Computer Systems 1
Lecture 21

Spectre and Meltdown

Dr. John T. O'Donnell
School of Computing Science
University of Glasgow

Topics

Announcements

- Coming up
  - Revision and support tutorials Wednesday through Friday, kindly volunteered by the tutors
  - Advanced topic: Friday 2pm in SAWB 303: the Sigma16 digital circuit (Version M1). We'll look at how the processor works, and we will run the ArrayMax program on the circuit
- Where we are
  - This is the last week of the course
  - The last lecture is Thursday
  - There's a lab this week: use it to finish up your assessed exercise
  - Handin deadline is Friday. You don't have to hand in a completed program! You get marks for what you have accomplished
  - A large portion of the assessment is for the comments in your code
  - Don't forget Quiz 10 (closes Friday) and Quiz 11 (opens Thursday)

About today's topic

- It's similar to the Advanced Topics we hold Fridays
- The subject is deep and complex
- It is also important and topical
- The lecture will focus on the main ideas
- The slides contain additional material
- At the end there is a short Summary
    - You should read and understand the Summary
    - The details in the rest of the slides are optional: you don't need to understand the details

The New York Times, 3 January 2018

Paul Kocher, left, moderating the RSA Conference 2016 in San Francisco. Mr. Kocher is an independent researcher who was an integral part of the team that discovered the flaws. Jim Wilson/The New York Times

SAN FRANCISCO — Computer security experts have discovered two major security flaws in the microprocessors inside nearly all of the world's computers.

The two problems, called Meltdown and Spectre, could allow hackers to steal the entire memory contents of computers, including mobile devices, personal computers and servers running in so-called cloud computer networks.

There is no easy fix for Spectre, which could require redesigning the processors, according to researchers. As for

MELTDOWN

SPECTRE

Two problems

- Two security flaws enable an attacker to read private data belonging to a victim
  - All data that's in memory, including passwords, financial data, everything
- Meltdown
  - Affects Intel processors
  - There is a "patch" but it slows down the system by 5% to 30%
- Spectre
  - A theoretical discovery, very difficult to exploit
  - But . . .
    - It affects nearly all processors
    - There is no known solution — no software patch
- For a computing professional, it's important to have some understanding of what these do

Important!

- You need to keep your system updated
- Any computer (Windows, Macintosh, Linux) that hasn't had its operating system updated since December 2018 is vulnerable to Meltdown
- If a hacker is able to execute code on your computer (have you ever visited a web page?), they can steal the entire contents of your memory, including all your passwords, financial records, plans to take over the world, . . .
- Microsoft has made automatic updates mandatory for Windows 10

Brings together several topics

- Spectre and related attacks are a major topic in computer systems
- They affect everyone (whether they know it or not)
- We're going to look at how they work
- This is a very advanced subject, and it involves
  - Digital circuits
  - Computer architecture
  - Instruction level parallelism and speculative execution
  - Machine language
  - Interrupts and processor status
  - Cache
  - Operating system
- *This lecture is a taster in what state of the art research looks like*

Overview

- A victim process has a secret data byte S in its memory. This is private data; others can't read it because of memory protection.
- An attacker process can determine whether a victim process has accessed a memory address recently
- There is a code snippet that, if it were executed, would cause the victim to access a memory address in a cache line which depends on the value of S.
- Although the code snippet cannot execute (because of memory protection), it's possible to trick the CPU into executing it *speculatively*.
- After the speculative execution, the CPU will discover that the code snippet is not to be executed, and it will unroll the result.
- The code snippet speculatively executes a load instruction using an (invalid) address that depends on S
- This causes the cache line to be loaded into the cache.
- Attacker measures cache speed and deduces the value of S.

