Attacks using speculative execution of processor instructions

Alexej Beňuš
Faculty of information technology
Brno University of Technology
Brno, Czech Republic
xbenus01@stud.fit.vutbr.cz

***Abstract*—This paper investigates the security implications of speculative execution in modern CPU architectures. Techniques such as out-of-order execution, branch prediction, and instruction pipelining are essential for achieving high performance, but they also introduce subtle and dangerous side effects. We begin by explaining how these mechanisms work at the microarchitectural level, with a particular focus on the promises of memory isolation and the risks of transient execution.**

We then examine the major classes of speculative execution vulnerabilities, beginning with Meltdown and Spectre, and progressing to more recent and advanced attacks such as TikTag, Downfall, and GhostRace. These vulnerabilities exploit microarchitectural side channels to leak sensitive data across isolation boundaries, often bypassing traditional software and hardware protections.

Throughout the paper, we assess the effectiveness of the industry's mitigation strategies, from kernel page-table isolation and microcode updates to serialization instructions and compiler-level defenses. Our analysis concludes by considering the future of CPU security and the need for architectural changes that prioritize isolation and side-channel resistance without sacrificing performance.

***Index Terms*—speculative execution, Meltdown, Spectre, out-of-order execution, side-channel attack, transient execution, CPU microarchitecture, memory isolation, kernel space, processor security**

I. Introduction

In the pursuit of higher computational performance, modern processors have adopted increasingly aggressive techniques such as pipelining, out-of-order execution, and speculative execution. These features enable CPUs to execute instructions more efficiently and increase throughput, but they also introduce complex behaviors that may interact poorly with the guarantees of software and hardware isolation.

This paper explores the architecture of modern CPUs, focusing particularly on the aspects of speculative execution and its associated vulnerabilities. It begins with an overview of the core microarchitectural components, including the instruction pipeline, branch prediction, and out-of-order execution, and then introduces the concept of memory isolation, which is foundational to operating system security.

We then present a detailed chronology of how speculative execution vulnerabilities such as Meltdown and Spectre were discovered, their impact, and how the industry responded to these revelations. The analysis continues with an examination of more recent and sophisticated transient execution vulnerabilities such as TikTag, Downfall, and GhostRace, showcasing that speculative execution attacks remain an evolving threat.

Ultimately, this work aims to provide a comprehensive view of how performance-driven CPU features have become both a strength and a liability, and what strategies exist to mitigate their associated risks.

II. Compute processor unit architecture

A. The Execution Pipeline in Modern CPUs

Modern CPUs use a pipelined architecture. This architecture uses several stages of execution of instructions, and all of them work simultaneously. This is great for improving performance as using paralel stages gives us bigger instruction throughput.

For simplification, here is a view of mostly used stages:

- **Fetch (IF)**: The CPU retrieves the next instruction from memory.
- **Decode (ID)**: The instruction is interpreted and translated into signals or microoperations.
- **Execute (EX)**: Arithmetic or logical operations are carried out.
- **Memory Access (MEM)**: Data may be read from or written to memory.
- **Writeback (WB)**: The results are written back to the register file or memory.

TABLE I
Pipeline Execution Timing Diagram

	Clock Cycle						
Instr. No.	**1**	**2**	**3**	**4**	**5**	**6**	**7**
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX

B. The Promise of Memory Isolation

Intel promised [2] that the CPU maintains the principle of memory isolation. That means that processes should run in an isolated environment, separated from the memory of the operating system kernel and other processes. This means that no process should be able to reach the memory region

of another process or kernel. These guarantees are critical in preventing one application from interfering or spying on another.

Processor vendors historically assumed that speculative execution did not pose a security threat because any speculatively executed instructions that violated privilege rules would be rolled back before their effects became visible to the software. In other words, the belief was that the architectural state of the CPU remained secure, regardless of what occurred during speculation.

C. Speculative Execution and Security

With parallel implementation of instructions, there comes a problem with execution of instructions in the speculative branches of the code. `if` statement breaks the code into two or more different branches of execution. Usual implementation is that the CPU guesses which branch will the code go to and it will start computing predicted instructions.

When the prediction is incorrect, the pipeline is flushed - that means that all the instructions that were speculatively computed are erased, and the CPU has to begin computation of new branch of code. However, while the architectural state is restored, microarchitectural traces, such as changes in cache contents, may remain. These traces can be probed using side-channel techniques, allowing attackers to infer the data accessed speculatively [1].

