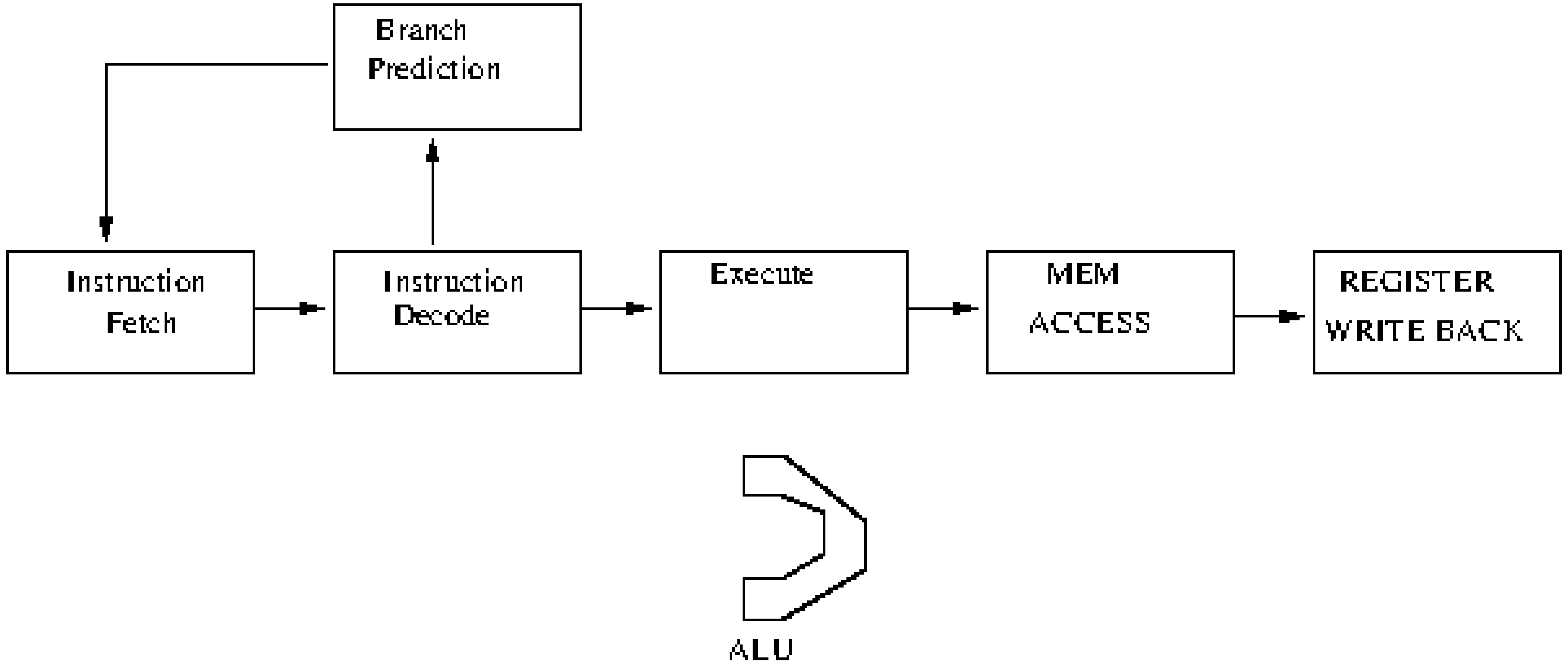
Micro-Architectural Attacks
Master SETI

Sumanta Chaudhuri

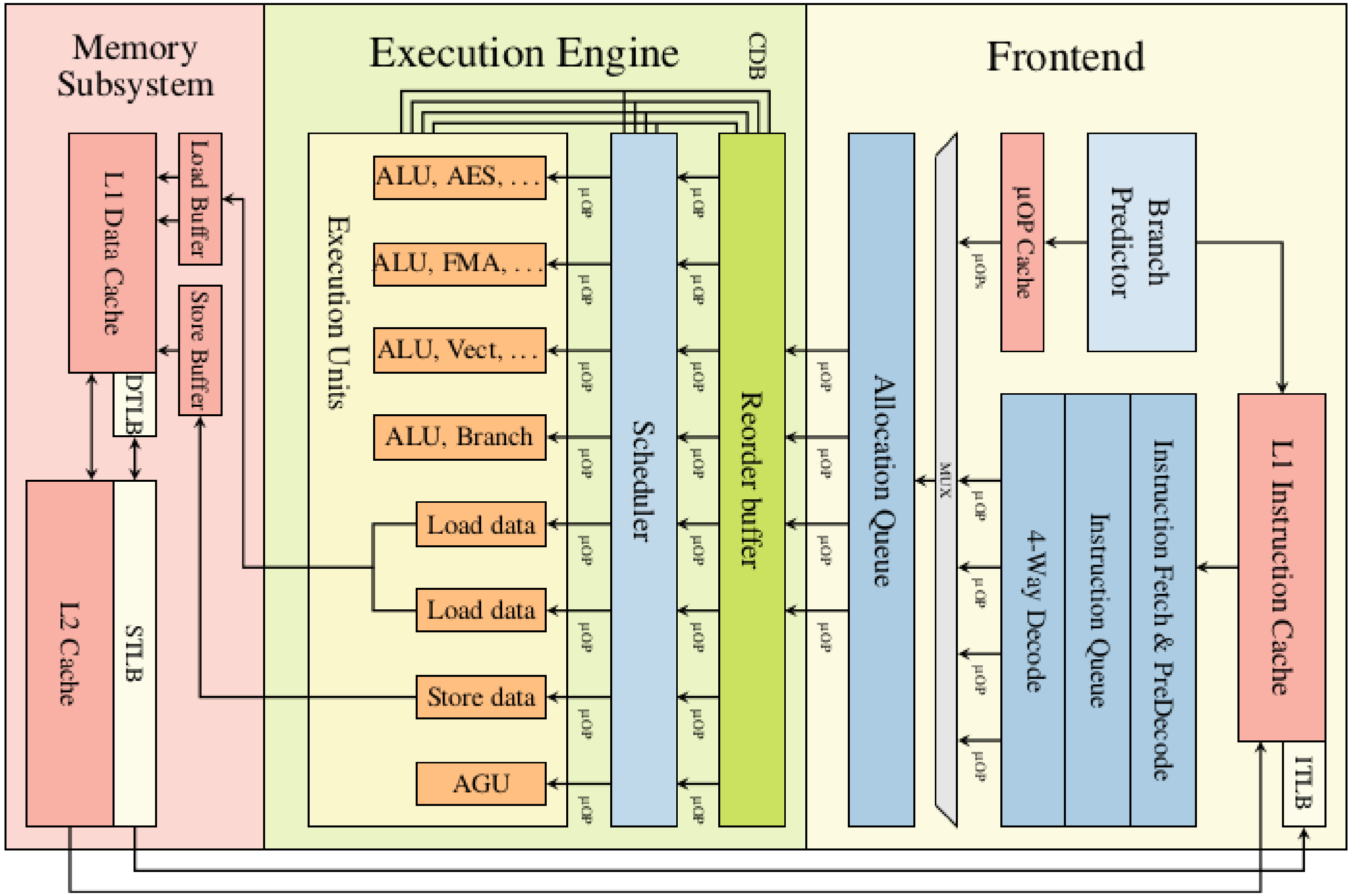
In-Order Processors

- Instruction Fetch (IF)
- Instruction Decode (ID)
- Execute (EX)
- Memory Access (MEM)
- Register Write Back (WB)

Basic gem5 In-Order Processor



Out of Order Processors



Out of Order Processors

- Multiple Insturction are fetched in parallel.
- Execute Instructions that are ready (I.e data available)
- Instructions are commited in-order using the reorder buffer

Out of Order Processors

- Hides Latency (Like Cache, & Multiple threads)
- Much More complex
- Security Hazards (As we will see later)

- Can not be done in compiler as compiler does not have runtime data.

Out of Order Processors

- Much longer look-ahead than in-order processors.

