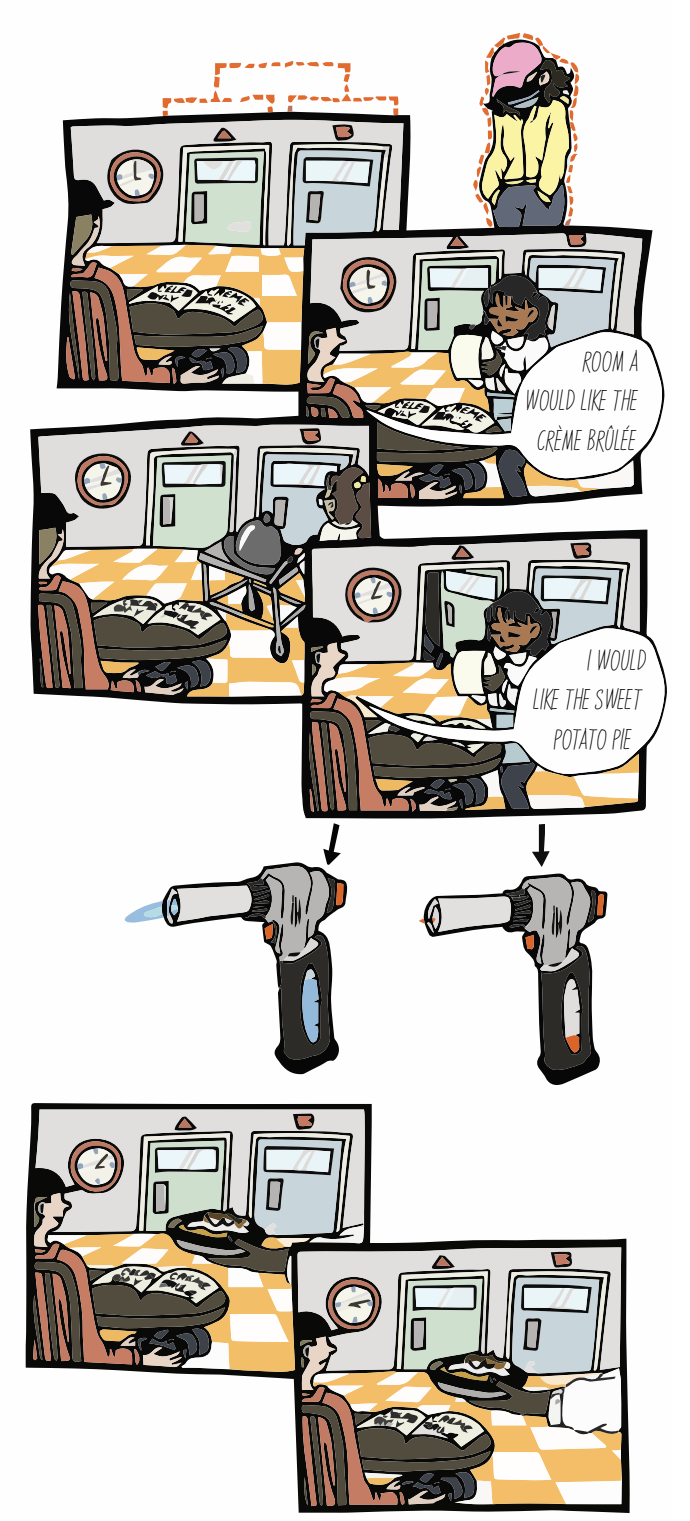
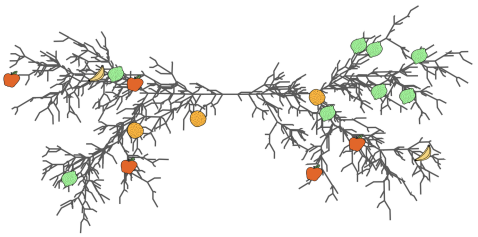
A Leaky Crystal Ball



Top (Celebrity in Disguise): A secret value is hidden in either Room A or Room B, represented by the celebrity and their doppelgänger. The attacker does not know which room contains the real secret.
Middle (Reporter's Probe): The reporter (attacker) sends a crafted request: ordering crème brûlée to Room A. This dish is special because only the celebrity would ever receive it. The CPU (chef) will prepare it only if the celebrity is in Room A.
Bottom (Timing Side Channel): The reporter then makes a normal order, sweet potato pie. If the torch is still warm, the pie finishes faster — revealing that the crème brûlée preparation occurred, and thus that the celebrity is in Room A. If no torch warmth remains, the celebrity must be in Room B. This mirrors speculative execution: incorrect paths are rolled back architecturally, but their side effects persist in microarchitectural state, leaking secrets through timing.

