JumpSwitches: Restoring the Performance of Indirect Branches In the Era of Spectre

Nadav Amit, *VMware Research;* Fred Jacobs, *VMware;* Michael Wei, *VMware Research*

https://www.usenix.org/conference/atc19/presentation/amit

This paper is included in the Proceedings of the
2019 USENIX Annual Technical Conference.

July 10–12, 2019 • Renton, WA, USA

ISBN 978-1-939133-03-8

JumpSwitches: Restoring the Performance of Indirect Branches In the Era of Spectre

Nadav Amit
VMware Research

Fred Jacobs
VMware

Michael Wei
VMware Research

Abstract

The Spectre family of security vulnerabilities show that speculative execution attacks on modern processors are practical. Spectre variant 2 attacks exploit the speculation of indirect branches, which enable processors to execute code from arbitrary locations in memory. Retpolines serve as the state-of-the-art defense, effectively disabling speculative execution for indirect branches. While retpolines succeed in protecting against Spectre, they come with a significant penalty — 20% on some workloads.

In this paper, we describe and implement an alternative mechanism: the JumpSwitch, which enables speculative execution of indirect branches on safe targets by leveraging indirect call promotion, transforming indirect calls into conditional direct calls. Unlike traditional inlining techniques which apply call promotion at compile time, JumpSwitches aggressively learn targets at runtime and leverage an optimized patching infrastructure to perform *just-in-time* promotion without the overhead of binary translation.

We designed and optimized JumpSwitches for common patterns. If a JumpSwitch cannot learn safe targets, we fall back to the safety of retpolines. JumpSwitches seamlessly integrate into Linux, and are evaluated in Linux v4.18. We show that JumpSwitches can improve performance over retpolines by up to 20% for a range of workloads. In some cases, JumpSwitches even show improvement over a system without retpolines by directing speculative execution into conditional direct calls just-in-time and reducing mispredictions.

1 Introduction

Spectre is a family of security vulnerabilities first disclosed to the public in January 2018 [38]. Spectre manipulates a processor to speculatively execute code which leaves side-effects that can be observed even after a processor decides the speculation is incorrect and reverses its execution. Spectre attacks have shown that it is practical to extract data which leaks from these observable side-effects, known as side-channels. As a result, both processor manufacturers and software developers have sought to defend against Spectre-style attacks by restricting speculative execution.

In order to execute a Spectre attack, malicious code must be able to control what the processor speculatively executes. While several vectors of attack exist, indirect branches, which enable a processor to execute code from arbitrary locations in memory have been shown to be especially vulnerable. Variant 2 of Spectre, known as “Branch Target Injection” [38], leverages indirect branches by causing the processor to mispredict the branch target and speculatively execute code designed to leave side-effects. To mitigate against this attack, a code sequence known as a *retpoline* [49] was developed, which is used in software and throughout the OS kernels today. Retpolines work by effectively disabling speculative execution of indirect branches. Instead of allowing the processor to speculate on the target of an indirect branch, retpolines force the processor to speculatively execute an infinite loop until the target is known. This is achieved by leveraging return (or `RET`) instructions, which are speculatively executed using a different mechanism than indirect branches.

While retpolines are effective in mitigating against Spectre variant 2, preventing speculative execution of indirect branches comes at a significant cost: in some benchmarks, we observe up to a 20% slowdown due to retpolines, making Spectre variant 2 one of the most expensive Spectre attacks to defend against. Unfortunately, indirect branches are used extensively in software: they are the basis for software indirection, they enable object-oriented constructs such as virtual functions and they are present throughout the OS kernels to enable their modular design. Spectre has made optimizing indirect branches in software more important than ever, since Spectre vulnerable hardware is no longer able to do so.

This paper describes our implementation of an alternative mechanism, the JumpSwitch, which leverages a technique known as indirect call promotion to transform indirect calls into conditional direct calls. Indirect call promotion is often employed by compilers [6, 9] to take advantage of profiling data, or binary translators and JIT engines to avoid expensive lookups in code caches [20]. These techniques alone, however, are insufficient for OS kernels. Mechanisms such as kernel address space layout randomization (KASLR) [45] dynamic modules and JITed code (e.g., eBPF) make it difficult to reliably determine branch targets. Compiler-based approaches such as PDO/FDO “lock” the results in a binary, requiring recompilation for best performance and preventing regressions [40] when the workload changes [54], which may be untenable as most distributions package kernels as binaries. Binary translation of the kernel incurs significant overheads while restricting functionality, obviating the benefits of indirect call promotion [27].

Building on this observation, JumpSwitches learn execution targets either statically at compile-time or dynamically at runtime and promote indirect calls using *just-in time promotion*, learning targets using a lightweight mechanism as the kernel runs. If correct targets cannot be learned, JumpSwitches fall back to using retpolines. JumpSwitches are fully integrated with the Linux kernel's build system.[1] Through the Linux build infrastructure, we are able to seamlessly update the kernel to leverage JumpSwitches without source code changes. However, to further improve performance, we provide more advanced JumpSwitches known as *semantic JumpSwitches* which are able to take advantage of the rich semantic information available within the kernel such as process tables, and we provide five different types of JumpSwitches tailored for different use cases.

Our evaluation of JumpSwitches shows a performance improvement over retpolines in a variety of workloads by up to 20%. In some cases, we are able to even show improvement on a system without retpolines, so JumpSwitches are useful even if Spectre variant 2 is mitigated in hardware. Since JumpSwitches are implemented within the kernel, they may potentially benefit every workload running on Linux.

While conditional direct calls are vulnerable to Spectre variant 1 ("Bounds Check Bypass") [30,39,49], another Spectre vulnerability which retpolines do defend against, other less expensive mechanisms which JumpSwitches are compatible with, such as static analysis and selective masking/serialization are typically used to defend against Spectre variant 1 [12]. Notably, future "Spectre safe" hardware which do not need retpolines will still be vulnerable to Spectre variant 1 [46] and require Spectre variant 1 mitigations. JumpSwitches provide the same level of safety as this future hardware.

This paper makes the following contributions:

- We explore Spectre and the impact of current and future mitigations on the performance of indirect calls. (§2.1)
- We explore previous work for indirect call promotion and describe how JumpSwitches revisit indirect call promotion in the era of Spectre. (§2.2, §2.4).
- We describe our implementation of JumpSwitches (§3) which features:
  - Five types of JumpSwitches, which are optimized for the common case (generic JumpSwitches) and special use cases (semantic JumpSwitches) (§3.1).
  - A mechanism for learning which updates JumpSwitches outside the path of execution (§3.2).
  - A integration with the Linux kernel which leverages semantic information within the kernel (system calls, process information, `seccomp`) (§3.4).
- We evaluate JumpSwitches and show that they improve performance in real-world benchmarks by up to 20% (§4).
- We conclude with remarks about the future of speculative execution (§5).

[1]We have submitted upstream patches to the Linux kernel, with positive feedback from the Linux community.

2 Indirect Branches

Indirect branches are a basic type of processor instruction which enable programs to dynamically change what code is executed by the processor at runtime. To execute an indirect branch, the processor computes a *branch target*, which determines what instruction to execute after the indirect branch. Since targets are dynamically computed, indirect branches can execute code depending on what data is present. This indirection is most frequently used to enable polymorphism. For example, in C++, different functions may be executed depending on an object's type, or in Linux, different functions may be executed depending on whether an address is IPv4 or IPv6. To enable this functionality, a compiler will generate an indirect branch which computes the correct function as a branch target based on the object's type. Indirect branches are not limited to object-oriented code: they are used by many constructs such as callbacks and jump tables as well.

The use of indirect branches, however, poses a significant problem for modern processors which are heavily pipelined and require a constant instruction stream to achieve maximum performance. With an indirect branch, the processor cannot fetch the next instruction until the branch target is computed, resulting in pipeline stalls that significantly degrade performance. To eliminate these stalls, processors leverage *speculation*, which guesses the branch target and rolls back executed instructions if the guess is later determined to be incorrect. In many processors, speculation is done through a *branch target buffer* (BTB), which serves as a cache for previous branch targets of indirect branches.

