Switchpoline: A Software Mitigation for Spectre-BTB and Spectre-BHB on ARMv8

Markus Bauer*
markus.bauer@cispa.de
CISPA Helmholtz Center for Information Security

Lorenz Hetterich*
lorenz.hetterich@cispa.de
CISPA Helmholtz Center for Information Security

Christian Rossow
christian.rossow@cispa.de
CISPA Helmholtz Center for Information Security

Michael Schwarz
michael.schwarz@cispa.de
CISPA Helmholtz Center for Information Security

ABSTRACT

Spectre-BTB, also known as Spectre Variant 2, is often considered the most dangerous Spectre variant. While there are widely-deployed software workarounds on x86, such as Retpoline, there are no automated software workarounds for protecting generic userspace applications on ARMv8. Moreover, hardware solutions do not consider in-place mistraining or variants such as branch-history injection (Spectre-BHI), also known as Spectre-BHB.

In this paper, we introduce Switchpoline, the first automated Spectre-BTB and Spectre-BHB software workaround protecting C and C++ userspace applications on ARMv8 against all variants of Spectre-BTB and Spectre-BHB. The main security of Switchpoline is that eliminating indirect branches eliminates attacks on indirect branches. Switchpoline is based on a static compiler pass and a dynamic just-in-time (JIT) compiler component that rewrite indirect control-flow transfers into direct control-flow transfers. Switchpoline successfully prevents Spectre-BTB and Spectre-BHB in userspace applications with a negligible mean performance overhead of 1.8 % measured in the SPEC CPU 2017 benchmark. Moreover, unlike many x86-specific mitigations, Switchpoline is compatible with existing orthogonal defenses, such as (hardware) CFI or Spectre-PHT mitigations. Hence, Switchpoline is a practical generic software mitigation on ARMv8.

CCS CONCEPTS

• **Security and privacy → Side-channel analysis and countermeasures**.

KEYWORDS

Microarchitecture, Spectre, Mitigation, ARMv8

ACM Reference Format:
Markus Bauer, Lorenz Hetterich, Christian Rossow, and Michael Schwarz. 2024. Switchpoline: A Software Mitigation for Spectre-BTB and Spectre-BHB on ARMv8. In *Proceedings of the 2024 ACM ASIA Conference on Computer and Communications Security (ASIA CCS '24), July 1–5, 2024, Singapore.* ACM, New York, NY, USA, 14 pages. https://doi.org/XXXXXXXX

*Both authors contributed equally to the paper.

1 INTRODUCTION

Since the discovery of Spectre [30], there has been ongoing research into solutions to mitigate this class of attacks. From early on, a huge focus was on mitigating Spectre-BTB [17, 30], also known as Spectre Variant 2. This variant is deemed especially dangerous, as any indirect call can be exploited to change the control flow during speculative execution arbitrarily. The original Spectre paper [30] shows that such an indirect call in the kernel suffices to leak memory from a different userspace application, kernel memory from an unprivileged application, and hypervisor memory from a VM.

Like research on transient-execution attacks [17], research on defenses also primarily focuses on the x86 platform. For x86, ISA extensions on most CPUs control certain aspects of speculative execution [2, 25, 28]. These ISA extensions allow the kernel to flush the branch predictor's state and to enable high-privilege modes where the predictor is not influenced by code running in lower privileges [23]. As a software-only mitigation, Google introduced Retpoline [49] to prevent Spectre-BTB attacks. Retpoline converts indirect branches into returns, exploiting the design of the return predictor on x86 [24, 49]. Retpoline is integrated into the LLVM compiler framework to protect userspace applications [18] and the Linux kernel [51]. JumpSwitches [4] improves the performance of Retpolines by promoting some indirect calls to direct calls, relying on Retpolines as fallback for branches that are not promoted.

While mitigations exist on x86 CPUs, they do not apply to ARMv8, even though many Arm CPUs are also affected by Spectre [9, 17, 30]. Although the attack surface is similar, no generic software solution exists for ARMv8 [6]. There are only ad-hoc solutions for the Linux kernel and firmware [6], which try to eliminate indirect calls manually on a case-by-case basis or add fences after specific code patterns [8]. CSV2 [6], an ISA extension in the form of a bitfield in a CPU register, allows changing the isolation of the branch predictor. If enabled, CSV2 prevents lower-privilege domains from influencing predictions of higher-privilege domains, such as the kernel. However, CSV2 provides limited security guarantees. Indeed, Arm acknowledges that *"the CSV2 hardware features introduced to mitigate against Spectre v2 do not work against Spectre-BHB"* [7]. But also for Spectre-BTB, Arm admits that *"there is no generic mitigation available that applies to all Arm CPUs"* [5]. Worse, CSV2 is ineffective against in-place mistraining in Spectre-BTB. Yet we show that Spectre-BTB affects a wide range of ARMv8 CPUs, even those not documented as vulnerable to Spectre-BTB [9], like the Apple M1 processor. We develop a proof of concept (PoC) that works on a wide range of CPUs, leaking up to 20 kB/s, which motivates the need for better defenses.

In this paper, motivated by these findings, we propose a generic open-source[1] software-only defense for C/C++ programs dubbed Switchpoline that mitigates exploitation of Spectre-BTB and Spectre-BHB. The high-level idea of Switchpoline is to replace indirect control-flow transfers with direct ones. Intuitively, an attacker cannot exploit indirect-branch mispredictions as required by Spectre-BTB and Spectre-BHB if there are no indirect branches in the victim. Using an LLVM compiler pass, we can statically reduce the possible targets of indirect control-flow transfers drastically. For such statically-resolved targets, we emit direct control-flow transfers in combination with comparisons laid out in an efficient manner. While such static analysis works well for most applications – in fact, all that we have evaluated – it does not guarantee completeness. Hence, for these rare cases, we augment the compiled binaries with a minimal just-in-time (JIT) compiler that emits a direct control-flow transfer to the target destination. In line with mitigations on x86, Switchpoline only requires recompilation of applications to secure it against Spectre-BTB and Spectre-BHB attacks. Switchpoline retains compatibility with dynamic linking. Switchpoline is also compatible with CFI, which is not true for Retpoline on x86 [15].

Switchpoline is the first automated, software-based Spectre-BTB and Spectre-BHB defense on Arm—and the only one that protects userspace applications. To demonstrate that Switchpoline successfully eliminates indirect control-flow transfers, we apply Switchpoline to lighttpd and the SPEC CPU 2017 benchmark, showing no impact on their functionality, while no indirect control-flow transfers remain in the protected binaries.

We also evaluate the performance of Switchpoline in terms of one-time compile and runtime overhead. Based on the SPEC CPU 2017 benchmark, the one-time compile-time overhead is 1.6 s on average, excluding some outliers. We evaluate the runtime overhead using both micro and macro benchmarks. Our microbenchmarks show an increase below 50 % for an indirect control-flow transfer with 450 possible branch targets and a mean decrease for control-flow transfers with fewer than 54 targets. However, indirect control-flow transfers with many possible targets are relatively rare in real-world applications. Hence, the cost of Switchpoline is amortized in typical applications. Our macro benchmark using SPEC CPU 2017 only shows an average overhead of 1.8 %. We also compare the performance of Switchpoline with a *jump/fence* approach. The latter is slower than Switchpoline even for branches with 1000 targets, and it is unclear whether it suffices to mitigate Spectre-BTB.

Switchpoline is the first automated software mitigation on ARMv8 that protects indirect control-flow transfers in userspace applications against Spectre-BTB and Spectre-BHB. Based on our security and performance evaluation, we show that Switchpoline is indeed a practical solution. Given the widespread deployment of ARMv8 devices, both in smartphones and with the Apple M1 also in laptops, we deem such a solution necessary until sufficient hardware mitigations are developed and deployed.

[1]Source code: github.com/cispa/Switchpoline

Contributions. The contributions of this paper are:

(1) We present a Spectre-BTB PoC implementation and demonstrate that it can leak up to 20 kB/s on several ARMv8 devices, motiving the need for a defense.
(2) We design Switchpoline, the first generic software-based countermeasure that protects Arm userspace applications against Spectre-BTB and Spectre-BHB.
(3) We show that Switchpoline is effective and only incurs an average overhead of 1.8 % in macro benchmarks.

Outline. Section 2 provides background. Section 3 motivates Switchpoline. Section 4 defines the threat model. Section 5 describes Switchpoline's design, and Section 6 evaluates its performance and security. Section 7 discusses limitations. Section 8 concludes.

2 BACKGROUND

This section provides background on transient execution, Spectre, and existing Spectre mitigations.

2.1 Transient Execution

To prevent pipeline stalls, modern CPUs execute instructions out-of-order and speculatively while retiring them in application order. Out-of-order execution and speculative execution are both instances of *transient execution*. Transiently-executed instructions, so-called *transient instructions* [17, 30, 33], are executed but do not have an architectural effect. If a transient instruction is erroneously executed, e.g., due to a branch misprediction, its results are discarded. Similarly, wrongly raised exceptions are not visible to the application but only result in a pipeline flush. However, microarchitectural state changes, e.g., the cache state of a cache line, are generally not reverted. Transient execution attacks rely on side channels to reveal these microarchitectural state changes to the architecture. There are two types of transient execution attacks: Spectre-type attacks, which exploit mispredictions of control- or data-flow [30], and Meltdown-type attacks, which exploit delayed exception handling during out-of-order execution [33].