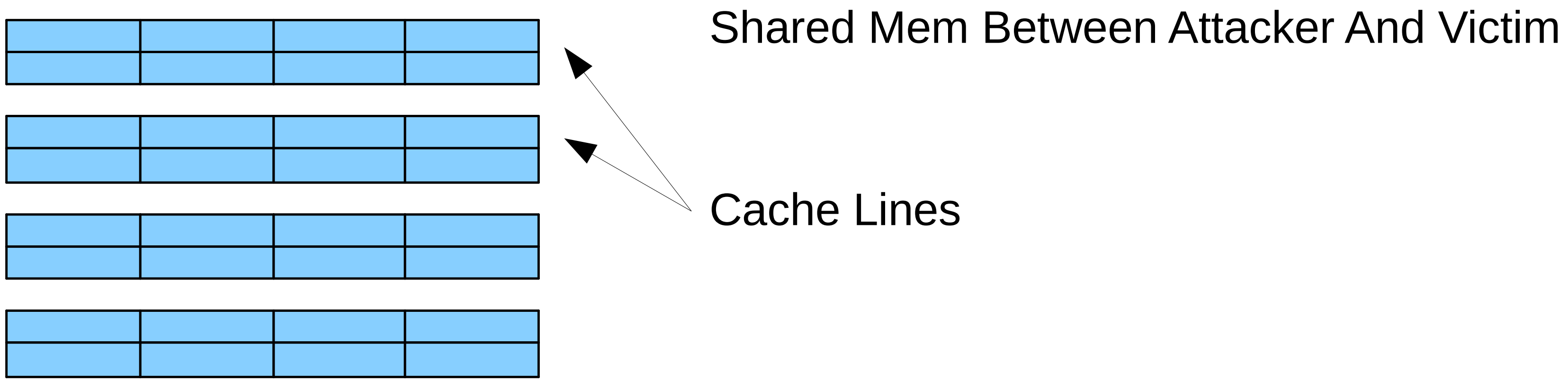
  (e.g 150-200 instructions , Intel)
- More complex branch predictor
  - Need to predict multiple branches.
  - Several Strategies
    - Static
    - Dynamic (based on the acces pattern)
    - Misprediction Penalty: In case of misprediction all instruction of that branch have to be flushed.

Micro-Architectural Attacks

Side Channels

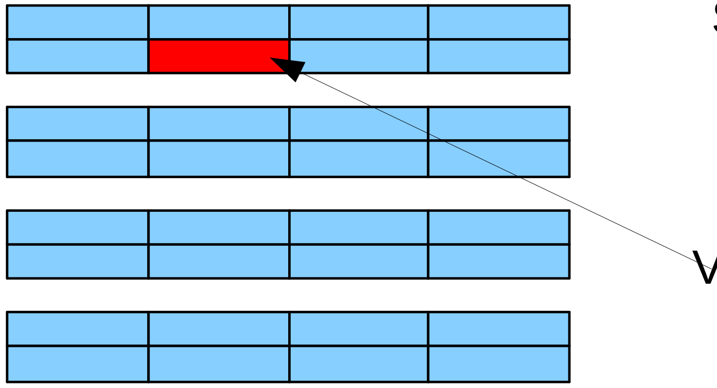
- Side Channels: Unintended Information leakage
  - e.g power consumption, EM emission
- Cache hit/miss Side Channels
  - The difference in hit and miss latency communicates the presence of a certain data in the shared cache.
- Speculative Execution Side Channels
  - Information leaked through speculative execution (e.g branch prediction)

Flush+Reload



Attacker **flushes** the whole array from the cache
#include <intrin.h>
__mm_clflush(array);

