SpecROP: Speculative Exploitation of ROP Chains

Atri Bhattacharyya *EPFL*
Andrés Sánchez *EPFL*
Esmaeil M. Koruyeh *UC Riverside*
Nael Abu-Ghazaleh *UC Riverside*
Chengyu Song *UC Riverside*
Mathias Payer *EPFL*

Abstract

Speculative execution attacks, such as Spectre, reuse code from the victim's binary to access and leak secret information during speculative execution. Every variant of the attack requires very particular code sequences, necessitating elaborate gadget-search campaigns. Often, victim programs contain few, or even zero, usable gadgets. Consequently, speculative attacks are sometimes demonstrated by injecting usable code sequences into the victim. So far, attacks search for monolithic gadgets, a single sequence of code which performs all the attack steps.

We introduce SpecROP, a novel speculative execution attack technique, inspired by classic code reuse attacks like Return-Oriented Programming to tackle the rarity of code gadgets. The SpecROP attacker uses multiple, small gadgets chained by poisoning multiple control-flow instructions to perform the same computation as a monolithic gadget. A key difference to classic code reuse attacks is that control-flow transfers between gadgets use speculative targets compared to targets in memory or registers.

We categorize SpecROP gadgets into generic classes and demonstrate the abundance of such gadgets in victim libraries. Further, we explore the practicality of influencing multiple control-flow instructions on modern processors, and demonstrate an attack which uses gadget chaining to increase the leakage potential of a Spectre variant, SMoTherSpectre.

1 Introduction

Spectre [1] demonstrated the power of speculative execution attacks by leaking information across various protection boundaries: sandboxes, processes, userspace/kernel, and virtual machines. These attacks reuse code gadgets (short instruction sequences with useful functionality) already present in the victim's code base to access and leak secrets such as cryptographic keys or arbitrary memory. As a form of code-reuse attacks, they require gadgets composed of specific instruction sequences to exist in victim binaries. Often, the length of these sequences, their complexity, or the rarity of their constituent instructions implies that the occurrence of usable gadgets is sparse. A case in point is the gadget required by Spectre. The attack requires a gadget where the attacker controls two registers, and contains two loads, of which the second load must access an address which depends on the value loaded by the first. In fact, the attack on the Linux kernel, which is a massive and diverse code base, relied on its eBPF (extended Berkeley Packet Filter) subsystem to essentially inject the gadget into the kernel.

To address the lack of powerful, monolithic gadgets, we propose the use of speculative gadget sequences. We present SpecROP, an attack based around the idea of using the effects of multiple, small gadgets to effectively perform computation equivalent to much larger, monolithic gadgets. The attack methodology leverages the relative abundance of the smaller gadgets as compared to larger gadgets to provide the required leakage gadgets. SpecROP is similar to existing code-reuse attacks (such as Return-Oriented Programming [2, 3] and Jump-Oriented Programming [4]). In comparison to these code reuse techniques, SpecROP leverages branch poisoning, a common starting step in speculative execution attacks, to effectively stitch the execution of these smaller gadgets. We search for small code sequences which perform common modifications on state (e.g., add, shift registers) and end in a return or indirect jump. During speculation, the CPU consults the branch predictor to decide the jump target, allowing us to redirect execution to the next gadget. We use automated binary analysis to discover the constituent gadgets of a SpecROP chain, using our tool, SpecFication.

This paper makes the following contributions:

- A study of the contexts in which branch targets may be maliciously influenced on modern processors (with hardware and microcode updates against Spectre-like attacks);
- Analysis of gadget chaining practicality, using indirect jump instructions as well as returns;
- A proof-of-concept attack, where we extend the capabilities of an existing speculative execution attack;

- A practical attack on a real target, `libcrypto` from OpenSSL, leaking multiple bits of the plaintext during encryption;
- A binary analysis tool, SpecFication, for discovering gadgets in real-world libraries, and a characterization of the existence of some classes of generic gadgets in commonly-used libraries.

2 Background

SpecROP extends the power and impact of speculative execution attacks [1,5,6] by enabling the combined use of multiple gadgets, similar in spirit as Return-Oriented Programming which enabled complex code-reuse attacks. Here, we provide the necessary background for SpecROP.

2.1 Speculative Execution Attacks

Modern processor design has led to a class of attacks known as Speculative Execution Attacks (SEA). These attacks target mechanisms designed to allow a processor to ameliorate the impact of long latency instructions on performance. Specifically, processors fetch and execute instructions out-of-order and speculate when lacking all the information needed to make decisions. Instructions executed by the processor following a misprediction are incorrect. While processors revert the architecturally visible effects of incorrect actions, microarchitectural side-effects remain. This allows SEA attacks to leak information encoded into cache residency [1,5], or port utilization [6] during the period of incorrect execution. The period starting from the point the processor mispredicts until it realizes its mistake is the *speculation window*.

The microarchitectural structure, and its behavior which encodes information defines a side-channel. Variants of SEA differ in their choice of side-channel and the reason for misspeculation. Spectre-v1 [1] exploits misspeculation following a bounds-check prior to an array access. Spectre-v2 exploits misprediction of the target of an indirect call or jump. Both of these variant use a Flush+Reload channel [7]. SMoTherSpectre [6] and NetSpectre [8] use alternate side-channels based in port contention (ports are microarchitectural structures used for scheduling instructions within the processor pipeline) and the power-up status of AVX units instead.

2.2 Microarchitectural Side-channels

Microarchitectural side-channels are data channels on a processor which leverage state stored in microarchitectural structures, such as the cache or branch predictors, to transfer information. As opposed to a covert-channel, the transmitter in a side-channel is not privy to the communication: the transmitter is inadvertently leaking information and is referred to as the victim. The receiver reads the information from the channel, and is referred to as the attacker.

SEA depend on side-channels to leak any secrets accessed during speculative execution, since any architectural channels are erased when the processor detects misspeculation, and rolls back architecturally visible state. Here, we focus on two side-channels: a cache residency based channel (Flush+Reload), and a port contention based channel (SMoTher).

Flush+Reload channel A Flush+Reload channel [7] encodes information in the cache residency of a cache block at a specific address (A). This channel requires the attacker and victim to temporally share a core and its cache. Initially, the attacker primes the channel by flushing the cache block, evicting it from all layers of caches: the block is now uncached. In the second step, the victim executes. During its execution, the victim encodes a secret bit into the channel by conditionally loading from the address A. If the secret is 0, it loads A (thereby caching the block); if the secret is 1, it does not. Finally, the attacker reads the channel by reloading the address A, and timing how long it takes. If the secret was 0, and the victim had already cached the block, the attacker's load is fast, else it is slow. This channel encodes a single bit.

A variant of this channel uses 256 unique address (A_0 through A_{255}) which map to different cache blocks. The victim encodes a secret byte (b) by only loading A_b. In the reload phase, the attacker reloads all addresses and times each load. The load A_b completes faster than the others, thereby leaking a byte.

SMoTher channel A SMoTher channel [6] encodes information in the port utilization of the victim's instructions. This side-channel requires the attacker and victim to share a Simultaneously Multi-Threaded (SMT) core, thereby sharing the ports on the core.

The victim encodes a secret bit by executing a SMoTher gadget, a secret-dependent conditional branch leading to (target and fallthrough) code sequences which utilize different ports. Concurrently, the attacker executes a specific sequence of instructions designed to cause port contention with the target sequence, and times its execution. In the case that the victim is executing the fallthrough sequence, there is no port contention and the attacker completes its own sequence faster. In the other case, when the victim executes the target sequence, port contention causes the attacker's execution to be slower. The attacker, therefore, uses its timing to infer which sequence the victim was executing. Since the victim's secret bit determines which way the conditional branch goes, the attacker is able to leak the secret bit. This channel encodes a single bit.

2.3 Return/Jump Oriented Programming

Return-Oriented Programming (ROP) is a code-reuse technique based on chaining multiple instruction sequences al-

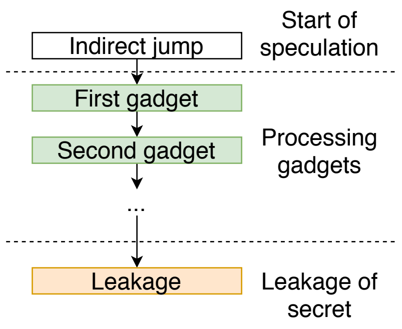


Figure 1: Phases of a Speculative ROP attack.

```
// C: array2[array1[x] * 4096]
mov (rax,rdi,8),rax
shl 0xc,rax
mov (rdx,rax,8),rax
```

(a) Spectre gadget where `rax` points to `array1`, `rdx` points to `array2`, and `rdi` is `x`

```
1  // Gadget 1: Load secret
2  mov  (rdx,rax,1), edx
3  call *(rbx + 0x40)
4  // Gadget 2: Shift secret
5  shl  0x20,rdx
6  mov  eax,eax           // Extraneous
7  or   rdx,rax           // Extraneous
8  ret
9  // Gadget 3: Leak secret
10 mov  (rbx,rdx,8),rsi
```

(b) Equivalent SpecROP chain where `rdx` points to `array1`, `rbx` points to `array2`, and `rax` is `x`. Lines 6 and 7 contain code irrelevant to the gadget chain.