This CPU optimization can be very useful in things such as loops, where most of the time you jump back to the start of the loop. When the prediction is successful, time is saved; otherwise the pipeline has to be flushed.

D. Out-of-Order Execution

While speculating which branches are going to be computed, we also need to know which data will be up to date with execution to prevent data race conditions, etc. Because of this modern CPUs are implemented with out-of-order execution of instructions. This technique allows processors to execute instructions out of programmed order if possible without changing the result.

This feature is possible by analyzing data dependencies at runtime. If instruction data are dependent on previous instructions or are waiting for memory access, another instruction that is ready can be executed to save time. Eventually results of the instructions are reordered before being commited to ensure that the behavior of the program remains the same. To manage this, CPUs use structures such as reservation stations, the reorder buffer (ROB), and register renaming to track instructions and their dependencies. [10]

Although this technique improves performance, it also introduces a subtle security risk. Instructions that execute early may speculatively access data that they would not be allowed to see under normal conditions. While the CPU discards the architectural results of these speculative executions if they turn out to be invalid, microarchitectural side effects—such as changes in cache state—can persist and be exploited through side channels.

E. Branch Prediction

When using loops, `if` statements and functions, new branches of computation are created. To improve performance, the CPU starts to predict which branch it is going to compute. This technique is used to soften the calculation of the wrong branch.

CPU needs to decide which branch is the correct one to compute. If it computes the wrong branch, there is going to be some stall in the pipeline that wastes cpu cycles. There are many strategies to solve this problem.

The most simple are static predictors. These static predictors were used in past - for example, jumping backward would always start computation of the jump, but jumping forwards would never compute the jump.

Modern procesors use dynamic predictors that are based on the historical pattern of execution. They use a branch history table to store the results of previous branches and use that information to make the decision as accurate as possible. This branch history table can be realized, for example, using one- and two-bit predictors. These bits are used and are always updated with the last value of the jump. If code jumped, then the bit is set to logical one so that the next time this jump is evaluated, the CPU will start computation of the jump section of the code.

F. Flush+Reload Cache Side-Channel Attack

This technique is a cache-based side-channel attack that exploits the timing differences between memory accesses served from the CPU cache versus main memory. It enables an attacker to monitor the memory accesses performed by a victim process with granuality at the level of individual memory lines. [9]

The attack consists of three main steps:

1) **Flush:** The attacker uses the `clflush` instruction to erase a specific memory from the cache. This will make any subsequent access to that memory will incur a cache miss and be fetched from main memory, resulting in a measurable delay.
2) **Victim Access:** The victim process executes and may speculatively or legitimately access the flushed memory line, causing it to be reloaded into the cache. This step is unobservable to the attacker directly.
3) **Reload and Measure:** The attacker accesses the same memory line and measures the access latency using a high-resolution timer (e.g., `rdtsc`). If the access is fast, it implies that the data is already in the cache, indicating that the victim accessed it. If it is slow, the victim did not access it.

Flush+Reload is particularly effective because it provides high spatial and temporal resolution, allowing it to distinguish access at the level of individual cache lines with very low noise.

III. Discovery of Vulnerabilities (2018)

In January 2018 came an announcement about two of the biggest security flaws found to date - Spectre and Meltdown

[3]. While Meltdown "only" affected all Intel x86 CPUs and some high-performance ARM designs, Spectre affected all of the processors that use speculative execution. This destroyed the false feeling of safety guaranteed by the processor manufacturers.

A. Meltdown

The Meltdown vulnerability was independently discovered by three separate researchers teams in mid-2017.

- Werner Haas and Thomas Prescher (Cyberus Technology)
- Daniel Gruss, Moritz Lipp, Stefan Mangard, Michael Schwarz (Graz University of Technology)
- Jann Horn (Google Project Zero)

"Meltdown is a novel attack that allows overcoming memory isolation completely by providing a simple way for any user process to read the entire kernel memory of the machine it executes on, including all physical memory mapped in the kernel region." [4]

1) Technical Overview: Meltdown exploits a design flaw in the out-of-order and speculative execution of instructions, to bypass memory protection mechanisms enforced by the hardware of the CPU.

