Hardware Speculation Vulnerabilities and Mitigations

Nathan Swearingen*, Ryan Hosler*, Xukai Zou*
*Department of Computer Science
Indiana University-Purdue University Indianapolis
Indianapolis, Indiana 46202, USA
nswearin@iu.edu, rjhosler@iu.edu, xzou@iupui.edu

***Abstract*—This paper will discuss speculation vulnerabilities, which arise from hardware speculation, an optimization technique. Unlike many other types of vulnerabilities, these are very difficult to patch completely, and there are techniques developed to mitigate them. We will look at many of the variants of this type of vulnerability. We will look at the techniques mitigating those vulnerabilities and the effectiveness and scope of each. Finally, we will compare and evaluate different vulnerabilities and mitigation techniques and recommend how various mitigation techniques apply to different situations.**

I. Introduction

Some CPUs today are affected by certain attacks related to hardware speculation. Modern CPUs use pipelining to increase performance, which involves overlapping the stages of successive instructions. However, the CPU often doesn't have the necessary information to carry the next instruction through the pipeline. For example, if the current instruction is a conditional branch instruction, the CPU likely won't know which instruction comes next. Therefore, when a CPU encounters a conditional instruction, it sometimes predicts the result and begins processing instructions that it expects will come next, and as a result, may expose information that is supposed to be protected [1].

While the effects of speculation cannot be seen from the architectural state, covert channels can be used to obtain information accessed during speculation [2]. Unfortunately, hardware vendors may not fix all such vulnerabilities, so it is crucial to create patches that protect against such attacks [1].

The paper is organized as follows. In Section II, we will look at what kinds of vulnerabilities arise from speculation. In Section III, we will look at many different techniques for mitigating some of the vulnerabilities. In addition to exploring various mitigation strategies, we will be comparing them in Section IV and recommending how to apply different techniques to different situations in Section V.

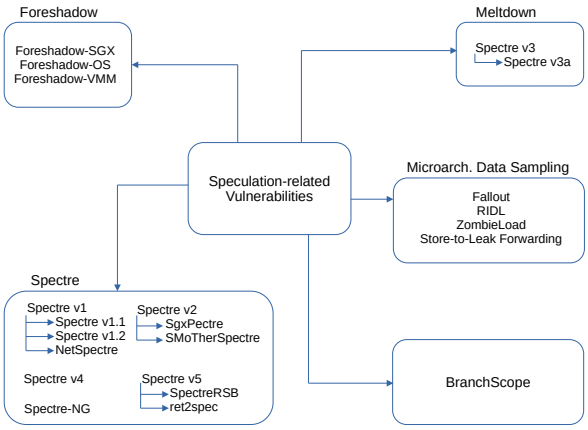

Fig. 1. Relationships between vulnerabilities

II. Vulnerabilities

Johnson and Davies give 10 different speculation vulnerabilities: Spectre variants 1, 2, 3a, and 4, Branchscope, Meltdown, three variants of Foreshadow, and SgxPectre [3]. Five additional vulnerabilities are referenced in [4]: Spectre-NG, Spectre v1.1, Spectre v1.2, Spectre v5, and NetSpectre. Described below are these and other related vulnerabilities. Also, see Figure 1 for a visual representation of the relationships between these vulnerabilities.

A. Bounds Check Bypass

Spectre v1, also called *bounds check bypass*, occurs when memory is speculatively accessed out of bounds [3]. For example, a mispredicted bound check may cause a read out of bounds. If confidential data is read, and the value is then used as the index to another array, that element of the second array may be cached, resulting in a detectable difference in the element's access time. Such difference can infer the secret data [5].

1) Bounds Check Bypass Store: Spectre v1.1, also called *bounds check bypass store* [4], involves exploiting an out-of-bounds write (as opposed to a read in Spectre v1) to leak information [5].

This is the author's manuscript of the article published in final edited form as:

Swearingen, N., Hosler, R., & Zou, X. (2021). Hardware Speculation Vulnerabilities and Mitigations. 2021 IEEE 18th International Conference on Mobile Ad Hoc and Smart Systems (MASS), 609–614. https://doi.org/10.1109/MASS52906.2021.00087

2) Read-Only Protection Bypass: Spectre v1.2, also called *read-only protection bypass* [4], is an attack that is similar to Spectre v1 and allows the attacker to overwrite read-only data [5].

3) Remote Bounds Check Bypass: NetSpectre, also called *remote bounds check bypass* [4], is an attack based on Spectre v1 that can be performed over the network. It involves exploiting both a *leak gadget* and a *transmit gadget*. The authors show that AVX2 can be used as a side-channel to leak 60 bits per hour [6].

B. Branch Target Injection

Spectre v2, also called *branch target injection* [3], involves "training" indirect branch predictors so that an attacker can cause certain code to be executed speculatively [7]. Assume that a secret value is stored in some register. If the attacker can cause an indirect branch to speculatively jump to a portion of code that reads this register, they may leak the secret through a side-channel [8].

