Specfuscator: Evaluating Branch Removal as a Spectre Mitigation

Martin Schwarzl[1], Claudio Canella[1], Daniel Gruss[1], and Michael Schwarz[2]

[1]Graz University of Technology, Austria [2]CISPA Helmholtz Center for Information Security, Germany

Abstract. Attacks exploiting speculative execution, known as Spectre attacks, have gained substantial attention in the scientific community and in industry with a broad range of defense techniques proposed. In particular, in-software defenses for commodity systems attempt to leave the program structure as is, but defuse every potential Spectre gadget by, e.g., stopping the speculation, or limiting value ranges. While these mitigations disrupt the program flow on every conditional branch, they still contain every single conditional branch instruction.
In this paper, we show that one dimension of Spectre mitigations has been overlooked entirely. We explore a novel principled Spectre mitigation that sits at the other end of the scale: the absence of conditional and indirect branches. Our mitigation is based on automatically linearizing the program flow through a special compiler pass, eliminating **all** conditional and indirect branches. We show that our Spectre mitigation has very clear security guarantees. We explore the feasibility of this unorthodox approach and evaluate its performance in comparison to the more conservative approaches presented so far. We observe that the performance overhead can be low, e.g., 5 %, for certain use cases, being on-par with state-of-the-art mitigations, but very high for other use cases, e.g., and overhead factor of 1000. Our results demonstrate the feasibility of Spectre defenses that eliminate branches and indicate good performance-security trade-offs for Spectre defenses can be achieved by sticking to neither of the extremes.

1 Introduction

Speculative execution is a significant factor in the performance of modern processors. Instead of waiting for a branch decision or branch target to be architecturally determined, the processor takes an educated guess based on behavior observed in the past. From a pipeline perspective, this linearizes the execution of instructions as the branch decision is omitted in the speculative execution flow and only subsequently validated. Spectre attacks [31] induce incorrect speculative execution flows into a victim context by manipulating the branch predictors. During this speculative execution, the attacker can make the victim access secrets and encode them into the microarchitectural state. Using a side-channel attack, e.g., Flush+Reload [54], the secrets can then be recovered.

Previous countermeasures [36,12,11] either attempt to thwart successful covert-channel transmission during speculation [28,27,53], abort the speculative execution before secrets can be accessed [22,1,3,41,23,13,39,50], or ensure that secrets cannot be accessed during speculative execution [43,44,56]. Amit et al. [4] tried to increase the performance of indirect branches by rewriting them into two direct branches. However, from the perspective of branches in a program, all these countermeasures remain in the same range of the scale, namely all conditional and indirect branches remain in the program, in some cases even with additional branches added. This raises an important scientific question:

Can the (substantial) reduction of branches, in particular the elimination of all vulnerable branches, be a viable Spectre mitigations? Can such Spectre mitigations maintain a reasonable overhead in certain use cases?

In this paper, we answer both questions in the affirmative. To answer these scientific questions, we explore a novel Spectre mitigation at the other end of the scale: the elimination of all conditional and indirect branches. While this may sound impractical at first, it has been used for years to implement cryptographic algorithms in constant time [7]. We demonstrate the feasibility of this approach with our new mitigation, Specfuscator. Specfuscator is based on the movfuscator [14] tool that automatically linearizes the program flow through a special compiler pass. In contrast to *M/o/Vfuscator*, we do not replace all operations, but just control-flow manipulating instructions, effectively eliminating **all** conditional branches. To improve the performance of *M/o/Vfuscator*, we bring back ALU operations, the `cmp` instruction and exploit the x86 addressing mode. In comparison to the *M/o/Vfuscator* we increase the runtime up to a factor of 50 and decrease the binary size by 30 % and compile time up to 46 %. We show that our Spectre mitigation is a principled approach with respect to security, following the simple argument that if there are no conditional or indirect branches, no branches can be mispredicted.

For our evaluation we analyzed Specfuscator in comparison with a set of other compilers: the related *M/o/Vfuscator* and LCC, a patched clang with `lfence` protections on all conditional branches, and an unpatched clang without any Spectre mitigations. We evaluate the performance of our unorthodox approach and discover that the overhead can be as low as 5 %, being on-par with state-of-the-art mitigations, but also much higher, up to factor 1000, performing clearly worse than state-of-the-art mitigations. Thus, for some use cases, the elimination of conditional and indirect branches is nearly as efficient as state-of-the-art mitigations but with a stronger security argument. This indicates that the space between the two extremes, all conditional and indirect branches and no conditional and indirect branches, should receive more attention for the design of future countermeasures.

Our key contributions are:

- We explore a previously unexplored mitigation space against Spectre: the absence of conditional and indirect branches.
- We present a solution based on a linearized control-flow with very clear and strong security guarantees.

- We evaluate our approach and observe that the performance overhead can be lower than state-of-the-art mitigations in some use cases, but also significantly higher in others.
- Our results shed light on a new direction for performance-security trade-offs for Spectre defenses.

The remainder of this paper is organized as follows. In Section 2, we provide background information. In Section 3, we discuss the landscape of existing Spectre defenses and point out blank spots. In Section 4, we present Specfuscator, our Spectre defense mechanism. In Section 5, we evaluate the performance and security of Specfuscator. In Section 6, we discuss the context and implications of our work. We conclude in Section 7.

2 Background

This section provides some background information about speculative execution attacks and the internals of the *M/o/Vfuscator*.

2.1 Speculative Execution Attacks

Modern CPUs extensively use out-of-order execution and prediction mechanisms to increase performance. Speculative execution uses branch predictions to advance the control flow speculatively. Branch prediction mechanisms are implemented via different structures, such as the Branch History Buffer (BHB) [8,31], the Branch Target Buffer (BTB) [33,16,31], the Pattern History Table (PHT) [18,31], and the Return Stack Buffer (RSB) [18,34,32].