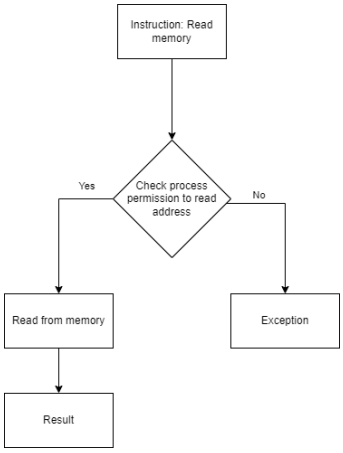

Fig. 1. How memory should be accessed

Meltdown targets the mechanism where the operating system maps kernel memory into the address space of every user process (for performance reasons), but with access permissions set to supervisor-only. The assumption was that the hardware would strictly enforce access rights, but Meltdown demonstrated that speculative execution could bypass this control transiently.

Meltdown purposely tries to execute an instruction read in the memory to which it has no access. The CPU detects this try and does not allow reading on that memory. But that is the issue. Intel CPUs check the permision only after speculatively fetching the data. Now, when permission is checked, the pipeline is flushed, but the side effects, such as cache state changes, can still be measured to infer the values that were speculatively accessed even after the flush.

Using a side-channel technique, such as cache timing analysis using the Flush+Reload version, the attacker can access

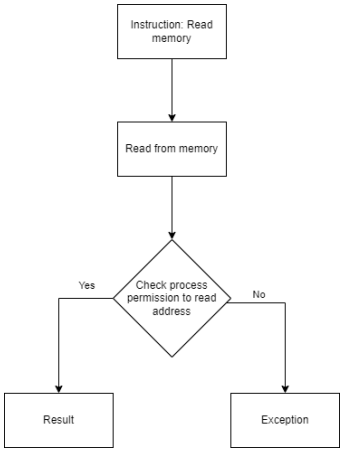

Fig. 2. How memory is accessed

every byte of the cache. Using this procedure, an attacker can map the whole kernel memory.

"The Flush+Reload attack has been used for attacks on various computations, e.g., cryptographic algorithms, web server function calls, user input, and kernel addressing information." [4]

2) Impact: Saying that the impact was severe would be an underestimation. Meltdown affected a wide range of Intel processors and some ARM cores. AMD processors were largely immune due to architectural differences in how they handle memory access checks during speculative execution.

It broke one of the most fundamental security guarantees of modern operating systems: that user-space processes cannot read kernel-space memory. With Meltdown, any unprivileged process running on an affected machine could potentially access arbitrary data belonging to the operating system or other processes, bypassing all conventional privilege boundaries.

The vulnerability not only revealed a design flaw in performance optimizations, but also exposed the systemic risks of relying on speculative execution without rigorous security validation.

B. Spectre

The Spectre vulnerability was discovered independently in 2017 by several researchers, including:

- Paul Kocher (Independent)
- Jann Horn (Google Project Zero)
- Daniel Genkin, Mike Hamburg, Yuval Yarom
- Daniel Gruss, Moritz Lipp, Stefan Mangard, Michael Schwarz (Graz University of Technology)
- Werner Haas, Thomas Prescher (Cyberus Technology)

"Spectre attacks induce a victim to speculatively perform operations that would not occur during correct program execution and that leak the victim's confidential information through a side channel to the adversary." [5]

1) Technical Overview: This attack uses different method than Meltdown. While meltdown targets specific hardware problem, Spectre uses speculative execution of instructions,

particularly the way they predict and speculatively execute conditional or indirect branches.

Spectre is not a single vulnerability but a class of attacks. The two most notable variants presented in the original publication are as follows [5]:

- **Spectre v1: Bounds Check Bypass** — This variant exploits the CPU's speculative execution past conditional branches, such as array bounds checks. The attacker deliberately mistrains the branch predictor to believe that a bound check will pass, even when it should not. As a result, the processor speculatively executes instructions that access memory beyond the intended bounds of an array or buffer. Although the invalid access is eventually discarded once the misprediction is resolved, the speculatively accessed data may have been brought into the cache. The attacker then uses a cache-based side channel technique, typically Flush+Reload, to infer the values of the accessed data. Since these accesses occur within the victim's own execution context.
- **Spectre v2: Branch Target Injection** — In this more sophisticated attack, the adversary manipulates the contents of the Branch Target Buffer (BTB). By carefully executing branch instructions in their own address space, the attacker "poisons" the BTB so that a subsequent indirect branch in the victim process is mispredicted to jump to an attacker-chosen location, typically a gadget sequence in the victim's memory. These gadgets consist of benign instructions that can be repurposed during speculative execution to leak secrets via a side channel (e.g., loading data into cache depending on secret values). This variant requires cross-process interference with the branch prediction unit and is more difficult to exploit reliably.

