Computer Security Activity 13

Title: Exploiting Speculative Execution: An Analysis of Spectre Variant 1 (CVE-2017-5753)

Team Members: Muhammad Anis Imran 29017 , Adeena Arif 29283

Date: 27/11/2025

Subject: Computer Security

1. Abstract

Modern processor design prioritizes execution speed through complex optimization techniques, most notably Speculative Execution. This project analyzes **Spectre Variant 1 (Bounds Check Bypass)**, a critical vulnerability that exploits the Branch Prediction Unit (BPU) to force a processor into executing instruction paths that should be architecturally forbidden. By utilizing a **Flush+Reload** cache side-channel attack, we demonstrate the ability to leak protected memory from a "victim" process to an unprivileged attacker. This report details the microarchitectural root cause, provides an annotated analysis of the Proof-of-Concept (PoC) code developed in C, and discusses the challenges of reproducing such timing attacks in virtualized environments (WSL).

2. Technical Background

To understand Spectre, one must understand the disparity between **Architectural State** (what the programmer intends) and **Microarchitectural State** (what the hardware actually does).

2.1 Speculative Execution and Branch Prediction

CPUs operate significantly faster than memory. To prevent the processor from stalling while waiting for data or decision results (such as an if statement condition), modern CPUs employ **Speculative Execution**.

The **Branch Prediction Unit (BPU)** maintains a history of recent branches. When the CPU encounters a conditional branch (e.g., if (x < size)), the BPU predicts the outcome based on history. If it predicts "True," the CPU begins executing the subsequent instructions immediately, loading data and performing calculations *before* it knows if the prediction was correct.

If the prediction is correct, performance is gained. If the prediction is wrong, the CPU performs a "rollback": it discards the computed values and reverts the architectural state (registers) to the pre-branch point. However, and this is the crux of the vulnerability, **traces of this transient execution remain in the CPU Cache.**

2.2 The Cache Side-Channel

The CPU Cache (L1, L2, L3) stores recently accessed data to reduce latency.

- **Accessing RAM:** ~200+ CPU cycles (Slow).
- **Accessing L1 Cache:** ~4-60 CPU cycles (Fast).

Spectre exploits this timing difference. Even though the CPU rolls back the *values* of a speculative operation, it does not "uncache" the memory that was loaded during the speculation. An attacker can determine what memory was accessed by measuring how long it takes to retrieve specific data.

3. Vulnerability Analysis: Spectre V1

The specific variant analyzed in this project is **CVE-2017-5753**, known as Bounds Check Bypass.

3.1 The Victim Mechanism

The vulnerability requires a specific code pattern, often called a "gadget," within the victim's kernel or application. The gadget must contain:

1. A bounds check (e.g., if (x < array_size)).
2. An array access dependent on that check.
3. A second memory access that uses the first value as an index.

In our simulation, the victim function is defined as:

C

```c
void victim_function(size_t x) {
  if (x < array1_size) {
    temp &= array2[array1[x] * 512];
  }
}
```

Here, array1 is the "public" array, and array2 is the "probe" array. The variable x is the index controlled by the attacker. If x is larger than array1_size, the execution should stop. However, during speculative execution, the CPU ignores the bounds check and executes the line inside.

3.2 The Secret Data

For this experiment, the "protected" information was a string stored in the process memory, located consecutively after array1. **The Secret String:** "The Magic Words are Squeamish Ossifrage."

The objective of the attack is to read this string character-by-character, despite not having a valid index x to access it.

4. Methodology and Code Analysis

Our Proof-of-Concept (PoC) utilizes three phases: **Flush**, **Train**, and **Reload**. Below is a line-by-line technical breakdown of the C code used in our simulation.

4.1 Phase 1: Flushing the Cache

Before we can measure anything, we must ensure the CPU cache is empty. We use the _mm_clflush (Cache Line Flush) intrinsic instruction.

C

```
// Evict the probe array (array2) from the entire cache hierarchy

for (i = 0; i < 256; i++)

  _mm_clflush(&array2[i * 512]);
```

By flushing array2, we ensure that any subsequent access to this array will come from main RAM (slow), *unless* the CPU speculatively loads it during the attack.

4.2 Phase 2: Mistraining the Branch Predictor

We must teach the CPU that the condition if (x < array1_size) is usually **TRUE**.