Mispredicted branches are reverted on the architectural level, but not on the microarchitectural level [31]. Hence, code that should not have been executed architecturally still leaves microarchitectural traces, e.g., in various caches. By leveraging traditional side-channel attacks, these microarchitectural traces can be brought into the architectural domain, potentially recovering data that was not supposed to be accessed, *i.e.*, secrets.

Kocher et al. [31] first discussed transient-execution attacks [12] using speculative execution and demonstrated that conditional branches and indirect jumps can be exploited to leak data. Subsequent work has then shown that the idea can be extended to function returns [34,32] and store-to-load forwarding [21]. Canella et al. [12] then systematically analyzed the field and demonstrated that the necessary mistraining can be done in the same and a different address space due to some predictors being shared across hyperthreads. Additionally, they also showed that many of the proposed countermeasures are ineffective and do not target the root cause of the problem. While the cache has been predominantly exploited for the transmission of the secret data [31,12,34,32], other channels have also been shown to be effective, *i.e.*, execution port contention [9].

To mitigate all these attacks, various proposals have been made by industry and academia. Canella et al. [11] analyzed the differences between countermeasures proposed by academia and by industry, highlighting that academia pro-

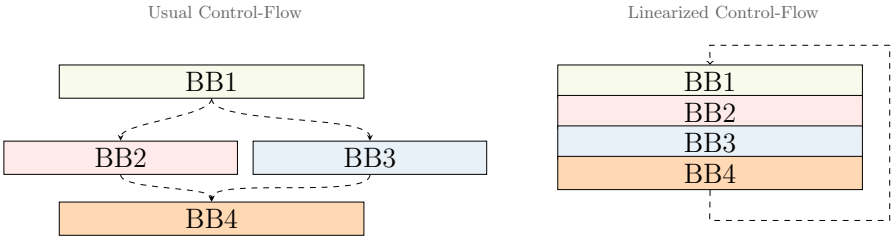

Fig. 1: Branch instructions typically split up the control flow. Constant-time cryptographic algorithms avoid branches (left) and instead linearize the control flow (right), e.g., square-and-always-multiply [15], turning the security-critical branches and basic blocks into one large basic block. *M/o/Vfuscator* follows the same idea of linearizing the control-flow and uses one main execution loop, turning the program into one large basic block.

poses more radical countermeasures compared to industry. In general, the proposed mitigations either require significant changes to the hardware [28,53,27], require a developer to annotate secrets [44,40,20], introduce data dependencies [39,13], or reduce the accuracy of timers [35,42,49,51].

2.2 *M/o/Vfuscator*

Turing completeness is a part of computability theory that describes a set of rules or instructions that can be simulated on a single-taped Turing machine. Dolan [48] showed that the x86 `mov` instruction is Turing-complete. Based on this observation, Domas [14] invented the single-instruction compiler *M/o/Vfuscator*. The *M/o/Vfuscator* patches the Little C compiler (LCC) to use an emitter that only emits `mov` instructions. *M/o/Vfuscator* is an x86 32-bit compiler and also only supports 32-bit arithmetic operations.

The compiled program runs in a virtual machine, which basically runs like a Turing machine. The entire program is branch-free and thus executed as a single basic block, leading to a linearized control flow graph. Figure 1 illustrates the linearized control flow graph. Thus, the program is always executed from start to end in a loop. To ensure the correctness of the program, a flag specifies whether an instruction should compute on the target location or a dummy *discard* location. All instructions that are not relevant for a specific iteration are discarded using this discard location. Hence, although the instruction is executed, it has no impact on the current behavior of the program. This technique is the same that is used to ensure constant-time implementations of, e.g., cryptographic algorithms [15].

Note that this is similar to constant-time cryptographic algorithms, e.g., square-and-always-multiply [15], the program executes both branches and, thus, always runs the algorithm from start to end in a loop.

Arithmetic operations, *i.e.*, additions, multiplications, divisions, bitwise-operations, are implemented using two-dimensional lookup tables. To save mem-

ory. 32-bit operations are split into two 16-bit operations, and, thus, only 16-bit lookup tables are required. By exploiting the addressing modes of x86-`mov`, the first `mov` looks up the row for the first operand, the second `mov` looks up the corresponding column for the second operand, and the value is reported as result.

M/o/Vfuscator handles internal jumps to specific parts of the code using a target register. *M/o/Vfuscator* installs two signal handlers for `SIGSEGV` and `SIGILL` to enable branching [14]. At the end of the program, an illegal instruction is emitted to trigger the `SIGILL` handler and jump back to the start of the program. To perform external library calls, *i.e.*, calling libc functions such as `printf`, segmentation faults are used [14]. To adhere to the x86 calling convention, the function's arguments are pushed onto the stack.

As the name indicates, *M/o/Vfuscator* can also be used as an obfuscation technique. However, as Kirsch et al. [29] demonstrated, it is possible to deobfuscate this technique with taint analysis.

3 Blank Spots in the Spectre Defense Landscape

Most Spectre countermeasures attempt to break different phases of Spectre attacks [12,11]. These phases are described in previous work as *preparation*, *misspeculation*, *access*, *encoding*, *leakage*, and *decoding*.

Preparation. Preventing the preparation phase can often be seen as equivalent to disabling performance optimizations in the CPU. By disabling either microarchitectural states or speculation at all, an attacker is unable to prepare a Spectre attack. While disabling speculation has been suggested as a mitigation [31], modern CPUs do not support disabling speculative execution. Moreover, it can be expected that disabling speculative execution results in a considerable slowdown. Similarly, disabling the cache also has an unacceptable performance overhead as every memory access has to be served from memory. Additionally, other microarchitectural elements could be used as side channel in the absence of the cache [12,9,45].

Misspeculation and Preventing Access. Most focus so far was on the main cause of Spectre attacks, the misspeculation phase, or the transient access of secrets following the misspeculation. Intel, AMD, and ARM [3,24,5] prevent Spectre-BTB and Spectre-RSB by restricting how an attacker can influence the predictors. For Spectre-PHT, serializing instructions are recommended to stop speculation at security-critical branches [24]. However, this means that branches have to be identified and separately patched.