Flush+Reload

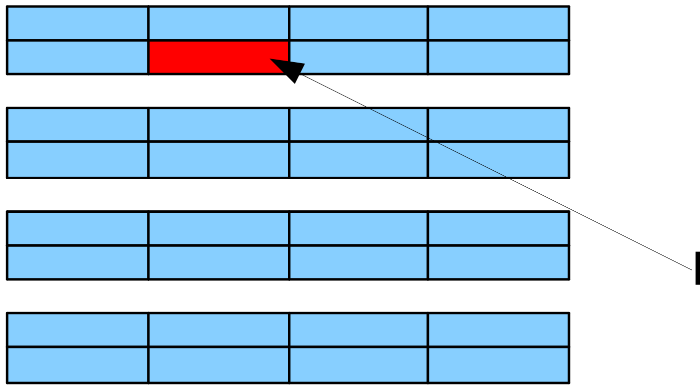


Shared Mem Between Attacker And Victim

Victim acces : bring line into cache

Victim Acceses the shared cache

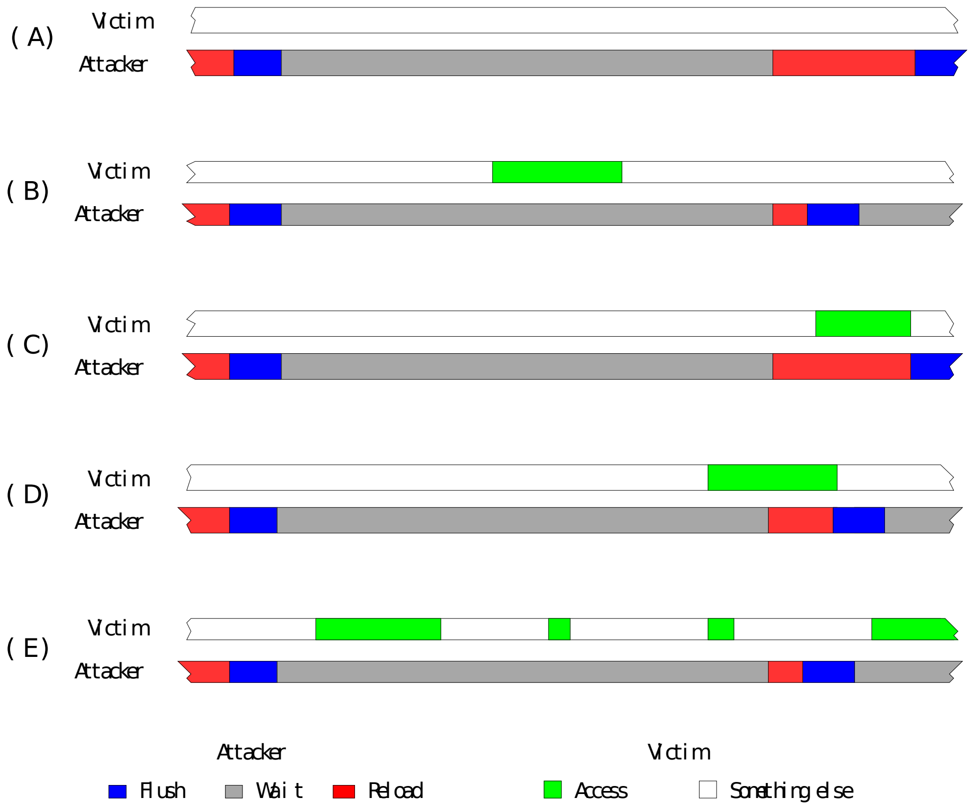
Flush+Reload



Shared Mem Between Attacker And Victim

Attackere **Re-acceses** the shared cache, Low access time due to cache hit; Measures access time
time1 =__rdtscp(array)

Flush+Reload



Flush+Reload Victim

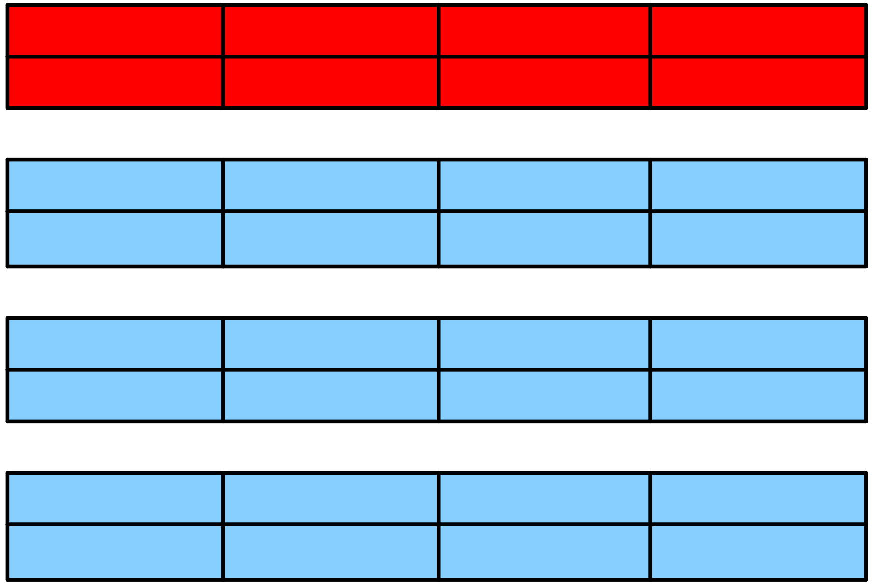
```
function exponent(b, e, m)
begin
 x    1
 for i    ♣e♣  1 downto 0 do
  x    $x^2$
  x    x mod m
  if ($e_i$ = 1) then
   x    xb
   x    x mod m
  endif
 done
 return x
end
```

