Preventing Speculative Probing Attacks

Neophytos Christou
neophytos_christou@brown.edu

Abstract

Spectre attacks have been a major focus of the security community in recent years. These types of attacks exploit speculative execution, a fundamental paradigm of modern CPUs, to transiently execute instructions that do not belong to the normal execution flow of the program, access sensitive program data and leak it thought microarchitectural side channels. Effective mitigations against certain Spectre variants have been rolled out, but new ways of exploiting speculative execution keep on surfacing.

Recent work has shown that by combining speculative execution attacks with memory corruption vulnerabilities, an adversary can effectively bypass both Spectre and memory corruption mitigations and manage to speculatively hijack the control flow of the vulnerable program. A certain class of these attacks, called speculative probing, leverages a single memory corruption to allow an attacker to disclose program information using Spectre-like primitives, without triggering any memory corruption defenses. Using the disclosed information, the attacker can then mount a traditional control-flow hijacking attack.

In this proposal, I will explore how speculative probing attacks can be effectively mitigated in software, while still keeping the performance benefits that are gained from speculation.

1 Introduction

Memory corruption bugs have been a prevalent problem for computer security researchers for decades. By corrupting sensitive program data, attackers can take control of the control flow of the vulnerable program. Security researchers have been developing defense mechanisms for years which aim to make it more difficult for adversaries to leverage such bugs [20]. A lot of these mechanisms, such as address space layout randomization (ASLR) [14], stack canaries [4] and non-executable memory can be found in most commodity software.

Another, more recent class of attacks is the Spectre family of attacks [10]. Using these types of attacks, adversaries can leverage speculative (or *transient*) execution [6] to speculatively access sensitive program data, bring them into microarchitectural buffers and leak the data using side channels [12]. Mitigations against a lot of the Spectre family variants have been rolled out [19], but researchers keep introducing new ways of exploiting speculative execution [15, 23].

Researchers have so far treated Spectre attacks and memory corruption attacks as two separate domains. However, recent work [5, 13] has shown that memory corruption vulnerabilities can be combined with Spectre-like primitives to bypass both memory corruption and Spectre mitigations. Adversaries can corrupt memory and then trigger transient execution to speculatively hijack the control flow of the vulnerable program, despite memory corruption defenses being in place.

These attacks are not trivial to mitigate, since Spectre mitigations do not usually account for corrupted memory and vice-versa. This proposal and outlines speculative probing, an attack primitive that combines memory corruption with speculative execution to gain control over a vulnerable program and introduces a software-based solution against these attacks.

2 Background

2.1 Spectre

Modern CPUs execute instructions out-of-order in order to avoid idle CPU cycles and gain performance. As a result, instructions that have a data dependency on a previous instruction might start executing before the dependency is resolved. To avoid stalling the pipeline, CPUs use various predictors to guess the data dependency and start *speculatively* executing the instruction, thus keeping the pipeline busy. If the CPU correctly predicts the dependency, the pipeline stall will be successfully avoided. If the prediction turns out to be incorrect, the CPU reverts its architectural state and re-executes using the correct data. However, the speculative execution may still leave observable side-effects on the microarchitectural state of the CPU.

Spectre attacks [10] exploit the fact that, after a misspeculation, data brought in the CPU's microarchitectural buffers during transient execution can be leaked. Even though there are multiple variants of Spectre [2], all the attacks can be divided in three general steps that an attacker needs to carry out. First, the adversary needs to train some CPU predictor – usually one that predicts the outcome of a control-flow instruction – to cause the CPU to later speculatively execute an attacker-chosen piece of code. Second, the attacker triggers transient execution. The attacker-chosen instructions speculatively execute and access sensitive program data, bringing them into the microarchitectural state (e.g. the cache). Finally, the attacker uses a side channel to exfiltrate the secret data from the microarchitectural state.

Researchers and CPU vendors have been coming up with both software and hardware techniques [1, 7] that mitigate many of the variants of the Spectre family of attacks by either preventing speculation [11, 21], hindering side channels [24] or preventing CPU predictors from being mistrained [8, 9].

