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 present 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 introducing data dependencies between vulnerable indirect branches and 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, programs can 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 [18]. A lot of these mechanisms, such as address space layout randomization (ASLR) [16], stack canaries [5] 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 [7] 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 [21, 20].

Researchers have so far treated Spectre attacks and memory corruption attacks as two separate domains. However, recent work [6, 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 [6], 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. In this work, we present a mitigation against Speculative Probing attacks. The core insight behind the mitigation is that while modern CPUs can speculate on the outcome of control-flow instructions, instructions that have unresolved data dependencies cannot be executed, even in the speculative domain. By introducing artificial data dependencies on potentially vulnerable indirect branches, the mitigation prevents them from being speculatively executed until the outcome of all conditional branches that were speculated upon is resolved. We implement the mitigation for the X86 architecture as a compiler pass using the LLVM toolchain [1].

We make the following contributions:

- Discuss how Speculative Probing attacks can be mitigated by artificially introducing data dependencies.
- 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 Speculative execution and Spectre attacks

Modern CPUs leverage various mechanisms such as out-of-order execution, instruction-level parallelism, etc., in order to avoid idle CPU cycles and maximize performance. One of these mechanisms is speculative execution. When the CPU tries to execute a control-flow instruction which relies on an unresolved data dependency, instead of stalling the pipeline until the dependency is resolved, the CPU will *speculate* on the outcome of the control-flow instruction and start *speculatively executing* instructions down the guessed path. If the CPU has correctly predicted the outcome of the control-flow instruction, 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 or tamper with some CPU predictor to cause the CPU to later speculatively execute an attacker-chosen piece of code. Second, the attacker triggers speculation by causing a control-flow instruction to be executed before its dependencies are resolved. The CPU will use the attacker-influenced predictor and start speculatively executing attacker-chosen instructions. These instructions will 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, 19], hindering side channels [23] 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. [6] 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. 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.

By leveraging 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 */
```

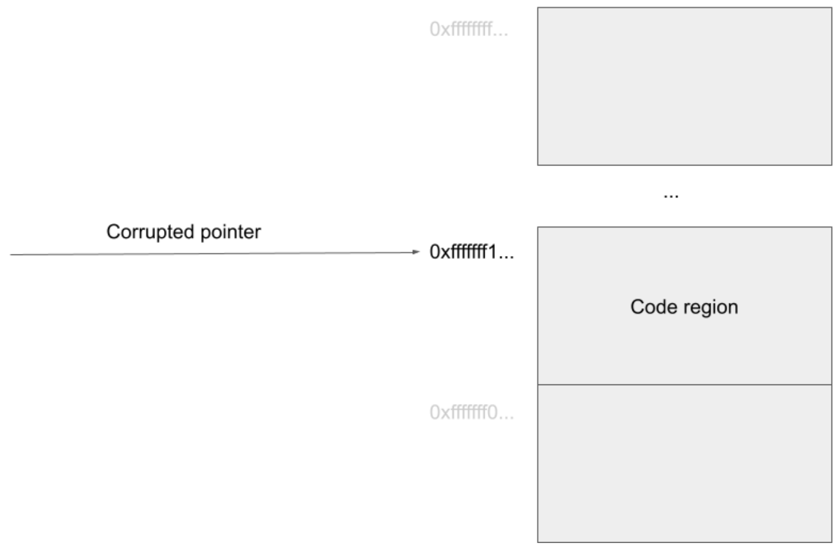

Figure 1: The attacker corrupts the code pointer and makes it point to an address which potentially contains the program's code region. When the he successfully guesses the code address, he will see activity in the cache.

2.4 Current approaches for preventing Speculative Probing attacks

Traditional Spectre mitigations try to prevent speculative hijacking of an indirect control-flow instruction by hindering attempts to tamper with indirect branch predictors [10, 11]. However, 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 described 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.

Currently, the main approach to prevent Speculative Probing would be to use a serializing instruction (e.g., LFENCE) to stop speculative execution before dereferencing code pointers. Stopping speculative execution using LFENCEs has been the recommended way to prevent any type of speculative execution attacks by CPU vendors [8]. However, this approach can incur large performance penalties, since it completely removes the CPUs ability to speculatively execute instructions following the LFENCE.

2.5 Conditional instructions in the x86 architecture

Conditional instructions is a family of instructions which have different outcome depending on whether or not a condition is met. In the x86 architecture, this condition is encoded in a special register called *rflags*, which is implicitly read by every conditional instruction. Each conditional instruction is associated with a *condition code*, which indicates which bits in the *rflags* register the instruction should read in order to determine its outcome.

The most common family of conditional instructions in the x86 architecture are the conditional branch instructions. Conditional branches jump to the address denoted by their operand if the condition associated with their condition code is satisfied, else the jump is skipped. For example, in Figure 2, the conditional jump (*je*) instruction will jump to the instructions located 16 bytes after the conditional jump if the *zero flag* bit in the *rflags* register is set, else the CPU will execute the instruction following the conditional jump in the instruction stream.

Another class of x86 conditional instructions are the *conditional move* instructions. Similarly to conditional branches, conditional moves rely on the certain bits of the *rflags* register, determined by the conditional move's condition code, to determine whether the value held in their source operand will be copied into their destination operand. The source operand can only be a register and the destination operand can either be a register or a memory locatoin. For example, in Figure 3, the conditional move (*cmove*) instruction will copy the value held in the *r11* register into the *r12* register if the *zero flag* bit in the *rflags* register is set, else the value of *r12* remains unchanged.

