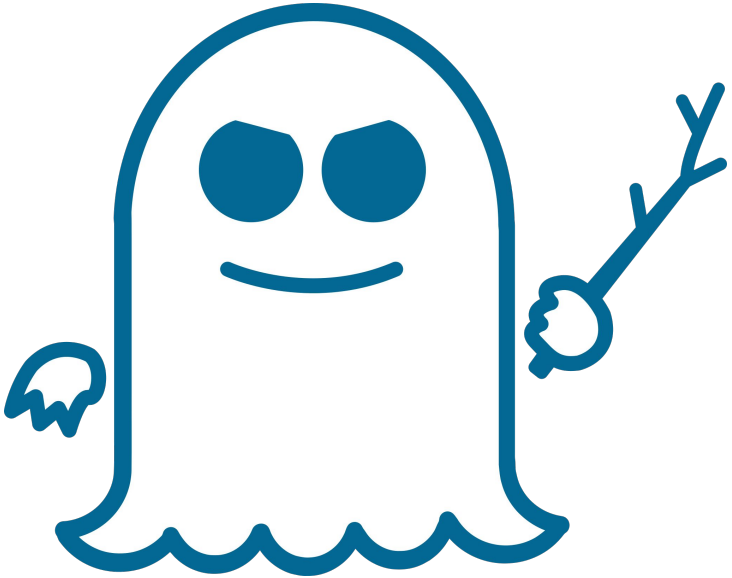
Side / covert channels

Meltdown & Spectre



Outline

- Definitions
- Cache-based covert channel as a side channel
- Cache Missing for Fun and Profit ~ Colin Percival
- Spectre & Meltdown shared concepts
- Spectre Variant 1
- Spectre Variant 2
- Meltdown
- Mitigations

Covert channels

Covert channel is a way of transferring information between processes that are not supposed to be allowed to communicate by the computer security policy.

"*not intended for information transfer at all, such as the service program's effect on system load*" ~ Lampson 1973.

Covert channels

- Don’t use legitimate communication scheme -> often bypass security systems.
- Hard to find.
- Hard to remove them - sometimes tightly coupled with legitimate channels.
- Low data-rate - bits/second.
- Low signal-to-noise ratio.
- Can be identified by monitoring system performance (channel noise) or by careful system analysis, but almost impossible to be eliminated completely. Computer Security Technology Planning Study

Side channel

Side-channel attack is any attack based on information gained from the implementation of a computer system, rather than weaknesses in the implemented algorithm itself (e.g. cryptanalysis and software bugs).

computer -> hardware -> real-life side effects.

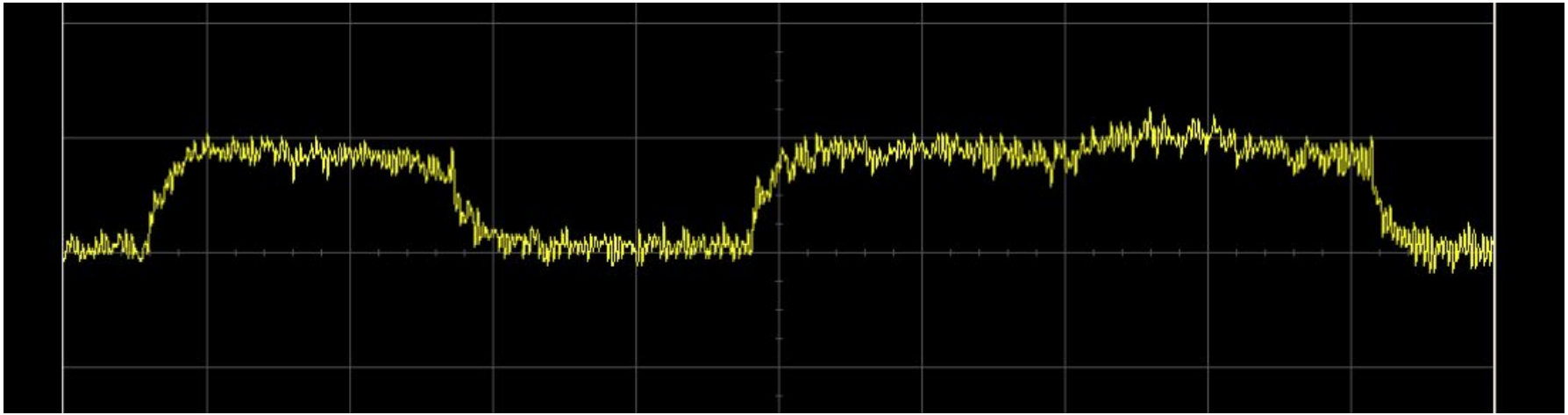
Covert channels are often used as a cryptographic side-channel.

Sometimes attacker controls both sides, the part that induces the side effect, and the part that measures the side effect.

Other covert / side channels.

- cache
- timing
- power-monitoring - ctf task
- acoustic - RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis
- electromagnetic field analysis
- differential fault analysis - induce faults with environmental changes e.g. voltage, current, overclocking, electric / magnetic field, radiation etc.
- data remanence - cold-boot - recovering encryption keys from RAM.
- software-initiated fault attacks - rowhammer, (bitsquatting?)