Listing 1: A SpecROP chain from `libc` which can be used in place of the Spectre gadget

ready present in victim code into a gadget capable of performing complex computations. ROP gadgets end in a return instruction, chained through the attacker-compromised return addresses on the stack. This mechanism is used to chain the smaller sequences into complex exploits. In 2007, Shacham [2] demonstrated how gadgets from instruction sequences in `libc` could be used to achieve Turing-complete computation. Jump-Oriented Programming (JOP) [4] is a related technique where indirect jumps are hijacked through memory corruption to similarly compose smaller gadgets into complete attacks.

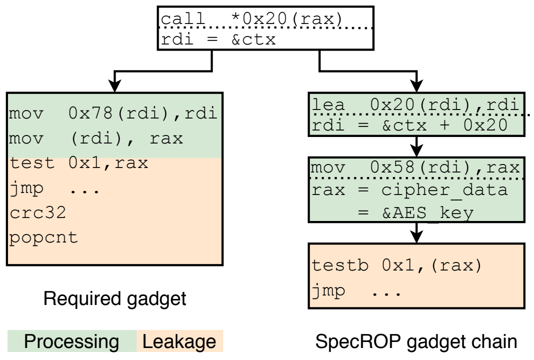
Similar to previously known buffer-overflow attacks, ROP requires the ability to corrupt the stack (or to pivot the stack to an alternate location). While buffer-overflow attacks were commonly used to inject and execute shell code onto the stack, they were effectively eliminated by one mitigation technique: hardware exclusion of writable and executable permissions on pages. ROP attacks bypass this mitigation by design, reusing instructions already existing within the victim's binary and which must necessarily have executable permissions.

Code-reuse attacks depend on the existence of gadgets which perform computation useful to the attacker. The statistical probability of a gadget existing in a binary is dictated by two parameters: the length of the sequence and the probability of occurrence of each instruction in the gadget within the binary. On an architecture with variable-length instructions, such as `x86_64`, short instructions such as `ret` (encoded as one byte) may be even found inside the machine code for longer instructions (such as `mov rax, rbx`). In general, commonly existing ROP gadgets are small sequences of common, sometimes unintended, instructions.

3 Speculative ROP

Speculative Return-Oriented Programming (SpecROP), is an exploit technique that leverages gadget chaining to enhance the capabilities of an SEA attacker. Figure 1 shows the steps of a SpecROP attack. The attack starts at a mispredicted control-flow instruction, an indirect jump/call or return. The attacker poisons the branch predictor on the processor to control the predicted target of the jump. Thereafter, the processor executes instructions along the first gadget in the chain: a *processing gadget*. Subsequently, the attacker manipulates a control-flow instruction at the end of the first gadget to direct execution to the second gadget, and so on through a gadget chain, similar to a ROP attack. The chain of processing gadgets is responsible for performing attacker-controlled computation, and is key to the increased capabilities of a SpecROP attacker (which we discuss later). The final gadget(s) in the chain, a *leakage gadget*, is used to leak secrets. *The key difference to ROP attacks is that SpecROP gadgets are executed and chained speculatively, i.e., the target of the indirect control flow transfer is not read from a memory location but indirectly influenced and "primed" by the attacker.*

SpecROP bypasses the reliance of existing SEA on two conditions which are hard to satisfy, by chaining and leveraging the execution of multiple gadgets. First, in classic SEA the victim's secret needs to be directly accessible, either in a register or in memory referenced to by a register. Using a gadget chain overcomes this requirement, enabling attacks which require multiple operations to access the secret. For example, functions in the generic interface for the OpenSSL library (called EVP) have a pointer to a context structure as the first argument. The context includes a pointer to cipher-specific data (Figure 2b). For AES ciphers, this data is the encryption key. Accessing the key from the context pointer requires pointer arithmetic and two dereferences (see the "re-



```
call   *0x20(rax)
rdi = &ctx
```

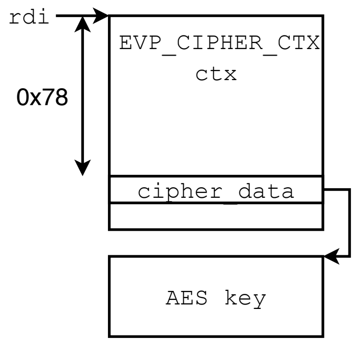
```
mov  0x78(rdi),rdi
mov  (rdi), rax
test 0x1,rax
jmp  ...
crc32
popcnt
```

```
lea   0x20(rdi),rdi
rdi = &ctx + 0x20
```

```
mov   0x58(rdi),rax
rax = cipher_data
    = &AES key
```

```
testb 0x1,(rax)
jmp  ...
```

(a) Control flow during an SEA attack, and for a SpecROP attack. Below the dotted lines, we show the relevant register state.



(b) OpenSSL's Memory layout: The EVP_CIPHER_CTX structure contains a pointer to the AES key at an offset of 0x78.

Figure 2: A SpecROP chain starting with a pointer to the OpenSSL cipher context structure, and capable of leaking the AES key.

quired gadget" in Figure 2a). Such a monolithic gadget which starts with a pointer to the context, and accesses and leaks the key requires a long sequence of instructions. We were unable to find such a gadget within multiple libraries. Existing SEA, therefore, are incapable of exploiting calls to the EVP functions to leak the key. In contrast, we found a chain using three gadgets from `libcrypto` ("SpecROP gadget chain" in Figure 2a) which allows attackers to access and leak the AES key. The full gadgets are listed in Appendix B. This chain requires the attacker to poison an extra indirect jump and an extra return instruction.

Second, classic SEA require a single gadget to both access the secret, and leak it into the microarchitectural channel. The gadget used to leak information in Spectre attacks requires two dependent memory loads with a left-shift of at-least 6 bits (to encode each value in a different, 64-byte cache line) in between (Listing 1a). So far, no natural gadget of this kind has been disclosed publicly, even for large, real-world binaries such as the Linux kernel. In fact, we applied our tool, SpecFication, to search for monolithic, Spectre-like gadgets in commonly-used libraries (listed in Section 5.2), the Linux kernel and its modules, and in QEMU without any results. In contrast, we found an equivalent SpecROP chain in `libc` (Listing 1b). The chain has three gadgets , two of which are dependent loads, with one shift gadget in between. Generally, each gadget in a SpecROP chain is shorter than a monolithic gadget, and there is a higher probability of finding them in the victim's code.

SpecROP offers several benefits compared to traditional ROP and JOP attacks. Unlike ROP attacks, SpecROP does not require any corruption of the victim's memory to chain gadgets. While there is an abundance of memory corruption attacks, they require the attacker to interact *directly* with the victim. In contrast, SpecROP attacks can be performed with only microarchitectural interactions between the attacker and their victim. Further, mitigations for memory safety which protect the stack state do not affect the SpecROP attacker. Unlike JOP attacks, SpecROP does not require a dispatcher gadget whose repeated indirect calls enable chaining of gadgets. Finally, ROP/JOP attacks require that none of the chained gadgets have unwanted side-effects such as exceptions. By executing speculatively, SpecROP bypasses these restrictions. For example, it allows the transient execution of code that dereferences a null pointer as long as the loading of the secret or the leakage gadget do not depend on it.

Limitations SpecROP chains are limited in their length, in terms of the number of instructions and the number of cycles they take to execute. Processors have microarchitectural limits on the number of instructions they can fetch and execute speculatively: the re-order buffer can hold around 200 instructions on modern processors. The entire gadget chain must also complete its execution before the mispredicted branch is resolved. Following a last-level cache miss, the speculation window is up to a few hundred cycles [1].

Unlike JOP, SpecROP gadgets using indirect jumps cannot be repeated. The technique used to poison jumps (Branch Target Injection) implies a unique predicted target for each address, precluding the reuse of these gadgets in more than one place in the same chain.

