POSTER: Restricting Control Flow During Speculative Execution

Zhuojia Shen
University of Rochester
zshen10@cs.rochester.edu

Jie Zhou
University of Rochester
jzhou41@cs.rochester.edu

Divya Ojha
University of Rochester
dojha@cs.rochester.edu

John Criswell
University of Rochester
criswell@cs.rochester.edu

ABSTRACT

Speculative execution is one of the key techniques that modern processors use to boost performance. However, recent research shows that speculative execution can be used to steal sensitive data. We present a software-based solution to mitigate Spectre attacks by restricting the control flow of speculatively-executed instructions.

CCS CONCEPTS

• **Security and privacy → Side-channel analysis and countermeasures**; **Systems security**;

KEYWORDS

speculative execution, side-channel defenses, Spectre attacks

ACM Reference Format:
Zhuojia Shen, Jie Zhou, Divya Ojha, and John Criswell. 2018. POSTER: Restricting Control Flow During Speculative Execution. In *2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18), October 15–19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3243734.3278522

1 INTRODUCTION

Modern high-performance processors support speculative execution and out-of-order execution. Speculative execution [11] is a technique in which the processor speculatively executes instructions prior to knowing that they are required in order to improve performance; if the processor later determines that the instructions should not have been executed, it discards the computation, rolling back all the architectural effects resulting from the speculatively-executed instructions. Out-of-order execution [11] is a performance improvement in which the processor executes instructions out of program order to maximize throughput. While these features are meant to improve processor performance, recent research (Meltdown [10] and Spectre [7]) has shown that attackers can leverage these optimizations to launch powerful side-channel attacks. Meltdown [10] exploits out-of-order execution and a security defect in which the processor performs the hardware MMU protection check late in the pipeline, using them to bypass hardware-enforced memory isolation. Spectre [7] tricks the processor into speculatively executing instructions that load a victim's secret data into registers and then leaks the data via a cache side channel. To date, at least four new variants of Meltdown and Spectre have been discovered [4–6, 8].

While there exist compiler-based transformations that defend against Meltdown [10] and Spectre [7] (such as Spectre-resistant software fault isolation (SFI) [3]), they typically cannot mitigate Branch Target Injection (i.e., Spectre Variant 2 [7]) which mistrains the processor's Branch Target Buffer (BTB) to hijack speculative control flow. To mistrain, or poison, the BTB, the attacker program repeatedly executes a set of branches that jump to a target address [7]. Since the BTB is shared, branches in the victim will reuse these BTB entries that were trained by the attacker. This causes the processor to mistakenly jump to the desired address while executing victim code, causing it to speculatively execute code of the attacker's choosing. Since there is no way for existing software defenses to prevent BTB poisoning, they cannot prevent such attacks. Worse yet, the retpoline defense [12] does not work as it is susceptible to a new Spectre variant called SpectreRSB [8] that manipulates the processor's Return Stack Buffer (RSB). SpectreRSB works similarly to BTB poisoning except that it poisons the RSB to influence the target prediction of return instructions.

In this paper, we present *Venkman*, a novel software-based solution to enhance existing software defenses against Spectre. Venkman employs compiler techniques to create bundles of instructions that are equally power-of-two sized and aligned. By bit-masking all indirect branch targets at run-time, Venkman restricts all branches to jump to the beginning of bundles. By transforming all code on the system with Venkman, all BTB entries can only be trained to jump to the beginning of bundles. If all instructions to mitigate Spectre attacks are contained within a bundle, training of the BTB will not be able to jump into the middle of a bundle to avoid executing them.