RSA - square and multiply power analysis



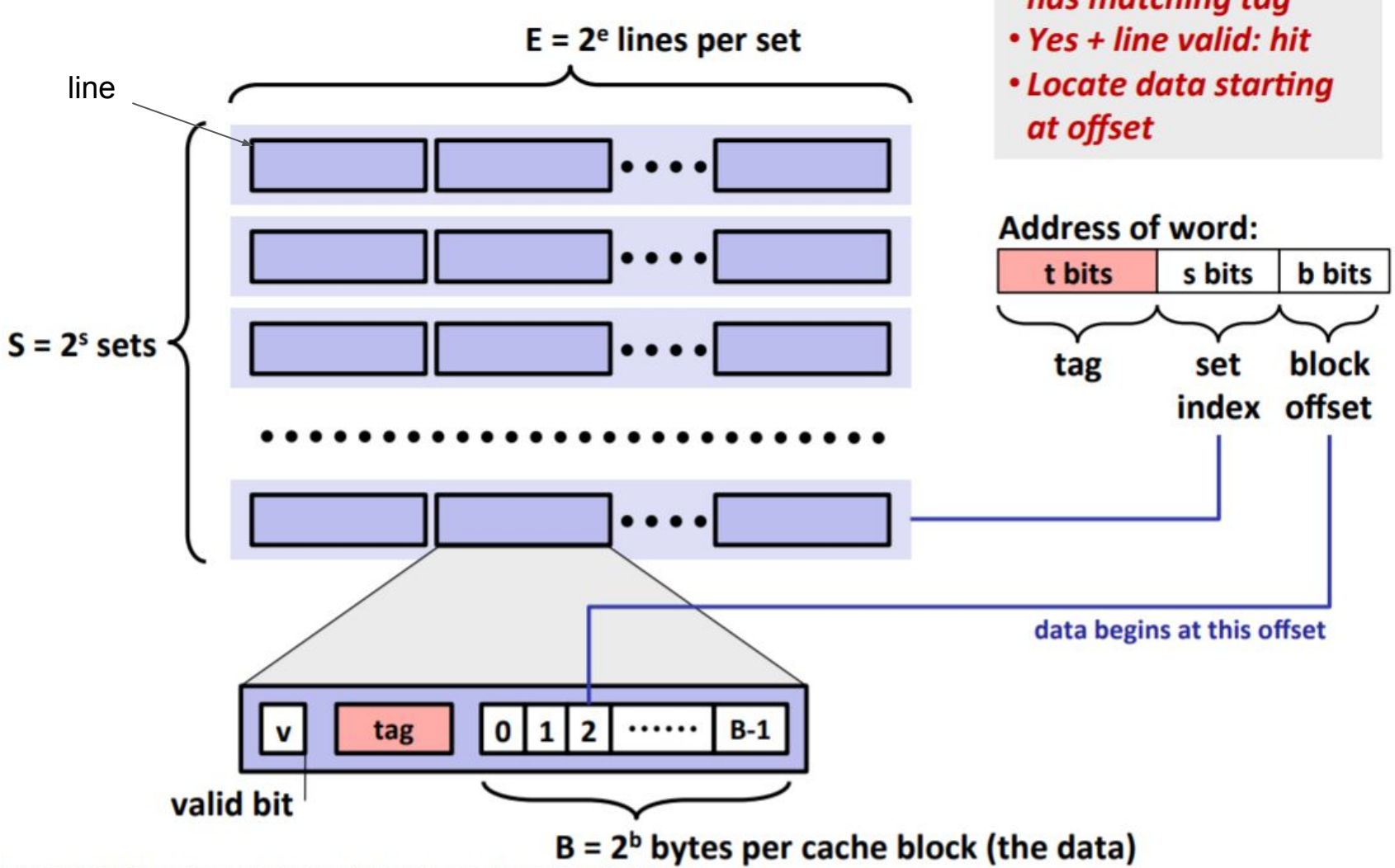
- left peak - squaring
- right peak - multiplication

Simple virtual memory paging covert channel

- P1 and P2 processes sharing a read-only file.
- P1 reads subset of pages causing page faults which load pages into memory.
- P2 reads all of the pages, while measuring time accesses.

Can do without shared file.

- Virtual memory space bigger than half of the physical memory. LRU page replacement policy.
- Concept: P1 faults P2 pages **out** of memory (evicts P2 pages).
- P1 reads either all virtual memory space (bit 1) or reads repeatedly only one page (bit 0). P2 reads whole virtual memory space while measuring time accesses.


has matching tag
• Yes + line valid: hit
• Locate data starting at offset
E = 2^e lines per set
line
S = 2^s sets
Address of word:
t bits
s bits
b bits
tag
set index
block offset
data begins at this offset
v
tag
0
1
2
B-1
valid bit
B = 2^b bytes per cache block (the data)

Cache-based covert channel

EVICT != FLUSH

- Memory access patterns affect data cache state.
- Cache state affects memory access timing.
- Measuring access timings reveals information about memory access patterns:
  - **FLUSH + RELOAD** - requires page deduplication, or shared pages. Attacker flushes line (clflush in x86), victim reloads/not reloads, attacker measures access time.
  - **EVICT + RELOAD** - same as FLUSH + RELOAD but without clflush instruction. Attacker evicts lines by accessing some other addresses.
  - **EVICT + TIME** - when no clflush instruction. Attacker evicts whole set, victim reloads/not reloads one line in a set, attacker triggers victim execution and measures it.
  - **PRIME + PROBE** - when no shared pages. Attacker primes (fills with his data) whole set (or cache), victim accesses/not accesses, attacker measures time - slow -> victim accessed.
- Usually used as a side channel.
- Other covert channel exists in a CPUs; eg. functional units latency.
- Attacks rely heavily on system architecture (cache hierachy, types, SMP, …)

Cache missing for fun and profit ~ Colin Percival

- “Hyper Threading” - sharing processor physical resources. Not only execution units but memory caches. Additional instruction fetch unit, instruction queue and a set of registers.
- Two orthogonal threads can better utilize CPU hardware resources.

Main concepts:

- Two threads paging each others data out of the shared L1 cache.
- Spy thread monitors cache state which is modified by victim thread.
- It identifies patterns.