3.1 Gadgets

Gadgets are the building blocks of a SpecROP attack, and an attacker will require the presence of a variety of gadgets to launch complicated attacks. We devise a system of categorization of gadgets in Section 3.1.1 and show metrics of the availability of each category in Section 4.

Gadgets in a typical SpecROP chain perform one of three important functions: (i) manipulate processor state to access secrets, and (ii) move said secrets into registers (iii) where a final gadget leaks them. At each step, one or more gadgets may be used. Figure 2 shows example gadgets from `libcrypto` which access and leak the key. The first gadget is an arithmetic gadget, where the `lea` instruction effectively adds an offset to a pointer. The second gadget is a data movement gadget, and moves a pointer to the secret into register `rax`. Finally, the last gadget is a leakage gadget (using the SMoTher side-channel) which dereferences a byte of the secret, and encodes the LSB into the channel.

3.1.1 Classification of gadgets

Gadgets in a SpecROP chain can be classified based on the functionality they provide. The main gadget categories are:

- *Arithmetic gadgets* perform simple arithmetic such as additions or subtractions. These might be useful for pointer manipulation, for example, allowing an attacker to craft pointers to secrets.
- *Shift gadgets* shift and rotate values in registers. Gadgets used in the SMoTherSpectre attack leak specific bits in registers (for example, the LSB). These gadgets allow an attacker to move secrets in other bits of the register into those bits which are leakable.
- *Data movement gadgets* move secrets or other data between registers, and between registers and memory. These can be used for moving secrets into a register targeted by the leakage gadget, for example.
- *Leakage gadgets* encode register/memory state into microarchitectural channels, enabling the attacker to later infer and leak information.
- *Multi-use gadgets* perform more than one of the above functions. An example is a `lea` gadget that performs an addition and multiplication.

This classification allows us to locate generic gadgets in existing code bases, rather than specific sequences for particular attack scenarios. In Section 5, we describe SpecFication, a tool for finding useful SpecROP gadgets, under constraints on their length, and starting and ending conditions.

4 Evaluation

We now evaluate the practical aspects of a SpecROP attack. First, we explore the contexts in which gadgets can be chained, and the limits on the number of control-flow instructions which can reliably be poisoned. Towards this goal, we explore both avenues of chaining SpecROP gadgets: indirect jumps and returns. Second, we create a prototype attack in a laboratory setting to explore how chained gadgets enhance a SMoTherSpectre attack, and whether there is any loss in accuracy of leaked information as a result of chaining gadgets. Finally, we describe a SpecROP attack on a real-world target, `libcrypto` from OpenSSL, demonstrating that such attacks are indeed feasible.

Context	6700K	8700	9700	10510U
Cross thread	Y	Y	N	N[1]
Cross process	N	N	N	N
Aliased	Y	Y	Y	Y

[1] even with factory microcode.

Table 1: Contexts in which branch poisoning is feasible

```
// Load unique address Ai
mov     (rdx), rcx
add     0x100, rdx
// Load address to next gadget
mov     (rdi), r8
add     0x8, rdi
// Jump to next gadget
jmpq    r8
```

Listing 2: Gadgets used to determine maximum chaining length using indirect jumps. The loads to A_i mark the execution of this gadget. The final jump chains to the next gadget.

4.1 Gadget chaining

The practicality of a SpecROP attack is strongly linked to the number of gadgets that can be reliably chained: the expressibility of the chain increases with the number of gadgets it contains. Most practical SEA attacks will require at least two (for the Spectre example) to three gadgets (for the OpenSSL example). We describe our experiments for chaining gadgets using indirect jumps and return instructions and evaluate the contexts under which we were able to influence the branch predictor.

Indirect jump poisoning in different contexts We evaluated the ability to poison the branch predictor

- across threads sharing an SMT physical core,
- across processes sharing an SMT physical core, and
- across instructions at different addresses, leveraging aliasing within the Branch Target Buffer (BTB).

We experimented with four generations of Intel's processors with updated microcode: *i*) Skylake i7-6700K, *ii*) Coffee Lake i7-8700, *iii*) Coffee Lake Refresh i7-9700, and *iv*) Comet Lake i7-10510U. From Table 1, we can see that branch poisoning is only possible between an attacker and victim who share code execution within a process, for example, a browser sandbox running JavaScript from multiple websites.

4.1.1 Chaining gadgets through indirect jumps

Let us investigate the chaining of gadgets ending in indirect jumps and calls. Assuming that the targets for the jump are unavailable, the processor will use target predictions from the BTB to speculatively fetch and execute instructions from multiple gadgets.

In our experiment, we execute two threads from the same process running on logical cores sharing a physical core. One of the threads takes the role of an attacker, using a sequence of indirect jumps through gadgets J_0-J_{15} to train the branch predictor. Listing 2 shows the code for the gadgets. The other thread takes the role of the victim, speculatively following the same path through the gadgets. The goal of this experiment is to determine how many gadgets are actually executed by the victim.

The target for the terminal jump of each gadget is loaded from an array in memory (line 5), allowing us to "program" different paths for the attacker and victim. On the attacker, the gadgets are chained in an order designed to appear random to the processor ($J_0 \rightarrow J_2 \rightarrow J_{13} \rightarrow J_4 \rightarrow J_{10} \dots J_{15}$). Architecturally, the victim is programmed to jump directly from J_0 to the end of J_{15}. However, we flush the targets for the victim from the cache, causing it to speculatively execute the same chain as trained by the attacker until the targets are fetched from memory. At this point, the victim state is rolled back.

To determine whether a gadget J_i is executed by the victim, we instrument them with memory accesses (line 2) to unique addresses A_i. Using per-address Flush+Reload channels (see Section 2.2), we can determine which gadgets are executed: if gadget J_i is executed speculatively, the access to A_i is faster in the Reload phase. Knowing which gadgets were actually executed by the victim allows us to infer how many indirect jumps were successfully poisoned.

Figure 3a shows results from running this experiment on two generations of Intel processors, the i7-8700 and the i7-6700K, both with and without the latest microcode updates. On each machine, we use 10 sets of 1,000 runs, plotting the median of the fraction of times the n^{th} gadget was executed by the victim. The limits show the minimum and maximum fractions across the sets. On both processors, up to four gadgets can be chained with more than 50% success. However, the probabilities of chaining five or more gadgets drops drastically, with less than 10% success for reaching six gadgets. A preliminary investigation suggests that TLB misses are not responsible for this trend, as moving the gadgets or the cache-lines for the Flush+Reload channels to 2MB hugepages do not improve it. We also see that microcode updates do not significantly affect the median success rate.

4.1.2 Chaining gadgets through return instructions

We now study the chaining of gadgets terminated by return instructions (`ret`). Modern processors use the Return Stack Buffer (RSB) to predict the target of a `ret` instruction when the return address on the stack is not immediately available. Return addresses are pushed onto the RSB by `call` instructions, and are popped by `ret` instructions. Using unmatched function calls, attackers can push excess values onto the RSB and cause misprediction on later returns.

To show the possibility of chaining gadgets using RSB we consider two experiments. In the first, the attacker and victim execute on the same thread. In the second, the attacker and victim run on different threads within the same process, using a `futex` to interleave their execution on a single core.

Same-thread chaining In this experiment, the attacker uses function calls to push a sequence of addresses onto the RSB. The victim executes subsequently, its return instructions using predictions from the RSB. The addresses on the RSB lead to a sequence of gadgets, each of which accesses memory at an unique address before executing a (poisoned) `ret` instruction. The memory accesses form a Flush+Reload channel to determine which of the gadgets were executed by the victim.

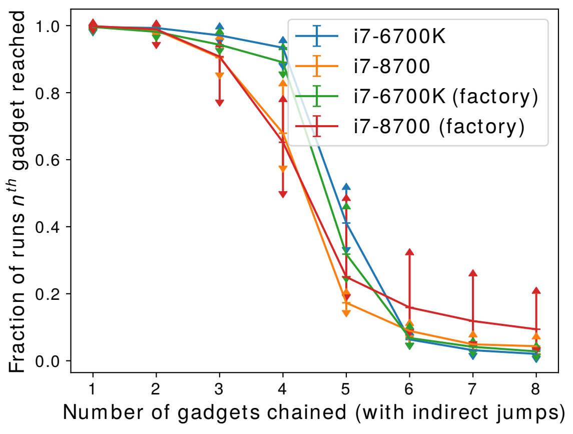
Figure 3b shows results of our experiments on an i7-6700K, an i7-8700 (16 RSB entries each), and a Xeon(R) E5-1620 (24 RSB entries). These experiments demonstrate the chaining of up to five gadgets with a reasonable success rate on the i7-6700K processor and up to two gadgets on the Xeon(R) E5-1620. The success rate drops precipitously to practically zero for more gadgets on all processors.

