Spectre Attacks:
Exploiting Speculative Execution

Paul Kocher | Jann Horn | Anders Fogh | Daniel Genkin | Daniel Gruss | Werner Haas | Mike Hamburg |
Mortiz Lipp | Stefan Mangard | Thomas Prescher | Michael Schwartz | Yuval Yarom

Presented by Christos Komis
sdi2200073@di.uoa.gr

What is Spectre?

- A security flaw discovered in the physical design of almost every modern computer chip.
- For decades, processors were built to "guess the future" so they could run faster. Spectre proved that these educated guesses can be manipulated to steal secret data.
- No easy fix.

```
$ lscpu | grep Spectre

Vulnerability Spectre v1: Mitigation;
usercopy/swapgs barriers and __user pointer
sanitization

Vulnerability Spectre v2: Mitigation;
Retpolines; IBPB conditional; STIBP always-on;
RSB filling; PBRSB-eIBRS Not affected; BHI Not
affected
```

Background: Increasing CPU performance?

No more easy gains from low-level physics, e.g.:

- Increase clock rates
- Improve memory speeds

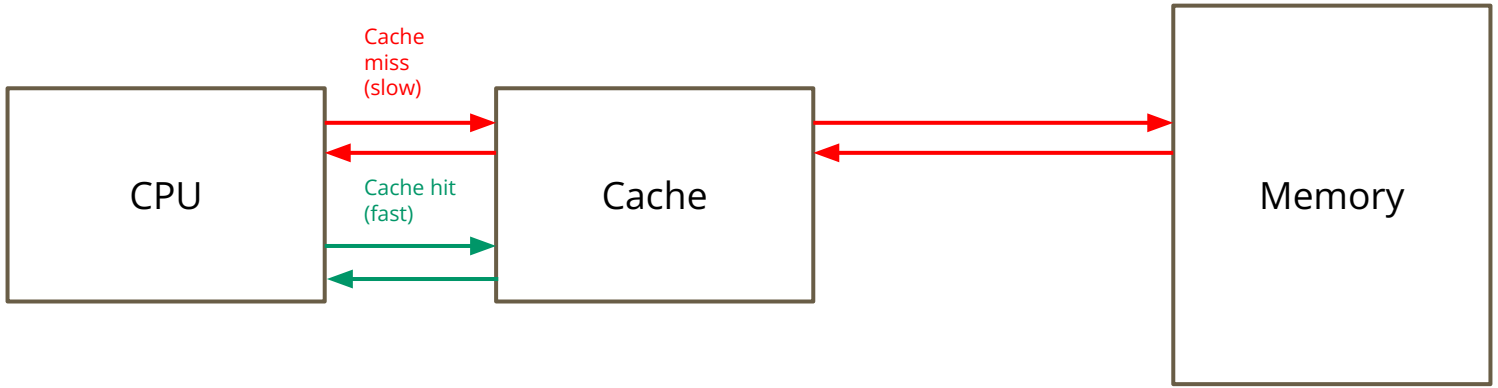
Industry focus on pipelining + boosting average-case performance, e.g.:

- Reducing memory delays → Caches
- Working during delays → Speculative Execution

Background: CPU Cache

Purpose: Bridge the massive speed gap between the lightning-fast CPU and slow Main Memory (RAM) by storing recently used data nearby.