Furthermore, it could be that memory barrier instructions are not fully serializing [2]. To entirely protect an application, speculation barriers are required for each branch that could be followed by cache fetches. Adding memory barriers for each conditional branch can lead to runtime overheads of up to 440 % [39]. Additional to that performance overhead, Schwarz et al. [45] have shown that speculation barriers for each branch do not suffice as other channels can be used

to leak data, such as the AVX unit or the TLB, as these barriers do not prevent interaction with these microarchitectural elements.

Oleksenko et al. [39] introduced data dependencies to branch conditions and the following instructions to force a stall if the branch cannot be decided yet. Similarly, Carruth [13] proposed to use branchless code to check loads, ensuring that the load is executed along a valid control-flow path. One pre-requisite for this approach is that the hardware supports branchless and unpredicted conditional updates of register values.

Schwarz et al. [44] and Fustos et al. [20] propose to annotate secrets and propagate these annotations to the CPU to ensure that secrets are inaccessible during transient execution. Speculative taint tracking (STT) [56] uses light-weight taint tracking to taint not yet committed data and delay instructions that use it. Similarly, NDA [52] prevents the execution of potentially leaking instructions if they depend on a not yet retired operation.

All of these mitigations keep the number of branches identical but ensure that no leakage occurs by breaking the link between the *misspeculation* phase and the subsequent *access* or *encoding* phases.

Other solutions attempt to add branches that are potentially less easy to exploit [4]. Google proposed *retpoline* [50], a code sequence replacing indirect branches with return instructions, to prevent Spectre-BTB. While *retpoline* also adds more jumps to the program, these are direct jumps and, thus, likely unexploitable. Hence, the total number of branches increases, although potentially fewer are exploitable. Branco [10] proposed a probabilistic alternative to *retpoline*, called *randpoline*, which is compatible with Intel Control-flow Enforcement Technology (CET). This alternative introduces a large number of indirect branches and randomly chooses one of them, reducing the chance that an attacker can mistrain the actually executed branch.

Encoding, Leakage, and Decoding. In these phases, the secret was already accessed transiently. Preventing exploitation in these phases would require ensuring that no covert channel exists between the transient and the architectural domain. However, the way modern CPUs work, it is unrealistic to assume that covert channels can be entirely prevented. While proposals exist to limit the resolution of timers [51] or to build microarchitectural shadow structures [27,53] to squash the results on mispredictions and leave no microarchitectural traces in the cache. However, these mitigations are typically incomplete [12].

Classification. While these defenses have different security properties, depending on the phase they target, they have in common that specific branches are either transformed into other branches, or that the flow from mispredicted branch to leakage is interrupted. We classify the existing Spectre defenses, as illustrated in Figure 2. From this figure, it becomes apparent that most solutions sit in the same range of keeping the number of branches identical, and some defenses increase the number of branches.

Existing software-based countermeasures try to surgically modify conditional branches or subsequent data access to prevent the exploitation of misspeculation.

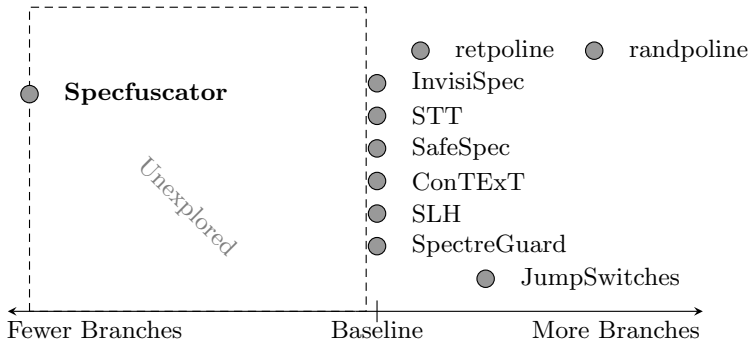

Fig. 2: Previous Spectre defenses were either not changing the number of conditional branches, but possibly adding more (direct) branches to a program. The space of eliminating branches is largely unexplored.

However, as an alternative to preventing speculative execution of conditional branches entirely [31], another possibility is to *eliminate* conditional branches. In this work, we analyze this largely unexplored mitigation technique of removing conditional branches, thus also eliminating the root cause of Spectre attacks.

4 Specfuscator

In this section, we introduce the design of Specfuscator in the first part. Then we discuss the security guarantees of Specfuscator and outline the implementation.

4.1 Design of Specfuscator

Specfuscator is based on the work by Dolan [48] showing that the x86 `mov` instruction is Turing complete. Hence, it is always possible to transform a regular application into an application that consists only of `mov` instructions, and thus *no conditional branches*. This approach has been implemented by Domas [14] as *M/o/Vfuscator* with the goal of obfuscating applications and making them difficult to reverse engineer.

The main idea is always to execute both code paths of every conditional branch, similar to the constant-time square-and-always-multiply algorithm for RSA [15]. Per conditional branch, a flag decides whether the calculated results are kept and committed to the program state, or discarded by specifying a dummy location as the target. Such an approach is also considered secure for implementing side-channel resilient cryptographic algorithms [15,38,55]. The advantage of this approach is that it can be fully automated in the compiler.

M/o/Vfuscator leverages the code generation of the LCC compiler but replaces the emitter for single instructions by a special emitter, generating the corresponding assembly code. *M/o/Vfuscator* labels all branches and uses a software-emulated target register to decide which of the branches is currently

executed. If the execution flag is set, all operations are performed as specified in the program code. Conversely, if the flag is not set, the results of the operations are discarded, similar as square-and-always-multiply [15].