Cross-thread chaining In this experiment, the attacker poisons the RSB (as in the previous experiment) before using a `futex` to switch to the victim thread. Due to the limited size of the RSB, and its pollution during the context switch (there are multiple function calls within the kernel code) only a few RSB entries remain untouched for the victim. The victim's execution is again similar to the previous experiment. On a i7-6700, the attacker can consistently chain up to two gadgets on the victim. On a Xeon(R) E5-1620, up to three gadgets can be chained using the RSB.

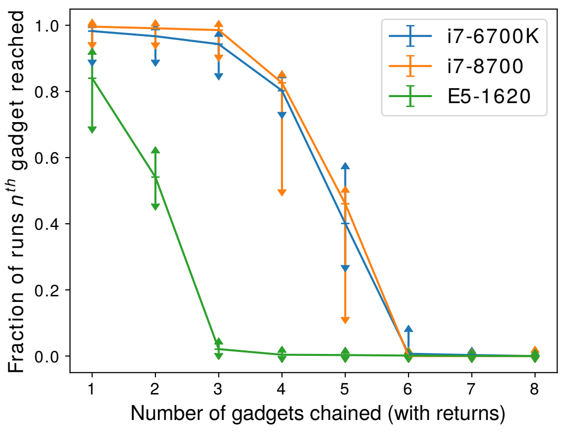
4.2 Proof-of-Concept

We now demonstrate the power of the SpecROP exploit technique. The proof-of-concept (PoC) attack is based on the SMoTherSpectre attack, which leaks specific bits using a side-channel based on port contention. Specifically, the SMoTher leakage gadget targets the least-significant bit (LSB) of the register `rdx`. In our PoC, we use this leakage gadget in different chains, augmenting the leakage capabilities of SMoTherSpectre. While this PoC uses the SpecROP approach to ameliorate one limitation of the SMoTherSpectre attack, we believe that it is indicative of the improvement possible for other attacks.

Our concept attack improves the SMoTherSpectre attack by leaking eight bits of the register, not just the LSB. The attacker achieves this by using shift gadgets for performing right-shift operations on register `rdx` before being redirected to the leakage gadget. Note that the attack can trivially be extended to leak the rest of the register. Figure 4 shows the flow of control on the victim. The attack starts at the basic block ending in



(a) Using indirect jumps



(b) Using ret instructions

Figure 3: The median success rate of chaining gadgets of different lengths on various processors. The limits represent the maximum and minimum rate across runs. Entries marked "factory" represents runs without microcode updates.

an indirect jump (labeled `jmp` in the figure). The actual jump target is the `end` block. By ensuring that the branch target is unavailable, the attacker causes the victim to start speculative execution, following the gadget chain trained by the attacker. The attacker repeats this process several times, leading the victim across the different paths to the leakage gadget. Along each path, the register holding the secret is shifted by different offsets, so that the leakage gadget ultimately leaks different bits of the key.

Let us illustrate the process of leaking different bits. At the jump gadget, suppose register `dl` holds a 8-bit secret $s = \{s_i | i \in [0,7]\}$ which the attacker wishes to leak. When the attacker directly chains the `jmp` to the leakage gadget, the value leaked is s_0, the LSB of `rdx`. Instead, when the attacker chains `jmp`→`shift 1`→`leak`, the register `rdx` holds s_1 in the LSB at the leakage gadget. The leakage gadget now encodes s_1 into the side-channel. Similarly, on repetitions where the attacker chains `jmp`→`shift j`→`leak`, the bit s_j is leaked. Therefore, by progressively redirecting the victim's speculative control flow through different SpecROP chains, the attacker extends the leakage scope of SMoTherSpectre.

The SpecROP attack is not constrained to leaking individual bits. It is the characteristic of the leakage channel used in this POC that a bit is leaked at a time. With a cache-based side-channel which leaks a byte at a time, and *addition gadgets* which manipulate the pointer used to access secrets, a SpecROP attacker can leak an entire byte per iteration.

This proof-of-concept attack models a behavior which is typical of many programs, for example OpenSSL. C++ virtual function calls use indirect jumps, and hold a pointer to the object as the first parameter. For a virtual function call within a loop, the attacker may run different chains on different iterations. If the attacker can access different secrets by chaining different sequences of gadgets, the attacker may progressively

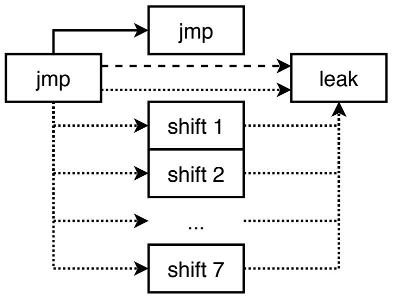


Figure 4: The flow of control during non-speculative execution (solid line), SMoTherSpectre attack (dashed line) and in SpecROP attack (dotted lines).

leak multiple secrets as the victim executes.

In our laboratory proof-of-concept, the attacker and victim run in separate processes. We run the experiment on an i7-6700K processor with microcode updates disabled. This allows cross-process branch poisoning on a shared physical core. With updated microcode, the attacker model changes to that described in Section 4.1.

Our laboratory proof-of-concept was used to leak 1,024 randomly generated bytes. As discussed previously, different chains leak each of the 8 bits per byte, and we treat samples for each bit as a separate channel in the evaluation. For each channel, we collect 1,024 attacker SMoTher timing samples (a measure of port contention with the victim), corresponding to 1,024 randomly generated "secret" bits on the victim. We then separate the attacker timings into two sets, depending on the value of corresponding secret bit on the victim, and plot the probability distribution function (pdf) for each set. When the distributions are clearly distinguishable, it means that the attacker's timing is strongly correlated to the victim's secret and can be classified with a high accuracy. Figure 5

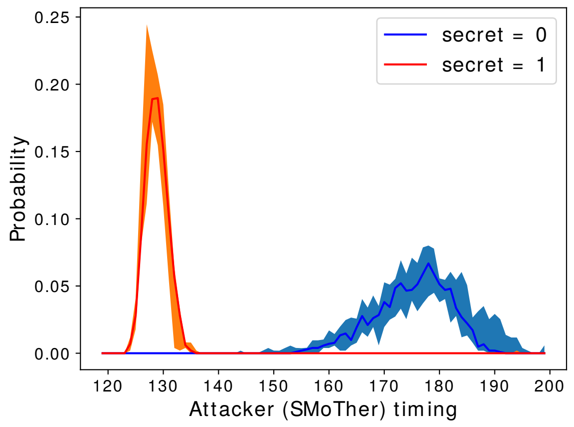


Figure 5: SMoTher timing for multiple gadget chain, separated according to the actual "secret" bit values. We have plotted the ranges of the individual probability distribution functions, separated by the secrets they represent, zero and one.

shows the results of the experiment. Our plot aggregates the results, showing the range of probabilities across the channels. The plot reveals that the attacker timings are similar across all channels, and that there is a clear separation within the distributions based on the actual secret. This means that the attacker can choose a threshold (around 140 cycles in this case), and classify each timing sample as a zero or one with a high accuracy. In fact, we can accurately guess victim secrets across all channels with an accuracy ranging between 99% and 100%.

In this experiment, the channel for bit 0 uses the shortest chain, directly connecting the `jmp` gadget to the leakage gadget. In fact, this chain is identical to the base SMoTherSpectre attack. This chain requires a single poisoned branch, whereas the chains for leaking other bits require two. We have seen (in Section 4.1.1) that longer chains lead to a diminishing probability of reaching the final (leakage) gadget. Therefore, the leakage accuracy for channel 0 is the target for the other channels. In fact, channel 0 leaks with 100% accuracy, and even the worst channel has an accuracy of greater than 99%. This shows that for a gadget chain of length two, SpecROP allows us to augment the leakage scope of SMoTherSpectre without suffering any loss of accuracy.