2.2 Spectre

Spectre [30] is a transient-execution attack exploiting speculative execution due to mispredictions. Instructions following a misprediction may perform unintended data accesses, e.g., out-of-bound access to an array, and then encode the illegally accessed data into a microarchitectural state. A side channel exposes this microarchitectural state to the architecture, leaking data.

Spectre Variants. Spectre variants are classified based on the targeted predictor [17]. The two most prevalent variants are Spectre-PHT and Spectre-BTB. Spectre-PHT exploits the pattern history table that predicts whether a conditional branch is taken [17]. Spectre-BTB exploits the branch target buffer (BTB) that predicts the destinations of indirect branches. The mistraining can happen within the same address space as the victim (same-address-space) or in a different address space (cross-address-space) using the same addresses as the victim (in-place) or different addresses (out-of-place) [17]. Out-of-place mistraining relies on collisions in the predictor's data structure. The BTB only has a limited number of entries. Thus, multiple branch instructions at different addresses map to the same entry [30]. In modern CPUs, the global branch history indexes the

```
br x8 ; mispredicted target is "gadget"
[...]
gadget:
 ldr x10, [shared_array, x11] ; x11 is a secret value
```

Listing 1: A mispredicted branch executing a cache-based Spectre gadget encoding a secret value into the cache state.

predictors data structure [52]. Spectre-BHB [11] injects a malicious global branch history to create collisions in the predictor's data structure.

Spectre Gadgets. The instruction sequence which encodes data into a microarchitectural state is called a Spectre gadget. Cache-based Spectre gadgets are the most common, even though several other types of Spectre gadgets exist [14, 31, 41, 50]. A simple cache-based Spectre gadget is a memory access to a shared array, indexed by the data to encode (cf. Listing 1). The memory access causes the secret-dependent entry of the array to be cached. An attacker can measure the access time to every entry of the shared array. The access to the cached entry is the fastest, as it is served from the cache. As the index of the cached entry is the encoded data, an attacker infers the data.

2.3 Spectre Mitigations

Several software-based mitigations against Spectre attacks have been proposed. These mitigations aim to prevent the exploitation of Spectre on affected CPUs. Speculative load hardening [19] masks pointers during speculative execution such that they are invalid in mispredicted control flow, protecting against Spectre-PHT. On x86 CPUs, all branches can be removed, mitigating multiple Spectre variants but usually with a high performance penalty [43]. This mitigation could also be applied to Arm CPUs. However, an overhead of up to several orders of magnitude is not feasible for widespread deployment. ConTExT [40] ensures that secrets cannot be accessed during speculative execution. Although this approach prevents all Spectre attacks, it requires a developer to identify secrets in the application, which is not straightforward. Similarly, approaches such as process isolation [39, 42] require secrets to be moved to different processes. Retpoline [49] transforms indirect branches into return instructions to protect against Spectre-BTB on x86 CPUs. According to Arm, Retpoline is not applicable to Arm CPUs [5]. JumpSwitches [4] improves the performance of Retpoline in x86 Linux kernels by promoting some indirect calls to direct calls instead of replacing them with Retpoline constructs. While JumpSwitches also replaces indirect branches with direct branches, JumpSwitches only targets the Linux kernel and cannot protect userspace applications. With Retpoline, JumpSwitches has a safe fallback and does not need to consider all indirect branches. JumpSwitches relies on dynamic runtime profiling to determine which calls to promote and what construct to use. Since Retpolines can branch to arbitrary addresses, JumpSwitches does not need to consider limited branches. According to Arm [5], Retpoline is not applicable to Arm CPUs because it relies on specifics in the microarchitecture that do not apply to Arm CPUs. Hence, JumpSwitches is also not applicable to Arm CPUs. Moreover, even on x86, Retpoline is not effective in all scenarios [51]. Randpoline [15] uses trampolines to randomize the location of indirect branches in memory, making attacks more difficult. While Randpoline raises the bar for successful attacks, it does not fully mitigate Spectre-BTB. The `jmp/lfence` mitigation used on AMD CPUs turned out to suffer from race conditions, drastically reducing its effectiveness [37]. To summarize, all these mitigations only work on x86 or do not fully mitigate Spectre-BTB.

Table 1: Leakage and error-rate of our Spectre-BTB PoC.

Device	CPU	Leakage [kB/s]	Errors [%]
ODROID-N2+	Cortex A73	13.46	0.16
Realme 3 Pro	Kryo 360 (Cortex A75)	19.83	0.02
Xiaomi Mi 9t	Kryo 470 (Cortex A76)	20.08	0.01
Apple Mac Mini	Apple M1	11.01	6.73
Apple Mac Studio	Apple M1	8.24	5.60

As hardware mitigation, CSV2 [6] prevents mistraining of the branch predictor across security contexts or software contexts on Arm. However, as we discuss in Section 3.2, the security guarantees are limited. It is not available on all CPUs, does not mitigate mistraining that does not cross security boundaries, and is ineffective against Spectre-BHB [7].

3 MOTIVATION OF SWITCHPOLINE

To demonstrate the lack of protection of Arm user programs against Spectre-BTB attacks, and therefore also Spectre-BHB attacks, we implement a Spectre-BTB PoC for a wide range of ARMv8 devices.

3.1 Spectre-BTB PoC

Our Spectre PoC for ARMv8 CPUs is based on the source from Hetterich and Schwarz [22]. The PoC emphasizes that Spectre-BTB is even relevant if hardware countermeasures such as CSV2 [6] are available or if the CPU is officially declared as being not susceptible to Spectre-BTB. For the PoC, we rely on in-place same-address-space mistraining [17], a variant that was already exploited in browsers [30, 44, 45], and the cloud [42]. This type of Spectre is hard to prevent, as no hardware security boundary is crossed during the exploit. Thus, it is also out of scope for CSV2.

PoC Overview. We use the same branch instruction to poison the BTB and to trigger speculative execution of the Spectre gadget. The victim contains an indirect branch caused by a function pointer that calls a dummy function or a function containing a Spectre gadget that leaks an array's content, similar to the original Spectre paper [30]. The PoC uses a cache-based bit-wise Spectre gadget similar to related work [22, 41, 44]. The encoded entry is transferred to the architectural level via Flush+Reload or Evict+Reload, depending on the CPU support [22]. The PoC supports a counter thread, the `clock_gettime` library function, and the system counter (`CNTVCT_EL0`) for precise time measurement. The victim's code is summarized in Listing 7 (Appendix B).

Results. We evaluate the PoC on 5 ARMv8 devices: The ODROID-N2+ with a Cortex A73 CPU, the Realme 3 Pro with a Kryo 360 CPU (based on the Cortex A75), the Xiaomi 9t with a Kryo 470

CPU (based on the Cortex A76), the Apple Mac Mini with an Apple M1 CPU, and the Apple Mac Studio with an Apple M1 Max CPU. All devices except the Apple Mac Mini and the Apple Mac Studio allow unprivileged flushing and provide accurate timing through the `clock_gettime` library function. We rely on eviction and a dedicated counter thread for the Apple Mac Mini and the Apple Mac Studio. The PoC works on all devices and achieves leakage rates beyond 8 kB/s (Table 1). The error rates for the two smartphones and the single-board computer are below 0.1 %, and the leakage rate scales with CPU power. The leakage rate for the Apple Mac Mini is 11.01 kB/s, and the error rate is higher at 6.73 %. The lower rate is because we have to rely on eviction, which is slower and less reliable than flushing, and a counter thread, which is susceptible to noise. The same applies to the Apple Mac Studio, with a leakage rate of 8.24 kB/s and an error rate of 5.60 %. Still, the results show that unprivileged Spectre-BTB attacks are a threat, and mitigations such as Switchpoline are necessary.

3.2 Applicability of Mitigations

As the software mitigations discussed in Section 2.3, such as Retpoline, do not apply to Arm, we focus on hardware mitigations in this section. Arm introduced a barrier to prevent speculation (`CSDB`). However, this instruction is a `nop` on older CPUs, like the Apple A10 Fusion, and introduces significant overhead on other devices like the Apple M1. Thus, this barrier cannot be used universally as it does not work on older devices. In our PoC, inserting a `DSB` barrier before the mispredicted branch stops leakage on all devices. While it is unclear whether this is sufficient to mitigate Spectre-BTB fully, we still analyze the overhead in a micro-benchmark and compare it to Switchpoline in Section 6.1.1. In our micro-benchmark, protecting indirect branches with barriers has an overhead of over 100 % compared to unprotected indirect branches. In contrast, as Figure 2 shows, Switchpoline improves performance by rewriting indirect branches for calls with fewer than 54 possible targets. With CSV2 [6], Arm introduced a hardware mitigation. Depending on hardware support, CSV2 either mitigates mistraining across privilege boundaries (*CSV2=1*) or can even inhibit mistraining across software contexts, i.e., processes (*CSV2=2*) [10]. Still, mistraining that does not cross process boundaries (in-place mistraining [17]), and Spectre-BHB are not mitigated by CSV2. Additionally, only recent CPUs fully support CSV2. For browsers, Google proposes mitigations for the JIT compiler and heavily relies on site isolation [21]. However, approaches like site isolation do not apply to general user applications and require additional deployment effort. Thus, it is important to provide a mitigation such as Switchpoline that works without special hardware support and protects user applications against attacks that do not cross process boundaries (Table 2).