Flush+Reload Victim

- The instruction for RSA is in shared memory.
- The Bit ‘1’ and bit ‘0’ of exponent can be determined from instructions executed.
- Let’s do flush+reload on the instruction cache.
  - Square+Modulo, Multiply+modulo : bit ‘1’
  - Square+Modulo : bit ‘0’

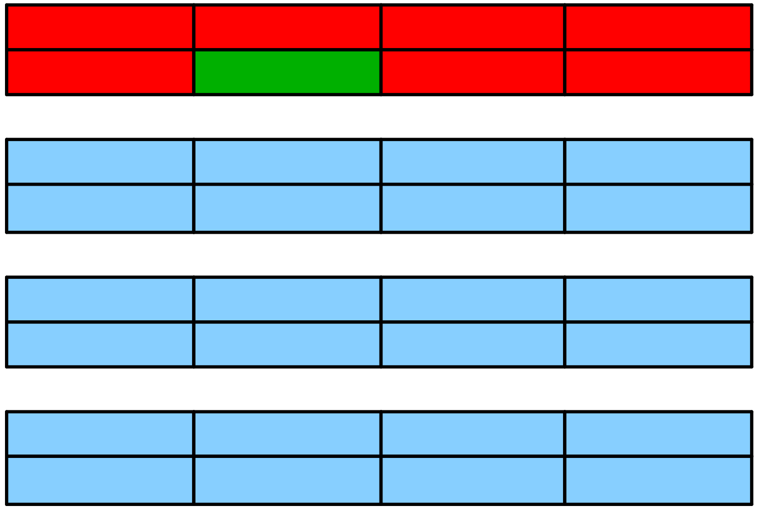
Prime+Probe

- Attacker fills the Cache with an array
-
- Tries to use the same cache lines as the victim
-
- Victim does not acces the cache.
-
- Attacker Probes (reads and measure timing for his own array)
-
-
- Guesses that Victim has not accessed the target memory location

