

Hardware-based Attacks and Countermeasures

Alessandro Pellegrini
a.pellegrini@ing.uniroma2.it

Setting up Reading Primitives

Time and Security

- The time taken to perform some operation on data might reveal information on the data itself

```
int strcmp(char *t, char *s)
{
    for( ; *t == *s ; s++, t++)
        if(*t == '\0')
            return 0;
    return *t - *s;
}
```

- Comparing “ABCD” with “ABxDE” will return after three iterations

Timing the Cache

- When reading data, the CPU actually reads the data from the cache
- Cache protocols try to serve a memory request as fast as possible
- If we can control the state of the cache, we can exploit the time required to serve a memory request to leak data
- Thanks to SMT, we can leak data from L1 caches
- Since lower level of caches are shared, we can also leak data from lower level caches
- The fundamental aspect relates to *how* we can control the state of the cache

Discovering Code Paths

Montgomery Ladder

Input: Point P , scalar n , k bits

Output: Point nP

$R_0 \leftarrow \mathcal{O}$

$R_1 \leftarrow P$

for i from k to 0 **do**

if $n_i = 0$ **then**

$R_1 \leftarrow R_0 + R_1$

$R_0 \leftarrow 2R_0$

else

$R_0 \leftarrow R_0 + R_1$

$R_1 \leftarrow 2R_1$

end

end

} cache line A

} cache line B

} cache line C

} cache line D

- Scalar multiplication is common in cryptosystems
- Nonces should remain secret to prevent reconstructing keys
- We cannot time this algorithm: the number of operations is even
- We can anyhow determine *which branch* of the if statement is taken *for each bit*

Side-channel Attacks

- Side channel: *memory which allows to read other memory content or detect access patterns*
- There are various ways to mount a side channel:
 - Prime + Probe
 - Flush + Reload
 - Flush + Flush
 - Evict + Time
 - Evict + Reload
 - Prime + Abort
- The key idea is the same: bring the cache into a known state and observe side effects generated by other processes
- In this way, we can leak data bypassing OS-based process isolation

Side-channel Attacks

- Basic construction of an attack follows through several phases
 1. *Pre-attack*: the target is acquired (single cache line, or cache set) and timing thresholds are established if needed
 2. *Active attack*:
 - a) Initialization: bring the channel in a known state
 - b) Wait for the victim to make an access to memory
 - c) Analyze the access, by observing side effects left by the victim
 - d) Repeat until all the data are leaked
- Aspects to take into account:
 - Caches are cached either virtually or physically
 - Caches are shared differently depending on the level
 - Interleave of execution between different processes

Evict + Time

- This approach uses the targeted eviction of lines, together with overall execution time measurement
- The attacker first causes the victim to run, preloading its working set, and establishing a baseline execution time
- The attacker then evicts a line of interest, and runs the victim again
- A variation in execution time indicates that the line of interest was accessed

esempio c: lavoriamo con cache l1 perchè facciamo flush di indirizzo logico, che lo ha la cache l1. Inoltre attaccante e attaccato lavorano su stesso processore e stessa cpu. Funziona bene con shared library (così fu rotto AES). Nell'esempio il target è deterministico, se così non fosse, sarebbe più complicato. Inoltre, nel loop, carichiamo il target che vogliamo, ma se così non fosse?

Flush + Reload

- Relies on the existence of shared virtual memory (such as shared libraries or page deduplication), and the ability to flush by virtual address
- The attacker first flushes a shared line of interest (by using dedicated instructions or by eviction through contention)
- Once the victim has executed, the attacker then reloads the evicted line by touching it, measuring the time taken
- A fast reload indicates that the victim touched this line (reloading it), while a slow reload indicates that it didn't