The Spectre family of attacks observed that rolling back speculative execution can be incomplete. For example, speculative execution of privileged code could leave memory as a result of that execution in the processor's caches. Timing attacks can be used by an unprivileged process to determine what memory was fetched, resulting in a data leak. Spectre variant 2, known as "Branch Target Injection" specifically targeted indirect branches and the BTB [30,38]. In variant 2, either training the predictor to speculatively execute a malicious branch target or causing a collision in the BTB (which uses a subset of address bits for performance) resulting in an indirect branch speculatively executing code which leaves side-effects in the processor cache. This work focuses on variant 2, and when we refer to Spectre we refer to variant 2 of the attack.

In the next sections, we discuss both current and future mitigations for Spectre and how performance is affected. We then discuss software mechanisms currently used to optimize indirect branches, and present how JumpSwitches enable dynamically optimization of indirect branches while mitigating

```
1  call %rax  |                call rax_retpoline
              |
2             |  rax_retpoline:
3             |                call target
4             |  capture:      pause
5             |                lfence
6             |                jmp capture
7             |  target:       mov %rax, [%rsp]
8             |                ret
```

Figure 1: Retpolines. Unsafe indirect branches on the left are replaced with a call (line 1) to a retpoline thunk (lines 2-8).

Spectre style attacks. For the rest of this paper, we focus on the Intel x86 architecture. We believe that JumpSwitch can be applied to other Spectre vulnerable architectures, such as AMD, ARM and IBM.

2.1 Spectre Mitigations

Mitigations for Spectre can be categorized into three general categories: software, hardware microcode and hardware.

Software mitigations. Retpolines [49] are the currently preferred method for mitigating Spectre. Retpolines, shown in Figure 1, work by directing speculative execution of indirect branches into an infinite loop. This is achieved by redirecting indirect branches to a thunk which captures the branch target on the return stack buffer (RSB) and uses a `RET` instruction instead of a `CALL`. A `RET` uses a different speculative prediction mechanism which leverages the RSB rather than the vulnerable BTB, so it does not suffer from the same vulnerability as a `CALL` instruction. Speculative execution of the return path will execute lines 4-6, labeled `capture`, which exhausts speculative execution by looping forever. The `PAUSE` instruction on line 4 is used to release processor resources to a hardware simultaneous multithread (SMT) or to save power if no execution is needed, and the `LFENCE` instruction acts as a speculation barrier [15, 36].

Retpoline safety is based on the behavior of the predictor for the `RET` instruction. On Intel architectures before Skylake, speculation for `RET` instructions were always computed from the RSB [14]. On Skylake and after however, the contents of the BTB may be used if the RSB is empty, making these architectures vulnerable even if retpolines are used [31]. To prevent an attack which causes the underflow of the RSB, a technique known as RSB stuffing can be used on these architectures on events which could cause the RSB to be emptied [51]. However, a deep call stack may still result in unpredictable overflows, leaving these architectures vulnerable even with both RSB stuffing and retpolines enabled.

Measuring precise performance overheads of retpolines is difficult because they affect deep microarchitectural state such as the RSB, and may cause future mispredictions. Empirically, retpolines have been observed to result in as much as a 20% slowdown on some workloads, increasing the cost of an indirect branch from a worst-case cost of around 10 cycles [13] (indirect branch misprediction) to almost 70 cycles in the common case. The performance penalty Spectre imposes on indirect branches makes optimizing them much more important.

Hardware microcode mitigations. As a result of Spectre, microcode updates have been introduced which add functionality to control indirect branch predictions. These mechanisms include Indirect Branch Restricted Speculation (IBRS), Single Thread Indirect Branch Predictors (STIBP) and Indirect Branch Predictor Barrier (IBPB) [14]. IBRS works by defining four privilege levels: host and guest modes with corresponding user and supervisor modes. The processor guarantees that lower privilege modes and other logical processors cannot control the predictors of more privileged modes. STIBP and IBPB are used to protect VMs and processes across context switches by preventing predictions in one thread or context from affecting another thread or context.

These microcode mechanisms can replace retpolines. Unfortunately, the performance penalty of IBRS is quite high: Since it is implemented in microcode, it likely flushes the BTB on each privilege transition and requires a model-specific register (MSR) write whenever a privilege transition occurs. IBRS has an extremely high overhead compared to retpolines (observed to be as high as 25–53%+) [17, 47, 50].

IBRS is the only mechanism which completely protects Skylake and future processors, because the BTB may be used if the RSB underflows. However, due to concerns with complexity and performance, the Linux kernel has adopted the use of retpolines on all vulnerable architectures instead, using RSB stuffing to protect the RSB [51, 53].

Hardware mitigations. Finally, Intel has proposed mitigations for Spectre in new microarchitectures which require new microprocessors to be deployed. These mitigations cannot address Spectre in the billions of already deployed processors [37]. This includes technologies such as Enhanced Indirect Branch Restricted Speculation (Enhanced IBRS) [16] and control flow enforcement technology (CET) [14]. Enhanced IBRS eliminates the overhead of IBRS by removing the requirement to write to an MSR. CET restricts the speculations which the processor can make by requiring that indirect branches target only special `ENDBR` (end-branch) instructions.

Hardware mitigations will be much more performant than software or hardware microcode mitigation today, since future processors will be able to make deep changes to the microarchitecture to account for Spectre. Practically implementing these technologies on code which must run on both secure and vulnerable hardware remains an open question, especially in the cloud where live migration between processors remains a real possibility [36, 48].

2.2 Indirect Call Promotion

The cost of indirect calls have been historically observed to be high, even before the discovery of Spectre. The most common technique used to reduce the cost of indirect calls is known as *indirect call promotion*, where likely call targets are promoted to conditional direct calls, reducing the chance that the processor will mis-speculate an indirect call using a code fragment similar to that shown in Figure 2 [6, 7].

Promoting indirect calls comes at a cost, however. Each promoted call increases code size, which significantly decreases performance if infrequently used targets are promoted. Code size also limits the number of targets that can be promoted. Indirect branches which call a large number of targets with equal likelihood are poor candidates for promotion.

Compilers can determine and promote likely targets at compile-time [9, 23], through the use of profile-guided optimization or feedback-directed optimization (PGO/FDO) [10, 11, 26, 32, 41, 42]. Collection of targets in the Linux kernel, which is the focus of this paper, is complicated by several factors. First, branches collected through the use of profiling may not resemble the actual execution at runtime and incorrect learned targets may even result in regressions [40], particularly if the code is executed under diverse conditions. Previous work has shown that simply executing the Linux kernel under different workloads results in different profiles [54, 55]. Second, mechanisms such as KASLR [45], JITed code (e.g., eBPF programs) and dynamically loaded modules, may make it impossible to learn targets until runtime. Finally, incorporating learned branch targets requires recompilation, which may mean having different binaries for each workload or a deployment infrastructure which adapts binaries over time [10, 42]. This is incompatible with how kernels are provided in most OS distributions: as binary packages.

While binary translators [18, 27, 28] and JIT compilers [25, 33, 44] are capable of collecting branch targets at runtime, binary translators result in a significant overhead as they are designed to translate and instrument the entire binary, and may need to restrict the kernel's functionality for best performance [27]. JIT compilers cannot directly run the Linux kernel, although some progress has been made to leverage LLVM to run applications as complex as Linux [3].