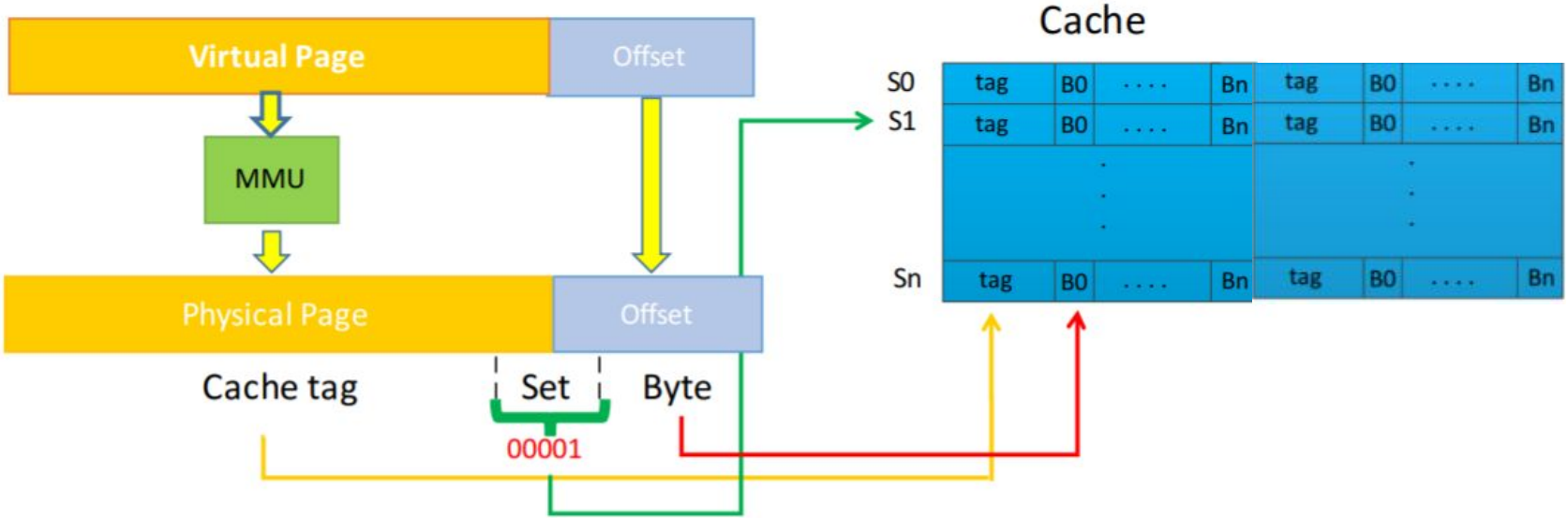
Trojan transmitting 32-bit word

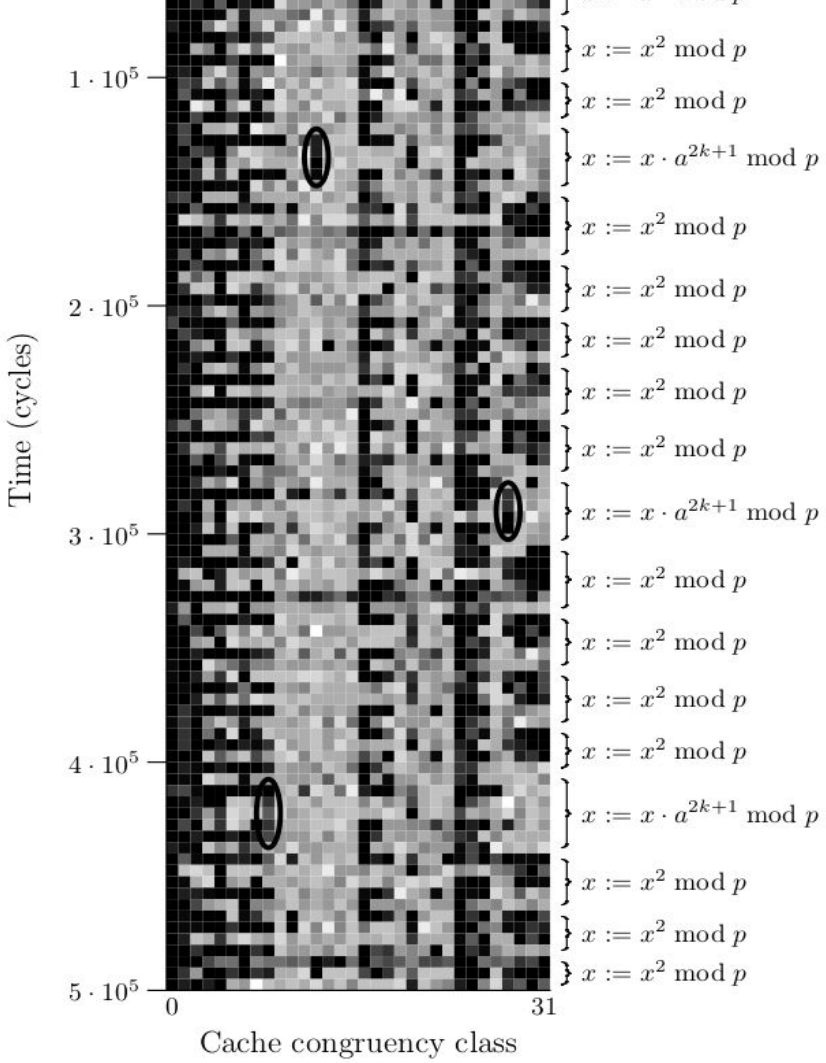
- Pentium 4 L1 cache: 32-sets, 4-way set associative, 64 (2^6)-byte cache line.
- Trojan has 2048 (2^{11})-byte array A. Makes accesses to A[64*i] for every i-th bit of 32-bit message word iff the bit is set. Every access hits different set.
- Spy has 8192 (2^{11} * 4)-byte array B. Repeatedly reads and measures accesses to addresses (4 different tags in every set):
  - 64i
  - 64i + 2048
  - 64i + 4096
  - 64i + 6114

Each memory access performed by the Trojan will evict a cache line owned by the Spy, which can be detected with time measurements.

Virtual -> physical address

- physically indexed and physically tagged (PIPT) cache
- $\log_2$(pagesize / hugepagesize) bytes are not translated!





512-bit RSA cache fingerprint

Part of a 512-bit modular exponentiation in OpenSSL. The shading of each block indicates the number of cycles needed to access all the lines in a cache set. The circled regions reveal information about the multipliers a^{2k+1} being used.

Spy process gets ~310 bits out of 512.