```
je      0x10
```

Figure 2: Conditional branch instruction

```
cmove  %r11, %r12
```

Figure 3: Conditional move instruction

3 Threat model

Our goal is to prevent an attacker to use Speculative Probing techniques, as described in Section 2.3. We assume that the attacker is co-located on the same machine where the vulnerable program is running and can observe the state of the cache using a microarchitectural side channel. The target program contains a memory corruption vulnerability which allows the attacker to arbitrarily corrupt memory. The program also contains an indirect branch which conditionally dereferences a code pointer based on the outcome of a preceding conditional branch. Both the code pointer and the outcome of the conditional branch are controlled by the attacker, whose goal is to use Speculative Probing to bypass randomization-based memory corruption mitigations, such as ASLR.

4 Mitigation approach

4.1 Low-level attack example

Figure 4 presents an assembly code snippet which corresponds to the C code of the vulnerable program as described in Section 2.3. Here, we step through execution of the program during the attack and explain what takes place on the assembly level.

Execution during the attack looks as follows:

1. First, the CPU tries to execute the compare (*cmpl*) instruction on line 1, which implicitly sets some bits in the *rflags* register. However, the instruction is data dependent on the value of the *rax* register, which has not yet been resolved. As a result, the compare instruction cannot yet be executed and the value of *rflags* is now unknown.

2. Next, the CPU executes the conditional branch (*je*) on line 2. Since conditional branches are data dependent on the *rflags* register, the true outcome of the branch is unknown and the CPU has to speculate.

3. Because of the mistraining the attacker performed on the CPU predictor, the CPU will speculate that the indirect branch on line 3 needs to be executed. Assuming that the *rcx* register contains the function pointer which was corrupted by the attacker, the CPU will speculatively dereference it and try to fetch instructions from the attacker-chosen address.

4. Eventually, the value of the *rax* register is resolved. The CPU then also resolves the value of *rflags* by executing the compare instruction. Finally, it resolves the correct outcome of the conditional branch, stops speculation and jumps to the correct target of the conditional branch on line 5.

The goal of our mitigation is to prevent the CPU from speculatively dereferencing the code pointer, as described in step 3.

4.2 Artificial data dependencies

The main observation we base our mitigation on is that while the CPU can predict the outcome of control-flow instructions (e.g., conditional branches) and start speculative execution, unresolved data dependencies cannot be predicted. As a result, the CPU has to stall the execution of encountered instructions which rely on unresolved data dependencies, even in the speculative domain.

When an attacker carries out a Speculative Probing attack, he relies on the fact that when execution reaches a conditional branch preceding an indirect branch which dereferences a corrupted function pointer, an unresolved data dependency will force the CPU to speculate on the outcome of the conditional branch and speculatively dereference the pointer.

The core idea behind our mitigation is to identify all indirect branches that can potentially be speculatively executed due to a conditional branch misprediction and artificially make the value of the code pointer data dependent on the same unresolved data that caused the conditional branch to be speculated. As a result, the value of the code pointer will remain unknown until the CPU resolves the data dependency, which results to also resolving the correct outcome of the preceding conditional jump, thus stopping speculative execution.

4.3 Introducing Data Dependencies with Conditional Moves

We leverage conditional moves, described in Section 2.5, as the main building block for our mitigation. Conditional moves allow us to introduce a common data dependency between code pointers dereferenced by indirect branches and their preceding conditional branches.

We present a code snippet hardened by our mitigation in Figure 5. Similarly to 4.1, we step through program execution during the attack and describe how our mitigation prevents the attacker from speculatively dereferencing the corrupted function pointer.

Execution during the attack with our mitigation in place looks as follows:

1. First, in lines 1-2, two registers are initialized with the values 0 and -1. These registers will later on be used as the operands of the conditional move instruction. Henceforth, we will be referring to the first register (*r12*) as the *state* register and the second register (*r11*) as the *poison* register. The reasoning behind initializing the registers with these specific values is discussed in 4.4 and 4.5.

2. Next, similarly to 4.1, the CPU tries to execute the compare instruction in line 3, but fails to do so because of the value of *rax* being unresolved.

3. The outcome of the conditional branch in line 4 is speculated. Because of the mistrained predictor, the CPU speculates that it needs to not take the branch and execute the next instruction.

4. The CPU now tries to execute the conditional move (*cmove*) on line 5. However, as mentioned in Section 2.5, the conditional move is also data dependent on the *rflags* register. Thus, the instruction cannot yet be executed and the value of the state register is now also unknown to the CPU.

5. The final instruction introduced by our mitigation is the OR instruction on line 6, which uses the state register as the source operand and the code pointer inside the *rcx* register as the destination operand. Since the value of the state register is unknown, this OR instruction can also not be currently executed. As a result, the value of the code pointer is now also unknown.

6. Finally, when execution reaches the indirect call on line 7, the code pointer cannot be speculatively dereferenced since its value is now unknown, thus stopping the attempt of the attacker to leak program information.

7. Eventually, the value of the *rax* register is resolved and the CPU executes the correct target of the conditional jump, stopping speculation.

4.4 Keeping regular program execution intact

Since the instrumentation should not have any effect on the (non-speculative) execution of the program, the masking operation should not alter the value of the code pointer, i.e., the source operand of the *or* instruction should always have the value 0 when executing architecturally. This is achieved as follows: first, the register that will be used as the destination operand for the conditional moves is initialized with zero at the beginning of the function (Figure 5, line 2). Second, the conditional move should never execute architecturally, such that it never changes the value of the masking register. This can be guaranteed by carefully choosing the condition code of the conditional move. If the conditional move is inserted in the fallthrough edge of the conditional jump, its condition code should match that of the jump (e.g., *cmove* on line 9 matches the condition code of the *je* on line 6). Else, if it inserted on the "taken" edge of the jump, it should have the opposite condition code.

4.5 Poisoning the Code Pointer

We have determined what the value of the masking register should be in the non-speculative domain, but we still need to decide what value the conditional move should move into the masking register when the conditional jump was mispredicted and the conditional move is executed speculatively. At first glance, this value seems irrelevant; the conditional move should never be executed when a misspeculation occurs due to the fact that resolving the value of *rflags* also resolves the correct direction of the conditional branch. However, there is a caveat: the ordering of the instructions after *rflags* is resolved is not guaranteed. The CPU can re-order the instructions and execute the conditional move (and, in turn, dereference the code pointer) before executing the correct outcome of the conditional branch and squashing speculative execution. As a result, this still gives a window for the attacker-controlled pointer to be dereferenced speculatively. To eliminate this possibility, the masking instruction should replace the value of the potentially-corrupted code pointer with a value that is guaranteed to not point to any code. We achieve this by initializing the register used as the source operand with a "poisoning" value (i.e., -1), as seen in line 2 of Figure 5. Paired with the fact that the condition code of the conditional moves was chosen such that it is the opposite of a valid control-flow (i.e., can only be reached if the conditional branch was misspeculated), the masking register is guaranteed to have this poisoning value when the conditional move gets executed due to a misspeculation. Consequently, even if the ordering of the instructions gets switched up, the value of the code pointer will be poisoned and the attacker-controlled code will not be dereferenced.