Branching is emulated using branch-free comparison using subtraction and logical operations. Depending on the result of the comparison, the corresponding flags (zero flag, signed flag, carry flag, and overflow flag) are set, and the target location is selected. A flag specific to this approach is the execution flag that can be changed by compare instructions. After disabling the execution flag, the results of the subsequent instructions are stored to a scratch location. If the instruction pointer (`EIP`) reaches the target basic block, the execution is enabled again, and the results are again made architectural.

Similar to the square-and-always-multiply loop [15], the code is always executed in its entirety in a loop. Hence, the execution speed suffers while secret-dependent operations, secret-dependent branches, and secret-leaking misspeculation are eliminated. This design leads to a linearization of the program flow. Therefore, the CPU does not need to predict the outcome of branch instructions. If there are no branches in the program, there can be no mispredictions and resulting pipeline stalls [26].

While the `mov`-based approach is already secure against Spectre attacks, it introduces a considerable performance overhead. Arithmetic operations are implemented via extensive use of two-dimensional arithmetic lookup tables. For instance, a 32-bit addition requires 50 x86 mov instructions, which use 16-bit lookup tables. To increase the performance of Specfuscator, we do not solely rely on the `mov` instruction. As we only aim to prevent Spectre attacks, we do not implement arithmetic operations using `mov`s. Instead, we rely on the native x86 arithmetic instructions, as they cannot be exploited using Spectre. In addition, we exploit the x86 addressing modes to operate directly in memory instead of moving both operands into registers. This optimization saves one additional `mov` instruction per memory operation.

Another instruction that is safe with respect to Specter is the `cmp` instruction. Thus, Specfuscator directly uses the `cmp` instruction instead of a subtraction for comparing two values. The required flag, e.g., the execution flag, is then set via arithmetic and logic instructions. Figure 3 illustrates how Specfuscator emits branch-free code using `mov` instructions.

The only jump instruction in Specfuscator is the jump from the end of the program to the top of the execution loop. In *M/o/Vfuscator*, this was solved using an illegal instruction and a corresponding exception handler. However, this causes a considerable performance overhead and might even lead to misspeculation in the interrupt handler [46]. Hence, as a Spectre attack cannot exploit a direct, unconditional jump, the illegal instruction can be replaced via a direct jump to the top of the execution loop.

4.2 Security of Specfuscator

Specfuscator is a defense against Spectre attacks that exploit control-flow misprediction, *i.e.*, Spectre-PHT [31], Spectre-BTB [31], and Spectre-RSB [32,34],

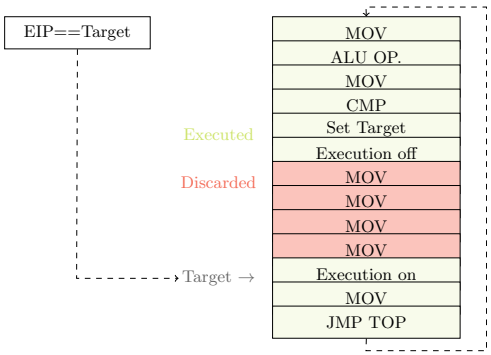

Fig. 3: Branching is handled via a target value for each basic block. If the target is reached, the execution flag is toggled, and the results modify the program's state. Conversely, until the target does not match, the results are written to scratch locations.

as classified by Canella et al. [12]. Straightline Spectre [6] is a special case of Spectre-BTB and Spectre-RSB, where the CPU speculatively skips a branch and continue with the instruction directly after the branch. Another Spectre variant, Spectre-STL [21], is a separate mechanism that relies on incorrect speculations for store-to-load forwarding, *i.e.*, it is a data-flow misprediction.

The idea of Specfuscator is that none of the control-flow mispredicting Spectre variants (Spectre-PHT [31], Spectre-BTB [31], and Spectre-RSB [32,34], including Straightline Spectre [6]), work if the corresponding control-flow modifying instructions are not used at all. Specfuscator strictly avoids these instructions and only permits direct, unconditional control flow changes. As the only emitted branch is the unconditional branch at the end of the program, adding a memory fence after this jump prevents Straightline Spectre. Due to the unconditional nature of the branch, this memory fence is never executed architecturally, and has therefore no performance impact. Hence, programs compiled with Specfuscator, by design, cannot be susceptible to the above Spectre variants as the corresponding instructions are not present in the binary. This is a very clear and strong security guarantee that most other defenses cannot provide [31,12].

Specfuscator is a software-only solution and does not require hardware modifications like other proposed Spectre defense mechanisms [44,28,27]. Thus, it can even work in environments where other mitigations cannot be applied, e.g., because `lfence` instructions are not serializing [2], or patches are unavailable for other reasons.

4.3 Implementation of Specfuscator

Specfuscator is a modification to the LCC C compiler [19]. The reason we chose LCC and not gcc or clang is that we base the implementation of Specfuscator on the open-source *M/o/Vfuscator*, as this compiler already generates a branch-free binary based on the technique from Dolan [48]. *M/o/Vfuscator* itself is a

patch to the current version (September 2020) of LCC. However, we require several custom changes, as outlined in Section 4.1. In contrast to *M/o/Vfuscator*, Specfuscator can use a broader range of native instructions without sacrificing security. By relying on arithmetic and logic operations, as well as complex addressing modes, the amount of `mov` instructions is reduced heavily, *i.e.*, for the addition, we now have 3 instructions instead of 50 `mov` instruction. For example, in a tiny AES program, the number of instructions is reduced from 222 935 to 127 631, *i.e.*, a reduction of about 43 %, when compiling with Specfuscator instead of *M/o/Vfuscator*.

As all of our changes are in the code emitter of the compiler, this could also be ported to a different compiler, such as clang. As Specfuscator is based on *M/o/Vfuscator*, we can already adopt the control-flow-linearization code from *M/o/Vfuscator* but also emit arithmetic and logic operations. Divisions and modulo operations require additional handling, as they can cause floating-point exceptions in case of a division by zero. We handle those special cases using conditional `mov` (`cmov`) instructions to ensure that we do not introduce conditional branches. The conditional `mov` instructions, e.g., `cmov`, are not affected by Spectre, as they are never predicted [25].