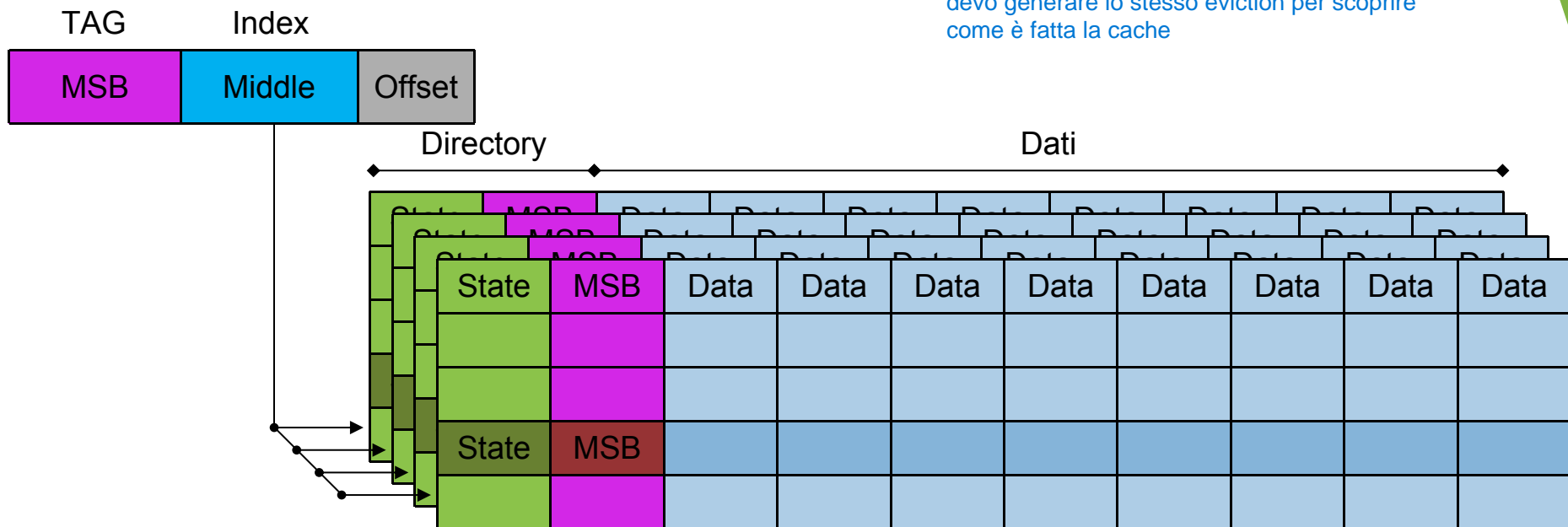
Prime + Probe

- The oldest and largest family of cache attacks, and the most general
- The original attack targeted L1, but it can be extended to L3 (without the need to rely on SMT)
 - L3 Prime + Probe can also work across VMs
- This technique detects the eviction of the attacker's working set by the victim:
 - The attacker first *primes* the cache by filling one or more sets with its own lines
 - Once the victim has executed, the attacker *probes* by timing accesses to its previously-loaded lines, to see if any were evicted
 - If so, the victim must have touched an address that maps to the same set

Prime + Probe

- A successful attack is not that simple (anymore)
 - The attacker requires sets of **colliding memory addresses (eviction sets)**
 - In conventional caches, the mapping of memory to cache is static

devo generare io stesso eviction per scoprire
come è fatta la cache



Prime + Probe: A Hardware Countermeasure

- Modern caches define the mapping of memory-to-cache at runtime
- This mapping changes over time
- The attacker has therefore a reduced time window in which to create the eviction set:
 - Choose randomly N memory addresses
 - Load them into the cache by accessing them
 - There is a possibility of self-collision
 - These self-colliding addresses should be removed from the set
- At this point, the classical attack can take place

Prime + Abort

- A Hardware Transaction is **aborted if there is a conflict in the write set**, *independently* of whether it happened in a transactional context
- The target is a single cache line (attacker must know a precise address to target)
- Simplest naïve attack:
 - Start a transaction
 - Access the target transactionally
 - Wait for an abort
 - **On abort, we know the address was accessed by another process**
- TSX works as an effective *hardware callback*
 - No need for a “timing phase”

Prime + Abort

- Prime + Abort on L1 cache:
 - A cacheline which has been *written* during the transaction (i.e., a cacheline in the transaction's write set) is evicted from the L1 cache
 - The transactional access entails a write in memory
 - We can therefore monitor evictions from L1
 - We can only spy on threads running on the same core, thanks to SMT
- Prime + Abort on L3 cache:
 - A cacheline which has been *read* during the transaction (i.e., a cacheline in the transaction's "read set") is evicted from the L3 cache
 - The transactional access entails reading from memory

Flush + Flush

- It is essentially a variant of Flush + Reload
- This attack is based on the fact that flushing a line can have different timings as well
 - `clflush` takes a different time to execute depending on whether a line is available in cache or not
- Stealthy: it's more difficult to detect flush della linea è ciò che fanno le app, ad esempio per i dataset. Quindi è difficile da capire chi fa cosa
- Faster: performance measures tell that Flush + Flush is faster than other cache attacks

Meeting Out-of-order Pipelines

- A speculative processor can let an attacker exploit micro-architectural effects by using *phantom instructions* even though the pipeline is flushed
- Phantom instructions leave effects in the micro-architectural state for a transient period
- These effects can be observed: *Transient Execution Attacks*
- This is the rationale behind *Spectre/Meltdown* attacks
 - It affects Intel, AMD and ARM processors

metto in pipeline istruzioni non valide, le quali generano effetti sulla cpu

Meltdown Primer

trovare l'address è un altro tipo di problema, che non vediamo qui



```
uint8_t *probe_array = new uint8_t[256 * 4096];  
// Make sure probe_array is not cached  
uint8_t kernel_memory = *(uint8_t*)(kernel_address);  
uint64_t final_kernel_memory = kernel_memory * 4096;  
uint8_t dummy = probe_array[final_kernel_memory];  
// handle (eventual) SEGFAULT  
// determine which of 256 slots in probe_array is cached
```

possible byte values

Intel OOO Execution

Avoid prefetching
& different lines

- Simply measuring the latency to access `probe_array` at the end of this execution path allows to know the value of the kernel memory byte

Fooling the Branch Prediction Unit

- The BPU will “learn” from recent past, independently of how complex is its organization
- Branch prediction units can be *poisoned*:
 - An attacker repeatedly executes a certain pattern of code which makes the BPU learn a certain outcome
 - Once the BPU becomes stable, the code alters its behavior
 - The BPU will continue to make the same prediction
 - The OOO pipeline might allow unauthorized code pass through the pipeline
 - This will be detected eventually, but micro-architectural side effects will still be around

Spectre Primer v1



not available in cache

```
if (x < array1_size) {  
    y = array2[array1[x] * 4096];  
}
```

speculate through (vendor specific)

byte of interest

address available in cache

finally out of bounds

- It is then possible to inspect the cache state of `array2` to see what was the speculatively accessed value of `array1[x]`.