1) SgxPectre: SgxPectre [9] is an attack involving Intel Software Guard eXtensions (SGX), based on Spectre v2. It poisons the branch target buffer and exploits gadgets to leak information through the cache. The authors show that Intel SGX SDK is vulnerable and conclude that any enclave program written with it is also vulnerable. They claim that indirect branch restricted speculation (IBRS) mitigates the vulnerability [9].

2) SMoTherSpectre: SMoTherSpectre relies on SMoTher, a vulnerability that exploits port contention. In the paper, SMoTherSpectre uses branch target injection to cause the victim to execute a conditional instruction that depends on some data and then uses SMoTher to determine which code section to execute. SMoTherSpectre can be mitigated by addressing the speculation artifact it relies on, e.g., branch target injection [10].

C. Speculative Store Bypass

Spectre v4, also called *speculative store bypass* [3], involves memory disambiguation predictors, which speculatively executes load instructions before potentially conflicting store instructions, which could leak information through a side-channel [7].

D. Spectre v5

Spectre v5 includes *Ret2Spec* and *SpecRSB* [4], both involving the return stack buffer (RSB) [11] [12].

1) SpectreRSB: Mis-speculation can occur if the software stack does not match the return stack buffer (RSB). In an example of SpectreRSB given by the authors, the attacker calls a function that pops from the stack. As a result, the stack and the RSB differ, and mis-speculation results. During this mis-speculation, a secret value is read, and then once execution continues on the correct path, the value is leaked through a cache side-channel [11].

2) ret2spec: The authors of ret2spec describe an attack in which the attacker adds addresses of code gadgets to the RSB and then causes a context switch to the victim process [12].

E. Lazy FP State Restore

Spectre-NG, also called *lazy fp state restore* [4], is a vulnerability that exploits lazy FPU context switching. When a context switch occurs, the FPU and SIMD register sets are sometimes not switched until needed. A fault may be generated on the first use of FPU and SIMD instructions. If the processor transiently executes some of the next instructions, they can leak information from these register sets through the cache [13].

F. Meltdown

Meltdown, also called *rogue data cache load* [3] and Spectre v3 [4], is a vulnerability that relates to out-of-order execution. When this vulnerability is exploited, it allows access to kernel memory from userspace by exploiting both the out-of-order execution after a trap instruction and the fact that on some CPUs, some speculatively executed instructions can circumvent the protection of memory [14].

1) Rogue System Register Read: Spectre v3a, also called *rogue system register read* [3], involves leaking system parameters through side-channel analysis [15]. This is done by speculatively loading a system register and using the value in speculative instructions. Moreover, this allows the attacker to read the values of system registers by using a cache side-channel [16].

G. Foreshadow

Foreshadow is an attack that breaks the security of the Intel SGX. To obtain one byte of information from an SGX enclave using Foreshadow, an "oracle buffer" is first allocated. For the attack to succeed, the confidential data must be in the L1 cache. An attacker may be able to accomplish this simply by executing the enclave. The attacker also needs to revoke access permissions on the enclave page. The secret pointer is then dereferenced and used as an index into the oracle buffer. Finally, the attacker times accesses to the oracle buffer to determine which value is cached, leaking the transiently read byte. The authors also discuss how an attacker in kernel mode may achieve better results [17].

H. Foreshadow-NG

Foreshadow-NG, or L1 Terminal Fault, refers to a class of vulnerabilities that includes two variants in addition to the original Foreshadow vulnerability.

Foreshadow-SGX refers to the original vulnerability described in the previous section. Foreshadow-OS refers to a vulnerability through which an unprivileged adversary can exploit out-of-order instructions to read certain data from the cache. Foreshadow-VMM refers to a vulnerability through which a guest virtual machine can clear the present bit in their page table and then read any memory in the physical cache [18].

I. Fallout

Fallout is a vulnerability that targets the store buffer, which contains values from recent stores and forwards them to subsequent loads. The attack can be performed by allocating a page and revoking the page's access permissions. The victim writes a value to their page, and then the attacker attempts to read from their page at the same offset. The read will fail since access permissions were revoked. However, before retiring, the read will transiently use the value recently written to the victim page due to what the authors refer to as the WTF (Write Transient Forwarding) optimization [19].

J. Rogue In-Flight Data Load (RIDL) and Store-to-Leak forwarding

Fallout was developed concurrently to another related attack called the RIDL attack [20]. While Fallout exploits the WTF optimization, the RIDL attack mainly exploits the Line Fill Buffer, which the CPU uses to store and optimize memory operations [20] [19].

Store-to-Leak forwarding is another related attack. As opposed to Fallout, which exploits a false positive match from the store buffer, store-to-leak forwarding exploits true positives and true negatives [21]. The authors of [22] consider this as microarchitectural data sampling attack (MDSA).

K. Other Fallout-related Vulnerabilities

ZombieLoad is an attack that exploits the fact that when load instructions fault, they may transiently dereference destinations that are in the fill buffer by the same logical CPU or a sibling one. The authors conclude that disabling hyperthreading is the only way to mitigate ZombieLoad on current hardware [23].

Other Fallout-related attacks are primarily based on microarchitectural data sampling [24] and can be found at mdsattacks.com. They allow an attacker to leak data that is "in-flight" from internal buffers. Moreover, they do not require that data are ever present in a CPU cache [24] [19].

