SpectreGuard: An Efficient Data-centric Defense Mechanism against Spectre Attacks

Jacob Fustos
University of Kansas
jacobfustos@ku.edu

Farzad Farshchi
University of Kansas
farshchi@ku.edu

Heechul Yun
University of Kansas
heechul.yun@ku.edu

ABSTRACT

Speculative execution is an essential performance enhancing technique in modern processors, but it has been shown to be insecure. In this paper, we propose SpectreGuard, a novel defense mechanism against Spectre attacks. In our approach, sensitive memory blocks (e.g., secret keys) are marked using simple OS/library API, which are then selectively protected by hardware from Spectre attacks via low-cost micro-architecture extension. This technique allows microprocessors to maintain high performance, while restoring the control to software developers to make security and performance trade-offs.

KEYWORDS

Spectre, Micro-architecture, Side-channel Attack

1 INTRODUCTION

Speculative execution is an essential performance enhancing technique in modern high-performance microprocessors. However, the recent disclosure of Spectre [16], Meltdown [18], and a growing number of related attacks [11, 15, 17, 20, 26] has shown that speculative execution can be a powerful security liability. Fundamentally, hardware speculation reduces CPU pipeline stalls—therefore improves performance—by predicting and speculatively executing a future instruction stream. If the prediction is wrong, the speculatively executed instructions (a.k.a., *transient* instructions [16]) are squashed, thus maintaining logical (architecturally visible program state) correctness. However, speculation can leave a footprint in micro-architectural states (e.g., loaded cache-lines in a cache), which can leak secret data to an adversary as demonstrated in the aforementioned attacks.

In this paper, we focus on the bounds check bypass variant of Spectre (Variant 1 [16]) as it affects all modern speculative out-of-order microprocessors and existing mitigation methods [7, 12, 15, 21, 27] are generally incur high overhead.

```
if (x < array1_size)
        y = array2[array1[x] * 4096];
```

Figure 1: A Spectre gadget. Adopted from [16]

Permission to make digital or hard copies of all or part 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 components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
DAC '19, June 2–6, 2019, Las Vegas, NV, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6725-7/19/06.

Figure 1 shows an example of vulnerable code (called a Spectre gadget). Here, an out-of-order processor can speculatively execute the second line before the array bounds check in the first line is completed. Speculatively executed instructions in the second line will eventually be discarded, if the x is out-of-bound, but the cache state changes will persist, which can leak secret.

Since the disclosure of Spectre, several software- and hardware-based mitigation strategies have been proposed. Software-based mitigation strategies prevent speculation by manually inserting a serializing instruction [12], or introducing additional data dependencies [7, 21] between the conditional jumps and later memory load instructions. However, manually identifying vulnerable branches of a program is difficult, while protecting all branches through compiler automation incurs too much performance overhead [21]. Hardware-based approaches generally focus on hiding attacker observable micro-architecture state changes by introducing additional hardware structures to buffer speculative outcomes [13, 27]. While they do not require manual code changes, they incur invasive hardware changes and high performance overhead [13, 27].

In this paper, we propose SpectreGuard, a novel cross-layer defense against Spectre attacks. Our approach is data-centric in the sense that we focus on a program's data rather than code. We observe that, for a software developer, identifying a program's secret data (e.g., secret keys) can be easier than identifying vulnerable code blocks (i.e., Spectre gadgets), which can appear in any branches of the program even if they are not related with processing the secret data. Thus, our approach begins by identifying sensitive memory blocks, which hold secret data, and marking them as *non-speculative memory regions.* The identified memory regions are informed to the OS, via simple OS/library API, and then utilized by the hardware to selectively and efficiently prevent speculative attacks via low-cost micro-architecture extensions.

Our micro-architecture extensions are small and based on a fundamental observation that a successful Spectre variant 1 attack requires the following three distinctive steps to occur speculatively: (Step 1) secret data is *loaded* from the memory hierarchy; (Step 2) it is then *forwarded* to dependent instructions; (Step 3) the dependent instructions are *executed*, leaking the secret via micro-architectural covert-channels (e.g., cache). It is important to note that the secret is leaked through the second and third steps, during which attacker observable, secret dependent, micro-architectural footprints are left. In other words, even if secret data is loaded on the CPU pipeline, unless it is forwarded to the secret dependent instructions, the secret cannot be leaked to the attacker.