Out-of-Order, speculative execution, branch prediction

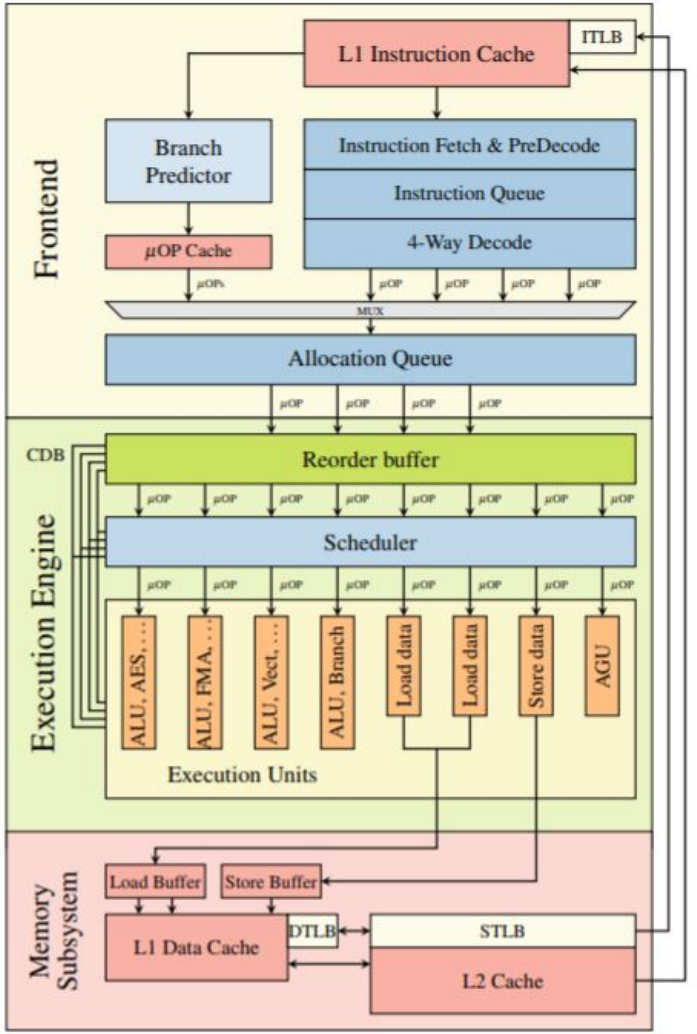
- Instructions can be executed in a different order and in parallel
- Branches are predicted before the target is known

```
1 if (foo_array[index1] ^ foo_array[index2] == 0) {
2       result = bar_array[100];
3 } else {
4       result = bar_array[200];
5 }
```

Out of Order - Tomasulo algo.

In-order issue and retirement. Out-of-order execution.

Architectural state is modified not during execution but later, when instruction is retired.

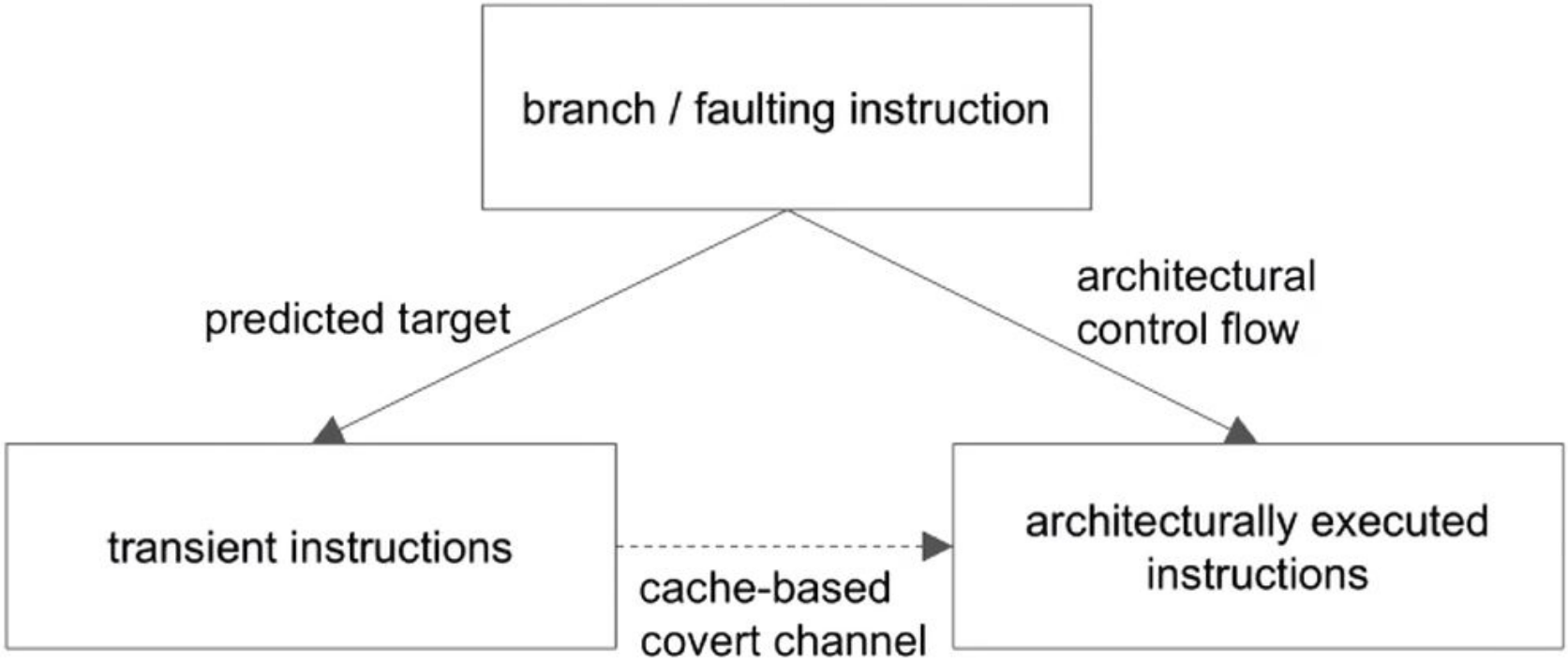


Simplified Intel Skylake microarchitecture

Branch misprediction

- Exceptions and incorrect branch prediction can cause “rollback” of transient instructions
- Old register states are preserved, can be restored (register renaming).
- Memory writes are buffered, can be discarded.
- **Cache modifications are not restored! It’s a microarchitectural state.**

Branch misprediction cache covert channel



Meltdown & Spectre background

- Issuer known from June 2017 (issued by Horn to Intela, ARM, AMD itd.).
- Embargo. First work on mitigations.
- KASLR is Dead: Long Live KASLR. KAISER - better KASLR. KAISER -> KPTI - prevents meltdown.
- December 20th - article “ The current state of kernel page-table isolation”. KPTI. Quick Linux kernel patches. Merge in 3 months.
- Mailing lists include main developers from Google, Amazon, Intel etc.
- The mysterious case of the Linux Page Table Isolation patches. One of first good guessings.
- On january 3rd Intel makes a statement. Tl;dr: “everything is OK”.
- Graz research group starts a website. Google Project Zero publishes post on their blog containing details of attacks.
- cat /proc/cpuinfo