Instructions don't all take the same time!

```
lea     R1,23[R0]         1 clock cycle
load    R2,x[R0]          100 clock cycles
add     R3,R1,R2          1 clock cycle
```

- Fast instructions
  - addition, comparison, simple computations
- Slow instructions
  - memory access (load, store), difficult computations (floating point division)

Dealing with slow operations

- Memory accesses
  - Cache memory: a small fast memory that holds recently used data: pairs of effective address and contents
  - On each memory access, the processor looks first in the cache. Usually it finds the data quickly
  - If the data isn't in the cache, a slow memory access is started
- Show computations in general
  - The processor looks ahead in the instruction stream, and executes instructions out of order

Memory protection

- User process runs in an *address space* from 0 to $N - 1$, where N is the size of the space
- The hardware has a *segment address register sa* which it automatically adds to each effective address generated by user
- This causes the user addresses to be *mapped* to physical locations sa up to $sa + N - 1$
- The user *cannot* refer to any physical location below sa, and the hardware checks *every* access to make sure it isn't above $sa + N$.
- If a user process tries to access (load, store, or execute) a location with effective address outside the segment, the hardware generates an interrupt called a *segmentation fault*
- The interrupt transfers control to the operating system, which terminates the user process

Cache

. . .	
0009	cache line 2
0008	
0007	
0006	
0005	cache line 1
0004	
0003	
0002	
0001	cache line 0
0000	

How the cache works

- The cache maintains a list of pairs: effective address and contents
- Instruction generates an effective address
  - If this address is in cache (*cache hit*), the corresponding contents are accessed quickly
  - If not (*cache miss*), then the processor waits for slow access to the main memory.
  - When the data arrives eventually, this is placed into the cache

Locality

- Cache wouldn't do much good if a program accesses memory at random locations
- But real programs access the same data again and again
  - Example: in a loop, the loop index is used several times each iteration
- Locality means data that either
  - Has been accessed recently (and is likely to be used again soon)
  - Has an address close to other data accessed recently
- Main points
  - Cache hit is fast, cache miss is slow
  - Looking at a word of memory puts it into the cache
  - There is no way for an instruction to ask *is address a in the cache*
  - But we could do a *timing experiment* to find out

Out of order execution

```
add   R1,R2,R3         fast
load  R2,x[R0]         may be slow
sub   (not using R2)   execute before load finishes
add   (not using R2)   execute before load finishes
lea   (not using R2)   execute before load finishes
add   (not using R2)   execute before load finishes
add   R8,R2,R7         executes after load finishes
```

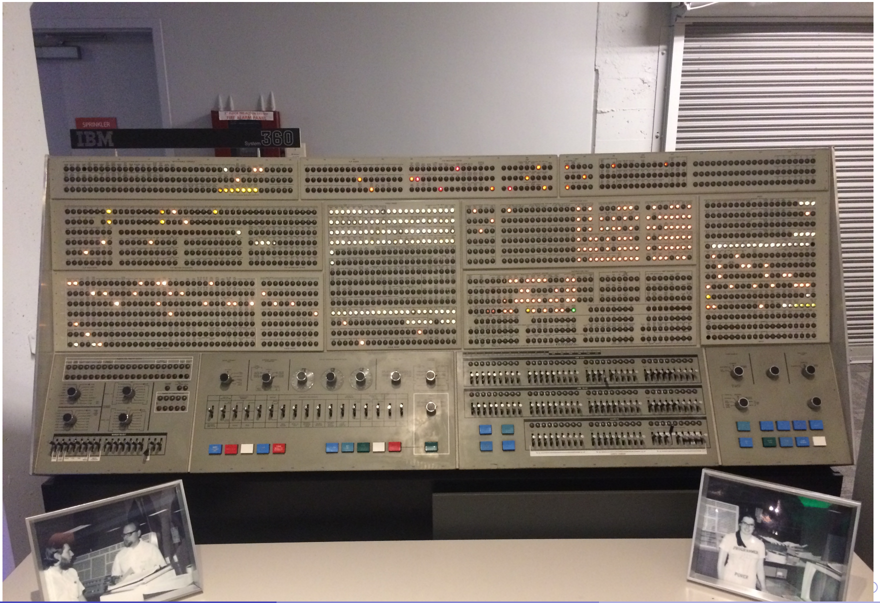
Superscalar processors

- Fast computers are fast because they have instruction level parallelism
- This means: they execute several instructions at the same time
- A basic form: pipelining
- A sophisticated form: superscalar
  - When the processor encounters an operation that will be slow (e.g. a memory access not in the cache), it continues on with the subsequent instructions

ALU vs functional units

- Fast operations — that can be completed in the ALU in one clock cycle — are just performed in a pipeline stage.
- Slow operations — that require many clock cycles — can be done in the datapath but they would stall the pipeline.
- In a high performance machine, the slow operations must be issued to a separate functional unit.
  - This includes memory accesses and floating point operations
  - This enables multiple functional units to operate in parallel.
  - It avoids stalling the pipeline.