2.2 Combining Spectre with memory corruption vulnerabilities

Recent work has shown ways of overcoming some of the limitations of Spectre attacks by combining them with memory corruption bugs.

SPEAR attacks [13] aim to bypass conventional memory safety mechanisms using speculative execution. The attacker overwrites some control-flow influencing data that would normally trigger a memory corruption mitigation, preventing program exploitation. However, the adversary can still achieve a *speculative control-flow hijack* and leak sensitive data from memory, before speculation eventually ends and the memory corruption mitigation is architecturally triggered.

Göktaş et al. [5] introduce a second, more powerful primitive, called speculative probing. Speculative probing allows an attacker to combine Spectre-like primitives with a single memory corruption vulnerability to leak data from a running processes without triggering any memory corruption mitigation mechanisms. This allows attackers to target more high-value, crash-resistant software such as the kernel and leverage leaked data to bypass strong memory mitigations such as ASLR to eventually mount an architectural control-flow hijacking attack without crashing the program.

2.3 Speculative probing

Speculative probing leverages a corrupted code pointer which is used as the target of an indirect control flow instruction. Particularly, the instruction which uses the corrupted pointer lies within the *speculation window* of a conditional branch – when the CPU speculates on the outcome of the branch, the indirect control flow instruction will be one of the instructions that are transiently executed before the CPU resolves the branch outcome.

One of the most comprehensive memory corruption mitigations an attacker needs to overcome to successfully mount an exploit is ASLR. In the presence of ASLR, the location of code and data regions in the address space are randomized. As a result, the attacker cannot reliably hijack the control flow of the program, since the location of the exploit payload is unknown.

Using speculative probing, the attacker can bypass such randomization schemes. For example, to carry out a successful ROP exploit [17], the attacker first needs to locate code region of the program, as seen in

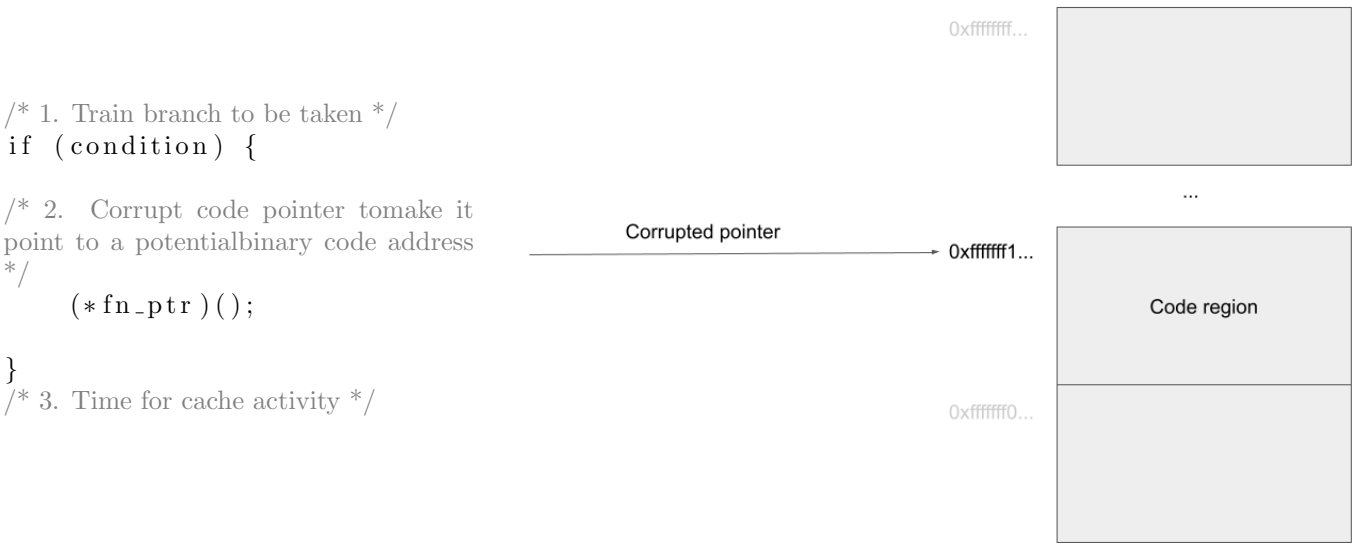

Figure 1: The attacker corrupts the code pointer and makes it point to an address which potentially contains the program's code region. When the he successfully guesses the code address, he will see activity in the cache.

