Bypassing memory safety mechanisms through speculative control flow hijacks

Andrea Mambretti[1,2], Alexandra Sandulescu[1], Alessandro Sorniotti[1],
William Robertson[2], Engin Kirda[2], and Anil Kurmus[1]

[1]IBM Research – Zurich, Rüschlikon, Switzerland
{asa, aso, kur}@zurich.ibm.com
[2]Northeastern University, Boston, USA
{mbr, wkr, ek}@ccs.neu.edu

***Abstract*—The prevalence of memory corruption bugs in the past decades resulted in numerous defenses, such as stack canaries, control flow integrity (CFI), and memory-safe languages. These defenses can prevent entire classes of vulnerabilities, and help increase the security posture of a program. In this paper, we show that memory corruption defenses can be bypassed using speculative execution attacks. We study the cases of stack protectors, CFI, and bounds checks in Go, demonstrating under which conditions they can be bypassed by a form of speculative control flow hijack, relying on speculative or architectural overwrites of control flow data. Information is leaked by redirecting the speculative control flow of the victim to a gadget accessing secret data and acting as a side channel send. We also demonstrate, for the first time, that this can be achieved by stitching together multiple gadgets, in a speculative return-oriented programming attack. We discuss and implement software mitigations, showing moderate performance impact.**

***Index Terms*—Transient Execution, Hardware Security, Side Channels, Speculative ROP, Memory Safety Mechanisms, Operating System Security**

1. Introduction

Memory corruption vulnerabilities have plagued the computer security field for more than 30 years. Multiple ways of exploiting memory bugs have surfaced, requiring controls to be placed at different levels in the software stack: mechanisms such as stack canaries and control flow integrity have been designed and deployed as a mitigation in existing software, while new languages were designed with memory safety to close this class of bugs in new programs [42], [46].

Recently, a new class of attacks, transient execution attacks [11], and more specifically speculative execution attacks [23], [21], [29], [25], [40], [10], [32] have been the subject of intense scrutiny. The ensuing vulnerabilities appear difficult to mitigate without considerable performance trade-offs, leading to the conclusion that speculative execution attacks will remain a problem for the foreseeable future, and therefore a possibly fruitful area of research [34].

A natural question to ask is whether the advent of transient execution attacks has changed the security stance of modern computing systems against memory corruption attacks: does the security of memory safety mechanisms, such as stack smashing protection (SSP), control flow integrity (CFI), and those embedded in memory safe languages, hold in the post-Spectre threat model?

In this paper, we show that multiple memory safety mechanisms that would otherwise successfully prevent exploitation of vulnerabilities can be speculatively bypassed to perform arbitrary memory reads. Because these attacks require a combination of techniques, we show that they do not apply to all memory safety mechanisms and a careful, case-by-case analysis is necessary.

At a high level, these attacks work by overwriting, either architecturally or speculatively, a backwards or forward edge, followed by the use of speculative code reuse attacks to leak data. In all cases, this overwrite achieves a speculative control flow hijack, i.e., a redirection of the speculative control flow to an attacker-chosen arbitrary address. One case of such an attack is the *speculative buffer overflow* discovered by Kiriansky and Waldspurger [21], where a return address is speculatively overwritten.

We demonstrate that SSP, GCC's vtable verification (VTV), and Go's runtime memory safety checks are all vulnerable. In particular, we develop a practical attack against SSP, where the mitigations against a stack-based buffer overflow in `libpng` can be speculatively bypassed to read arbitrary bytes from the victim program. This attack additionally leverages a last level cache (LLC) eviction attack to extend the speculative execution window, and a speculative return-oriented programming (ROP) attack to achieve a Flush+Reload side channel by reusing 5 gadgets from the victim program. Both components of the attack are not specific to SSP and generalize beyond our selected use case. Our results demonstrate that, although such end-to-end attacks are not trivial to mount, they are realistic. For this reason, we evaluate countermeasures for each attack scenario, showing that mitigations are both effective and viable from a performance standpoint.

This paper makes the following contributions:

- Demonstration of a practical attack against SSP-based buffer overflow mitigations, together with proof-of-concept attacks against GCC VTable Verification (VTV) and against Go's array bounds checks.
- Demonstration of speculation window lengthening leveraging LLC eviction of victim data.
- Practical speculative code reuse attack (ROP) to achieve side-channel send.

- Custom mitigations derived from `lfence`-based and masking-based approaches, withstanding the class of speculative architectural control flow hijacking attacks, together with a performance evaluation.

2. Speculative execution attacks on memory safety mechanisms

In this section, we describe end-to-end speculative execution attacks on abstracted memory safety mechanisms. We begin with a high-level overview of the various components necessary to perform such an end-to-end attack. We then proceed to analyze the class of speculative control flow hijacks which is at the heart of the attack; we refer to this general category of attacks as SPEculative ARchitectural control flow hijacks, or SPEAR, and detail them in Section 2.1. Furthermore, we analyze the eviction mechanism in Section 2.2, and the speculative ROP in Section 2.3.

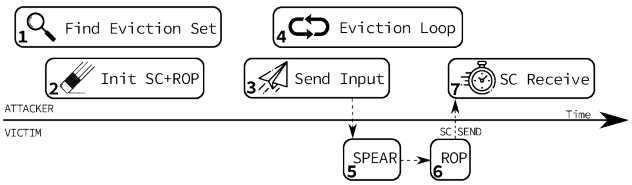


Figure 1: Overview of speculative attack against memory safety mechanisms.

Figure 1 shows an overview of the steps required to perform an end-to-end attack. The attack has a preparation phase (Steps 1 and 2), where eviction sets (to ensure the existence of a suitably long speculation window) are identified, memory used by the side channel is flushed and ROP gadgets are primed in the instruction cache. The attacker then submits an input to the victim in Step 3, crafted to trigger a violation of a memory safety property. We assume that traditional exploitation of the violation is prevented by a suitable memory safety mechanism. However, the attacker uses a speculative execution attack to bypass the mechanism by overwriting (architecturally or speculatively) control-flow data, and obtaining a speculative control flow hijack (Step 5). As a result, the victim is tricked into executing a side-channel send of attacker-chosen memory in Step 6: this is achieved with the ROP component, which reuses code snippets from the victim program, appropriately selected and primed in the initialization phase. The attacker can then execute the corresponding side-channel receive in Step 7. The success rate of the attack is increased by concurrently executing an eviction loop to lengthen the speculation window (Step 4) using the eviction sets found in Step 1.

Threat model: The general threat model for all attacks in this paper is a local unprivileged attacker, targeting a process holding a secret in memory. We do not assume that the attacker is able to inject code in the victim program's address space. We assume the attacker has knowledge of the victim program code, as well as the virtual address of code at runtime as is the case for Go, or that they can recover this information if randomized, possibly by using microarchitectural side channels [41], [16], [13]. Finally, because we opt to use a speculative ROP payload, we assume a hyperthread-colocated attacker, thereby sharing the instruction cache with the victim program, which the attacker leverages during the ROP chain warm-up phase. The goal of the attacker is, as in all transient execution attacks, to leak secrets from the target program. Attacks based on the architectural overwrite of a backward or forward edge correspond to the case where an attacker can provoke a memory safety violation whose traditional exploitation is prevented by hardening mechanisms in place. This is demonstrated in the SSP and CFI use cases. In this case, we assume that the victim program can either be executed multiple times by the attacker or that the program automatically restarts, given that each attack run leaks a limited volume of information and likely leads to abnormal program termination. This assumption remains realistic in practice because modern Linux distributions with `systemd` automatically restart services after abnormal termination.

Attacks based on the speculative overwrite of a forward edge correspond to a victim program with a memory safety check that the attacker can exercise and speculatively bypass. This is demonstrated in the Go use case. In this case, given that the overwrite of control flow data occurs speculatively, the attack does not lead to program termination, and so the attack does not require the ability to restart the victim.

2.1. SPEAR attacks

A SPEAR-vulnerable sequence is a code sequence that results in a speculative control flow hijack. A speculative control flow hijack allows an attacker to gain control of the target program's speculatively-executed code. This is a powerful primitive: an attacker can follow such an attack with a speculative ROP sequence to speculatively execute code gadgets that access a secret and send it to the attacker via a side channel.

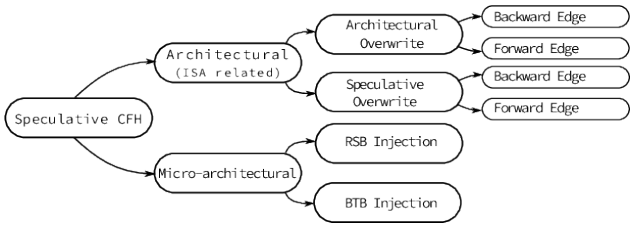


Figure 2: Overview of various Speculative control flow hijacking attacks.

Figure 2 shows a breakdown of the various instances in the SPEAR attack class in the context of different variants of speculative control flow hijacks. Classic speculative control flow hijacking attacks can be performed through microarchitectural components such as the Branch Target Buffer (BTB) and Return Stack Buffer (RSB) [23], [29], [25]. At the same time, the speculative control flow can also be influenced by instruction sequences that only affect architectural components, such as registers or memory: we refer to these as SPEAR attacks. For instance, executing the `call %rbx` x86 instruction speculatively when the value of `%rbx` is available to the execution unit will result in speculative execution continuing at the address in the `%rbx` register. Therefore, if the `%rbx` register can

be controlled by the attacker, a speculative control flow hijack can occur. This control by the attacker can either be architectural or speculative, as we will see next.

Similarly, a `push %rbx; ret` instruction sequence with the register value available would also simply continue execution at the provided address, with no need to predict where speculative execution continues via the RSB. Hence, SPEAR-vulnerable code patterns can concern both forward edges (`jmp` and `call`) and backward edges (`ret`).

The SPEAR classification offers us a convenient way to reason on attacks triggered by control flow data overwrite. SPEAR covers all attack scenarios studied in this paper, namely, speculative bypass of memory safety mechanisms; in addition, it covers other known attacks, such as the speculative overwrite of a backward edge [21], and the speculative bypass of manually-inserted array bounds checks [36].