Table 2: Applicability of Spectre-BTB mitigations for Arm devices. Google's mitigations [21] prevent attacks from JITted code in the browser. CPUs implementing CSV2 can protect against mistraining across privilege levels. CPUs that report CSV2=2 can prevent mistraining across software contexts. Switchpoline protects user applications from any Spectre-BTB attack, including Spectre-BHB.

Target / Attacker	Userspace	Browser	Kernel
Native (Different context)	**our work** CSV2=2 (only BTB)	CSV2=2	CSV2
Native (Same context)	**our work**	N/A	manual patches
Browser	Google's mitigations	Google's mitigations	CSV2

4 ATTACKER MODEL

We base our attack model on the attacker model used in previous Spectre attacks [17, 30].

Protected Targets. Switchpoline protects userspace applications on ARMv8 against an unprivileged attacker exploiting Spectre-BTB [30]. For Switchpoline, all variants of Spectre-BTB are in scope, i.e., same-address-space and cross-address-space target injection, both in-place and out-of-place [17], and Spectre-BHB attacks. We do not rely on Spectre-related hardware features, such as CSV2 [6], as they are not implemented on all vulnerable CPUs and cannot protect user applications. Moreover, there is no OS or firmware support required for Switchpoline. While Switchpoline does not mitigate other Spectre variants, such as Spectre-PHT [30] or straight-line speculation [8], it is compatible with other mitigations. For example, the automated insert of speculation barriers (`CSDB` instructions) or speculative load hardening [19] do not conflict with Switchpoline.

Protection Scope. When referring to indirect control-flow transfers, we refer to *forward* transfers such as branches or calls in this paper. Switchpoline does not protect indirect *backward* transfers (return statements). While Spectre-BTB-style attacks using returns were demonstrated on other platforms [51], no details or PoCs are available for Arm. It is unclear whether this is a realistic attack vector. Still, returns can be rewritten into indirect branches [27]. These rewritten branches can be protected using Switchpoline.

Attacker Capabilities. The attacker can execute unprivileged native code on the system. Moreover, the attacker can enforce colocation with the victim. We also assume that the attacker can share memory with the victim, e.g., using a shared library, or use any other covert channel demonstrated in Spectre PoCs [14, 30–32, 41, 48].

Orthogonal Mitigations. Switchpoline is compatible with orthogonal mitigations deployed against memory safety vulnerabilities, such as CFI, ASLR, stack canaries, and pointer authentication for data pointers and other Spectre mitigations like Speculative Load Hardening [19].

5 DESIGN OF SWITCHPOLINE

This section introduces the design of Switchpoline and its challenges (Section 5.1-Section 5.5), as well as implementation details of our prototype (Section 5.6). The main idea of Switchpoline is to eliminate all indirect branches at compile time, as illustrated in Figure 1. Without indirect branches in the binary, the prerequisite of Spectre-BTB, and thus also Spectre-BHB, is eliminated. To eliminate branches, indirect branch locations are statically resolved and replaced by switches containing direct calls. As static analysis is inherently incomplete, we introduce a combination of over-approximation and a dynamic live-patching approach. As a

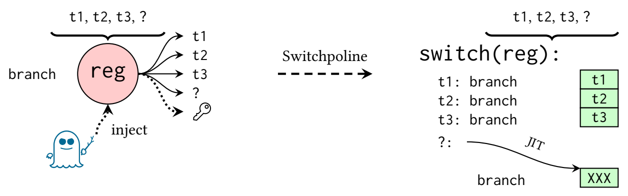

Figure 1: Switchpoline: Statically determine indirect branch targets (t1, t2, t3) and rewrite them into a switch of direct branches unaffected by Spectre-BTB. A small JIT emits direct branches for call destinations not resolvable at compile time (?). Only indirect branches (red circle) are prone to injected targets for misspeculation (dotted arrow).

result, no indirect branches are present. Thus, Spectre-BTB and Spectre-BHB are mitigated. Other Spectre variants, such as Spectre-PHT [30] or Spectre-RSB [36], are out of scope for Switchpoline. However, in line with other Spectre mitigations [15, 23, 49], we only focus on one variant, as mitigations against other variants are orthogonal. Moreover, Switchpoline aims to be compatible with mitigations against other variants. Hence they can be used with Switchpoline to protect against more Spectre variants.

5.1 Challenges

To be compatible with a wide range of applications, Switchpoline is implemented directly in the compiler and does not rely on hardware mechanisms. Still, while the idea of eliminating all branches sounds straightforward, multiple challenges have to be overcome. This section introduces the main challenges.

$C1$**: Sources of Indirect Branches**. Every indirect branch instruction reachable during execution can be mispredicted and thus is a possible entry point for Spectre-BTB attacks. To fully mitigate Spectre-BTB and Spectre-BHB, we must eliminate all those indirect branches, namely all `br` and `blr` instructions. In a systematic investigation, we found that compilers and linkers emit `br` and `blr` instructions in four situations. To get programs free of indirect branches, we must implement these situations without relying on indirect branches: Indirect calls in C/C++, including virtual dispatch, compiler-generated jump tables, PLT stubs for dynamic linking, and raw assembly in the standard library.

For an effective Spectre-BTB and Spectre-BHB mitigation, *all* indirect branches that may be executed must be eliminated. Hence, changing the compiler, the linker, and the standard library is necessary to ensure that no executable indirect branch remains. Given the different stages where Switchpoline has to apply the mitigation, we have to deal with source code, intermediate representation, and machine code.

$C2$**: Target Set Computation**. The set of possible targets must be identified to rewrite indirect branches to direct branches at compile time, This challenge is a well-known research problem, especially for control-flow integrity (CFI) [1, 12]. The correctness of CFI depends on correctly identifying indirect branch targets at compile time. Any missed location leads to a program crash at runtime, as this is considered a security violation. Similarly, Switchpoline also needs to infer all possible branch targets. Like CFI, over-approximating the set of targets leads to a larger attack surface by increasing the number of possible targets on branch mispredictions. However, missed targets do not lead to program crashes but to a costly invocation of the JIT component. Hence, the challenge is to identify a set of targets that is neither too large nor too small to guarantee a low performance overhead and maximum security guarantees for Switchpoline.

$C3$**: Dynamic Code**. Static analysis of branch destinations is inherently incomplete. While over-approximation of targets is widely used to "hide" this problem, it still cannot guarantee functional correctness. For example, static analysis in CFI cannot handle functions loaded at runtime. To ensure functional correctness, Switchpoline must be able to expand target sets dynamically. Hence, Switchpoline must support adding targets from dynamically loaded libraries and dealing with function pointers to dynamically generated code, e.g., from a JIT compiler.

$C4$**: Limited Direct Branches**. In contrast to x86's complex instruction set with instructions ranging from 1 to 15 bytes [26], the ARMv8 instruction set only supports 4 B instructions. While this is not a problem for indirect calls, it severely limits direct calls. Direct calls require the branch offset to be encoded in the opcode. However, with the instruction's limited size, it is impossible to encode arbitrary 64 bit addresses into the instruction. As a result, direct branches are limited to jumping ±128 MB relative to the current instruction pointer. Hence, Switchpoline has to avoid jumps that are further than this offset, e.g., caused by dynamically-loaded shared libraries or large code segments.

5.2 Sources of Indirect Branches ($C1$)

This section describes our approaches to eliminating all indirect branches.

```c
int f1(int x) { ... }
int f2(int x) { ... }
int f3(int x) { ... }
typedef int (*function_pointer)(int);
function_pointer fp = &f1; // or &f2 or &f3
fp(0); // indirect call
```

Listing 2: C code example with an indirect call.

5.2.1 Indirect Calls. The compiler requires an indirect branch instruction to generate assembly for indirect function calls. Listing 2 shows an example of an indirect call in C. Indirect calls have their target, i.e., the address of the function's first instruction, stored in a function pointer. From the source code, we can infer the possible legal values of a function pointer: Only the addresses of functions are legal values, and only expressions such as `&f1` generate function pointers. By analyzing the entire source code, we can identify all *address-taken* functions: functions whose address is used as a function pointer. We know all possible values for each function pointer, and therefore, we know all possible targets for each indirect call: the *target set*. This target set is an over-approximation of the actual target set, but at the same time, misses targets that are not available during the static analysis pass, e.g., dynamically-loaded libraries.