A Leaky Crystal Ball

Speculative execution optimizes performance by executing instructions before knowing if they're needed, leaving microarchitectural traces in cache memory even when results are discarded. Attacks such as Meltdown and Spectre exploit this, by constructing code sequences where a secret value determines which memory addresses are accessed during speculation. By measuring which addresses load quickly afterward (indicating they were cached), attackers can determine if specific bits were 0 or 1, allowing secrets to be extracted across privilege boundaries.



CPU Pipeline Architecture ∘ Speculative Execution ∘ Meltdown & Spectre ∘ Cache Timing Channels ∘ Branch Predictor Training ∘ Microarchitectural State ∘ Kernel Memory Extraction ∘ Mitigations & Performance ∘ KPTI & Retpolines ∘ Hardware Vulnerabilities ∘ 30% Performance Cost

"Be wary of anyone who claims to be able to see the future."

— Wit to Shallan, Rosharan year 1174

A Leaky Crystal Ball

> The core techniques behind speculative execution — pipelining, branch prediction, and out-of-order execution — originated as performance optimizations in the 1960s and matured across RISC and superscalar designs in the 1980s–1990s. The IBM System/360 Model 91 (1966) introduced dynamic scheduling. Tomasulo's algorithm and register renaming were foundational (Tomasulo, 1967). By the 2000s, speculative execution had become ubiquitous in high-performance processors.
>
> Meanwhile, side-channel attacks emerged independently in cryptography. Kocher (1996) demonstrated timing attacks on modular exponentiation. Cache timing attacks followed, with Bernstein (2005) showing cache-based AES key recovery. Rowhammer (2014) revealed that hardware itself could be attacked — rapidly accessing memory rows caused electrical interference that flipped bits in adjacent rows, allowing privilege escalation. These hardware vulnerabilities showed that physical properties of components could be weaponized. Yet usually such attacks were seen as requiring special conditions, deliberate software flaws, or physical proximity.
>
> Spectre and Meltdown, disclosed in 2018, showed that speculative execution — once considered internal and safe — could be manipulated into violating memory isolation. The result was a universal class of vulnerabilities.

Every program — from your web browser to your text editor — is a sequence of simple instructions that tell the processor what to do: load data from memory, store results back, add numbers, compare values. Think of it like a recipe where each step must be precise: "take flour from cabinet," "add to bowl," "mix ingredients." These instructions manipulate data stored in memory, the computer's workspace.

Memory is like a vast warehouse with different storage areas. Closest to the processor are registers — think of them as the processor's hands, holding just a few items it's actively working with. Next come caches, like workbenches near the assembly line, storing frequently-used data. Main memory is the large warehouse floor, holding gigabytes of data but taking longer to access. When you open a document, it moves from your hard drive into main memory, then pieces flow through caches to registers as the processor works on them.

Processors execute instructions through pipelines — assembly lines where different stages happen simultaneously. While one instruction is being decoded (figuring out what "add" means), another is being executed (actually adding numbers), and a third is being fetched from memory. This parallelism makes processors fast, executing billions of instructions per second. Dependencies constrain this parallelism. Adding $A + B$ requires knowing both values first. Processors analyze these dependencies and reorder independent operations to keep the pipeline flowing.

The instruction `if (x < y)` determines whether certain code will be executed, but evaluating the condition takes time. Waiting would leave execution units idle. Instead, processors predict the outcome and speculatively execute that path. Branch predictors track patterns

©2026 David H. Silver, CC BY-NC 4.0

— loops usually continue, error checks usually pass, sorted data produces predictable comparisons.

If the prediction is correct, the speculative work becomes part of normal execution. If incorrect, the speculative instructions are discarded and execution switches to the correct path. From the perspective of the committed values of memory and registers, it is as if the speculation never occurred. Internally, though, speculative execution modifies shared state: caches (fast memory that stores recently accessed data), branch predictors, translation buffers, and other timing-sensitive components. As we will see, these modifications leave traces.

Modern computers set strict boundaries between different programs and between user programs and the operating system kernel. The kernel — the core of the operating system — manages hardware, controls security, and stores sensitive data such as passwords, encryption keys, and private information from all running programs. User programs (your browser, text editor, games) are forbidden from reading kernel memory. Similarly, one user's programs cannot read another user's data. These restrictions are enforced through memory protection mechanisms built into the processor.

But what if these protections could be bypassed? Imagine requesting a book from a library where some rooms are restricted. To save time, the librarian starts walking toward the room before confirming whether you have access. If you are authorized, the book is delivered. If not, the librarian returns — nothing was given. But the door was opened. Now suppose you are not told which room contains which book. Later, you notice that one door swings more easily. You did not receive the book, but you know where the librarian went. This is a side channel attack — extracting secrets through indirect observations rather than direct access.