For comparisons, we cannot merely emit the x86 instructions instead of the `mov`-based constructs, as *M/o/Vfuscator* uses its own internal representation of CPU flags to select whether the computation results of a branch are stored or discarded. Hence, to ensure correct branching with e.g., , the `cmp` instruction, we need to update the internal flags in a branch-free way. We achieve this by transferring the CPU flags to an unused general-purpose register via the stack and using binary masks to extract the required bits.

In total, we changed (added, removed, or replaced) 437 lines of code of *M/o/Vfuscator*, which is about 10 % of the *M/o/Vfuscator* codebase.

5 Evaluation

In this section, we first verify the security of Specfuscator by compiling and executing Spectre-PHT, Spectre-BTB, and Spectre-RSB gadgets. Furthermore, we evaluate the performance of Specfuscator and compare it to the original *M/o/Vfuscator*, LCC, and a modified clang version, which emits `lfences` for each conditional branch, and a basic clang compiler without Spectre mitigations activated. We compare each compiler on a set of benchmark programs and compare the averaged runtime performance, binary size, and compile time. The results of this evaluation are given in Table 1 and Table 2. Our test system was equipped with Ubuntu 20.04 (5.4.0-42-lowlatency) running on an Intel i5-8250U CPU.

5.1 Security Evaluation

We demonstrate that it is impossible to successfully use an existing Spectre proof-of-concept attack on Specfuscator compiled code. To verify that the *misspeculation* is indeed prevented, we separately validate all other Spectre attack

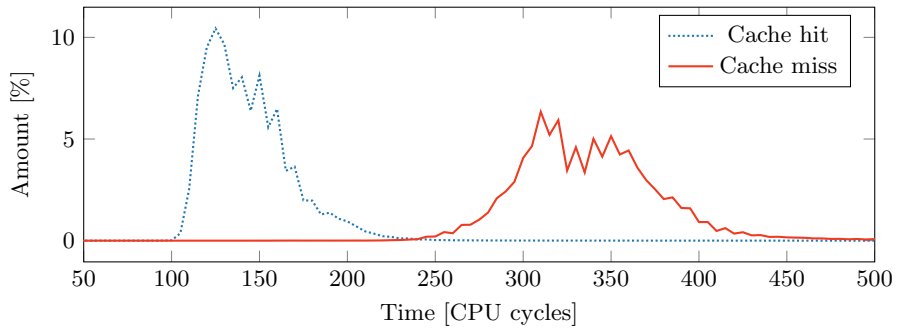

Fig. 4: Flush+Reload within a Specfuscator-compiled program works successfully as intended.

steps. We add additional functionality to the compiled binaries to obtain accurate time measurements with `rdtsc` and enable flushing of a virtual address using the `clflush` instruction. This allows us to accurately verify the cache encoding of the Spectre attack with a Flush+Reload side-channel.

We verify that the cache covert channel in a compiled binary works exactly as in a regular Spectre attack by creating a histogram of cached and uncached data. Figure 4 shows that it is still possible to distinguish between cached and uncached data in a program compiled with Specfuscator. Therefore, cache-based side-channel attacks are still possible in Specfuscator-compiled programs.

To validate whether Spectre is still possible, we use the 15 sample Spectre gadgets from Kocher [30]. First, we evaluate that these gadgets indeed sucessfully show Spectre attacks. We compile them using the unmodified LCC and execute each gadget 100 000 times. For all gadgets, we successfully leak data using Spectre.

For the security evaluation, we compile all sample gadgets using Specfuscator. We again execute each gadget 100 000 times on our test device, and check whether the secret is leaked. As we do not observe any leakage on our test device using any of the gadgets, we practically confirm that our mitigation that should work in theory due to the absence of misspeculation, also works in practice.

In addition, we port a Spectre-BTB and Spectre-RSB proof-of-concept to 32-bit and evaluate it on our unmitigated clang. Again, as expected, these proof-of-concepts work on an unmitigated clang. When the programs are compiled with Specfuscator, no indirect jumps, calls, or return instructions are emitted. To experimentally show that Specfuscator indeed stops the leakage for these attacks, we again compile them using our defense. We execute the proof-of-concept implementations 100 000 times and do not observe any leakage for either Spectre-BTB or Spectre-RSB.

5.2 Performance Evaluation

For the evaluation, we extend LLVM 10.0.1 with a new compiler pass that runs just before the final code is emitted. In this pass, we analyze every conditional branch using the `analyzeBranch` function and insert an `lfence` instruction if this instruction is not already present. To mitigate speculation on both sides of a conditional branch, we also emit an `lfence` instruction in its fall-through basic block if this code path is not already fenced. This compiler pass required changing or adding 125 lines of code across 4 files. In addition to enabling our fencing pass, we enable the `retpoline` mitigation for `clang` by adding the `-mretpoline` flag. As a result, speculative execution is stopped for all conditional branches and jumps, as e.g., , suggested by Intel [24].

For our evaluation, we compare different programs, including cryptographic implementations and real-world applications [14]. We compile each program as a 32-bit binary since our Specfuscator proof-of-concept only supports 32-bit. However, while we showcase our compiler for this architecture, our approach is generic and is equally applicable to other architectures as well.

We compile the same benchmark program in 5 different configurations. Each test case is compiled with `clang` without any Spectre mitigations, `clang` with `lfences` and `retpoline` active, the LCC, the unmodified *M/o/Vfuscator*, and Specfuscator. To get stable benchmarking results, we fixed the CPU frequency to 3.4 GHz and ran our test program on an isolated core.

Run time

We use the runtime of the `clang`-compiled programs without mitigations as a baseline to compute the runtime overhead. To measure the runtime of the programs, we use the `perf` command-line tool. We run each test case 1000 times. For the individual test cases, we observe standard deviations between 0.1 % and, for some cases, 3 %. The maximum value of 3 % was observed in the case of clang. The reason for this higher standard deviation might be speculative execution.