```
cmpl   $0x0, %rax
je     no_call
callq  *%rcx
.no_call:
    ...
```

Figure 4: Vulnerable code snippet. The indirect call can be speculatively dereferenced by training the CPU predictor to not take the conditional jump.

```
mov     $0x0, %r12
mov     $0xffffffffffffffff, %r11
cmpl    $0x0, %rax
je      no_call
cmove   %r11, %r12
or      %r12, %rcx
callq   *%rcx
.no_call:
...
```

Figure 5: Hardened code snippet. The indirect call is now data dependent on the *rflags* register and cannot be executed speculatively.

5 Implementation

We implemented the mitigation as an LLVM machine-function pass for the X86 architecture (≈1100 LoC). The pass runs right before the register-allocation phase of the compiler pipeline.

The pass runs on each function and performs the following actions:

1. Collects all indirect branches that be conditionally executed (i.e., lie on a path that can be reached from an edge of a conditional branch). If there are none, it returns without modifying the function.

2. Collects all conditional branches that lie on the path of the collected indirect branches.

3. Initializes a register with a poisoning value (i.e., -1) to be used as the poisoning register.

4. Initializes a register with a neutral value (i.e., 0) to be used as the masking register.

5. Inserts a conditional move after every edge of the collected conditional branches that lie within a path of a vulnerable indirect branch. The condition required to execute the conditional move is picked such that the move will never execute along a valid control-flow path.

6. Hardens all vulnerable indirect branches by masking them with an OR instruction, using the masking register as the source operand and the register holding the code pointer to be used by the indirect branch as the destination operand.

The assumed threat model, as described in Section 3, allows an attacker to arbitrarily corrupt memory. As a result, if the state register is spilled in memory, the attacker could corrupt it and change is value, which would allow him to bypass our mitigation. In order to avoid memory spilling, our pass reserves a general-purpose register to be used only as a state register.

In order to verify the correctness of the pass and ensure that no modifications were made to the instrumentation during the later stages of the compilation pipeline, we leveraged the Egalito binary rewriting tool [22] to perform static analysis on the instrumented binary. The analysis pass (≈550 LoC) follows a similar approach to the compiler pass. It first disassembles the binary, collecting the vulnerable indirect branches and conditional branches in their path for each function. It then ensures that the masking and poisoning registers are properly initialized, conditional moves are inserted on every necessary conditional branch edge and every vulnerable indirect branch is properly masked.

6 Evaluation

6.1 Performance evaluation

We evaluated the performance overhead introduced by our mitigation using the SPEC2017 benchmarking suite.

6.1.1 Experimental setup

Our experiments were run on a Linux machine running Debian v11 (Bullseye), on a 16-core Intel Xeon W-2145 3.70GHz CPU and 64GB of RAM.

6.1.2 Comparing against LFENCE mitigation

We compared our approach against mitigating the attack by stopping speculative execution using a serializing instruction, which is the recommended way of preventing speculative execution attacks, as described in Section 2.4. Specifically, we modified our compiler pass to insert the x86 LFENCE instruction at the beginning of every basic block containing an indirect branch and is preceded by a conditional branch. The main advantage of our approach is that it does not block speculative execution and only delays the CPU from executing the data-dependent conditional branch, but still allows speculative execution of any other non-data-dependent instruction preceding the branch.

6.1.3 Results

As can be seen in Table 1 our approach introduces up to a 6.6% overhead, whereas the LFENCE approach introduces up to 40% overhead.

Benchmark	Our Mitigation	LFENCE Mitigation
600.perlbench_s	≈0%	≈0%
602.gcc_s	2.1%	1.85%
605.mcf_s	≈0%	40.1%
619.lbm_s	≈0%	≈0%
625.x264_s	≈0%	7.62%
638.imagick_s	6.61%	2.77%
644.nab_s	≈0%	≈0%
657.xz_s	1.46%	0.73%
Range	0 - 6.6%	0 - 40%

Table 1: Overhead of the mitigations over uninstrumented baseline.

We observe that for some benchmarks (i.e. 602.gcc_s, 638.imagick_s and 657.xz_s), the LFENCE approach slightly outperforms our approach. We hypothesize that the reason for this is the following: overall, our approach requires more instructions to be introduced in the binary: indirect branches need to be masked with an OR instruction and CMOVs need to be inserted at the edges of preceding conditional branches. Furthermore, it requires a register to be reserved (Section 5), which increases register pressure in the program and cause more memory spills. On the other hand, the LFENCE approach only inserts one LFENCE per indirect branch. When LFENCEs are inserted in blocks containing a large number of instructions, the performance penalty is more observable, since it prevents speculative execution of all the instructions in the block. However, when the basic blocks containing LFENCE have a small number of instructions, the performance cost of stopping speculation is lower and the extra instructions and register pressure introduced by our mitigation dominate the performance benefits gained from not stopping speculative execution.

6.2 Security evaluation

In order to evaluate whether our mitigation successfully protects against speculative probing attacks, we built a small proof-of-concept code snippet simulating the attack as described in the original Speculative Probing paper [6] and used it to test our instrumentation.

The main parts of the snippet can be seen in Figure 6. A vulnerable struct, v, holds a function pointer (line 8), which is dereferenced if a flag is set (lines 19-22). The condition on line 19 is first called multiple times with the flag set, such that the CPU predictor will be trained to take the branch and dereference the code pointer. In the final iteration, the flag is set to 0 and the value of the function pointer is replaced with the address of a gadget which uses a secret value to access an attacker-controlled side-channel array, simulating a Flush and Reload gadget. Lines 11-17 flush the flag from memory, in order to increase the speculation window, bring the secret value into the cache and flush the attacker-controlled array to prepare for the Flush and Reload side-channel. When execution reaches line 19, the CPU will start speculating until the value of the flag is retrieved from memory. Since the conditional branch predictor was trained to take the branch, the code pointer will be dereferenced speculatively and bring the secret value in the cache. The attacker performs a cache-access timing measurement on line 26 to determine whether the secret value was brought into the cache.

We run this code snippet with and without our mitigation. When the code is not protected, we can measure hits in the cache, signalling that the code pointer was speculatively dereferenced and accessed the secret data. When applying our mitigation, we can no longer measure any hits.

