Meltdown and Spectre: Complexity and the death of security

Dr. Michael S. Kirkpatrick

January 24, 2018
Meltdown and Spectre: Wait, my computer does what?

Dr. Michael S. Kirkpatrick
January 24, 2018
Meltdown and Spectre:
Whoever thought that was a good idea?

Dr. Michael S. Kirkpatrick
January 24, 2018
Meltdown and Spectre:
I give up. Can I just retire now?

Dr. Michael S. Kirkpatrick
January 24, 2018
No one alive understands how computers behave.
Meltdown and Spectre

Kernel co-location
Race conditions
Page tables

Cache mapping
Branch prediction
Out-of-order execution
High-resolution timers
Physical memory map

Process forks
Cache hierarchy
Pipelined CPU design
Speculative execution
| a | Twitter | Facebook | Amazon | YouTube | Google | Chrome | Office | Firefox | Skype | Windows | Linux | Android | macOS | Xen | Docker | CPU | RAM |

**JMU Computer Science Systems Core**
Dr. Michael S. Kirkpatrick
You Shall Not Pass!

Read kernel location deadbeef
Memory hierarchy and cache

1: Access allowed?
2: OK, here’s your data

1: Access allowed?
2: OK, here’s your data

1: Access allowed?
2: OK, here’s your data

L1 data cache
L1 inst. cache
L2 unified cache
L3 unified cache (shared by all cores)
Main Memory
If orange 5 doesn’t depend on 3 & 4, why wait?
x86 Pipelining

On the Intel architecture, the pipeline consists of the frontend, the execution engine (back-end) and the memory subsystem. Instructions are fetched by the frontend, decoded, and then directly begin the execution of the instruction. Execution units are continuously sent to the execution engine by individual execution units. The pipelines of the execution units are connected via a common data bus (CDB). If an operand is not available, the reservation unit can listen on the CDB until it is available.

Since CPUs usually do not run linear instruction sequences, Tomasulo [33] introduced the concept of speculative execution to refer to an instruction sequence following a branch, and use the term out-of-order execution to refer to speculative execution in a more restricted meaning, where it refers to an instruction sequence following a branch, and use the term out-of-order execution to refer to speculative execution in a more restricted meaning, where it refers to an instruction sequence following a branch. Tomasulo [33] developed an algorithm [33] for instruction scheduling, which is the basis for most out-of-order execution processors.

In 1967, Tomasulo [33] developed an algorithm [33] for instruction scheduling, which is the basis for most out-of-order execution processors.

In 1967, Tomasulo [33] developed an algorithm [33] for instruction scheduling, which is the basis for most out-of-order execution processors.

In 1967, Tomasulo [33] developed an algorithm [33] for instruction scheduling, which is the basis for most out-of-order execution processors.
What we know so far…

- You’re not supposed to access kernel
- L1 cache timing is wrong
- x86 pipelining is crazy
- Macroarchitecture != microarchitecture
  - “First” != first
  - Invisible side effects are visible
Cache-based side channels

Guess what y is!

\[ x = \text{array}[y]; \]

for (i = 0; i < 8; i++) {
    start_timer();
    y = \text{array}[i];
    stop_timer();
}
Meltdown

Step 1: Reading the secret.

```assembly
1 ; rcx = kernel address
2 ; rbx = probe array
3 ret
4 mov rbx, qword [rcx]
5 shl
6 jz
7 mov rbx, qword [rbx + rax]
```

- **Read a byte of the kernel**
- **Multiply byte by 4096**
- **Use value to hit cache line**

8: Maybe I should check if step 4 was valid…
Meltdown

O-ho! Second byte is 1.

fork()

ATTACK!

fork()
Meltdown

Process memory contains...
the kernel, which contains...
physical memory, which contains...
the memory contents of every process.

