Restricting Control Flow During Speculative Execution with Venkman

Zhuojia Shen, Jie Zhou, Divya Ojha, and John Criswell

Department of Computer Science
University of Rochester

Abstract

Side-channel attacks such as Spectre that utilize speculative execution to steal application secrets pose a significant threat to modern computing systems. While program transformations can mitigate some Spectre attacks, more advanced attacks can divert control flow speculatively to bypass these protective instructions, rendering existing defenses useless.

In this paper, we present *Venkman*: a system that employs program transformation to completely thwart Spectre attacks that poison entries in the Branch Target Buffer (BTB) and the Return Stack Buffer (RSB). Venkman transforms code so that all valid targets of a control-flow transfer have an identical alignment in the virtual address space; it further transforms all branches to ensure that all entries added to the BTB and RSB are properly aligned. By transforming all code this way, Venkman ensures that, in any program wanting Spectre defenses, all control-flow transfers, including speculative ones, do not skip over protective instructions Venkman adds to the code segment to mitigate Spectre attacks. Unlike existing defenses, Venkman does not reduce sharing of the BTB and RSB and does not flush these structures, allowing safe sharing and reuse among programs while maintaining strong protection against Spectre attacks. We built a prototype of Venkman on an IBM POWER8 machine. Our evaluation on the SPEC benchmarks and selected applications shows that Venkman increases execution time to 3.47× on average and increases code size to 1.94× on average when it is used to ensure that fences are executed to mitigate Spectre attacks. Our evaluation also shows that Spectre-resistant Software Fault Isolation (SFI) built using Venkman incurs a geometric mean of 2.42× space overhead and 1.68× performance overhead.

1 Introduction

Spectre attacks [20] pose a significant threat to computing systems. Such attacks can be launched by unprivileged code and leverage speculative execution within processors to trick victim programs into leaking confidential data. Current Spectre attacks first direct control flow into infeasible program paths which load sensitive data (often loaded from an out-of-bounds array access) into a processor register and then cause the victim to leak the information via a side channel. To date, Spectre has been used by malicious processes to steal information from other victim processes [20], by malicious JavaScript code to steal information from the web browser [20], and by malicious code to steal secrets contained within Trusted Execution Environments (TEEs) like Intel SGX [34]. Spectre attacks work on Intel and AMD x86 processors and on ARM processors [20], and this work demonstrates that Spectre attacks work on the IBM POWER8 processor. Any processor implementing speculative execution and branch prediction is likely vulnerable to Spectre attacks. Consequently, Spectre poses a significant threat to nearly every laptop, desktop, and server computer.

Variant-1 Spectre attacks [20] require that victim programs contain specific code patterns that are exploitable during speculative execution. Variant-2 Spectre attacks which poison the processor's Branch Target Buffer (BTB) [20] or Spectre variants that poison the Return Stack Buffer (RSB) [21, 24] direct a victim's control flow to small pieces of code that exhibit the same behavior even if such paths are infeasible in the program's non-speculative control flow. Spectre attacks which poison the BTB or RSB [20, 21, 24] are especially nefarious as attackers can use them to bypass instructions inserted into the program that mitigate Spectre attacks. For example, Intel suggests adding fence instructions to mitigate Spectre Variant-1 [16], and Dong et al. [9] propose a software fault isolation (SFI) [35] mechanism that operates correctly even when subjected to a Spectre Variant-1 attack. However, if an attacker poisons the BTB or RSB, speculative execution can jump straight to load instructions without first executing the fence or SFI instructions.

For some processors, there are microcode updates that allow the operating system (OS) kernel to limit sharing of the BTB and provide new mechanisms for flushing the BTB and RSB [16]. However, only processors that use microcode can be updated without a physical processor replacement, and microcode updates are not available for all afflicted proces-

sors [15]. The only software defense is the retpoline; versions exist for mitigating BTB poisoning [33] and RSB poisoning [24]. However, all retpolines contain an unconditional direct branch instruction. Evidence shows that direct branches consult the BTB just as computed branches do [10]. This means that retpolines are likely vulnerable to BTB poisoning. Additionally, retpolines use return instructions to branch to target addresses. This makes them inherently incompatible with control-flow integrity (CFI) defenses [3].

We present a new, comprehensive, software-only defense against Spectre attacks that poison the BTB and RSB. Named *Venkman*, our solution performs two transformations on code to mitigate Spectre attacks. The first transforms *all* code on a system so that it cannot poison the BTB and RSB with target addresses of the attacker's choosing. Venkman transforms programs to group instructions into *bundles*. All bundles have the same power-of-two length and are aligned at the same power-of-two boundary in memory. Venkman then transforms the program so that all computed branches first align the target to a bundle boundary. The net effect is that all BTB and RSB entries for all code on the system are addresses at the beginning of a bundle. We also propose a system architecture that ensures that all binary code running on a system has been transformed as described, ensuring that only aligned bundle addresses are inserted into the BTB and RSB. The second transformation modifies programs wanting defense against Spectre attacks by adding instructions into their bundles that mitigate Spectre attacks. For example, this second transformation can insert fence instructions into each bundle to mitigate all Spectre attacks (as suggested by previous defenses for Spectre variant-1 [16]), or it can insert Spectre-resistant SFI instructions [9] into bundles containing load instructions. Since Venkman ensures that instructions that need protection from Spectre attacks (such as loads) are in the same bundle as the instructions providing the protection (such as fences), training of the BTB and RSB cannot cause execution to bypass the instructions providing protection. The second transformation can be excluded on programs that don't want Spectre defenses.

We built a prototype of Venkman for the POWER architecture. Our IBM POWER8 machine employs speculative and out-of-order execution, and we have demonstrated that the Spectre proof-of-concept code [20], ported to POWER and changed to poison the BTB, works on our POWER8 machine. We evaluated the performance of our prototype on the SPEC CPU 2017 benchmarks and on several real-world applications. In geometric means, our results show that the bundling transformation incurs a performance overhead of 1.09× and a space overhead of 1.61×. Our results also show that Venkman, when used to ensure that fence instructions are executed to mitigate Spectre attacks, increases execution time to 3.47× on average and increases code size to an average of 1.94×. We have also used Venkman to build a Spectre-resistant sandbox using SFI; this system has a geometric mean of 2.42× space overhead and 1.68× performance overhead.

To summarize, our contributions are as follows:

- We have designed a complete software-only solution that prevents poisoning of the BTB and RSB. To the best of our knowledge, Venkman is the first complete software-only solution to such attacks.
- We have evaluated the overheads that Venkman's padding and alignment transformations incur and found that our solution induces a geometric mean of 1.61× space overhead and 1.09× performance overhead.
- We have evaluated the performance overheads of adding barrier instructions to mitigate Spectre attacks in code requiring defense from Spectre attacks. We found that our solution induces a geometric mean of 1.94× space overhead and 3.47× performance overhead.
- We have evaluated Venkman in providing an SFI system that resists Spectre attacks with a geometric mean of 2.42× space overhead and 1.68× performance overhead.

The rest of the paper is organized as follows. Section 2 provides background on Spectre attacks. Section 3 describes our threat model. Section 4 describes the design of our defense, and Section 5 describes the implementation of our prototype. Section 6 conducts an empirical study of Venkman's security guarantees. Section 7 presents our space and performance evaluation of Venkman, Section 8 examines related work, and Section 9 concludes and discusses future work.

2 Background on Spectre Attacks

Spectre attacks [18, 20, 21, 24] are a family of attacks that leverage speculative execution to trick a victim program into speculatively executing a sequence of instructions which it normally would not execute. Although the processor will eventually revert the architectural effects of speculatively executed instructions [31], the execution of the instructions may change the state of internal structures (such as the caches and branch prediction buffers) within the processor. A typical Spectre attack consists of two basic steps. First, a processor is tricked into speculatively executing instructions chosen by an adversary which load secret data into the processor's registers. Second, the adversary uses a side channel, such as a FLUSH+RELOAD [36] attack on the caches, as a covert channel to leak the secret information in the registers to the attacker. There are two major variants of Spectre attacks. One exploits conditional branches; the other poisons indirect branches and returns [20, 21, 24].