Based on this observation, our approach allows the first step to occur but delays the second step until after all prior branches are resolved. Note that this delay is needed only when the virtual address in the first step is within a non-speculative memory region. All other "normal" addresses can be immediately forwarded to any waiting dependent instructions. Thus, secrets that are accessed infrequently, will have negligible overhead on overall performance.

Furthermore, even when we delay the second step (the forwarding step) the processor pipeline can still speculatively execute any independent instructions, if exist, thereby further reducing potential performance impact.

We implement SpectreGuard on Gem5 simulator and Linux. Our experimental results show that SpectreGuard can efficiently mitigate Spectre variant 1 attacks—we observe negligible overhead when the amount of secret data is small, while we observe modest 20% average overhead for SPEC2006 benchmarks when their entire address spaces are marked as non-speculative.

In summary, this paper makes the following **contributions**:

- We propose a novel data-centric defense approach, which we call SpectreGuard, to efficiently defend against Spectre variant 1 attacks. SpectreGuard drastically reduces the programming complexity and performance overhead by focusing on secret data, rather than code.
- We present a detailed micro-architecture and OS support mechanisms of SpectreGuard.
- We implement SpectreGuard on gem5 full system simulator and Linux kernel, and provide extensive evaluation results showing the effectiveness of SpectreGuard.

2 BACKGROUND

In this section, we provide necessary background on out-of-order architecture and known variants of speculative execution attacks.

2.1 Out-of-order Processors

Modern high-performance processors can execute multiple instructions in parallel. To maximize the instruction level parallelism, and thus to improve performance, they employ sophisticated hardware structures to execute as many instructions as quickly as possible, potentially *out-of-order*, while respecting data dependencies among the instructions.

A *reorder buffer (ROB)* is a key hardware structure that enables out-of-order execution. In an out-of-order processor, instructions are placed in the ROB in program order, executed in out-of-order, and retired in program order. An instruction in the ROB can begin execution as soon as its operands are available, and once it is executed, the result is maintained until the instruction is retired.

To maximize performance, branch predictors are used to *speculatively* issue and execute potential future instructions (those in the predicted execution paths) in parallel. If a prediction turns out to be incorrect, all subsequent speculative instructions after the branch are squashed. These speculative instructions are called *transient instructions* and a modern processor can hold a large number of transient instructions in its ROB.

2.2 Speculative Execution Attacks

Speculative execution attacks exploit the side-effects of executing transient instructions and generally involve the following three phases: access, transmit, and receive [27]. In the *access* phase, secret data is loaded from memory via speculative execution of memory access instructions. In the *transmit* phase that follows, secret dependent instructions are speculatively executed before the access instructions are squashed, which leave externally observable micro-architectural footprints. Lastly, in the *receive* phase, the attacker learns the secret from the micro-architectural footprints that remain, which form covert-channels that encode the secret value. While caches are the most commonly used micro-architectural covert-channels, other micro-architectural resources, as diverse as AVX (Advanced Vector eXtionsions) unit execution timing [23], can also be used as covert-channels.

Attack	Description
Variant 1 (Spectre) [16]	Bounds Check Bypass
Variant 1.1 [15]	Bounds Check Bypass Store
Variant 1.2 [15]	Read-only Protection Bypass
Variant 2 (Spectre) [16]	Branch Target Injection
Variant 3 (Meltdown) [18]	Supervisor Protection Bypass
Variant 3a [12]	System Register Bypass
Lazy FP [24]	FPU Register Bypass
Variant 4 [9]	Speculative Store Bypass
ret2spec [20]	Return Stack Buffer
L1 Terminal Fault [11, 26]	Virtual Translation Bypass

Table 1: Known speculative execution attacks.

Table 1 summarizes the known speculative execution attacks, which can be categorized as exception- and prediction-based attacks. In exception-based attacks (Variants 1.2, 3, 3a, Lazy FP, and L1TF), a transient access instruction triggers an exception, but the subsequent transient transmit instructions are executed before the exception is fully handled. In prediction-based attacks (Variant 1, 1.1, 2, 4, and ret2spec), hardware prediction components (e.g., branch predictors) are mis-trained to execute transient access and transmit instructions before the misprediction is realized and resolved.