2.1.1. Architectural overwrite. The case where an attacker controls the control-flow-influencing register architecturally, i.e., via the Instruction Set Architecture (ISA), is closely related to traditional memory corruption attacks. These attacks can nowadays be mitigated by mechanisms such as stack smashing protection (SSP) and, in general, CFI implementations that check the validity of control flow metadata before control flow is transferred, thus detecting and preventing outcomes induced by attacker-controlled overwrites. SPEAR architectural overwrite attacks focus on the opportunity that the attacker has to speculatively bypass the checks introduced by these mitigations.

```
1  ;Copy of RET Value
2      mov rax,[rsp]
3      mov QWORD[stored_ret], rax
4
5  ;Architectural Overwrite
6  ; (Attacker Controlled)
7      mov rax, QWORD[target]
8      mov [rsp], rax
9
10 ;Evict RET Value Copy
11     clflush [stored_ret]
12     lfence
13
14 ;Backward Edge Integrity Check
15 ; (Speculation Trigger)
16     mov rax, [rsp]
17     cmp rax, QWORD[stored_ret]
18     jne my_exit
19
20 ;Backward Edge Hijack
21     ret
```

Listing 1: Architectural backward edge overwrite.

We provide in Listing 1 the snippet of code that illustrate the backward edge case for architectural overwrites. A similar snippet for the forward edge case is reported in Listing 17, Appendix B. The structure of both cases is similar: the original value of the edge (line 3) is preserved in a safe location, after which, we assume that the architectural overwrite is performed (line 8) with an attacker-controlled value (e.g., through a buffer overflow). Afterwards, the program executes an integrity check on the forward or backward edge (line 17) before performing the control flow transfer (e.g., SSP or CFI check). To increase the success rate of the attack, we try to maximize the speculation window caused by the integrity check, for instance by evicting its reference value – in the snippet, this step is captured by a `clflush` instruction (line 11). If the CPU mispredicts the outcome of the check, it might execute either a `ret` (backward edge) or a `call` (forward edge) which transfers the control towards the attacker-controlled value used in the architectural overwrite (line 21).

2.1.2. Speculative overwrite. Alternatively, the attacker may control the control-flow-influencing register speculatively. This means that in a first phase, speculative execution is triggered (for example by a conditional branch). In a second phase, the attacker speculatively influences the control flow edge, thus hijacking speculative control flow. The control-flow-influencing value may be the result of a load from an address that is generated during the speculative execution phase, or it may be loaded from a location that is speculatively overwritten by a preceding store operation, resulting in speculative store-to-load forwarding.

We provide in Listing 2 the snippet of code that illustrate the backward edge case for speculative overwrites. A similar snippet for the forward edge case is reported in Listing 17, Appendix B. Both cases share the same structure. First, speculative execution is triggered by a condition (line 2). Then, the speculative overwrite is performed through some instruction within the speculated part of the code. Here, the value used for the overwrite is under attacker control (line 7). Finally, the overwritten value is used for control flow transfer allowing the attacker to hijack the speculative control flow (line 10).

	Architectural		**Speculative**	
Family	*Fwd*	*Bwd*	*Fwd*	*Bwd*
Intel Broadwell	99.5	94.9	99.5	98.7
Intel Skylake	97.6	98.3	98.2	92.1
Intel Coffee Lake	99.8	98.1	99.7	99.4
Intel Kabylake	99.5	95.9	100	99.5
AMD Ryzen	100	100	100	100

Table 1: Success rate (in percentage) computed over 1000 iterations for architectural or speculative overwrites of backward and forward edges performed on various architectures families.

2.1.3. SPEAR experimental results. We follow the methodology of Mambretti et al. [31] and test all four snippets using the Speculator tool [30], which aids the detection of speculative control flow transfers by using performance monitor counters (PMC) and *speculation markers*.

The SPEAR experimental results are shown in Table 1. Each success rate is computed on 1000 iterations. In the *architectural overwrites* case, speculative control flow hijacks are observed at least 95% of the time for Listing 1 and 97% of the time for Listing 17 on all tested architectures. The results prove that control flow is indeed speculatively transferred to the overwritten location, thereby bypassing the checks during speculative execution. Therefore, we conclude that SPEAR attacks with architectural overwrites can result in speculative control flow hijacks. In the *speculative overwrites* case, for the backward edge case the success rate is at least 92% while for the forward edge case it is at least 98% . The experiment results demonstrate that speculative overwrites

are feasible and lead to speculative control flow hijacks provided a sufficiently large speculation window exists to facilitate the edge overwrite followed by the dereference.

```
1  ;Speculative execution trigger
2      ...
3
4  ;Speculative Overwrite
5  ; (Attacker Controlled)
6      mov rax, QWORD[target]
7      mov [rsp], rax
8
9  ;Backward Edge Hijack
10     ret
```

Listing 2: Speculative backward edge overwrite.

2.2. Speculation window and eviction

SPEAR attacks require the existence of a speculation window to permit the execution of the control flow transfer and the side channel send operation, a common precondition for all speculative execution attacks. This requires a speculative execution trigger, i.e., an instruction that causes a wide-enough window of dependent instructions that are executed but not retired. This is usually achieved when the process accesses uncached data: the speculation window then corresponds to the time for the access to main memory to complete. In Listing 17 for example, this is achieved with the `clflush` instruction. To show the necessity of a wide speculation window, we re-run the snippet without `clflush` in the Speculator tool and verify that indeed the control flow hijack only takes place in about one run out of 1000. When it does, the window is only a couple of instructions wide. We therefore conclude that without eviction, or other similar approaches to lengthen the speculation window, SPEAR attacks are unlikely to be practical.

In all snippets referenced by this section, the speculation window is artificially lengthened by flushing one of the memory operands of the compare instruction. This may not be realistic, as it imposes a strong requirement on the victim code to include a flush (or comparable) instruction. Instead, because the last level cache (LLC) is shared and often inclusive, the same effect can be accomplished more realistically by an external attacker thread computing an *eviction set* and performing a small number of accesses to addresses in this set. An LLC eviction set competes for the same LLC slice and cache set as the target address to be evicted. Existing techniques for performing such attacks typically assume knowledge of the targeted physical address, as the LLC is physically indexed. As a consequence of rowhammer attacks, this is no longer realistic, as most OSes have removed access to physical mappings for unprivileged users. In Linux, privileged-only `/proc/PID/pagemap` access [1] was introduced in release 4.0.

We demonstrate here that such eviction attacks can still be performed without knowledge of the physical address. To this end, we perform the eviction in two steps. The first step consists of the identification of an eviction set for a cache line in a page under the attacker's control, by following the approach of Maurice et al. [33]. The second step consists in releasing this page to the OS, and executing the victim process such that it reuses the previously-created page. This permits the reuse of the eviction set constructed and verified to be working in the first step. To increase the victim data eviction success rate, we follow the eviction set loading method proposed by Liu et al. [28]. We show details of such a practical attack in Section 3.1.2 for SSP.

2.3. Speculative ROP

To perform a complete speculative execution attack, the speculative control flow hijack must be followed by a *side channel send* gadget with a secret input. Unfortunately, Spectre v1-type Flush+Reload side channel send gadgets are known to be difficult to find [23], [50]. As in classical control flow hijacks [37], however, a speculative code reuse attack can be performed by chaining the speculative execution of gadgets to construct a Flush+Reload side channel send sequence. To chain the gadget sequences, we proceed in a similar way to traditional ROP attacks, with sequences ending in `ret` instructions, yet with two additional requirements. These requirements for performing speculative code reuse are the following: *i*) execution of all instructions in the gadgets must fit into the speculation window; *ii*) all code pages in which the gadgets reside must be present and mapped in the victim process.

The first requirement is a consequence of the behavior of speculative execution. In particular, all return values used to chain gadgets need either to be in store buffers or in cache. Indeed, whether the return addresses are speculatively or architecturally written to the stack, execution of return instructions will make use of these addresses if they are available, with the CPU preferring those values for steering front-end fetches over values provided by the RSB. If the return address is not in cache (or in store buffers), loading the return address from memory will exceed the speculation window in practice and only RSB-based branch prediction will be in use, which will result in failure of the attack. A similar approach and analogy exists with forward edges for code reuse. Using the Speculator tool, we obtain experimentally that the maximum number of empty gadgets that fit in the largest speculation window is 20 on our Kaby Lake i7-8550 test platform.

The second requirement is needed to avoid page misses during gadgets execution. In the event of a page miss, speculative execution might halt or nested speculation might be triggered. Despite of the two strict requirements, we show in Section 3.1.3 that speculative ROP can be achieved for a practical use case.

3. Case studies

In this section, we analyze different case studies where memory safety mechanisms can be bypassed with SPEAR attacks. In particular, in Section 3.1 we use a practical attack that speculatively bypasses SSP leveraging architectural overwrites of backward edges. Section 3.2 analyzes architectural overwrites of forward edges, targeting two prominent CFI frameworks, GCC VTV and LLVM CFI. In the GCC VTV case, we show how the integrity check of the forward edge can be used to perform a speculative control flow hijack. For LLVM CFI, we conclude that the constraints of its implementation does not allow SPEAR attacks to be mounted in practice, demonstrating

the importance of careful feasibility analysis. Finally, in Section 3.3, we demonstrate two types of speculative bounds check bypasses in the Go language using speculative overwrites of a forward edge. We show how the attacker may influence the control flow target through both a load whose address value is attacker controlled and a load of a value that was speculatively overwritten by the attacker. We demonstrate practical implementation of speculative ROP and LLC eviction techniques as part of the end-to-end practical attack on SSP, i.e., we implement all the stages in Figure 1. We do not further re-implement them in the case of CFI and Go, where they would equally apply and where we focus instead on the central part of the attack as a proof of their feasibility, i.e., we implement only Step 5 in Figure 1. Therefore, the success rates reported below refer either to all the stages together for the SSP case (7.19%) or just the hijack stage for the Go (above 80%) and the GCC VTV (85%) use cases, hence the large difference.

3.1. Attacking stack canaries

Stack canaries are one of the earliest mitigations against buffer overflows [42], and are widely used to this day. Among the most broadly adopted implementations are LLVM's and GCC's Stack Smashing Protection (SSP) and Microsoft's `/GS`. At a high level, stack canaries work by inserting a value (the *canary*) between stack buffers and control-flow influencing data on the stack, in particular the saved return value. The integrity of the canary is then checked prior to using the saved return value. Local stack variables are reordered such that buffers, likely to be overflowed, reside adjacent to the canary while code pointers remain further away. This way, contiguous overflows of local stack buffers can be detected by the integrity check. The chosen canary value is randomly generated once during process execution start, and stored in a safe location.