L. Branchscope

Branchscope is a vulnerability involving the directional branch predictor. When two processes run on the same physical core, they may share a directional branch predictor. The attack first involves causing predictions for both the victim and the attacker to use 1-level branch predictors. Next, the attacker causes the victim code to execute and can then guess the state of the branch predictor by performing conditionals and testing whether they cause mispredictions or correct predictions. Although Branchscope does not involve speculation, it is related to branch target injection since it also exploits branch predictor collisions [25].

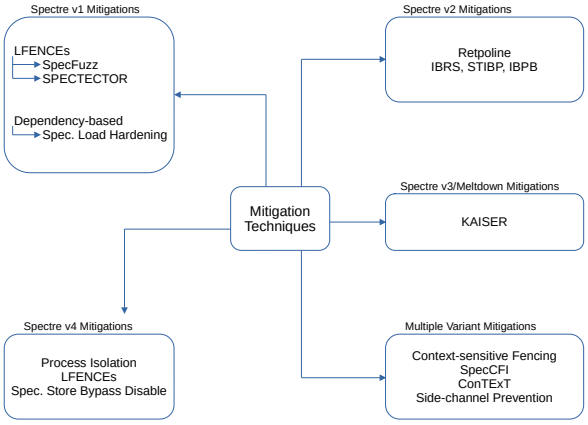

Fig. 2. Relationships between mitigations

III. Mitigation Techniques

There are many proposed mitigations for speculation vulnerabilities. In this section, we will look at many of these proposed mitigation techniques. Moreover, we will discuss the underlying mechanisms behind these techniques and the targeted vulnerabilities. See Figure 2 for a visual representation of the similarities between the techniques.

A. Intel Recommendations

In January 2018, Intel released a white paper explaining some of the vulnerabilities and recommended mitigations [26]. For Spectre v1, they recommend using LFENCE instructions (Load FENCE) to prevent speculation. The LFENCE instruction provides a performance-efficient way of ensuring load ordering between routines that produce weakly-ordered results and routines that consume that data. For Spectre v2, they recommend two techniques. One involves three new CPU capabilities: Indirect Branch Restricted Speculation (IBRS), Single Thread Indirect Branch Predictors (STIBP), and Indirect Branch Predictor Barrier (IBPB). The second technique uses retpolines, which we discuss later. For Meltdown, they recommend a technique called KAISER presented in [27] originally intended to prevent Kernel Address Space Layout Randomization side-channel attacks. It involves keeping privileged pages from being mapped during the execution of user code, and it accomplishes this using a "User" paging structure along with a separate "Supervisor" paging structure [26].

Another paper by Intel recommends process isolation or LFENCEs to mitigate speculative store bypass. It also offers to set Speculative Store Bypass Disable, which prevents loads from executing speculatively until the CPU determines the address of all prior stores [7].

B. Speculative Load Hardening

To mitigate Spectre v1, Carruth suggests *speculative load hardening*, an approach that requires modifications to compilers and operating systems [28]. This approach has a significant performance impact compared to no mitigation but performs better than using LFENCE instructions for every conditional. Speculative load hardening involves "hardening" data with operations that depend on the result of speculation. For example, a mask may be applied to a loaded value. The value of the mask depends on the condition that led to speculation. Operating systems must be updated to protect certain parts of memory.

Carruth discusses several of the implementation details and challenges to the mitigation, considering factors such as processor-specific details and interaction with unmitigated code. Benchmarks on Google's microbenchmark suite and a "large highly-tuned server built using ThinLTO and PGO" ultimately showed that this mitigation approach was 1.77 times faster than LFENCE-based mitigation, and for most large applications had an overhead of no more than 30% [28].

C. Dependency-based Mitigation

Oleksenko et al. discuss and compare two types of Spectre v1 mitigation: LFENCE-based mitigation and dependency-based mitigation [29]. LFENCE-based mitigation, as discussed before, involves using LFENCE instructions to prevent speculation. Dependency-based mitigation involves creating data dependencies to protect data without preventing safe instructions from being executed, which reduces performance. In addition to speculative load hardening, dependencies can be created with LAHF instructions (Load AH from Flags) or on arguments of the comparison. However, when dependencies are created this way, the vulnerability is not entirely mitigated because of potentially reordered instructions.

The authors compare LFENCE-based mitigation, speculative load hardening, and mitigation with dependencies on arguments in terms of performance using benchmarks from the Phoenix benchmark suite [30]. LFENCE-based mitigation was consistently the slowest, but whether speculative load hardening was faster or slower than using dependencies on arguments varied from program to program [29].

D. SpecFuzz

This approach, proposed by Oleksenko et al., suggests a new technique they call *speculation exposure*, which involves simulating speculation in software [1]. They implement this technique in what they call *SpecFuzz*, a tool to check for bounds check bypass vulnerabilities (although they also consider how to mitigate other vulnerabilities). The basic idea is as follows: when a program is compiled to be tested, before every conditional, a checkpoint is inserted, followed by the conditional inverted.