2.1 Exploiting Conditional Branches

For a conditional branch, modern processors predict whether the branch will or will not be taken and proceed to specula-

```
1 if (x < arr1_boundary) {
2     y = arr2[arr1[x] * 256];
3 }
```

Listing 1: Conditional Branch Example

```
1 (*func_ptr)();
2
3 if (x < arr1_boundary) {
4     load_fence();
5     y = arr2[arr1[x] * 256];
6 }
```

Listing 2: Indirect Branch Example

tively execute the instructions that it predicts are needed next. This keeps the processor pipeline busy, increasing throughput [31].

Consider the conditional branch code in Listing 1. During a Variant-1 Spectre attack [20], before the condition at Line 1 is resolved, the processor proceeds to speculatively execute code at Line 2 if the branch predictor predicts that the condition is true. After the processor determines that `x` is not less than the length of the array `arr1`, it reverts changes it has made to registers. However, data brought into the cache during speculative execution remains in the cache. Therefore, `arr2`'s element whose index is `arr1[x] * 256` is still within the cache even though it is not needed. If the variable `x` is controlled by an attacker, and a secret value is located at `arr1[x]`, then the attacker can infer the secret data by using a side-channel attack such as FLUSH+RELOAD [36] or PRIME+PROBE [28].

Intel [16] recommends placing a load fence (`lfence`) instruction [14] before instructions reading memory that are control-dependent on branches; a fence makes sure that all prior instructions are retired before executing subsequent instructions, ensuring that the load executes only if it was supposed to be executed.

2.2 BTB and RSB Poisoning

For an indirect branch, before the destination address is resolved, the processor consults the BTB (or RSB if the indirect branch is a return instruction) to predict the next address from which to fetch instructions [31]. Similar to branch prediction, this optimization improves the processor's throughput.

However, an adversary process can poison the BTB and RSB and trick a victim process into speculatively executing code gadgets chosen by the adversary [20, 21, 24]. Two features enable BTB and RSB poisoning. First, all processes running on the same physical CPU core share the same BTB and RSB [11]. Second, the operating system kernel does not save or flush the state of the BTB and RSB when context switching between processes or threads, allowing one process to add values into the BTB and RSB that are used by a subsequent process running on the same core. Therefore, a malicious process can mistrain the BTB and RSB to fill in target addresses to which it wants a victim to jump. For example, in Listing 2, an attacker could train the BTB entry for the call through a function pointer at Line 1 so that it speculatively jumps straight to Line 5, bypassing the load fence placed before the load at Line 4. With BTB and RSB poisoning, an attacker can trick a victim process into executing any code within the victim's code segment; the attacker can target code that leaks sensitive information as described in Section 2.1.

While current Spectre attacks use BTB poisoning to alter speculative control flow for indirect branches [20], we believe they can also change the speculative control flow of direct branches. Evtyushkin et al. [10] demonstrated that an unconditional jump can be used to create BTB collisions; they trained the BTB entries used by direct branches to launch side-channel attacks against the OS kernel. Even though the target of a direct branch is specified as an immediate operand to the instruction, the processor may still use the BTB to predict the target of a direct branch; this allows the processor to start fetching instructions at the target of the branch during the fetch stage of the processor pipeline before learning that an instruction is a branch during the decode phase of the pipeline [31].

The poisoning of targets for direct branches threatens to break retpolines. Retpolines [24, 33] are the only existing software defense for Spectre attacks that poison the BTB and RSB. Retpolines use direct branches to jump to code within the retpoline. An attacker could poison the BTB entry used by the direct branch, causing the retpoline to speculatively execute code at an attacker's desired location. While existing Spectre attacks have only exploited the indirect branches [20], we believe that direct branches are likely to be exploited at some point. Unlike retpolines [24, 33], Venkman can mitigate any type of BTB poisoning as it ensures that only valid addresses are inserted into the BTB and RSB.

2.3 Read-Only Protection Bypass

Similar to how Meltdown [23] leverages the late enforcement of user/supervisor protection flags, an attacker can exploit the late checking of read and write permissions on pages to change the value read from a write-protected memory location during speculative execution [18]. Speculative stores combined with store-to-load forwarding could corrupt the code segment if the instruction fetch unit can read values speculatively written to memory that are stored in the processor's store buffer. This could allow an attacker to modify instructions added by a compiler that mitigate Spectre attacks. For example, the attacker could replace fences or SFI instructions with NOP instructions, effectively disabling the protections. Any defense using compiler instrumentation techniques must defend against speculative modifications to the code segment.

3 Threat Model

Our threat model assumes that an attacker will attempt to steal data with a Spectre attack [20] that poisons the BTB and/or RSB. Since some processors predict direct branch targets using the BTB [10], we assume that the attack can poison both direct and indirect branches and calls. Our model is restricted to attacks that leak data through the cache via data accesses. Other potential attacks that leak data through other side channels, such as the instruction cache, translation look-aside buffers (TLBs), branch predictors, and functional units, are out of scope.

Our model is very broad: any piece of software may be a potential attacker. This includes all user-space software as well as the OS kernel. A subset of software on the system comprises potential victims of the attack. This model covers attacks by one application against another, attacks by a distrusted part against another within one application, attacks launched by applications against the OS kernel, and attacks launched by a compromised OS kernel against applications (similar to the Foreshadow attack [34]).

Our model assumes that the hardware is implemented correctly with respect to the processor's instruction set architecture (ISA), meaning that the processor updates memory and processor registers correctly but that speculative execution performed by the processor may allow Spectre attacks [20] to leak information through microarchitectural state.

4 Design

Spectre variants that poison the BTB [20] and RSB [21, 24] work because one program can insert entries into the BTB and RSB that are correct for its address space but incorrect for a victim program in the victim program's address space. Venkman transforms all code running on the system so that *any* BTB or RSB entry created by one program does not cause any other program on the system to bypass fence instructions (or other instructions inserted by a compiler) that protect load instructions from Spectre attacks. Venkman transforms code so that instructions are grouped into bundles and then instruments branches to ensure that they can only target the first instruction in each bundle. As long as load instructions and the instructions that protect them are within the same bundle, attackers cannot execute a load without first executing the protecting instructions. In short, Venkman ensures that branches can only insert the initial address of a bundle into the BTB and RSB.

All code running on a system must be transformed as described above. We first describe a software architecture that can ensure that *all* code on a system has been transformed using Venkman. We then present how Venkman lays out the virtual address space of the system to facilitate its instrumentation and how Venkman transforms code to ensure that only "safe" code addresses are inserted into the BTB and RSB.

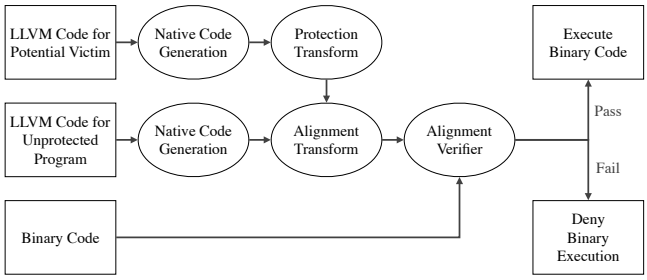

Figure 1: Venkman Architecture

Finally, we present how Venkman can be used to ensure that fences are used to prevent loads from accessing invalid memory and how Venkman can be integrated with Spectre-resistant SFI [9] to provide speculation-safe sandboxing.

4.1 Venkman Architecture

Venkman must ensure that all code running on the system is transformed so that the code only inserts "safe" values into the BTB and RSB as Sections 4.3 and 4.4 will describe. We therefore need a system that can transform code on site if possible and verify that binary code from third parties has already been transformed by Venkman.

Figure 1 shows Venkman's architecture. Venkman supports programs encoded in one of two formats. The first format is native binary code; this is how software is shipped today. Venkman cannot statically transform such programs due to the challenges of static binary rewriting e.g., accurate disassembly of the native code and reconstruction of its control-flow graph (CFG). However, a binary verifier can verify that the native code within the executable has already been transformed as Venkman requires. Binaries can, for example, come with Typed Assembly Language (TAL) annotations [27] which can help the verifier efficiently prove that the native code conforms to Venkman's requirements. Since the compiler that created the binary must have transformed it, it can easily insert the TAL annotations required for verification. A verifier like RockSalt [26] could be modified to perform the verification that Venkman requires.