Each compiler performs the instrumentation differently but in essence the mechanics are identical with respect to SPEAR attacks; we therefore focus on the example of LLVM on Linux `x86_64`. Implementations consist of two distinct *instrumentation atoms*. The instrumentation atoms on our target system are shown in Listing 3. The first, the prologue SSP atom, is placed after the function prologue and local variable allocation, and is responsible for storing the canary value on the current stack frame. The second, the epilogue SSP atom, is placed before local variable deallocation and the function epilogue. It compares for equality between the global and local canary values; if the values differ, the `__stack_chk_fail` function is called, terminating the program. If the local canary value was not modified during function execution, the function returns normally. We show next that this particular comparison can be the target of a SPEAR attack.

3.1.1. SPEAR attack on LLVM-SSP. The pattern of the SSP instrumentation closely resembles that of Listing 1. Under our threat model, an attacker with a buffer overflow against a function protected by SSP can perform a SPEAR architectural overwrite attack of the return value of that function. We describe a practical attack targeting a version of `libpng` with a reported buffer overflow (CVE-2004-0597): the bug is not exploitable in the traditional way owing to the fact that the function is compiled with SSP. We show how a speculative adversary can exploit the SPEAR architectural overwrite to leak arbitrary secrets from the victim.

The attack proceeds as follows: in the first step, the attacker overwrites the saved return address of the victim function. In the second step, the attacker leverages a misprediction in the conditional jump of the canary integrity check, thus transiently executing a return to the previously overwritten return address. This PHT-based misprediction is achieved by the attacker in a way similar to Spectre v1, by executing the canary integrity check with an intact local canary sufficiently many times. As discussed in Section 2.2, another requirement is that a sufficiently long speculation window exist. We achieve this by evicting the global canary from the LLC, as we show in Section 3.1.2. The attacker is then able to perform a side-channel send operation by constructing a speculative ROP chain to access a secret as we show in Section 3.1.3.

```
1  func:
2      prologue
3
4  ; Store canary on the stack
5      mov rbx, QWORD[fs:0x28]
6      mov QWORD[stack_canary], rbx
7      ...
8      body
9      ...
10 ; Check for corrupted canary, if yes fail
11     mov rbx, QWORD[stack_canary]
12     xor rbx, QWORD[fs:0x28]
13     je exit
14     call __stack_chk_fail
15 exit:
16     epilogue
17     ret
```

Listing 3: Stack canary check instrumentation example.

3.1.2. LLC eviction of the global canary. We apply the two-step method described in Section 2.2 for the eviction of the global canary from LLC, and thus from all cache levels by the property of inclusiveness of caches on the target platform. The global canary value is always stored at a fixed offset in a page: we use this property to find eviction sets for this particular offset by using the undocumented Intel LLC slice function reverse engineered by Maurice et al. [33].

The attacker process first identifies a page with a known eviction set and then unmaps it to be reused by the victim to store its canary. This is achieved with two processes under attacker control, as follows. At first, one of them maps a hugepage and enters a loop in which it brings an eviction set into cache and waits for feedback from the second attacker process. The latter in turn probes its own stack canary and reports back a success as soon as the canary is no longer cached. Once the eviction set is identified, the attacker releases the page, which is now ready to be reused by the victim process to store its canary. The page release is done via `madvise` which instructs the system about the process memory behavior, in this case indicating that a certain memory range will not be accessed soon (`MADV_DONTNEED`). We manually craft the memory area released by the attacker in order to shift the target page frame in the right position in

the kernel buddy freelist. We empirically verify that the reuse of a page frame for the victim canary page occurs with 100% success rate when attacker controls victim startup. When the attacker does not control victim startup, the success rate drops (but remains above 50%), because synchronization is more difficult and all processes in the system consume resources from the buddy freelist. Factors that influence the success rate include the order of the page in the freelist, the "distance" between the release operation by the attacker and the request operation by the victim process. We note that we do not use any artificial synchronization mechanism between the victim and attacker, which makes this attack widely applicable.

While data eviction is a common part for speculative execution attacks, we adapt an LLC eviction technique only used previously in the context of side channels. Existing techniques for Spectre attacks evict large quantities of data from the caches, lowering success rate. For the SSP attack, this technique ensures the ROP gadgets executed speculatively remain in cache. Their eviction would result in the attack failing, because the RSB would be used to predict return location.

3.1.3. Speculative ROP. We now focus on building and using a speculative ROP chain that accesses a secret and leaks it through a side channel. We use the Flush+Reload cache side channel initially used by Kocher et al. [23], although other side channels can be used similarly [10], [40], [32].

In Section 2.3, we have identified two major constraints on the attack: *i*) a limited number of instructions can fit into the speculation window; and, *ii*) all code pages in which the gadgets reside must be present and mapped with corresponding TLB entries. In addition to these requirements, we note that gadget code, as well as any data accessed by gadgets, must be available in cache during speculative execution. Typically, this is not an issue in speculative execution attacks because the attacker can run several attack iterations as warm up phase to bring the required data in the cache, whereas this attack is *single shot*: the process terminates after each attempt and this is an additional requirement.

```
1  mov rax, secret
2  shl rax, 8
3  add rax, shared_array
4  mov rax, [rax]
```

Listing 4: Example of Flush+Reload gadget.

Concerning the first requirement, the Flush+Reload side-channel send gadget only requires a few instructions: there are sufficiently short gadgets available, and length is therefore not an issue in practice. For the second requirement instead, we create a tool to search for gadgets in code that was recently accessed by the victim program, for which pages are present and mapped in the victim process. The tool traces the victim process and collects all executed shared (library) code pages, which are then fed into an existing ROP gadget search tool, ROPgadget [3]. We run the tool on the victim program and find 26 mapped code pages within the 4 different modules used by the victim: `libc`, `libpng`, `libz` and `ld`. In total, the tool discovered 2096 gadgets, out of which 406 are candidates for building the side-channel send gadget. Per-gadget occurrences are shown in Table 2. Finally, to ensure that all gadget sequences are in cache, a hyperthread-colocated attacker performs a ROP chain warm up phase by executing the chain in close temporal proximity with the SPEAR attack.

Gadget type	*Occurrence*
pop reg ; ret	262
mov reg1, [reg0] ; ret	69
shl reg, 8 ; ret	4
add reg1, reg0 ; ret	71

Table 2: ROP gadgets used for building Spectre v1 chain with their corresponding occurrences. The search space is a subset of `libc`, `libpng`, `libz` and `ld` executable pages, obtained by filtering out pages unmapped in the victim's address space and pages without a valid TLB mapping.

We build a 5-gadget ROP chain using the ROP gadgets found by our gadget search tool. The chain is functionally equivalent to the Flush+Reload gadget shown in Listing 4. The chain accesses a target address computed using a secret byte value, as in the initial Spectre attacks [23]. Because Flush+Reload requires shared memory, we choose the target address to reside in such a shared memory area between attacker and victim, the first 16 readable and executable pages of the `libpthread` library. To leak one byte we use an array size of 256. To avoid prefetching effects during side-channel receive, we choose the element size to be 256, i.e., four cache lines. The total array size equals `256 x 256` bytes, 16 pages.

By splitting the Flush+Reload gadget in small sequences of instructions as shown in Listing 4, we easily find the required gadgets within the constraints of the attack. The ROP chain that we find and use in the attack is shown in Listing 5. This chain pops the addresses (controlled by the attacker) of the start of the 16 pages and of the targeted secret from the stack. Next, the secret value is loaded at line 8. The next speculative gadgets multiply the secret value by 256 and compute the target address. The last speculative gadget dereferences the target address, resulting in a load being issued during speculative execution. This eventually brings the value into the cache to be observed by the attacker. The whole chain therefore allows the attacker to implement a universal read primitive over the victim process speculatively, using a Flush+Reload attack and the attacker's control over the stack.

```
1  libpng.so.3.1.2.5 : 0x7960
2      pop rdx
3      ret
4  libpng.so.3.1.2.5: 0x7f0a
5      pop rsi
6      ret
7  libpng.so.3.1.2.5 : 0x128ec
8      mov eax, dword ptr [rsi]
9      mov byte ptr [rdi + 6a], al
10     ret
11 libpng.so.3.1.2.5 : 0x9f4b
12     shl rax, 8
13     add rax, rdx
14     ret
15 libpng.so.3.1.2.5 : 0x9fde
16     add eax, dword ptr [rax]
17     add byte ptr [rdi], cl
18     xchg eax, ebp
19     ret
```

Listing 5: Flush+Reload gadget ROP chain.

3.1.4. Attack evaluation and results. The attacker targets the `libpng` version 1.2.5 which is vulnerable to

CVE-2004-0597 [4].

CVE-2004-0597 is a stack buffer overflow which allows the attacker to read length bytes in readbuf. Due to improper sanitization of length, a read larger than PNG_MAX_PALETTE_LENGTH is allowed in a stack buffer. The target victim is a program that receives a .png file and parses the file using the unpatched libpng-1.2.5. When building the victim target with stack canaries enabled, the compiler will instrument png_handle_tRNS with the corresponding prologue and epilogue SSP atoms. As expected, SSP protects png_handle_tRNS from exploitation by stopping execution before the function returns. However, using a SPEAR architectural overwrite attack, we can perform a speculative control flow hijack. During the SPEAR attack, the attacker feeds .png files of the legitimate length to train the pattern history table to bypass the stack canary check. Then, the attacker provides a length larger than PNG_MAX_PALETTE_LENGTH that overwrites the value of the return address to trigger the speculative ROP attack.