SpecFuzz relies on traditional input fuzzing along with the speculation exposure described above. Using these two techniques, SpecFuzz identifies which instructions seem to be safe and whitelists them. Then, only the remaining instructions need to be instrumented to protect against vulnerabilities and improve performance without compromising security significantly.

They also evaluated SpecFuzz in terms of performance. Depending on the configuration, the proportion of relevant branches that weren't instrumented ranged from 15% to 77%. Hardened libraries showed a performance benefit when using this technique as opposed to full hardening. The benefit varied depending on the library and patching criteria, but they improved up to 234% with the JSMN library, a minimalistic JSON parser in C. The authors also recognize the limitations of their work, such as the complexity of nested simulation and false negatives caused by insufficient input fuzzing [1].

E. SPECTECTOR

The following approach, proposed by Guarnieri et al., involves a property referred to as *speculative non-interference* (SNI) [31]. This is a property that the authors define that describes security against speculative attacks. According to the authors, it is "the first semantic notion of security against speculative execution attacks" [31]. Their work relies on using symbolic execution to prove this property in compiled programs. They define an assembly language that they call μASM, which they use to analyze programs for SNI. To use the concept of speculative non-interference, the authors created a program called *SPECTECTOR*, which mainly detects Spectre v1 vulnerabilities.

To determine how scalable SPECTECTOR is, they tested it on the Xen hypervisor. They had to make simplifications, and as a result, could not make any security conclusions about Xen. Despite the caveats to the test, the authors believe it is a meaningful test of scalability. They found that checking for the SNI property was not significantly more costly than finding symbolic paths.

The authors recognize some weaknesses of SPECTECTOR, such as inaccurate simulation of specific hardware. The authors of SpecFuzz [1] mention that tools such as SPECTECTOR use symbolic execution, which

can offer better security guarantees than fuzzing but can result in "combinatorial explosion" [1] [31].

F. Retpoline

Retpoline is a mitigation technique for Spectre variant 2. It is a software technique that may require microcode updates on some processors in order to work properly. Retpoline involves patching indirect branches so that the processor branches to an infinite loop and stays there during speculation, rather than potentially branching to dangerous code. On some processors, the RSB (Return Stack Buffer) may be emptied for various reasons. One way to work around this is with "RSB stuffing" [8].

Although Retpoline uses LFENCE and PAUSE instructions, it can still attain reasonable performance. An LFENCE instruction that executes speculatively may not ever need to retire, resulting in a lesser impact on performance than is often associated with LFENCE instructions. Because Retpoline requires programs to be recompiled to be mitigated, it may not be a suitable solution in all situations. Future Intel processors that use "enhanced IBRS" mitigation will not need Retpoline [8]. IBRS "restricts speculation of indirect branches" [26], the cause of branch target injection [8].

G. Context-sensitive Fencing

Context-sensitive fencing (CSF), proposed by Taram et al., is a mitigation that can defend against Spectre variants 1, 1.1, 1.2, 2, and 5 [4]. It functions at the microcode level. CSF involves instrumenting instructions with newly proposed fences, LSQ-LFENCE, LSQ-MFENCE, and CFENCE, which are fence instructions that, unlike LFENCE, are explicitly targeted at mitigation of speculation vulnerabilities and therefore do not need to be as strict, resulting in improved performance. These fences are inserted by a microcode engine with context-sensitive decoding capabilities.

The authors also limit the number of instances in which fencing is used for additional performance gains. They accomplish this by only adding a fence for the first instruction in a basic block. Another technique they use to reduce the amount of fencing is Decoder-Level Information Flow Tracking (DLIFT) [32] to track potentially dangerous data so that only loads that involve such data can be protected. Variants 2 and 5 are protected against with model-specific range registers (MSRRs) that mark protected portions of code. When such code is entered, the branch predictor is reset, preventing an attacker from having any influence.

The authors evaluate their mitigation on the SPEC CPU2006 suite. Performance varied depending on the techniques used and the program being tested but stayed below twice the execution time of no mitigation. Generally, LFENCE performed the worst, followed by LSQ-MFENCE, and then CFENCE [4].

H. SpecCFI

The following approach we will look at, proposed by Koruyeh et al., is a hardware-level technique that uses control flow integrity (CFI) to prevent certain speculation attacks [2]. In their paper, the authors describe how it can be used to prevent two particular speculation attacks: Spectre-BTB (Spectre v2; Branch Target Buffer) and Spectre-RSB (Spectre v5; Return Stack Buffer). They argue that when combined with existing Spectre-PHT (Spectre v1) mitigation techniques, they can close "all known non-vendor-specific Spectre vulnerabilities" [2].

CFI requires that when executing a program, control flow must agree with a pre-determined control flow graph (CFG). Once the CFG has been constructed, labels are created that restrict which control flow transfers are allowed—control flow may only be transferred between points with the same label. To make use of the labels, the Instruction Set Architecture can be extended with two modifications: extending `jmp` and `call` with CFI labels, and adding a `cfi_lbl` instruction to label indirect branch targets. Furthermore, the compiler must add these labels to compiled code, but most compilers already support this through CFI.