4.3 OpenSSL attack

In this section, we describe a realistic attack on a target program using OpenSSL's generic EnVeloP (EVP) interface to encrypt/decrypt data. This attack improves upon the base SMoTherSpectre attack on the same target, enabling the attacker to leak an additional bit of the secret by modifying a pointer held in register `rdx` using an arithmetic gadget. By poisoning the BTB, the base SMoTherSpectre attack redirects an indirect call in the `EVP_EncryptUpdate` function directly to a leakage gadget. The register `rdx` holds a pointer to the secret plaintext and this gadget leaks a bit of it from memory. In contrast, the SpecROP attacker first redirects the indirect call to an arithmetic gadget which increments `rdx` by a constant (for eg. `0x40`), and subsequently to a leakage gadget. As a result, the attacker leaks a different bit from the plaintext. With different arithmetic gadgets, this approach can vastly increase the leakage scope of SMoTherSpectre. Figure 6 illustrates this: the leftmost path shows the speculative control-flow in the base attack, and the other paths illustrate the leakage possible through different chains of gadgets. All the processing and leakage gadgets are taken from `glibc`, which is likely to be linked for most C programs, and are listed in full in Appendix B.

In particular, we implemented an attack using the chain which increments `rdx` by `0x40`. The basic procedure of the attack is very similar to SMoTherSpectre: the attacker and victim are threads in the same process running on logical cores on an SMT (hyperthreaded) physical core. Over $100,000$ runs, the victim sets/resets the targeted bit in the plaintext and initiates an AES encryption. Concurrently, the attacker thread passes through a sequence of indirect jumps to poison the BTB. In contrast to basic SMoTherSpectre, this chain poisons more than one indirect branch on the victim. During the consequent period of speculative execution on the victim, the attacker times a sequence of instructions using `rdtsc` timestamps. Due to port contention, the attacker's readings should be correlated to the victim's secret. After the run, we separate the attacker's timings into sets based on the actual value of the victim's secret, and use the Student's t-test to validate that the distributions are actually distinguishable with high confidence (95%).

We ran our attack on an i7-8700 processor. Compared to SMoTherSpectre, the victim in our attack executed the leakage gadget in 33% of the runs instead of 80%. This is explained by our observations in Section 4.1.1: the success rate of reaching the final gadget drops with the length of the chain. Despite the added noise which results from runs where the leakage gadget was not reached, the Student's t-test reports that the distributions are distinguishable with 95% confidence. The test reports a timing difference of $4.41\% \pm 0.06\%$.

5 SpecFication: Gadget Search

To automate and improve the search for complex gadget chains, we develop a tool. The goal of our gadget search tool is two-fold:

1. To characterize the presence of processing gadgets which will enable an attacker to perform useful transformations on program state. Section 3.1.1 lists the target gadget classes.
2. To automate the process of finding gadget chains as per a set of constraints (described below).

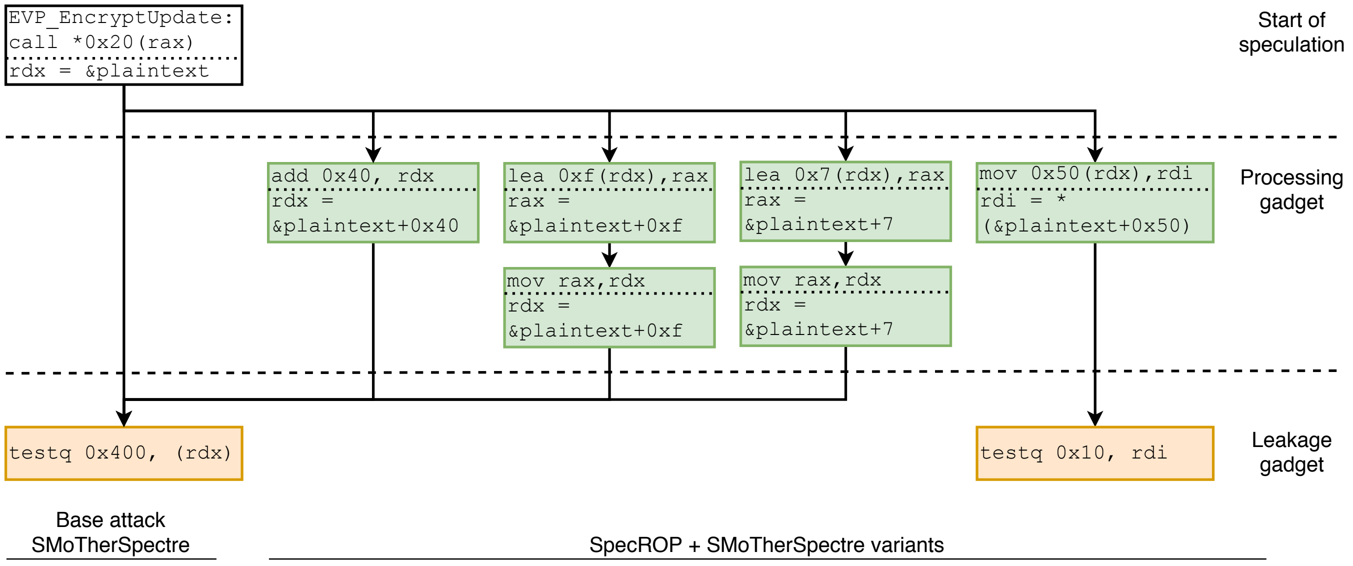


Figure 6: SpecROP allows an attacker to extend the leakage scope of the SMoTherSpectre attack on OpenSSL, targeting plaintext during encryption. The base attacker can leak a bit from byte 1 of the plaintext, whereas SpecROP chains enable leakage from bytes 8, 16, 65 and 80 using different processing gadgets. The relevant register state is shown below the dotted lines for each gadget.

The first goal allows us to support our claim that gadgets enhance the capability of an attacker to access and leak secrets. The second goal will enable attackers to construct useful chains from gadgets in real binaries, where the number of individual gadgets is large, and intractable for manual analysis.

Constraint handling SpecROP gadget chains have to respect constraints to prevent scenarios which stop speculation. For example, a gadget which loads an `rip` relative offset into register `rax` prevents speculation on a later indirect jump using the same register. Another constraint is that the register holding the secret must not be overwritten. A final constraint is that the gadget chain must make the secret available at the location (register/memory) disclosed by the leakage gadget. Other constraints are important for specific side-channels, for example a NetSpectre attacker requires intermediate gadgets to not use AVX instructions.

SpecFication uses symbolic representation of `x86_64` instructions to model their effects on processor state. This approach allows us to both compose the effects of instructions to express the effects of a gadget and to express constraints over our gadgets. As in ROP-chain tools [9], SpecFication starts by enumerating every address which can be interpreted as a valid instruction sequence ending in an indirect jump or return instruction. For each of these sequences, we model the semantics of the instructions over the registers. Currently, we only handle certain classes of instructions such as data movement, logic, arithmetic and branch instructions. SpecFication uses the Z3 theorem prover [10] for testing constraints over each sequence.

5.1 SpecFication architecture

SpecFication works in three phases: (i) binary disassembly and preprocessing, (ii) gadget characterization, and (iii) constraint enforcement.

In the *binary disassembly phase*, the Capstone [11] framework disassembles our target binaries. We create instruction sequences, including unintended instructions, which end in a return or indirect jump. Since we prioritize short gadgets, we limit the length of explored sequences to 6 instructions. We also remove gadgets containing specific instructions such as unintended control-flow and privileged instructions.

Based on the intermediate representation of the gadgets provided by Capstone, we map gadget semantics of the gadgets in the *characterization phase*. We express the semantics of each instruction in the Hoare logic space and compose the effects of all constituent instruction to generate the overall effect of the gadgets. This makes the gadgets amenable to processing by the Z3 theorem prover [10] in the *solving phase*. An alternate approach would be to leverage previous work which provide the formal specification of `x86_64` instructions, such as Strata [12] and Dasgupta et. al. [13]. For determining the effect of a gadget chain, the code for each of the individual gadgets must be composed after removing the terminating control-flow instructions (which the attacker will poison).

5.2 Evaluation and Results

We now evaluate the effectiveness of SpecFication in finding usable SpecROP gadgets in common libraries. Particularly,

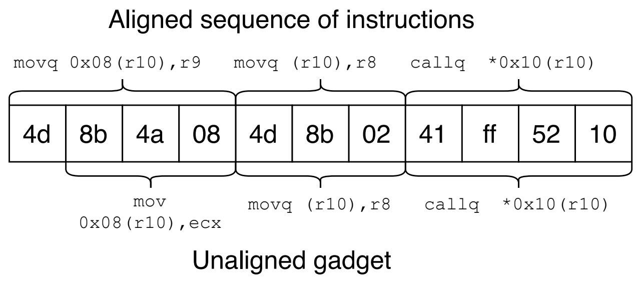


Figure 7: Example of an unaligned SpecROP gadget found in `libcrypto`

we look at instances of generic gadget types discussed in Section 3.1.1.