In practice, Spectre enables attackers to read array of arbitrary memory from the same address space, including private data from other functions or security domains. [5]

2) Impact: Spectre revealed a previously unknown systemic security flaw in virtually all high-performance CPUs from Intel, AMD, and ARM.

Spectre types of attack work entirely within the rules of the system. They do not rely on hardware bugs or illegal memory accesses. They exploit the fact that speculative paths can operate on sensitive data even though the sensitive data is never committed to the architectural state.

The scope of the attack was enormous. Demonstrations showed how even JavaScript running in a browser tab could potentially access data from the browser's internal memory, violating site isolation. Proof-of-concept implementations were shown to leak kernel memory using eBPF in Linux, and even to work cross-process through branch poisoning.

C. Meltdown vs. Spectre

Meltdown and Spectre both exploit the speculative execution capabilities of modern processors to leak sensitive information via side channels, but they differ in both mechanism and scope.

While both of them work with CPU performance optimization techniques, they work differently. Meltdown exploits a hardware mistake of most of Intel CPUs and some high performance ARM cores that do not check the permission of addresses in time, while Spectre type of attacks uses branch speculation which is a feature and not a bug itself. Spectre is more difficult to exploit, but also significantly harder to mitigate. It affects virtually all modern high-performance processors, including those of Intel, AMD, and ARM.

TABLE II
COMPARISON OF MELTDOWN AND SPECTRE VULNERABILITIES

Characteristic	Meltdown	Spectre
Type of Attack	Out-of-order execution bypasses memory protection	Speculative execution with branch prediction manipulation
Privilege Violation	Yes – reads kernel memory from user space	No – exploits legal speculative paths
Affected CPUs	Primarily Intel, some ARM	Intel, AMD, ARM
Exploitation Complexity	Relatively straightforward	More complex and context-dependent
Scope	User space → Kernel space	Same-process, inter-process, sandbox escape, etc.
Mitigation Difficulty	OS-level fixes (e.g., KPTI)	Software, compiler, and hardware changes
Number of Variants	One main variant	Multiple variants (v1, v2, RSB, STL, etc.)

IV. INDUSTRY RESPONSE AND MITIGATIONS

Unlike traditional software bugs, these attacks use design of high-performance CPUs.

A. Meltdown Fixes

1) Kernel Page-Table Isolation (KPTI): This technique was introduced in 2016 under the name KAISER (Kernel Address Isolation to have Side-channels Efficiently Removed) [4], that is, before the discovery of Meltdown and Spectre. KPTI separates user-space and kernel-space page tables entirely. Traditionally for performance reasons, this is not the case as operating systems map the kernel into every precess's address space, albeit protected by privelege bits. With KPTI kernel memory is completely unmapped when executing a user process. When an system call or interrupt, there is a context switch to a separate page table that includes kernel mappings.This prevents speculative access to kernel memory from user-space code, even if the CPU speculatively executes past a protection fault.

KPTI was quickly adopted by major operating systems, including Linux, Windows, and macOS. However, it introduced measurable performance overhead, especially in workloads that involve frequent user-kernel transitions.

2) Microcode-Based Constraints: In addition to software patches, CPU vendors issued microcode updates to add additional speculation barriers and control mechanisms. These updates enhanced the processor's behavior to reduce the impact of speculative execution following privilege faults. For example, certain instructions could now be marked as speculation barriers, preventing speculative execution across security boundaries. [11]

These microcode changes did not fix the vulnerability directly, but they complemented KPTI and other mitigations by helping constrain speculative behavior. This approach allowed older CPUs to receive partial mitigation without requiring a complete hardware redesign.

B. Spectre Fixes

1) Retpolines: Due to Spectres functionality, mitigation is much more difficult. It cannot be fixed with one patch alone; this would require a complete overhaul of the modern CPU architecture.