The authors study the security, performance, and complexity of their proposal. They state that SpecCFI can only mitigate Spectre-BTB and Spectre-RSB but that SpecCFI and Spectre-PHT mitigations should be combined to achieve certain security properties.

To test the performance of their proposal, the authors used a simulator called *MARSSx86*. The benchmarks they used were the SPEC2017 benchmarks (those which they were able to compile). They measured performance relative to no protection and compared this relative performance to the relative performance of *Retpoline-style software fencing*, and *All Target Fencing*, which involves adding an LFENCE instruction at the target of every indirect call, jump, or return. They found that while the other two solutions impacted performance significantly, SpecCFI did not.

Finally, the authors compared SpecCFI to nine other mitigation techniques/categories: DAWG, SafeSpec/InvisiSpec, LFENCE, IBRS/IBPB/STIBP, speculative load hardening/dependency-based mitigation, Retpoline, RSB stuffing, Context-sensitive fencing, and ConTExT. Looking at the level of mitigation for different vulnerabilities, SpecCFI, when combined with Spectre-PHT mitigation, was the only technique that fully mitigated all considered vulnerabilities [2].

I. ConTExT

ConTExT, proposed by Schwarz et al. [22], is a mitigation technique that prevents all Spectre attacks known at the time of the paper, as well as microarchitectural

data sampling attacks. It involves changes to applications, compilers, operating systems, and hardware. It requires an indication of what data may be secret. The compiler then marks such data as secret. Such memory is marked as *non-transient* by the operating system and cannot be accessed in transient execution. When *non-transient* memory is loaded into a register, that register is tainted, meaning it is marked as *non-transient*. *Non-transient* data is also tracked through operations, and registers can be untainted if their contents are replaced or written to memory that is not *non-transient*. As a result, confidential data can be kept track of as it propagates through storage.

The authors evaluate the performance of ConTExT through emulation with the Bochs emulator. They conclude that the overhead would be less than 1% for typical workloads. They also implement ConTExT-light, which roughly emulates ConTExT but can be implemented in software. Although ConTExT-light does not have the same security properties as ConTExT, the authors use it to obtain a rough upper bound on the overhead of ConTExT. Using this technique, they conclude an overhead of 71.14% for security-critical applications and observe that this overhead is lower than that produced by combining recommended existing mitigations such as serialization barriers, Spectre-BTB mitigations like IBRS, and additional mitigations for other variants [22].

J. Side-channel Prevention

In contrast to inhibiting speculation, some techniques have been proposed to prevent side-channels [2]. By restricting the use of a side-channel, vulnerabilities can be prevented since no information accessed during speculation can be leaked. We will briefly consider three such techniques: SafeSpec [33], InvisiSpec [34], and DAWG [35]. SafeSpec uses separate structures to store information during speculation, and the information is effectively removed if mis-speculation occurs [33]. With InvisiSpec, speculative loads are stored in a speculative buffer and do not affect the cache hierarchy [34]. DAWG (Dynamically Allocated Way Guard) addresses cache side-channels with a set-associative structure that involves protection domains [35]. These techniques defend against multiple speculation vulnerabilities. Furthermore, they exhibit reasonable performance. However, mitigation using these kinds of techniques generally requires hardware modifications [33] [34] [35].

IV. Comparison

Speculation vulnerabilities are a complex topic. They arise from hardware speculation, which is when the CPU makes predictions to pipeline instructions that depend on each other. Speculation is important for achieving good performance [2], so disabling it isn't a practical option.

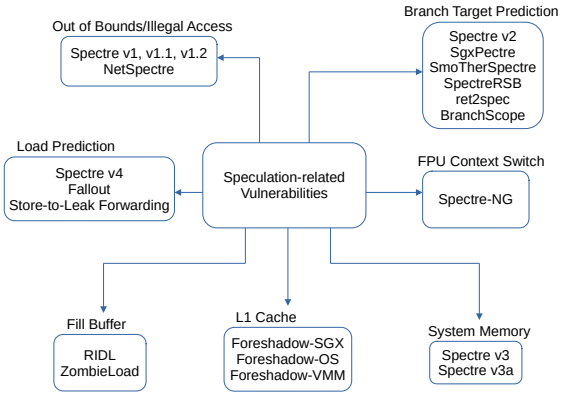

Fig. 3. Vulnerability targets/mechanisms

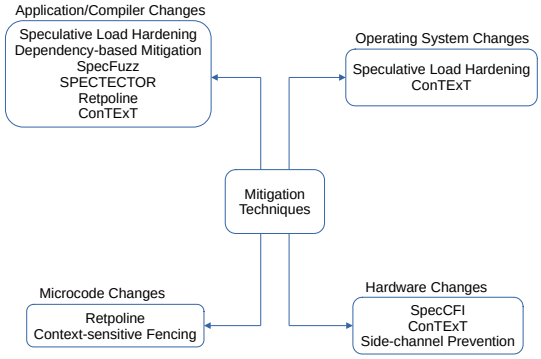

Fig. 4. Changes required for each mitigation

There are no simple solutions that fix the vulnerabilities completely, but there's a wide variety of potential ways to handle the problem, ranging from compilation patches to hardware assistance, and each solution has its scope of effectiveness. In this paper, the proposals we have looked at all take a different approach and do not all handle the same types of vulnerabilities. Therefore, none of the solutions on its own provides mitigation that is reasonable and sufficient for all situations.