2.3 Alternative Solutions

After we released parts of our code [4], several solutions that employ indirect call promotion have been proposed by Linux developers. Hellwig added a fast-path to code that invokes DMA operation based on the I/O memory management unit (IOMMU) that the system uses. This solution effectively performs indirect branch promotion of each call to a single static

```
1  call %rax  |        cmp ${likely},%eax
2             |        jnz miss
3             |        call {likely}
4             |        jmp done
5             |  miss: call %eax
6             |  done:
```

Figure 2: Indirect call promotion. Indirect branches on the left are replaced with a direct branch (line 3) which is called if the prediction is correct at runtime (lines 1, 2). Otherwise, a normal indirect call is made (line 5).

predetermined target: the functions that are used when no IOMMU is used. As a result, the solution does not provide any benefit when an IOMMU is used [21].

A similar solution was introduced by Abeni to reduce the overhead of indirect branches in the network stack [2]. This solution is also static, requiring the developer to determine at development time what the likely targets are of each call. In addition, this solution is not suitable for calls to functions in loadable modules, whose address is unknown at compilation time. The implementation, which requires changing function calls into C macros reduces code readability. As a result this solution is limited to certain use-cases.

Finally, Poimboeuf [43] proposed a mechanism that addresses indirect calls whose target rarely changes. The proposed solution uses direct calls, and when the target changes, performs binary rewriting to change the direct call target. As this mechanism does not have a fallback path, it is only useful in certain calls. In addition, it requires the programmer to explicitly invoke binary rewriting and the use of C macros negatively affects code readability.

2.4 JumpSwitches

JumpSwitches are designed to mitigate Spectre by taking advantage of indirect call promotion and leveraging semantic information not available by compile-time. JumpSwitches are motivated by the following observations:

- Retpolines induce $\approx$70 cycles of overhead which can be leveraged by software to resolve indirect branches.
- Promoting indirect branches can be combined with ensuring the safety of speculative branch execution.
- Committing to specific optimizations at compile-time can limit performance and potentially risk safety (such as assuming Spectre-safe hardware).

Threat Model. JumpSwitches assume that only indirect branches and returns are vulnerable to Spectre variant 2, as stated by Intel [14] and the original Spectre disclosure [30, 38]. It is also to important to note that while retpolines can serve as a defense against Spectre variant 1 ("Bounds Check Bypass") [30, 39, 49], JumpSwitches do not defend against Spec-

Mechanism	Overhead	Spectre-Safe	Learning
retpolines [49]	medium	yes	none
EIBRS [48]	low	yes	none
PGO/FDO [54]	low	w/retpoline	static
DBT/JIT [27]	high	w/retpoline	dynamic
JumpSwitches	low	dynamic	dynamic+semantic

Table 1: Comparison of various indirect call mechanisms, their relative overheads and if they are Spectre-safe.

tre variant 1. We assert that replacing retpolines with JumpSwitches does not make the Linux kernel vulnerable to Spectre variant 1, as retpolines are used solely as a Spectre variant 2 defense. In Linux, static analysis and selective masking/serialization is used to defend against Spectre v1 [12], since Spectre variant 1 defenses are still necessary even in "Spectre safe hardware", where retpolines are turned off for performance. Spectre variant 1 is not expected to be fixed in future hardware [46]. We assert that JumpSwitches are as safe as future "Spectre safe" processors currently on vendor roadmaps.

Compared to indirect call promotion implemented with a compiler, JumpSwitches are able to dynamically adapt to changing workloads and take advantage of semantic information only available at runtime, much as a JIT compiler may employ polymorphic inline caching [22]. Unlike binary translation, JumpSwitches are integrated in the kernel and designed for minimal overhead to only instrument indirect calls rather than the entire binary. Furthermore, JumpSwitches are able to take advantage of the rich semantic information available in the kernel source and lost in the binary. A summary of indirect call mechanisms can be found in Table 1.

Our work is focused on the Linux kernel since it fully supports retpolines and runs privileged code which must be protected against Spectre. We believe, however, that JumpSwitches can be applied as a general mechanism and be widely deployed as retpolines are today. The goals of JumpSwitches are to:

- Predict indirect branch targets while preserving safety.
- Require minimal programmer effort: leveraging JumpSwitches should not require source code changes.
- Be flexible: allow the programmer to provide additional semantic information when available.

3 JumpSwitch Architecture

JumpSwitches are code fragments which, like retpolines, serve as trampolines for indirect calls. Their purpose is to leverage indirect call promotion and use direct calls which have a much lower cost than indirect calls, especially in the era of Spectre. JumpSwitches are Spectre aware: if a JumpSwitch cannot promote an indirect call, a Spectre mitigated indirect call is made. In Spectre-vulnerable hardware, JumpSwitches fall back to retpolines, but future hardware may easily take advantage of technologies such as Enhanced IBRS.

We have developed five different types of JumpSwitches, each optimized for a different purpose. The simplest and default type of JumpSwitch is known as an *Inline JumpSwitch*, which is optimized for code size and covers the majority of use cases. The *Outline JumpSwitch* is used when we learn that an indirect branch has multiple targets. Inline and outline JumpSwitches are known as *generic JumpSwitches* and can be used on all indirect branches. We also provide three *semantic JumpSwitches* which support commonly encountered indirect branches where deeper semantic information is available from the programmer. *Search JumpSwitches* support a large number (hundreds) of targets. The *Registration JumpSwitch* covers the commonly used registration pattern used in callback lists. Finally, the *Instance JumpSwitch* covers the case where call targets are strongly correlated with an instance, such as an object or a process. Within the Linux kernel, the *JumpSwitch worker* learns about new branch targets and makes decisions on whether a JumpSwitch should be changed to a different type, outside the path of execution. To change the active JumpSwitch, the worker makes use of a multi-stage patching mechanism which atomically updates a JumpSwitch without risking safety.

JumpSwitches are integrated into the Linux build infrastructure through the use of a compiler plugin. Thanks to our integration with the Linux build system, taking advantage of inline JumpSwitches and outline JumpSwitches requires no source code changes. We also supply additional JumpSwitches which leverage semantic information available in the kernel provided by developers, enabling the search, registration and instance JumpSwitches.

In the next sections, we first describe each type of JumpSwitch. Then, we show how the JumpSwitch worker learns and adapts JumpSwitches during runtime. Finally, we show how we patch Linux with JumpSwitches as well as optimizations we made during integration.

3.1 JumpSwitch Types

Generic JumpSwitches can be used on all indirect calls, while semantic JumpSwitches cover common use cases. We enumerate the types of JumpSwitches below:

Inline JumpSwitch. These JumpSwitches serve as generic trampolines which replace an indirect call. They act as a basic building block for other JumpSwitches and enable dynamic learning of branch targets. Because they replace code, they must be short, otherwise they risk bloating code and increasing pressure on the instruction cache. At the same time, they must be upgradable by the JumpSwitch worker at runtime and support learning. In order to fulfill these three requirements, inline JumpSwitches are designed to be safe by default and easily patched. An example of an inline JumpSwitch and the indirect call it replaces is given in Figure 3.

```
1  call %eax  |          cmp ${likely},%eax
2             |          jnz miss
3             |          call {likely rel}
4             |          jmp done
5             |     miss: call {slow rel}
6             |     done:
```

Figure 3: Inline JumpSwitch. Indirect branches on the left are replaced with a sequence that may promote a *likely* call to a direct call (lines 1–4), falling back to a *slow* path if the likely target was not in **%eax**.

```
cmp     ${ entry0 }, %eax
jz      { entry0 rel }
cmp     ${ entry1 }, %eax
jz      { entry1 rel }
        ...
jmp     learning rel
```

Figure 4: Outline JumpSwitch. Multiple targets are supported, using `JZ` instructions to avoid the need for a function epilogue and falling back to learning if the target is not found.

The inline JumpSwitch may point to one of two targets: *likely* or *slow*. *Likely* represents a branch target that the JumpSwitch has learned to be likely and is promoted to avoid an indirect jump. *Slow* can represent one of three targets, depending on which mode the inline JumpSwitch is in:

- Learning mode, where *slow* points to learning code which updates a table of learned targets.
- Outline mode, where *slow* points to an outline JumpSwitch leading to more targets.
- Fallback mode, where *slow* points to either a retpoline or is a normal indirect call, depending on if the system is Spectre vulnerable.