Spectre Primer v2

- The attacker chooses a *gadget* from the victim's address space
 - This gadget can be anywhere, also in eBPF programs!
 - It must be in the victim's executable address space
- The Branch Target Buffer is poisoned to mispredict an indirect branch instruction to the address of the gadget, resulting in its speculative execution
- There is no vulnerability in victim's code here
 - The code is correct, it's only exploited to run according to the attacker's plan
- There are several megabytes of shared libraries where to look a gadget for

Spectre Primer v2



- A possible gadget (taken from Windows 10):
`adc edi,dword ptr [ebx+edx+13BE13BDh]`
`adc dl,byte ptr [edi]`
- The attacker can:
 - set `edi` to the base address of a probe array
 - this can be a memory region in a shared library
 - $ebx = m - 0x13BE13BD - edx$
- The first instruction in the gadget will read 32 bits from m and add it into `edi`
- The second instruction fetches the index m in the probe array into the cache

Mitigating Side Channel Attacks

Possible Mitigations and Detection

- Timing attacks are difficult to mitigate
 - The behaviour of the caches should be modified
 - This would require a constant-time cache, with no difference between hit/miss time
 - This would be a fundamental performance drop, making the caching architecture mostly pointless

Mitigations: the hard way

- Remove or restrict access to high-resolution timers such as `rdtsc`
 - unlikely: necessary to benchmark various hardware properties
- Allow certain memory to be marked as uncacheable
 - hardware challenge!
- Use AES-NI instructions in Intel chips to compute AES
 - but what about other encryption algorithms?
- Scatter-gather: secret data should not be allowed to influence memory access at coarser-than-cache-line granularity.
- Disable TSX (as they did!)
- Disable OOO pipelines (performance-critical!)

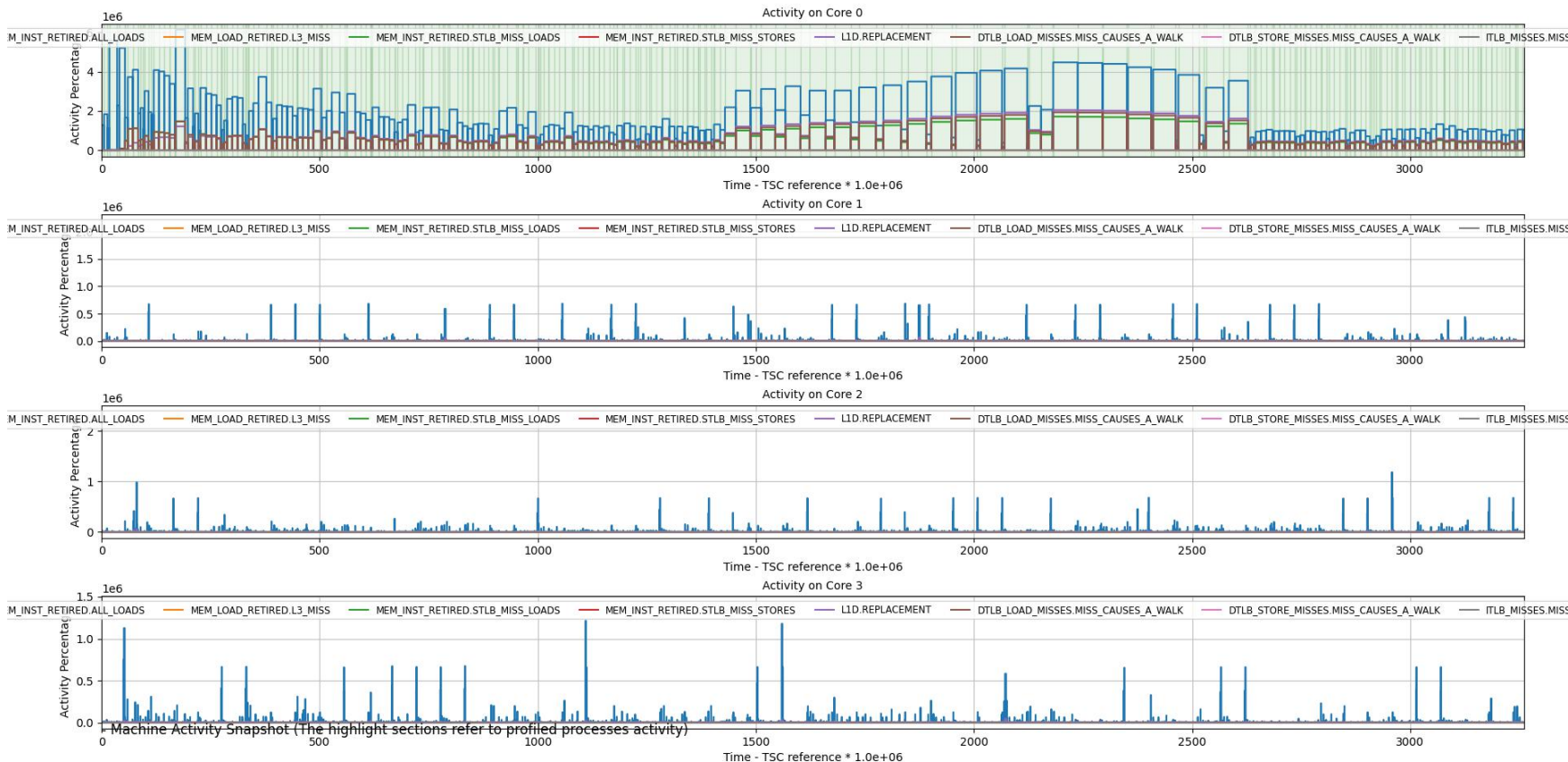
Mitigations to L1 attacks

- Cross-process L1 attacks are only possible when two separate threads are sharing the same L1
- L1 are private per-core
- Therefore, L1 attacks are only viable if relying on Simultaneous Multi-Threading
- The solution is therefore to turn off SMT