Google made a new technique called Retpotline (return trampoline). This technique can be used on an x86 architecture. This fix works on second version of Spectre which exploits the branch target buffer.It is a software construct that replaces indirect branches with a controlled return sequence, effectively blocking speculative jumps to attacker-influenced locations. This technique is easily explained with this simple assembly code:

```
call setup          ; push return address

capture:
    pause           ; speculative execution gets stuck
    jmp capture     ; infinite loop

setup:
    mov %rax, (%rsp) ; overwrite address with target
    ret              ; return to the safe target
```

Listing 1. Retpoline: Mitigation for Spectre Variant 2

The purpose of this assembly sequence is to prevent speculative execution from reaching an attacker-controlled target after an indirect branch. Instead of executing a direct indirect jump or call (such as `jmp %rax`), the code uses a return-based trampoline mechanism.

The CPU is tricked into speculating into a harmless loop (capture) instead of following the real target. The setup block sets the correct target on the stack, and the ret transfers control to it - but only after speculation has been halted. This effectively traps speculative execution and mitigates Spectre Variant 2 without requiring hardware support.

2) Serialization: Serialization is a low-level technique used to mitigate Spectre Variant 2 (Branch Target Injection) by halting speculative execution at critical points in the instruction stream. The core idea is to prevent the CPU from speculatively executing indirect branches that could be influenced by attacker-controlled BTB (Branch Target Buffer) poisoning.

This mitigation uses special CPU instructions known as serialization or speculation barrier instructions. These include:

- `LFENCE` — a lightweight fence that orders loads; often used for Spectre v1 mitigation.
- `MFENCE` — a memory fence that orders memory operations more generally.
- `CPUID` — a heavyweight instruction that fully serializes the pipeline, halting all speculation.

To mitigate Spectre v2, a serialization instruction is placed directly before an indirect branch. For example:

```
cpuid               ; fully serialize execution
call *%rax          ; indirect branch
```

Listing 2. Serialization before an indirect branch

In this example, the `cpuid` instruction ensures that speculative execution does not proceed beyond the call until the target is resolved. This prevents speculative execution from entering attacker-influenced gadgets, thus blocking potential information leaks through side channels.

Although effective, this approach comes with a significant performance cost. Instructions like `cpuid` cause full pipeline flushes, introducing delays that are unacceptable in performance-critical code. For this reason, serialization is generally used only in highly sensitive execution paths, such as system call boundaries or context switches.

3) Compiler and Software Hardening: To address Spectre Variant 1 (Bounds Check Bypass), compiler-level mitigations were developed. These include inserting speculation barriers after sensitive branches and rewriting memory access patterns to prevent speculative leaks.

Software libraries and browsers were also updated to reduce exposure. For example, JavaScript engines were modified to include artificial timing jitter and restrict access to high-resolution timers, preventing accurate side-channel measurements. In cloud environments, providers adopted stricter process isolation strategies and hardware-level controls such as extended page tables (EPT) protection.

V. Evolving Threats and Outlook

A. TikTag

1) Discovery and Context: TikTag was publicly disclosed in June 2024 by researchers from the University of California, Riverside, and Purdue University [6]. The vulnerability targets ARM processors that implement the Memory Tagging Extension (MTE) feature. MTE is designed to prevent memory safety issues like use-after-free or buffer overflows by associating memory regions with specific tags and checking those tags during access.

2) Technical Details: MTE works by associating a 4-bit tag with every 16 bytes of memory and embedding the same tag in the upper byte of the pointers. A tag mismatch during memory access leads to a tag check fault (TCF). TikTag shows that these checks can be bypassed or distinguished via speculative execution, allowing attackers to infer the memory tag without triggering an architectural fault. TikTag introduces two main variants:

- **TikTag-v1:** Exploits speculation shrinkage in CPUs. It relies on multiple speculative loads to the same address using a pointer with a guessed tag. Depending on whether the tag matches, the CPU speculatively continues or halts, affecting the cache state. By observing cache hits or misses on a secondary test pointer, the attacker can infer the correct memory tag.
- **TikTag-v2:** Leverages speculative store-to-load forwarding. A store operation is issued to a guess pointer

followed by a load that depends on the forwarded value. If the tag matches, the forwarding happens, and the test pointer is cached. If the tag mismatches, forwarding is blocked and the cache remains untouched, enabling tag leakage through timing analysis.

3) Mitigation: At the hardware level, ARM CPUs must avoid altering speculative execution and prefetching behavior based on tag check results. At the software level, inserting speculation barriers (`isb`, `sb`) or adding instruction padding can prevent gadget construction.