For processors that use dedicated return instructions, Venkman transforms returns to indirect jumps so that they use the BTB instead of the RSB for prediction. Bundle alignment and branch target restrictions are the key to ensuring that existing defenses are not bypassed: with Venkman, existing compiler-based Spectre defenses such as load fences [5] and Spectre-resistant SFI [3] on loads are always guaranteed to be executed on any program path taken during speculative execution. Venkman also utilizes Spectre-resistant SFI [3] on stores to protect the program's code segment from a new Meltdown variant [6] that breaks read-only memory protection. Our defenses are self-protecting; bundle alignment and branch target restrictions prevent SFI on stores from being speculatively

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
CCS '18, October 15–19, 2018, Toronto, ON, Canada
© 2018 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-5693-0/18/10.
https://doi.org/10.1145/3243734.3278522

bypassed, while SFI on stores ensures that the code segment is not modified during speculative execution.

Venkman instruments programs to resist Spectre. However, the instrumentation must be performed on every program in the system in order to take effect: one single uninstrumented program can mistrain the BTB to attack instrumented programs. Several solutions exist for ensuring that all code is transformed by Venkman. One is to use a system like Secure Virtual Architecture (SVA) [2] which forces all code to be shipped as virtual instruction set code; the SVA virtual machine will translate the code to native code before execution, transforming it with Venkman during code generation. Alternatively, the operating system and dynamic loader can use a binary code verifier like that in Google's Native Client (NaCl) [13] to verify that all programs it loads into memory for execution have been previously transformed with Venkman (the operating system kernel must be transformed with Venkman as well). If a program is found to be non-compliant with Venkman's restrictions, the operating system kernel can refuse to load the code for execution.

2 DESIGN

Venkman consists of two defenses: one combining bundle alignment and branch target restrictions that constrain the addresses to which branches can jump, and the other leveraging Spectre-resistant SFI [3] on memory writes to prevent speculative writes to the code segment. We show that our first defense mitigates mistraining of the processor's BTB, and our second defense defeats Read-only Protection Bypass attacks on the code segment [6].

2.1 Restricting Branch Targets to Bundles

Bundle Alignment. A bundle is a sequence of instructions that has a fixed power-of-two size and alignment in which the size and alignment are the same value. Venkman ensures that the target of all branches is at the beginning of a bundle. In this way, Venkman ensures that all the instructions inside a bundle are executed as a whole in the control flow. This requirement also implies that branches such as function calls must be at the end of a bundle so that their return addresses are aligned at the beginning of the next bundle. This is similar to NaCl [13].

Venkman transforms code during code generation to create bundles. It searches for basic blocks of instructions that are larger than the bundle size and breaks them into smaller basic blocks. When it breaks up larger basic blocks, Venkman ensures that instructions that must be co-located within the same bundle are not separated. For example, if the Spectre defense that Venkman is enhancing adds two instructions before every load, then Venkman will ensure that the load and the two instructions before it will end up in the same bundle, adding NOP instructions to the beginning of the bundle if keeping the instructions together results in a basic block that is smaller than the bundle size. For basic blocks with fewer instructions than the bundle size, Venkman adds NOP instructions to the beginning of the basic block to increase its size to the bundle size.

Once all the basic blocks are of the correct size, Venkman aligns each basic block. It also aligns the start address of each function to ensure it falls on a bundle boundary.

Branch Target Restrictions. Venkman uses compiler instrumentation to restrict the targets of a branch to be at the beginning of a bundle. This is similar to NaCl's control flow policy [13] that prevents jumps around required compiler instrumentation.

For direct branches, no instrumentation is needed; all basic blocks and functions are aligned, so the target address of all direct branches is also aligned. For indirect branches, Venkman adds two bit-masking instructions before every indirect branch. First, it adds an instruction which aligns the target address to a bundle boundary; for a bundle size of 2^S bytes, the bit-masking instruction clears the lowest-order S bits. Second, it adds a bit-masking instruction that ensures that the target is within the code segment; for example, if we place the code segment in the first 2^S bytes of the virtual address space, then our instrumentation clears the upper $(64 - S)$ bits of the target address.

If all code on a system is transformed in this way, and if all code segments reside in the same region of the virtual address space in all processes, then these restrictions on control flow prevent mistraining of the BTB with arbitrary addresses: only the starting address of a bundle can go into the BTB. Additionally, Venkman converts all return instructions into indirect branches. If such instructions use the RSB, it ensures that all addresses in the RSB reside on a bundle boundary (since all call instructions occur at the end of a bundle). If the conversion causes the branch to use the BTB instead of the RSB, then it still ensures that all targets are bundle addresses.