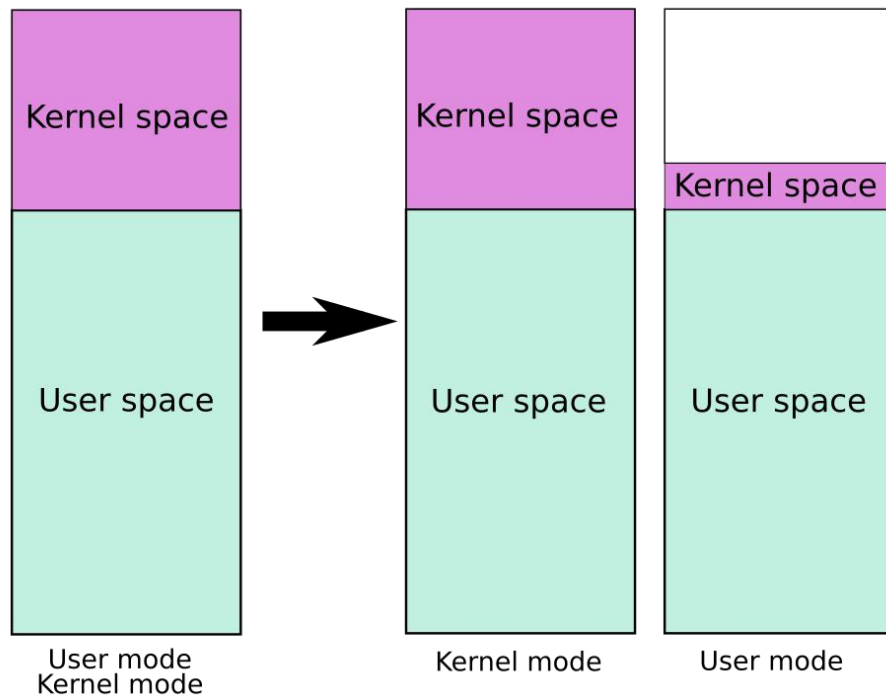
Mitigations: the detection way

- Setting up a side channel creates noise on the caching hierarchy
 - This is especially true for the *prepare* phase of some attacks
- Modern CPUs are equipped with Hardware Performance Counters
- They allow to track micro-architectural level events and count them
- They can be used to *observe* the behaviour of applications running in the system, and *selectively activate* countermeasures or mitigations
- Unfortunately, performance counters are not stable across vendors and different generations of CPUs

Mitigations: the detection way



Meltdown Mitigation: Kernel Page Table Isolation

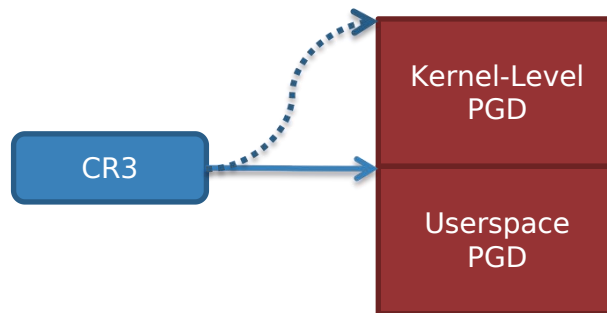


cpu_entry_area

```
struct cpu_entry_area {  
    char gdt[PAGE_SIZE];  
  
    struct entry_stack_page entry_stack_page;  
  
    struct tss_struct tss;  
  
    char exception_stacks[...];  
};  
  
DECLARE_PER_CPU(struct cpu_entry_area *, cpu_entry_area);
```

Double Page General Directory

- The first level of the page table is composed of a buffer of 8 KBs (two actual pages)
- One page is used to map the kernel-level memory view
- The other one is used to map the userspace memory view



CR3 is updated when transitioning to and from kernel mode

Switch CR3

/arch/x86/entry/entry_64.S:

SYM_CODE_START(entry_SYSCALL_64)

...

SWITCH_TO_KERNEL_CR3 scratch_reg=%rsp

...

SWITCH_TO_USER_CR3_STACK scratch_reg=%rdi

/arch/x86/entry/calling.h:

.macro SWITCH_TO_KERNEL_CR3 scratch_reg:req

mov %cr3, \scratch_reg

andq \$(~PTI_USER_PGTABLE_AND_PCID_MASK), \scratch_reg

mov \scratch_reg, %cr3

.endm

Retpoline

- The retpoline (*return trampoline*) is a software construct to prevent branch-target injection
 - Indirect branches are isolated from speculative execution
 - It is built around return instructions
 - Speculative execution will spin indefinitely
- This is a compiler-assisted technique, which prevents several indirect branch instructions to be emitted in a binary, and replaces them with thunks
 - A thunk is a subroutine used to delay a calculation until the result is actually needed
- Examples of avoided x86 instructions:
 - `jmp *%rax`
 - `call *%rax`