As shown in Table 1, the runtime overhead factor strongly depends on the different tasks being executed. We gained a runtime speedup in comparison to *M/o/Vfuscator* by a runtime factor of up to 50. For our benchmark programs, we observe that the LCC has a runtime overhead between 3 % and an overhead factor of 26 over `clang`. The overhead of *M/o/Vfuscator* is substantially higher, and the overhead of Specfuscator is in between. We observe the highest performance penalties in terms of runtime for a tiny program that calculates the square root of 2. Also, the modified clang reaches a maximum runtime overhead factor of 20.89. The performance of *M/o/Vfuscator* and Specfuscator deteriorates, particularly on programs where small amounts of code are executed a large number of times, as the whole program has to be completely executed for each iteration.

We leave it as future work to further optimize Specfuscator optimizing the way how branches are performed. Partial control flow linearization could be integrated as compiler optimization with a similar approach proposed by Moll et al.

Table 1: Average runtime overhead factor of our benchmarks for the different compilers compared to our baseline (clang). The baseline is given in milliseconds on the right for the unmodified `clang`

Test program	*M/o/Vfuscator*	Specfuscator	Clang (fences)	LCC	Clang (baseline)
aes	424.17	221.53	1.31	1.17	1.13 ms
arcfour	36.86	5.18	1.01	1.14	0.81 ms
base64	27.12	8.95	1.19	1.15	0.80 ms
blowfish	129.41	40.79	1.26	1.14	1.10 ms
des	1046.20	520.47	1.15	1.04	0.93 ms
md2	85.57	62.73	1.07	1.20	0.82 ms
md5	18.30	4.71	1.03	1.13	0.80 ms
rot-13	2.20	1.46	1.02	1.24	0.76 ms
arithmetic	1.25	1.05	1.05	1.03	0.96 ms
crc32	7.80	3.45	1.24	1.17	0.88 ms
hello	1.10	1.11	1.00	1.04	0.89 ms
maze	310.03	88.98	1.10	1.13	0.97 ms
mersenne	4.12	1.31	1.02	1.13	0.80 ms
sqm	1.33	1.25	1.02	1.15	0.80 ms
nqueens	319.84	234.46	1.99	4.99	1.89 ms
prime	980.27	161.59	1.93	0.96	1.65 ms
s2	46085.82	981.20	20.89	26.64	0.71 ms
sudoku	656.91	149.69	2.15	1.17	1.13 ms

[37]. The partial control flow linearization improved the performance of the overall program by a factor of 146 %[37]. Furthermore, we leave it as future work to extend Specfuscator to 64-bit architecture or integrating a similar approach to LLVM. As LLVM has significantly better optimizations than LCC, as can be seen in the benchmarks, porting Specfuscator to LLVM will also improve its performance.

In addition to the runtime, we evaluate the binary size and compile time of the different compilers. For this purpose, we compile each program 1000 times for our 5 compilers and measure the compilation time using the `perf` command-line tool. Table 2 illustrates the averaged overhead factor in terms of binary size and compilation time.

Compile-time

Table 2 lists the compile-time and the binary size of our benchmark programs. In comparison to *M/o/Vfuscator*, we reduce the compile time by up to 46 %. The compile-time of *M/o/Vfuscator* and Specfuscator depends on a part in how

Table 2: Average compile time in ms and binary size in kB overhead factor for *M/o/Vfuscator*, Specfuscator, and `clang` with active mitigations compared to `clang` without active mitigations (rightmost column).

Test program	*M/o/Vfuscator*		Specfuscator		Clang (fences)		LCC		Clang (baseline)	
	time	size	time	size	time	size	time	size	time	size
hello	2.23	388.28	1.86	279.18	1.07	1.01	0.71	0.89	38.36 ms	13.62 kB
maze	3.93	394.09	2.12	274.96	1.05	1.01	0.63	0.86	46.80 ms	13.82 kB
mersenne	3.00	396.28	1.83	279.84	1.02	1.01	0.70	0.89	41.90 ms	13.63 kB
nqueens	2.39	386.75	2.05	278.22	1.17	1.01	0.75	0.88	40.19 ms	13.64 kB
prime	2.39	389.97	1.81	279.47	1.06	1.01	0.62	0.89	39.02 ms	13.64 kB
s2	2.87	395.22	1.89	279.72	1.00	1.01	0.78	0.89	39.34 ms	13.62 kB
sudoku	3.47	398.10	2.05	280.39	1.10	1.01	0.68	0.91	37.76 ms	14.00 kB
aes	4.80	218.69	2.95	151.15	1.20	1.00	0.53	1.01	101.89 ms	33.21 kB

many instructions are needed to generate the assembler. Thus, with the use of fewer instructions per operation, the compile-time is halved in most cases for Specfuscator in comparison to the original *M/o/Vfuscator*. As the results of Table 2 show, the compile-time is about two times higher than with the `clang` compiler. For small programs, the compile-time appears to be relatively constant for the *M/o/Vfuscator* and also Specfuscator. While this is not problematic for smaller binaries, compiling large software projects such as browsers or web servers would take substantial amounts of time with Specfuscator. We note that our approach of eliminating all conditional branches is extreme. Still, it shows that solutions that eliminate conditional branches are not infeasible, and less extreme solutions in this direction could maintain higher performance levels.

Binary size

Stripping the binary reduces the binary size by 50 %, as it removes debugging information. Hence, for a fair comparison, we strip all the binaries to only compare the actual code footprint. Compared to *M/o/Vfuscator*, Specfuscator reduces the binary size by roughly 30 % This reduction was achieved by removing most of the two-dimensional lookup tables used for arithmetic operations. The binary size could additionally be reduced by decreasing the size of the virtual stack, which is currently constant at 1.68 MB. As can be seen from Table 2, the binary size is about 280 times larger for Specfuscator than for binaries compiled with `clang` and for *M/o/Vfuscator* even 398 times. Again, this overhead is due to our extreme solution, but it shows that solutions eliminating conditional branches are not infeasible. Surprisingly, the programs compiled with LCC are smaller than the programs compiled with the unmodified `clang`.