Note that while exception based speculative execution attacks such as Meltdown are important, an effective software-based mitigation method exists [8], which is already adopted on all major operating systems. Furthermore, Intel already introduced their next-generation architectures that implement hardware-based defense mechanisms against these attacks [2]. Meanwhile, most processors from AMD and ARM are already immune to Meltdown and many other exception based attacks [6]. Therefore, the problem appears to be largely solved or will be solved in the near future.

However, the same cannot be said for the prediction-based speculative attacks because existing solutions generally involve high performance and/or programming overhead [7, 21]. In particular, for Spectre Variant 1 attacks, which we focus on in this paper, existing software-based mitigation methods rely on either programmer's manual identification of potentially vulnerable code, which is not only time consuming but also likely to miss many true vulnerabilities, or compiler solutions which can falsely identify code that does not need to be modified [16, 21]. Furthermore, there are no easy hardware-based fixes because branch speculation that Spectre Variant 1 attacks target is so fundamental to all modern out-of-order processors and disabling it would incur unacceptably high performance overhead [16]. Because of this reason, processor vendors do not have an immediate plan for hardware fixes and recommend the software-based mitigation solutions [12].

In this work, we focus on efficient mitigation of Spectre Variant 1 attacks via a software/hardware collaborative approach.

3 THREAT MODEL

We assume the attacker can control a subset of victim's input. We assume the victim program and the OS are logically correct and do not have exploitable software bugs (e.g., buffer overflow). We

assume that the attacker knows the (virtual) memory addresses of the secret data (e.g., private keys, pass phrases) but he does not have direct access to them. Thus, the attacker's goal is to learn the content of the secret data using speculative execution attacks.

As discussed in Section 2, speculative execution attacks involve speculative execution of access and transmit gadgets. In particular, the speculative execution of transmit instructions encodes the secret data via micro-architectural covert channels, such as cache, through which the attacker recovers the secret. We do not assume specific covert channels—i.e., any micro-architectural covert-channels can be used by the transmit gadgets.

Note that covert channels created by executing logically valid instructions (i.e., non-transient instructions) are out-of-scope. For example, traditional cache timing attacks against cryptographic algorithms exploiting secret dependent (valid) memory accesses and execution time variations (e.g., [3, 19]) are important but orthogonal problems that should be protected using existing solutions (e.g., constant-time crypto implementations [4]).

Lastly, we do not make any assumption about receiver's locations. The receiver can be in the victim's address space, or in different address space in different SMT, different core, or remote node [23].

In this setting, our goal is to prevent leaking victim's secret data, with the assumption that the memory addresses of the secret data are known to the victim, just like we assume they are known to the attacker. In the following section, we will describe how we can leverage this knowledge of secret data locations to devise an efficient and effective defense mechanism against speculative execution attacks.

4 SPECTREGUARD

In this section, we describe the high-level design and implementation details of SpectreGuard.

4.1 Design

The design goal of SpectreGuard is to effectively defend against Spectre attacks while maximizing application performance and reducing required software/hardware complexities.

Marking Secret Data. Our first insight is that identifying a program's secret data (e.g., private keys, passwords) can be much easier than identifying vulnerable code parts (i.e., Spectre gadgets) because the former only requires understanding of the program itself but the latter may also require understanding of the underlying hardware. For instance, many security sensitive programs (e.g., browsers) already identify and manage security sensitive memory blocks differently by judiciously checking memory access bounds, and/or employing in-process memory isolation mechanisms [22, 25]. In contrast, identifying Spectre gadgets in a program is challenging because they can exist in anywhere in the program—even in places that do not directly access secret—and it may require in-depth micro-architectural understanding.

Based on this insight, our approach begins by identifying sensitive memory blocks, which hold secret data, and marking them as *Non-Speculative (NS) memory regions* to defend them against Spectre attacks. In this work, we propose to leverage page-based hardware memory management unit (MMU). Specifically, we introduce an additional *NS* bit in each page table entry to indicate the page is a non-speculative memory page. The non-speculative memory regions can be declared via OS system calls (e.g., `mmap`, `mprotect`), libraries, or compiler/linker support, all of which internally update the *NS* bit in the program's page table entries. The *NS* bit information of a page table entry is then passed to the TLB, along with address translation and other auxiliary information. This information is then used by the CPU's out-of-order instruction scheduler as described in the following.