B. Downfall

1) Discovery and Context: Downfall was disclosed in August 2023 by Daniel Moghimi. The vulnerability (CVE-2022-40982) exploits speculative execution through the `gather` instruction on Intel x86 CPUs. Contrary to the belief that recent CPUs with hardware mitigations are resistant to transient execution attacks, Downfall demonstrates that Intel Ice Lake, Tiger Lake, and other CPUs remain vulnerable due to unflushed SIMD register buffers. [7]

2) Technical Details: The core of Downfall is a class of vulnerabilities called **Gather Data Sampling (GDS)**. The `gather` instruction collects scattered memory values in vector registers, based on the provided indices. Due to speculative and out-of-order execution, the CPU may forward stale or unintended data into SIMD registers using temporal buffers, which are not properly flushed across context switches or hyperthreads.

Downfall introduces multiple attack techniques [7]:

- **Gather Data Sampling (GDS):** The attacker increases the transient execution window (e.g., via cache misses), then executes a speculative `gather` instruction that accesses uncacheable or faulting memory. Data is speculatively fetched and forwarded into SIMD registers. The attacker encodes leaked bytes into cache state and recovers them using side channels like Flush+Reload.
- **Cross-Process Covert Channel:** GDS leaks up to 22 bytes per attack, enabling high-speed covert channels between sibling hyperthreads at up to 5.8 kB/s. This channel works even across virtual machines.
- **Key Extraction:** Using GDS, attackers extracted AES-128 and AES-256 encryption keys from the OpenSSL tool within seconds. The attack requires no knowledge of the cryptographic algorithm or plaintext/ciphertext, making it particularly stealthy and efficient.
- **Gather Value Injection (GVI):** Combines GDS with Load Value Injection (LVI) to inject stale values into dependent memory accesses. This allows attackers to exploit gadgets that use gather-derived indices to access memory, resulting in arbitrary data leakage.
- **Breaking Intel SGX:** Downfall leaks sensitive register values from SGX enclaves, including the AES sealing key. Even at SGX's highest security configuration (with microcode updates and SMT disabled), GDS was able to extract data due to incomplete buffer flushing.

3) Real-World Attacks: Downfall demonstrated complete attacks on Linux systems to steal arbitrary data at rest, cryptographic material, and kernel-level memory. In particular, researchers created proof-of-concept attacks that exploited common code patterns like `memcpy` and `rep mov` to leak protected memory via speculative prefetches.

4) Mitigation: Intel addressed Downfall with a microcode update designed to prevent transient forwarding from the gather instruction. Additional mitigations include:

- Disabling Simultaneous Multithreading (SMT) to block sibling-thread data leakage.
- Compiler or OS restrictions to disable gather or replace it with safer alternatives.
- Insertion of `lfence` barriers after gather to block speculative data forwarding.
- Use of tools like Transynther to fuzz instruction-level side channels and find GDS vulnerabilities.

Despite these measures, the attack shows that modern CPUs still suffer from previously unknown transient execution behaviors, particularly involving advanced SIMD instructions. The Downfall attack underscores the need for proactive hardware design changes to ensure robust memory isolation.

C. GhostRace

1) Discovery and Context: GhostRace was introduced in 2024 by researchers from Vrije Universiteit Amsterdam and IBM Research Europe. It represents a novel class of speculative execution vulnerabilities called Speculative Race Conditions (SRCs), where speculative bypassing of synchronization primitives like mutexes or spinlocks results in exploitable race conditions [8]. The vulnerability is tracked as CVE-2024-2193.

2) Affected Systems: GhostRace affects all major CPU architectures (Intel, AMD, ARM, IBM) that allow speculative execution of conditional branches in synchronization primitives. The researchers specifically demonstrated the attack on Intel x86-64 systems using the Linux kernel, targeting device drivers and system calls.

3) Technical Details: GhostRace exploits the fact that synchronization primitives (e.g., mutexes) rely on conditional branches to enforce mutual exclusion. These branches are vulnerable to speculative execution. If a branch predicting access to a critical section is mistrained, the CPU may speculatively enter it—even when the lock is architecturally held by another thread.

This turns architecturally race-free code into a Speculative Race Condition (SRC), allowing speculative execution to bypass synchronization. The paper focuses on a subclass called Speculative Concurrent Use-After-Free (SCUAF), where freed memory is speculatively accessed before the pointer is updated or cleared.

To reliably exploit SCUAF, GhostRace introduces an unbounded race window technique. Using high-precision hardware timers and IPI (Inter-Processor Interrupt) storms, the attacker interrupts the victim thread at a precise moment,

creating a large speculative execution window. This enables speculative control-flow hijacking and kernel data leakage.