The second program format is LLVM Bitcode [4, 22]. This format represents programs in a virtual instruction set that makes program analysis and transformation efficient and accurate. The LLVM virtual instruction set [22] organizes a program as a set of functions; each function has an explicit CFG, alleviating the need to reconstruct the CFG from binary code. Instructions in LLVM virtual instruction set are in Static Single Assignment (SSA) form [8], allowing efficient SSA-based algorithms to be employed to analyze code. An extended version of the LLVM virtual instruction set can encode a complete OS kernel completely within the LLVM instruction set [7], so both application code and OS kernel code can be shipped in LLVM Bitcode format.

Once Venkman generates native code for an LLVM Bitcode executable, it passes the native code (annotated with CFG information) through an optional set of transformations that add instructions to mitigate Spectre attacks e.g., fence instructions as Intel recommends [16]. The code is then passed through the alignment transformations (described in Sections 4.3 and 4.4) that prevent the program from poisoning BTB and RSB entries. Once transformed, the verifier checks that the BTB and RSB defenses have been applied correctly before allowing the code to execute. By reusing the binary verifier to verify the native code it generates, Venkman removes its compiler transformations and native code generator from its Trusted Computing Base (TCB).

On a system running Venkman, the OS kernel and dynamic binary loader must already have been transformed with Venkman. Additionally, the OS kernel must ensure that programs do not modify or extend their code segments without the binary verifier first verifying the new code. This can be accomplished by modifying the `mmap()` and `mprotect()` system calls in the OS kernel so that they verify code within a page before making the page executable. Systems such as SecVisor [30] can ensure that all kernel code has been verified before it is loaded.

4.2 Virtual Address Space Layout

Venkman divides the virtual address space as Figure 2 depicts. Venkman places the application code and data in the lower portion of the virtual address space and kernel code and data in the upper portion of the virtual address space; this arrangement is used by many current operating systems, including Linux [5] and FreeBSD [25].

All application code must be located within the lower contiguous portion of the virtual address space denoted as *code segment* in Figure 2. Unlike existing systems, this code segment includes code loaded by the dynamic linker. Likewise, all kernel code, including the code for dynamically-loaded kernel modules, must be loaded within the region reserved for kernel code shown in Figure 2. Venkman must ensure that all entries in the BTB and RSB are addresses within the application code segment or the kernel code segment. By requiring that all code be loaded within the application or kernel code segment, and by strategically selecting the placement and size of these code segments, simple bit-masking of control data can easily ensure that all computed branches target an address in the application code segment (for application code) or the kernel code segment (for kernel code).

We evenly split the code area and the data area (Data Segment, Heap, and Stack in Figure 2) in the user space; with this split, Venkman can identify to which area an address belongs by checking the highest bit (which is the x'th bit in Figure 2). This helps us implement SFI [35] more easily. Section 5 explains this choice in more detail.

Our implementation, described in Section 5, reserves 32 KB of virtual address space at the beginning and at the end of the code segment; Figure 2 denotes these reserved areas with dark gray boxes. This platform-dependent change allows Venkman to enforce SFI with lower overhead on a POWER machine. Section 5 explains the reasons in detail.

4.3 Code Alignment

Venkman transforms code so that each basic block within the program has a power-of-two size and is aligned at an address divisible by the same power-of-two. For example, Venkman can divide a program's basic blocks into bundles of 32 bytes (8 instructions per bundle on a POWER machine) with each bundle aligned on a 32-byte boundary as Figure 3 shows. Basic blocks larger than the required size are broken into smaller basic blocks, and basic blocks smaller than the required size are padded with NOPs until they are the required size. This simple transformation creates an invariant for all targets of control flow transfers: each address to which a branch or call instruction can jump is aligned on a specific boundary (a 32-byte boundary in our example). Venkman ensures that all entries in the BTB and RSB fulfill this invariant with static and dynamic checks; Section 4.4 explains how.

When transforming basic blocks to have the correct size and alignment, Venkman must enforce several restrictions. First, any instruction needing protection from a Spectre attack must appear in the same basic block as the instructions that protect it. For example, if a defense against Spectre inserts a load fence before a load instruction, the load fence and the load must occur within the same bundle. Second, targets of control flow transfers must be at the start of a basic block. The alignment of basic blocks ensures that the targets of branch and call instructions are the first address of a bundle. However, return addresses require additional processing: call instructions must always occur at the end of a bundle so that the return address is the beginning of the next bundle in memory.

4.4 Control Flow Instrumentation

With all basic blocks properly sized and aligned, Venkman must enforce the following two invariants on entries in the BTB and RSB:

1. **Code Segment**: All entries in the BTB and RSB must be within the application code segment or kernel code segment. If the processor checks page permissions late in the pipeline or uses a unified TLB, failure to ensure this invariant may cause the processor to speculatively execute data as code.

2. **Alignment**: All entries in the BTB and RSB must be aligned to the first address of a basic block (in our running example, a 32-byte boundary bundle). This prevents the program from bypassing instructions such as fences during speculative execution.

Figure 2: Venkman Virtual Address Space Layout

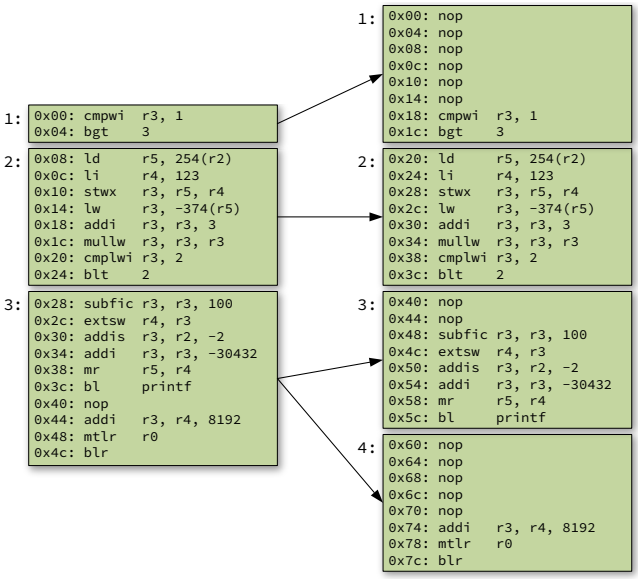

Figure 3: Example of Venkman Code Alignment with 32-Byte Bundles

Furthermore, Venkman must enforce these invariants even if the program has memory safety errors.

To do this, Venkman must ensure that all branches, jumps, and calls target only the first address of an aligned basic block at run-time. Venkman must enforce this requirement on direct jumps and calls, indirect jumps and calls, and return addresses. The processor may use the BTB on both direct and indirect jumps to determine the next address from which to fetch instructions, and it will similarly use the RSB for return instructions [31].

Direct control flow transfers are correct by construction: the target address is always the beginning of a specific basic block. Since Venkman aligns the basic block, the target of direct jumps to a basic block are aligned by construction.

For indirect control flow transfers such as calls through registers and returns, Venkman must insert bit-masking instructions before the indirect jump, indirect call, or return instruction that align the target of the branch to the beginning of a bundle. For example, before the return instruction `blr` in the last bundle in Figure 3, Venkman inserts instrumentation instructions to ensure that the link register (which contains the return address) always point to the beginning of a bundle. Additionally, the bit-masking must ensure that the target address is within the program's code segment. By placing the code segment strategically in memory, simple bit-masking of the higher-order bits suffices.

On processors supporting target operands in memory (e.g., `ret` instructions on x86 [14]), Venkman must transform the code so that the target address is first loaded into a register, bit-masked as described above, and then used in an indirect branch instruction that takes its argument in a register. Otherwise, one thread could corrupt the address in memory and change its alignment after the bit-masking has occurred but before the indirect jump uses the target in memory.

The above changes ensure that all values in the BTB are aligned to a basic block entry point within the code segment. For the RSB, since call instructions are placed at the end of a basic block, the return address is constructed to be at the beginning of the next aligned basic block. This ensures that all RSB entries are the address of an aligned basic block.