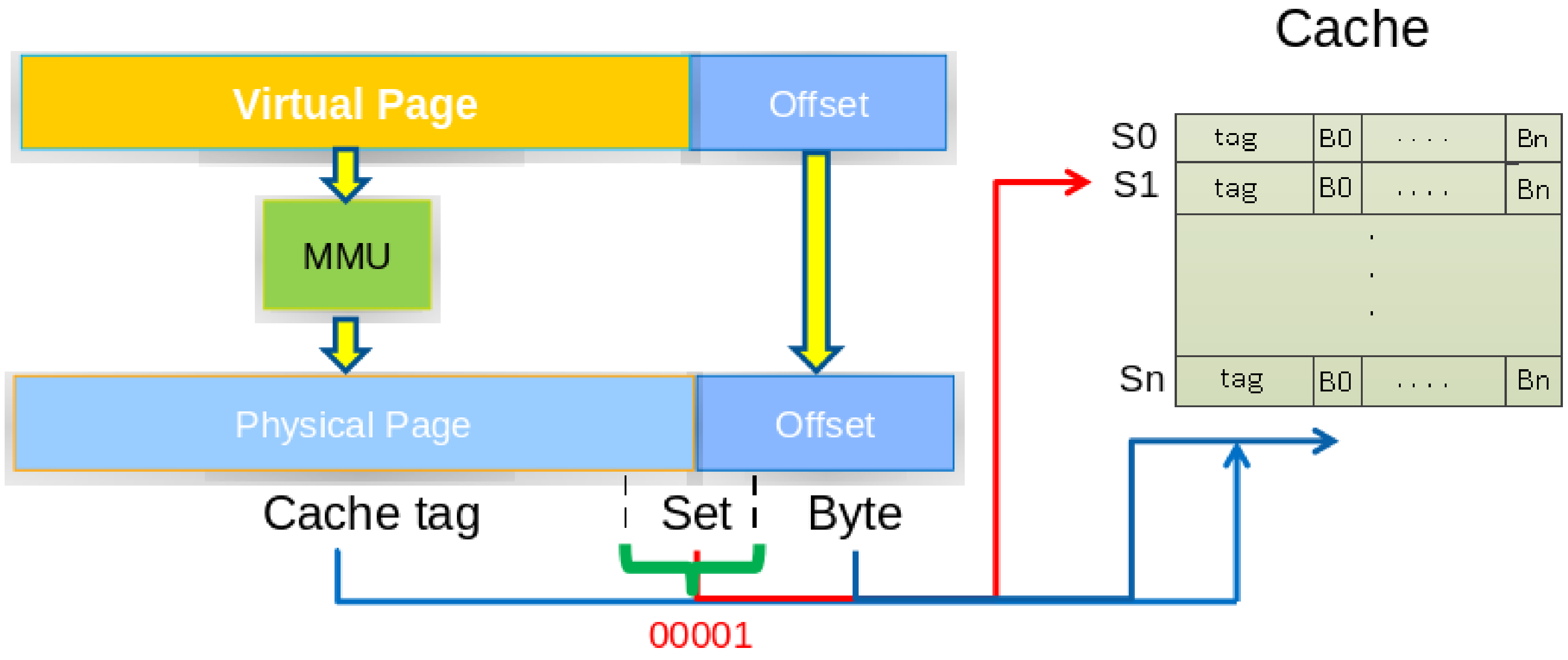


Prime+Probe

- Attacker fills the Cache with an array. (Prime)
- Tries to use the same cache lines as the victim
- Victim access the cache.
- Attacker Probes (reads and measure timing for his own array)
- Guesses that Victim has accessed the target memory location because it has evicted his own array cache line.

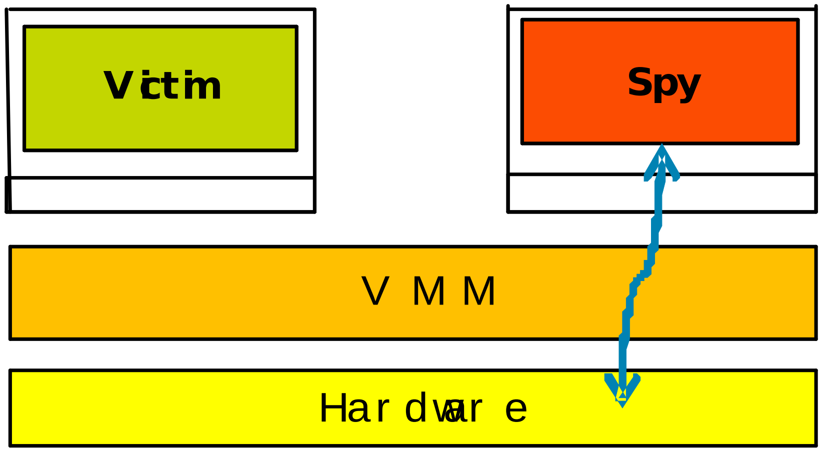


Prime+Probe



How to calculate the eviction sets ?
What is the offset for a cache line size of 64 bytes ?

Attack Scenario



Other Attack Scenarios can be Javascript running on browser

CounterMeasures

- Prohibit clflush() instruction for users.
- Prevent Sharing pages between victim and attacker code.
  - Page sharing is good for memory use.