4) Mitigation: The proposed mitigation involves inserting a serialization instruction (`lfence`) after the lock acquisition instruction in synchronization primitives. This ensures speculative execution is terminated before entering critical regions. The developers of the Linux kernel acknowledged the issue but declined full integration due to performance concerns. However, mitigations against IPI storming were introduced.

VI. Conclusion

This paper has reviewed the fundamental mechanisms of speculative execution in modern processors and highlighted the security consequences of performance optimizations. The introduction of Meltdown and Spectre in 2018 revealed that trusted design assumptions in CPU architectures were flawed, and the impact was profound: attackers could use microarchitectural side effects to extract sensitive information from memory, even across privilege boundaries.

Since then, industry efforts to mitigate these vulnerabilities have included kernel isolation (KPTI), software-level fixes like retpolines, serialization techniques, and microcode updates. However, these fixes often come with significant performance trade-offs, and in some cases only partially address the root causes. The emergence of more advanced attacks such as TikTag, Downfall, and GhostRace underscores the persistent challenges of securing speculative execution.

A. Interpretation of Results

Speculative execution attacks exploit fundamental flaws in CPU behavior. Evidence suggests that while many mitigation techniques are effective in isolation, they often fail to address the root of the problem, that is, that CPUs expose sensitive internal state through side effects. Even with modern hardware protections and revised software practices, new attack surfaces continue to emerge, highlighting the need to rethink performance-security trade-offs in processor design.

B. My Thoughts

When I started with this work, I knew what Meltdown and Spectre are and I knew some kind of mitigation existed. After finishing this work I am really not sure what to think. The problem lies deep inside the CPU design and I don't think there is a way to fix this. The current direction is to solve all the problems using software and hardware co-design and hope for the best. The problem is too deep rooted, so unfortunately with all the new attacks that are emerging every year, fixing all of them is quite literally a Sisyphean task.

I think that the CPU architecture will need huge hardware overhaul if problems like this become bigger, as the performance cost for fixing all the bugs will be too huge to implement. Future architectures may need to integrate hardware-based speculation barriers, redesign prediction mechanisms, or offer secure-by-default execution paths.

References

[1] N. Abu-Ghazaleh, D. Ponomarev, and D. Evtyushkin, 'How the spectre and meltdown hacks really worked', IEEE Spectrum, vol. 56, no. 3, pp. 42–49, 2019.

[2] Intel, 'Intel 64 and IA-32 Architectures Software Developers Manual', Sep-2016. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.pdf.

[3] A. Staff, 'Meltdown and Spectre: Heres what Intel, Apple, Microsoft, others are doing about it', Jan-2018. [Online]. Available: https://arstechnica.com/gadgets/2018/01/meltdown-and-spectre-heres-what-intel-apple-microsoft-others-are-doing-about-it/.

[4] M. Lipp et al., 'Meltdown: Reading Kernel Memory from User Space', in 27th USENIX Security Symposium (USENIX Security 18), 2018.

[5] P. Kocher et al., 'Spectre Attacks: Exploiting Speculative Execution', in 40th IEEE Symposium on Security and Privacy (S&P'19), 2019.

[6] J. Kim et al., 'TikTag: Breaking ARM's Memory Tagging Extension with Speculative Execution', arXiv [cs.CR]. 2024.

[7] D. Moghimi, 'Downfall: Exploiting Speculative Data Gathering', in 32th USENIX Security Symposium (USENIX Security 2023), 2023.

[8] R. Hany, A. Mambretti, A. Kurmus, and C. Giuffrida, 'GhostRace: Exploiting and Mitigating Speculative Race Conditions', in (to appear) 33rd USENIX Security Symposium (USENIX Security 24), 2024.

[9] Y. Yarom and K. Falkner, 'FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack', in 23rd USENIX Security Symposium (USENIX Security 14), 2014, pp. 719–732.

[10] D. S. McFarlin, C. Tucker, and C. Zilles, 'Discerning the dominant out-of-order performance advantage: is it speculation or dynamism?', SIGPLAN Not., vol. 48, no. 4, pp. 241–252, Mar. 2013.

[11] N. Mosier, H. Nemati, J. C. Mitchell, and C. Trippel, 'Analyzing and Exploiting Branch Mispredictions in Microcode', arXiv [cs.CR]. 2025.