When compiled, the inline JumpSwitch is set to fallback mode and both *likely* and *slow* point to a retpoline. At runtime, the JumpSwitch worker may patch *likely* and *slow* depending on which mode is required and what targets have been learned. When an inline JumpSwitch is executed, if *likely* matches the contents of the register holding the branch target (in this case, *eax*), the call on line 3 is executed. Otherwise, the call to *slow* is executed. It is important to note that while the value of *likely* is used by the `CMP` instruction represents a direct address, the target of the `CALL`s are relative, since there are no direct absolute jumps in x86-64.

Outline JumpSwitch. To support multiple targets dynamically without increasing code size in the common case where there is only a single learned target, outline JumpSwitches may be called by inline JumpSwitches. Unlike inline JumpSwitches, outline JumpSwitches are dynamically allocated and generated by the JumpSwitch worker as targets are learned. As an optimization, since outline JumpSwitches are called by inline JumpSwitches, we avoid the normal work of setting the frame pointer and returning by using jump instructions, as shown in Figure 4. As a result, each target uses two instructions: a `CMP` and a `JZ`. If the target is not in the outline JumpSwitch, we fall back to learning code, as in the inline JumpSwitch in learning mode.

While outline JumpSwitches support multiple targets, each target adds an additional conditional branch, which induces overhead. To avoid reducing performance due to excessive or unpredictable branch targets, we limit targets in an outline JumpSwitch to 6, for a total of 7 (1 in the inline JumpSwitch and 6 in the outline). To support more targets, a search JumpSwitch can be used.

In addition to the generic indirect branches targeted by inline and outline JumpSwitches, we also target common indirect branches when semantic information is available.

Registration JumpSwitch. The registration pattern occurs when a list of callbacks is registered for later use. Registration is used in the kernel for structures such as callback lists (e.g., notifiers such as `user_return_notifier` and `mmu_notifier`), or filter lists (e.g. `seccomp`).

In such cases, since the callbacks are called from a single call-site in a loop, learning targets would fail if the callback list length is greater than the maximum number of learning JumpSwitch targets. Instead, we use a registration JumpSwitch, which unrolls callback list invocation code, sets multiple call instructions to callbacks. When a function is added or removed callbacks from a callback list the registration JumpSwitch is explicitly invoked and patches the callback addresses into the call instructions.

Instance JumpSwitch. Another common pattern that JumpSwitches target are cases where the likely branch targets are strongly correlated with a process. For example, the running process may dictate which `seccomp` filters are running, or per-process `preemption_notifier` used. Instance JumpSwitches take advantage of semantic knowledge about the running workload, and contain one of the previous three JumpSwitch types, but on a per-instance basis.

To support instance JumpSwitches, a separate executable memory area is allocated for each process. This memory area contains the instance JumpSwitches. While each area is located in a different physical memory address, the instance JumpSwitch is always mapped in a fixed virtual address. This allows us to invoke process specific JumpSwitches by direct calls, as context switches between different processes also switch instance JumpSwitches to the process-specific ones. Learning and code modifications are then done on a per-process basis.

```
 1  int syscall(int nr, regs_t *regs)
 2  {
 3    int direct_nr = private_nr[nr];
 4
 5    if (direct_nr == INVALID)
 6          return call_table[nr](regs);
 7
 8    /* Hit; 4 entries per-process */
 9    if (direct_nr < 2) {
10      if (direct_nr < 1)
11        return private_call_0(regs);
12      return private_call_1(regs);
13    }
14    if (direct_nr < 3)
15      return private_call_2(regs);
16    return private_call_3(regs);
17  }
```

Figure 5: Search JumpSwitch pseudo-code, similar to the one that is used to invoke system-calls. In this example there are 4 slots for the most common targets. If there is a miss, an indirect call is initiated (line 6). Otherwise, a direct branch is performed (lines 9-16). System-call indirection table and functions (prefixed with "private") are set per instance (i.e., process). They are located at a fixed virtual address, by mapped to different physical memory on each process.

Search JumpSwitch. Some indirect branches may have potentially hundreds of targets, such as in call tables (e.g., syscall, hypercalls) and event handlers (e.g., virtualization traps) and other jump tables commonly used to execute selection control, as commonly done by C "switch" statements. These constructs typically translate a key such a handler number to a function for that key, and can be compiled to machine code that use binary decision tree or to code that use jump-tables. In practice, compilers prefer jump-tables for densely packed case items, as they usually require fewer instructions and branches [8]. However, this behavior is based on the assumption that an indirect branches are inexpensive, an assumption which has changed and should be reevaluated in the era of Spectre, particularly when retpolines are required.

In these cases, the JumpSwitch may benefit from a search tree which may reduce the number of branches needed for an indirection lookup and support many more targets. However, in the Linux kernel, we experimented with the most commonly used table, the system call table, which is used to dispatch the function that handles system calls based on a well known ("magic") number. Linux implements this table manually and does not use a switch statement. In x86 Linux, there are over 300 system calls. When retpolines are enabled, we found both static jump tables and binary decision trees to be inefficient. Our experiments with static binary decision tree showed they easily degrade performance when more than very few system calls are used. Unfortunately, outline JumpSwitches do not perform well as all the slots quickly fill up and the learning mechanism disables the outline block to prevent excessive overhead and performance degradation.

To address this situation, search JumpSwitches use an adaptive binary decision tree that caches the most frequent call translations (in the case of a system call, from system call number to handler), and the jump-table is used as a fallback. We dynamically construct a binary decision tree and keep a bitmap, where each bit corresponds to the respective translation in the jump-table. When the translation is called, the search JumpSwitch checks and updates the bitmap if learning is enabled, then dispatches the request (Figure 5). The JumpSwitch worker periodically updates the decision tree based on the learned frequency data.

3.2 Learning and the JumpSwitch Worker

To learn new targets, JumpSwitches use a learning routine, which buffers learned targets in an optimized hash table, and the worker periodically updates JumpSwitches using multi-stage patching. Different routines are used for learning on generic (inline and outline JumpSwitches) and search JumpSwitches. Registration and instance JumpSwitches do not learn asynchronously and do not require the worker.

Generic Learning Routine. To learn new branch targets, we could have used hardware mechanisms such as Precise Event-Based Sampling (PEBS) [24], which is used by many PGO and FDO frameworks. However, PEBS has several limitations, such as not being able to selectively profile branches and cannot be run in most virtual environments. To ensure that JumpSwitches are hardware agnostic, we developed a lightweight software mechanism for learning branch targets.

Our software learning routine records branch targets in a small 256-entry per-core hash table. The branch source and destination instruction pointers are XOR'd and the low 8 bits are used as a key for the table. Each entry of the table saves the instruction pointer of the source and the destination (only the low 32-bits, as the top 32-bits are fixed), as well as a counter of number of invocations. We ignore hash collisions, which can potentially cause destinations that do not match the source to be recorded and wrong invocation count, as they only lead to suboptimal decisions in the worst case and do not affect correctness. This allows us to keep the learning routine simple and short (14 assembly instructions), which is important for keeping the overhead of learning low.

After the learning routine is done, fallback code is called, which may either be a retpoline if Spectre-vulnerable hardware is present or a normal indirect call.

Search Learning Routine. For search JumpSwitches, an alternate method is used. Search JumpSwitches are tracked per process. Each thread holds a flag that indicates whether learning needs to be done. When learning is on, each thread records which calls have been performed in a per-core frequency table. Upon context-switch the frequencies are added to a per-process frequency table that sums them up, and the per-core table is cleared. When training is over, an inter-processor interrupt (IPI) is sent to cores that still run threads of the process to sum them up into the per-process table.

