Preventing Speculative Probing Attacks

Neophytos Christou
neophytos_christou@brown.edu

Abstract

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. A certain class of these attacks, called speculative probing, allows an attacker to leak sensitive program data using Spectre-like primitives, by causing a corrupted code pointer to be transiently dereferenced via a speculatively-executed indirect branch instruction. Speculative probing attacks have severe security implications, since the attacker can use the disclosed information to bypass memory corruption mitigations and eventually mount an end-to-end exploit.

In this project, we a mitigation against speculative probing attacks. The mitigation leverages the inability of CPUs to speculatively execute instructions that rely on unresolved data dependencies. By artificially making data dependent on the outcome of preceding conditional branches, the mitigation prevents potentially corrupted code pointers from being dereferenced in the speculative domain. By only restricting speculative execution of potentially vulnerable instructions, the mitigation allows programs to maintain some performance benefits 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) [16], stack canaries [6] and non-executable memory can be found in most commodity software.

Another, more recent class of attacks is the Spectre family of attacks [12]. Using these types of attacks, adversaries can leverage speculative execution [8] to speculatively access sensitive program data, bring them into microarchitectural buffers and leak the data using side channels [14]. Mitigations against a lot of the Spectre family variants have been rolled out [2], but researchers keep introducing new ways of exploiting speculative execution [23, 22].

Researchers have so far treated Spectre attacks and memory corruption attacks as two separate domains. However, recent work [7, 15] has shown that memory corruption vulnerabilities can be combined with Spectre-like primitives to bypass both memory corruption and Spectre mitigations. In particular, a class of these types of attacks, namely speculative probing attacks, allow an adversary to speculatively dereference a corrupted code pointer and subsequently extract sensitive program information. Since the corrupted code pointer is only dereferenced in the speculative domain, the attacker circumvents memory corruption mitigations, which are only triggered when the offending instructions are executed architecturally. As a consequence, the attacker can use the leaked information as a means to bypass state-of-the-art memory corruption mitigations. In [7], the authors have demonstrated how an adversary, armed with a single memory corruption vulnerability, can bypass both standard and function granular KASRL to leak the programs code layout, discover ROP gadgets, leak the program's data regions and eventually mount an end-to-end exploit on the Linux kernel to gain root privileges. Adding the fact that these attacks can be carried out without crashing the attacked software, crash-sensitive, critical software such as operating system kernels become an attractive target.

Speculative probing attacks are not trivial to mitigate, since standard memory corruption mitigations are not effective when instructions are executing speculatively, while none of the deployed mitigations for speculative execution attacks is effective against speculative probing either. How can these attacks be effectively mitigated without sacrificing the performance benefits gained from speculative execution? In this work, we present a mitigation against speculative probing attacks. The core idea behind the mitigation is to use conditionally executed instructions (i.e., the CMOVcc in the x86 architecture), to mask the potentially vulnerable code pointers whenever they are speculatively dereferenced after a conditional branch misspeculation. By carefully placing conditionally executed instructions after every conditional branch that leads to a code pointer dereference, the program maintains its execution semantics when executing architecturally. We implement the mitigation as a compiler pass using the LLVM toolchain [1].

We make the following contributions:

- Study how speculative probing attacks can be mitigated by speculatively masking the value of potentially vulnerable code pointers.
- Implement a mitigation against speculative probing attacks as a compiler pass.
- Evaluate the performance of the mitigation using the SPEC2017 benchmarking suite.

2 Background

2.1 Spectre

Modern CPUs leverage various mechanisms, such as parallel and out-of-order execution, in order to avoid idle CPU cycles and maximize 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 various predictors to *speculate* on the data dependency and start speculatively executing instructions down the speculated path, 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 down the correct path. However, the *transient* instructions executed during speculation may still leave observable side-effects on the microarchitectural state of the CPU.

Spectre attacks [12] exploit the fact that, after a misspeculation, data brought in the CPU's microarchitectural buffers by the transiently executed instrucitons can be leaked. Even though there are multiple variants of Spectre [4], 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 speculation. 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 [3, 9] that mitigate many of the variants of the Spectre family of attacks by either preventing speculation [13, 21], hindering side channels [24] or preventing CPU predictors from being mistrained [10, 11].