6 Discussion

The goal of our paper is to clearly demonstrate the feasibility of branch reduction up to complete elimination as a Spectre mitigation. While we demonstrated the feasibility, we also identified the limitations of our extreme approach. Due to these limitations, we do not consider Specfuscator a real-world solution, but an important contribution as an explorational study that yields interesting insights. Eliminating all branches to reduce the susceptibility to Spectre has not been explored so far. Our solution inherits the performance overheads of the underlying compiler (LCC and its modification *M/o/Vfuscator*) that falls far behind the state of the art performance-wise. The fact that it can still achieve on-par performance for specific programs protected with state-of-the-art mitigations with a state-of-the-art compiler shows that the elimination or reduction of branches is a strategy to defeat Spectre that must be examined in more detail. In particular, we see potential synergies with the compiler community that explored the question of branch elimination in the past for performance reasons. For instance, Moll et al. [37] developed a technique to partially linearize the program flow by removing branches, improving performance by 146 %. Exploring related techniques, even if they incur a subtle performance overhead, may yield more efficient Spectre mitigations in future compilers. Software-based solutions are especially important as there is a lot of hardware without in-silicon fixes, and existing software-workarounds are often expensive. While Intel recommends keeping the number of branches as low as possible to achieve the highest possible runtime performance [26], actually reducing branches is a complex task. Although branch elimination can boost the program's performance, it might also be exploited, as it has been demonstrated in the JavaScript engine V8 [47,17]. Another direction of research is to investigate the susceptibility to control-flow hijacking attacks. Future work should evaluate whether branch-less binaries, like those compiled with Specfuscator, or branch-reduced binaries, could realistically mitigate such attacks and, thus, provide control-flow integrity.

7 Conclusion

Speculative execution attacks, known as Spectre attacks, have gained substantial attention both in the scientific community and in industry with a broad range of defense techniques proposed. In particular, in-software defenses for commodity systems attempt to leave the program structure as is, but defuse every potential Spectre gadget, e.g., by stopping the speculation, or limiting value ranges. While these mitigations disrupt the program flow on every conditional branch, they still contain every single conditional branch instruction. In this work, we explore a new possibility of mitigating Spectre attacks by using a branch-free compiler. Our mitigation is based on automatically linearizing the program flow through a special compiler pass, eliminating **all** conditional and indirect branches. We showed the security guarantees of this approach and evaluated the feasibility by evaluating its performance in terms of its runtime. In addition, we discussed

the compile-time and the binary size of this approach. Furthermore, we verified that existing Spectre-PHT, Spectre-BTB, and Spectre-RSB proof-of-concepts compiled with Specfuscator do not leak secret data anymore. We observe that the performance overhead can be very low, e.g., 5 %, for specific use cases, being on-par with state-of-the-art mitigations. However, we also observed very high overheads of factor 1000 for other use cases. Our results indicate that the best performance-security trade-off for Spectre defenses can be achieved by sticking to neither of the extremes.

Acknowledgments

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No 681402). Funding was provided by generous gifts from Cloudflare, from Intel, and from ARM. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

1. Vulnerability of Speculative Processors to Cache Timing Side-Channel Mechanism (2018), https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability
2. x86/cpu/AMD: Make LFENCE a serializing instruction (2018), https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=e4d0e84e490790798691aaa0f2e598637f1867ec
3. Advanced Micro Devices Inc.: Software Techniques for Managing Speculation on AMD Processors (2018), revison 7.10.18
4. Amit, N., Jacobs, F., Wei, M.: Jumpswitches: restoring the performance of indirect branches in the era of spectre. In: USENIX ATC (2019)
5. ARM: Cache Speculation Side-channels (2018), version 2.4
6. ARM: Straight-line Speculation (2020), version 1.0
7. Bernstein, D.J.: Cache-Timing Attacks on AES (2005), http://cr.yp.to/antiforgery/cachetiming-20050414.pdf
8. Bhattacharya, S., Maurice, C.m.t.n., Bhasin, S., Mukhopadhyay, D.: Template Attack on Blinded Scalar Multiplication with Asynchronous perf-ioctl Calls. Cryptology ePrint Archive, Report 2017/968 (2017)
9. Bhattacharyya, A., Sandulescu, A., Neugschwandt ner, M., Sorniotti, A., Falsafi, B., Payer, M., Kurmus, A.: SMoTherSpectre: exploiting speculative execution through port contention. In: CCS (2019)
10. Branco, R., Hu, K., Sun, K., Kawakami, H.: Efficient mitigation of side-channel based attacks against speculative execution processing architectures (2019), uS Patent App. 16/023,564
11. Canella, C., Pudukotai Dinakarrao, S.M., Gruss, D., Khasawneh, K.N.: Evolution of Defenses against Transient-Execution Attacks. In: GLSVLSI (2020)

