Understanding Meltdown and Spectre: Some Processor Design Background

Bob Rogers

Presentation for zED Talks

January 17, 2018

Session Abstract

This short presentation provides some insights into the CPU design aspects that enable the Meltdown and Spectre threats. The purpose of this presentation is so that one can better understand what is being said about the threats themselves, the measures being shipped to help protect against these threats, and the degradation those measures may cause.

Modern processors actually execute programs quite differently than one is lead to believe when learning computer programming. In fact, many engineering tricks are used to allow the processor to gain speed while maintaining the illusion of executing instructions one at a time in the order seen in the program listing. Two potential attacks against Intel processors running Window or Linux have been uncovered that exploit these engineering tricks. The attacks have been named Meltdown and Spectre. They exploit CPU design techniques called out-of-order execution, branch prediction, speculative execution, and memory caching. Defenses are being created to neutralize the threats, but they come with significant performance penalty - potentially over 30% for some workloads.

Caveat

I am not an expert on Intel architecture or microarchitecture, but I know basic processor design principles. I am not an expert on the Windows or Linux operating systems, but I know basic operating system design principles.

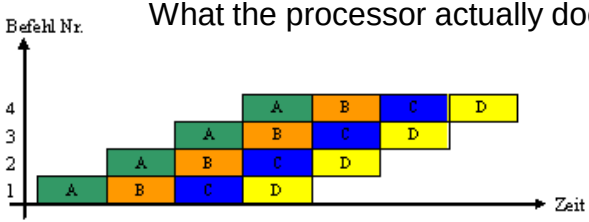
Therefore, the details of any example in this presentation may not be literally true for Intel or Windows or Linux, but the principles are correct, based on what I have read about the problems.

Some Processor Design Information

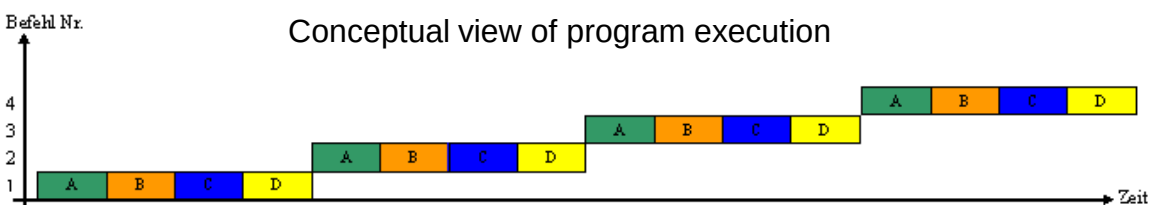
Conceptual vs Pipeline Execution

We tend to think that the processor executes one instruction after the other in the order seen in a program listing. This is an illusion. The processor actually processes different parts of several instructions at any given time.

What the processor actually does - pipeline execution



Conceptual view of program execution



* Chart is compliments of some German speaker who donated it to Google.

Pipeline Hazards (stalls)

Besides waiting for data, the pipelines can stall because of inter-instruction dependencies.

- Address generation interlock (AGI)
- Operand store-compare
- Register load-use
- Register use-reload

Another case for stalls is waiting for targets of conditional or indirect branches to be resolved.

To optimize machine cycles, processors can execute instructions out of order to fill the “bubbles” caused by these dependencies.

Out-of-Order Execution

- Processors that execute instructions Out-of-Order use detailed bookkeeping and some tricks to appear to execute the program as it was written.
- The processor maintains a table to track the status of all in-flight instructions.
- The results of an instruction cannot be stored until all older instructions have previously been retired – i.e. their results stored and made visible.

Speculative Execution

- Processors that execute instructions Out-of-Order typically will not allow an unresolved branch to stall the pipelines.
- Instead, they will try to predict where the branch will go and then continue OOO execution there speculatively.
- If the prediction turns out to be incorrect, any results of instruction execution along the incorrect path must be discarded. The instructions must, in effect, be forgotten.
- Exceptions, of course, cause unpredicted branches.
- But – and this is the important point – side-effects of the “forgotten” instructions might persist in the microarchitecture state of the processor.