2.2 Speculative Code Segment Integrity

Not only can speculative execution leak secret data, but it can also corrupt memory locations temporarily during speculative execution via speculative memory writes [6]. Even if the operating system configures the code segment to be non-writeable, speculative corruption of the code segment may still be possible if the processor checks page permissions too late in the processor pipeline [6] and the result of the speculative write is forwarded to the instruction fetch unit for subsequent reads. To solve this problem, we use Spectre-resistant SFI [3] on store instructions to ensure that they do not speculatively write to the code segment. Spectre-resistant SFI works by creating a data dependence between the bit-masking SFI code and the subsequent memory access so that the memory access are stalled until the SFI code completes execution [3]. By using Spectre-resistant SFI on stores, Venkman ensures that the target address of stores are outside the code segment before the store begins to write memory speculatively.

In order to make Spectre-resistant SFI on stores efficient, Venkman must ensure that all code resides in one portion of the virtual address space and the heap, stack, globals, and memory mapped files occupy a separate region of the virtual address space. If the different regions for code and data are chosen carefully, a simple bit-masking operation with a constant value suffices to ensure that store instructions write to the region of the virtual address space that contains program data.

Venkman instruments each store with a bit-masking instruction to ensure that the target address of the store is outside the code segment. Additionally, Venkman ensures that the store and the bit-masking instruction before it are always in the same bundle. In this way, no changes in the control flow can execute the store without first executing the bit-masking instruction that prevents it from writing to the code segment.

Table 1: SPEC CPU 2017 Performance Results. NV = Normalized Venkman.

Benchmark	Baseline (s)	NV	NV Std. Dev.	Benchmark	Baseline (s)	NV	NV Std. Dev.	Benchmark	Baseline (s)	NV	NV Std. Dev.
500.perlbench_r	46.9	1.134	0.013	525.x264_r	72.5	1.040	0.038	619.lbm_s	228.1	0.990	0.026
502.gcc_r	68.5	1.099	0.015	531.deepsjeng_r	99.8	1.099	0.033	620.omnetpp_s	98.6	1.063	0.091
505.mcf_r	64.7	0.987	0.032	538.imagick_r	83.4	0.980	0.009	623.xalancbmk_s	111.9	1.217	0.014
508.namd_r	51.6	1.094	0.013	541.leela_r	133.4	1.083	0.018	625.x264_s	72.0	1.025	0.016
510.parest_r	68.9	1.009	0.011	544.nab_r	184.0	1.032	0.014	631.deepsjeng_s	117.6	1.060	0.043
511.povray_r	9.8	1.101	0.020	557.xz_r	53.0	1.025	0.094	638.imagick_s	82.9	0.993	0.017
519.lbm_r	29.8	1.027	0.029	600.perlbench_s	46.6	1.166	0.057	641.leela_s	132.8	1.072	0.012
520.omnetpp_r	105.3	1.030	0.122	602.gcc_s	68.4	1.108	0.010	644.nab_s	184.4	1.027	0.013
523.xalancbmk_r	112.4	1.232	0.030	605.mcf_s	66.0	0.960	0.034	657.xz_s	54.4	0.934	0.059

3 IMPLEMENTATION

We built an initial prototype of Venkman for the 64-bit POWER8 architecture [1] using the LLVM compiler [9]. We chose POWER8 because it uses fixed-length instructions, making it easier to write code that creates aligned bundles of instructions. Currently, only our code alignment and branch instrumentation is implemented.