4.5 Speculative Stores to the Code Segment

SFI [35] can be used to prevent an application from speculatively reading and writing memory regions to which it does not have access [9]. On processors that can forward the results of speculative stores to instruction fetches (as Section 2.3 describes), we can use Spectre-resistant SFI [9] to ensure that the instructions that Venkman adds do not get speculatively overwritten by speculative store instructions. This SFI instrumentation inserts instructions before each store instruction to ensure that the address used in the store instruction is outside of the application and kernel code segments.

4.6 Venkman with Other Defenses

Venkman provides a framework into which other defenses can be integrated. For example, existing Spectre attacks [20] leak secrets through the data cache by using a load on an address that is computed from the secret data. Venkman can ensure that all instructions retire before each load to prevent leaks through the data cache that do not exist within the program's non-speculative control flow. For each bundle that has at least one load instruction, Venkman can insert a barrier instruction, such as x86's `lfence` [14] or POWER's `eieio` [13], at the beginning of the bundle to guarantee that all the instructions executed prior to loads in the bundle are executed and retire first. With Venkman, this approach completely thwarts Spectre Variant-1, Variant-2, and other variants that poison the RSB. Venkman can also use Spectre-resistant SFI techniques [9] to prevent load instructions from accessing a specific region of

```
1 clrrdi r1, r1, 5
2 clrldi r1, r1, 19
3 mtlr   r1
4 blr
```

Listing 3: CFI Instrumentation

```
1 rldicr r1, r1, 32, 63
2 ori    r1, r1, 0x2000
3 rldicl r1, r1, 32, 18
4 std    r3, 8(r1)
```

Listing 4: SFI Instrumentation on Stores

```
1 clrldi r1, r1, 1
2 ld     r3, 8(r1)
```

Listing 5: SFI Instrumentation on Loads

the virtual address space; this creates a speculation-resistant sandbox for the application. Section 5 explains in more detail how we integrate other defenses into Venkman.

5 Implementation

5.1 Base Venkman Implementation

We implemented Venkman by extending the code generator in the LLVM [22] 4.0 compiler with two new MachineFunction Passes. To create a prototype of Venkman as quickly as possible, we opted to build our prototype for the POWER architecture first. POWER remains an important and competent architecture for high performance computing, cloud computing, and enterprise-level workloads [12]. Our IBM POWER8 machine utilizes speculative out-of-order execution and is therefore vulnerable to Spectre attacks. Beneficially, POWER has fixed-sized instructions which hastened development of our prototype [13]. We leave x86 and ARM implementations of Venkman for future work.

The first MachineFunctionPass inserts the code that bit-masks indirect branch targets as Section 4.4 describes. On POWER, indirect branch targets are stored either in the link register or the counter register [13]. The first MachineFunctionPass searches for instructions that move values into the link register and counter register and inserts instructions to clear the lower-order 5 bits (because we choose to use 32-byte bundle). Since the counter register is also used for purposes other than indirect branch targets, our MachineFunctionPass scans for the next use of the counter register and only bit-masks the value if the next use of the counter register within the same function is a branch instruction.

Listing 3 shows an example for a return address (stored in register `r1` that is moved into the link register with the `mtlr` instruction. The `clrrdi` instruction clears the lower 5 bits of the code pointer stored in `r1` so that it is aligned on a bundle boundary. The `clrldi` instruction clears the upper 19 bits of the code pointer to ensure that it is located within the lower 32 TB of the virtual address space where we put the code segment, as Section 5.3 describes.

The second MachineFunctionPass breaks up basic blocks and aligns them as Section 4.3 describes. It ensures that all indirect control flow instructions and the bit-masking inserted by the previous MachineFunctionPass remain within the same bundle. Our implementation uses 32-byte bundle, so they are aligned on 32-byte boundaries.

5.2 Venkman with Fences

Our prototype has a command-line option to insert a barrier instruction (`eieio`) in each bundle containing at least one load instruction. The `eieio` instruction enforces ordering of memory accesses issued prior to the barrier with memory accesses issued after the barrier [13]. By placing an `eieio` in each bundle that contains one or more load instructions, we can mitigate Spectre attacks completely by ensuring that all checks on the pointer used in load instructions retire before the load instruction commences. We therefore use our `eieio` option in experiments to evaluate the performance of Venkman when it is used to ensure that `eieio` instructions are executed before load instructions to mitigate Spectre attacks.

5.3 Venkman with SFI for Stores and Loads

We also implement SFI for Venkman. Our prototype provides SFI on stores and (optionally) SFI on loads. SFI on stores prevents store-bypass attacks [18] from speculatively modifying the code segment. SFI on both stores and loads can provide Spectre-resistant isolation for software plugins. We therefore implemented sandboxing using Spectre-resistant SFI [9] for the POWER architecture.

In order to achieve an efficient SFI implementation, we divide the whole virtual address space for user programs (`0` to `0x3fffffffffff`) equally into two regions, one (`0` to `0x1fffffffffff`) for the code segment and the other (`0x200000000000` to `0x3fffffffffff`) for data segments (which includes global variables, thread stacks, and the heap). As Section 4.2 describes, in this way we could easily identify to which region a pointer is pointing by examining the highest bit used in addressing user space memory (which is bit 45). Furthermore, since POWER's D-Form store instructions [13] allow the target address to be the sum of a register operand's contents and a 16-bit signed immediate, Venkman must reserve a 32-KB hole between `0` and the beginning of the code segment and another 32-KB hole between the end of the code segment and the beginning of the first data segment. In this way, neither a store with a register operand pointing to the beginning of the data segment and a negative immediate, nor a store with a register operand pointing to the end

of virtual address space and a positive immediate, can speculatively overwrite the code segment. This narrows down the code segment region to `0x8000` to `0x1fffffff7fff`. With this arrangement, before every store instruction, our prototype inserts code that clears bit 45 of the pointer register content used by the store to ensure that it points outside of the virtual address region reserved for the code segment.

To evaluate the overhead of SFI on loads for programs that want to employ sandboxing, we implemented an optional feature to add SFI instrumentation on loads. Since there is no special memory region needing protection from speculative loads in our testing programs, we choose to instrument every load to ensure that each reads user-space (as opposed to kernel-space) memory; this is done by inserting code that clears the most significant bit of the pointer. This instrumentation is mainly for the purpose of mimicking the code size and performance overhead of incorporating SFI on loads into our defenses; it also happens to prevent Meltdown [23] attacks on the OS kernel.

Listing 4 shows the instrumented code for a D-Form store. The `std` instruction takes the sum of the contents of register `r1` and an immediate 8 as the target address, and it stores the contents of register `r3` as a double word into the target address. Our instrumentation rotates the contents of the pointer by 32 bits, sets bit 13 (bit 45 of the actual pointer), rotates the pointer back, and clears bits 46 to 63. This ensures that the pointer always points to memory in the data segments and prevents speculative writes to the code segment.

Listing 5 shows the instrumentation for a D-Form load. The `clrldi` instruction clears the most significant bit of the contents of register `r1`, which is used in the `ld` instruction as a pointer from which to load. The instrumentation prevents kernel space memory from being speculatively read by a user space application.

On POWER, load and store instructions have another form called X-Form [13]. X-Form loads and stores compute the target address by adding the contents of two registers (a base and an index register). For such loads and stores, our prototype adds code to add the base and index register (placing the result in the base register), bit-mask the result, and use the result in a D-Form instruction that replaces the original X-Form instruction. A subsequent subtract instruction restores the original contents of the base register.

Our prototype does not use the POWER predicated move instruction, `isel`, because our MachineFunction Passes execute after register allocation. Consequently, use of compare and `isel` instructions can overwrite the condition registers. Our current instrumentation overwrites no condition registers and can be safely inserted anywhere before a store or a load.