Spectre

- CVE-2017-5753
- "Variant 1"
- "Bounds Check Bypass"

- Primarily affects interpreters/JITs

- CVE-2017-5715
- "Variant 2"
- "Branch Target Injection"

- Primarily affects kernels/hypervisors

Meltdown

- CVE-2017-5754
- "Variant 3"
- "Rogue Data Cache Load"

- Affects kernels (and architecturally equivalent software)

Spectre variant 1 - bounds check bypass

```
if (x < array1_size)
    y = array2[array1[x] * 256];
```

- Execution without speculation is safe. Out of bounds read is not possible.
- Attacker needs to train branch predictor to assume ‘if’ is likely true and make array1_size and array2[] uncached.
- Attacker passes malicious x such that array1[x] is a secret to be read.
- Attacker measures accesses (flush + reload, or some other technique)
  - Access to array2[k*256] would be fast when k=array1[x].

- Without access to array2 attacker can use PRIME + PROBE
  - fills whole cache with his data. Executes victim. Checks access times.
- Alternatively, the adversary can immediately call the target function again with an in-bounds value x’ and measure how long the second call takes. If array1[x’] equals k, then the location accessed in array2 will be in the cache and the operation will tend to be faster than if array1[x’]! = k.
- There are plenty of other methods to figure out k.

Javascript JIT

- Javascript runs in a sandbox
  - not permitted to read arbitrary memory
  - array accesses are bounds checked
- Compiled just-in-time to machine code

We need to trick JIT not to insert bounds check -> generate malicious binary code.

```
1 if (index < simpleByteArray.length) {
2   index = simpleByteArray[index | 0];
3   index = (((index * TABLE1_STRIDE)|0) & (TABLE1_BYTES-1))|0;
4   localJunk ^= probeTable[index|0]|0;
5 }
```

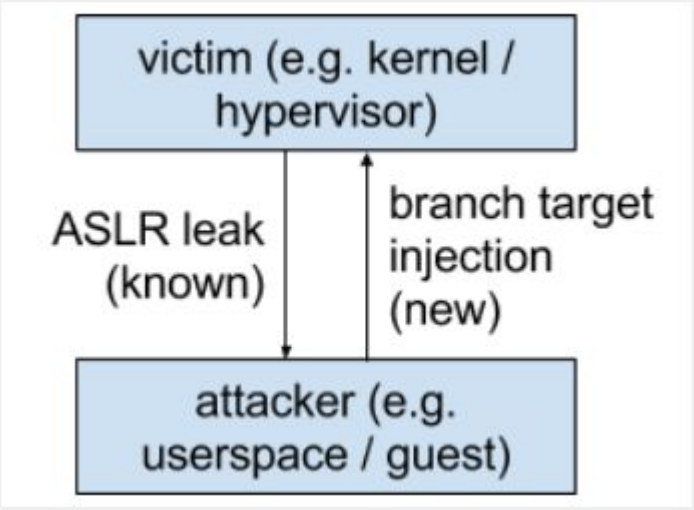
Spectre variant 2 - Branch Target Injection

Jump over ASLR: Attacking branch predictors to bypass ASLR

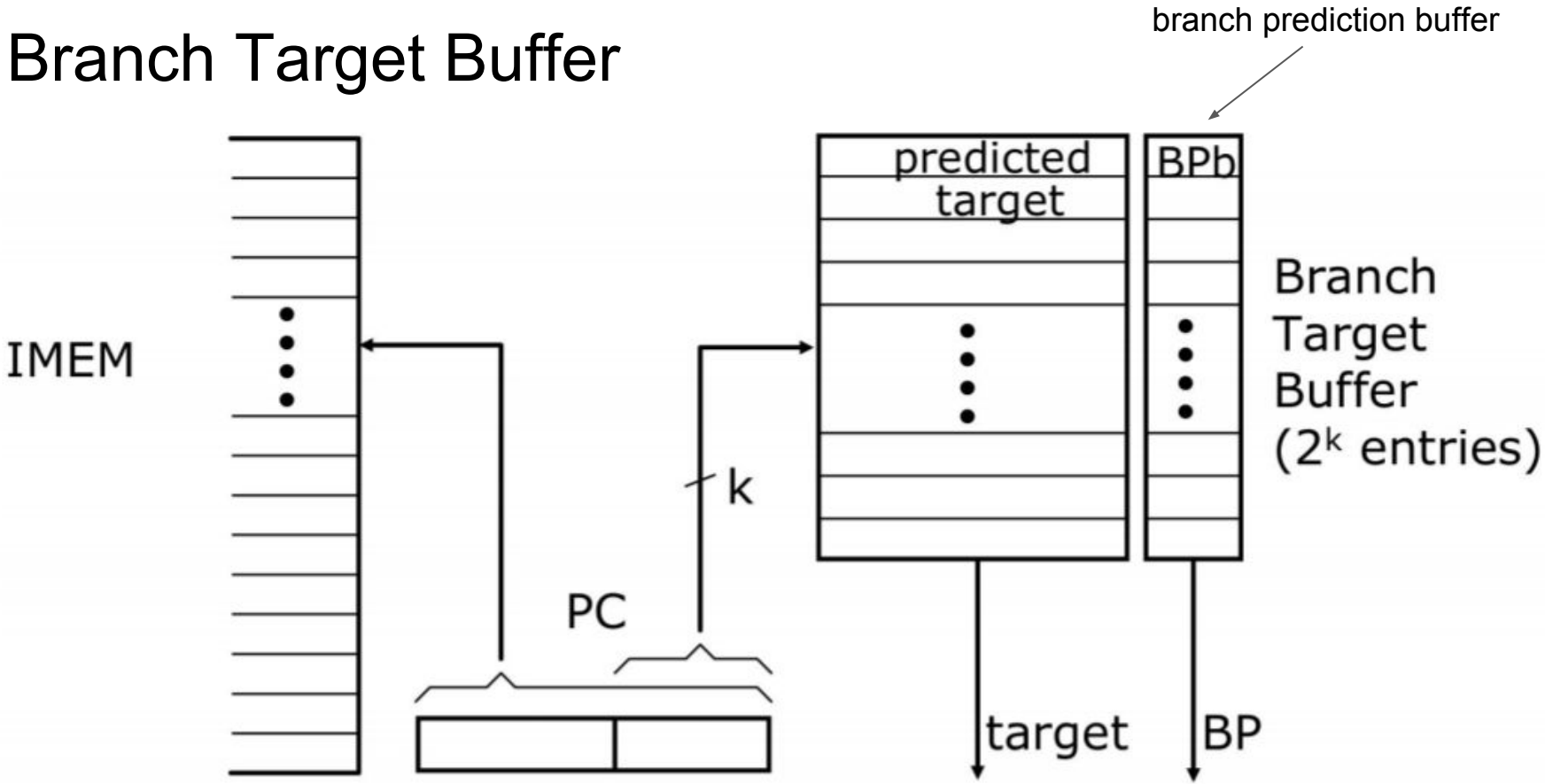
Targeting indirect branches. Trick the branch predictor.
Jump to arbitrary code.