Indirect Call to Direct Call Conversion. For a naïve implementation, we can translate every indirect call into a series of direct

```
// translated indirect call
if (fp == &f1)      f1(0);
else if (fp == &f2) f2(0);
else if (fp == &f3) f3(0);
```

Listing 3: Source-level transformation of the example. The indirect call is replaced by a series of direct calls.

```
switch (fp) {
    case &f1:  f1(0);  break;
    case &f2:  f2(0);  break;
    case &f3:  f3(0);  break;
}
```

Listing 4: Pseudocode for the transformation idea: an efficient `switch` replaces multiple comparisons.

calls by comparing the function pointer to each possible value and calling the corresponding target function directly if the pointer matches. Listing 3 shows an example of such a translation. However, this translation is highly inefficient, even if implemented in the compiler. With such an implementation, the comparison time increases linearly with the size of the target set.

Hence, to boost the performance of the direct-call selection, we re-empower the compiler's algorithms for switch-case statements. Instead of a series of `if-then-else`, we use a single `switch` and a `case` for each possible target function, as shown by the pseudocode in Listing 4. The compiler's backend uses various strategies to convert this language construct to efficient assembly. Binary search trees are most prominent, but bitset tests or direct comparisons are also used if applicable. In particular, binary search trees reduce the worst-case runtime from $O(n)$ to $O(\log n)$.

Function Identifiers. While the general idea is straightforward, engineering challenges must be solved. Function addresses are not fixed during compile time and can vary between runs, e.g., if ASLR is enabled. Thus, function addresses are unsuitable for compiler algorithms that work on constant values, e.g., binary search trees.

To fix this, we use a property of function pointers: according to the C standard, programmers can only use them in indirect calls and comparisons with other function pointers. Accessing the memory behind a function pointer is considered undefined behavior. Hence, we can replace function pointers with arbitrary values if indirect calls and comparisons do not break. Like TyPro [12], Switchpoline assigns each address-taken function a constant integer, called the *function identifier*, as shown in Listing 5. Whenever a function's address is taken, we use this constant instead. The switch statements we generate for indirect calls use these constant identifiers for their cases. Function pointer comparisons need no change because each function has only one unique identifier. We can use fast and efficient switch-based assembly constructs with this change to replace indirect calls. The first function identifier of a binary is derived from a hash of the file name to avoid function-identifier collisions with dynamically linked libraries. Since we use 64-bit function identifiers, collisions are unlikely.

5.2.2 C++ Virtual Dispatch. C++ has an additional pattern that requires indirect branches: inheritance and virtual dispatch. In C++,

```
int f1(int x) { ... }  // ID 15
int f2(int x) { ... }  // ID 16
int f3(int x) { ... }  // ID 17
typedef int (*function_pointer)(int);
function_pointer fp = (function_pointer) 15;
// or 16 for f2, or 17 for f3

switch (fp) { // indirect call
    case 15:  f1(0);  break;
    case 16:  f2(0);  break;
    case 17:  f3(0);  break;
}
```

Listing 5: The final transformed example code, using function IDs instead of function pointers.

"child" classes can *inherit* from one or more "parent" classes. Every parent class method can also be called on a child class. If necessary, the child class can *override* methods from its parent, replacing them with their implementation. Overridable methods are called *virtual* methods and are marked in the source code with the keyword *virtual*. When a virtual method is invoked on a class instance (*virtual dispatch*), the compiler does not know in advance if the class instance is of parent or child type. Therefore, it cannot build a static call to the original or overridden method. All real-world C++ compilers solve this problem using *vtables* (virtual function tables) as specified in the Itanium C++ ABI [20]. Each class instance in memory starts with a pointer to a vtable. The vtable contains pointers to all virtual methods a class implements, whether inherited or overridden. The order of methods is similar in both parent and child classes. When calling a virtual method on a class instance, the compiler emits code that first loads the vtable pointer, then loads a pointer to the virtual method implementation from the vtable, and finally indirectly calls this pointer. There, the compiler introduces indirect branches.

In theory, we can resolve these branches similar to C-style function pointers: replace all method addresses in the vtables with IDs and replace all virtual dispatch with switches over the ID loaded from vtables. Our type-based target set computation from Section 5.3 can perfectly deal with C++ classes and method types. However, NoVT [13] proposes a faster solution that we adapt, which Listing 6 shows. We can replace the vtable pointers directly with class IDs instead of filling the vtable with function IDs. We can then dispatch virtual methods based on the class ID stored in an instance without referencing an additional table. We save one additional memory access and can empower the C++ type system to drastically reduce the number of possible targets.

To this end, we extract information about C++ classes, their inheritance, their virtual methods, and their memory layout from the source code. We compute the inheritance graph of all classes in an application and assign a unique identifier to each class. This information allows us to build switches over direct calls for virtual dispatch locations. Each time a virtual method is called on a class instance, we first load the identifier from memory, which is stored in the position that contained the vtable pointer before our changes. Then, we emit a `switch` over this value. We start traversing the inheritance graph at the class denoted by the dispatch's static type, e.g., the type of the instance as written in the source code. We

```
class Parent {
    virtual void f() { ... }
};
class Child : public Parent {
    void f() override { ... }
};

// virtual dispatch
Parent *instance = ...;
instance->f();
[...]
```

(a) C++ virtual dispatching example with insecure indirect calls.

```
class Parent {                  // ID 1
    virtual void f() { ... }
};
class Child : public Parent {  // ID 2
    void f() override { ... }
};

// virtual dispatch
Parent *instance = ...;
switch (instance->_id) {
    case 1:  Parent::f(instance);  break;
    case 2:  Child::f((Child*) instance);  break;
}
```

(b) Same example *after* transformation, without indirect calls.

Listing 6: C++ virtual dispatch before and after the transformation that rewrites indirect calls into `switch` constructs.

traverse the initial class and all classes that inherit from it. For each traversed class, we build a case with its ID. In the case body, we directly call that instance's inherited or overridden method implementation. We further track the memory layout of each inheritance and adjust class references if necessary.

Finally, we have collected all possible methods for a virtual dispatch in our switch. The target sets for virtual dispatch are minimal and complete concerning inheritance, thus solving $C2$. We also expect better performance than the naive approach. If many classes inherit the same method, we can join the case bodies, and the backend uses efficient bitset tests to target the correct method. The target sets of C-style function pointers are not affected because we can exclude C++ methods, avoiding performance penalties.

Replacing vtables does not only affect virtual dispatch: some class memory layouts require *virtual offsets*, which are stored in the vtable, too. Moreover, *runtime type information* (RTTI) features, such as exceptions, require additional information like the class name, which is also referenced by a pointer in the vtable. We replace all these usages with additional switches. We traverse the inheritance graph and generate cases that return each class's corresponding virtual offset or type information. Replacing vtables is still good for performance, as shown by NoVT [13], which reports a slight performance boost for a similar methodology on x86.

C++ has its own language feature along the lines of function pointers—method pointers. A method pointer is a reference to a possibly virtual method of a base class. When the program calls a method pointer on an instance, the corresponding, possibly overridden method of the instance's class is invoked. Despite replacing vtables with identifiers, indirect branches remain in code, handling method pointers. In the Itanium ABI [20], method pointers in memory are either a function pointer to a non-virtual method implementation or the vtable index of a virtual method. The code that invokes a method pointer either calls the function pointer directly or reads the method pointer from the vtable first. We change this process in two steps: First, when a method pointer is taken, we always store a function pointer—of a newly generated *dispatcher function* that contains only a C++ switch case invoking the taken method as outlined above. This step already fixes method pointers; they no longer rely on vtables. Second, we mark this function pointer as address-taken and let the C indirect call removal pass from Section 5.2 process it. We replace it with an additional switch similar to all normal C functions. To summarize, a method pointer invocation is handled by up to two switch-case constructs. The first switch branches over the concrete method pointer; it calls either a non-virtual method or a second switch. The second switch branches over the class ID of the underlying class instance and calls a virtual method. We can implement all C++ language constructs without indirect branches with this extension.

5.2.3 *Compiler-Generated Jump Tables.*

Compilers can introduce indirect branches as part of jump tables, even if not explicitly written by a programmer. The compiler generates jump tables mainly for larger `switch case` constructs in C. With jump tables, a switch can be implemented in a short construct, with execution time independent of the number of cases. Assuming `x0` holds the address reference to the pre-generated jump table and `x1` is the already bounds-checked value to switch over, the assembly of a switch can be `ldr x0, [x0, x1, lsl #3] ; br x0`. The second instruction is an indirect branch that is vulnerable to Spectre-BTB. We modify Clang's code generation backend never to emit jump tables. Instead, Clang now uses direct comparisons, binary trees, bitset tests, or combinations of these constructs. While these constructs are still pretty efficient, they might have a slight performance penalty — particularly for switches with many cases.

5.2.4 *PLT Stubs for Dynamic Linking.*

Compilers rely heavily on indirect branches to make dynamic linking possible. Each shared object, including the program itself, contains two special sections: The *global offset table* (GOT) and the *procedure linkage table* (PLT). The GOT contains one entry for each imported symbol. The dynamic loader fills this entry with the symbol's address, e.g., the function's address. The PLT contains one executable entry for each imported function that loads the address from GOT and tail-calls it. On Arm, each PLT entry ends with a vulnerable `br` instruction. We rewrite GOT usages to omit indirect branches.

In most cases, function pointers from the GOT are used in PLT entries. With Switchpoline, each module rewrites its PLT entries at runtime after the GOT is populated with a direct branch to the address from the GOT. Musl libc does not use lazy binding; therefore, it populates the GOT directly after loading the modules.