Preventing Secret Dependent Speculative Execution. Our second insight is that Spectre attacks can leak secret only when *secret dependent instructions* are speculatively executed. A Spectre gadget (e.g, Figure 1) executes two types of transient instructions: (1) an *access* instruction that loads (secret) data from the memory hierarchy and (2) a *transmit* instruction that changes micro-architectural states depending on the content of the secret data. Note that the secret is leaked through executing the transmit instruction, which creates micro-architectural covert-channels, not the access instruction. In other words, even if the secret data is loaded into the CPU's internal buffers—by completing the first step—unless it is forwarded to the secret dependent instructions, the secret cannot be leaked to the attacker.

Based on this insight, our idea is to delay the result forwarding of a speculative memory access instruction until after the execution is deemed safe. This can be achieved by slightly modifying the CPU's result forwarding mechanism. In an out-of-order processor, an issued instruction is kept in a reorder buffer (ROB) until the instruction is retired. Normally, when an instruction is executed, the result is immediately forwarded to any dependent instructions. In our approach, however, when a memory instruction's address is marked as *NS*, obtained from the TLB, the result is not immediately forwarded but kept in the reorder buffer instead.

The important question is then, when do we forward the result of the memory instruction? Forwarding the result of the memory instruction is the most safe when the instruction is at the head of the ROB at which point all preceding instructions have already been retired. However, this approach can cause frequent pipeline stalls when there are not many independent instructions in the ROB. We instead improve performance by forwarding the result of a memory instruction *before* it reaches the head of the ROB, but only *after* all prior branch instructions are completed (verified predicted correctly), at which point, barring exceptions, the memory instruction and subsequent dependent instructions are no-longer speculative and will eventually be retired.

Security Analysis. SpectreGuard mitigates Spectre Variant 1 attacks targeting secret data in the victim's address space as long as the secret data is located in non-speculative memory regions. Efficient mitigation of other Spectre Variants are, while important, out-of-scope of this work.

4.2 Implementation

We implement SpectreGuard on a Gem5 full system simulator [5] and Linux kernel.

Gem5 Extensions. We have modified the O3CPU out-of-order CPU model of the gem5 simulator in order to implement SpectreGuard. Figure 2 shows the path of a load instruction through the IEW (issue/execution/writeback) and commit units of the O3CPU model, showing how that path is modified for addresses marked as non-speculative regions in the page tables (via the *NS* bit in each page table entry).

First, instructions are placed in instruction queue (IQ), load/store queue (LSQ)—for memory instructions—and reorder buffer (ROB) in