Retpoline thunks

```
jmp *%r11    jmp retpoline_r11_trampoline;  
call *%r11   call retpoline_r11_trampoline;
```

← Target known at compile time: will not trigger speculative target resolution

```
retpoline_r11_trampoline:
```

```
    call set_up_target
```

```
capture_spec: ←
```

← Speculation is captured here, which is a loop which will be eventually flushed

```
    pause
```

```
    jmp capture_spec
```

```
set_up_target:
```

```
    mov %r11, (%rsp) ←
```

← Modify the target of the return instruction on stack, leaving the Return Stack in the CPU unaffected

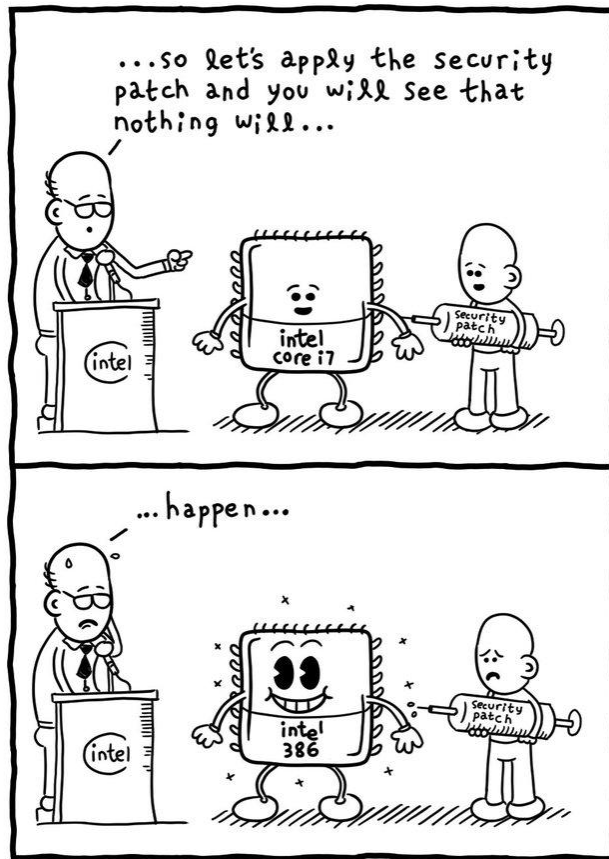
```
    ret
```

- In the code above, there is no point that can be controlled by an attacker
- There is one shared trampoline for each register in the final binary!

Prevent Branch Poisoning

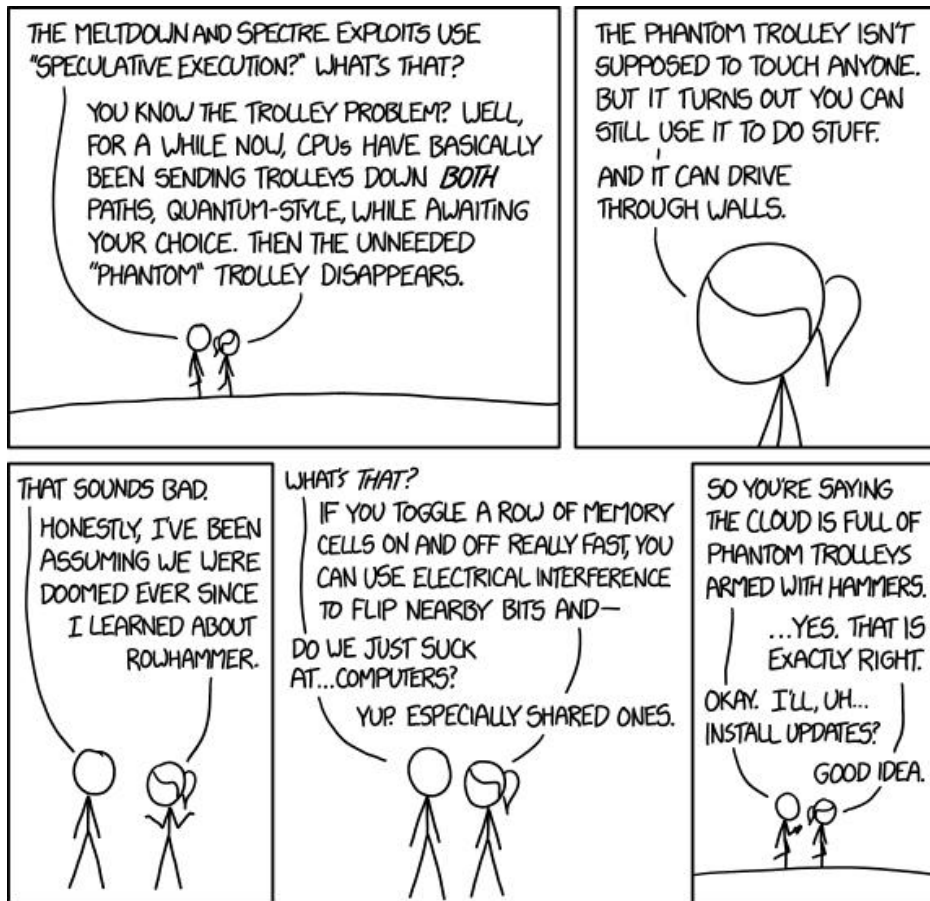
- Additional facilities by the ISA
- Indirect Branch Restricted Speculation (IBRS)
 - The processor enters in a special IBRS mode
 - BPU is not affected by predictions outside of IBRS
- Single Thread Indirect Branch Prediction (STIBP)
 - Restricts branch prediction sharing between software executing on hyperthreads of the same core
- Indirect Branch Predictor Barrier (IBPB)
 - Prevents software running before setting the barrier from affecting branch prediction by software running after the barrier
 - This is done by flushing the state of the Branch Target Buffer