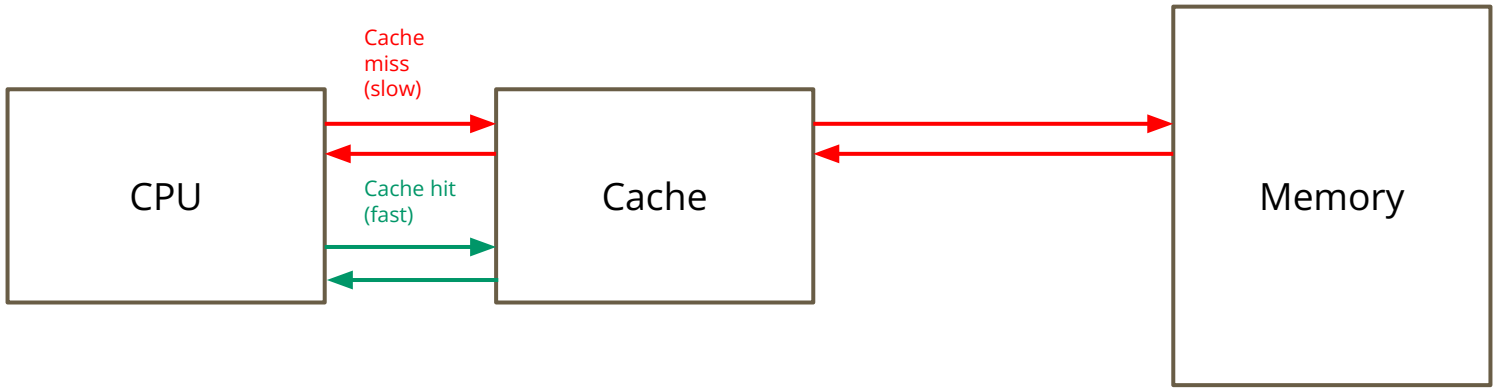
- Load Instructions that are served by the cache are faster
- Load Instructions that are served by the RAM are slower.