The first superscalar computer: 360 Model (1966)



Parallel execution with functional units

- When pipeline encounters a “show” operation, it *issues* the calculation to a functional unit
- Several instructions can execute in parallel, and may finish out of order
- If there are data dependencies between the instructions, the pipeline must stall

Executing the operation

- When a functional unit has received both of its operands, it fires
- When it finishes, several cycles later, it transmits the result back to the destination register
- The destination register loads the result and sets its busy bit to 0
- If the pipeline was stalled on this register, it will resume at the next clock tick

Distributed control and the Common Data Bus

- The basic idea:
  - Issue operations to functional units, even if the data isn't known
  - Send a "name" indicating the register
  - In the destination register, place a "name" for the unknown result
  - The names can be passed around as if they were actual data
  - When a functional unit calculates a result, it broadcasts a message to the entire machine: "name = 3.14"
  - Every unit checks to see if it is waiting on "name" and if so, loads the actual value

The Common Data Bus

- Send data values from wherever they originate to wherever they are needed.
- We need a signal to carry them: the *common data bus*
- We need a way of identifying values (because a data value may not be an actual floating point number). These are called *tags*.
- Abstractly, it's useful to think of a tag as just a name for a value. Concretely, a tag is implemented as in id number for the unit that originates the value.

Distributed control and the Common Data Bus

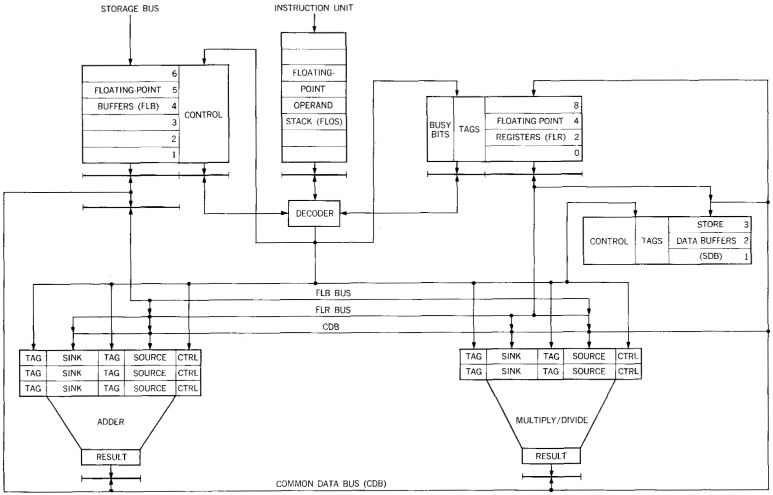


Figure 4 Data registers and transfer paths, including CDB and reservation stations.

Structure of the CDB

- Sources (places that can write a value onto the CDB): FLB (data coming from memory), and results from all of the functional units. (This includes all sources that can alter a register.)
- Destinations (places that can read a value from the CDB): operands of all the reservation stations (both source and sink); FLR (the floating point registers); SDB (the buffer registers for writing to the memory). (This includes all units that can receive data from a register.)
- Bus control: several sources may request to write onto the CDB at the same time. There is a bus control that chooses a winner, and delays the others.

Issuing an instruction

- If the operands are not busy
  - the values are sent to the reservation station using the FLR bus
  - The sink (destination) is set to busy, and also its tag is set to the id of the reservation station (this is the "name" of the result)
- If the sink is busy
  - *No stall!* The instruction is issued and the pipeline continues
  - The *value* of the sink (this is the first operand) is not transmitted to the reservation station; instead the tag is sent
  - The busy bit remains 1, but the tag in the register is changed to the name of the reservation station for the new result. *The contents of the register changes from the name of one unknown result to the name of another unknown result.*

When a functional unit finishes

- It requests to write on the CDB (several functional units could do this at the same time)
- It transmits a pair (id number of the reservation, value of the floating point number)
- *Every* unit that can read from the CDB checks to see whether it contains a tag that matches the broadcast tag. If so, the unit loads the data value from the CDB and sets control bits to indicate it is not busy
- It's quite possible for several destinations to receive the value simultaneously

Effect of the CDB

- Data is identified by its “name” (the tag of the unit that produces it), rather than its location
- Data is transmitted to where it is needed as soon as it becomes available

Speculative execution

- Fast processors a form of superscalar execution called speculative execution
- This means: they execute instructions that *might* be needed, before it's know if these instructions *should* be executed
- If it turns out that the instructions *should not* be executed, their effects are rolled back (the effects are undone)

R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units, IBM Journal, January 1967.