See Table I for a comparison of mitigation techniques. Figure 3 compares the targets/mechanisms of each vulnerability. Figure 4 shows which types of changes are required for different mitigations.

The first technique we studied, *speculative load hardening* [28], involved creating dependencies to completely mitigate Spectre v1 (see Table I). Although this has a significant performance impact, it allows for complete mitigation while only relying on modification to existing software. We also looked at how to use dependencies to mitigate Spectre v1 [29]. Such software modifications can have a significant performance impact. There are ways to optimize software modifications; one example is discussed next.

Mitigation Technique	Main Targeted Variants	Security	Expected Changes Required	Practicality
Speculative Load Hardening [28]	Spectre v1	Complete	Applications/Compilers, operating systems	Decreased performance; programs must be recompiled; mitigation is per application
Dependency-based Mitigation [29]	Spectre v1	Depends	Applications/Compilers at a minimum	Decreased performance; recompilation/patching necessary; mitigation is per application
SpecFuzz [1]	Spectre v1	Partial	Applications	Reasonable performance; reasonable mitigation time, but per application
SPECTECTOR [31]	Spectre v1	Complete	Applications	Reasonable performance; mitigation may be impractical on large projects; mitigation is per application
Retpoline [8]	Spectre v2	Complete	Applications/Compilers, possibly microcode	Somewhat reasonable performance [8] [36] [2]; mitigation may require microcode updates; mitigation is per application
Context-sensitive Fencing [4]	Spectre v1, v1.1, v1.2, v2, v5	Complete	Microcode	Reasonable performance; mitigation requires microcode changes
SpecCFI [2]	Spectre v2, v5	Complete	Hardware	Reasonable performance; mitigation involves hardware modification
ConTExT [22]	All Spectre variants, microarchitectural data sampling attacks	Complete	Applications, compilers, operating systems, hardware	Mitigation involves hardware modifications, and also requires indication of secret data
Side-channel Prevention [33] [34] [35]	Potentially Many Variants	Complete	Hardware	Mitigation likely involves hardware modification

TABLE I
CHARACTERISTICS OF EACH MITIGATION

Note that this table does not consider potential improvements that could be made to the tools or techniques. Also note that according to/implied by [2], Spectre-PHT (Spectre v1) defenses and mitigations like SpecCFI may be incomplete without each other; this table does not take this into account.

SpecFuzz [1] used simulations to detect Spectre v1 vulnerabilities. It relied on existing memory safety techniques and a new technique called *speculation exposure*. Together, these techniques allow for the detection of vulnerabilities in software. The authors used this technique to whitelist instructions that did not seem vulnerable in a tool they created called *SpecFuzz*. One advantage to this technique is that vulnerabilities are mitigated unless tests suggest that they can't be exploited, so this technique allows the developer to improve the performance of a program while, for the most part, not negatively impacting security (with respect to complete LFENCE mitigation). However, false negatives are still possible. Also, SpecFuzz only addresses Spectre v1 vulnerabilities, although the authors indicate that the techniques could be applied to other types of vulnerabilities.

In one paper we looked at, the authors defined security from certain vulnerabilities formally [31]. They referred to this property as *speculative non-interference* and implemented the concept in a program called *SPECTECTOR*, which can detect Spectre v1 vulnerabilities. One advantage to this approach is that it relies on formal proof of security rather than simulations that can be misleading. However, although the program relies on formal proof of security, the definition of security used may not correspond perfectly to vulnerabilities on actual hardware [31].

Retpoline is a mitigation technique for Spectre v2 [8]. While it is a software-based technique, it may require microcode updates on certain processors. Again, being a software technique, it only applies to software that has incorporated Retpoline. However, as newer processors are released and become standard, Retpoline may become obsolete to techniques provided by newer processors such as enhanced IBRS.

Context-sensitive fencing (CSF) [4] is a technique that completely mitigates multiple variants of Spectre, as listed in Table I. It works at the microcode level, which could make mitigation challenging in some cases. The idea behind CSF is to make use of custom fences that are intended for Spectre mitigation and therefore have better performance than LFENCE. It also involves limiting the number of fences added. Because of the improvements over complete LFENCE mitigation, CSF maintains a reasonable performance overhead.

SpecCFI [2] involved hardware modifications. The paper addressed multiple Spectre variants, namely Spectre-BTB and Spectre-RSB. To prevent vulnerabilities, the authors relied on CFI. While the techniques presented in the paper do not provide sufficient mitigation on their own, the authors believe that broad Spectre mitigation can be achieved by combining their techniques with existing Spectre v1 mitigation techniques. A disadvantage to this approach is that it requires hardware

Type of Situation	Potential Mitigation Approaches
Isolated System	Mitigation not necessary
Low-risk	SpecFuzz, Retpoline, KAISER
High-risk	LFENCEs, Speculative Load Hardening, SPECTECTOR, Retpoline, Enhanced IBRS, Context-sensitive Fencing, SpecCFI, ConTExT, Side-channel Prevention

TABLE II
APPLICATIONS OF MITIGATION TECHNIQUES

modifications. However, the authors' tests did not show a significant performance impact. Because the authors rely on CFI, an already existing technique, significant modifications to compilers are likely unnecessary since many can already add the labels necessary for CFI.