Processor Cache

- Processors reduce memory access latency by maintaining recently accessed chunks of memory in structures called caches. Modern processors have multiple levels of caches.
- The data in cache is store in cells called cache lines that are typically 64, 128 or 256 bytes in size.
- Data can be accessed much more quickly if it is in cache than if it must be brought in from main memory.
- A program can tell if data is being accessed from cache by simply timing some code that accesses it.

Branches and Branch Prediction

In order to keep the pipeline supplied with instructions, the processor must "guess" the next instruction.

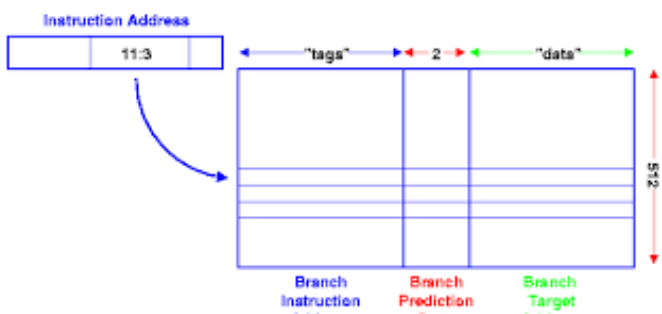
- The processor maintains a Branch Target Buffer (BTB) of taken branches.
- Branches not taken typically do not take up space in the BTB.
- Some branches are predicted statically, e.g. unconditional branches
- A mispredicted branch causes a partial flush of the pipeline. The instruction which were executed speculatively must, in effect, be “forgotten”

Branch Prediction using Branch Target Buffer (BTB)

The processor keeps track of taken branches in the BTB. The entries are indexed by a subset of the address of a suspected branch instruction.

- It is a suspected branch because the lookup is done before the instruction at the address has been decoded. It may not be a branch at all.
- For each branch represented in the BTB there is a predicted direction – taken or not taken – and a predicted target if the branch is taken.

After the branch is resolved, the BTB is updated and if the prediction was wrong, the results of speculatively executed instructions are discarded.



* Chart is compliments of someone at University of Auckland who donated it to Google.

Meltdown

What is Meltdown?

- Meltdown is an attack that uses memory caching as a side channel.
- Basically, it coaxes the processor into making an out-of-order speculative access to a “protected” byte and then to speculatively use the value to access memory that the attacker has authority to access.
- After the processor has flushed the results of the speculative execution, the data which was accessed speculatively is still in cache – including the attacker’s data that was loaded based on the value of the “protected” byte.

Schematic of Meltdown

Here's an example taken from *Meltdown* by Moritz Lipp, et al.

```
; rcx = kernel address
; rbx = probe array
retry:
mov al, byte [rcx]                  {load a byte at location x}
shl rax, 0xc                        {multiply by 4096}
jz retry
mov rbx, quadword [rbx + rax]       {load using 4096 times the value at x}
```

"The core instruction sequence of Meltdown. An inaccessible kernel address is moved to a register, raising an exception. The subsequent instructions are already executed out of order before the exception is raised, leaking the content of the kernel address through the indirect memory access"

By timing accesses to the probe array elements, it can be determined if the data is in cache or not. If properly initialized, this can reveal the value of the "protected" byte.

What does Meltdown Depend On?

Aside from speculative execution, Meltdown depends on other conditions:

- The “protected” data needs to be architecturally accessible to the attacker. This is true when the Linux kernel is mapped into the high half of the user space.
- The processor must delay raising an exception when an unauthorized access is made speculatively.
- The attacker must have a way to flush all parts of the probe array from cache. Intel has an instruction (clflush) to do this, or it can be done brute force if the cache design is known.
- On some Linux systems, the exposure is magnified by the fact that all of real memory is mapped into the kernel half of the address space. This allows access to the data of other processes that is currently in real memory.

A variant that doesn't raise an exception

- The processor can be coaxed into speculative execution without generating an exception.
- Branch prediction can be used to cause the speculative execution of instruction.
  - Train the processor to expect a branch to not be taken
  - Change values so that the branch will be taken and that the index is out of bounds.
  - This time the fall-through path will access "protected" data
  - When the processor resolves the branch, it throws away the results of the speculative executed code, but side-effects remain in cache.