5.4 Limitations

Our current implementation has two limitations. First, we didn't instrument the OS kernel with Venkman. Second, we

```
1 void benign_function(unsigned long x) {
2   return;
3 }
4
5 void victim_function(unsigned long x) {
6   if (x < array1_size)
7     y = array2[array1[x] * 256];
8 }
9
10 void dispatcher(func_t func, unsigned long x) {
11   (*func)(x);
12 }
13
14 int main() {
15   /* Attacker: train BTB */
16   dispatcher(&victim_function, in_bounds_x);
17
18   /* Victim: run */
19   dispatcher(&benign_function, malicious_x);
20
21   /* Attacker: probe cache accesses */
22   probe_cache();
23 }
```

Listing 6: Core Component of Attack Prototype

didn't instrument most of the C/C++ standard library code; the GNU C Library (glibc) and C++ Library (libstdc++) are tightly bound with the GNU C/C++ compiler (gcc and g++) and thus cannot be compiled by LLVM/Clang entirely. Instead, we opted to compile most of glibc and libstdc++ code by gcc and g++ with alignment options[1] and instrument only `exit.c`, `msort.c`, `elf-init.c`, and `libc-start.c` in glibc by Venkman separately. These files contain initialization routines and library functions that call functions in application code. Since Venkman inserts bit-masking instructions that clear the last few bits of indirect branch targets, an instrumented callee in application code might return to an incorrect address in its unaligned caller in library code. For a similar reason, we also didn't bit-mask indirect branch targets in some application functions including `main` and all global constructors and destructors of C++ programs. These functions are called by library code that is not instrumented by Venkman.

6 Security Evaluation

To evaluate Venkman's effectiveness, we implemented a proof-of-concept Spectre Variant-2 attack on an IBM POWER8 machine and demonstrated that Venkman with fences successfully prevents the attack.

We first constructed a working proof-of-concept Spectre Variant-2 attack on POWER. To construct this attack, we took the original Spectre Variant-1 attack code in C [20], ported it to POWER, modified it to perform both Spectre Variant-1 and Variant-2 attacks, and reused the cache side channel code that

[1] `-falign-functions=32`, `-falign-jumps=32`, `-falign-labels=32`, and `-falign-loops=32`.

verifies the data leakage. Listing 6 shows the core component of our attack code, abstracting away other technical details to improve clarity. In our proof-of-concept attack, the attacker and the victim share the same virtual address space, which is a common scenario (e.g., malicious JavaScript code attacking a web browser). First, the attacker calls `dispatcher` with a pointer to `victim_function` and an in-bounds input, mistraining the processor's branch predictor and BTB to trick the processor into believing that the function pointer call at Line 11 always branches to `victim_function` and that the conditional branch at Line 6 is always taken. After training, the victim then calls `dispatcher` with a pointer to `benign_function` and an attacker-supplied malicious input. Since the processor has been mistrained, speculative execution will jump to `victim_function`, take the conditional branch, and perform memory accesses that brings contents of `array2` into the cache. Even though the processor eventually squashes the speculative reads, the attacker infers the secret data via a cache side channel in `probe_cache`.

The above attack works on POWER when we compile the attack code using Clang without Venkman. When we compile the code with Venkman with fences, the attack no longer works, showing that Venkman effectively prevents the Spectre Variant-2 attack.

7 Space and Performance Evaluation

7.1 Experimental Setup

We evaluated Venkman by compiling and running the SPEC CPU 2017 benchmark suite and several real-world applications (Nginx, GnuPG, and ClamAV) on a POWER machine. We used a 64-bit model 2.1 (pvr 004b 0201) 20-core IBM POWER8 machine running at 4.1 GHz. The machine, running CentOS 7 Linux with kernel version 3.10.0, has 8 threads per core and 64 GB of RAM. We compiled the SPEC CPU 2017 benchmarks and applications with both the original LLVM 4.0.1 (as the baseline) and Venkman. We statically linked each benchmark program and application; as SFI on stores requires code and data be within separate virtual address regions, static linking gives us the easiest control of where code and data sections are loaded. In our experiments, we constantly used 32 bytes as the bundle size which was determined experimentally to be the best choice on POWER ISA.

For SPEC, we used LLVM's *lit* tool to run the SPEC benchmarks. It measures both the execution time and code size of the benchmark programs. Of all the SPEC benchmark programs that LLVM can compile (programs written in C or C++ or both), only *526.blender_r* does not build on our POWER machine because of an incompatible C++ header file. We therefore exclude it from our experiments.

We begin by evaluating Venkman's performance on SPEC when Venkman uses fences to mitigate Spectre attacks. We then show Venkman's performance on SPEC when Venkman uses Spectre-resistant SFI. Finally, we evaluate Venkman's overhead on real-world applications.

Table 1: SPEC CPU 2017 Baseline Code Size

Benchmark	Size (Byte)	Benchmark	Size (Byte)
500.perlbench_r	2,747,020	557.xz_r	681,420
502.gcc_r	9,579,404	600.perlbench_s	2,747,020
505.mcf_r	551,852	602.gcc_s	9,579,404
508.namd_r	1,979,680	605.mcf_s	551,852
510.parest_r	11,339,744	619.lbm_s	553,900
511.povray_r	1,695,168	620.omnetpp_s	3,664,352
519.lbm_r	553,804	623.xalancbmk_s	6,124,896
520.omnetpp_r	3,664,352	625.x264_s	1,102,188
523.xalancbmk_r	6,124,896	631.deepsjeng_s	609,964
525.x264_r	1,102,188	638.imagick_s	2,686,828
531.deepsjeng_r	609,964	641.leela_s	1,354,144
538.imagick_r	2,686,828	644.nab_s	811,660
541.leela_r	1,354,144	657.xz_s	681,420
544.nab_r	811,660		

7.2 Complete Spectre Defense with Fences

To show how Venkman performs with existing Spectre defenses, we use Venkman plus SFI on stores and `eieio` instructions inserted before the first load in each bundle as a complete system that defends against Spectre Variant-1, Variant-2, and Spectre variants that poison the RSB. To study the sources of overhead, we break down the overhead into Venkman with just the creation and alignment of bundles (dubbed **Alignment**), Venkman with the alignment overhead and the overhead of bit-masking control data (dubbed **CFI**), Venkman with alignment, control data bit-masking, and SFI on stores (dubbed **SFI-Store**), and the full protection with `eieio` instructions (dubbed **Fence**). We evaluate this system and analyze Venkman's impact on the code size and performance of SPEC benchmark programs by comparing it with the baseline (i.e., the original program compiled with the same compiler with no Venkman transformations enabled).

Code Size We got the code segment sizes of SPEC benchmarks from the *lit* tool running the benchmarks. The code size information is actually measured using the *llvm-size* tool by reading the ELF binaries and reporting size of the text segment. Since we compiled the benchmark programs statically, the text segment of a program contains all the library code that the program uses. Table 1 shows the text segment size of the SPEC benchmarks compiled by the original Clang, which is our baseline. Figure 4 shows the text segment size of the SPEC benchmarks; the results are normalized to the baseline. It reports the results for our complete system as well as the breakdown of the code size overhead, showing how much the overhead is coming from each of the above defenses.

When all the defenses are deployed, the code size overhead ranges from 1.37× to 2.66× with a geometric mean of 1.93×. As Figure 4 shows, a significant component of the