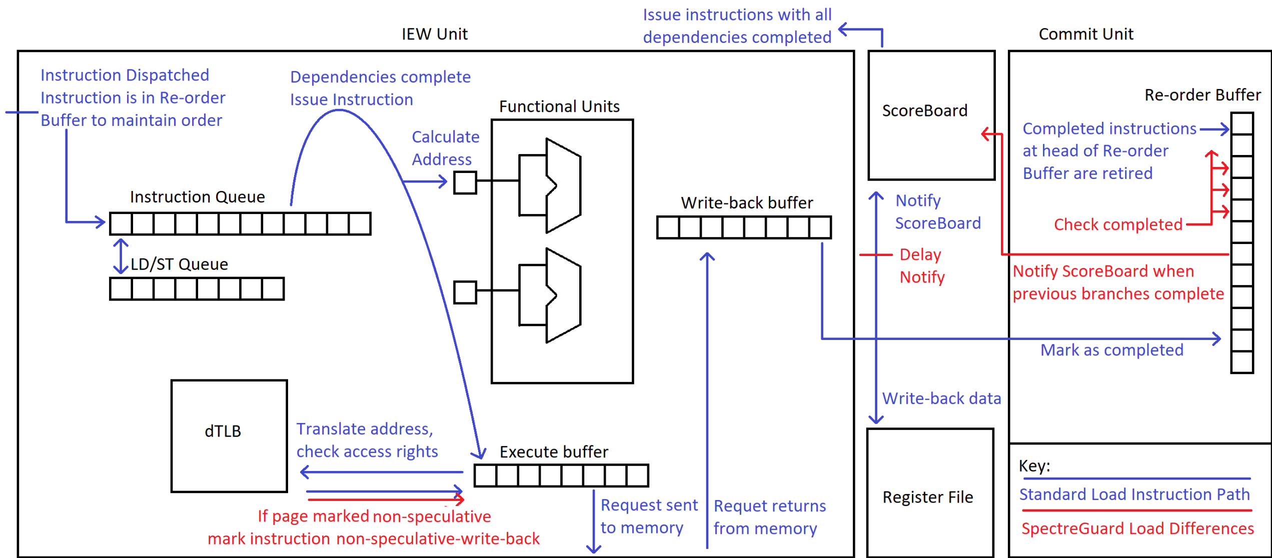

Figure 2: Micro-architectural changes needed to implement SpectreGuard.

program order. They then are dispatched (issued) to functional units in out-of-order as soon as their data dependencies are resolved.

Once an instruction's dependencies have been marked as completed, the ScoreBoard will issue the instruction. If it is a memory instruction (load/store), its memory address is calculated with the help of a data translation lookaside buffer (dTLB). A dTLB entry includes a virtual-to-physical page address mapping and other auxiliary information, including the non-speculative (NS) bit, in our modified implementation. If the address of a memory instruction is marked as *NS*, then the instruction is marked as non-speculative-write-back (NSWB) in the ROB.

After the memory request is completed, the returned value is written back to the temporary register file (called rename register) and the instruction is marked as completed in the ROB. Normally, a completed instruction in the ROB is immediately notified to the ScoreBoard, so that it can issues any dependent instructions as soon as possible. In our modification, however, if the instruction is marked as NSWB, notification to the ScoreBoard is *delayed* until all prior branch instructions in the ROB are completed.

Linux Extensions. We have modified Linux kernel 4.18.12 to allow users to map pages as non-speculative to be protected from Spectre v1 attacks. Within the Linux kernel, a task's virtual address space is represented with a set of virtual memory areas (VMAs). Each VMA shares common properties through a per-VMA data structure. When a new page is allocated (at a page fault), a page table entry is created based on the VMA defined memory flags. We add a new flag `VM_WBNS` to indicate the page is part of non-speculative memory. At the user-level, we extend Linux's program (ELF) loader and the `mmap` system call implementation to be able to set the `VM_WBNS` flags in creating VMA descriptors.

Note that the above code changes are minimal. In total, we only have added/modified less than 50 lines of C in the Linux kernel source tree. Furthermore, because most changes are in page table descriptors and their initialization, no observable run-time overhead is incurred by the code changes.

5 EVALUATION

In this section, we present the evaluation results of SpectreGuard.

5.1 Setup

We modify and configure the Gem5 simulator to support the following processor configurations: *Native, SG, InvisiSpec, and Fence*. *Native* is the baseline configuration vulnerable to Spectre attacks. *SG* represents our SpectreGuard approach. For SG, we additionally use parenthesis to denote data or memory regions marked as non-speculative. For example, SG(Heap) denotes that the entire heap area of the program is marked as non-speculative. *InvisiSpec* is a recently published fully hardware-based speculative attack defense mechanism [27]. Lastly, *Fence* represents a software-based protection method that inserts a `lfence` instruction in every branch of the program [10]. Table 2 shows the basic simulation parameters we used in all processor configurations.