```
1  /* access_secret accesses a 'secret' byte, simulates leaking gadget attacker
2   * wants to speculatively dereference */
3  int (*fptrs[6]) () = {&hello, &hello, &hello, &hello, &hello, &access_secret};
4  int flags[6] = {1, 1, 1, 1, 1, 0};
5  /* Repeat multiple times to train the conditional branch */
6  for (int i = 0; i < TRAINING_ITERS; i++) {
7      /* Load function pointer */
8      v->func_ptr = fptrs[i];
9      if (i % (TRAINING_ITERS - 1) == 0) {
10         /* Flush the flags to cause speculative execution */
11         _mm_clflush(&flags[i]);
12         /* Bring secret into the cache to avoid stalling when speculating */
13         dummy ^= secret;
14         /* Flush the attacker-controlled side-channel array
15         we will use to leak the secret and fence */
16         _mm_clflush(side_channel_arr);
17         _mm_mfence();
18     }
19     if (flags[i]) {
20         /* Dereference the pointer */
21         (*v->func_ptr)();
22     }
23 }
24 /* Time the side-channel array slot. If the secret was accessed
25 speculatively, this will be a hit */
26 hit_time = probe(side_channel_arr);
27 if (hit_time < CACHE_THRESHOLD)
28     /* Record hit */
```

Figure 6: Code snippet simulating Speculative Probing . The attacker trains the conditional branch on line 19 to be taken, then flips the condition and corrupts the code pointer with an address pointing to a secret-leaking gadget.

7 Conclusion

To conclude, we introduced an mitigation which prevents Speculative Probing attacks by introducing artificial data dependencies to prevent the CPU from speculatively dereferencing code pointers. We implemented our approach as an LLVM compiler pass and evaluated its security and performance overhead. Our approach introduces up to 6.6% overhead on the evaluated benchmarks, compared to up to 40% when mitigating with a more naive approach.

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] 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.

[6] 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.

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

[8] Intel. Intel analysis of speculative execution side channels. https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf.

[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. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/single-thread-indirect-branch-predictors.html, March 2018.

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

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P. Kemerlis. Egalito: Layout-agnostic binary recompilation. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 133–147, New York, NY, USA, 2020. Association for Computing Machinery.

[23] 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.