Speculation

```
1.    load    R1,...
2.    load    R2,...
3.    cmp     R1,R2
4.    jumplt  OverThere[R0]
5.    add     ...
6.    sub     ...
7.    mul     ...
```

- Instructions 1 and 2 require memory access; if the data is not in the cache, these will take a couple hundred clock cycles to execute
- Therefore the processor doesn't know whether the jump will take place
- To gain speed, the processor predicts (i.e. guesses)
- If it guesses the jump won't occur, it goes ahead and executes instructions 5, 6, 7, . . .

Commit or rollback

- The processor can run far ahead of the point where it *knows* the data (up to 150–200 instructions ahead)
- Eventually the data needed for the cmp arrives, and the processor now knows whether its guess was right (whether the jump actually occurred)
    - If guess was right, the processor commits the work it has done speculatively (it marks it as valid)
    - If the guess was wrong, the processor unrolls the work by abandoning the speculative register file and reverting to the saved register file

Spectre and Meltdown

- They are similar and use the same basic ideas
- Both use cache speed to find out a secret byte from the victim
- They differ in how they trick the processor into executing dangerous code speculatively
  - Spectre uses branch prediction, and requires that a known code snippet (a “gadget”) occurs in the victim
  - Meltdown exploits the way memory mapping is done on Intel processors, and uses traps
- The details are different

Effect of cache

- Each address *a* in user space corresponds to some cache line
- If *a* has been used recently, the data will be in the cache (access in 1 clock cycle)
- Otherwise, the data will not be in cache (access in 100 clock cycles)
- When you access data not in cache (cache miss), the data is loaded (which is slow) but also placed in the cache (so next access to it will be fast)
- Question: Has a process accessed address *a* recently?
    - Let's perform a load from address *a* and measure its speed
    - If fast, then the process recently used *a*
    - If slow, then the process hasn't looked at *a* for a while
- The Flush–Reload algorithm does this, and it's used in Spectre and Meltdown

Handling input

- If a user process tries to read memory belonging to another user, there will be a segmentation fault
- That terminates the user
- But what if a user reads in some input data (which is always suspect)?
  - Consider the assessed exercise: each input record has a code indicating whether to insert, delete, etc
  - The code should be between 0 and 4
  - What if somebody supplies a code of 25,000, or -1300?
  - The program checks that the code is valid and if not it ignores the input record
  - If the program didn't do that, it would generate a segmentation fault and be terminated forcibly by the operating system
  - Thus the error checking by the user program isn't necessary for *security*; it's just necessary to prevent the program from getting terminated if it receives bad input

Covert channels

- Suppose two processes, A and B, are running in the same computer
- A wants to get information from B but memory protection prevents it
    - Example: a cloud computer, e.g. in Amazon cloud services
    - It's running separate software for different customers
    - One customer is legitimate: your bank
    - The other customer wants to steal your information (but they pretend to Amazon to be just an ordinarty legitimate business)
- A may be able to get information *indirectly*
- How fast is the system? If B is idle, A will be faster but if B is doing a lot, A will be slower
- How much power is being consumed?
- What about stray radio emissions?