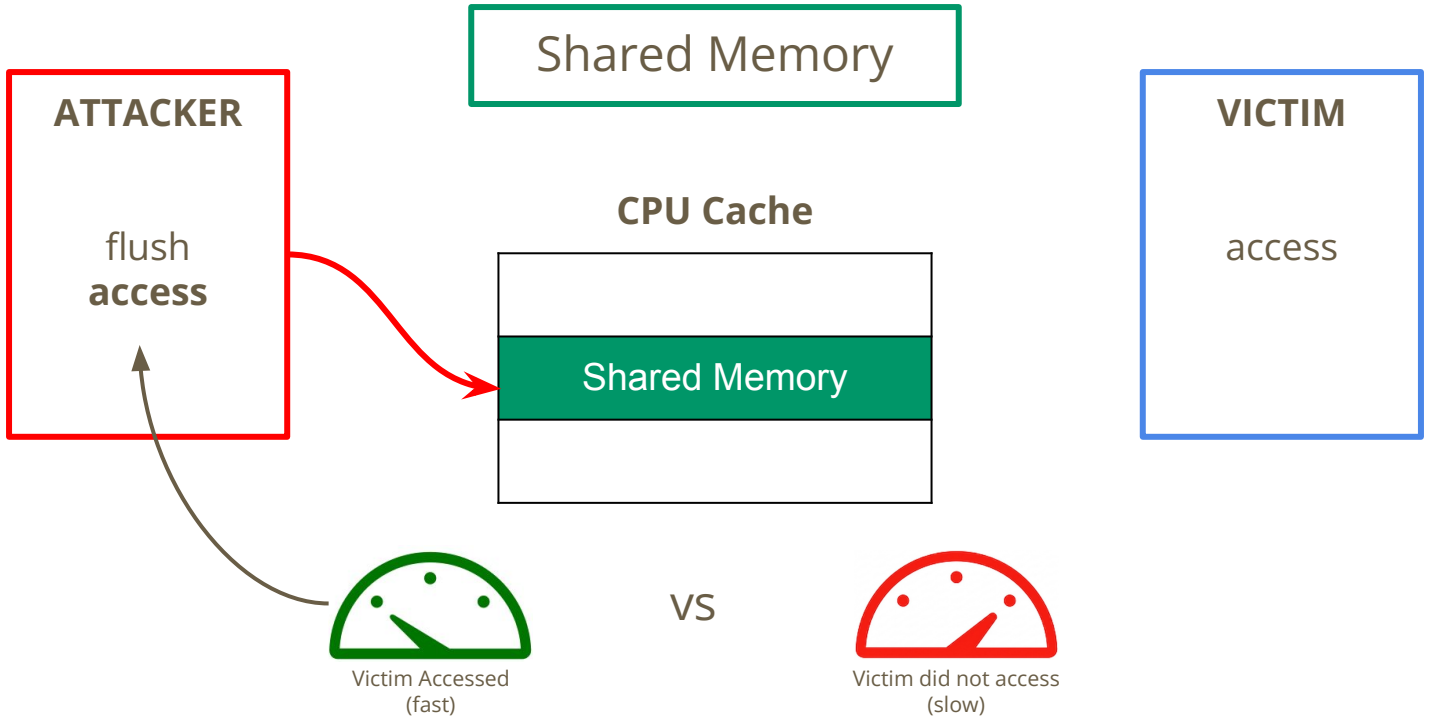
Background: CPU Cache

Side Effect: An attacker can time load instruction to determine whether a process accessed a specific address.

- **Flush+Reload:** A common cache side-channel attack on x86 that exploits this vulnerability.



Background: Flush+Reload



Background: Speculative Execution

Speculation: Start likely tasks early, then clean up errors.

Example:

```
if (x == 1) {
    abc…
} else {
    xyz…
}
```

If x is uncached, processor faces a long delay CPU can guess execution path & proceed speculatively. When x arrives from DRAM, check if guess was correct.

- **Correct guess:** commit speculative work = performance gain
- **Wrong guess:** Discard faulty work

Question

Are there any security implications from speculative execution?

Variant 1: Conditional Branch Attack

```
if (x < array1_size)
      y = array2[array1[x] * 512];
```

Variant 1: Conditional Branch Attack

```
if (x < array1_size)
      y = array2[array1[x] * 512];
```

Attack Scenario:

- Code runs in trusted context
- Adversary wants to read memory and controls unsigned integer *x*
- Branch predictor will expect *if* to be true (e.g. because prior calls had *x* < *array1_size*)
- *array1_size* and *array2[k]* are not in cache

Variant 1: Conditional Branch Attack

```
if (x < array1_size)
      y = array2[array1[x] * 512];
```

Attacker wants to read address *target*, where **target* = *13*, and calls victim code with *x = target - array1*.

- Speculatively execute while waiting for *array1_size*
- Predict that *if()* is true
- Read address *array1 + x*
- Read returns byte = 13 (fast – in cache)

Attacker times reads from *array2[i * 512]*

- Read for *i = 13* is fast (cached), revealing secret byte

Example Implementation in JavaScript

```
if (index < simpleByteArray.length) {
      index = simpleByteArray[index | 0];
      index = (((index*4096) | 0) & (32*1024*1024 - 1)) | 0;
      localJunk ^= probeTable[index | 0] | 0;
}
```

Goal: Leak browser secrets from within the JS sandbox

Challenge: The JIT Compiler automatically inserts safety checks and performs dead code elimination, ruining the speculative attack

Solution:

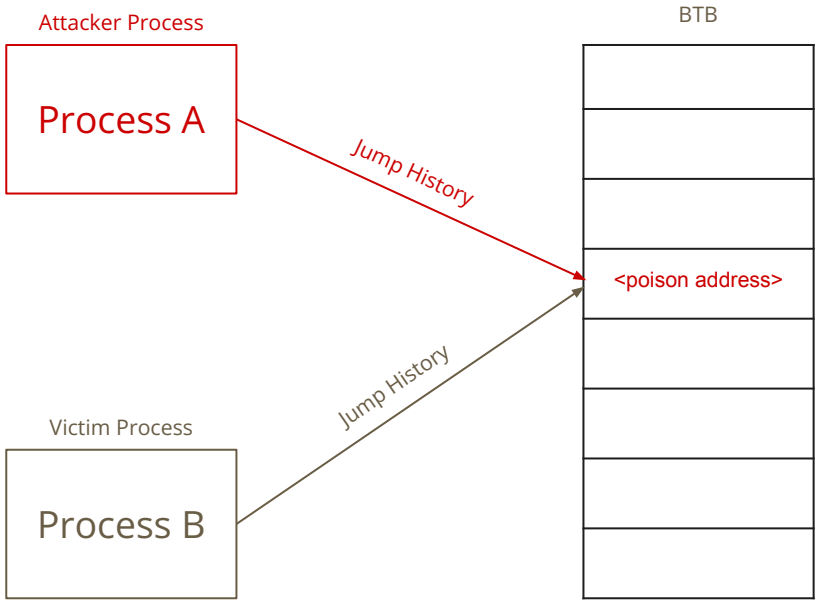
1. Explicit ***if*** triggers Bounds Check Elimination
2. ***\|0*** forces strict integers
3. ***^ (XOR)*** prevents Dead Code Elimination