How to make your CPU more secure

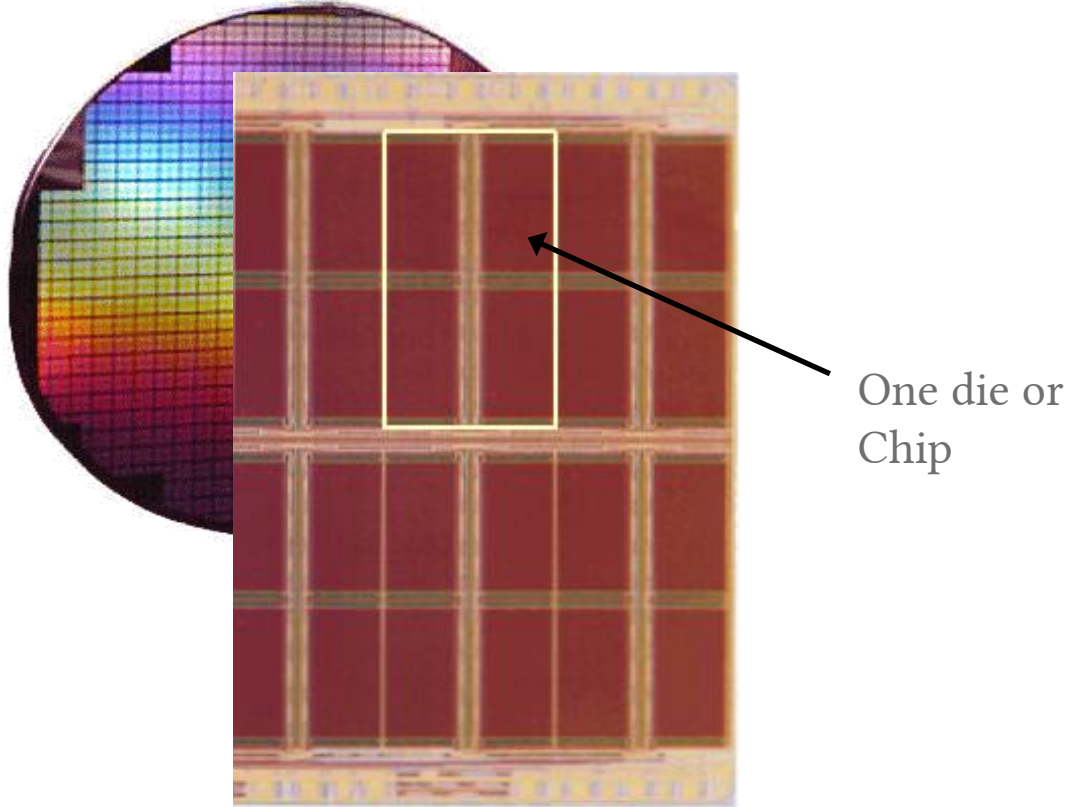


Daniel Stori {turnoff.us}

Setting up a Writing Primitive

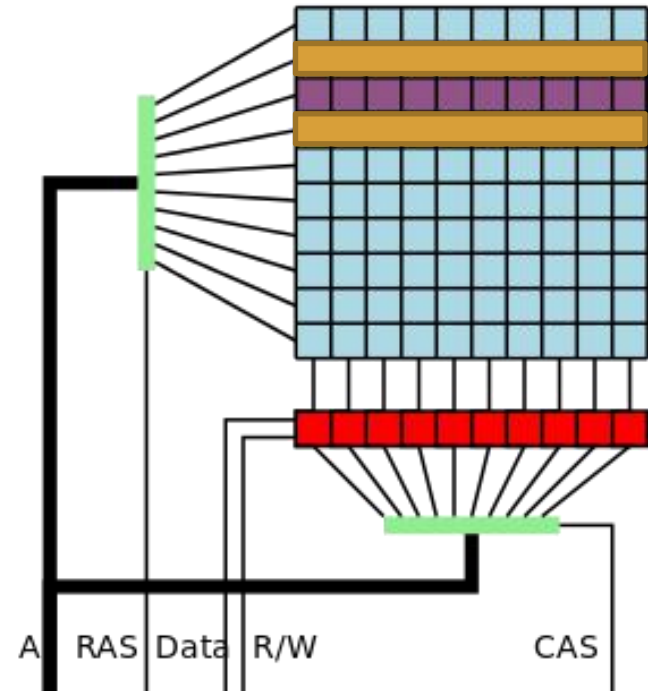


Anatomy of a Memory Chip



RowHammer

- Frequent row activations cause voltage fluctuations on the associated row selection lines
- These voltage fluctuations create an electromagnetic field (Lorentz force)
- This field can generate electric effects in nearby capacitors (adjacent, in most cases)
- Higher-than-natural discharge rates or charged capacitors
 - bit flips!