We confirm the attack works and leaks bytes at arbitrary, attacker-chosen addresses from the victim memory, on Intel Skylake and Coffee Lake with latest microcode updates, and on Ubuntu 16.04 and 18.04 (both with kernel version 4.15.0) with all default Spectre mitigations enabled. Namely, both setups include __user pointer sanitization and usercopy/swapgs barriers mitigations against Spectre v1. Moreover, default mitigations against Spectre v2 are present (retpoline, IBPB, IBRS_FW, and RSB filling), excepting STIBP which is disabled on Ubunutu 16.04. We report the attack evaluation results on Intel(R) Core(TM) i7-6700K CPU @ 4.00GHz (Skylake) running Ubuntu 16.04.6 with kernel version 4.15.0. As described in Section 2, the attack has an initialization phase where eviction sets are identified, memory used by the side channel is flushed and the ROP sequence is primed. Concurrently with the submission of the malicious payload, the attacker also runs the eviction loop to lengthen the speculation window by causing the eviction of the stack canary in the victim.

```
1  void /* PRIVATE */
2  png_handle_tRNS(png_structp png_ptr, png_infop
       info_ptr, png_uint_32 length)
3  {
4    ...
5    png_byte readbuf[PNG_MAX_PALETTE_LENGTH];
6    ...
7    if (png_ptr->color_type ==
         PNG_COLOR_TYPE_PALETTE) {
8      if (!(png_ptr->mode & PNG_HAVE_PLTE))
9      {
10       /* Should be an error, but we can cope with
              it */
11       png_warning(png_ptr, "Missing PLTE before
             tRNS");
12     }
13     ...
14     png_crc_read(png_ptr, readbuf, (png_size_t)
           length);
15     png_ptr->num_trans = (png_uint_16)length;
16   }
17   ...
18 }
```

Listing 6: libpng vulnerable snippet related to CVE-2004-0597.

We measure the attack success rate as the number of times the attacker is able to correctly guess a secret byte from the victim memory space, per total number of runs. We report means over 100 runs with 95% confidence level. The end-to-end attack success rate is $7.19\% \pm 0.62$, for a single run. In practice the attacker does, as in most other Spectre attacks, re-run the attack as many times as necessary to improve its guesses and reach close to 100% success rate. Therefore, we compute the leakage rate based on the attack time, which is measured as the full duration of repeating the attack 100 times against the re-startable victim. The duration includes the restart time of the victim and the attacker execution time. The end-to-end leakage rate of victim bytes is 0.3 bytes per second (with all correct guesses), which we deem sufficiently high for practical use. Due to different binutils versions in the two distribution versions, we observe a slight leakage rate drop in the Ubuntu 18.04 environment.

For improving the success rate, and therefore improving the leakage rate of the end-to-end SSP attack, one needs to improve the success of each individual stage of the attack showed in Figure 1. In addition, the attacker may run too fast or too slow with respect to the victim (the attacker simply attempts to synchronize with busy loops), which can also lead to failure of the attack. We have verified such synchronization is successful in our PoC in 78% of cases. We already report in Section 3.1.2 results for the LLC eviction stage: 100%. Because changes to the victim can affect the success rate, measuring the success of other individual steps within the end-to-end PoC is very difficult. However, based on these numbers and experiments outside the PoC, we infer that the greatest area for improvement in the leakage rate should come from improving the ROP gadget phase (e.g., limiting cases where the gadget code is not in cache) and side channel receive/send (e.g., limiting cache noise from eviction activity and other sources, or using another side channel).

3.2. Attacking CFI

Control Flow Integrity (CFI) of forward edges aims to protect the integrity of code pointers used in indirect calls and jumps. CFI implementations contain two main parts: instrumenting all indirect control transfers to check their validity at runtime, and classifying valid control flow transfers (typically using static analysis at build time). We analyze here two prominent cases: the GCC Virtual Table Verification (VTV) [43] mechanism to prevent c++ virtual table corruption, as well as LLVM-CFI [26], a publicly available, low overhead, forward-edge CFI implementation. In the GCC VTV, we prove that a SPEAR attack is possible, while in the LLVM-CFI case we conclude that eviction-related considerations result in the speculation window being too short for practical exploitation. In particular, this case study demonstrates that we cannot conclude that SPEAR attacks apply equally to all implementations of memory safety-related defenses, and case-by-case analysis is necessary.

3.2.1. GCC VTV. In the GCC VTV implementation, for every call to a virtual function in the program, the compiler inserts a check to make sure that the pointer used for the indirect call belongs to the virtual table of the object. Such check is represented by a call to the function __VLTVerifyVtablePointer implemented in libvtv.so library. Within this function, the pointer is

looked up from the table; if found, the function simply returns to the program which will perform the call, otherwise, it gracefully fails. If an attacker can successfully evict the cache line related to the variable the pointer is tested against, speculative execution is triggered during the evaluation of the check. In that case, the indirect call to the virtual function is speculatively executed and the code at the corrupted pointer is executed. At this point, the attacker has performed speculative control flow hijack and can mount a data exfiltration attack as described in Section 3.1.

In our proof-of-concept implementation of this attack, we artificially evict from all cache levels the variable related to the vtable of the object within the `libvtv.so` code. Then, we create a `c++` program that defines two different classes each containing one virtual method. The first class is our target for the forward edge overwrite. To verify whether speculative control flow hijack takes place, we instrument the program to read performance monitor counters and set the speculative control flow hijack target to contain a *speculation marker*. We use the second class to instantiate the object that is later corrupted.

After object initialization, we perform a vtable pointer overwrite in our victim object making it point to the vtable of the first class. Finally, we perform the virtual call for the control flow transfer which is instrumented by GCC VTV with a call to the integrity check inside the `libvtv.so` library. During normal execution, this overwrite is detected by the library which reports the corruption and prevents the control flow transfer by terminating the application. With a SPEAR attack as described here, we verify that control flow hijacking occurs in 85% (n=1000), demonstrating that a SPEAR architectural forward-edge attack is viable against GCC VTV. We note also that the redirection is performed to a vtable of a completely unrelated class, a case which should be prevented by VTV. A real-world attack would additionally require evicting the compare variable, for example by using the same method as in Section 3.1.2, as well as a way of achieving a side-channel send for the attacker, as in Section 3.1.3.

3.2.2. LLVM CFI. The CFI solution implemented in LLVM uses function types as *equivalence classes*: an indirect call to a function of a different type than the one specified by the programmer is forbidden by the CFI instrumentation. This is achieved by placing functions of an equivalence class in a jump table, thereby having as many jump tables (whose addresses are carefully chosen) as equivalence classes. The instrumentation for indirect calls then consist in simply checking that the address of the target fall within the range of the jump table, and at the right alignment.

This range check can be seen as a check against a compile-provided constant value, using the address of the provided target. Both of these components are by design available and cached while performing this check: evicting the code that contains the range check would result in speculative execution stopping, and evicting the address of the target would result in the iBTB being used for speculative execution. In either case, a SPEAR attack would fail. The attack may be triggered without any attempt to artificially extend the speculation window, but, as demonstrated experimentally in Section 2.2, the resulting speculation window is rare and short, making such attacks unlikely to be practical. We conclude that LLVM CFI is in practice not vulnerable to SPEAR attacks.

3.3. Attacking memory safe languages

Most modern languages are designed to ensure memory safety. Instrumental to achieving this property are bounds checks for load and store operations into arrays. In this section, we show how bounds checks may be speculatively bypassed, allowing the transient execution of out-of-bounds load and store operations. We show under which conditions this leads to a SPEAR attack.

We focus in this case study on the popular Go programming language, runtime and compiler. We present two variants, one where data that influences a forward control flow edge is architecturally overwritten and one where a forward edge is speculatively overwritten. In either case, the attacker is able to achieve a speculative control flow hijack. We prototype both variants and show the conditions under which the attack succeeds at a rate exceeding 80%.

```
1  type slice struct {
2      array unsafe.Pointer
3      len   int
4      cap   int
5  }
```

Listing 7: Arrays in Go.

Before detailing the two attacks, we give a brief introduction to the way the Go compiler manages arrays and bounds checks. Arrays in Go are represented in memory as the `struct` shown in Listing 7. The address of the contiguous chunk of virtual memory backing the array is stored in `array`. The number of elements that `array` can hold (and implicitly the size of the memory chunk since Go is statically typed and the size of the elements is always known) is stored in `cap`. The current number of elements that have been stored in the array is stored in `len`.

Whenever an array access is performed in Go, the compiler will add appropriate bounds checks. This is achieved in the course of the compiler pass to translate the abstract syntax tree (AST) into the static single assignment (SSA) intermediate representation by adding an `IsInBounds` meta-operation before every array load or store. `IsInBounds` takes two arguments, the index of the current access and the length of the array, and drives a conditional jump either to the basic block that performs the array access if the index is between zero and length minus one, or a jump to a function that raises a `panic` otherwise.

```
1  mov rcx, [array]
2  cmp [array+0x8], rax
3  jbe runtime.panicindex
4  mov rax, [rcx+rax*8]
```

Listing 8: Bounds check in Go.

`IsInBounds` is translated by later passes into a sequence of instructions similar to the one shown in Listing 8. The snippet shows a load from an array of integers: at first `rcx` is loaded with the address of the memory array, a compare instruction is issued between the index of the array access in `rax` and the array length at `array+0x8`.

If the index is negative or not strictly less than the length, the code jumps to a call to the `runtime.panicindex` function. Otherwise the array access is performed.

The conditional jump generated by the `IsInBounds` meta-operation may speculatively execute the wrong jump target and perform a transient load or store operation out of bounds. We show two distinct code patterns, one leveraging a load and one a store, that may lead to speculative control flow hijack.

```
array[index].function()
```

Listing 9: Load-based speculative control flow hijack code pattern.

3.3.1. Load-based SPEAR speculative attack. The first pattern is shown in Listing 9. It represents an instance of a SPEAR-speculative attack and consists of an interface function call, where the interface is stored into an array of interfaces `array`, dereferenced at position `index`. Note that the array must be an array of interfaces so that calling the function is achieved by an indirect call. For the attack to be successful, we need `index` to be attacker-controlled and the attacker must be able to store the value of two pointers in the memory space of the target process at a known location. The first condition is met whenever a process accesses an array using an index that is received as an external input. The second condition is very commonly met since programs store user-provided input for processing. Knowledge of the location of the stored pointers depends on the memory area being used, and is aided by the deterministic nature of the Go allocator.

```
1  type iface struct {
2      tab  *itab
3      data unsafe.Pointer
4  }
5
6  type itab struct {
7      inter *interfacetype
8      _type *_type
9      hash  uint32
10     _     [4]byte
11     fun   [1]uintptr
12 }
```

Listing 10: Structs used by interface calls.

Without loss of generality, we describe the case where `function` is the first function defined by the interface. Exploitation proceeds as follows: first, the attacker prepares the memory structures that are used when an interface call is performed. The structures are shown in Listing 10, and are used by dereferencing the `tab` pointer from the `iface` struct and then calling into the `fun` array.

```
fake iface:
0x0000:         <fake itab>
0x0008:  0x0000000000000000
              ...
fake itab:
0x1000:  0x0000000000000000
0x1008:  0x0000000000000000
0x1010:  0x0000000000000000
0x1018:        <CFH target>
              ...
CFH target:
0x2000:     <attacker code>
```

Listing 11: Memory layout in preparation for the exploitation of load-based speculative control flow hijack. The attacker fake `iface` starts at offset 0x0. The fake `itab` prepared by the attacker starts at offset 0x1000. The control flow hijack target is located at offset 0x2000.