Figure 1. To do so, he first trains the conditional branch predictor by repeatedly invoking it with a value which causes the branch to be taken. Then, he corrupts the code pointer, overwriting it with an address where he believes the binary was potentially loaded and flips the branch condition to cause speculative execution. Since the predictor was trained to take the branch, the corrupted pointer is speculatively followed, dereferencing a potentially invalid address. However, after speculation ends, no crash is caused, since the branch will follow another path in normal program execution. During transient execution, the CPU tries to fetch instructions to execute from the attacker controlled address. If the binary is indeed loaded at that address, the instructions are brought into the cache, else speculative execution stops. The attacker now uses a side channel to check if the cached was filled. If the address is indeed in the cache, the attacker successfully learns the address of the binary. If there was no activity in the cache, the attacker repeats the process using a new address until he eventually finds the correct address.

Using similar techniques, the attacker can locate other desired addresses in the address space, such as the address of data regions, specific objects in the data regions, or even specific code gadgets. After having all the necessary information, the attacker builds a payload and architecturally hijacks the control flow using the corrupted code pointer to mount an exploit and gain control of the program.

3 Preventing speculative probing attacks

Traditional Spectre mitigations would prevent speculative hijacking of an indirect control-flow instruction by hindering attempts to mispredict indirect branch predictors. Since speculative probing uses an architecturally corrupted function pointer, such mitigations are ineffective and the attacker is still able to speculatively hijack the control flow. Similarly, using the speculative primitives, speculative probing leaks information that allows the attacker to bypass conventional memory corruption mitigations, such as ASLR, that would normally hinder the attacker's attempts of exploiting a corrupted code pointer.

For my research project, I will seek to find ways to prevent speculative probing attacks using a compiler-assisted software solution. By implementing my solution as a compiler optimization pass, I aim to make it easy to deploy in commodity software.

Current solutions that could be effective against speculative probing either require preventing transient execution (e.g. using lfences), incurring a large performance penalty, or rely on newly deployed hardware-

assisted mitigations (e.g. Intel CET [18]) which are not yet available in most systems. Other proposed work requires hardware redesigns [16, 24] which are not easy to deploy in practice. An easy-to-deploy software solution without a significant performance overhead is thus necessary to prevent these types of attacks.

LLVM, a widely used compiler infrastructure toolchain [22], implements a pass that protects loaded values against certain Spectre variants [3]. This pass instruments conditional branch outcomes in a way that if the outcome of a conditional branch was misspeculated, the load address is masked such that no sensitive data is loaded during incorrect speculation.

By following a similar approach, I will locate and harden only the indirect control-flow influencing instructions that lie within the speculation window of conditional branches, which may load their target from a (potentially corrupted) code pointer. By avoiding instrumentation of unrelated loads, while also still allowing speculation on correct conditional branch predictions, I aim to keep a low performance overhead compared to solutions that would further restrict speculation.

Furthermore, I will explore different instrumentations that can achieve this result and compare the performance of each instrumentation, as well as other existing potential solutions that could mitigate speculative probing.

References

[1] AMD. July 2020. URL: https://developer.amd.com/wp-content/resources/Managing-Speculation-on-AMD-Processors.pdf.

[2] Claudio Canella et al. "A Systematic Evaluation of Transient Execution Attacks and Defenses". In: *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 249–266. ISBN: 978-1-939133-06-9. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/canella.

[3] Chandler Carruth. *Speculative Load Hardening*. URL: https://llvm.org/docs/SpeculativeLoadHardening.html.