RowHammer POC

code1a:

```
mov (X), %eax    // read from address X
mov (Y), %ebx    // read from address Y
clflush (X)      // flush cache for address X
clflush (Y)      // flush cache for address Y
mfence
jmp code1a
```


Mitigations?

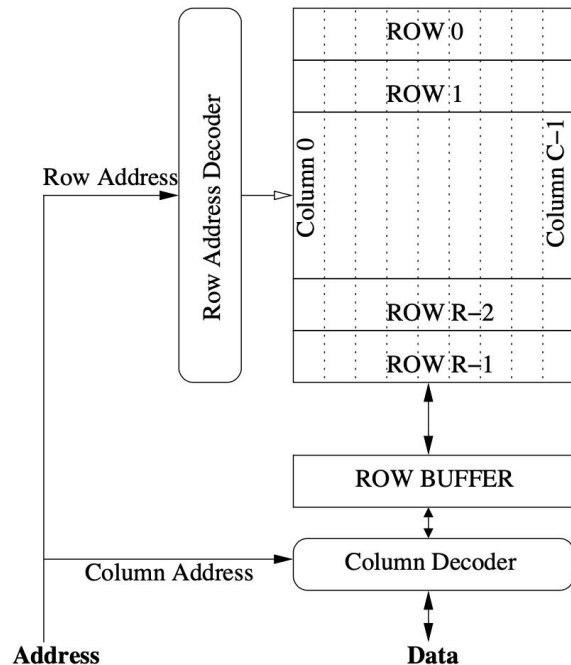
- Error correction codes in memory
 - might not detect multiple bit flips per memory word
- Reduce the 64 ms refresh interval
 - higher power consumption
 - increased processing overhead
- Detection: repeatedly accessing memory creates noise on cache
 - HPCs can be used to detect this noise as in side-channel attacks
- Pseudo Target Row Refresh (pTRR)
 - track row activation and refresh victim rows
 - implemented in the memory controller
 - “security through obscurity”

Memory Performance Attacks

Denial of Memory Service in Multi-
Core Systems

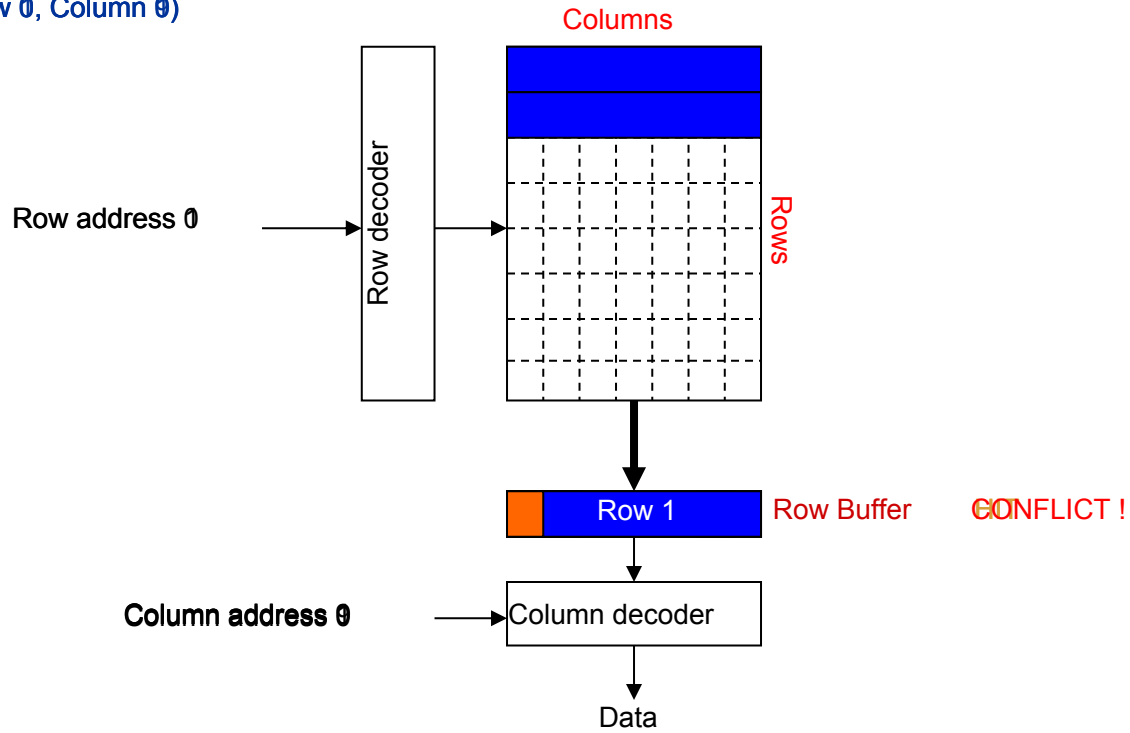
DRAM Memory Scheduler

- DRAM schedulers have been designed for single core chips!
- To improve speed, each memory bank has a *single* row buffer
- Memory access is performed only through this buffer
- Request hits or misses the current content of the row buffer