ConTExT [22] is the only technique presented here that provides full mitigation against Spectre attacks on its own by addressing speculation. However, mitigation requires changes to applications, compilers, operating systems, and software, and the technique requires that confidential data be marked. As a result, ConTExT may be impractical for many situations but could potentially be used to provide full mitigation in high-risk situations where implementation is reasonable.

One final technique we looked at involved addressing side-channels rather than speculation [2] [33] [34] [35]. This approach can be very effective; this technique, along with ConTExT, can be applied to many variants. Unfortunately, implementing it will likely require hardware modifications.

V. EVALUATION

There are several different approaches to handling speculation vulnerabilities. Moreover, of course, this paper is not exhaustive. With a large amount of research available on the topic, there are many options to choose from when it comes to patching existing software and systems, ranging from no patching to patching compiled software, hardware modifications, or a combination of multiple approaches.

The best approach for a given situation will depend on the security needs of the application. Next, we will look at some examples of which types of situations might benefit from which mitigation approaches we looked at. However, it is important to note that security is a rapidly evolving field. Any actual mitigation carried out should be done in light of the most up-to-date research and will also require careful consideration of the risks and trade-offs of the situation. These examples show how the proposals we looked at could potentially be useful in real applications. They are summarized in Table II.

For local, isolated systems, no mitigation is necessary, even if the system handles confidential data. This is because an attacker will be unable to leak secrets through a side-channel. A low-risk situation may involve only non-confidential data or may present little opportunity for side-channels. In such a scenario, mitigation can focus on techniques that perform well, such as SpecFuzz and Retpoline. KAISER could be used to mitigate Meltdown. Full mitigation is likely unnecessary. Lastly, a scenario that presents high-risk may need to use mitigation techniques that provide complete protection. Spectre v1 could be mitigated with LFENCEs, speculative load hardening, or SPECTECTOR. Retpoline or enhanced IBRS could be used to mitigate Spectre v2. Context-sensitive fencing and SpecCFI could be used for further protection. Alternatively, full mitigation could be achieved with techniques such as ConTExT or side-channel prevention.

Finally, it is worth mentioning that entities that are not working directly with source code or compiled applications may not need to concern themselves with which mitigation approaches are being used. Keeping all software, operating systems, and microcode up-to-date is the best way to defend against vulnerabilities. However, depending on the situation, it may still be important to evaluate which vulnerabilities exist and whether they are mitigated.

VI. CONCLUSION

Speculation is an important element in the performance of modern CPUs. Unfortunately, it can result in vulnerabilities. There are many types of these vulnerabilities, and mitigation has proven difficult. Here, we looked at many proposed techniques and saw how mitigations could take different approaches and have their applications and limitations. We also recommended a few possible mitigation approaches involving existing techniques. As new research is developed, we can expect mitigations to become even more practical, and eventually, the vulnerabilities may no longer be a security concern to developers and organizations.

ACKNOWLEDGMENT

This work is partially supported by U.S. National Science Foundation (grants #OAC-1839746 and #DGE-2011117).

REFERENCES

[1] O. Oleksenko, B. Trach, M. Silberstein, and C. Fetzer, "Specfuzz: Bringing spectre-type vulnerabilities to the surface," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 1481–1498. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/oleksenko

[2] E. Koruyeh, S. H. A. Shirazi, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Speccfi: Mitigating spectre attacks using cfi informed speculation," in *2020 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2020, pp. 39–53. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP40000.2020.00033

[3] A. Johnson and R. Davies, "Speculative execution attack methodologies (seam): An overview and component modelling of spectre, meltdown and foreshadow attack methods," in *2019 7th Inter. Sympo. on Digital Forensics and Security (ISDFS)*, 2019, pp. 1–6.

[4] M. Taram, A. Venkat, and D. Tullsen, "Context-sensitive fencing: Securing speculative execution via microcode customization," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 395–410. [Online]. Available: https://doi.org/10.1145/3297858.3304060

[5] V. Kiriansky and C. Waldspurger, "Speculative buffer overflows: Attacks and defenses," 2018.

[6] M. Schwarz, M. Schwarzl, M. Lipp, and D. Gruss, "Netspectre: Read arbitrary memory over network," 2018.

[7] "Speculative execution side channel mitigations," https://software.intel.com/content/www/us/en/develop/download/speculative-execution-side-channel-mitigations.html, Jul. 2018.

[8] "Retpoline: A branch target injection mitigation," https://software.intel.com/security-software-guidance/api-app/sites/default/files/Retpoline-A-Branch-Target-Injection-Mitigation.pdf, Jun. 2018.