5.2.5 *Raw Assembly.*

After removing compiler-generated indirect branch instructions (Section 5.2 $C1$), branches explicitly written by a programmer remain—either in inline assembly or assembly files linked to the final program. The problem has no generic solution because assembly is harder to analyze than higher-level languages. However, this problem is rare: few to no userspace applications use hand-written indirect branches. We only detect three branches in the musl C standard library. Even all other low-level libraries, such as compiler-rt, libcxxabi, and libunwind are free of indirect branch instructions. Switchpoline provides utilities to the programmer to find and remove indirect branches from assembly code. First, we provide a script that disassembles generated binaries and reports any remaining indirect branches with their locations. Second, Switchpoline can generate dispatcher functions that assembly code can call instead of an indirect branch. With these techniques, we patch three locations in musl libc to remove indirect branches and ship these patches with our library environment described in Section 5.6. For example, musl's `__clone` assembly implementation calls a function using `blr x1`. We replace this instruction with `bl __switchpoline_handler_clone`, containing a generated switch of possible arguments of `clone`. The C function-pointer type of `x1` is inferred from the C declaration of `__clone`, and the target set of the handler is computed accordingly. If targets of programmer-written indirect branches are not C functions or cannot be determined at compile time, the JIT compiler (Section 5.4) generates direct branches on the fly. With our 3 one-line patches, Switchpoline builds all sources without indirect branches.

5.3 Target Set Computation ($C2$)

Switchpoline has to know the possible targets of each indirect call to build appropriate switches, the so-called *target sets*. Over-approximating the set of possible target functions is fine but might come with a performance penalty. Under-approximating the target requires the JIT and also comes with a performance penalty.

We start with the set of all address-taken functions, which contains all possible functions for all indirect calls, an over-approximation. We refine this set based on the number of function arguments and function types. We filter the target set for an indirect call by the number of arguments. A function with n parameters can only be in the target set of indirect calls with exactly n arguments. A variadic function with n declared parameters can only be in the target set of indirect calls with at least n arguments (Section 6.5.2.2 (2) and (6) of the C standard [29]). A similar condition for target set computation is used in IFCC [47], where an evaluation of large real-world programs shows that this condition is applicable.

We further refine our target sets using a type-based approach. The C standard states in 6.5.2.2 (6) that the function's parameters must be "compatible" with the argument types of an indirect call, and (9) requires the same for the return type. The C standard details compatible types in 6.2.7. However, real-world software sometimes uses incompatible types in indirect calls, for example, pointers and `long`. We thus relax the "compatible" types requirement to "assignable" types (C standard 6.5.16.1) and ignore type qualifiers. This condition is broad enough to cover all combinations of types that might occur in existing software. To this end, we use the following rules to check parameter and argument types. A pair of two types is valid if both types are (1) pointer type (ignoring the pointee), (2) integer type (ignoring width and sign), (3) floating-point types, (4) `void`, or (5) pointer, integer, or composite type and have the same size (in bits). According to these rules, a function is a valid target for an indirect call if all its parameters match the call's argument types and the function's return type matches the indirect call's return type. These rules can over-approximate the possible target sets but do not under-approximate them. Two types not following these rules inherently trigger differences in typical calling conventions, producing errors even in other, unmodified compilers.

We can evaluate both conditions on LLVM's intermediate representation without information from the frontend. The checks are fast and can be performed in little additional compilation time. Relying on the C standard avoids under-approximation of target sets and introduces no errors, while the computed target sets are smaller than related work [47] and guarantee compatibility.

5.4 Dynamic Code ($C3$)

Our static target-set analysis cannot take code loaded at runtime into account, resulting in two cases to consider at runtime. First, function identifiers can be passed between protected dynamically linked libraries. However, other libraries are not aware of foreign identifiers. These identifiers are not part of the statically computed target set at a call site outside the library. Secondly, function pointers to dynamically generated code can be passed to protected code. When called, such a pointer would not be part of the statically computed target set.

The JIT component. A small JIT component handles both of these problems. This component can expand the target set of call sites at runtime. To do so, the default cases of switches invoke a global dispatching sled. This sled is a dynamically expandable switch statement, which invokes the JIT component in the default case. Whenever the switch for a call site does not contain the destination, the global dispatching sled is invoked. If the destination is part of this sled, the global dispatching sled invokes the correct destination with a direct branch. Otherwise, the JIT component is invoked, adding a case for the missing target.

Handling function identifiers. Instead of adding one case per function identifier to the global dispatching sled, one case handling all function identifiers is added on library loading. This case checks whether the function identifier to invoke is in the range of function identifiers of the library. If so, it invokes a special `all_id_handler` function generated during library compilation. The `all_id_handler` is a compiler-generated switch over all address-taken functions in the library. We handle C++ virtual dispatch similarly.

Handling function pointers. To handle function pointers, the default case of the global dispatching sled is set up to save the current execution state and invoke the JIT component. The JIT component adds a direct branch instruction to the target guarded by an equality check to the global dispatcher sled. After patching, the execution state is restored, and the global dispatcher sled is executed again. This time, the target of the function pointer is contained in the sled, and the correct function is invoked. While handling function pointers in the JIT component is required for correctness, this case rarely occurs in programs.

Limitations. The goal of our Switchpoline JIT implementation is to keep it as simple as possible. Thus, the global dispatching sled contains a chain of `if` statements. Hence, the execution time of the sled grows linearly with the number of cases in the sled. The performance could be optimized to grow logarithmically with the number of functions using binary trees. In our implementation, we do not consider multithreading. Thus, the patching of the global dispatching sled is not thread-safe. This causes no issues in practice since we never need the JIT component during runtime for function pointers in any of the evaluated applications, and the loading of libraries happens sequentially at program startup. Our global dispatching sled also has a hardcoded maximum size. However, all of these limitations are artifacts of our implementation and not limitations of the approach. They do not break any of the programs we test and could be removed with some additional engineering effort. Only the limitation of a maximum branch distance remains. If a function pointer too far from the global dispatching sled is invoked, it is impossible to encode the branch offset into a direct branch instruction. In this case, the program is terminated.

5.5 Limited Direct Branches ($C4$)

By default, Linux maps libraries at high addresses, such as `0x7f0000000000`, while position-independent programs are mapped at lower addresses, such as `0x550000000000`. The distance between libraries and the program is too large for a direct branch in ARMv8, which can jump at most ±128 MB. We thus change musl's dynamic loader to map libraries to the 128 MB memory region before the program. With this change, all loaded code is within the range of direct branches. Using position-independent executables ensures the Linux loader maps the program to a sufficiently large address (not `0x400000`) to gain memory space for a program's libraries. PIC in libraries ensures they can be shared, although they might be mapped to different virtual addresses per program. The libraries thereby also "inherit" the ASLR'ed addresses of the main program.

What remains is code loaded by the kernel, namely the dynamic loader itself and the libc, which is the same file for musl. To this end, we build a new, small dynamic loader without dependency on the libc, which initiates the program loading process. This small loader maps musl libc to an address near the program image. Then, it transfers control to musl's dynamic loader. Musl's loader initializes the program and loads its dependencies as usual. This control transfer is out of range for simple direct branches. We solve this case with a signal handler that modifies the application's context. Consequently, the kernel sets the program counter to the desired value. The overhead of this signal is negligible because it happens only once during program startup. Our additional dynamic loader is free of indirect branches. Each generated program contains a reference to the new dynamic loader. Linux can thus invoke the correct loader for protected and unprotected programs.

5.6 Implementation Details

Our compilation toolchain is based on Clang/LLVM 10 [12, 13]. We use link-time optimization (LTO) in all components so that source code is compiled to LLVM's intermediate representation (IR). We compile against an environment containing musl libc [38], LLVM's compiler-rt [34], and LLVM's libc++ [35], including its support libraries libc++abi and libunwind. All libraries are compiled with the Switchpoline compiler and support static and dynamic linking to protected applications. We choose these libraries because they are compatible with Clang (which, for example, glibc is not). LTO reduces the additional binary size of the final application: uncalled functions from the standard libraries can be detected and removed during linking. We include our JIT compiler into the musl libc. We add the runtime PLT rewriter to the compiled code if that code generates a PLT section.

We do not change the compilation of C files to IR. When compiling C++ files, we store additional language information from Clang in the generated object file as metadata. We run our transformations at link time within LLVM's linker `lld`. In this step, we see all application code in IR form and fully know all C++ classes, including any statically-linked libraries. We pull the necessary information about address-taken functions and their parameters from IR; the type-matching algorithm runs on LLVM IR types. We generate our switches as LLVM's `switch` instructions. The subsequent LLVM passes optimize them, and LLVM's backend converts them to assembly. In particular, LLVM can remove switch instructions if only one target is possible, and LLVM can inline targets of rewritten indirect calls. Also, LLVM might use additional information: After transformation, all possible callers of a function are known, and optimizations such as argument promotion or dead code elimination can be more efficient.

Another optimization to reduce the application size is to move the generated switch into a new dispatcher function whenever the same target set is used in multiple calls. We set the threshold at least five calls. Instead of generating the same switch over and over, we call this dispatcher function. When a program contains many indirect calls and switches get larger, this optimization saves space without any significant performance drop.