JumpSwitch Worker. The worker runs once every epoch (1 second default, configurable), or when a relearning event is triggered. The worker performs learning by reading the targets that were discovered by the learning routines and updating the JumpSwitch. The worker processes generic JumpSwitches and search JumpSwitches differently:

Generic JumpSwitch Updates. During each epoch, the JumpSwitch worker checks if new call targets were encountered by reading the hash tables on each core and summing the total calls for each source-destination pair on all cores. For each source, the worker sorts each destination by the number of hits and promotes those destinations. If the destinations have already been promoted, they are ignored, and if the maximum capacity of an outline JumpSwitch is reached, the inline JumpSwitch is put into fallback mode, which disables learning for the target. As a result, a worker run can result in the following changes to a JumpSwitch:

- Update an inline JumpSwitch's *likely* target.
- Switch an inline JumpSwitch from learning to outline.
- Create or add targets to an outline JumpSwitch.
- Switch an inline JumpSwitch to fallback mode.

Once the worker is done processing data, it clears all hash tables allowing calls with hash conflicts to save their data.

Generic Learning Policy. Learning imposes some overhead and should only be performed when a performance gain will likely result. To mitigate learning overheads, the worker holds generic JumpSwitches in three lists:

- Learning JumpSwitches, which are in learning mode. They do not improve performance, but track targets.
- Stable JumpSwitches, which have a single target. These JumpSwitches do not need to be disabled for relearning, as their fallback path jumps to the learning routine.
- Unstable JumpSwitches, which have more than a single target. These include JumpSwitches with an outlined block, and those that have too many targets, and were therefore set not to have an outlined block.

During each epoch, if no JumpSwitches were updated, the JumpSwitch worker picks a number of JumpSwitches (configurable, 16 default) from the unstable list and converts them into learning JumpSwitches, disabling them and setting the fallback to jump to the learning routine. To avoid being too aggressive, the worker does not switch a JumpSwitch into learning mode more than once every 60 seconds.

Search Learning Updates. For search JumpSwitches, the worker sums up per-cpu frequency tables, but instead of updating an inline JumpSwitch, a binary-search tree is updated, promoting frequent values and clearing the bitmap.

Search Learning Policy. Learning is turned on periodically per-process. For each process, we save whether learning is on and the last time it was performed. When a thread is scheduled to run, it checks if learning is on. If so, it caches the status in thread-local memory. If it is off, a check is performed to see whether a time interval passed since learning was last on (20 seconds default). If that time has elapsed, learning is turned back on, and the process is added into a list of learning processes. Learning is stopped on the next update.

3.3 Patching and Updating

To minimize the performance impact of patching inline JumpSwitches, the worker employs a multi-phase mechanism to ensure that JumpSwitches are safely updated as multiple instructions are patched live without locks or halting execution. We leverage the Linux `text_poke` infrastructure [29], designed for live patching of code.[2] Patching is a three phase process, outlined below, and line numbers refer to inline JumpSwitch shown on the right of figure 3.

1. A breakpoint is installed on line 1 by writing the single-byte breakpoint opcode onto the first byte of the instruction. If the breakpoint is hit, the handler emulates a call to the retpoline, as if it was executed on line 5. To simplify implementation, the handler does not perform the emulation directly, but instead moves the instruction pointer to a newly created chunk that pushes onto the stack the return address (line 6) and executes `JMP` branch to the retpoline chunk.
2. The patching mechanism waits for a quiescence period, to ensure no thread runs the instructions in lines 2–5. In the Linux kernel, this is performed by calling the `synchronize_sched` function. Afterwards, the instructions on lines 2, 3 and 5 are patched. The instruction on line 4 is not changed, to allow functions that return from the `CALL` in line 3 to succeed in completing the JumpSwitch.
3. The same breakpoint mechanism in phase 1 is used, this time restoring the `CMP` on line 1 with the new promoted target, re-enabling the JumpSwitch.

When a fully-preemptable kernel is used, we also check whether the JumpSwitch code was interrupted before the target function was called and rewind the saved instruction

[2] We have submitted upstream patches to Linux to further harden the security of this mechanism [5].

pointer to line 1. This ensures the code will be executed again when the thread is re-scheduled. To efficiently determine if JumpSwitch code was interrupted, a check is only performed only if instructions we use (cmp, jnz, jz, call and jmp) are interrupted.

3.4 Linux Integration

We implement and integrate JumpSwitches on Linux v4.18 through a gcc-plugin integrated into the Linux build system. The gcc-plugin is built during the kernel build and replaces call sites with our JumpSwitch code, allowing generic JumpSwitches to be seamlessly integrated into the Linux kernel without source code changes. We write the instruction pointer and the register used for the indirect call into a new ELF section. This information is read during boot to compose a list of calls, allowing the worker to easily realize which register is used in each JumpSwitch. It also serves as a security precaution to prevent intentional or malicious memory corruption of the JumpSwitch sampling data from causing the JumpSwitch worker from patching the wrong code. The use of JumpSwitches is configurable via a Kconfig `CONFIG` option.

Semantic JumpSwitches require slight changes to kernel source. To support instance JumpSwitches for processes, we mapped a per-process page in kernel space. We implemented both search and registration JumpSwitches to be placed in per-process instance JumpSwitches. We modified `seccomp` to use a registration JumpSwitch, which accounts for the fact that filters are per process by being placed in an instance JumpSwitch. We also patch system call dispatching to use a search JumpSwitch, to account for the large system call table which is used, and place it in an instance JumpSwitch. Overall, implementing our semantic JumpSwitches in the kernel required changing about 30 SLOC.

The JumpSwitch worker is integrated into the kernel in a similar manner to other periodic tasks in Linux which patch code such as the `static-keys`, `jump-label` and `alternatives` infrastructure in Linux.

3.5 Direct Kernel Entry

During our Linux integration, we observed that JumpSwitches did not provide the full speedup we expected. Further analysis revealed that this was due to the overhead of page-table isolation (PTI), which was introduced to mitigate against Meltdown, a different speculative execution vulnerability [1, 34]. PTI introduced a new trampoline used during system calls to switch between the user and kernel page tables. This trampoline is mapped to a different virtual address on each core so the trampoline can determine the correct per-core transitional kernel stack. The trampoline is part of a per-core data structure, and since this data-structure is large, it is located "far" (more than 2GB away from the kernel [19]), preventing a direct jump. Unfortunately, the resulting indirect jump must use a retpoline [52] on Spectre vulnerable hardware, resulting in lower than expected gains when we used JumpSwitches.

To eliminate the need for this retpoline, we split the per-core data structure into two data structures: a small one which includes the transitional kernel stack, TSS and trampoline code, which are all needed to transition into the kernel during a system call; and a second larger one that includes the other fields. This allowed us to move the small part of the trampoline back into the same 2GB in which the kernel code is mapped, and use a relative jump instead of a retpoline, resulting in a significant performance gain. Our solution requires replicating the trampoline page, which consumes a minimal amount of extra physical memory ($<$ 32MB for 8192 cores).

4 Evaluation

Our evaluation is guided by the following questions:

- How does the specialization of each JumpSwitch impact the performance of the kernel in isolation? (§4.1)
- How do JumpSwitches perform with real-world applications and benchmarks? (§4.2)
- How does learning impact JumpSwitches? (§4.3)
- How many targets are needed per indirect branch? (§4.4)
- Is JumpSwitch is useful after the recent security vulnerabilities are resolved in hardware? (§4.5)

Testbed. Our testbed consists of a Dell PowerEdge R630 server with Intel E5-2670 CPUs, a Seagate ST1200 disk, which runs Ubuntu 18.04. The benchmarks are run on guests with a 2-VCPUs and 16GB of RAM. Each measurement was performed at least 5 times and the average result and standard deviation are reported. All workloads were executed with a warm-up run prior to measurement.