The Spectre attack

- Victim has array1 with array1_size elements, and array2 with 64k elements (2^{16})
- Attacker wants to read the byte at address *a* in the victim's memory
- Victim reads input *x* (e.g. a code indicating what operation to perform) and executes this:

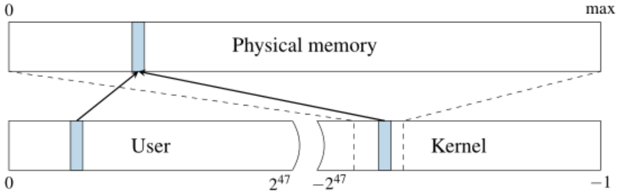
```
if x < array1_size
  then y := array2 [array1[x] * 256]
```

- Attacker supplies the input value $x = a - \&array1$
- Victim executes y calculation speculatively, and this probes a cache line for array2 that depends on the value of mem[a]
- Attacker measures speed of each cache line and deduces value of mem[a]

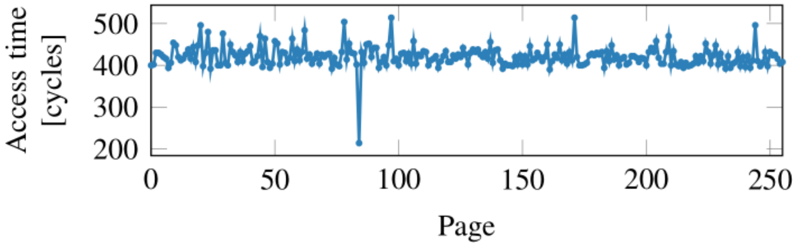
The Meltdown attack

- Exploits the way operating systems handle memory on Intel processors
- The virtual memory hardware makes it possible, safe, and convenient to keep the kernel space shared with the user address space
- This is safe (or it was safe, before Meltdown) because each page has independent security
- Meltdown uses a separate process (the attacker) to perform invalid memory accesses into kernel space, which is mapped onto the victim's address space
- Normally the attacker can't read that data (memory protection!) but it can read the data speculatively, and then read out the value by using cache speed measurements

Meltdown: Mapping user space to kernel space



Measuring the speed of access



Deducing secret information

- Get the victim program to contain instructions which
  - Would be dangerous to execute (access private data)
  - Are not executed (e.g. a conditional branch avoids them)
  - Are present in instruction stream after the conditional branch or trap
- The processor executes these instructions speculatively
  - Later it discovers these instructions should not be executed and they are "rolled back"
  - But when they were tentatively executed, they affected the cache
- Measure the speed of legitimate memory accesses
  - Deduce what has been brought into the cache
  - Deduce a byte of secret data

Summary: Speculative execution

- Some instructions are fast, some are slow
- The processor executes instructions out of order so it can do many operations at the same time
- Some of these instructions may turn out not to be needed: they are *rolled back*, while the necessary instructions are *committed*
- A program may contain instructions that
    - would violate security if executed
    - but aren't executed because of checking a condition

Summary: Memory

- Memory is divided into sections (“segments”, and segments are divided into “pages”)
- Each process has access only to its own segments
- The architecture interrupts a program if it tries to read or write to a segment it doesn’t have access to

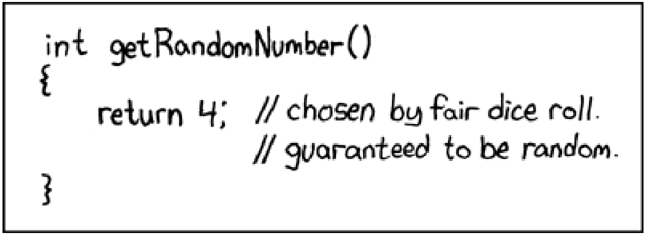
Summary: Spectre and Meltdown

- A side channel (or “covert channel”) is a means of transmitting data that circumvents the architecture’s security mechanisms
- Spectre and Meltdown trick the program into executing instructions speculatively
- These instructions shouldn’t be executed, but they access sensitive data
- The instructions are rolled back: they don’t actually execute “for real”
- But when they are executed speculatively, they bring data into the cache
- The attacker can find out which data is in cache by performing timing experiments
- From this, the attacker can deduce contents of victim’s memory

Summary: Conclusions

- For the casual computer user
  - Make sure your system is updated
  - Avoid allowing attackers to run code on your system: use firewalls, anti-virus and anti-malware tools
- For the computing professional
  - You need a solid grounding in the fundamentals in order to work on state of the art issues in computing

Random number



```
int getRandomNumber()
{
    return 4;  // chosen by fair dice roll.
               // guaranteed to be random.
}
```

https://xkcd.com/221/