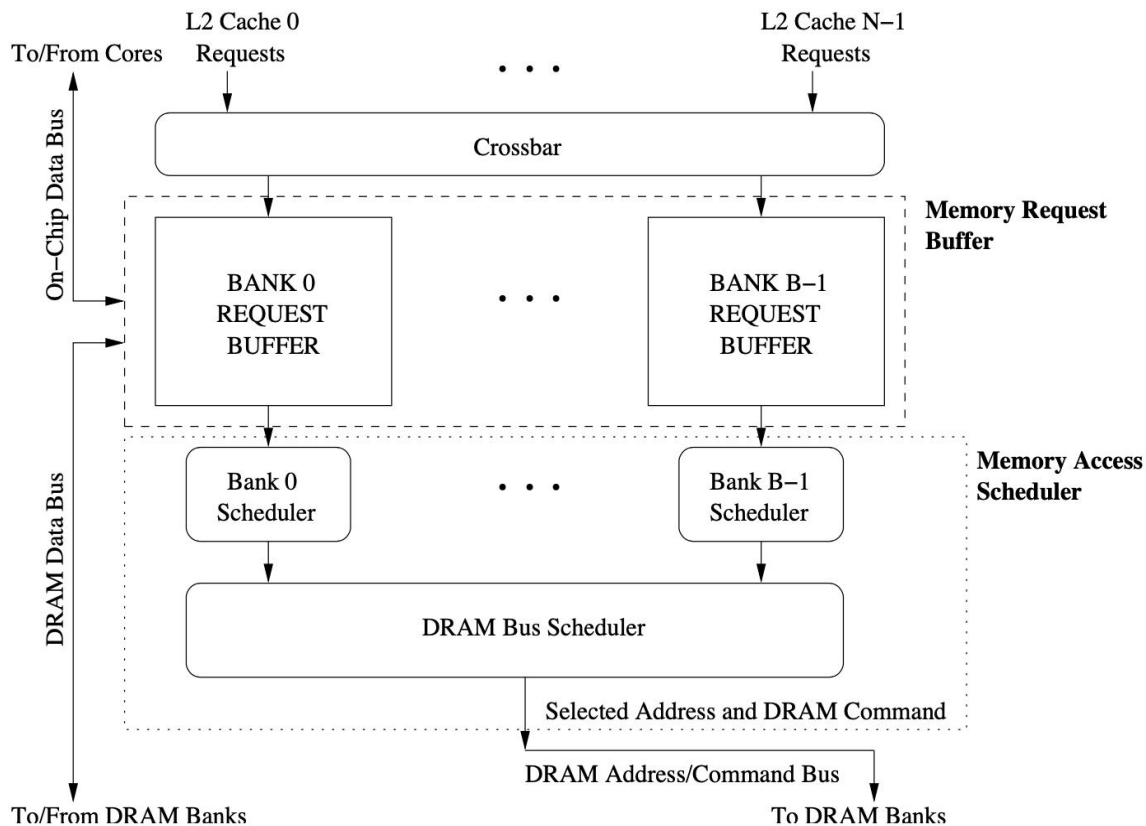


DRAM Bank Operations

Access Address
(Row 0, Column 0)



Multi-bank DRAM Memory Systems



DRAM Memory Access Scheduling: FR-FCFS

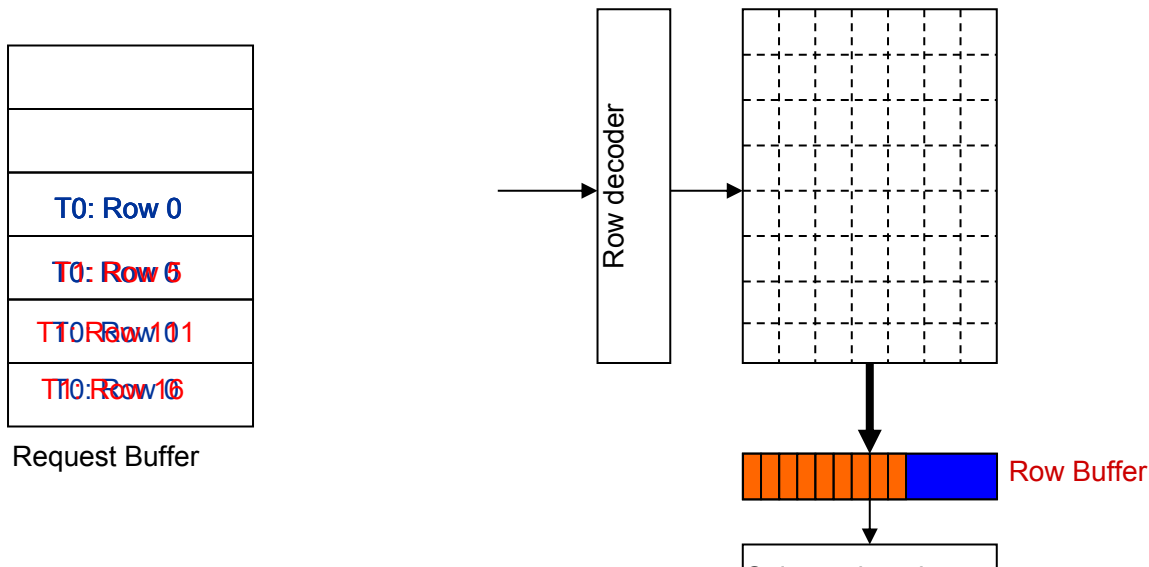
Bank scheduler:

- Row-hit-first
 - Low row-buffer locality threads have low priority
- Oldest-within-bank-first
 - Prioritizes threads that generate requests fast

Bus scheduler:

- Oldest-across-banks-first (among all requests proposed by individual bank scheduler)

An Aggressive, High Row-Buffer Locality Thread



It hogs DRAM!