In preparation for exploitation, the attacker ensures that the memory layout of the target program contains a pattern similar to that shown in Listing 11. Assuming that the attacker wants to speculatively redirect the control flow to address `0x2000`, the attacker creates a fake `itab` structure (in the example at `0x1000`) such that the first entry in the `fun` pointer array points to the desired target. Then the attacker creates a fake `iface` structure (in the example at offset `0x0`) such that the `tab` pointer points to the aforementioned `itab` structure. With the memory thus prepared, the attacker supplies the index into the array such that the resulting address (the base address plus index multiplied by the size of an `iface` structure) equals the fake `iface` structure (`0x0` in our example). With the index thus set the program will call the `runtime.panicindex` function; however if the conditional jump of the bounds check is mispredicted, the dereference and subsequent indirect call will take place transiently. Note that, contrary to the case studies in Section 3.1 and Section 3.2, the attack is not necessarily "single shot": if the program calls `recover`, the attacker might be able to execute the vulnerable sequence multiple times.

We prototype the attack to evaluate its effectiveness in a proof of concept. The proof of concept only aims to establish the feasibility of the attack: in particular we do not integrate into an end-to-end attack and refer to Section 3.1.4 for cache eviction and speculative ROP. The PoC contains the pattern of Listing 9 called in a loop to train the pattern history table and ensure that the bounds check conditional jump as strongly non-taken. The index used to access the array in the loop is in bounds during the training phase and is then set to the target index computed as described above in the last iteration.

To verify whether speculative control flow hijack takes place, we instrument the program to read PMCs during the execution of the loop, and set the speculative control flow hijack target to contain a speculation marker. The `runtime.panicindex` function is modified to read and persist PMC values for each execution.

This instrumentation permits us to verify that speculative control flow hijack indeed takes place. The success rate is influenced by several factors that we review here. The most relevant factor is the size of the speculation window, which is influenced by how quickly the correct jump target is determined. The speculation window is maximized if the variables used in the compare instruction that drives the jump – especially the array length – are not present in any of the levels of the cache. In order to get empirical evidence of this fact, we instrument the program with a `clflush` instruction right before the array dereference to ensure that the array length is not cached. In practice, an attacker may achieve the same result by performing cache eviction code sequences. However flushing the cache alone does not ensure a high success rate: this is because the array length is stored right after the base address of the array, whose address is loaded into memory as the first instruction of the dereference sequence. We verify that if the two memory locations belong to different cache lines, the speculation window is maximized. Another factor that influences the success rate is whether the target of the speculative control flow hijack is already in the instruction cache. We make sure that this be the case by insert a call to the marker function in the

warm up phase before the loop. We report success rates exceeding 80% (n=1000) when the array length is flushed and is in a separate cache line as the base address on multiple platforms (Xeon CPU E5-2640, Core i7-8650U, Core i7-6700K) and different versions of the Go runtime (1.13.4, 1.12, 1.10.4).

3.3.2. Store-based SPEAR speculative attack. The second pattern is shown in Listing 12.

```
array[index] = value
...
interface.function()
```

Listing 12: Store-based speculative control flow hijack code pattern.

The pattern consists of a store operation of an attacker-controlled value at an attacker-controlled location into an array. The elements stored in the array must permit storage of a pointer. Smaller sizes would permit partial control over the speculative control flow hijack target. The pattern requires that the array store be followed by an interface call. The interface call does not need to be related to the array. It only needs to be in close proximity of the store operation so that it may still be speculatively executed. This pattern does not require any ability to perform preparatory store operations in the memory space of the target program. The pattern makes use of store-to-load forwarding, since the store in the array is used to (speculatively) overwrite a function pointer which is later (speculatively) loaded and called. This corresponds to the "speculative overwrite of forward edge" variant of a SPEAR attack.

The store part of the pattern consists of a speculative version of a "write-what-where" condition. It may be exploited in several ways to hijack the interface call: the most basic one would be to overwrite the `tab` pointer in the `iface` struct (see Listing 10). However this would either require the attacker to perform a set of preparatory stores identical to those discussed in Section 3.3.1, or it would restrict the freedom of the attacker to choose a target out of the existing interface pointers. Another strategy would be for the attacker to overwrite the `fun` pointer in the `itab` structure directly. These structures are stored in a non-writable virtual memory region. However, given that the store takes place speculatively, the attacker is able to bypass the write restrictions and overwrite the pointer. Therefore, we choose to prototype this simpler and more effective variant.

Exploitation proceeds as follows: at first the attacker speculatively overwrites the `fun` pointer in the `itab` of the interface that is later dereferenced. This is achieved, as the attacker controls value and index. The former is set to the address of the desired speculative control flow hijack target; the latter is set such that base array and index multiplied by the size of the array elements add up to the address of the `fun` pointer to be overwritten. As in the previous section, with the index thus set the program will panic; however if the bounds check is mispredicted, the store-to-load forwarding and subsequent indirect call will take place, achieving speculative control flow hijack.

We prototype the attack to evaluate its effectiveness employing a similar instrumentation as the previous section, with PMCs and speculation markers employed to identify successful runs, and a loop to set the predictor state. The success rate is similarly influenced by ensuring that the variables driving the conditional branch are not cached, and that the speculative control flow hijack target is in cache. Under these conditions, we report success rates exceeding 80% (n=1000) on the same platforms listed in Section 3.3.1.

4. Mitigations

In this section, we implement and analyze serializing-based (lfence) and masking-based mitigations for SPEAR-architectural attacks (SSP) in Section 4.1 and SPEAR-speculative ones (Go) in Section 4.2. We show that in both cases the masking-based solution results in a low overhead. Finally, we discuss possible mitigations for GCC VTV case in Section 4.3.

4.1. Mitigations for SSP

We investigate two possible mitigations for the SPEAR-architectural attack against SSP. A serializing instruction such as `lfence` can be inserted after loading the canary in the epilogue instrumentation, thereby ensuring that the comparison can only lead to a short enough speculation window. Alternatively, the return value can be masked architecturally with a generated value that is set to `0` when the check fails (the canary is corrupted), and all ones when it passes, as shown in Listing 13.

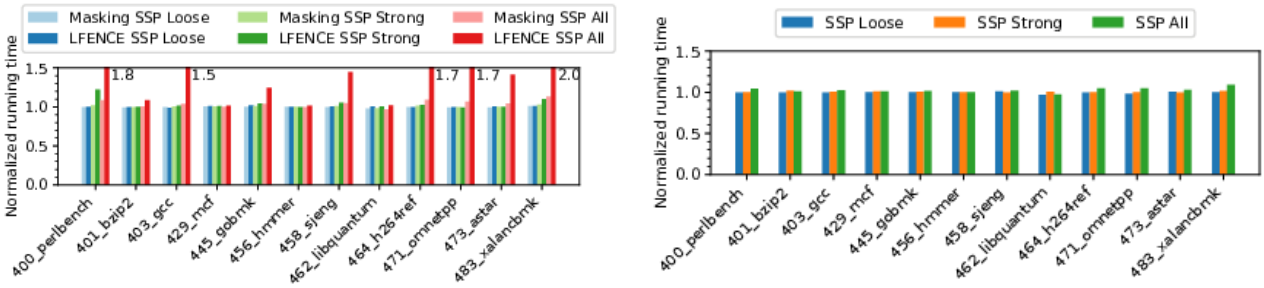
```
1  mov rax, QWORD[fs:0x28]
2  mov rcx, QWORD[stack_canary]
3  xor rdx, rdx
4  cmp rax, rcx
5  setne dl
6  add rdx, 0xffffffffffffffff
7  and QWORD[rsp + 8], rdx
```

Listing 13: Masking mitigation sequence; `rax` contains global canary value and `rcx` contains the stack canary; `rsp + 8` points to the return address.

We implement both mitigations as compiler passes in `clang+llvm`. The masking-based mitigation implementation is an extension of Speculative Load Hardening [12]. SSP is architecture specific, therefore our solution is built for `x86_64` Linux systems. We run the SSP mitigations benchmarking on Intel(R) Core(TM) i7-6700K CPU @ 4.00GHz. We measure the normalized runtime of both *return address masking* and `lfence` on SPECint CPU 2006. The normalized runtime is computed as runtime over the baseline runtime constituted by execution with SSP Disabled. For reference, we additionally plot the normalized runtime for all existing SSP implementations, SSP Loose (`-fstack-protector` flag), SSP Strong (`-fstack-protector-strong` flag), and SSP All (`-fstack-protector-all` flag).

The results are shown in Figure 3a. The `lfence` mitigation shows a high overhead in 9 out of 12 benchmarks, the highest being 100%, in the SSP All case with `xalancbmk`. Return address masking incurs a significantly lower, albeit still not negligible performance penalty, reaching a maximum of 13% for the same benchmark.

Based on this evaluation, we find the return address masking mitigation to be viable and superior to the `lfence` mitigation: the overhead of vanilla SSP (shown



(a) SSP with speculative bypass mitigations.

(b) Vanilla SSP.

Figure 3: Overhead computed as normalized runtime over SSP Disabled baseline.

in Figure 3b on SPECint CPU 2006 is at most 9%, in the case of SSP All on `xalancbmk`). In addition, we note that most Linux distributions either use the SSP Loose or SSP Strong options, both of which incur a low overhead on all SSP benchmarks: we record a maximum of 2.1% overhead over the SSP Disabled baseline. With return address masking, the maximum overhead becomes 2.7% over the SSP Disabled baseline. We conclude that return address masking does not impose a significant overhead with the most commonly used SSP compiler options.

4.2. Mitigations for the Go compiler

We investigate possible mitigations for the SPEAR-speculative attack on Go. The mitigations consist of two different compiler passes that ensure that the vulnerability is no longer exploitable. The first is based on `lfence`, whereas the second is based on branchless index masking sequences. As part of responsible disclosure we have notified the Go team, who have implemented 2 families of compiler-based mitigations for Spectre, namely, index masking (through the `-spectre=index` compiler switch) and retpoline (through the `-spectre=ret` compiler switch).