BTB does not tag by security domains.
BTB aliasing.

We don’t really know how BTB looks like on specific architecture - experiments and speculation.



Branch Target Buffer



KVM exploit

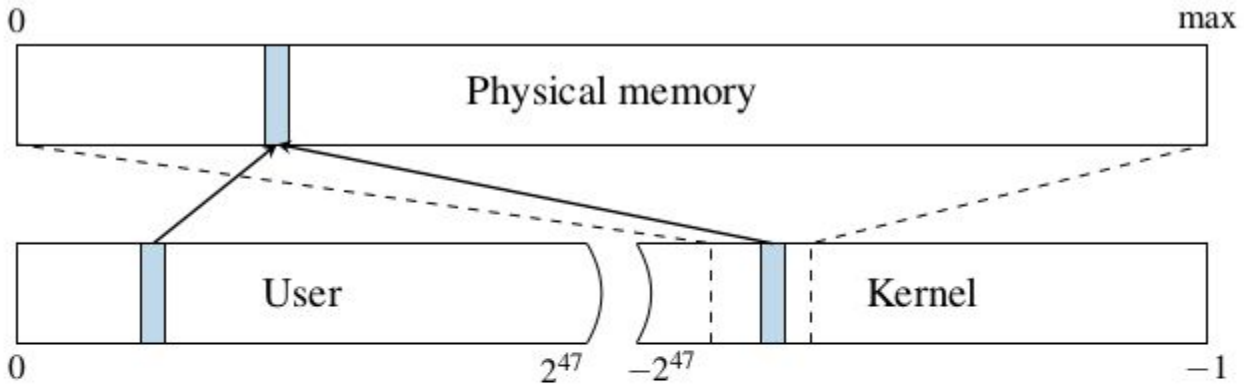
- break hypervisor ASLR
- misdirect first indirect call with memory operand after guest exit
- flush cache line containing memory operand
- guest register state stays across VM exit
- guest memory is mapped in the hypervisor - access guest data in the hypervisor
- abuse eBPF bytecode interpreter; call through register-loading gadget

```
ffffffff81514edd: mov    rsi,r9
ffffffff81514ee0: call   QWORD PTR [r8+0xb0]
```

controlling r8, r9 registers

```
static unsigned int __bpf_prog_run(void *ctx, const struct bpf_insn *insn)
```

Meltdown



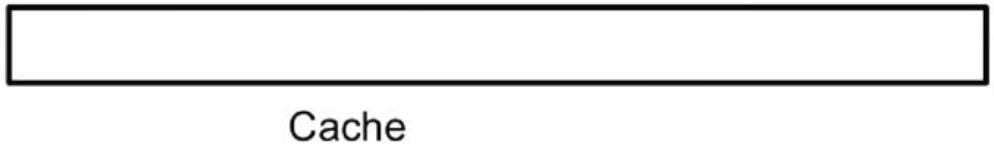
Whole system physical memory is mapped in virtual kernel address space -> kernel can access memory from every process. Can we (as an unprivileged process) read it?

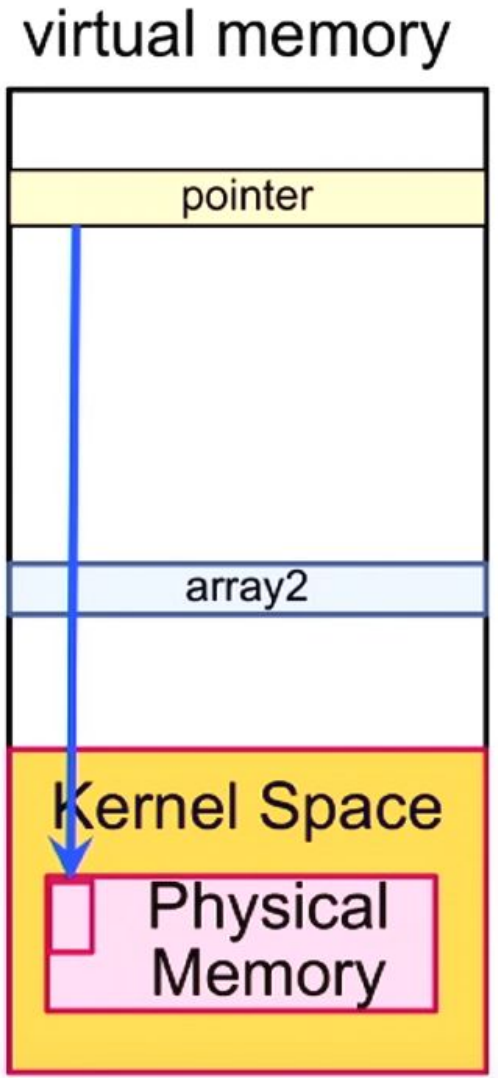
- Every virtual address space has kernel pages mapped, but access to them is limited by permission bits in page tables.
- We need to know where kernel is mapped -> KASLR can be easily broken.