Target libraries We have chosen a set of target libraries based on their ubiquity and security criticality. Specifically, we analyze:

- `libcrypto` from OpenSSL v1.1.1d,
- `mod_ssl`, `mod_proxy` and `mod_http2` from Apache v2.4.41,
- `libdl` v2.30, and
- `libc` v2.30.

Testing setup SpecFication is written in Python, and depends on Capstone v4.0 and Z3 v4.8. All reported running times were measured on an i7-8700 processor with 16GB of memory running Debian 10 and Linux v5.4.

Results In the binary disassembly phase, SpecFication creates SpecROP gadgets: sequences of instructions ending in a return or indirect jump (an endpoint). For the analyzed binaries (which range from a few kilobytes to a few megabytes), the number of gadgets we analyze range from a few hundreds to thousands. Table 2 highlights statistics about the number of gadgets processed by SpecFication. Note that there are roughly 10 gadgets for each unique endpoint in these binaries. A large fraction of the gadgets also contain at least one unintended instruction.

As a particular use case, we report statistics for a run in which we search for data movement gadgets which load from memory at any address with register `r12` as a base. We can see that the tool finds at least one usable gadget in each library, the exception being the smallest library (`libdl`). The running time for the tool ranges from a few seconds to a few minutes depending on the size of the binary. The constraint solving phase, which involves calling the Z3 solver, is the largest contributor to the runtime.

We report statistics on the occurrence of gadgets classified as per the classes described in Section 3.1.1.

- Arithmetic gadgets: Table 4 highlights the number of arithmetic gadgets found in our target libraries, separated as per the register on which the operation is done. There are a larger number of arithmetic gadgets operating on the first eight registers (`rax` to `rbp`) than on the remaining (`r8` to `r15`). We do find a large number of gadgets, specially in `libc` which operate on the first four argument registers used by the System-V calling convention: `rdi`, `rsi`, `rdx` and `rcx`. This enables a SpecROP attacker to perform ample range of computation with function arguments: for example, if these arguments are pointers to secrets, the attacker can manipulate and access different parts of the secret. Finally, some gadgets target registers `rsp` and `rbp` which allow the attacker to access secrets on the stack.
- Shift gadgets: We found a smaller number of gadgets performing bit movement on registers: 25 in `libcrypto`, 95 in `libc` and a handful in other libraries. A detailed breakdown of the occurrence of such gadgets is shown in Appendix A.
- Data movement gadgets: We searched for gadgets in the target libraries which can move data between unique pairs of source and destination registers. Overall, there are a maximum of 240 unique pairs from the 16 general purpose registers (ignoring sub-registers) in `x86_64`. Table 3 reports the number of such gadgets in each library, as well as the number of gadgets which result from unintended instructions (one in five gadgets, on average). By chaining more than one such gadget, we can increase the number of register pairs, allowing greater flexibility in data movement. The column labeled "Chained" shows the number of register pairs possible by chaining two data-movement gadgets. In line with the general aim of gadget chaining, most libraries exhibit a significant increase in data-movement possibilities with increased chain length. In fact, chaining two gadgets allows 84% of the register pairs possible with chains of any length.
- Leakage gadgets: Table 4 also reports (in even rows) the occurrence of gadget leaking information into cache-based side channels, assuming that loads to secret-dependent addresses can leak information.

The results in this section not only illustrate the abundance of usable SpecROP gadgets in real libraries, but also demonstrate the practicality of using binary analysis for performing automated gadget search with formally specifiable constraints. This methodology has allowed us to construct practical SpecROP chains against OpenSSL (Figure 2) and similar to Spectre (Listing 1). We have demonstrated that this methodology can streamline the often manual process of finding gadgets in new binaries, and for newer side-channel attacks.

6 Mitigation

The mitigations against a SpecROP attack include generic defenses against SEA, such as preventing speculation, pre-

Library	Binary size	Endpoints	Gadgets	Unaligned	Data-movement gadgets addressing `r12` Endpoints	Gadgets	Analysis time (s)
`libcrypto`	`3.3M`	1,209	13,437	9,545	19	65	233
`libc`	`1.8M`	1,282	15,044	10,130	5	13	333
`libdl`	`15K`	22	266	205	0	0	4
`mod_ssl`	`235K`	48	490	332	2	4	8
`mod_proxy`	`131K`	30	338	246	1	1	5
`mod_http2`	`244K`	112	1,113	796	3	8	18

Table 2: Number of endpoints, gadgets, and unaligned gadgets found per library. We also show statistics for a particular use-case: searching for gadgets which load from an address based on register `r12`

Library	Data movement gadgets (unaligned)	Chained	Analysis time (s)
`libcrypto`	116 (9)	210	5,644
`libc`	101 (23)	204	8,432
`libdl`	2 (2)	2	305
`mod_ssl`	32 (10)	47	419
`mod_proxy`	34 (11)	46	295
`mod_http2`	27 (5)	72	875

Table 3: Occurrence of data movement gadgets moving data between registers. We report how many unique combinations of source and destination `x86_64` registers were found in each library.

venting branch predictor poisoning and control flow integrity. Other defenses particular to SpecROP might include limits on the number of branches followed speculatively. Static binary analysis techniques are inherently limited in their ability to detect whether usable SpecROP chains exist in binaries due to the large number of possible targets for a poisoned indirect jump or return instruction, and the exponential explosion in the number of possible sequences with the number of chained gadgets.

Preventing speculation in software The simplest protection against SEA is to restrict speculation following sensitive branches (where there is access to secrets). This can be done by using serializing instructions (for example `cpuid`), or memory fences (for example `lfence`) when the side-channel uses load instructions. If implemented by a shotgun approach, the performance implications are significant. However, we have shown how SpecROP chains expand the reach of SEA to access secrets, precluding fine-grained application of speculation barriers. Another mitigation, retpolines [14], protects indirect jumps by replacing them with return instructions. It also pollutes the Return Stack Buffer with the address of an infinite loop to prevent speculation on the `ret`. A practical, though partial, mitigation would be to identify code which might access secrets (e.g., array accesses following a bounds check), and insert retpolines on subsequent indirect calls and returns. This approach would still be vulnerable to gadget chains where the attacker is able to manipulate existing state to access secret state in unforeseen, and therefore unprotected, gadgets.

Limiting speculation in hardware Architectural proposals which limit the number of speculative control-flow instructions will effectively constrain the maximum length of a SpecROP chain, reducing the attack surface. However, the typical speculation window on a modern, high-performance processor extends to hundreds of instructions, where it is likely to contain tens of speculative control-flow instructions. We have already seen that it is impractical to chain more than 4-5 gadgets on these processors Figure 3a. A smaller limit on speculative control-flow instructions (say 2-3) may have an unacceptable performance overhead.

Preventing leakage Numerous proposals exist for mitigating SEA by closing the leakage channels, particularly for memory-based channels. InvisiSpec [15] proposes an separate buffer to hold speculatively loaded values, preventing them from affecting cache state. DAWG [16] dynamically partitions the cache to prevent cross-context cache channels. Other proposals which restrict execution of instructions dependent on speculatively accessed values [17] will effectively close all speculative side-channels, even if they do not prevent the chaining of gadgets.

Preventing branch poisoning Existing processors offer some degree of protection against control-flow hijacking across processes, and between different processor privilege levels (particularly userspace and the kernel) [18–20], either in hardware or through microcode updates. However, as we can see from results in Table 1, these measures do not completely mitigate all branch poisoning attack scenarios.