Configurations. We run and report the speedup relative to the baseline system which uses retpolines as a mitigation against Spectre. We report the results of:

- **base**: The baseline system with retpolines enabled.
- **direct-entry**: direct jump kernel entry trampoline.
- **+inline**: *direct-entry* with inline JumpSwitches.
- **+outline**: *+inline* with outline JumpSwitches when there are multiple targets.
- **+registration**: *+outline* with per-process (instance) registration JumpSwitches for `seccomp`
- **+search**: *+registration* with per-process (instance) search JumpSwitches for system calls.
- **unsafe**: the baseline system with retpolines disabled.

4.1 Microbenchmarks

Given the diversity of JumpSwitches we implemented, we first wanted to evaluate how each type of JumpSwitch would

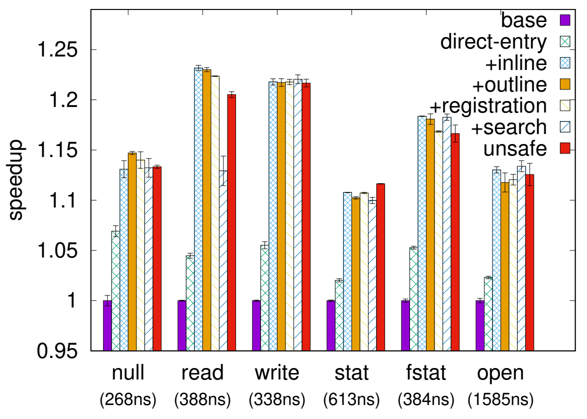

Figure 6: System call speedup relative to the baseline setup that uses retpolines. The runtime of the baseline system is reported in parentheses.

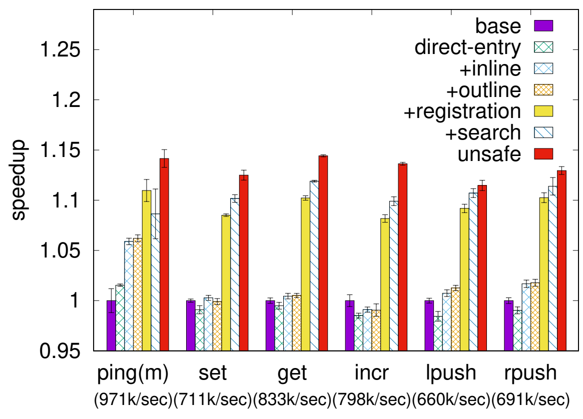

Figure 7: Relative speedup of Redis using JumpSwitch. The operations are are invoked and measured by redis-benchmark. The runtime of the baseline system is reported in parentheses.

improve the performance of the kernel in isolation. We evaluate system calls in a microbenchmark as they have shown to a particular source of slowdown as a result of both the Meltdown [1] and Spectre hardware vulnerabilities, as system calls stress user-kernel transitions, which must now be protected.

We measure the impact of JumpSwitch on the time it takes to run common system-calls using the `lmbench` tools for performance analysis [35]. Figure 6 shows the speedup relative to the baseline protected system. As shown, eliminating the indirect call in the entry trampoline provides a benefit of roughly 15ns which is more pronounced in short system calls. The inline JumpSwitch improves performance by up to 15%, making the system calls run as fast as they would without retpolines. It is noteworthy that this is due to the fact that the workload is very simple, which allows the training mechanism to inline call targets with very high precision.

This high precision is the reason that Outline JumpSwitches do not provide any benefit in this benchmark. Semantic JumpSwitches also offer little benefit here: `seccomp` filters are not installed and the same system call is called repeatedly.

To further understand the performance benefits of JumpSwitches, we some simple operations: Redis key-value store commands using redis-benchmark. Snapshotting is disabled to reduce variance. Each test runs the same command repeatedly, and the results are depicted in Figure 7. As seen, using registration JumpSwitches to avoid indirect calls when `seccomp` filters are invoked, provide up to 9%, performance improvement. This is due to the fact that `systemd`, software which acts as a system and service manager in Ubuntu, attaches 17 `seccomp` filters, which are executed upon each system call. Search JumpSwitches, whose benefit was not shown when the same system call was repeatedly executed, improve performance by up roughly 2%.

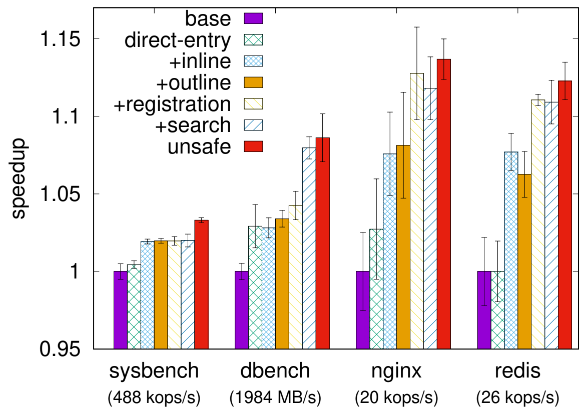

Figure 8: Relative speedup of macro-benchmarks using JumpSwitch. The baseline system is reported in parentheses.

4.2 Macrobenchmarks

Next, we want to see how JumpSwitches perform with real-world workloads which do not necessarily stress the userspace-kernel transition. We run the following benchmarks: *sysbench*, which runs a mixture of file read, write and flush operations, running on a temporary file-system (tmpfs); *dbench*, a disk benchmark that simulates file server workload, running on tmpfs; Nginx web-server, using ApacheBench workload generator to send 500k requests for a static webpage using with 10 concurrent requests at a time; and *Redis*, using Yahoo Cloud System Benchmark (YCSB) as a workload generator (running `workloadA`). In all cases we ensure the workload generator is not the bottleneck.

Figure 8 depicts the results. Eliminating the indirect branch from the entry trampoline provides a modest performance

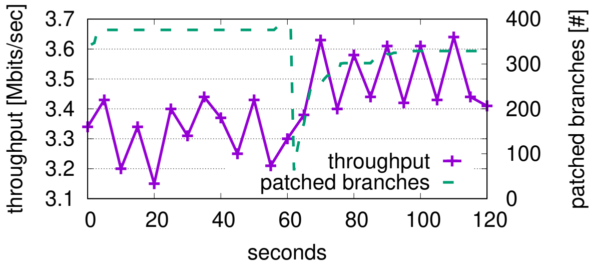

Figure 9: UDP throughput and the number of patched branches when the workload changes and learning is initiated after 60 seconds. A single inlined JumpSwitch is used.

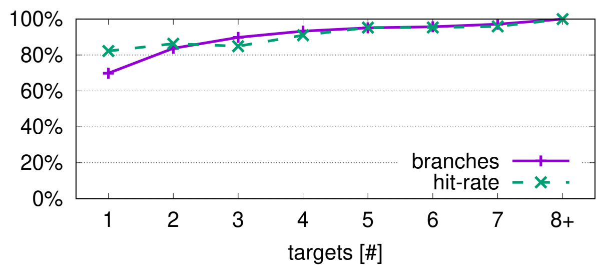

Figure 10: Hit-rate of the inlined and outlined JumpSwitch compared against a CDF of the number of targets that each indirect call-site holds while running Nginx.

improvement, which is most pronounced in dbench as it invokes many system calls. The inline JumpSwitch provides the major part of the performance gains, up to 8% improvement over the baseline system for the Redis benchmark. Like the Redis micro-benchmarks, the macro-benchmark shows a considerable gain (3%) from a registration JumpSwitch, due to the multiple (17) `seccomp` filters which are attached to it. Dbench shows a performance gain of up to 4% from the search JumpSwitch, as it repeatedly runs a small subset of system calls, which are quickly learned by the adaptive search tree. It appears that this mechanism also improves Redis performance, but due to the high standard deviation, it is hard to say so definitively. For the same reason it is hard to conclude whether the experienced performance degradation in nginx with some mechanisms is meaningful, especially since registration JumpSwitches have almost no effect on nginx, whose system calls do not go through `seccomp` filters.

Overall, the macro-benchmarks evaluation show that JumpSwitches can restore most of the performance loss due to retpolines, narrowing the difference between protected and non-protected systems to less than 3%.