We built Venkman as two `MachineFunctionPass` components added to the LLVM 4.0 compiler. The first pass searches for instructions that move values into the link register and counter register; all indirect branches on POWER8 use these two registers to hold the target of the branch [1]. In the case of the counter register, Venkman searches forward in the basic block to see if the counter register is used as an indirect branch target or if it is read by another instruction. In the former case, the value moved into the counter register is the target of a branch; in the latter case, it is a non-target value and is not modified. For values moved into the link register and target addresses moved into the counter register, Venkman adds two instructions that clear the lowest 4 bits (so that the target address is aligned on a 16-byte bundle boundary) and clears the highest 33 bits (to force the target to be a code segment address; Venkman will eventually force all code to be loaded into the lower 2 GB of the virtual address space).

The second `MachineFunctionPass` transforms each basic block to be a bundle of instructions that is 16 bytes long and aligned on a 16-byte boundary. It breaks larger basic blocks up into smaller basic blocks and adds NOP instructions to smaller basic blocks to make them equal to 16 bytes. During this process, Venkman ensures that instructions that must belong to the same bundle are co-located in the same basic block. For example, the bit-masking instructions on values moved to the link or counter register are always located in the same basic block as the branch using the target address value. Venkman also ensures that all call instructions appear at the end of a basic block; this ensures that the return address saved in the link register is the address of the next contiguous bundle in memory.

4 PRELIMINARY RESULTS

To evaluate Vekman's performance, we compiled the SPEC CPU 2017 benchmarks without Venkman (as the baseline) and with Venkman. We used a 64-bit 20-core IBM POWER8 machine running at 4.1 GHz. The machine has 64 GB of RAM and runs Linux 3.10.0.

Table 1 shows the baseline performance using the train inputs, and the overhead induced by Venkman normalized to the baseline. As Table 1 shows, the alignment and bit-masking of control data imposed by Venkman induces 0% to 23% overhead with an average of 5.9%.

5 CONCLUSIONS AND FUTURE WORK

This paper presents a software-based solution that mitigates BTB poisoning by ensuring that all software trains BTB entries to jump to aligned addresses within the code segment, thereby preventing malicious branches to arbitrary instructions. By deploying SFI on all store instructions, we can ensure that the processor never corrupts the code segment during speculative execution.

In future work, we will complete the Venkman prototype and incorporate our work into existing defenses such as Spectre-resistant SFI [3], providing a complete SFI solution that resists Spectre attacks. We will also measure the memory overhead induced by Venkman.

REFERENCES

[1] 2018. *Power ISA™ Version 2.07 B.* IBM.

[2] John Criswell, Andrew Lenharth, Dinakar Dhurjati, and Vikram Adve. 2007. 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).* 351–366. https://doi.org/10.1145/1294261.1294295

[3] Xiaowan Dong, Zhuojia Shen, John Criswell, Alan Cox, and Sandhya Dwarkadas. 2018. Spectres, Virtual Ghosts, and Hardware Support. In *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy (HASP'18).* Article 5, 9 pages. https://doi.org/10.1145/3214292.3214297

[4] Intel Corporation 2018. *Intel Analysis of Speculative Execution Side Channels.* Intel Corporation. Document Number: 336983-004.

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

[6] Vladimir Kiriansky and Carl Waldspurger. 2018. Speculative Buffer Overflows: Attacks and Defenses. *ArXiv e-prints* (July 2018). arXiv:1807.03757

[7] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (SP'19).*

[8] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2018. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *Proceedings of the 12th USENIX Workshop on Offensive Technologies (WOOT'18).* https://www.usenix.org/conference/woot18/presentation/koruyeh

[9] Chris Lattner and Vikram Adve. 2004. 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).* 75–86. http://dl.acm.org/citation.cfm?id=977395.977673

[10] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *Proceedings of the 27th USENIX Security Symposium (Security'18).* 973–990. https://www.usenix.org/conference/usenixsecurity18/presentation/lipp

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

[12] Paul Turner. 2018. Retpoline: A Software Construct for Preventing Branch-Target-Injection. https://support.google.com/faqs/answer/7625886.

[13] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2009. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *Proceedings of the 30th IEEE Symposium on Security and Privacy (SP'09).* 79–93. https://doi.org/10.1109/SP.2009.25