Enforcing control-flow integrity Any control-flow integrity mechanism aiming to mitigate SpecROP must account for speculative control-flow. Therefore, off-the-shelf

Library	Type	rax	rbx	rcx	rdx	rdi	rsi	rsp	rbp	r8	r9	r10	r11	r12	r13	r14	r15
libcrypto	A	665	259	34	78	69	65	186	48	0	0	0	0	15	33	10	3
	L	218	255	137	192	238	59	899	159	26	21	7	0	128	106	106	35
libc	A	889	317	128	171	419	421	32	3	13	29	2	0	8	12	6	15
	L	188	171	65	96	570	110	643	231	8	8	36	360	34	28	41	32
libdl	A	25	6	0	0	0	0	1	0	0	0	0	0	0	0	0	0
	L	9	11	0	8	0	0	43	0	0	0	0	0	0	0	0	0
mod_ssl	A	12	8	0	4	0	0	0	0	0	0	0	0	0	0	2	0
	L	2	38	0	2	0	10	10	1	0	0	0	0	8	2	23	0
mod_proxy	A	12	6	0	0	2	0	1	0	0		0	0	7	2	2	0
	L	8	2	0	0	0	0	13	0	0	0	0	0	2	0	0	1
mod_http2	A	46	5	0	5	0	9	0	0	0	0	0	0	0	0	0	0
	L	32	48	6	22	43	15	112	60	0	0	0	0	8	13	16	46

Table 4: Occurrence of arithmetic (A) and leakage (L) gadgets, listed on the first and second rows respectively for each library

CFI mechanisms are insufficient. Hardware mechanisms include Intel's upcoming CET technology [21] which plans to limit speculative execution following a jump or call. However, as we see with the example in Figure 2, SpecROP attacks are still possible within the limits imposed by the early implementations[1]. Since the set of targets allowed by CET is a superset of the actual set of targets, it remains to be seen if this imprecision can be used for speculative exploitation. There are newer proposals [22] for more complete CFI under speculative execution.

7 Related Work

The hypothesis that an advanced SEA might chain multiple code gadgets by having "multiple outstanding speculative changes of address stream caused by branch prediction" first appeared in a white-paper by ARM [23]. The paper, however, lacks an investigation of this idea, its practicality or the existence of the required code-gadgets. A similar hypothesis also appears in a more recent work by Canella et al. [24].

More recently, Mambretti et al. [25] demonstrated a practical SEA using multiple, chained gadgets to leak information following a mispredicted conditional branch. Unlike SpecROP, the attack requires a memory corruption bug in the victim binary to be able to inject arbitrary return addresses on to the stack. The targets for the return instructions used to chain gadgets comes from the overwritten stack. In contrast, SpecROP considers a stronger attacker model and is more stealthy. Our attacker cannot write arbitrary values to the victim's stack, and leaves no architecturally visible traces.

Bulck et al. [26] mention the possibility of using Load Value Injection(LVI) to transiently poison the values loaded from memory and used by return and indirect jump instructions, thereby chaining gadgets like in ROP/JOP. LVI depends on faulty behavior in specific CPU models. Moreover, their attack requires the ability to induce faults in the victim on the load instructions which they wish to poison. In general, this makes their attack practical only against victims running within an SGX enclave. The paper also does not comment on the practicality of causing multiple LVIs.

Other work which uses automated program analysis include Spectector [27] and oo7 [28]. Both of these works aim to prevent Spectre-like attacks by doing a static analysis of the code. oo7 statically applies taint analysis to binaries to check whether tainted values (secrets) can affect the outcome of conditional branches and speculative memory accesses. Spectector analyzes binaries for speculative code paths which leave microarchitectural traces which the architecturally intended path does not. Given the exact path through a SpecROP chain, Speculator would be able to detect that the instruction sequence is leaky. However, neither of these analyses can practically detect information leakage resulting from a SpecROP gadget chain since the set of available sequences to analyze grows exponentially with the length of the gadget chain.

SplitSpectre [29] also proposes an approach to implement the Spectre attack where the target does not contain any single gadget which loads the secret and performs a dependent load, instead relying on an attacker with the ability to inject the second load on the natural control-flow following the first. In contrast, SpecROP reuses existing gadgets for both loads. Further, Spectector will be able to detect that the code sequence used in SplitSpectre is leaky since there is a direct path between the dependent loads.

8 Conclusion

Through SpecROP, we extend our understanding of practical Speculative Execution Attacks by studying the ability to

[1]The early implementations of Intel CET will restrict execution following indirect jumps/calls to 8 instructions and 5 loads.

chain multiple gadgets. We demonstrate that poisoning multiple control flow instructions (returns and indirect jumps) is possible on modern processors. In fact, on tested processors, we can poison up to 4 indirect jumps with more than 50% success rate. This opens up the potential for attackers to use gadget chains instead of monolithic gadgets. With the use of the SpecROP technique, we show how generic gadgets can extend the reach of secrets accessible by an attacker, with an example of how this methodology may be applied to access and leak the AES key from a pointer to the context. We also demonstrate a practical attack which is able to leak multiple plaintext bits from a victim during encryption using the OpenSSL library.

To facilitate the gadget search, we design SpecFication, a gadget search tool for searching for generic SpecROP gadgets, from which we build up useful chains. Applying SpecFication to existing code bases, we see the abundance of small, generic SpecROP gadgets. We also study the possible mitigation techniques for SpecROP attacks. Our results lead us to believe that modern processors and programs are indeed vulnerable to a SpecROP attack, and that processors require hardware solutions for preventing malicious influence on branch/return prediction.

We have published as open-source the code for our experiments, the proof-of-concepts and SpecFication to enable others to further explore this methodology. The code is available at the GitHub repository `https://github.com/HexHive/specrop-public`.

Acknowledgments

We thank the anonymous reviewers for their insightful comments. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 850868).

References

[1] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[2] H. Shacham, "The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)," in *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007*, P. Ning, S. D. C. di Vimercati, and P. F. Syverson, Eds. ACM, 2007, pp. 552–561. [Online]. Available: https://doi.org/10.1145/1315245.1315313

[3] V. van der Veen, D. Andriesse, M. Stamatogiannakis, X. Chen, H. Bos, and C. Giuffrida, "The Dynamics of Innocent Flesh on the Bone: Code Reuse Ten Years Later," in *CCS*, Oct. 2017. [Online]. Available: Paper=http://vvdveen.com/publications/newton.pdfWeb=https://www.vusec.net/projects/newton

[4] T. K. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: a new class of code-reuse attack," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, ASIACCS 2011, Hong Kong, China, March 22-24, 2011*, B. S. N. Cheung, L. C. K. Hui, R. S. Sandhu, and D. S. Wong, Eds. ACM, 2011, pp. 30–40. [Online]. Available: https://doi.org/10.1145/1966913.1966919

[5] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. B. Abu-Ghazaleh, "Spectre returns! speculation attacks using the return stack buffer," in *12th USENIX Workshop on Offensive Technologies, WOOT 2018, Baltimore, MD, USA, August 13-14, 2018*, C. Rossow and Y. Younan, Eds. USENIX Association, 2018. [Online]. Available: https://www.usenix.org/conference/woot18/presentation/koruyeh

[6] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "Smotherspectre: Exploiting speculative execution through port contention," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, L. Cavallaro, J. Kinder, X. Wang, and J. Katz, Eds. ACM, 2019, pp. 785–800. [Online]. Available: https://doi.org/10.1145/3319535.3363194

[7] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack," in *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*, K. Fu and J. Jung, Eds. USENIX Association, 2014, pp. 719–732. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom

[8] M. Schwarz, M. Schwarzl, M. Lipp, J. Masters, and D. Gruss, "Netspectre: Read arbitrary memory over network," in *Computer Security - ESORICS 2019 - 24th European Symposium on Research in Computer Security, Luxembourg, September 23-27, 2019, Proceedings, Part I*, ser. Lecture Notes in Computer Science, K. Sako, S. Schneider, and P. Y. A. Ryan, Eds., vol. 11735. Springer, 2019, pp. 279–299. [Online]. Available: https://doi.org/10.1007/978-3-030-29959-0_14

[9] S. Schirra, "Ropper - rop gadget finder and binary information tool," http://scoding.de/ropper/, 2014.

[10] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[11] N. A. Quynh, "Capstone: Next-gen disassembly framework," *Black Hat USA*, vol. 5, no. 2, pp. 3–8, 2014.

[12] S. Heule, E. Schkufza, R. Sharma, and A. Aiken, "Stratified synthesis: Automatically learning the x86-64 instruction set," in *Programming Language Design and Implementation (PLDI)*. ACM, June 2016. [Online]. Available: http://dx.doi.org/10.1145/2908080.2908121

[13] S. Dasgupta, D. Park, T. Kasampalis, V. S. Adve, and G. Rosu, "A complete formal semantics of x86-64 user-level instruction set architecture," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, K. S. McKinley and K. Fisher, Eds. ACM, 2019, pp. 1133–1148. [Online]. Available: https://doi.org/10.1145/3314221.3314601

[14] P. Turner, "Retpoline: a software construct for preventing branch-target-injection," https://support.google.com/faqs/answer/7625886, 2018.