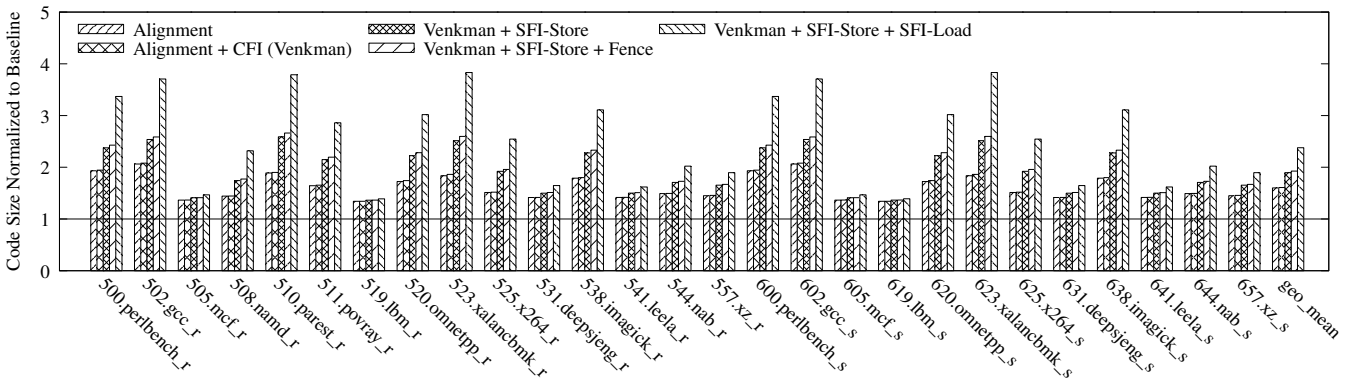

Figure 4: Code Size Overhead on SPEC CPU 2017

Table 2: SPEC CPU 2017 Baseline Execution Time

Benchmark	Time (s)	Benchmark	Time (s)
500.perlbench_r	48.6	557.xz_r	54.7
502.gcc_r	68.4	600.perlbench_s	48.0
505.mcf_r	64.0	602.gcc_s	69.3
508.namd_r	52.1	605.mcf_s	65.0
510.parest_r	68.8	619.lbm_s	237.3
511.povray_r	10.1	620.omnetpp_s	103.6
519.lbm_r	30.1	623.xalancbmk_s	115.3
520.omnetpp_r	99.2	625.x264_s	74.1
523.xalancbmk_r	114.9	631.deepsjeng_s	121.1
525.x264_r	73.8	638.imagick_s	84.0
531.deepsjeng_r	99.6	641.leela_s	132.3
538.imagick_r	83.7	644.nab_s	241.7
541.leela_r	133.1	657.xz_s	50.2
544.nab_r	241.7		

space overhead comes from Alignment (from 1.34× to 2.08× with a geometric mean of 1.61×). SFI-Store also incurs a non-negligible portion of the overhead (from 2.16% to 68.8% with a geometric mean of 30.1%) since we instrumented every store using 3 to 7 instructions. The rest of the defenses contributes minor overhead: CFI overhead is from 0.017% to 2.6% with a geometric mean of 0.83%, and Fence overhead is from 0.18% to 8.3% with a geometric mean of 3.5%.

Performance Table 2 shows the baseline execution time of the SPEC benchmarks. Figure 5 shows the normalized overhead on SPEC benchmarks. The results include the overhead of our complete system, while also providing a breakdown of the performance overhead. The execution time is measured by the LLVM Test Suite timing tool called *timeit*, which measures the total wall time of a given command by recording the time difference of two `gettimeofday()` calls. We ran each benchmark ten times and report the geometric mean.

As Figure 5 shows, the overhead with all defenses enabled ranges from 1.87× to 7.72× with a geometric mean of 3.66×. The performance breakdown shows that, for most of benchmark programs, Fence and SFI-Store degrade the performance most; the overhead is from 1.79× to 7.67× with a geometric mean of 3.29× and from 1.04× to 1.61× with a geometric mean of 1.28×, respectively. In contrast, the Venkman defenses (i.e., Alignment and CFI) only have moderate overhead: Alignment incurs overhead from 0.95× to 1.21× with a geometric mean of 1.08×, while CFI incurs overhead from 0.99× to 1.08× with a geometric mean of 1.01×.

Note that we need SFI-Store to protect the code segment from being speculatively overwritten only if the processor's store buffer forwards its content to the instruction fetch unit. If one needs Venkman to defend against Spectre attacks and is sure that the processor on which Venkman is deployed does not forward data in the store buffer to the instruction fetch unit (which is typically the case if the processor disallows self-modifying code), then he/she can regain the performance lost of using SFI-Store by simply disabling SFI-Store.

7.3 Spectre-Resistant Sandboxing

In addition to the evaluation of Venkman with Fence, We also did experiments on a Spectre-resistant sandboxing system in which Alignment, CFI, and SFI-Store are still deployed and Fence is replaced with SFI on loads (dubbed **SFI-Load**), as Section 5.3 describes.

Figure 4 reports the code size overhead of the sandboxing SPEC normalized to the same baseline as in Section 7.2. As Figure 4 shows, the overall code size overhead of the sandboxing ranges from 1.39× to 3.83× with a geometric mean of 2.38×. SFI-Load's contribution alone is from 1.03× to 2.32× with a geometric mean of 1.52×. Compared with using Fence, the sandboxing approach occupies more storage space because every load is instrumented with 1 to 3 instructions, whereas only a single `eieio` instruction is inserted before all the loads in a bundle using Fence.

Figure 5 reports the performance overhead of the sandboxing normalized to the same baseline as in Section 7.2. As Figure 5 shows, the sandboxing approach slows down the performance by 1.12× to 2.51× with a geometric mean of 1.74×. Separating the overhead apart, SFI-Load causes a slowdown of 7.1% to 68.6% with a geometric mean of 38.8%. Compared

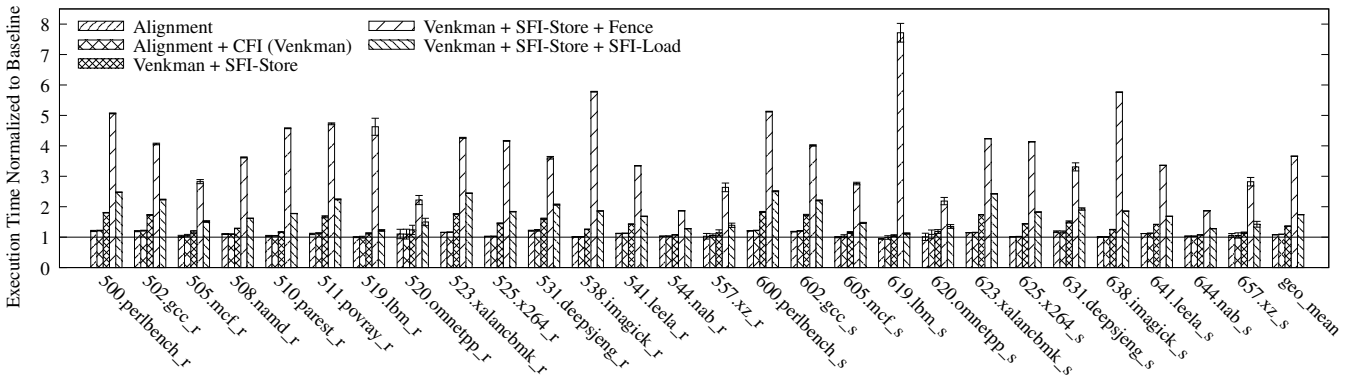

Figure 5: Performance Overhead on SPEC CPU 2017

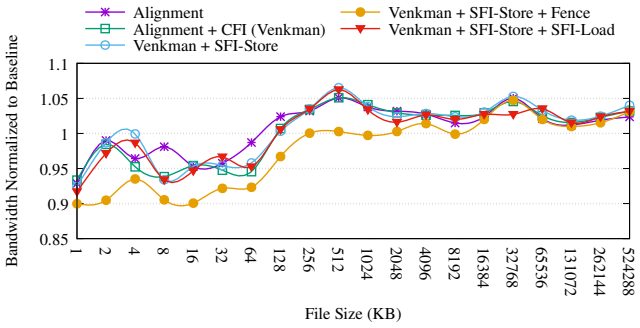

Figure 6: Throughput Overhead on Nginx

with Fence, SFI-Load boosts the performance of all the benchmarks, confirming the conclusion that using hardware fences are much more expensive than creating data dependencies between protecting and protected instructions [9].

7.4 Application Evaluation

In addition to evaluating standardized benchmarks like SPEC, we also evaluated Venkman on selected real-world applications including Nginx, GnuPG, and ClamAV. We chose these applications because each one represents a different type of workload (specifically, I/O intensive, CPU intensive, and file system intensive). These programs may also fall victim to Spectre attacks [20] as they manipulate sensitive information.

Nginx We evaluated Nginx 1.15.8 to show how Venkman performs on server applications. Nginx [32] is an open source web server that is designed for high performance and is widely deployed. We compiled Nginx with the same configurations as SPEC: baseline, Venkman with SFI-Store and Fence, Venkman with SFI-Store and SFI-Load, and configurations that enable us to separate the overhead.

We obtained the code segment size of Nginx using the `objdump` tool. Table 3 shows both the baseline code size of Nginx as well as the code size overhead induced by Venkman. It also reports the breakdown of the overhead. Table 3 shows that the overall space overhead is 2.04× for Fence and 2.68× for SFI-Load. Of all the defenses, SFI-Load, Alignment, and SFI-Store contribute most of the space overhead, which are 67%, 61%, and 39%, respectively. The rest of the defenses add little space overhead: CFI induces 1%; Fence induces 3%. This roughly conforms with SPEC's code size overhead shown in Sections 7.2 and 7.3.