6 EVALUATION

This section evaluates Switchpoline. In Section 6.1, we evaluate the performance of Switchpoline using microbenchmarks, SPEC CPU 2017 as a macro benchmark, and `lighttpd` as a real-world application. In Section 6.2, we evaluate the security of Switchpoline, showing that Switchpoline-compiled applications do not contain indirect control-flow transfers, preventing Spectre-BTB and Spectre-BHB exploitation.

6.1 Performance Evaluation

This section evaluates the performance of Switchpoline using microbenchmarks (Section 6.1.1 to Section 6.1.3) as well as macrobenchmarks (Section 6.1.4) and `lighttpd` as real-world application (Section 6.1.5). The benchmarks show that the performance overhead of Switchpoline is similar to state-of-the-art x86 software mitigations. All benchmarks run on an Apple Mac Studio with an Apple M1 Max chip and 32 GB of memory. We use Asahi Linux with kernel 5.19. Benchmarks are pinned to a reserved P-core to reduce noise.

6.1.1 Micro Benchmark: Qualitative Analysis. As a micro benchmark, we evaluate the pure performance overhead of Switchpoline. We measure the time for indirect control-flow transfers with a statically known number of targets from 1 to 1000. In our benchmark, a single indirect branch calls all targets, each target at least 500 times. We repeat this 100 times and report the median. The branch can be a C-style function pointer call or a C++ virtual method call. We

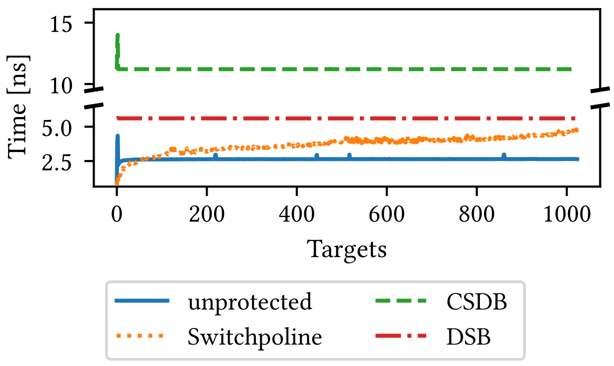

Figure 2: Overhead of Switchpoline-protected indirect calls compared to unprotected branches and branches protected with CSDB/DSB barriers. Switchpoline is fastest up to 54 targets.

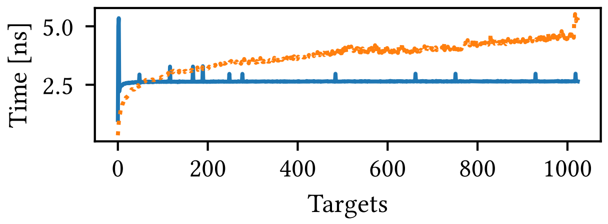

Figure 3: Overhead of Switchpoline-protected virtual dispatch. Switchpoline improves the performance up to 57 possible targets.

also measure the time it takes to call a function pointer that needs to be added by the JIT component.

Figure 2 compares the runtime of Switchpoline-protected calls to standard indirect branches depending on the number of possible branch targets. For unprotected indirect branches, the number of possible targets is irrelevant. We measure a mostly flat line. With Switchpoline, the execution time of the transformed branch grows roughly logarithmic with the number of possible targets. In this benchmark, Switchpoline outperforms unprotected indirect calls if there are fewer than 54 possible targets. Even for branches with as many as 450 possible targets, the overhead is below 50 %. We observe a similar pattern for C++ virtual method calls (Figure 3). Here, Switchpoline outperforms unprotected indirect calls for fewer than 57 targets. Figure 2 also evaluates a `jump/fence` approach with different memory barriers. While unclear whether this fully mitigates Spectre-BTB, it also comes with a much higher overhead than Switchpoline of at least 110 %.

6.1.2 Micro Benchmark: Quantitative Analysis. For a quantitative micro benchmark, we analyze the 16 SPEC applications for the number of targets of indirect branches. In total, these 16 applications contain 24 396 branches that Switchpoline converts. Only 7 out of 16 benchmarks have branches with more than 100 targets: *perlbench* (max 389), *gcc* (1353), *parest* (138), *povray* (159), *omnetpp* (402),

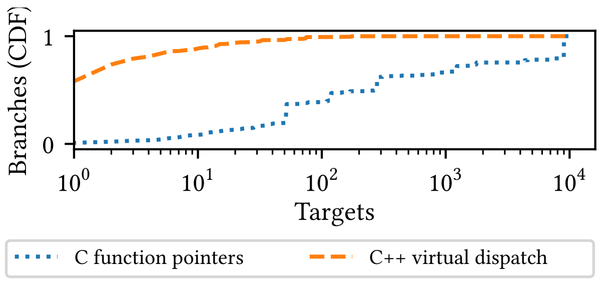

Figure 4: Cumulative distribution function of the number of branch targets for indirect calls and virtual dispatch in all analyzed SPEC CPU 2017 applications.

xalancbmk (155), and *blender* (10 164). Figure 4 shows a cumulative distribution function plot of the number of branch targets. Most C-style function pointer calls (61 %) have fewer than 280 possible targets. C++ virtual dispatch has fewer targets than function pointer calls. 87 % of C++ virtual dispatches have fewer than 10 targets.

Comparing the target-set distribution with the results from Section 6.1.1, we see that 35 % of all C calls have at most 54 targets and, thus, should outperform indirect calls. For C++, 97 % of all virtual calls have at most 57 targets and should outperform the reference.

6.1.3 Micro Benchmark: Dynamic Overhead.

Dynamic Linking. Switchpoline with dynamic linking introduces additional computation at program startup due to the dynamic loader and the PLT rewriter. When measuring the time between program invocation and the start of the main function, we measured zero difference: It takes 24.7 µs to start a minimal program before and after protection with Switchpoline. Rewriting the PLT is fast: Rewriting the PLT for 1000 imported functions on startup takes 3.7 µs. Using the rewritten PLTs does not have overhead—in fact, the rewritten PLT entries should be faster because they can omit the memory access to the GOT.

JIT Patching. On the M1 Mac Studio, the JIT component needs 5 µs to emit code for an unknown target. Subsequent calls to the same target do not invoke the JIT component. Thus, the overhead is only present the first time an unknown target is called.

6.1.4 Macro Benchmark SPEC CPU 2017. For SPEC CPU 2017, we run all C and C++ programs and omit Fortran programs. Benchmarks are repeated 5 times and show an average standard deviation of 0.1 %. We benchmark Switchpoline with static and dynamic linking. The baseline is an unmodified Clang 10 in the same setup, including musl libc and libc++. Compiler flags are `-O3` and LTO. The SPEC CPU benchmark also has the side effect that the functional correctness of Switchpoline is tested.

Functional Correctness. Two of the 16 benchmarks have non-trivial incompatibilities with musl libc and libc++. First, the benchmark `gcc` crashes when built with the unmodified reference compiler; we omit it from further evaluation. Second, the benchmark `parest` is unstable with static linking and the reference compiler; we use it with dynamic linking only. All remaining benchmarks complete successfully with Switchpoline enabled. Switchpoline does not impair the functional correctness of any tested program.

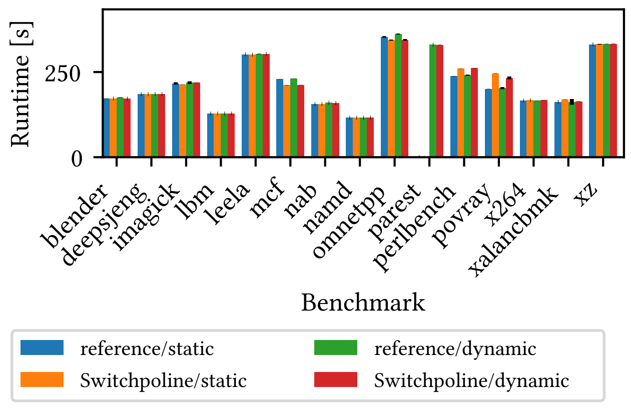

Figure 5: Runtime of SPEC CPU 2017 benchmarks. From left to right: Reference with static linking, Switchpoline with static linking, reference with dynamic linking, and Switchpoline with dynamic linking.

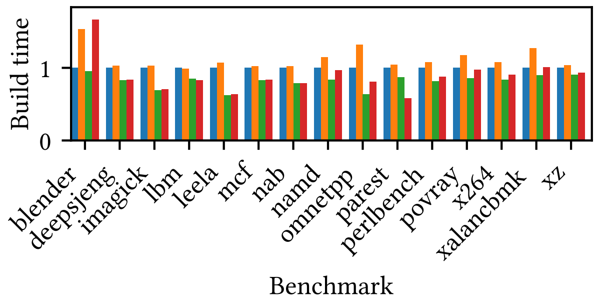

Figure 6: SPEC CPU 2017 build time relative to reference with static linking. Same ordering as in Figure 5.