Core	Single-core (x86 ISA), 8 issue, out-of-order, 2 GHz IQ: 64, ROB: 192, LSQ: 32/32
Cache	Private L1-I/D: 16/64 KiB (4/8-way), 1 cycle latency Shared L2: 256 KiB (16-way), 8 cycle latency
DRAM	Read/write buffers: 32/64, open-adaptive policy DDR3@800MHz, 1 rank, 8 banks

Table 2: Gem5 simulation parameters

5.2 Performance of Synthetic Workloads

In this experiment, we show the effects of SpectreGuard using a synthetic benchmark. Figure 3 shows the pseudo code of our synthetic benchmark, which loosely mimics a sand-boxed application runtime scenario (e.g., JavaScript engine in a browser). The first part of the benchmark represents a client workload (e.g., attacker's JavaScript code), which contain Spectre gadgets. Specifically, `do_work()` function includes a Spectre v1 gadget similar to

```
char *secret_key; // secret data

void benchmark(int S, int C)
{
    // (S)pectre gadget, unrelated to the secret
    for (i = 0; i < S; i++)
        do_work();

    // En(C)ryption task accessing the secret
    for (i = 0; i < C; i++)
        encrypt();
}
```

Figure 3: Pseudo code of our synthetic benchmark composed of (S)pectre gadget and en(C)ryption workloads (the input parameters control respective workload sizes.)

Figure 1. The second part represents background communication activities that use a secret key (secret_key) for data encryption (e.g., encrypted HTTPS communication). For data encryption, we use the AES encryption function in OpenSSL v1.1. The amounts of work of the two code sections, denoted as 'S' and 'C', are adjusted to simulate various execution scenarios. For example, "25S/75C" indicates that it spends 25% time on executing the Spectre gadgets workload, while spending 75% of time on executing the encryption workload, which accesses the secret, on the native processor.

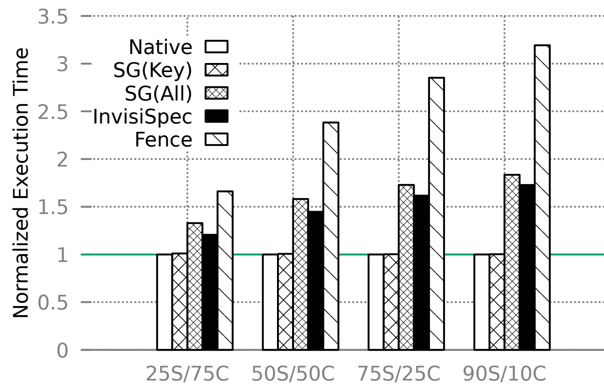

Figure 4: Execution times of synthetic workloads.

Figure 4 shows the results. First, note that *SG(Key)*, in which only the secret key is marked as non-speculative, achieves near native performance in all workload configurations. This is because of the following two reasons: (1) Most instructions that do not depend on the secret key are executed natively without any performance penalty; (2) Even among the dependent instructions that access the secret key, in the OpenSSL library, the majority of them are executed natively without delay because there were no prior unresolved branch instructions in the ROB that delay their execution under SpectreGuard. However, the execution times increase considerably in *SG(All)*, where the entire address space of the benchmark is marked as non-speculative. This is expected as it effectively turn all memory accesses (heap, data, stack) as sensitive and thus any memory dependent instructions may be delayed until all prior branch instructions are resolved (especially bad for Spectre gadget section part of the code, which contains many branch instructions.) Nevertheless, *SG(All)* achieves comparable performance with *InvisiSpec*, which is a fully hardware based state-of-the-art defense mechanism. Note that InvisiSpec requires significantly more complex hardware modifications than SpectreGuard. Lastly, *Fence*, today's software-only Spectre mitigation mechanism, suffers significantly worse execution time increases in all workload configurations.

5.3 Performance of SPEC2006 Workloads

We repeat the experiment in 5.2 using a set of SPEC CPU2006 benchmarks. Note that *SG(Heap)* and *SG(All)* are different SpectreGuard configurations where the former marks only the heap area of a benchmark as non-speculative, while the latter marks the entire address space of a benchmark non-speculative.

Figure 5 show the results. On average, *SG(Heap)* shows 8% overhead while both *SG(All)* and *InvisiSpec* show similar 20% overhead. However, the overhead of *Fence* is significantly higher at 74%. Note that the performance improvement of *SG(Heap)* comes from the fact that it only protects heap, while *SG(All)* protects the entire address space. The results shows that there exists performance security trade-offs when using SpectreGuard. Nevertheless, the fact that even the most conservative *SG(All)* achieve comparable performance to *InvisiSpec*, which requires significantly more hardware resources than our approach, shows effectiveness of our approach. For an actual security sensitive program, we expect that the programmer can identify smaller subset of memory regions as security sensitive and mark them as non-speculative memory under SpectreGuard, further lowering overhead.