[15] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. W. Fletcher, and J. Torrellas, "Invisispec: Making speculative execution invisible in the cache hierarchy (corrigendum)," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*. ACM, 2019, p. 1076. [Online]. Available: https://doi.org/10.1145/3352460.3361129

[16] V. Kiriansky, I. A. Lebedev, S. P. Amarasinghe, S. Devadas, and J. S. Emer, "DAWG: A defense against cache timing attacks in speculative execution processors," in *51st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2018, Fukuoka, Japan, October 20-24, 2018*. IEEE Computer Society, 2018, pp. 974–987. [Online]. Available: https://doi.org/10.1109/MICRO.2018.00083

[17] O. Weisse, I. Neal, K. Loughlin, T. F. Wenisch, and B. Kasikci, "NDA: preventing speculative execution attacks at their source," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*. ACM, 2019, pp. 572–586. [Online]. Available: https://doi.org/10.1145/3352460.3358306

[18] Intel Corporation, "Deep dive: Indirect branch restricted speculation," https://software.intel.com/security-software-guidance/insights/deep-dive-indirect-branch-restricted-speculation, accessed: 2020-03.

[19] ——, "Deep dive: Indirect branch predictor barrier," https://software.intel.com/security-software-guidance/insights/deep-dive-indirect-branch-predictor-barrier, accessed: 2020-03.

[20] ——, "Deep dive: Single thread indirect branch predictors," https://software.intel.com/security-software-guidance/insights/deep-dive-single-thread-indirect-branch-predictors, accessed: 2020-03.

[21] ——, "Control-flow enforcement technology specification," https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf, May 2019, accessed: 2020-03.

[22] E. M. Koruyeh, S. H. A. Shirazi, K. N. Khasawneh, C. Song, and N. B. Abu-Ghazaleh, "SPECCFI: mitigating spectre attacks using CFI informed speculation," *CoRR*, vol. abs/1906.01345, 2019. [Online]. Available: http://arxiv.org/abs/1906.01345

[23] R. Grisenthwaite, "Cache speculation side-channels," January 2018.

[24] C. Canella, J. V. Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A systematic evaluation of transient execution attacks and defenses," in *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, N. Heninger and P. Traynor, Eds. USENIX Association, 2019, pp. 249–266. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/canella

[25] A. Mambretti, A. Sandulescu, A. Sorniotti, W. K. Robertson, E. Kirda, and A. Kurmus, "Bypassing memory safety mechanisms through speculative control flow hijacks," *CoRR*, vol. abs/2003.05503, 2020. [Online]. Available: https://arxiv.org/abs/2003.05503

[26] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yarom, B. Sunar, D. Gruss, and F. Piessens, "Lvi: Hijacking transient execution through microarchitectural load value injection," in *41th IEEE Symposium on Security and Privacy (S&P'20)*, 2020.

[27] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sanchez, "Spectector: Principled detection of speculative information flows," in *IEEE Symposium on Security and Privacy*. IEEE, May 2020. [Online]. Available: https://www.microsoft.com/en-us/research/publication/spectector-principled-detection-of-speculative-information-flows/

[28] G. Wang, S. Chattopadhyay, I. Gotovchits, T. Mitra, and A. Roychoudhury, "oo7: Low-overhead defense against spectre attacks via program analysis," *IEEE Transactions on Software Engineering*, 2019.

[29] A. Mambretti, M. Neugschwandtner, A. Sorniotti, E. Kirda, W. K. Robertson, and A. Kurmus, "Speculator: a tool to analyze speculative execution attacks and mitigations," in *Proceedings of the 35th Annual Computer Security Applications Conference, ACSAC 2019, San Juan, PR, USA, December 09-13, 2019*, D. Balenson, Ed. ACM, 2019, pp. 747–761. [Online]. Available: https://doi.org/10.1145/3359789.3359837

A SpecROP: Shift gadgets

Table 5 breaks down the shift gadgets found in each library based on the register which is operated on.

B OpenSSL attack gadgets

This section fully lists the gadgets used in the OpenSSL attacks shown in Figure 2 and Figure 6. The gadgets are found in `glibc` and `libcrypto`.

B.1 Processing gadgets

The following processing gadget increments the register `rdi` by a constant `0x20`. It is found in `libcrypto`.

```
1f6cc6:  lea    0x20(rdi),rdi
1f6cca:  callq  *0x18(%rax)
```

The following processing gadget loads a pointer from the memory referenced by `rdi` at offset `0x58` into register `rax`. It is found in `libcrypto`.

```
b2f1b:  mov    0x58(%rdi),%rax
b2f1f:  retq
```

The following processing gadget increments the value in `rdx` by `0x40`. It is found in `libcrypto`.

```
16b87f:  add     0x40, rdx
16b883:  add     rdx, rsi
16b886:  add     rdx, rdi
16b889:  lea     0x2e920(rip), r11
16b890:  movsxd  (r11, rdx, 4), rcx
16b894:  add     r11, rcx
16b897:  jmp     *rcx
```

The following processing gadget stores the value of `rdx + 0xf` in `rax`. It is found in `glibc`.

```
17df7e:  lea     0xf(rdx),rax
17df82:  retq
```

The following processing gadget stores the value of `rdx + 7` in `rax`. It is found in `glibc`.

```
17df26:  lea     0x7(rdx),rax
17df2a:  retq
```

The following processing gadget stores the value of `rax` in `rdx`. It is found in `glibc`.

```
12afdf:  mov     rax,rdx
12afe2:  callq   *0x28(r12)
```

The following processing gadget loads 8 bytes at address referenced by `rdx` at an offset of `0x50` into register `rdi`. It is found in `glibc`.

```
12ef33:  mov     0x50(rdx),rdi
12ef37:  mov     rdx,rsi
12ef3a:  callq   *rax
```

B.2 Leakage gadgets

The following gadget leaks the LSB of register `rax`. It is found in `glibc`.

```
cf6ac:  mov    -0x1b0(rbp),rdx
cf6b3:  mov    -0x1a8(rbp),rdi
cf6ba:  mov    r15d,esi
cf6bd:  or     0x2,esi
cf6c0:  mov    rbx,rcx
...
cf939:  test   0x1,al
cf93b:  je     cf6ac
cf941:  mov    r15d,r13d
cf944:  movb   0x0,(rdx)
cf947:  mov    -0x1b0(rbp),rdx
cf94e:  and    0xfffff7ef,r13d
cf955:  mov    rbx,rcx
cf958:  mov    r13d,esi
cf95b:  or     0x2,esi
...
```

The following gadget leaks the 3^{rd} LSB from the byte at offset 1 from the pointer in `rdx`. It is found in `glibc`.

```
f5393:  testq  0x400, (rdx)
f539a:  je     f5382
f539c:  mov    -0xb0(rbp), rdi
f53a3:  mov    -0xf0(rbp), edx
f53a9:  mov    (rdi, rax, 8), rax
f53ad:  test   edx, edx
f53af:  mov    rax, 0x50(rbx)
...
f5382:  add    0x1, rax
f5386:  add    0x20, rdx
f538a:  cmp    rax, -0x100(rbp)
...
```

Library	rax	rbx	rcx	rdx	rdi	rsi	rsp	rbp	r8	r9	r10	r11	r12	r13	r14	r15
libcrypto	3	10	2	6	3	1	0	0	0	0	0	0	0	0	0	0
libc	21	44	0	29	1	0	0	0	0	0	0	0	1	1	0	0
libdl	0	2	0	0	0	1	0	0	0	0	0	0	0	0	0	0
mod_ssl	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
mod_proxy	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0
mod_http2	0	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 5: Occurrence of shift gadgets in each library, broken down based on the affected register

The following gadget leaks the 6th LSB from byte referenced by the pointer in `rbp`. The first instruction is an unaligned instruction. It is found in `glibc`.

```
6aa23:  testb  0x20, (rbp)
6aa27:  je     6aa2f
6aa29:  mov    (rbx),rax
6aa2c:  orl    $0x20,(rax)
6aa2f:  add    $0x18,rsp
6aa33:  mov    r13,rdi
6aa36:  pop    rbx
6aa37:  pop    rbp
6aa38:  pop    r12
6aa3a:  pop    r13
...
```

The following gadget leaks the 5th LSB from the register `rdi`. It is found in `glibc`.

```
8ed34:  test      0x10,rdi
8ed3b:  je        8ed5a
8ed3d:  movdqu    (rdi,rsi,1),xmm0
8ed42:  pcmpeqb   (rdi),xmm0
8ed46:  pmovmskb  xmm0,edx
8ed4a:  sub       0xffff,edx
8ed50:  jne       8ee80
8ed56:  add       0x10,rdi
8ed5a:  mov       r11,r10
8ed5d:  and       0xffffffffffffffe0,r10
```