What does the KAISER fix do?

- Since Meltdown depends on the attacker having addressability to the victim's data, one fix is to no longer make the victim's memory addressable by the attacker.
- KAISER is a fix to Linux so that the kernel is no longer mapped into the user space while the user is running.
- The kernel was mapped into the user's address space in the first place to reduce the overhead of system calls.
- With KAISER, the overhead of entering and leaving the kernel is increased and, depending on workload, can reduce throughput by as much as 30%* (or maybe even more).

* See for example: https://blog.appoptics.com/visualizing-meltdown-aws/

Spectre

What is Spectre?

Spectre is superficially similar to Meltdown in that it also exploits, out-of-order speculative execution, branch prediction and data caching. But differs in significant ways.

- Spectre does not depend on the attacker's code having addressability to the victim's data
- It does depend on finding code sequences, called "gadgets", in the victim's code that can be used to make speculative accesses to the victim's data.
- It depends on training branch prediction to mispredict a branch target to branch to the gadget speculatively.
- JavaScript programs can be written to attack the browser.

Schematic of Spectre

Here's an example gadget taken from *Spectre Attacks: Exploiting Speculative Execution* by Paul Kocher, et al.

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

The value x, in this case, is controlled by the attacker. By setting it to values outside the bounds of array1, the attacker can cause the processor to speculatively load a "protected" byte and use it to speculatively access a location in array2 that depends on the value of the "protected" byte.

Two branches need to be subverted for this it work.

By timing accesses to array2 elements, it can be determined if the data is in cache or not. If properly initialized, this can reveal the value of the "protected" byte.

How to get the gadget executed

- If there is an indirect branch that the attacker can cause the victim to execute, it can “teach” the branch prediction logic to think that the gadget is the likely target of that branch.
- Since the BTB is indexed by a subset of the bits of the address of the actual branch, the attacker can fool the processor into making a prediction that is favorable to the attacker.
- It is not necessary for the attacker to actually execute the victim’s branch. It is sufficient to exercise branches within its own space that are synonyms of the victimized indirect branches.

How to Thwart a Spectre Attack

- Unlike Meltdown, Spectre does not depend on any aspects of underlying operating system. Therefore, KAISER is ineffective against Spectre.
- To defend against Spectre, each potential victim must implement its own protection and get it shipped to users.
  - It has been recommended to insert “serializing instructions” to inhibit speculative execution in code that might be used as a gadget.
  - Another mitigation is to force the index (x) to be no larger than the lowest power of 2 greater than the size of the array (ANDing off high-order bits). This limits the scope of what can be accessed speculatively.
- Browser developers have suggested that reducing the granularity of timer stamps would thwart side-channel attacks.
  - In Firefox, the precision of performance.now() has been reduced from 5μs to 20μs.

Summary

Summary

- We covered some processor design topics and related them to how they enable the Meltdown and Spectre attacks.
- Hopefully, this short zED Talk will enable you to read and understand the source articles and commentary on the subject. And, most importantly, to see that these threats are real.
- This is all I know so I can only accept questions on what I meant by something I put on one of the charts. I will not speculate (pun intended).
- Thanks to Peter Enrico and Scott Chapman of Enterprise Performance Strategies (EPS) for providing reference materials and insight into these issues.

Thank you for your attention.

Partial Bibiography

- Lipp, Moritz et al., *Meltdown*
  - https://arxiv.org/pdf/1801.01207.pdf
- Kocher, Paul et al., *Spectre Attacks: Exploiting Speculative Execution*
  - https://arxiv.org/pdf/1801.01203.pdf
- Heffner, Mike, *Visualizing Meltdown on AWS* (a blog post)
  - https://blog.appoptics.com/visualizing-meltdown-aws/
- Horn, Jan (Project Zero), *Reading privileged memory with a side-channel*
  - https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html
- Cheng, Roger, *Lowering JavaScript Timer Resolution Thwarts Meltdown and Spectre (a blog post)*
  - https://hackaday.com/2018/01/06/lowering-javascript-timer-resolution-thwarts-meltdown-and-spectre/
- XSA-254 - Xen Security Advisory
  - http://xenbits.xen.org/xsa/advisory-254.html