The first mitigation consists of adding an `lfence` instruction after the `cmp` instruction in the sequence that implements the `IsInBounds` meta-operation. With reference to Listing 8, the `lfence` instruction is inserted after the `cmp` on line 2. The insertion ensures that all prior instructions have completed, which means that there will be no misprediction of the branch target and any out-of-bound access will result in a panic with no transient execution. The instruction is added explicitly in the pass that translates the AST into SSA form by defining a new `Lfence` meta-operation and adding it after each `IsInBounds` operation. We ensure that the operation is neither reordered nor eliminated.

The second mitigation we investigate entails the addition of an appropriate masking sequence that ensures that the index is set to a "safe" value in case of out-of-bounds accesses. The masking sequence amounts to a no-op in case the access is in bounds by performing an `and` operation on the index with a sign extended -1 mask. If the access is not in bounds, in our implementation, the masking operation forces an access of the element at index 0 in the array by performing an `and` operation on the index with a 0 mask. We can see the masking sequence in Listing 14: after the usual `cmp` and `jmp` instructions, length and index are subtracted in order to set the carry flag. Then, the `sbb` instruction is used to set a register to -1 in case of an in-bounds access or 0 otherwise. The array is subsequently accessed after performing an `and` operation on the index with the mask thus obtained. The pattern might be further optimized by using the `cmp` instruction of the bounds check to set the carry flag. This, however, is not always possible since the compiler will use a compare instruction with an immediate whenever possible. The immediate can only be the second source operand, forcing the direction of the comparison instruction. For the sake of simplicity we therefore rely on an extra subtraction operation. The masking instruction sequence is added by defining three new meta-operations – `OpMaskStep1`, `OpMaskStep2` and `OpMaskStep3` – which are later lowered into a `sub`, `sbb` and `and` instruction, respectively.

We measure the overhead of both mitigations by building the Go runtime version 1.12.0 and running the full benchmark suite. We run the experiments on a 40-core Xeon E5-2640 machine with 64 GiB of RAM. Figure 4 displays the empirical cumulative distribution function of the overhead of each of the two mitigation strategies. We can see how the `lfence`-based approach incurs a high overhead (143% mean and 84% median) due to the fact that `lfence` will terminate any speculative execution and thus severely curtail the instruction throughput. On the other hand, the masking approach shows a much lighter overhead (12% mean and 6% median) since the instructions involved are simple and do not cause any memory-related operation.

```
1  cmp   rcx, rdx
2  jae   <raise-panic>
3  mov   rbx, rdx
4  sub   rdx, rcx
5  sbb   rcx, rcx
6  and   rcx, rbx
7  shl   rcx, 0x4
8  mov   rax, [rax+rcx*1]
```

Listing 14: Masking mitigation sequence; `rdx` contains the index and `rcx` contains the length of the array and `rax` contains the base address of the array.

4.3. Mitigations for GCC VTV

The same mitigations considered in Section 4.1 and Section 4.2 work in the GCC VTV use case. Serializing mechanisms (e.g., lfence) are a viable solution, albeit likely with high overhead. A branchless masking

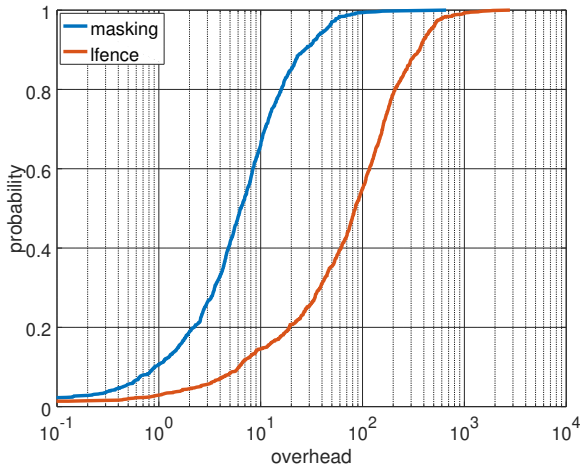


Figure 4: Empirical CDF of the logarithm of the overhead percentage for the considered mitigations. Overhead data is gathered by running the full set of benchmarks of the Go runtime version 1.12.0.

solution or retpoline could also be used in this context for what we expect to have better performance, however we did not implement these.

We believe a better approach, from a performance point of view, for GCC VTV would be a re-design with the principles observed for LLVM CFI described in Section 3.2.2 where the metadata and the pointer that have to be verified co-exist within the same cache line. This condition prevents the attacker to achieve the correct data eviction and, consequently, the speculation window to perform the attack is too small.

5. Related work

5.1. Speculative execution attacks

Transient execution attacks can be subdivided into two main categories: fault-based and speculation-based attacks [11]. The speculation-based, or Spectre-family, attacks comprise those leveraging microarchitectural components such as the Pattern History Table (PHT) for Spectre v1 [23], the Branch Target Buffer for Spectre v2, the Return Stack Buffer (RSB) for Ret2Spec [29] and Spectre returns [25]. Both BTB and RSB attacks are cases of speculative control flow hijacks, i.e., they provide the ability for an attacker to steer speculative execution to an arbitrary location. Varied and powerful attacks leveraging the BTB for speculative control flow hijacks have been demonstrated, in combination with port contention-based, instruction cache-based, or BTB-based side channels [32], [10]. In contrast, this paper focuses on SPEAR attacks, where the speculative control flow hijack step is based on architecturally visible control-flow influencing instructions. In Spectre v1.1 [21], Kiriansky and Waldspurger point out that speculative overwrites of backward edges lead to speculative control flow hijacks. The SPEAR class includes their finding and complements it with three new types, including the architectural overwrite case. Also, we demonstrate practical use cases on Go memory safety and GCC VTV, and a full working attack on SSP.

In practice, BTB gadgets are hard to find, thus attacks have only been shown to be practical if the gadget is injected (e.g., by loading attacker-controlled eBPF bytecode into the kernel). In the SPEAR attacks, we reuse gadgets existing within the victim program. The idea of chaining speculative gadgets in a way similar to ROP was suggested shortly after the first publication of Spectre attacks. Some publications have referred to the same idea [21], [32], the former only briefly mentioning speculative ROP attacks but practical aspects are neither discussed nor experimented on. In contrast, this paper presents a practical case of chaining multiple speculative gadgets to form a cache side-channel send gadget.

Netspectre [40] introduces a victim data eviction technique based on coarse-grained cache eviction. The method, *Thrash+Reload*, is a remote variant of *Evict+Reload* [15]. The attacker starts a large file download from the victim via a network interface. On the victim's side, this action results in victim data eviction with a probability which depends on the file size. *Thrash+Reload* applicability is limited to scenarios where cache thrashing does not compromise the attack: it would for instance be detrimental in our SSP end-to-end exploit where the availability of ROP gadget sequences in the icache is a necessary precondition.

5.2. Speculative execution terminology

Canella et al. present a thorough taxonomy and evaluation of speculative execution variants [11]. In this terminology, the SSP attack falls in the Spectre-PHT, *same address-space, in-place* category, given that the attacker triggers the to-be mispredicted victim path (successful canary check) prior to the attack, to force the target program to self-train. This terminology, however, does not help distinguish the different types of speculative control flow hijacking attacks, provided in our categorization in Figure 2. In the general sense, a Spectre-PHT type misprediction is not required for SPEAR attacks: other types of misprediction leading to the overwrite of a control-flow influencing data (SPEAR-speculative) or other types of misprediction following the overwrite of control-flow influencing data (SPEAR-architectural) are also concerned, which justifies the need for our categorization. For instance, SPEAR-speculative attack instances, such as the Go attacks here, can be classified under the existing Spectre-PHT writing out-of-bounds category. However, the SSP attack cannot be covered in that category, given that no misprediction of a bounds check occurs and the write is not speculative.

Most Spectre attacks including SPEAR rely on a covert channel and/or gadget to achieve information leak. Intel terminology [2] refers to it as *disclosure gadget*. According to their taxonomy based on disclosure gadget location, SPEAR falls under the *cross-domain transient execution attack* category. While useful, this categorization also does not help distinguish between different speculative control flow hijacks.

5.3. Concurrent work

Three recent papers are concurrent to this work and partially relate to it. Goktas et al. [14] demonstrate that

speculative execution attacks can be used to bypass randomization based defenses including ASLR. Their main assumption is the presence of a powerful memory corruption vulnerability, allowing the attacker to overwrite (architecturally) function pointers. In contrast, SPEAR does neither aim to bypass randomization-based defenses, nor does it assume that the attacker has access to such a powerful vulnerability. For instance, in Section 3.1 we assume a vulnerability which cannot be traditionally exploited due to the presence of SSP. Bhattacharyya et al. [9] demonstrate that speculative ROP chains can be mounted in speculative execution attacks by carefully training the BTB (or RSB) to chain multiple speculative ROP gadgets. In contrast, we do not make use of BTB or RSB training to chain gadgets but simply use store-to-load forwarding of the return value on the stack. Finally, Van Bulck et al. [45] demonstrate chaining `pop-ret` instructions in a transient ROP attack triggered by a LVI attack. This is complementary to SPEAR, which explores ROP in the context of speculative execution attacks.

5.4. Mitigations

Since the first speculative execution attacks have been disclosed in early 2018, different mitigations have been proposed to prevent each variant. Some mitigations are introduced at hardware level meanwhile others are software-based. Many of these mitigations target Spectre v2 type of attacks, meanwhile no software-transparent mitigation has been introduced for Spectre v1.

The available software-based Spectre v1 mitigations consist in either deploying a serializing instruction (e.g `lfence`) around each sensitive bounds check or, alternatively, masking the index used for accessing arrays [22], [12], [21], [47].

While `lfence` is an effective mitigation, it incurs huge performance penalties if widely applied. Static analysis tools have been proposed to search for sensitive code patterns. One example is the Linux kernel where vulnerable code is instrumented on a case by case basis either through manual audit or automatic tools (e.g., smatch [5]) detection [6]. The drawback of current available tools is that they target Spectre v1 code patterns such as array-out-of-bounds cases only and therefore are not useful in the general memory corruption case (where an overwrite of a control-flow influencing value can occur for any other mispeculation).

At the hardware level, SpecShield [8] changes microarchitectural handling of loads and prevents forwarding of sensitive data to probable covert channels during transient execution. It proposes three strategies to delay load broadcast to dependent instruction until sensitive load instructions are at the top of the re-order buffer. They demonstrate these techniques can improve performance compared to software barriers.