To study Venkman's impact on Nginx performance, we ran Nginx compiled by Venkman with 1 worker process delivering static files ranging in size from 1 KB to 512 MB. The files were generated by reading bytes from the `/dev/urandom` pseudorandom number generator. We used ApacheBench (`ab`) [1] as the client running on the same machine as Nginx to measure Nginx's performance. Since our test machine has 160 logical cores, the client and server can run on different logical cores without stealing CPU time from each other when executed on the same machine. For each configuration and for each file size, we ran `ab` for 50 iterations, each iteration lasting 10 seconds in which `ab` continuously fetched the same file until timeout. We then collected performance numbers from the ApacheBench output and report the geometric means. Table 4 shows the baseline average file transfer throughput of Nginx, and Figure 6 shows Venkman's overhead on Nginx throughput normalized to baseline. Due to space, we omit numbers on Nginx latency; they are usually the reciprocal of the throughput. Standard deviations are also not shown; they are as much as 4.4%. As Figure 6 shows, Venkman reduces Nginx's throughput by at most 10.0% when using Fence and 8.3% when using SFI-Load. Venkman incurs higher overhead when transferring small files. As file size increases, the overhead becomes negligible. Overall, Venkman's impact on Nginx performance is small but not as clear as on SPEC due to high standard deviations.

GnuPG We evaluated the code size and performance impact of Venkman on GnuPG 1.4.23. GnuPG [19] is an open source cryptography program that provides encryption and signing services. We compiled GnuPG with the same config-

Table 3: Application Code Size

Application	Baseline (MB)	Alignment	Alignment + CFI (Venkman)	Venkman + SFI-Store	Venkman + SFI-Store + Fence	Venkman + SFI-Store + SFI-Load
Nginx	1.20	1.61×	1.62×	2.01×	2.04×	2.68×
GnuPG	1.91	1.62×	1.63×	2.20×	2.27×	3.53×
ClamAV	1.14	1.71×	1.71×	2.02×	2.05×	2.52×

Table 4: Application Baseline Performance Results

File Size (KB)	Nginx (MB/s)	GnuPG (ms)	ClamAV (ms)	File Size (KB)	Nginx (MB/s)	GnuPG (ms)	ClamAV (ms)
1	14.3	8.2	73.4	1,024	2247.2	178.0	144.9
2	25.7	8.3	73.4	2,048	2565.1	344.8	217.4
4	48.3	8.4	73.4	4,096	2701.3	684.1	145.8
8	94.4	8.9	73.5	8,192	2655.2	1357.9	651.4
16	185.5	9.8	73.6	16,384	2473.7	2717.5	550.2
32	344.1	12.1	73.8	32,768	2326.2	5414.0	1026.9
64	620.4	17.3	74.4	65,536	2267.6	-	1980.6
128	977.8	27.9	75.6	131,072	2319.2	-	3889.2
256	1438.0	49.0	77.8	262,144	2316.6	-	7701.2
512	1874.6	91.3	82.5	524,288	2319.7	-	9335.1

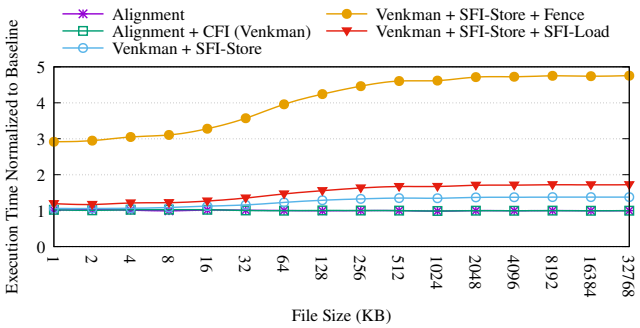

Figure 7: Performance Overhead on GnuPG Encryption

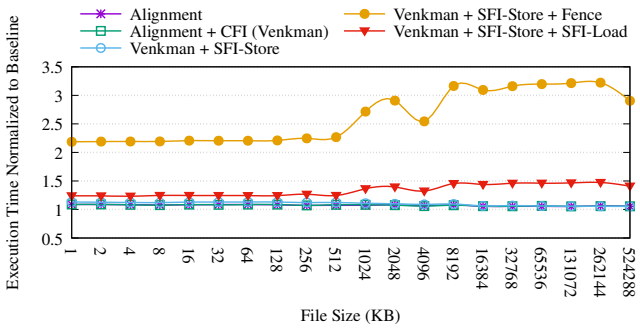

Figure 8: Performance Overhead on ClamAV

urations as we did for SPEC and Nginx.

Table 3 provides the code size information of GnuPG. This information was also collected by the `objdump` tool. As Table 3 shows, when SFI-Load is in place, it contributes to a significant portion of GnuPG's code size overhead, resulting in a 2.33× larger code segment. Alignment and SFI-Store also cause a non-negligible increase of GnuPG code size, making it larger by 62% and 57%, respectively. In contrast, CFI and Fence incur minor space overhead on GnuPG.

We ran GnuPG compiled by Venkman to encrypt, decrypt, sign, and verify the signatures of files from 1 KB to 32 MB in size. The files are a subset of the files we used in evaluating Nginx. For each configuration and for each file size, we ran GnuPG 10 times and recorded the execution time of each iteration. The geometric mean over all 10 iterations is calculated and reported. Due to limited space, we only show the GnuPG encryption results which have the highest performance overhead among all four functionalities we tested; the other three exhibited similar but lower overhead. Table 4 lists the baseline execution time of GnuPG encryption, and Figure 7 shows the normalized execution time of GnuPG compiled by Venkman. The standard deviations are within acceptable ranges and thus not shown. As Figure 7 shows, Fence hurts performance most, increasing the execution time to 2.86× to 4.38× with a geometric mean of 3.73×. If we adopt the sandboxing approach instead, SFI-Load only gives us a 34.4% performance degradation at most. SFI-Store also reports a slowdown of at most 37.9%, while Alignment and CFI incur a minor overhead of at most 2.6% and 3.1%, respectively.

ClamAV ClamAV [2] is an open source antivirus program that is typically used for detecting viruses and malware on mail servers. We compiled ClamAV 0.92 from the LLVM Test Suite using Venkman with the same configurations as we did for SPEC, Nginx, and GnuPG, and we report Venkman's overhead on ClamAV's command-line scanning tool `clamscan`.

Again, Table 3 shows the code size measurements of ClamAV obtained from `objdump`. When complete defenses against Spectre attacks are deployed, Alignment, SFI-Load, and SFI-Store are the three major sources of ClamAV's code size overhead (71%, 50%, and 31%). Fence and CFI, on the other hand, only expand the code segment slightly (3% and

less than 1%, respectively).

To test ClamAV, we used `clamscan` to scan the files used in the Nginx experiment for malware. We ran `clamscan` 10 times for each configuration and for each file size and report the geometric means. Our ClamAV uses a virus database from the LLVM Test Suite; while it is old, it works for our performance evaluation. Table 4 reports the baseline execution time of ClamAV scanning various-sized files; Figure 8 shows the performance overhead incurred by Venkman. The standard deviations, again, are not shown; they are acceptable with respect to each configuration. Figure 8 shows that Fence reduces performance most, by 2.05× to 3.17× with a geometric mean of 2.47×. In comparison, SFI-Load's overhead (at most 41.8%) is much cheaper than Fence. SFI-Store on ClamAV causes unusually less overhead (at most 5.0%), and Venkman (Alignment and CFI) adds little to minor overhead.