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 [15] aim to abuse speculation to bypass conventional memory safety mechanisms. 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. [7] 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 [18], the attacker first needs to locate code region of the program, as seen in 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 speculative 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.

```
/* 1. Train branch to be taken */
if (condition) {

/* 2. Corrupt code pointer tomake
it point to a potentialbinary code
address */
      (*fn_ptr)();

}
/* 3. Time for cache activity */
```

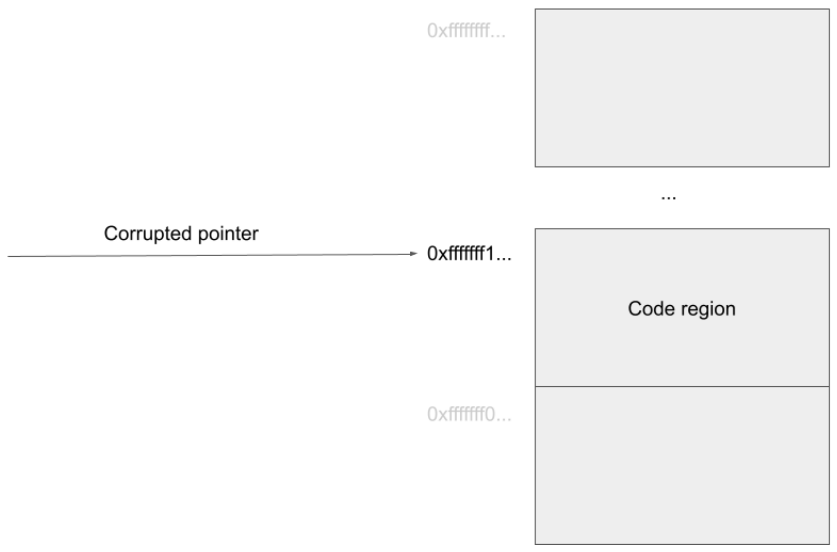

Figure 1: The attacker corrupts the code pointer and makes it point to an address which potentially contains the program's code region. When the he successfully guesses the code address, he will see activity in the cache.

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

Current solutions that could be effective against speculative probing either require preventing speculation (e.g., using lfences), incurring a large performance penalty, or rely on newly deployed hardware-assisted mitigations (e.g., Intel CET [19]) which are not yet available in most systems. Other proposed work requires hardware redesigns [17, 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 [1], implements a pass that protects loaded values against certain Spectre variants [5]. 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, we 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, we aim to keep a low performance overhead compared to solutions that would further restrict speculation.

Furthermore, we 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] The llvm compiler infrastructure. https://llvm.org/.

[2] Spectre side channels. https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/spectre.html.

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

[4] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 249–266, Santa Clara, CA, August 2019. USENIX Association.

[5] Chandler Carruth. Speculative load hardening.

[6] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*, SSYM'98, page 5, USA, 1998. USENIX Association.

[7] Enes Göktas, Kaveh Razavi, Georgios Portokalidis, Herbert Bos, and Cristiano Giuffrida. Speculative probing: Hacking blind in the spectre era. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 1871–1885, New York, NY, USA, 2020. Association for Computing Machinery.

[8] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. ISSN. Elsevier Science, 2017.

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

[10] Intel, March 2018.

[11] Intel, March 2018.

[12] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2019.

[13] Peinan Li, Lutan Zhao, Rui Hou, Lixin Zhang, and Dan Meng. 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)*, pages 264–276, 2019.

[14] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy*, pages 605–622, 2015.

[15] Andrea Mambretti, Alexandra Sandulescu, Alessandro Sorniotti, Wil Robertson, Engin Kirda, and Anil Kurmus. Bypassing memory safety mechanisms through speculative control flow hijacks, 03 2020.

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

[17] Michael Schwarz, Moritz Lipp, Claudio Canella, Robert Schilling, Florian Kargl, and Daniel Gruss. Context: A generic approach for mitigating spectre. 01 2020.

[18] 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, page 552–561, New York, NY, USA, 2007. Association for Computing Machinery.

[19] 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, New York, NY, USA, 2019. Association for Computing Machinery.

[20] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62, 2013.

[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, page 395–410, New York, NY, USA, 2019. Association for Computing Machinery.

[22] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *41th IEEE Symposium on Security and Privacy (S&P'20)*, 2020.

[23] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Ridl: Rogue in-flight data load. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 88–105, 2019.

[24] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. Invisispec: Making speculative execution invisible in the cache hierarchy (corrigendum). In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 1076, New York, NY, USA, 2019. Association for Computing Machinery.