4.3 Dynamic Learning

One of the main benefits of the runtime instrumentation of JumpSwitches over compile-time decisions is the ability to dynamically learn branch targets. To evaluate the value, effectiveness and performance of dynamic learning we create a scenario where the workload behavior changes. In this experiment we only enable inline JumpSwitches, emulating how compilers perform indirect branch prediction. To control learning, we disable the automatic learning mechanism and use a user-visible knob that initiates the relearning.

First, we run `iperf`—a network bandwidth measurement tool—to send and receive UDP packets using IPv6, and we set the kernel to learn and adapt the branches accordingly. Then we use iperf to measure the throughput of IPv4 UDP performance, by sending messages of a single byte. After 60 seconds, we restart the learning process.

Figure 9 shows the throughput (sampled every 5 seconds), and the number of adapted branches (sampled every second). As shown, restarting the learning process resets all the branches, dropping the number of patched branches to zero momentarily. Yet, within 5 seconds, over half of the branches that were patched prior to the benchmark execution are patched again with updated targets. The adaptation of the branches improves the overall throughput by ≈5%.

This demonstrates the value of dynamic learning over profile-based techniques, which “bake” in indirect branch promotions at compile time. Since learning is fast, it can be done periodically without degrading the overall performance of the system. Future work may apply a more advanced algorithm to do relearning, such as on a system event.

4.4 Branch Targets

The usefulness of inline and outline JumpSwitches depends on the number of branch targets and the distribution of the frequency in which they are used. To study this distribution, we used our system by modifying the number of branch target slots in each JumpSwitch and measuring the hit rate. In addition, we measured how many targets each branch has and created a cumulative distribution function to compare with the hit-rate. As shown in Figure 10, 71% of the function calls had a single target, and the inlined JumpSwitch by itself (only 1 allowed destination) achieved a hit-rate of 82%. As we increase the number of allowed destinations, the outline JumpSwitch further improves the hit-rate up to 96% when both outlined and inlined JumpSwitch are used.

This shows that inline JumpSwitches alone are able to handle a majority of cases, while outline JumpSwitches provide more complete coverage for branches with many targets. Only few branches require beyond 7 targets. In the small fraction of branches with 8 or more targets, outline JumpSwitches are disabled because the cost of the additional conditional branches outweigh the benefits of call promotion.

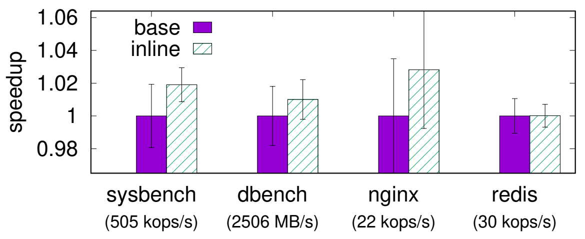

Figure 11: Benchmark speedup relative to the baseline when Spectre and Meltdown protections are disabled. The runtime of the baseline system is reported in parentheses.

4.5 Post-Spectre Benefits

Finally, we examine whether JumpSwitches are relevant when hardware solutions for Spectre and Meltdown mitigations are present, and retpolines are not longer needed. The benefit of JumpSwitches would be smaller, since the cost of an indirect branch becomes considerably lower. Some of our proposed mechanisms will become irrelevant. If Meltdown is resolved, the indirect branch in the trampoline page is not used. If Spectre is resolved, search and outline JumpSwitches become too costly relatively to the cost of an indirect branches. Registration JumpSwitches may improve performance, but our experiments indicate they require further micro-optimizations.

In contrast, inline JumpSwitches are potentially valuable even after the recent CPU bugs are fixed. Their performance benefit might be lower, making aggressive retraining of inline JumpSwitches inappropriate, yet retraining can still be done infrequently, based on user requests or in response to system events (e.g., CPU saturation). To evaluate the impact of JumpSwitches in such setups, we run the same macro-benchmarks while directing Linux to drop the protections against Meltdown (no page table isolation) and Spectre (no retpoline). We disable the automatic relearning mechanism and relearn manually when the workload is first invoked.

The results are shown in Figure 11. JumpSwitch can provide up to 2% performance improvement on systems which that are not vulnerable to Spectre and Meltdown. As shown on Redis, however, this benefit can be nullified in some cases.

5 Conclusion

The recent CPU vulnerabilities due to speculative execution revealed that the CPU cannot be regarded as a black-box that can be blindly relied on. Hardware bugs are hard to fix in existing systems, which necessitates the mitigation against the vulnerabilities using software techniques. JumpSwitch performs this task by extending compiler-optimization to make runtime decisions, while reducing the overhead of the current mitigation techniques. We have shown that JumpSwitches achieve our goal of leveraging speculative execution cycles to predict indirect branch targets while preserving safety, requiring minimal programmer effort while providing the flexibility for the programmer to add rich semantic information.

JumpSwitches show that software can efficiently perform hardware tasks such as branch prediction. Since proposed hardware mitigations against speculation will come with a cost in performance, a hardware-software solution would allow software to define in fine granularity which speculation is permitted and which needs to be blocked. JumpSwitches limit the allowed speculation using direct branches. Hardware mechanisms provide tools for software to perform this task more efficiently, for example, by providing raw interfaces to content addressable-memory. The benefit of combining both solutions is both in performance, by leveraging software knowledge, and in security, by allowing easier mitigation of potential hardware vulnerabilities.

6 Acknowledgment

We would like to thank the paper shepherd, Yuval Yarom and the anonymous reviewers for the insightful comments. We would also like to thank Andy Lutomirski, Josh Poimboeuf and Peter Zijlstra for their imperative technical feedback. This research was supported by the Blavatnik Interdisciplinary Cyber Research Center, Tel Aviv University, grant 0605714731.

References

[1] CVE-2017-5754. Available from NVD, CVE-ID CVE-2017-5754, https://nvd.nist.gov/vuln/detail/CVE-2017-5754, January 1 2018. [Online; accessed 21-May-2019].

[2] Paolo Abeni. Patch: net: mitigate retpoline overhead. Linux Kernel Mailing List, https://lwn.net/ml/linux-kernel/cover.1544032300.git.pabeni@redhat.com/, 2018. [Online; accessed 21-May-2019].

[3] Varun Agrawal, Amit Arya, Michael Ferdman, and Donald Porter. Jit kernels: An idea whose time has (just) come. http://compas.cs.stonybrook.edu/~mferdman/downloads.php/SOSP13_JIT_Kernels_Poster.pdf. [Online; accessed 21-May-2019].

[4] Nadav Amit. Rfc dynamic indirect call promotion. Linux Kernel Mailing List, https://lkml.org/lkml/2018/10/18/175, 2018. [Online; accessed 21-May-2019].

[5] Nadav Amit. PATCH: x86: text_poke() fixes and executable lockdowns. https://lore.kernel.org/patchwork/cover/1067359/, 2019. [Online; accessed 21-May-2019].

[6] Andrew Ayers, Richard Schooler, and Robert Gottlieb. Aggressive inlining. In *ACM SIGPLAN Notices*, volume 32, pages 134–145. ACM, 1997.

[7] Ivan Baev. Profile-based indirect call promotion. 2015. [Online; accessed 21-May-2019].

[8] Robert L Bernstein. Producing good code for the case statement. *Software: Practice and Experience*, 15(10):1021–1024, 1985.

[9] Brad Calder and Dirk Grunwald. Reducing indirect function call overhead in C++ programs. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 397–408. ACM, 1994.

[10] Dehao Chen, David Xinliang Li, and Tipp Moseley. AutoFDO: Automatic feedback-directed optimization for warehouse-scale applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, pages 12–23. ACM, 2016.

[11] Dehao Chen, Neil Vachharajani, Robert Hundt, Xinliang Li, Stephane Eranian, Wenguang Chen, and Weimin Zheng. Taming hardware event samples for precise and versatile feedback directed optimizations. *IEEE Transactions on Computers*, 62(2):376–389, 2013.