For Spectre v2 instead, there are software and hardware mitigations. The software mitigation currently available is Retpoline [44]. This mitigation targets indirect calls and indirect jumps and prevents them from being speculatively executed by trapping speculation within a loop. As in the barrier cases for Spectre v1, Retpoline requires code modification and therefore each program has to be recompiled to enforce such mechanism.

On the hardware side, Intel published three major protections: *i*) IBRS [18], which prevents speculation of indirect branches using target values computed using lower privileged predictor modes, *ii*) STIBP [19], which prevents BTB poisoning from sibling threads, and *iii*) IBPB [17], which ensures that code before a barrier does not influence the behavior of the code after. IBRS and IBPB are meant to protect higher privileged code from lower privileged code. The only mitigation that provides protection within the same privilege level is STIBP, which is not enabled by default for performance reasons. None of these Spectre v2 prevention mechanisms apply to SPEAR attacks, given that SPEAR does not use branch target injection.

Finally, Intel announced as part of its Control Flow Enforcement (CET) extension, the future introduction of a new mitigation that will constrain the target of near indirect jumps and calls to only `ENDBRANCH` instructions. Based on the release specifications, these constraints should also apply during speculative execution. Therefore, this mitigation reduces the number of possible gadgets where speculative execution can be redirected to during branch target injection attacks. For SPEAR attacks, this mitigation applies for the forward edge overwrite case, where it should restrict possible speculative control flow hijack targets. For the backward edge case, Intel has implemented a shadow stack which, if adequately enforced during speculative execution, should stop all SPEAR backward edge overwrites.

5.5. Safe speculation designs

In addition to mitigations that aim to protect already existing systems, several new design proposals have been presented for future architectures to prevent speculative execution attacks.

A line of research concentrates on analyzing the data flow within the CPU pipeline and preventing unsafe operations from leaving observable effects upon misprediction. NDA [24] restricts speculative data propagation that follows an unresolved branch (potential control flow misprediction) or unresolved store address (potential memory dependence misprediction). STT [49] selectively forward secrets based on a speculative taint tracking system. Dolma [7] presents a lightweight speculative information flow scheme with secure performance optimizations. All these designs should prevent SPEAR attacks.

Another set of work, instead, proposes new cache designs. InvisiSpec [48] removes cache covert and side channels by confining Unsafe Speculative Loads (USL) into a speculative buffer until the USL is considered safe and the changes can be exposed to the cache hierarchy. In a similar fashion, CleanupSpec [38] prevents the cache side-effects, however, its strategy differs from InvisiSpec because it allows the USL to modify the cache. CleanupSpec applies an Undo operation only when misprediction is detected, therefore limiting performance overhead. Conditional Speculation [27] and Sakalis et al. [39] block during speculation memory accesses that do not hit the L1 cache, as the L1 accesses are safe. Finally, DAWG [20] proposes a mechanism to partition the caches into domains to provide isolation. These cache based defenses stop SPEAR attacks as described in this paper but they do not cover cases of SPEAR where non-cache side channels

are used, such as BTB-based [32] or port contention-based [10].

6. Discussion

Applicability to other use cases. Beyond the highlighted use-cases, SPEAR attacks may be employed against other targets. For example, other memory-safe languages may be targeted with SPEAR attacks to speculatively bypass bounds checks as we show for the Go programming language. Preliminary investigation suggests that this is likely to be possible, since instruction sequences for bounds checks similar to those detailed in Section 3.3 are also present in `Rust` and `Java` (for JITted blocks). We analyze in more detail the Rust use case and report our findings in Appendix A.

Theoretically, any security check that directly or indirectly gates a control flow transfer may be turned into a SPEAR attack. For instance, all the heap hardening mechanisms that verify the integrity of the heap metadata and pointers within `libc` can potentially lead to one of the SPEAR variant through the speculative use of a corrupted data to decide the application control flow. However, as demonstrated in the LLVM CFI case, a case-by-case analysis is necessary to establish whether SPEAR attacks are applicable.

Data leaked in SPEAR-architectural attacks. SPEAR attacks allow an adversary to leak sensitive information from the victim address space. In the case of SSP, we demonstrate that arbitrary memory can be leaked, one byte per iteration. While we can target any memory location, we cannot target data that is not deterministic across runs. In particular, we cannot target to leak the stack canary, given that its value is re-randomized at every program start. We note that SPEAR-speculative attacks do not have this constraint, given that they do not require a program restart.

General applicability of speculative ROP. The speculative ROP and LLC eviction techniques are demonstrated as part of the SSP, SPEAR-architectural overwrite of a backward edge, use case. Nevertheless these techniques are generally applicable for the exploitation of other SPEAR use cases, with exploitability always depending on the scenario at hand. For the general forward edge cases, we note that this requires, as in classical ROP attacks, a technique known as a stack pivot, which consists in the attacker setting up a fake return stack somewhere under its control in memory, and having the first control flow hijack point to an instruction setting the stack pointer to that address (for instance, the `push rax; pop rsp; ret` stack pivot gadget). Using the Speculator tool, we verify that such stack pivots do work for SPEAR-architectural as well as SPEAR-speculative attacks.

General applicability of LLC eviction. In our end-to-end attack over SSP, we employ a new more precise LLC eviction technique which is described in details in 3.1.2. The necessity for developing our own, more precise, LLC eviction technique stems from the fact that our attack poses two additional requirements. The first is the fact that we require the eviction process to be very selective, since we cannot allow elements such as the addresses injected on the stack or the gadgets code to be evicted because that will stall speculative execution and prevent the completion of the attack. The second is that the eviction process needs to complete within a short amount of time to avoid the scenario where the line containing the canary is first evicted and then re-cached by the natural execution of the victim while the eviction process completes. With our technique, we can keep the number of possible cache-sets as small as possible and therefore minimize the length of the eviction process. We explore an existing LLC flush method discovered by Oren et al. [35] which could potentially fit the second requirement. However, we conclude that this method is too intrusive in a setting where the attacker relies on cached data and code (victim secret, ROP gadgets) available in the speculation window.

7. Conclusion

In this paper, we investigate variants of speculative control flow hijacking attacks, called SPEAR, that exploit and bypass current mitigations against classic memory corruption vulnerabilities to leak information from local processes. With SPEAR, we show that Spectre-like vulnerabilities drastically increase attack vectors for local attackers. Therefore, they force not only the creation of new mitigations but also the re-design of previously deployed protections. In this work, we present attacks against stack canaries, CFI and memory-safe languages. We demonstrate a practical attack against SSP buffer overflow mitigations and proof-of-concept implementations against GCC VTV and Go's runtime. We show the use of multiple ROP gadgets and details on how to use LLC eviction without knowledge of physical addresses in the context of SPEAR attacks. Finally, we discuss how SPEAR attacks can be mitigated and report our performance results.

Disclosure

We submitted the PoC exploits and our findings to the Go security team on November 22nd, 2019. As a result of our notification, the Go security team has deployed hardening measures (index masking and retpoline) which were released in Go 1.15.

Acknowledgement

We would like to thank Russ Cox, Matthias Neugschwandtner, the anonymous reviewers, and our shepherd for their valuable comments on an earlier draft of this paper.

This work was partially-supported by National Science Foundation under grant CNS-1703454, and ONR under the "In Situ Malware" project.

References

[1] "pagemap: do not leak physical addresses to non-privileged userspace," https://lwn.net/Articles/642074/.

[2] "Refined speculative execution terminology," https://software.intel.com/security-software-guidance/insights/refined-speculative-execution-terminology.

[3] "ROPgadget," http://shell-storm.org/project/ROPgadget.

[4] "CVE-2004-0597," https://nvd.nist.gov/vuln/detail/CVE-2004-0597, 2004.

[5] https://repo.or.cz/w/smatch.git, 2018.

[6] "The Linux Kernel user's and administrator's guide," https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/spectre.html, 2019.

[7] "DOLMA: Securing speculation with the principle of transient non-observability," in *30th USENIX Security Symposium (USENIX Security 21)*. Vancouver, B.C.: USENIX Association, Aug. 2021. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/loughlin

[8] K. Barber, A. Bacha, L. Zhou, Y. Zhang, and R. Teodorescu, "Specshield: Shielding speculative data from microarchitectural covert channels," in *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2019, pp. 151–164.

[9] A. Bhattacharyya, A. Sánchez, E. M. Koruyeh, N. Abu-Ghazaleh, C. Song, and M. Payer, "Specrop: Speculative exploitation of {ROP} chains," in *23rd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2020)*, 2020, pp. 1–16.

[10] 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.*, 2019, pp. 785–800. [Online]. Available: https://doi.org/10.1145/3319535.3363194

[11] 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 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 249–266. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/canella

[12] C. Carruth, "Speculative load hardening," https://llvm.org/docs/SpeculativeLoadHardening.html, 2018.

[13] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Jump over aslr: Attacking branch predictors to bypass aslr," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 2016, p. 40.

[14] E. Goktas, K. Razavi, G. Portokalidis, H. Bos, and C. Giuffrida, "Speculative Probing: Hacking Blind in the Spectre Era," in *CCS*, Nov. 2020. [Online]. Available: https://download.vusec.net/papers/blindside_ccs20.pdf

[15] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive last-level caches," in *Proceedings of the 24th USENIX Conference on Security Symposium*, ser. SEC'15. USA: USENIX Association, 2015, p. 897–912.

[16] R. Hund, C. Willems, and T. Holz, "Practical timing side channel attacks against kernel space aslr," in *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 191–205.

[17] Intel, "Deep dive: Indirect branch predictor barrier," https://software.intel.com/security-software-guidance/insights/deep-dive-indirect-branch-predictor-barrier, 2018.

[18] ——, "Deep dive: Indirect branch restricted speculation," https://software.intel.com/security-software-guidance/insights/deep-dive-indirect-branch-restricted-speculation, 2018.

[19] ——, "Deep dive: Single thread indirect branch predictors," https://software.intel.com/security-software-guidance/insights/deep-dive-single-thread-indirect-branch-predictors, 2018.

[20] V. Kiriansky, I. Lebedev, S.Amarasinghe, S. Devadas, and J. Emer, "Dawg: A defense against cache timing attacks in speculative execution processors," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 974–987.

[21] V. Kiriansky and C. Waldspurger, "Speculative Buffer Overflows: Attacks and Defenses," https://people.csail.mit.edu/vlk/spectre11.pdf, 2018.

