiple adresses.

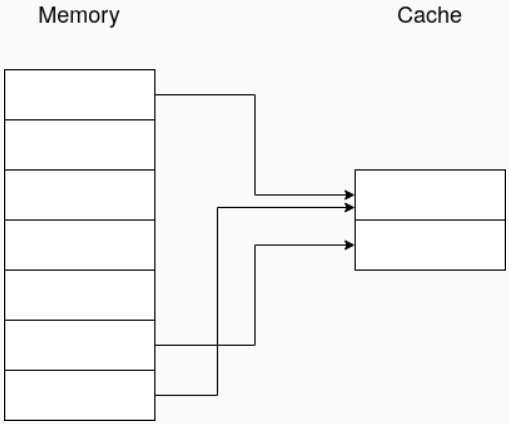


Figure 3: cache line repartition

Taking advantage of this

How could we take advantage from this mechanism ?

Taking advantage of this

We could measure the time taken for cache misses and hit!

Fixes

Hardware

- Microcode update
- Architectural changes (Safespec)

Software

- Constrain speculation
  - Rewrite code with speculation in mind
  - Insert instructions to strop out of order executions
  - Place the sensitive process (kernel) in another virtual address space

- Bound check bypass

```
unsigned int array_inner_size = 16;
uint8_t array_inner[160] = { /* Random values */ };
uint8_t array_outer[256 * CACHE_LINE_SIZE];// Properties
string secret = "secret data";

int fetch_function(size_t idx)
{
    if (idx < array_inner_size)
    {
        return array_outer[array_inner[idx] * CACHE_LINE_SIZE];
    }
    return -1;
}
```

Actual attack

1. Flush array_outer out of cache
2. mistrain the BP
3. pass the address of the secret
4. check wether current char is in the cache or not

Actual attack

```
unsigned int d = 0;
uint64_t start, diff;
char *target_address;
char current;

for (int i = 0; i < 256; i++) {
    _mm_clflush(&array_outer[i * CACHE_LINE_SIZE]);
}

for (int i = 0; i < T; i++) {
    _mm_clflush(&array_inner_size);

    idx = (attacking * target_address) + (train_index * !attacking);

    fetch_function(idx);
}
```

Actual attack

```
for (int i = 'a'; i <= 'z'; i++) {
    current = randomized[i];
    addr = &array_outer[current * CACHE_LINE_SIZE];
    start = __rdtscp(&d);
    d = *target_address;
    diff = __rdtscp(&d) - start;

    // Make an histogram of values
    HIST(diff);
}
```

Constant time mask

```
static inline unsigned long array_index_mask_nospec(unsigned long index,
                                unsigned long size)
{
    /*
     * Always calculate and emit the mask even if the compiler
     * thinks the mask is not needed. The compiler does not take
     * into account the value of @index under speculation.
     */
    OPTIMIZER_HIDE_VAR(index);
    return ~(long)(index | (size - 1UL - index)) >> (BITS_PER_LONG - 1);
}
```

Constrained index

```
#define array_index_nospec(index, size)                                 \
({                                                                      \
    typeof(index) _i = (index);                                         \
    typeof(size) _s = (size);                                           \
    unsigned long _mask = array_index_mask_nospec(_i, _s);              \
                                                                        \
    BUILD_BUG_ON(sizeof(_i) > sizeof(long));                            \
    BUILD_BUG_ON(sizeof(_s) > sizeof(long));                            \
                                                                        \
    (typeof(_i)) (_i & _mask);                                          \
})
```

Target indirect branch predictor.

Function pointers

```
// We assume "quack_function" is not a constant that could be
// propagated by the compiler.
void (*quack)(Duck*) = &quack_function;
quack(donald);
```

Vtables

```
Animal *duck = new Duck();
duck->eat();
```

Retpoline: Jump

Without retpoline

```
jmp *%rax
```

With retpoline

```
call load_label
capture_ret_spec:
    pause
    jmp capture_ret_spec
load_label:
    mov %rax, (%rsp)
    ret
```

Retpoline: Call

Without retpoline

```
call *%rax
```

With retpoline

```
    jmp label2
label0:
    call label1
capture_ret_spec:
    pause
    jmp capture_ret_spec
label 1:
    mov %rax, (%rsp)
    ret
label2:
    call label0
; Continue
```

Marssx86 demo

Consequences

- Deprecation of intel SGX
- Most mitigation/fixes cover only one variant of this attack category
- Those mitigations lead to high overhead (30% slowdown on average)

Bibliography

- A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware - Ge & al
- Verifying Constant-Time Implementations - Almeida & al
- SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation - Khasawneh & al
- A Survey of Techniques for Dynamic Branch Prediction - Mittal Two level adaptative branch prediction - Yeh & al