Meltdown (2018) is an example of an exploit that targets privileged kernel memory — the protected area containing the operating system's secrets. User programs attempting to read kernel memory trigger an immediate error, like trying to enter a secured building without a keycard. But during speculative execution, the processor starts fetching the forbidden data before checking permissions. The security check eventually catches this and triggers an exception — Meltdown never "officially" reads the secret. But in the microseconds between the speculative read and the security check, the secret value exists in the processor. The attack uses this value to access a specific location in the attacker's own memory array. When the security exception fires and speculation is canceled, the secret itself is erased — but the access pattern remains in the cache. By timing how fast different array locations load, the attacker deduces which location was accessed, revealing the secret byte value.

Many operating systems historically mapped kernel pages into every user address space for performance. Meltdown exploited deferred permission checks on affected Intel CPUs, allowing transient use of privileged data before the fault was architecturally raised. AMD reported its CPUs were not affected by Meltdown; ARM susceptibility varied by core implementation.

Spectre trains the branch predictor, then exploits its predictions (Kocher et al., 2018). During training, call a victim function repeatedly with valid array indices. The predictor

learns that bounds checks like `if (x < array_len)` succeed. During attack, provide an out-of-bounds index. Expecting success, the hardware speculatively reads `array[x]` beyond the array boundary. The speculative path uses this secret value as an index: `probe[secret * 4096]` . Each possible byte value maps to a different memory page, separated by 4 KB to avoid cache-line collisions. After the misprediction triggers rollback, the attacker times accesses to all 256 probe pages. The fastest access reveals which page was cached, exposing the byte. Repeat to extract regions.

Branch predictors maintain 95%+ accuracy by detecting patterns in program behavior. They track individual branches and correlations between them. A global history register records recent branch outcomes, indexing into pattern tables that predict future behavior. Some predictors track paths — sequences of branches — to capture control flow patterns. The predictor's state is shared across privilege levels and between different programs, creating a cross-domain communication channel.

Spectre variant 1 exploits conditional branches. Variant 2 targets indirect branches — jumps to addresses computed at runtime, common in object-oriented code and function pointers. The hardware must predict not just taken/not-taken but the actual destination address. The Branch Target Buffer (BTB) caches these predictions, but entries can be poisoned to redirect speculation to attacker-chosen gadgets.

Cache timing provides the physical channel. Processors use multiple cache levels — L1 (closest to CPU, 4 cycles), L2 (12 cycles), L3 (40 cycles), and main memory (200+ cycles). These differences reveal which addresses were accessed. Single-bit leaks, extracted repeatedly, compromise entire keys through differential analysis.

These attacks can extract any data the CPU has access to: passwords stored in memory, cryptographic keys, personal files, browser history, emails, database contents. If the kernel has it in memory, Meltdown can read it. If a program processes sensitive data, Spectre can extract it — even from JavaScript running in a web browser.

Mitigations constrain speculation at every level. `LFENCE` instructions create serialization barriers. Kernel page table isolation (KPTI) unmaps kernel memory from user space. Indirect-branch mitigations include retpolines — a Google-developed technique replacing indirect branches (Turner, Sugerman, & Broido, 2018) with a return-based construct that traps speculation via the return predictor — and hardware IBPB/IBRS/STIBP controls. Some predictors are flushed or partitioned on context switches on newer systems. Each fix degrades the optimization it protects.

Processors achieve high performance through pipelines — 14–19 stages in contemporary designs. Without speculation, a mispredicted branch would flush the pipeline, wasting dozens of cycles. At 4 GHz, each wasted cycle represents 250 picoseconds of lost computation. Multiply by billions of branches per second, and performance would drop drastically.

The x86 `RDTSC`/`RDTSCP` instructions return a cycle count (cycle-level resolution). Time can be derived given clock frequency; on invariant TSC systems this can approach sub-nanosecond granularity. When restricted, alternatives exist: thread scheduling provides a coarse clock, contention on shared resources amplifies timing differences, and browser

JavaScript enables attacks through SharedArrayBuffer spin loops or WebAssembly instruction counting.

After Spectre and Meltdown, each processor optimization became a potential side channel. Vector units, return predictors, schedulers, and virtualization boundaries revealed new attack surfaces.

Later attacks exploited other processor features. Downfall (2023) targets Intel's AVX gather instructions — when speculatively gathering data, these vector operations transiently load values from unauthorized memory regions. The values leave cache footprints after rollback, allowing extraction of cryptographic keys by timing which vector elements were accessed. Intel processors from sixth through eleventh generation carry this flaw.

Retbleed (2022) demonstrated that even retpolines fail. Return instructions — used in every function call — use their own prediction mechanism. By poisoning the return stack buffer, attackers force speculative execution of arbitrary gadgets, bypassing the carefully constructed retpoline defenses.