Runtime. The runtime of SPEC CPU 2017 benchmarks is graphed in Figure 5. With static linking, we measure a maximum overhead of 26 % for Switchpoline in the `povray` benchmark compared to the reference compiler. With dynamic linking, the same benchmark records the highest overhead of 19 %. In some cases, Switchpoline improves performance with both static and dynamic linking. `mcf` runs 8.7 % faster with Switchpoline. On average, we measure an overhead of 1.8 % for Switchpoline with static and dynamic linking. In summary, Switchpoline adds an overhead of −8.7 %–26 %.

Build Time. In most cases, Switchpoline does not introduce a noticeable slowdown in the build process (Figure 6). Except for the following three outliers, the compilation time increased by less than 10 s: *omnetpp* (38 s ⟶ 50 s), *xalancbmkr* (56 s ⟶ 71 s), and *blender* (455 s ⟶ 700 s). Excluding those outliers, the compilation time increased by 1.6 s on average.

Binary Size. Figure 7 shows program sizes before and after Switchpoline protection. We measure differences of −3 % to 33 %. We record the biggest size increase for *lbm* with static linking (33 %). On the other hand, Switchpoline with dynamic linking decreases the size of *leela* by 3 %. On average, SPEC 2017 benchmarks are 6 % larger with dynamic linking and 14 % larger with static linking.

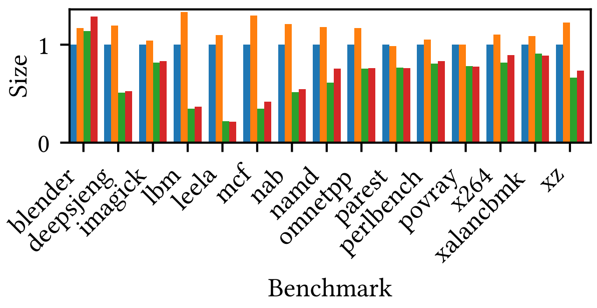

Figure 7: Size of SPEC CPU 2017 binaries relative to reference with static linking. Same ordering as in Figure 5.

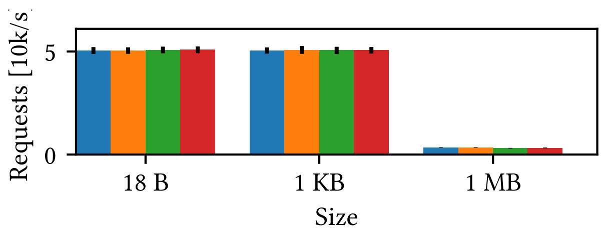

Figure 8: Throughput of `lighttpd` for different file sizes. Same ordering as in Figure 5.

6.1.5 Real-World Application: Lighttpd. We choose `lighttpd` (version 1.4.59) as a real-world application which we benchmark using the Apache HTTP server benchmarking tool (`ab`). Like the SPEC CPU 2017 benchmarks, we compile `lighttpd` with Switchpoline and an unmodified compiler as reference, once with static and once with dynamic linking. We configure `lighttpd` to serve three files of sizes: 18 B, 1 kB, and 1 MB. We use `ab` with 5 concurrent connections to send 100 000 requests. These measurements are repeated 200 times. All measurements for requests per second and request latency are within error of the reference (Figure 8), i.e., we do not see any relevant overhead for Switchpoline.

6.2 Security Evaluation

The security of Switchpoline is based on eliminating *all* indirect branches. This section evaluates if this property holds and whether attacks are prevented. Moreover, we argue that Switchpoline does not undermine other mitigations.

Compatibility with Orthogonal Mitigations. While Switchpoline only targets Spectre-BTB and Spectre-BHB, it is fully compatible with other mitigations, targeting both Spectre attacks [17] and traditional memory-corruption attacks [46]. Intuitively, Switchpoline converts Spectre-BTB gadgets that can be mistrained to speculatively jump to code at any location into Spectre-PHT gadgets that can only be mistrained to speculatively call any function in the computed target set, significantly reducing the number of available gadgets. While reducing call targets already improves security, the remaining Spectre-PHT gadgets can be mitigated automatically using orthogonal mitigations such as Speculative load hardening (`-mspeculative-load-hardening` or selectively for individual functions). Similarly, memory barriers could be inserted at the beginning of functions to stop speculative execution on mispredictions. In contrast to Retpoline, Switchpoline is fully compatible with backward-edge CFI mechanisms, such as shadow stacks [16], as it does not modify return addresses. Moreover, Switchpoline can be combined with orthogonal defenses against memory corruption vulnerabilities, e.g., stack canaries, Pointer Authentication Codes (PAC) for data pointers, and ASLR. These mitigations do not affect or need indirect branches. Hence, they do not interfere with Switchpoline. Switchpoline impacts ASLR as libraries are mapped close to the executable's code to ensure code is within ±128 MB (cf. $C4$).

Security Argument. Intuitively, an attacker cannot inject a branch target into indirect branches if an application does not contain indirect branches. To verify that our compiler mitigations remove all indirect branches, we disassemble all SPEC benchmarks, standard libraries, including the dynamic loader, and custom test programs. No program contained indirect branch instructions (`br` or `blr`) anymore. Hence, we conclude that all checked programs are no longer vulnerable to Spectre-BTB. Spectre-BHB attacks are mitigated as they also rely on indirect branch mispredictions.

7 DISCUSSION & LIMITATIONS

Code Size. While we can reduce the distance between code of libraries and the executable by fixing the memory layout as described in Section 5.5, a limit of 128 MB for the total size of the code segment remains. To evaluate the impact of this limitation, we analyze the first 1000 most frequently installed Debian packages available for ARMv8 according to the Debian popularity contest [3]. Within those, we find 5178 unique binaries. To give an upper bound on their code segment size if Switchpoline were applied, we recursively resolved all library dependencies and calculated the sum of all code segment sizes. We conclude that only 133 (2.57 %) of the analyzed binaries exceeded the limit of 128 MB, 57 (42.86 %) of which are from the `kdepim-addons` package. Most binaries exceeding the limitation are GUI applications (e.g., `krita`, `obs-studio`, or `shotwell`). With static linking, unreachable code can likely be eliminated, and even more binaries fit the limitation. Thus, Switchpoline can be applied to most applications without nearly hitting this limitation. While this limit affects our implementation of Switchpoline, it is not a fundamental limitation of the idea behind Switchpoline: Proxy direct branch instructions can be added between the dispatcher and branch target if the distance is too large.

Linking with Unprotected Libraries. Switchpoline can link an application with unprotected dynamic libraries. However, while linking with unprotected libraries is possible, such libraries likely contain indirect branches. Such indirect branches in unprotected libraries re-enable Spectre-BTB attacks. Hence, the security guarantees of Switchpoline are weakened when linking with unprotected libraries. Moreover, there are unhandled corner cases when linking with unprotected libraries. While function pointers from the unprotected library can be called in the protected library due to the JIT, passing function pointers to an unprotected library is not currently supported. Function identifier numbers replace function pointers in protected code. These identifiers are not valid addresses in the unprotected library. Thus, unprotected code is not able to call these identifiers and crashes. A more advanced implementation could use the JIT to translate function identifiers to function pointers before passing them to an unprotected library. Similarly, passing C++ objects with virtual methods between unprotected and protected code may crash for the same reason.

JIT Compilation. Switchpoline is a compile-time mitigation, so it cannot protect indirect branches that do not exist at compile-time. Hence, if an application creates indirect branches at runtime, either by just-in-time compiling code or by dynamically loading unprotected code, Switchpoline is circumvented. However, it is fair to shift the responsibility for protecting the runtime-generated code to the developer in such a case. As shown in this paper, Switchpoline can be implemented purely in the compiler. Hence, JIT compilers, as used in, e.g., browsers, could also implement Switchpoline to protect the emitted code against Spectre-BTB and Spectre-BHB attacks. For this paper, we do not implement Switchpoline in a browser, as this is an enormous engineering task, which requires understanding the JIT compiler and re-implementing everything for this compiler architecture. Given that our proof-of-code implementation already required around 5000 lines of code, even though it could reuse many parts of previous work [13], we consider such an implementation out of scope for demonstrating the viability of Switchpoline.

Other uses of the BTB. Switchpoline mitigates Spectre-BTB on indirect branches. Cases where other instructions use the BTB as a predictor are not handled in our current PoC implementation. For example, our implementation does not eliminate returns to harden against possible Spectre-BTB-like variants that target the return stack buffer, as it is unclear whether they form a possible attack vector on Arm devices. However, this is not a limitation of our approach. It is principally possible to rewrite returns as indirect branches [27] and eliminate them using Switchpoline.

Applicability to other Architectures. This paper focuses on ARMv8, as it lacks generic software-based mitigations against Spectre-BTB and Spectre-BHB. However, our approach is not limited to this architecture. Other Arm architectures, e.g., ARMv9 and ARMv7, are compatible with Switchpoline but might require minor changes for the JIT. On ARMv7, the direct branch offset is further limited to the range ±32 MB. Switchpoline can generally also be used on x86. With recent discoveries that x86 software defenses might be incomplete [51], Switchpoline can be a viable alternative.