6 RELATED WORK

The most closely related work is InvisiSpec [27], which is a recently proposed fully hardware-based speculative attack mitigation technique (not requiring any software changes). InvisiSpec introduces an additional hardware structure, called speculative buffer, that buffers speculative memory accesses to hide speculation from cache hierarchy. While the approach is effective in eliminating cache-based side-channels, it is vulnerable to attacks that use other types of side/covert-channels such as the AVX unit based timing channel [23]. In contrast, our approach is not vulnerable to non-cache based covert-channels because we prevent secret dependent transient instructions, which create covert channels, as long as secret data are marked as non-speculative. Moreover, our approach requires much more modest hardware changes compared to InvisiSpec. SafeSpec [13] is similar to InvisiSpec as it also introduces additional hardware structures to hide speculation and thus suffer similar drawbacks as InvisiSpec. DAWG [14] prevents cache-based side/covert channels among different security domains by partitioning the cache. However, it does not protect attacks originated from the same domain as the victim.

As briefly discussed in Section 2.2, for Spectre variant 1 attacks, adding a fence instruction to each vulnerable branch, either manually or automatically with compiler support, is a practically viable mitigation method on existing processors as recommended by the chip vendors, but it suffers high performance overhead. To reduce performance impact, several software based techniques have been proposed. Data dependent masking [15] reduce the scope of out-of-bound memory access, thereby reducing the attack possibility. Speculative Load Hardening (SLH) [7] uses an additional data-dependent

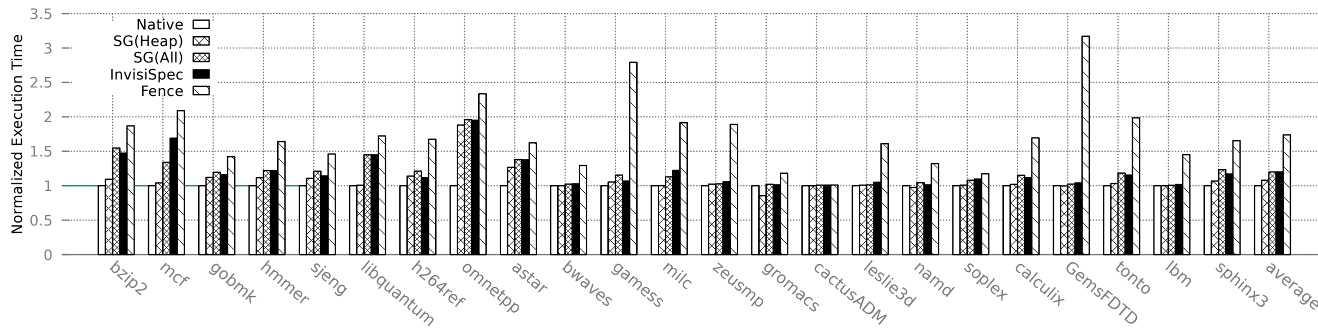

Figure 5: SPEC2006 benchmark results.

conditional instruction, which creates a data-dependency, which achieves similar effect as to the standard fence-based mitigation. Unlike the fence-based approach, however, because independent instructions can still be speculatively executed, it generally achieves higher performance, although the overall performance loss is still undesirably high for high performance applications [21]. Oleksenko et al. also proposed different ways of introducing data dependencies (comparison arguments, conditional flag, etc.), all of which prevent unsafe speculation. However, these software-based approaches suffer from high performance overhead and error-prone manual code modifications. In contrast, our approach requires minimal software and hardware modifications while offering effective defense against Spectre attacks with low performance overhead.

7 CONCLUSION AND FUTURE WORK

In this paper, we proposed SpectreGuard, a novel defense mechanism against Spectre attacks which allows microprocessors to maintain high performance, while restoring the control to software developers to make security and performance trade-offs. In our approach, sensitive memory blocks (e.g., secret keys) are marked as non-speculative memory regions by the programmer. The OS updates the corresponding page table entries to encode the information, which is then passed to the hardware via MMU/TLB. When the out-of-order CPU speculatively executes a memory instruction, execution of dependent instructions are delayed if the memory address is tagged as non-speculative, until they can be executed safely. We implement our approach in Gem5 and Linux. Our evaluation showed that in many application scenarios where secure memory is accessed infrequently, our approach can achieve near native performance. Furthermore, our approach incurs significantly lower hardware complexity compared to the state-of-the-art hardware mitigation techniques. As future work, we plan to implement and evaluate the effectiveness of our approach on a RISC-V based out-of-order core [1] on FPGA.