C

```
// We run this loop 30 times

// 29 times with a valid 'training_x' (Safe)

// 1 time with a 'malicious_x' (The Attack)

x = ((j % 6) - 1) & ~0xFFFF;

x = (x | (x >> 16));

x = training_x ^ (x & (malicious_x ^ training_x));
```

This complex bit-twiddling code is used to avoid using a conditional if statement to select between training_x and malicious_x, which might itself be predicted. When x is malicious (out of bounds), the CPU executes victim_function(x). Because it expects x to be valid, it speculatively computes: index = array1[x] (This loads the SECRET byte!) access = array2[index * 512] (This loads the probe array at the secret index).

The Result: The specific cache line corresponding to the value of the secret byte is loaded into the L1 cache.

4.3 Phase 3: Reload and Measure (Timing Attack)

We now iterate through all 256 possible values of the byte and measure the read time.

C

```
// RDTSCP reads the Time-Stamp Counter

time1 = __rdtscp(&junk);

junk = *addr; // Read from array2

time2 = __rdtscp(&junk) - time1; // Calculate delta


// Check against the Threshold
```

```
if (time2 <= CACHE_HIT_THRESHOLD)
   score[mix_i]++;
```

If time2 is low (e.g., < 120 cycles), it means array2[i*512] was already in the cache. This confirms that the secret byte value was i.

5. Challenges and Experimental Results

Conducting this attack in a controlled lab environment presented significant challenges, primarily due to the virtualization layer.

5.1 The Impact of Virtualization (WSL)

The experiment was conducted on the **Windows Subsystem for Linux (WSL)**. Unlike a bare-metal Linux installation, WSL shares the CPU scheduler with the host Windows OS.

1. **Noise:** Background Windows processes caused context switches, interrupting the rdtscp timer.
2. **Latency:** The "Cache Hit" time, which is typically <80 cycles on native hardware, was observed to be highly variable (100–150 cycles) in WSL.

5.2 Threshold Tuning & Stabilization

Initially, the standard threshold of **80 cycles** yielded false negatives (outputting ? or garbage data). Through iterative testing, we adjusted the CACHE_HIT_THRESHOLD to **120 cycles**. Furthermore, to stabilize the video demonstration, we implemented a "Retry Logic" mechanism. Instead of printing the result of a single pass, the code was modified to accumulate "scores" over 1,000 iterations per byte. This statistical approach allowed us to filter out the virtualization noise and successfully reconstruct the sentence: "The Magic Words are Squeamish Ossifrage."

6. Mitigations

Since the discovery of Spectre, vendors have released several mitigations.

6.1 Software Mitigations

- **LFENCE (Load Fence):** Developers can insert serializing instructions (LFENCE) before bounds checks. This forces the CPU to complete all prior instructions (including the bounds check) before proceeding, effectively killing speculative execution at that point.
- **Retpoline:** A compiler-level mitigation that replaces indirect branches with return instructions to prevent the Branch Target Buffer (BTB) from being poisoned.

6.2 Hardware Mitigations

- **Intel Hardware Shield:** Newer CPUs (Coffee Lake Refresh and later) include hardware-level fixes that restrict speculative access to certain memory areas.
- **Increased Isolation:** Techniques like Kernel Page Table Isolation (KPTI) separate user-space and kernel-space memory more strictly, though Spectre V1 (user-space to user-space) remains difficult to patch fully without performance penalties.

7. Conclusion

This project demonstrated that hardware optimization features, specifically Speculative Execution, introduce fundamental security flaws that cannot be fixed by traditional software patches alone. By successfully executing a Spectre V1 attack, we proved that it is possible to bypass logical bounds checks and leak protected memory via side-channels. While virtualization adds complexity and noise to the timing measurements, the fundamental physics of the vulnerability remain valid and exploitable.

8. References

1. Kocher, P., Horn, J., Fogh, A., et al. (2019). "Spectre Attacks: Exploiting Speculative Execution." *40th IEEE Symposium on Security and Privacy (S&P)*.
2. Lipp, M., Schwarz, M., Gruss, D., et al. (2018). "Meltdown: Reading Kernel Memory from User Space." *27th USENIX Security Symposium*.
3. Intel Corporation. (2018). "Speculative Execution Side Channel Mitigations." *Intel Analysis White Paper*.
4. Yarom, Y., & Falkner, K. (2014). "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack." *23rd USENIX Security Symposium*.
5. AMD. (2018). "Software Techniques for Managing Speculation on AMD Processors." *AMD Technical Documentation*.