8 CONCLUSION

Many Arm CPUs in devices the worldwide are affected by Spectre-BTB and Spectre-BHB. With no simple compiler-based mitigation for userspace applications, they can only be secured individually and with high effort. We change this by introducing Switchpoline, the first automated Spectre-BTB software workaround protecting userspace applications on ARMv8. Switchpoline rewrites indirect control-flow transfers into direct control-flow transfers, fully mitigating any Spectre-BTB and Spectre-BHB vulnerability in C or C++ code with a simple recompilation. We show that the runtime overhead of Switchpoline-protected applications is negligible, with an average of 1.8 % measured with SPEC CPU 2017. By providing a Spectre-BTB PoC that leaks up to 20 kB/s and works on all tested ARMv8 devices, we stress that Switchpoline should be widely adopted to prevent exploitation of Spectre-BTB on ARMv8 devices.

REFERENCES

[1] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. 2005. Control-Flow Integrity. In *CCS*.

[2] Advanced Micro Devices Inc. 2018. Software Techniques for Managing Speculation on AMD Processors. Revison 7.10.18.

[3] Bill Allombert. 2022. Debian Popularity Contest. https://popcon.debian.org/stable/by_vote

[4] Nadav Amit, Fred Jacobs, and Michael Wei. 2019. Jumpswitches: restoring the performance of indirect branches in the era of spectre. In *USENIX ATC*.

[5] ARM. 2018. Cache Speculation Side-channels. https://armkeil.blob.core.windows.net/developer/Files/pdf/Cache_Speculation_Side-channels_03May18.pdf

[6] ARM. 2020. Cache Speculation Side-channels. Version 2.5.

[7] Arm. 2020. Spectre-BHB: Speculative Target Reuse Attacks. https://developer.arm.com/documentation/102898/latest/ Version 1.7.

[8] ARM. 2020. Straight-line Speculation. Version 1.0.

[9] Arm. 2022. Speculative Processor Vulnerability. https://developer.arm.com/Arm%20Security%20Center/Speculative%20Processor%20Vulnerability

[10] ARM. 2023. Arm Architecture Reference Manual for A-profile architecture.

[11] Enrico Barberis, Pietro Frigo, Marius Muench, Herbert Bos, and Cristiano Giuffrida. 2022. Branch history injection: On the effectiveness of hardware mitigations against cross-privilege Spectre-v2 attacks. In *USENIX Security Symposium*.

[12] Markus Bauer, Ilya Grishchenko, and Christian Rossow. 2022. TyPro: Forward CFI for C-Style Indirect Function Calls Using Type Propagation. In *ACSAC*.

[13] Markus Bauer and Christian Rossow. 2021. NoVT: Eliminating C++ Virtual Calls to Mitigate Vtable Hijacking. In *IEEE EuroS&P*.

[14] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. 2019. SMoTherSpectre: exploiting speculative execution through port contention. In *CCS*.

[15] Rodrigo Branco, Kekai Hu, Ke Sun, and Henrique Kawakami. 2019. Efficient mitigation of side-channel based attacks against speculative execution processing architectures. US Patent App. 16/023,564.

[16] Nathan Burow, Xinping Zhang, and Mathias Payer. 2019. SoK: Shining light on shadow stacks. In *S&P*.

[17] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *USENIX Security Symposium*. Extended classification tree and PoCs at https://transient.fail/..

[18] Chandler Carruth. 2018. https://reviews.llvm.org/D41723

[19] Chandler Carruth. 2018. Speculative Load Hardening. https://docs.google.com/document/d/1wwcfv3UV9ZnZVcGiGuoITT_61e_Ko3TmoCS3uXLcJR0/edit#heading=h.phdehs44eom6

[20] CodeSourcery, Compaq, EDG, HP, IBM, Intel, Red Hat, and SGI. 2021. Itanium C++ ABI. https://itanium-cxx-abi.github.io/cxx-abi/abi.html

[21] Google. 2019. A year with Spectre: a V8 perspective. https://v8.dev/blog/spectre

[22] Lorenz Hetterich and Michael Schwarz. 2022. Branch Different - Spectre Attacks on Apple Silicon. In *DIMVA*.

[23] Intel. 2018. Indirect Branch Restricted Speculation. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/indirect-branch-restricted-speculation.html

[24] Intel. 2018. Retpoline: A Branch Target Injection Mitigation. Revision 003.

[25] Intel. 2018. Speculative Execution Side Channel Mitigations. Revision 3.0.

[26] Intel. 2019. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide.

[27] Intel. 2020. An Optimized Mitigation Approach for Load Value Injection. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/optimized-mitigation-approach-load-value-injection.html

[28] Intel Corporation. 2020. Single Thread Indirect Branch Predictors. https://software.intel.com/security-software-guidance/deep-dives/deep-dive-single-thread-indirect-branch-predictors

[29] ISO. 2011. *ISO/IEC 9899:2011 Information technology — Programming languages — C*. International Organization for Standardization.

[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. 2019. Spectre Attacks: Exploiting Speculative Execution. In *S&P*.

[31] Moritz Lipp, Daniel Gruss, and Michael Schwarz. 2022. AMD Prefetch Attacks through Power and Time. In *USENIX Security*.

[32] Moritz Lipp, Vedad Hadžić, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. 2020. Take a Way: Exploring the Security Implications of AMD's Cache Way Predictors. In *AsiaCCS*.

[33] 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. 2018. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium*.

[34] LLVM Project. 2022. "compiler-rt" runtime libraries. https://compiler-rt.llvm.org/

[35] LLVM Project. 2022. "libc++" C++ Standard Library. https://libcxx.llvm.org/

[36] G. Maisuradze and C. Rossow. 2018. ret2spec: Speculative Execution Using Return Stack Buffers. In *CCS*.

[37] Alyssa Milburn, Ke Sun, and Henrique Kawakami. 2022. You cannot always win the race: Analyzing the lfence/jmp mitigation for branch target injection. *arXiv:2203.04277* (2022).

[38] musl authors. 2022. musl libc. https://musl.libc.org/

[39] Charles Reis, Alexander Moshchuk, and Nasko Oskov. 2019. Site Isolation: Process Separation for Web Sites within the Browser. In *USENIX Security Symposium*.

[40] Michael Schwarz, Moritz Lipp, Claudio Canella, Robert Schilling, Florian Kargl, and Daniel Gruss. 2020. ConTExT: A Generic Approach for Mitigating Spectre. In *NDSS*.

[41] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. 2019. NetSpectre: Read Arbitrary Memory over Network. In *ESORICS*.

[42] Martin Schwarzl, Pietro Borrello, Andreas Kogler, Kenton Varda, Thomas Schuster, Daniel Gruss, and Michael Schwarz. 2022. Robust and Scalable Process Isolation against Spectre in the Cloud. In *ESORICS*.

[43] Martin Schwarzl, Claudio Canella, Daniel Gruss, and Michael Schwarz. 2021. Specfuscator: Evaluating Branch Removal as a Spectre Mitigation. In *FC*.

[44] Martin Schwarzl, Thomas Schuster, Michael Schwarz, and Daniel Gruss. 2021. Speculative Dereferencing of Registers: Reviving Foreshadow. In *FC*.

[45] Stephen Röttger and Artur Janc. 2021. A Spectre proof-of-concept for a Spectre-proof web. https://security.googleblog.com/2021/03/a-spectre-proof-of-concept-for-spectre.html

[46] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *S&P*.

[47] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *USENIX Security*.

[48] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. 2018. CheckMate: Automated Synthesis of Hardware Exploits and Security Litmus Tests. In *MICRO*.

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

[50] Daniel Weber, Ahmad Ibrahim, Hamed Nemati, Michael Schwarz, and Christian Rossow. 2021. Osiris: Automated Discovery of Microarchitectural Side Channels. In *USENIX Security*.

[51] Johannes Wikner and Kaveh Razavi. 2022. RETBLEED: Arbitrary Speculative Code Execution with Return Instructions. In *USENIX Security Symposium*.

[52] Tao Zhang, Kenneth Koltermann, and Dmitry Evtyushkin. 2020. Exploring Branch Predictors for Constructing Transient Execution Trojans. In *ASPLOS*.

APPENDIX

A AVAILABILITY

The source code of Switchpoline alongside benchmarks is available under github.com/cispa/Switchpoline.

B SPECTRE-BTB POC

```c
void (**attack)(int); // indirect branch target
char *array2;         // shared array
char *secret;         // private array

void secret_fun(int offset) {
    int shift = (offset % OFFSETS_PER_BYTE) * BITS;
    size_t idx = offset / OFFSETS_PER_BYTE;
    memory_access(&array2[
        ((secret[idx] >> shift) & (VALUES - 1)) * ENTRY_SIZE
    ]);
}

void dummy_fun(int offset){}

void victim(void(**func)(int), int offset) {
    (*func)(offset);
}
```

Listing 7: Victim Code of our Spectre-BTB PoC

Listing 7 shows the relevant part of our Spectre-BTB PoC. The vulnerable branch is emitted for the function pointer called in the `victim` function (passed as `func`). While architecturally executing `dummy_fun`, the indirect branch can be mistrained to speculatively execute `secret_fun` instead. The `secret_fun` accesses the `secret` array at the given offset, and encodes the speculatively accessed data into the cache state of array `array2`.