Accuracy of Recovered Data

- Errors can arise for several reasons
  - Prefetching
  - Noise from neighbouring processes
- Repeat attack multiple times to make reliable determinations
- Error rates of approximately 0.005%

Variant 2: Indirect Branch Poisoning

- **Target:** Indirect Branches
- **Mechanism:** The CPU uses a Branch Target Buffer (BTB) to remember where these branches jump to
- **Flaw:** The BTB is a shared hardware resource; it does not isolate predictions between different applications
- **Attack:** An attacker process “poisons” the BTB, tricking the CPU into speculatively redirecting the victim’s control flow



Indirect Branch Poisoning PoC on Windows

- **Goal:** Force the victim application to leak its key.
- **Step 1:** Defeat ASLR. Exploit Windows “Once-Per-Boot” randomization for shared system libraries (*ntdll.dll*)

```
while (1) {
      read(fd, buffer, size);
      sleep(0);
      hash(key, buffer);
}
```

- **Step 2:** Find a Gadget. Locate an instruction sequence that leaks memory
- **Step 3:** Poison the BTB. Mistrain the BTB using attacker’s copy to link a common function call to the Gadget’s address
- **Step 4:** The Execution. Victim performs a function call, CPU speculatively jumps to the Gadget, and the secret is leaked to the cache.

Indirect Branch Poisoning PoC on Windows

Victim's Code at Address 0xffeeddcc

```
adc edi, dword ptr [ebx+edx+13BE13BDh]
adc dl, byte ptr [edi]
```

C-like Representation

```
edi += *(ebx+edx+0x13BE13BDh);
dl = *edi;
```

- The above sequence is located in *ntdll.dll*
- *When Sleep()* call is done
  - registers *ebx* and *edi* contain attacker-controlled values
  - register *edx* contains attacker-known value
  - Set *edi = probeArray*, *ebx = key_address - edx - 0x13BE13BDh*
- The instruction sequence is a perfect gadget for our attack

Indirect Branch Poisoning PoC on Windows

Victim's Code at Address 0xffeeddcc

```
adc edi, dword ptr [ebx+edx+13BE13BDh]
adc dl, byte ptr [edi]
```

Victim's Code at Address sleep

```
jmp qword ptr [<address x>]
```

Modified via Copy-on-Write

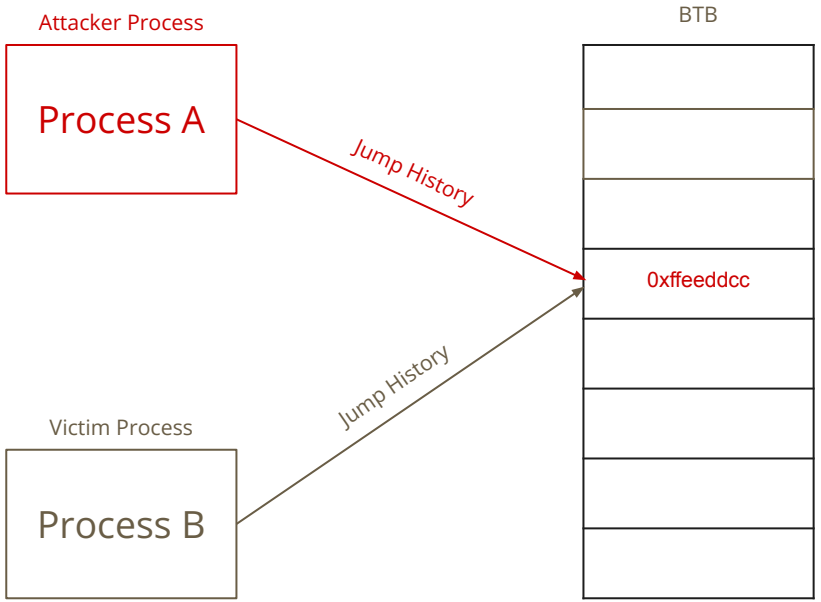
Attacker's Code at Address 0xffeeddcc

```
ret
```