ACKNOWLEDGMENTS

This research is supported in part by NSF grant CNS 1718880 and NSA Science of Security initiative contract no. #H98230-18-D-0009.

REFERENCES

[1] The berkeley out-of-order RISC-V processor code repository. https://github.com/ucb-bar/riscv-boom.
[2] Spectre and Meltdown in Hardware: Intel Clarifies Whiskey Lake and Amber Lake. https://www.anandtech.com/show/13301, 2018.
[3] D. J. Bernstein. Cache-timing attacks on aes. 2005.
[4] D. J. Bernstein, T. Lange, and P. Schwabe. The security impact of a new cryptographic library. In *International Conference on Cryptology and Information Security in Latin America*, pages 159–176. Springer, 2012.
[5] N. Binkert et al. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, Aug. 2011.
[6] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. Von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. Technical report, 2018.
[7] C. Carruth. Speculative Load Hardening: A Spectre Variant #1 Mitiation Technique. 2018.
[8] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard. KASLR is Dead: Long Live KASLR. In *ESSoS*, pages 161–176. Springer, 2017.
[9] J. Horn. speculative execution, variant 4: speculative store bypass. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528, 2018.
[10] Intel. Analyzing potential bounds check bypass vulnerabilities (Rev. 002). Technical report, July 2018.
[11] Intel. Deep Dive: Intel Analysis of L1 Terminal Fault. Technical report, 2018.
[12] Intel. Intel Analysis of Speculative Execution Side Channels (Rev. 4.0). Technical report, July 2018.
[13] K. N. Khasawneh, E. M. Koruyeh, C. Song, D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh. SafeSpec: Banishing the Spectre of a Meltdown with Leakage-Free Speculation. 2018.
[14] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, J. Emer, M. I. T. Csail, and N. M. I. T. Csail. DAWG : A Defense Against Cache Timing Attacks in Speculative Execution Processors. In *MICRO*, 2018.
[15] V. Kiriansky and C. Waldspurger. Speculative buffer overflows: Attacks and defenses. *arXiv preprint arXiv:1807.03757*, 2018.
[16] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.
[17] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *WOOT*, 2018.
[18] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading kernel memory from user space. In *USENIX Security)*, 2018.
[19] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-level cache side-channel attacks are practical. In *Security and Privacy (SP)*, pages 605–622. IEEE, 2015.
[20] G. Maisuradze and C. Rossow. ret2spec: Speculative execution using return stack buffers. In *ACM(CCS)*, pages 2109–2122. ACM, 2018.
[21] O. Oleksenko, B. Trach, T. Reiher, M. Silberstein, and C. Fetzer. You shall not bypass: Employing data dependencies to prevent bounds check bypass. *arXiv preprint arXiv:1805.08506*, 2018.
[22] S. Park, S. Lee, W. Xu, H. Moon, and T. Kim. libmpk: Software Abstraction for Intel Memory Protection Keys. *arXiv preprint arXiv:1811.07276*, 2018.
[23] M. Schwarz, M. Schwarzl, M. Lipp, and D. Gruss. NetSpectre: Read Arbitrary Memory over Network. In *ACM(CCS)*, 2018.
[24] J. Stecklina and T. Prescher. Lazyfp: Leaking fpu register state using microarchitectural side-channels. *arXiv preprint arXiv:1806.07480*, 2018.
[25] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, D. Garg, and P. Druschel. ERIM: Secure and Efficient In-process Isolation with Memory Protection Keys. *arXiv preprint arXiv:1801.06822*, 2018.
[26] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, R. Strackx, and K. Leuven. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security Symposium (Security)*, pages 991–1008, 2018.
[27] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. W. Fletcher, and J. Torrellas. InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy. In *International Symposium on Microarchitecture (MICRO)*, 2018.