[22] P. Kocher, https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html, 2018.

[23] P. Kocher, 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 *IEEE Symposium on Security and Privacy*, 2018.

[24] 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," *Commun. ACM*, vol. 63, no. 7, p. 93–101, Jun. 2020. [Online]. Available: https://doi.org/10.1145/3399742

[25] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre returns! speculation attacks using the return stack buffer," in *USENIX Workshop On Offensive Technologies*, 2018.

[26] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 147–163. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/kuznetsov

[27] P. Li, L. Zhao, R. Hou, L. Zhang, and D. Meng, "Conditional speculation: An effective approach to safeguard out-of-order execution against spectre attacks," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 264–276.

[28] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, ser. SP '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 605–622. [Online]. Available: https://doi.org/10.1109/SP.2015.43

[29] G. Maisuradze and C. Rossow, "Ret2spec: Speculative execution using return stack buffers," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: ACM, 2018, pp. 2109–2122. [Online]. Available: http://doi.acm.org/10.1145/3243734.3243761

[30] A. Mambretti and et al., "Speculator," https://github.com/ibm-research/speculator/wiki, 2019.

[31] A. Mambretti, M. Neugschwandtner, A. Sorniotti, E. Kirda, W. Robertson, and A. Kurmus, "Speculator: A tool to analyze speculative execution attacks and mitigations," in *Proceedings of the 35th Annual Computer Security Applications Conference*, ser. ACSAC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 747–761. [Online]. Available: https://doi.org/10.1145/3359789.3359837

[32] A. Mambretti, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, and A. Kurmus, "Two methods for exploiting speculative control flow hijacks," in *13th USENIX Workshop on Offensive Technologies (WOOT 19)*. Santa Clara, CA: USENIX Association, Aug. 2019. [Online]. Available: https://www.usenix.org/conference/woot19/presentation/mambretti

[33] C. Maurice, N. L. Scouarnec, C. Neumann, O. Heen, and A. Francillon, "Reverse engineering intel last-level cache complex addressing using performance counters," in *Research in Attacks, Intrusions, and Defenses - 18th International Symposium, RAID 2015, Kyoto, Japan, November 2-4, 2015, Proceedings*, 2015, pp. 48–65. [Online]. Available: https://doi.org/10.1007/978-3-319-26362-5_3

[34] R. Mcilroy, J. Sevcik, T. Tebbi, B. L. Titzer, and T. Verwaest, "Spectre is here to stay: An analysis of side-channels and speculative execution," 2019.

[35] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, "The spy in the sandbox: Practical cache attacks in javascript and their implications," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 1406–1418. [Online]. Available: https://doi.org/10.1145/2810103.2813708

[36] C. Robertson and et al., "C++ developer guidance for speculative execution side channels," https://docs.microsoft.com/en-us/cpp/security/developer-guidance-speculative-execution, 2018.

[37] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Trans. Inf. Syst. Secur.*, vol. 15, no. 1, pp. 2:1–2:34, Mar. 2012. [Online]. Available: http://doi.acm.org/10.1145/2133375.2133377

[38] G. Saileshwar and M. K. Qureshi, “Cleanupspec: An "undo" approach to safe speculation,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO ’52. New York, NY, USA: Association for Computing Machinery, 2019, p. 73–86. [Online]. Available: https://doi.org/10.1145/3352460.3358314

[39] C. Sakalis, S. Kaxiras, A. Ros, A. Jimborean, and M. Själander, “Efficient invisible speculative execution through selective delay and value prediction,” in *Proceedings of the 46th International Symposium on Computer Architecture*, ser. ISCA ’19. New York, NY, USA: Association for Computing Machinery, 2019, p. 723–735. [Online]. Available: https://doi.org/10.1145/3307650.3322216

[40] M. Schwarz, M. Schwarzl, M. Lipp, J. Masters, and D. Gruss, “Netspectre: Read arbitrary memory over network,” in *Computer Security – ESORICS 2019*, K. Sako, S. Schneider, and P. Y. A. Ryan, Eds. Cham: Springer International Publishing, 2019, pp. 279–299.

[41] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, “On the effectiveness of address-space randomization,” in *Proceedings of CCS 2004*, B. Pfitzmann and P. Liu, Eds. ACM Press, Oct. 2004, pp. 298–307.

[42] L. Szekeres, M. Payer, T. Wei, and D. Song, “SoK: Eternal War in Memory,” in *IEEE Symposium on Security and Privacy*, 2013.

[43] C. Tice, G. Inc, T. Roeder, G. Inc, P. Collingbourne, G. Inc, S. Checkoway, Úlfar Erlingsson, G. Inc, L. Lozano, G. Inc, and G. Pike, “Enforcing forward-edge control-flow integrity,” in *in GCC & LLVM. In 23rd USENIX Security Symposium (USENIX Security 14) (Aug. 2014), USENIX Association*, 2014, pp. 941–955.

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

[45] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yuval, 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.

[46] V. van der Veen, N. dutt Sharma, L. Cavallaro, and H. Bos, “Memory errors: The past, the present, and the future,” in *Proceedings of the 15th International Conference on Research in Attacks, Intrusions, and Defenses*, ser. RAID’12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 86–106. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-33338-5_5

[47] D. Williams, “Sanitize speculative array de-references,” https://lore.kernel.org/patchwork/patch/874621/, 2018.

[48] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. W. Fletcher, and J. Torrellas, “Invisispec: Making speculative execution invisible in the cache hierarchy,” in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-51. IEEE Press, 2018, p. 428–441. [Online]. Available: https://doi.org/10.1109/MICRO.2018.00042

[49] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. W. Fletcher, “Speculative taint tracking (stt): A comprehensive protection for speculatively accessed data,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO ’52. New York, NY, USA: Association for Computing Machinery, 2019, p. 954–968. [Online]. Available: https://doi.org/10.1145/3352460.3358274

[50] G. P. Zero, “Reading privileged memory with a side-channel,” https://googleprojectzero.blogspot.ch/2018/01/reading-privileged-memory-with-side.html, 2018.

Appendix A.
SPEAR attack against Rust bounds checking

The implementation of Rust panicking mechanism is abundant of SPEAR speculative control flow hijacking patterns similar to those discussed in the Go case study (Section 3.3). Here, we examine the safety features employed by Rust for index expressions and demonstrate a proof of concept SPEAR attack against out of range access hardening.

In Rust, memory safety for index expressions is established during Mid-level Intermediate Representation (MIR) building, with static and dynamic arrays, slices and strings being subject to sanitization. At compiler level, index expressions are instrumented with bounds checks which prevent out of range access. However, similarly to the case of Go, CPU misprediction of bounds check outcome leads to speculative out of bounds access.

```
1    const PADDING_SIZE: usize = 7;
2    pub type Fptr = fn(u64) -> u64;
3
4    pub struct Data {
5      _padding: [u64; PADDING_SIZE],
6      buf: Box<[Fptr]>,
7    }
8    let data: Data = Box::new(Data { ... });
9    data.buf[index]();
```

Listing 15: SPEAR speculative control flow hijacking target in Rust. The `index` value is attacker controlled. We assume that the attacker writes the CFH target in memory prior to the attack.

```
1       mov rsi, [index]
2       mov rax, [buf len]
3       cmp rsi, rax
4       jle ok
5       call <core::panicking::panic_bounds_check>
6    ok:
7       mov rcx, [buf]
8       mov rdx, [index]
9    ; Calls function pointer when index is in bounds
10      call QWORD[rcx+rdx*8]
```

Listing 16: Disassembly of Rust index expression bounds check instrumentation.

The attack targets the array index access followed by an indirect call in Listing 15 at line 9. To trigger the panicking system, the array is accessed with an attacker controlled index which is out of bounds. Rust MIR instruments the array index access with a bounds check. We analyze the index expression bounds check instrumentation at Assembly level in Listing 16. The instrumentation starts with array length loading and comparison against the attacker provided index, at line 3. Depending on the comparison outcome, the execution proceeds with accessing the array element requested or aborting in case of in-bounds requirement violation.

When the comparison between index and length is slow (due to uncached operands), the CPU may mispredict the result and continue execution speculatively, on the wrong path.

In the PoC, the victim data structure is chosen such that the array length can be evicted prior to the attack. The array length is stored together with the array data pointer in `buf`. At line 8 the `Data` object is initialized using `Box`, therefore the object is placed on heap. This avoids Rust default stack allocation which lowers the array length eviction success. Furthermore, the eviction may affect attack critical data, like the `buf` data pointer. In the PoC, `Data` uses a large enough padding so that the array length and the data pointer land on different cache lines.

The `buf` length eviction triggers mispeculation of the jump direction taken (Listing 16, line 4). Inside the speculation window, an out of bounds array access with the attacker controlled index leads to reading a function pointer from an attacker-owned memory area. Subsequently, the attacker controlled function pointer is the `call` instruction destination (line 10), therefore facilitating speculative execution of attacker chosen code (in this case, a speculation marker). Despite of the CPU rolling back the speculative execution effects on registers and memory, we use Intel Performance Monitoring Counters for counting speculation marker hits. We carry out the experiments on an Intel Skylake machine running Ubuntu 18.04. We measure an overall success rate of 90% (n=1000) for the SPEAR attack against Rust bounds checking mechanism. As for Go and GCC VTV, this success rate refers to the hijack phase only.

Appendix B.
Further Code Snippets

```
1  ;Copy of Target Value
2      mov rax, [orig_target]
3      mov QWORD[stored_target], rax
4
5  ;Architectural Overwrite
6  ; (Attacker Controlled)
7      mov rax, QWORD[hijacked_target]
8      mov QWORD[target], rax
9
10 ;Evict Target Value Copy
11     clflush [stored_target]
12     lfence
13
14 ;Forward Edge Integrity Check
15 ; (Speculation Trigger)
16     mov rax, QWORD[target]
17     cmp rax, QWORD[stored_target]
18     jne my_exit
19
20 ;Forward Edge Hijack
21     call QWORD[target]
```

Listing 17: Architectural forward edge overwrite.

```
1  ;Speculative execution trigger
2      ...
3
4  ;Speculative Overwrite
5  ; (Attacker Controlled)
6      mov rax, QWORD[hijacked_target]
7      mov QWORD[target], rax
8
9  ;Forward Edge Hijack
10     call QWORD[target]
```

Listing 18: Speculative forward edge overwrite.