[4] Crispin Cowan et al. "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks". In: *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*. SSYM'98. San Antonio, Texas: USENIX Association, 1998, p. 5.

[5] Enes Göktas et al. "Speculative Probing: Hacking Blind in the Spectre Era". In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. CCS '20. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 1871–1885. ISBN: 9781450370899. DOI: 10.1145/3372297.3417289. URL: https://doi.org/10.1145/3372297.3417289.

[6] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. ISSN. Elsevier Science, 2017. ISBN: 9780128119068. URL: https://books.google.com/books?id=cM8mDwAAQBAJ.

[7] Intel. Mar. 2018. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/runtime-speculative-side-channel-mitigations.html.

[8] Intel. Mar. 2018. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/single-thread-indirect-branch-predictors.html.

[9] Intel. Mar. 2018. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/indirect-branch-restricted-speculation.html.

[10] Paul Kocher et al. "Spectre attacks: Exploiting speculative execution". In: *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2019, pp. 1–19.

[11] Peinan Li et al. "Conditional Speculation: An Effective Approach to Safeguard Out-of-Order Execution Against Spectre Attacks". In: *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2019, pp. 264–276. DOI: 10.1109/HPCA.2019.00043.

[12] Fangfei Liu et al. "Last-Level Cache Side-Channel Attacks are Practical". In: *2015 IEEE Symposium on Security and Privacy*. 2015, pp. 605–622. DOI: 10.1109/SP.2015.43.

[13] Andrea Mambretti et al. *Bypassing memory safety mechanisms through speculative control flow hijacks*. Mar. 2020.

[14] Team PaX. URL: https://pax.grsecurity.net/docs/aslr.txt.

[15] Stephan van Schaik et al. "RIDL: Rogue In-Flight Data Load". In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019, pp. 88–105. DOI: 10.1109/SP.2019.00087.

[16] Michael Schwarz et al. "ConTExT: A Generic Approach for Mitigating Spectre". In: Jan. 2020. DOI: 10.14722/ndss.2020.24271.

[17] Hovav Shacham. "The Geometry of Innocent Flesh on the Bone: Return-into-Libc without Function Calls (on the X86)". In: *Proceedings of the 14th ACM Conference on Computer and Communications Security*. CCS '07. Alexandria, Virginia, USA: Association for Computing Machinery, 2007, pp. 552–561. ISBN: 9781595937032. DOI: 10.1145/1315245.1315313. URL: https://doi.org/10.1145/1315245.1315313.

[18] Vedvyas Shanbhogue, Deepak Gupta, and Ravi Sahita. "Security Analysis of Processor Instruction Set Architecture for Enforcing Control-Flow Integrity". In: *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*. HASP '19. Phoenix, AZ, USA: Association for Computing Machinery, 2019. ISBN: 9781450372268. DOI: 10.1145/3337167.3337175. URL: https://doi.org/10.1145/3337167.3337175.

[19] *Spectre Side Channels*. URL: https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/spectre.html.

[20] László Szekeres et al. "SoK: Eternal War in Memory". In: *2013 IEEE Symposium on Security and Privacy*. 2013, pp. 48–62. DOI: 10.1109/SP.2013.13.

[21] Mohammadkazem Taram, Ashish Venkat, and Dean 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*. ASPLOS '19. Providence, RI, USA: Association for Computing Machinery, 2019, pp. 395–410. ISBN: 9781450362405. DOI: 10.1145/3297858.3304060. URL: https://doi.org/10.1145/3297858.3304060.

[22] *The LLVM Compiler Infrastructure*. URL: https://llvm.org/.

[23] Jo Van Bulck et al. "LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection". In: *41th IEEE Symposium on Security and Privacy (S&P'20)*. 2020.

[24] Mengjia Yan et al. "InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy (Corrigendum)". In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '52. Columbus, OH, USA: Association for Computing Machinery, 2019, p. 1076. ISBN: 9781450369381. DOI: 10.1145/3352460.3361129. URL: https://doi.org/10.1145/3352460.3361129.