[9] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. Lai, "Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution," *IEEE Security Privacy*, vol. 18, no. 3, pp. 28–37, 2020.

[10] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "Smotherspectre: exploiting speculative execution through port contention," *CoRR*, vol. abs/1903.01843, 2019. [Online]. Available: http://arxiv.org/abs/1903.01843

[11] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre returns! speculation attacks using the return stack buffer," in *12th USENIX Workshop on Offensive Technologies (WOOT 18)*. Baltimore, MD: USENIX Association, Aug. 2018. [Online]. Available: https://www.usenix.org/conference/woot18/presentation/koruyeh

[12] G. Maisuradze and C. Rossow, "ret2spec," *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, Jan 2018. [Online]. Available: http://dx.doi.org/10.1145/3243734.3243761

[13] J. Stecklina and T. Prescher, "Lazyfp: Leaking fpu register state using microarchitectural side-channels," 2018.

[14] 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 (SP)*, 2019, pp. 1–19.

[15] "Nvd - cve-2018-3640," https://nvd.nist.gov/vuln/detail/CVE-2018-3640, Aug. 2020.

[16] "Cve-2018-3640," https://access.redhat.com/security/cve/cve-2018-3640, May 2018.

[17] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution," in *Proceedings of the 27th USENIX Security Symposium*. USENIX Association, August 2018, see also technical report Foreshadow-NG [18].

[18] O. Weisse, J. Van Bulck, M. Minkin, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, R. Strackx, T. F. Wenisch, and Y. Yarom, "Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution," *Technical report*, 2018, see also USENIX Security paper Foreshadow [17].

[19] M. Minkin, D. Moghimi, M. Lipp, M. Schwarz, J. V. Bulck, D. Genkin, D. Gruss, F. Piessens, B. Sunar, and Y. Yarom, "Fallout: Reading kernel writes from user space," 2019.

[20] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "RIDL: Rogue in-flight data load," in *S&P*, May 2019.

[21] M. Schwarz, C. Canella, L. Giner, and D. Gruss, "Store-to-leak forwarding: Leaking data on meltdown-resistant cpus (updated and extended version)," 2021.

[22] M. Schwarz, R. Schilling, F. Kargl, M. Lipp, C. Canella, and D. Gruss, "Context: Leakage-free transient execution," *CoRR*, vol. abs/1905.09100, 2019. [Online]. Available: http://arxiv.org/abs/1905.09100

[23] M. Schwarz, M. Lipp, D. Moghimi, J. V. Bulck, J. Stecklina, T. Prescher, and D. Gruss, "Zombieload: Cross-privilege-boundary data sampling," 2019.

[24] "Mds: Microarchitectural data sampling," https://mdsattacks.com/, 2019.

[25] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, ECE, and D. Ponomarev, "Branchscope: A new side-channel attack on directional branch predictor," in *Proc, of the 23rd Inter. Conf. on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 693–707. [Online]. Available: https://doi.org/10.1145/3173162.3173204

[26] "Intel analysis of speculative execution side channels," https://www.intel.com/content/www/us/en/architecture-and-technology/intel-analysis-of-speculative-execution-side-channels-paper.html, Jan. 2018.

[27] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard, "Kaslr is dead: Long live kaslr," in *Engineering Secure Software and Systems - 9th International Symposium, ESSoS 2017, Proceedings*, ser. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 10379 LNCS. Italy: Springer-Verlag Italia, 2017, pp. 161–176.

[28] C. Carruth, "Speculative load hardening (a spectre variant #1 mitigation)," https://lists.llvm.org/pipermail/llvm-dev/2018-March/122085.html, Mar. 2018.

[29] O. Oleksenko, B. Trach, T. Reiher, M. Silberstein, and C. Fetzer, "You shall not bypass: Employing data dependencies to prevent bounds check bypass," *CoRR*, vol. abs/1805.08506, 2018. [Online]. Available: http://arxiv.org/abs/1805.08506

[30] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating mapreduce for multi-core and multiprocessor systems," in *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 2007, pp. 13–24.

[31] M. Guarnieri, B. Kopf, J. F. Morales, J. Reineke, and A. Sanchez, "Spectector: Principled detection of speculative information flows," in *2020 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2020, pp. 1–19. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP40000.2020.00011

[32] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," in *Proc. of the 11th Inter. Conf. on Architectural Support for Programming Languages and OS*, ser. ASPLOS XI. New York, NY, USA: Association for Computing Machinery, 2004, p. 85–96. [Online]. Available: https://doi.org/10.1145/1024393.1024404

[33] 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.

[34] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas, "Invisispec: Making speculative execution invisible in the cache hierarchy," in *51st Annual IEEE/ACM Inter. Symposium on Microarchitecture (MICRO)*, 2018, pp. 428–441.

[35] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, "Dawg: A defense against cache timing attacks in speculative execution processors," in *51st Annual IEEE/ACM Inter. Symp. on Microarchitecture (MICRO)*, 2018, pp. 974–987.

[36] P. Turner, "Retpoline: a software construct for preventing branch-target-injection," https://support.google.com/faqs/answer/7625886.