Attacker's Code at Address sleep

```
jmp qword ptr [<address y>]
```

Where *y = 0xffeeddcc



Indirect Branch Poisoning PoC on Windows

Additional Challenges:

1. Branch predictor uses the preceding jump history in making predictions. Thus, attacker must mimic the victim's branch history prior to the jump.
   Good News: Mistraining doesn't need to match the exact virtual addresses – only the low 20 bits.
2. Attacker needs to locate victim's stack to locate key's address. This can be done using the following single-instruction gadget:

```
sbb eax, [esp + ebx]
```

Variant 1 vs Variant 2

- **The Target:** Conditional Branches (e.g., *if* statements)
- **The Vulnerability:** Requires the code to actually have a vulnerable code pattern
- **The Exploit:** Tricking the True/False branch predictor

- **The Target:** Indirect Branches (e.g., function pointers)
- **The Vulnerability:** The victim code can be perfectly secure. The hardware itself is the vulnerability.
- **The Exploit:** Poisoning the Branch Target Buffer (BTB) to hijack the execution path to a malicious gadget.

Variations

- **Spectre Variant 4:** Uses speculation in the store-to-load forwarding logic.
- **Evict+Time:** Measure the timing of operations that depend on the state of cache.

```
if (false but mispredicted as true)
        read array1[R1]
read [R2]
```

- **Instruction Timing:** Instructions whose timing depends on the values of the operands may leak information on the operands.
- **Variations on Speculative Execution:** An adversary could also mistrain the branch predictor such that, after an interrupt occurs, the interrupt return mispredicts to an instruction gadget.

Mitigation Options

- Implement methods to disable speculative execution in future processors.
  - Unlikely to provide an immediate fix to the problem.
- Introduce *serializing* or *speculation blocking* instructions to prevent speculative execution.
  - Compiler performs static analysis to identify *dangerous* code blocks and inserts *fence* instructions.
  - Can help mitigate indirect branch poisoning, if placed before an indirect branch. Pipeline cleared → branch resolved quickly → less speculatively executed instructions.

Mitigation Options

- Google Chrome: Execute each web site in separate process
- Replace bounds checking with index masking
- Degrade timer resolution and disable *SharedArrayBuffers* used to create software timers
- Microcode Patches
  - *Indirect Branch Restricted Speculation* (*IBRS*)
  - *Single Thread Indirect Branch Prediction* (*STIBP*)
  - *Indirect Branch Predictor Barrier* (*IBPB*)
  - Performance Impact varies from a few percent to a factor of 4 or more

Mitigation Options

- Retpolines
  - Replace indirect branches with return instructions.
  - Reads *Return Stack Buffer* (*RSB*) instead of *BTB*.
  - Return instruction is predicted to an endless loop.
  - *RSB* can underflow and fall back to *BTB* for prediction → Intel issued microcode updates disabling this mechanism.

Before Retpoline

```
jmp rax
```

After Retpoline

```
call load_label

capture_ret_spec:
pause
lfence
jmp capture_ret_spec

load_label:
mov [rsp], rax
ret
```

Conclusion

- **Fundamental Flaw**: Decades of processor optimization prioritized performance over security, creating deep-rooted vulnerabilities
- **Widespread Vulnerability**: Spectre isn’t limited to one manufacturer but affects the physical design of almost every modern CPU
- **Exploitation is Complex but Deadly**