[12] Jonathan Corbet. Finding spectre vulnerabilities with smatch. Linux Kernel Mailing List, `https://lwn.net/Articles/752408/`, 2018. [Online; accessed 21-May-2019].

[13] Intel Corporation. Intel 64 and IA-32 architectures optimization reference manual. 6 2016. [Online; accessed 21-May-2019].

[14] Intel Corporation. Intel analysis of speculative execution side channels. `https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf`, 1 2018. [Online; accessed 21-May-2019].

[15] Intel Corporation. Retpoline: A branch target injection mitigation - white paper. `https://software.intel.com/sites/default/files/managed/1d/46/Retpoline-A-Branch-Target-Injection-Mitigation.pdf`, 2018. [Online; accessed 21-May-2019].

[16] Intel Corporation. Speculative execution side channel mitigations. 5 2018. [Online; accessed 21-May-2019].

[17] Matthew Dillon. Clarifying the Spectre mitigations... `http://lists.dragonflybsd.org/pipermail/users/2018-January/335637.html`, 2018. [Online; accessed 21-May-2019].

[18] Timothy Garnett. *Dynamic optimization if IA-32 applications under DynamoRIO*. PhD thesis, Massachusetts Institute of Technology, 2003.

[19] Thomas Gleixner. x86/cpu_entry_area: Move it out of fixmap. Linux Kernel Mailing List, `https://lore.kernel.org/patchwork/patch/866046/`, 2017. [Online; accessed 21-May-2019].

[20] Byron Hawkins, Brian Demsky, Derek Bruening, and Qin Zhao. Optimizing binary translation of dynamically generated code. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 68–78. IEEE Computer Society, 2015.

[21] Christoph Hellwig. RFC: avoid indirect calls for DMA direct mappings v2. Linux Kernel Mailing List, `https://lwn.net/ml/linux-kernel/20181207190720.18517-1-hch@lst.de/`, 2018. [Online; accessed 21-May-2019].

[22] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *European Conference on Object-Oriented Programming*, pages 21–38. Springer, 1991.

[23] Urs Hölzle and David Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *ACM SIGPLAN Notices*, volume 29, pages 326–336. ACM, 1994.

[24] Intel Corporation. Intel 64 and IA-32 architectures optimization reference manual, 2016.

[25] Jose A Joao, Onur Mutlu, Hyesoon Kim, Rishi Agarwal, and Yale N Patt. Improving the performance of object-oriented languages with dynamic predication of indirect jumps. In *ACM SIGOPS Operating Systems Review (OSR)*, volume 42, pages 80–90, 2008.

[26] Teresa Johnson, Mehdi Amini, and Xinliang David Li. Thinlto: scalable and incremental lto. In *Code Generation and Optimization (CGO), 2017 IEEE/ACM International Symposium on*, pages 111–121. IEEE, 2017.

[27] Piyus Kedia and Sorav Bansal. Fast dynamic binary translation for the kernel. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 101–115. ACM, 2013.

[28] Hyesoon Kim, José A Joao, Onur Mutlu, Chang Joo Lee, Yale N Patt, and Robert Cohn. Vpc prediction: reducing the cost of indirect branches via hardware-based dynamic devirtualization. In *ACM SIGARCH Computer Architecture News (CAN)*, volume 35, pages 424–435, 2007.

[29] Andi Kleen. Add a text_poke syscall. LWN.net https://lwn.net/Articles/574309/, 2013. [Online; accessed 21-May-2019].

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

[31] Esmaeil Mohammadian Koruyeh, Khaled Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. *arXiv preprint arXiv:1807.07940*, 2018.

[32] David Xinliang Li, Raksit Ashok, and Robert Hundt. Lightweight feedback-directed cross-module optimization. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 53–61. ACM, 2010.

[33] Tao Li, Ravi Bhargava, and Lizy Kurian John. Adapting branch-target buffer to improve the target predictability of Java code. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2(2):109–130, 2005.

[34] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[35] Larry W McVoy, Carl Staelin, et al. lmbench: Portable tools for performance analysis. In *USENIX annual technical conference*, pages 279–294. San Diego, CA, USA, 1996.

[36] Microsoft. Mitigating speculative execution side channel hardware vulnerabilities. https://blogs.technet.microsoft.com/srd/2018/03/15/mitigating-speculative-execution-side-channel-hardware-vulnerabilities/, 2018. [Online; accessed 21-May-2019].

[37] MIT Technology Review. At least three billion computer chips have the spectre security hole. https://www.technologyreview.com/s/609891/at-least-3-billion-computer-chips-have-the-spectre-security-hole/, 1 2018. [Online; accessed 21-May-2019].

[38] MITRE. CVE-2017-5715: branch target injection, spectre-v2. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5715, 2018. [Online; accessed 21-May-2019].

[39] MITRE. CVE-2017-5753: bounds check bypass, spectre-v1. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-5753, 2018. [Online; accessed 21-May-2019].

[40] Paweł Moll. FDO: Magic "make my program faster" compilation option? https://elinux.org/images/4/4d/Moll.pdf, 2016. [Online; accessed 21-May-2019].

[41] Sungdo Moon, Xinliang D Li, Robert Hundt, Dhruva R Chakrabarti, Luis A Lozano, Uma Srinivasan, and Shin-Ming Liu. Syzygy-a framework for scalable cross-module ipo. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 65. IEEE Computer Society, 2004.

[42] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. Bolt: A practical binary optimizer for data centers and beyond. *arXiv preprint arXiv:1807.06735*, 2018.

[43] Josh Poimboeuf. Patch: Static calls. Linux Kernel Mailing List, https://lkml.org/lkml/2018/11/26/951, 2018. [Online; accessed 21-May-2019].

[44] Michiel Ronsse and Koen De Bosschere. JiTI: A robust just in time instrumentation technique. *ACM SIGARCH Computer Architecture News*, 29(1):43–54, 2001.

[45] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *ACM Conference on Computer and Communications Security (CCS)*, pages 298–307. ACM, 2004.

[46] Ryan Smith. Intel publishes spectre & meltdown hardware plans: Fixed gear later this year. https://www.anandtech.com/show/12533/intel-spectre-meltdown, 2018. [Online; accessed 21-May-2019].

[47] Linus Torvalds. Create macros to restrict/unrestrict indirect branch speculation. 2018. [Online; accessed 21-May-2019].

[48] Linus Torvalds. x86/speculation: Add basic IBRS support infrastructure. 2018. [Online; accessed 21-May-2019].

[49] Paul Turner. Retpoline: a software construct for preventing branch-target-injection. https://support.google.com/faqs/answer/7625886, 2018. [Online; accessed 21-May-2019].

[50] Vertica. Update: Vertica test results with microcode patches for the meltdown and spectre security flaws. 2018. [Online; accessed 21-May-2019].

[51] David Woodhouse. Fill RSB on context switch for affected cpus. https://lkml.org/lkml/2018/1/12/552, 2018. [Online; accessed 21-May-2019].

[52] David Woodhouse. x86/retpoline/entry: Convert entry assembler indirect jumps. Linux Kernel Mailing List, https://lore.kernel.org/patchwork/patch/876057/, 2018. [Online; accessed 21-May-2019].

[53] David Woodhouse. x86/speculation: Add basic IBRS support infrastructure. 2018. [Online; accessed 21-May-2019].

[54] Pengfei Yuan, Yao Guo, and Xiangqun Chen. Experiences in profile-guided operating system kernel optimization. In *ACM Asia-Pacific Workshop on Systems (APSys)*, page 4, 2014.

[55] Pengfei Yuan, Yao Guo, and Xiangqun Chen. Rethinking compiler optimizations for the Linux kernel: An explorative study. In *Proceedings of the 6th Asia-Pacific Workshop on Systems*, page 2. ACM, 2015.