![Diagram of memory spaces](image-url)

- **User** region
- **Kernel** region
- **Physical memory** region
x86 Pipelining

The x86 architecture uses a pipelined execution engine, allowing for the execution of multiple instructions concurrently. The pipeline consists of several stages:

1. **Frontend**
   - Instruction Fetch & PreDecode: Retrieves instructions from memory and prepares them for execution.
   - Instruction Queue: Stores instructions waiting for execution.
   - 4-Way Decode: Decodes instructions into micro-operations (µOPs).
   - µOP Cache: Caches µOPs for efficient retrieval.

2. **Execution Engine**
   - Scheduler: Schedules µOPs for execution.
   - Execution Units: Process µOPs, which may include ALU, AES, FMA, and memory operations.
   - Reorder Buffer: Stores µOPs and manages their order of execution.

3. **Memory Subsystem**
   - Load Buffer: Handles load data requests.
   - Store Buffer: Handles store data requests.
   - L1 Data Cache: Quickly retrieves data from recently used memory locations.
   - L2 Cache: Stores data for faster access than main memory.
   - DTLB, STLB: Translate virtual memory addresses to physical addresses.

Branch predictors try to determine which instruction will be executed next. Branch prediction techniques are used to improve the efficiency of the pipeline. If the prediction is incorrect, the reorder buffer allows the system to obtain an educated guess of which instruction will be executed next. Branch predictors analyze the instruction stream to predict branch outcomes.

Various approaches to predict branches exist, with one-level branch prediction using a 1-bit or 2-bit counter to record the last outcome of the branch. This helps in determining the path for instruction execution.

In summary, the x86 architecture efficiently manages instruction flow through its pipeline, using branch prediction to optimize performance and reduce execution latency.
Speculative execution

```c
if (x < array_length)
  y = array[x];
```

- x is 2 → y becomes c
- x is 5 → y becomes f
- x is 1 → y becomes b
- x is 327 → y becomes
Spectre variant 1

Choose $x = \&\text{target} - \&\text{array1}$

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

From a security perspective, speculative execution involves executing a program in possibly incorrect ways. As the effects of these instructions are not reverted when the processor realizes that the speculative execution is incorrect, changes to other microarchitectural parts of the CPU (such as cache contents) can survive the processor discards the (incorrect) speculative execution and needs to be concretely instantiated with a way to induce erroneous speculative execution as well as data exfiltration via microarchitectural covert channels. While the changes to the nominal CPU state are eventually reverted, changes to other microarchitectural state (e.g., branch prediction) can remain.

In this paper, we present a practical Spectre attack targeting modern microarchitectures that exploit microarchitectural state. We analyze the frequency and accuracy of the branch predictor and the speculative execution state of the processor. We then search for instruction sequences that, when speculatively executed, can induce errors in the branch predictor and other microarchitectural state.

Exploiting Speculative Execution.

Choosing the right speculative instruction sequence, the attacker is able to leak information across security domains. The leakage occurs by reverting the register state back to the stored checkpoint (for dynamic branches) or by checking if the guess was correct (for unconditional branches, the attacker needs the branch predictor to mispredict the direction of the branch, then the processor must speculatively execute code that would not be expected to be executed).

At a high level, Spectre attacks trick the CPU into speculatively and erroneously executing code with a value of $x$ that is within a legal range, ensuring that the access to controlled data is not reverted when the processor realizes that the speculative execution was incorrect.

1.1 Our Results

The above description of Spectre attacks is general, and needs to be concretely instantiated with a way of inducing erroneous speculative execution.

1.2 Our Techniques

Attacks using Native Code.

Attacks using JavaScript.

In addition to violating process isolation boundaries by combining speculative execution with transient instructions, Spectre attacks can also be used to violate browser sandboxing, by executing a program that contains secret data within its memory. By carefully choosing the value of $x$, the adversary to find a byte of the victim's memory. By repeating with different values of $x$, the attacker retrieves the victim's information over the covert channel. While many choices are possible for the covert channel component, it is possible to use a cache-based covert channel using Flush+Reload [37] or to induce erroneous speculative execution as well as microarchitectural concurrency.