Meltdown - variant 3 - Rogue Data Cache Load

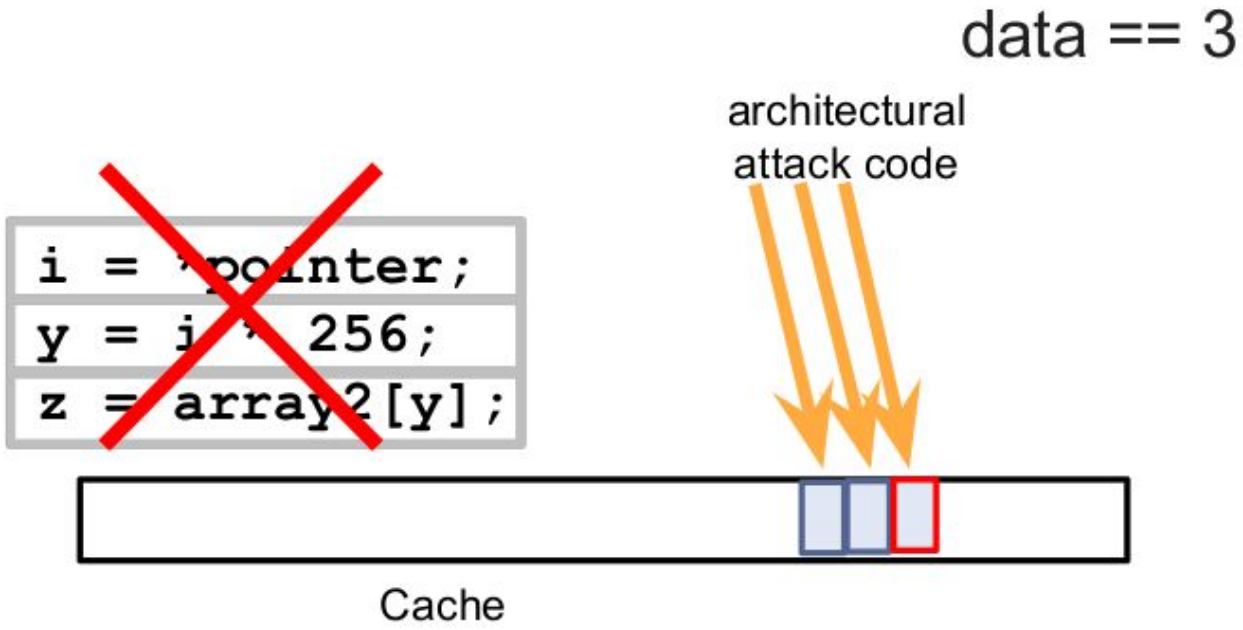
- array2[] uncached
- pointer cached

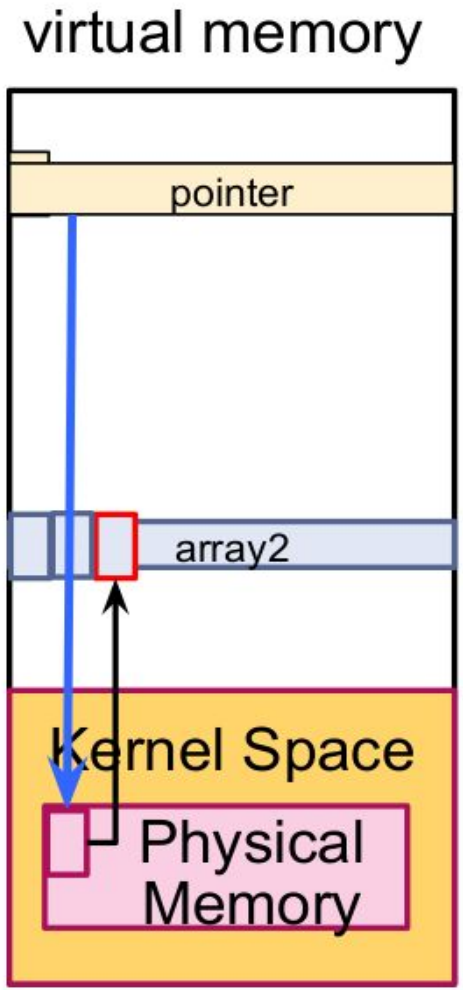
```
i = *pointer;
y = i * 256;
z = array2[y];
```





Processor rollbacks instructions.



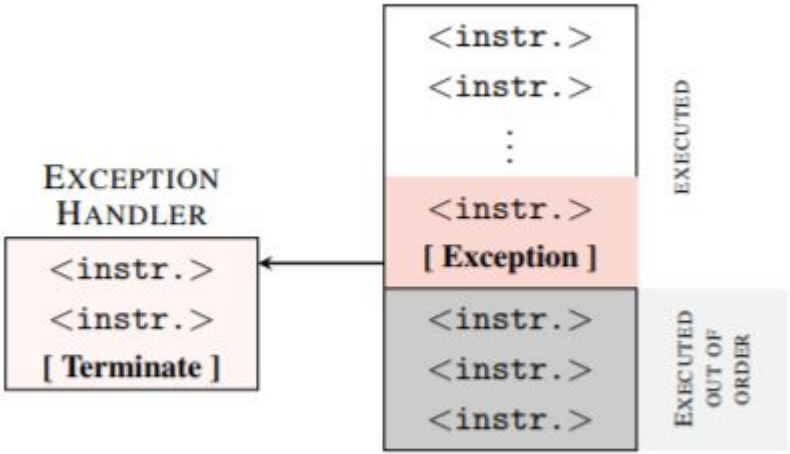


Why does this even work?

To load data from the main memory into a register, the data in the main memory is referenced using a virtual address. In **parallel** to translating a virtual address into a physical address, the CPU also checks the permission bits of the virtual address, i.e., whether this virtual address is user accessible or only accessible by the kernel.

L1 is usually VIPT cache - read from cache parallel to page table permission bits check.