8 Related Work

There are several software-only approaches that mitigate Spectre Variant-1 [20]. Intel recommends inserting load fences before load instructions to ensure that branch instructions retire before loads are executed [16]. Carruth proposed pointer hardening [6] which creates a data dependence between branch conditions and the pointer used in loads. The compiler inserts code before loads that will mask the pointer value to zero if the branch was mispredicted and leave the pointer unaltered otherwise. Dong et al. [9] developed SFI techniques that work against Spectre Variant-1 attacks. All of these approaches are vulnerable to Spectre Variant-2 attacks since poisoning the BTB [20] or RSB [21, 24] permits an attacker to jump over the instructions that protect loads. Venkman can protect these approaches from Spectre Variant-2 attacks if it places instructions protecting each load in the same bundle as the load itself. Venkman already does this for load fences [16] and SFI [9]. Pointer hardening may require a large bundle size in order to ensure that the bit-masking instructions and the branch condition are computed within the same bundle; we leave the integration of pointer hardening with Venkman to future work.

Spectre Variant-2 poisons the BTB [20], and other variants poison the RSB [21, 24]. An early mitigation for Spectre Variant-2, the Retpoline [33] transformation, changes indirect call and jump instructions into return instructions. A retpoline explicitly moves the target address of an indirect function call or jump to the return address on the stack, causing speculation to predict the target address with the RSB instead of with the BTB. The retpoline sets up the RSB so that the processor speculatively executes a busy loop until the target of the return is read from the stack. As Section 2.2 explains, the processor uses the BTB to predict the target addresses of direct branches, making direct branches susceptible to BTB poisoning. In contrast, Venkman mitigates exploitation of the BTB and RSB by forcing all control flow targets to be aligned to a bundle's start address.

Intel processors provide three hardware mitigations for BTB poisoning [16]. The first prevents code running at a lower privilege level from affecting branch prediction for code running at higher privilege levels. Unfortunately, several Spectre attacks [20] target victims running at the same hardware privilege level as the attacker. In contrast, Venkman protects software running at all hardware privileges levels. The second Intel processor defense prevents sharing of BTB entries between code running on different logical processors. This defense fails to mitigate attacks by programs executing on the same logical processor as the victim. Venkman, on the other hand, protects software regardless of which logical processors execute the attacker and victim code. The third defense adds a BTB training barrier command: branch predictions following the barrier do not use BTB entries that were created on the same logical processor prior to the barrier. This approach prevents BTB poisoning but reduces performance as valid BTB entries prior to the barrier are lost. Venkman ensures that all BTB and RSB entries are properly aligned on a bundle boundary, allowing safe sharing of BTB entries across programs. Only programs using fences or SFI instructions incur significant performance loss.

Venkman employs transformations similar to those of Google's Portable Native Client (PNaCl) [29]. Like PNaCl, Venkman must break code into individual bundles, align bundles on a constant alignment, and align control data (such as function pointers) before using them in a branch. However, since Venkman regulates control flow for *speculatively* executed instructions, it must also place call instructions at the end of bundles to ensure that return addresses are aligned to the start of a bundle. It must also ensure that protecting instructions (e.g., fences and SFI) and protected instructions (e.g., loads and stores) be located within the same bundle.

9 Conclusions and Future Work

In this paper, we presented and evaluated Venkman, a solution that thwarts Spectre attacks that poison the BTB and RSB. To the best of our knowledge, no existing defense completely mitigates poisoning of these structures. Our evaluation shows that Venkman increases code size by 1.94× on average. We also observe an average of 3.47× performance overhead.

Several directions exist for future work. First, we will improve Venkman's precision. Venkman currently ensures that speculative execution adheres to a very conservative CFG. More sophisticated code placement strategies might allow Venkman to restrict speculative branches to a set of specific targets. Second, we will investigate whether transformations like super-block construction [17] and if-conversion [17] can reduce the number of fences that Venkman inserts by creating more straight-line code. Finally, we will port Venkman to x86 and ARM.

References

[1] Apache HTTP server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html.

[2] ClamAV. https://www.clamav.net.

[3] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-Flow Integrity Principles, Implementations, and Applications. *ACM Transactions on Information Systems Security*, 13:4:1–4:40, November 2009.

[4] Vikram Adve, Chris Lattner, Michael Brukman, Anand Shukla, and Brian Gaeke. LLVA: A Low-Level Virtual Instruction Set Architecture. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-36, pages 205–216, San Diego, CA, 2003. IEEE Computer Society.

[5] Daniel P. Bovet and Marco Cesati. *Understanding the LINUX Kernel*. O'Reilly, Sebastopol, CA, 3rd edition, 2005.

[6] Chandler Carruth. Speculative load hardening: A Spectre variant #1 mitigation technique, 2018. Available at https://docs.google.com/document/d/1wwcfv3UV9ZnZVcGiGuoITT_61e_Ko3TmoCS3uXLcJR0/edit#heading=h.phdehs44eom6.

[7] John Criswell, Andrew Lenharth, Dinakar Dhurjati, and Vikram Adve. Secure Virtual Architecture: A Safe Execution Environment for Commodity Operating Systems. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles*, SOSP'07, pages 351–366, Stevenson, WA, 2007. ACM.

[8] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, pages 13(4):451–490, October 1991.

[9] Xiaowan Dong, Zhuojia Shen, John Criswell, Alan Cox, and Sandhya Dwarkadas. Spectres, Virtual Ghosts, and Hardware Support. In *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP'18, pages 5:1–5:9, Los Angeles, CA, 2018. ACM.

[10] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49, pages 40:1–40:13, Taipei, Taiwan, 2016. IEEE Press.

[11] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering*, 8(1):1–27, October 2016.

[12] Brian Hall, Peter Bergner, Alon Shalev Housfater, Madhusudanan Kandasamy, Tulio Magno, Alex Mericas, Steve Munroe, Maurício Oliveira, Bill Schmidt, Will Schmidt, et al. *Performance Optimization and Tuning Techniques for IBM Power Systems Processors Including IBM POWER8*. IBM Redbooks, 2nd edition, 2015.

[13] IBM. *Power ISA™ Version 2.07 B*, January 2018.

[14] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*, October 2016.

[15] Intel Corporation. Microcode revision guide, April 2018. https://newsroom.intel.com/microcode.

[16] Intel Corporation. *Speculative Execution Side Channel Mitigations*, May 2018. Document Number: 336996-003.

[17] Ken Kennedy and John R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 2002.

[18] Vladimir Kiriansky and Carl Waldspurger. Speculative buffer overflows: Attacks and defenses. *arXiv preprint arXiv:1807.03757*, July 2018.

[19] Werner Koch. GnuPG, 1999. https://www.gnupg.org.

[20] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *Proceedings of the 40th IEEE Symposium on Security and Privacy*, SP'19, San Francisco, CA, 2019. IEEE.

[21] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *Proceedings of the 12th USENIX Workshop on Offensive Technologies*, WOOT'18, Baltimore, MD, 2018. USENIX Association.

[22] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO'04, pages 75–86, Palo Alto, CA, 2004. IEEE Computer Society.

[23] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *Proceedings of the 27th USENIX Security Symposium*, Security'18, pages 973–990, Baltimore, MD, 2018. USENIX Association.

[24] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative execution using return stack buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS'18, pages 2109–2122, Toronto, Canada, 2018. ACM.

[25] Marshall Kirk McKusick, George V. Neville-Neil, and Robert N. M. Watson. *The Design and Implementation of the FreeBSD Operating System*. Addison-Wesley Professional, 2nd edition, 2014.

[26] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. RockSalt: Better, faster, stronger SFI for the x86. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI'12, pages 395–404, Beijing, China, 2012. ACM.

[27] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999.

[28] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology*, CT-RSA'06, pages 1–20, San Jose, CA, 2006. Springer-Verlag.

[29] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. Adapting software fault isolation to contemporary CPU architectures. In *Proceedings of the 19th USENIX Security Symposium*, Security'10, pages 1–11, Washington, DC, 2010. USENIX Association.

[30] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, SOSP'07, pages 335–350, Stevenson, WA, 2007. ACM.

[31] John Paul Shen and Mikko H. Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors*. Waveland Press Inc., Long Grove, IL, 1st edition, 2013.

[32] Igor Sysoev. Nginx, 2004. https://nginx.org.

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

[34] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Security Symposium*, Security'18, pages 991–1008, Baltimore, MD, 2018. USENIX Association.

[35] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, SOSP'93, pages 203–216, Asheville, NC, 1993. ACM.

[36] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *Proceedings of the 23rd USENIX Security Symposium*, Security'14, pages 719–732, San Diego, CA, 2014. USENIX Association.