12. Canella, C., Van Bulck, J., Schwarz, M., Lipp, M., von Berg, B., Ortner, P., Piessens, F., Evtyushkin, D., Gruss, D.: A Systematic Evaluation of Transient Execution Attacks and Defenses. In: USENIX Security Symposium (2019), extended classification tree and PoCs at https://transient.fail/.
13. Carruth, C.: RFC: Speculative Load Hardening (a Spectre variant #1 mitigation) (2018)
14. Christopher Domas: M/o/Vfuscator (2015), `https://github.com/xoreaxeaxeax/movfuscator`
15. Coron, J.S.: Resistance against differential power analysis for elliptic curve cryptosystems. In: CHES (1999)
16. Evtyushkin, D., Ponomarev, D., Abu-Ghazaleh, N.: Jump over aslr: Attacking branch predictors to bypass aslr. In: MICRO (2016)
17. Fetiveau: Circumventing Chrome's hardening of typer bugs (2019), `https://doar-e.github.io/blog/2019/05/09/circumventing-chromes-hardening-of-typer-bugs/`
18. Fog, A.: The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers (2016)
19. Fraser, C.W., Hanson, D.R.: A retargetable C compiler: design and implementation (1995)
20. Fustos, J., Farshchi, F., Yun, H.: SpectreGuard: An Efficient Data-centric Defense Mechanism against Spectre Attacks. In: DAC (2019)
21. Horn, J.: speculative execution, variant 4: speculative store bypass (2018)
22. Intel: Intel Analysis of Speculative Execution Side Channels (2018), revision 4.0
23. Intel: Retpoline: A Branch Target Injection Mitigation (2018), revision 003
24. Intel: Speculative Execution Side Channel Mitigations (2018), revision 3.0
25. Intel: Intel 64 and IA-32 Architectures Optimization Reference Manual (2019)
26. Intel: Avoiding the Cost of Branch Misprediction (2020), `https://software.intel.com/content/www/us/en/develop/articles/avoiding-the-cost-of-branch-misprediction.html`
27. Khasawneh, K.N., Koruyeh, E.M., Song, C., Evtyushkin, D., Ponomarev, D., Abu-Ghazaleh, N.: SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation. In: DAC (2019)
28. Kiriansky, V., Lebedev, I., Amarasinghe, S., Devadas, S., Emer, J.: DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. In: MICRO (2018)
29. Kirsch, J., Jonischkeit, C., Kittel, T., Zarras, A., Eckert, C.: Combating control flow linearization. In: 32nd International Conference on ICT Systems Security and Privacy Protection (IFIP SEC) (May 2017), `https://www.sec.in.tum.de/i20/publications/combating-control-flow-linearization/@@download/file/CFL.pdf`
30. Kocher, P.: Spectre Mitigations in Microsoft's C/C++ Compiler (2018)
31. Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., Yarom, Y.: Spectre Attacks: Exploiting Speculative Execution. In: S&P (2019)
32. Koruyeh, E.M., Khasawneh, K., Song, C., Abu-Ghazaleh, N.: Spectre Returns! Speculation Attacks using the Return Stack Buffer. In: WOOT (2018)
33. Lee, S., Shih, M., Gera, P., Kim, T., Kim, H., Peinado, M.: Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In: USENIX Security Symposium (2017)
34. Maisuradze, G., Rossow, C.: ret2spec: Speculative Execution Using Return Stack Buffers. In: CCS (2018)

35. Microsoft: Mitigating speculative execution side-channel attacks in Microsoft Edge and Internet Explorer (2018)
36. Miller, M.: Mitigating speculative execution side channel hardware vulnerabilities (2018)
37. Moll, S., Hack, S.: Partial control-flow linearization. In: Proceedings of the 39th Conference on Programming Language Design and Implementation. pp. 543–556. ACM (2018)
38. Molnar, D., Piotrowski, M., Schultz, D., Wagner, D.: The program counter security model: Automatic detection and removal of control-flow side channel attacks. In: International Conference on Information Security and Cryptology (2005)
39. Oleksenko, O., Trach, B., Reiher, T., Silberstein, M., Fetzer, C.: You Shall Not Bypass: Employing data dependencies to prevent Bounds Check Bypass. arXiv:1805.08506 (2018)
40. Palit, T., Monrose, F., Polychronakis, M.: Mitigating data leakage by protecting memory-resident sensitive data. In: ACSAC (2019)
41. Pardoe, A.: Spectre mitigations in MSVC (2018)
42. Pizlo, F.: What Spectre and Meltdown mean for WebKit (2018)
43. Reis, C., Moshchuk, A., Oskov, N.: Site Isolation: Process Separation for Web Sites within the Browser. In: USENIX Security Symposium (2019)
44. Schwarz, M., Lipp, M., Canella, C., Schilling, R., Kargl, F., Gruss, D.: ConTExT: A Generic Approach for Mitigating Spectre. In: NDSS (2020)
45. Schwarz, M., Schwarzl, M., Lipp, M., Gruss, D.: NetSpectre: Read Arbitrary Memory over Network. In: ESORICS (2019)
46. Schwarzl, M., Schwarz, M., Schuster, T., Gruss, D.: It's not Prefetch: Speculative Dereferencing of Registers. (in submission) (2020)
47. Sense Post: v8 - Documentation (2020), `https://sensepost.com/blog/2020/intro-to-chromes-v8-from-an-exploit-development-angle/`
48. Stephen Dolan: mov is Turing-complete (2013), `https://drwho.virtadpt.net/files/mov.pdf`
49. The Chromium Projects: Actions required to mitigate Speculative Side-Channel Attack techniques (2018)
50. Turner, P.: Retpoline: a software construct for preventing branch-target-injection (2018), `https://support.google.com/faqs/answer/7625886`
51. Wagner, L.: Mitigations landing for new class of timing attack (2018)
52. Weisse, O., Neal, I., Loughlin, K., Wenisch, T.F., Kasikci, B.: Nda: Preventing speculative execution attacks at their source. In: MICRO (2019)
53. Yan, M., Choi, J., Skarlatos, D., Morrison, A., Fletcher, C.W., Torrellas, J.: InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy. In: MICRO (2018)
54. Yarom, Y., Falkner, K.: Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In: USENIX Security Symposium (2014)
55. Yu, J., Hsiung, L., El Hajj, M., Fletcher, C.W.: Data Oblivious ISA Extensions for Side Channel-Resistant and High Performance Computing. In: NDSS (2019)
56. Yu, J., Yan, M., Khyzha, A., Morrison, A., Torrellas, J., Fletcher, C.W.: Speculative taint tracking (stt) a comprehensive protection for speculatively accessed data. In: MICRO (2019)