Meltdown

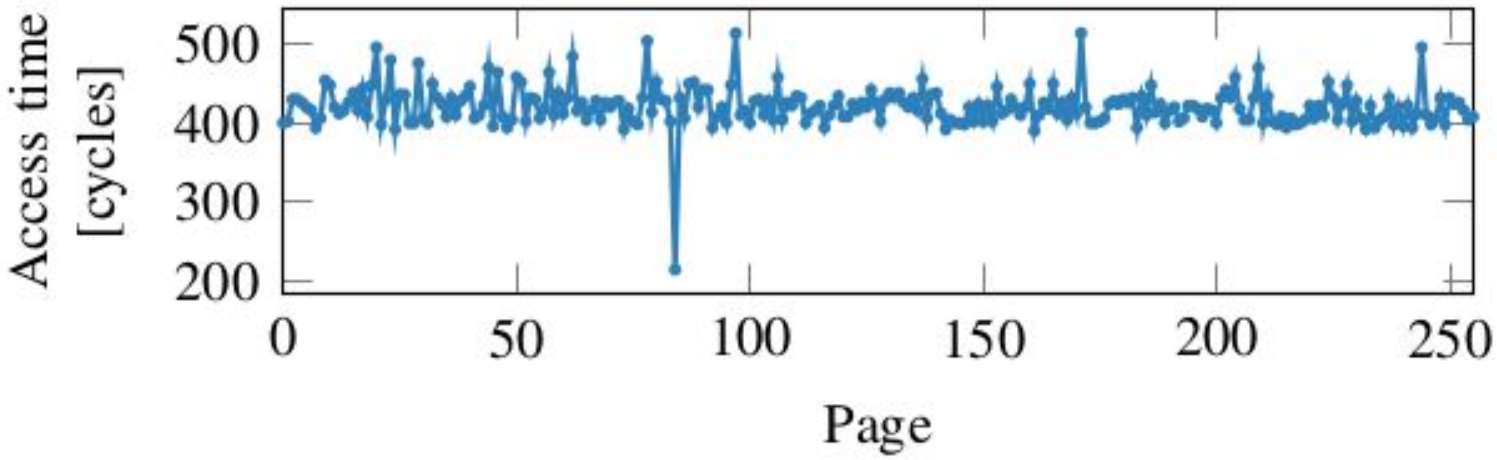


Access to kernel memory triggers exception. How to avoid it?

- subprocess
- TSX
- define signal handler
- catch exception
- dereference kernel address as transient instruction

Meltdown - measurements

- Race condition between privileges check and memory access
- Exceptions are handled in-order during instruction retirement phase.



Another covert channel for Meltdown

The sender could, for example, issue an instruction (sequence) which occupies a certain execution port such as the ALU to send a ‘1’-bit. The receiver measures the latency when executing an instruction (sequence) on the same execution port. A high latency implies that the sender sends a ‘1’-bit, whereas a low latency implies that sender sends a ‘0’-bit

Race condition

When the kernel address is loaded, it is likely that the CPU already issued the subsequent instructions as part of the out-or-order execution, and that their corresponding μOPs wait in the reservation station for the content of the kernel address to arrive. As soon as the fetched data is observed on the common data bus, the μOPs can begin their execution. When the μOPs finish their execution, they retire inorder, and, thus, their results are committed to the architectural state. During the retirement, any interrupts and exception that occurred during the execution of the instruction are handled.

- Privilege checks for memory access based on pagetable entries
- Privilege checks can be performed asynchronously
- Dependent instructions can execute before execution is aborted!
- Race condition in the privilege check
- Straightforward attack: Leak cached data
- TU Graz result: Uncached data can also be leaked
- Suppression of architectural pagefault:
  - signal handler
  - TSX
  - misprediction branch
  - fault in fork (not exactly suppression, but works)

Meltdown & Spectre non-technical articles

- Triple Meltdown: How So Many Researchers Found a 20-Year-Old Chip Flaw At the Same Time
- How a 22-Year-Old Discovered the Worst Chip Flaws in History
- The Hidden Toll of Fixing Meltdown and Spectre
- Meltdown and Spectre Fixes Arrive—But Don't Solve Everything

Mitigations

- retpolines
- code recompilation to prevent generation of vulnerable code patterns
- CSA to find already vulnerable code
- KPTI

Retpolines

```
class Base {
 public:
  virtual void Foo() = 0;
};

class Derived : public Base {
 public:
  void Foo() override { … }
};

Base* obj = new Derived;
obj->Foo();
```

- Indirect branches:
  - polimorphic code
  - jump tables
  - call
  - rtn

Return stack buffer - hardware’s cache for return address, used upon speculative return.

Actual destination maintained on the stack.

Indirect jump

1. Call to set_up_target is known at compile time and will not trigger speculative target resolution. Stack and RSB entries with a return target of capture_spec.
2. Modify the on-stack entry generated by (1), to instead direct to %r11. Note that this does not affect the RSB entry generated above, which will still target capture_spec.
3. Return from original call:
   a. Speculative execution consumes the RSB entry generated by (1), and is captured within the pause loop at (4). These instructions are only executed by the speculative path.

```
jmp *%r11
```

```
 call set_up_target;   (1)
capture_spec:          (4)
  pause;
  jmp capture_spec;
set_up_target:
  mov %r11, (%rsp);    (2)
  ret;                 (3)
```

b) Our return is ultimately retired, the on-stack value is used to locate the actual new instruction pointer and the benign results of any speculative execution in the loop at (4) are discarded

Indirect call

```
call *%r11      jmp set_up_return;
              inner_indirect_branch:
                call set_up_target; }
              capture_spec:         }
                pause;              }
                jmp capture_spec;   } Indirect branch
              set_up_target:        } sequence.
                mov %r11, (%rsp);   }
                ret;                }
              set_up_return:
                call inner_indirect_branch; (1)
```

KPTI

- Every process has two page tables: with kernel mapped and with only interrupt handlers mapped. Userspace mapped in both.
- Every change of hardware register containing page table root node requires TLB flush -> every syscall flushes TLB!
- PCID (process context identifier) to tag TLB entries:
  - possible two entries with the same addresses but different processes
  - no need to flush whole TLB

References

- https://cyber.wtf/2017/07/28/negative-result-reading-kernel-memory-from-user-mode/
- https://github.com/IAIK/armageddon/tree/master/libflush
- https://spectreattack.com/spectre.pdf
- https://meltdownattack.com/meltdown.pdf
- https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html
- https://rwc.iacr.org/2018/Slides/Horn.pdf
- https://www.cs.tau.ac.il/~tromer/papers/cache.pdf

References

- https://conference.hitb.org/hitbsecconf2016ams/materials/D2T1%20-%20Anders%20Fogh%20-%20Cache%20Side%20Channel%20Attacks.pdf
- https://cyber.wtf/2016/06/16/cache-side-channel-attacks-cpu-design-as-a-security-problem/
- https://www.blackhat.com/docs/asia-17/materials/asia-17-Irazoqui-Cache-Side-Channel-Attack-Exploitability-And-Countermeasures.pdf
- http://dreamsofastone.blogspot.com/2015/09/cache-side-channel-attacks.html

Questions?