We created a simple victim program that successfully reads data from the entire victim's memory address space, including the secret code with valid inputs, training the branch predictor to expect that the code with a value of $x$ will be true. The attacker then invokes the program execution. As the effects of these instructions are not reverted when the processor realizes that the speculative execution is incorrect, changes to other microarchitectural state (such as cache contents) can survive the processor discards the (incorrect) speculative execution and needs to be concretely instantiated with a way to induce erroneous speculative execution as well as data exfiltration via microarchitectural covert channels.

...
Spectre variant 2 gadgets

https://34c3.cyber-itl.org/slides.pdf
<table>
<thead>
<tr>
<th>Short-term fix</th>
<th>Meltdown</th>
<th>Spectre</th>
</tr>
</thead>
<tbody>
<tr>
<td>KAISER/PTI</td>
<td>Microcode patch (IBPB/STIBP/IBRS)</td>
<td>All BP</td>
</tr>
<tr>
<td></td>
<td>OS update</td>
<td>Hyperthreading</td>
</tr>
<tr>
<td></td>
<td>Recompile binaries</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Change compiler</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Browser hardening</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Long-term fix</th>
<th>Meltdown</th>
<th>Spectre</th>
</tr>
</thead>
<tbody>
<tr>
<td>Split address space</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Replace hardware</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Change of CPL</td>
<td></td>
</tr>
</tbody>
</table>
# CVE-2018-5093: Buffer overflow in WebAssembly during Memory/Table resizing

**REPORTER** OSS-Fuzz

**IMPACT** HIGH

**Description**

A heap buffer overflow vulnerability may occur in WebAssembly during Memory/Table resizing, resulting in a potentially exploitable crash.

**References**

- [Bug 1415291](#)

# CVE-2018-5094: Buffer overflow in WebAssembly with garbage collection on uninitialized memory

**REPORTER** OSS-Fuzz

**IMPACT** HIGH
And so it begins…

Skyfall and Solace

More vulnerabilities in modern computers.

Following the recent release of the Meltdown and Spectre vulnerabilities, CVE-2017-5175, CVE-2017-5753 and CVE-2017-5754, there has been considerable speculation as to whether all the issues described can be fully mitigated.

Skyfall and Solace are two speculative attacks based on the work highlighted by Meltdown and Spectre.

Full details are still under embargo and will be published soon when chip manufacturers and Operating System vendors have prepared patches.

Watch this space…
A return to the past...
THE MELTDOWN AND SPECTRE EXPLOITS USE “SPECULATIVE EXECUTION.” WHAT’S THAT?
YOU KNOW THE TROLLEY PROBLEM? WELL, FOR A WHILE NOW, cpus HAVE BASICALLY BEEN SENDING TROLLEYS DOWN BOTH PATHS, QUANTUM-STYLE, WHILE AWAITING YOUR CHOICE. THEN THE UNNEEDED “PHANTOM” TROLLEY DISAPPEARS.

THE PHANTOM TROLLEY ISN’T SUPPOSED TO TOUCH ANYONE. BUT IT TURNS OUT YOU CAN STILL USE IT TO DO STUFF. AND IT CAN DRIVE THROUGH WALLS.

THAT SOUNDS BAD. HONESTLY, I’VE BEEN ASSUMING WE WERE DOOMED EVER SINCE I LEARNED ABOUT ROUHAMMER.

WHAT’S THAT?
IF YOU TOGGLE A ROW OF MEMORY CELLS ON AND OFF REALLY FAST, YOU CAN USE ELECTRICAL INTERFERENCE TO FLIP NEARBY BITS AND— DO WE JUST SUCK AT...COMPUTERS?

YUP. ESPECIALLY SHARED ONES.

SO YOU'RE SAYING THE CLOUD IS FULL OF PHANTOM TROLLEYS ARMED WITH HAMMERS.

...YES. THAT IS EXACTLY RIGHT.

OKAY. I’LL, UH... INSTALL UPDATES?

GOOD IDEA.