Inception (2023) showed AMD processors aren't immune — it creates "phantom speculation" by nesting mispredictions within mispredictions. GhostRace (2024) weaponized something previously thought safe: synchronization primitives. Race conditions during speculative execution leak data even from properly synchronized code.

A fully-mitigated system may run 30% slower than its vulnerable predecessor. Some workloads see greater degradation — databases that rely heavily on indirect calls, JIT compilers that generate dynamic code, virtualized environments with frequent context switches. Mitigations cost years of Moore's Law gains.



"Don't quote me, but the oracle at Delphi told me she's using ChatGPT."

Folklore About Some Interesting Bugs

Geographic Email Limits

A university sysadmin faced a bizarre bug: emails wouldn't travel more than 500 miles. Boston at 420 miles worked fine; Memphis at 520 miles failed completely. A timeout was set in nanoseconds instead of milliseconds. Light travels 200,000 km/s through fiber. The timeout expired at exactly 500 miles.

The Ice Cream Correlation

Drivers reported their keyless entry failed — but only after buying vanilla ice cream at one specific shop. Not chocolate, not strawberry. Just vanilla. Engineers were skeptical until they confirmed it.
The vanilla counter was at the front, chocolate in back. Vanilla buyers returned before the engine cooled, triggering a temperature-sensitive component failure.

Posture-Dependent Passwords

An IT team dismissed reports that a password worked standing but failed sitting — until they saw it happen. Same user, same password, different postures, different results.
Two keyboard keys had been swapped. Standing users looked down and typed what they saw. Seated users touch-typed from muscle memory, entering the wrong sequence.

Night Shift Crashes

Night-shift staff reported server crashes that never occurred during the day. No software changes, no power issues — just nightly failures.
The cleaning crew's floor polisher vibrated at 60 Hz, matching certain hard drives' resonant frequency. The vibration misaligned read/write heads just enough to crash the system.

The 2:45 PM Crash

One server crashed daily at exactly 2:45 PM. Logs showed thermal throttling, but only at that precise time.
Sunlight through a west-facing window hit the poorly ventilated rack at just the right angle. The temperature spike at 2:45 PM pushed the server past its limit.

Spectre and Meltdown: Mechanics and Leakage Rates

Introduction

Spectre and Meltdown exploit transient execution: instructions issued before permissions or branches resolve. Although architecturally squashed, these instructions leak data through persistent microarchitectural side effects — most notably, the cache — observable via timing measurements.

Meltdown: Transient Load + Fault Deferral

The CPU speculatively executes a faulting memory load and uses the result before the exception is raised. A dependent load encodes the secret byte into cache state.

```
; RCX ← kernel memory address
movzx rax, byte [rcx]
shl   rax, 12
mov   rbx, [probe + rax]
```

Comments:

- Line 1: Transiently loads a protected byte (zero-extended) into RAX.
- Line 2: Multiplies the secret by 4096 (page alignment).
- Line 3: Loads from `probe[s × 4096]`, caching a secret-dependent line.

Observation: The attacker probes access times to `probe[i × 4096]` and identifies the secret by locating the cache hit.

Spectre: Mistrained Bounds Bypass

The attacker mistrains the branch predictor to speculatively skip a bounds check. This leads to out-of-bounds access during the transient window.

```
if (x < array_length)
    temp = probe[secret[x] * 4096];
```

Only executed speculatively. The loaded cache line leaks `secret[x]`.

Setup: Train with in-bounds x; switch to out-of-bounds. Side effects persist even when the branch is mispredicted.

Cache Timing Oracle

Let $S \in \{0, \ldots, 255\}$ be the secret. Let C_i be the access time to `probe[i × 4096]`. Then:

$$\widehat{S} = \arg\min_i C_i$$

$$\text{where} \quad C_i = \begin{cases} T_{\text{hit}} + \epsilon_i & \text{if } i = S \\ T_{\text{miss}} + \epsilon_i & \text{otherwise} \end{cases},$$

With low noise, each measurement leaks up to approximately 8 bits.

Information-Theoretic View

Let X be the secret, Y the timing observation. Leakage is given by:

$$I(X;Y) = H(X) - H(X \mid Y).$$

Under uniform X and low timing noise, $I(X;Y) \to 8$ bits. Repetition and majority decoding mitigate jitter and measurement error.

Measured Bandwidth

Meltdown (local):	up to ∼500 KB/s
Spectre (native):	∼10 KB/s
Spectre (JavaScript):	order of bits/s

Reported rates vary widely across microarchitectures, operating systems, and mitigation settings.

References:

Kocher et al. (2018). *Spectre Attacks: Exploiting Speculative Execution.* arXiv:1801.01203.

Lipp et al. (2018). *Meltdown: Reading Kernel Memory from User Space.* USENIX Security.