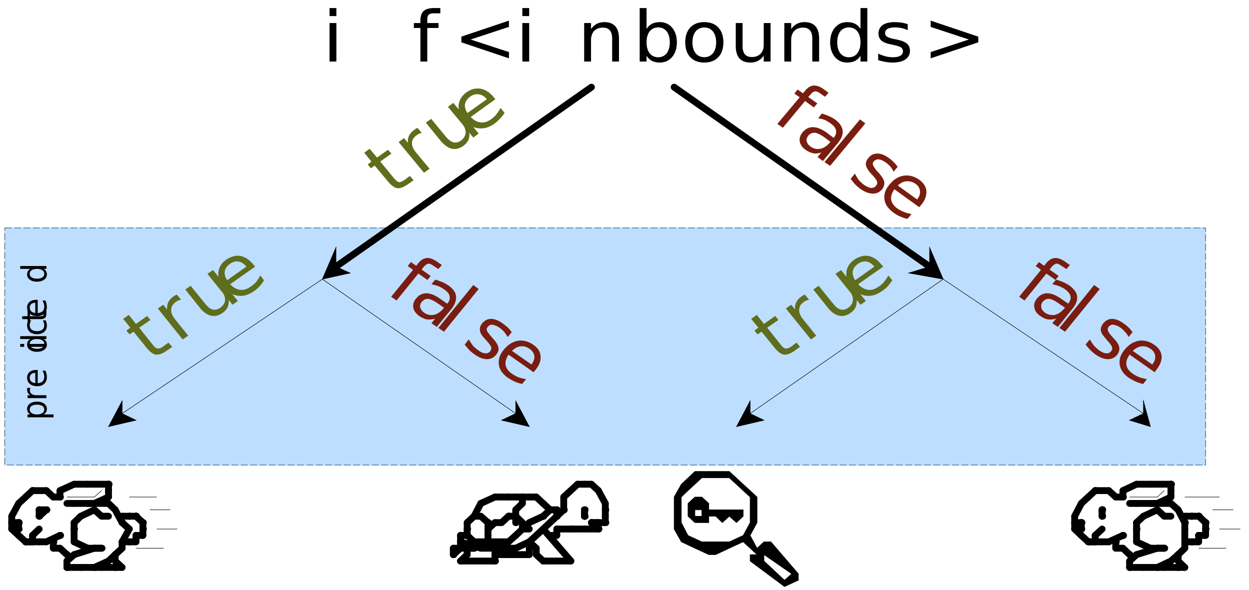
- Detect cache attack behaviour at runtime.
- Write code to avoid cache side channel dependencies

Spectre

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

- Speculative execution of branch even when x > array1_size.
- Train the branch predictor for some iterations. Force it to mispredict.

Spectre



Spectre

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

To attack

- victim_address=array1+x
- So → x=victim_address-array1
- The array2 index accessed is the value stored in victim_address.

Spectre

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

To attack

- Find out the **array2 index accessed** with **Flush+Reload**
- Why do we need to multiply by a stride of 64 ?

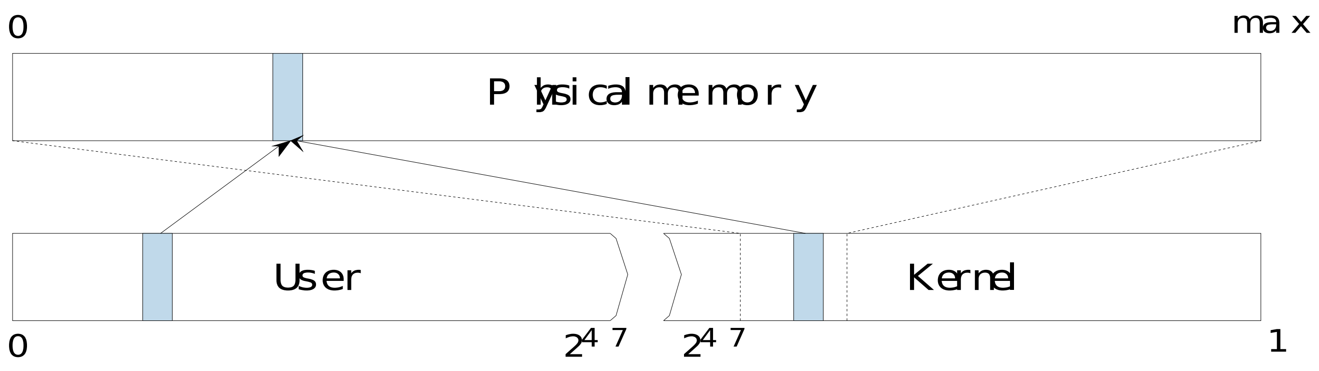
Spectre Mitigations

- All Out-of-Order Processors are affected by spectre.
- However it is harder to exploit. Need to find code pattern in the victim:

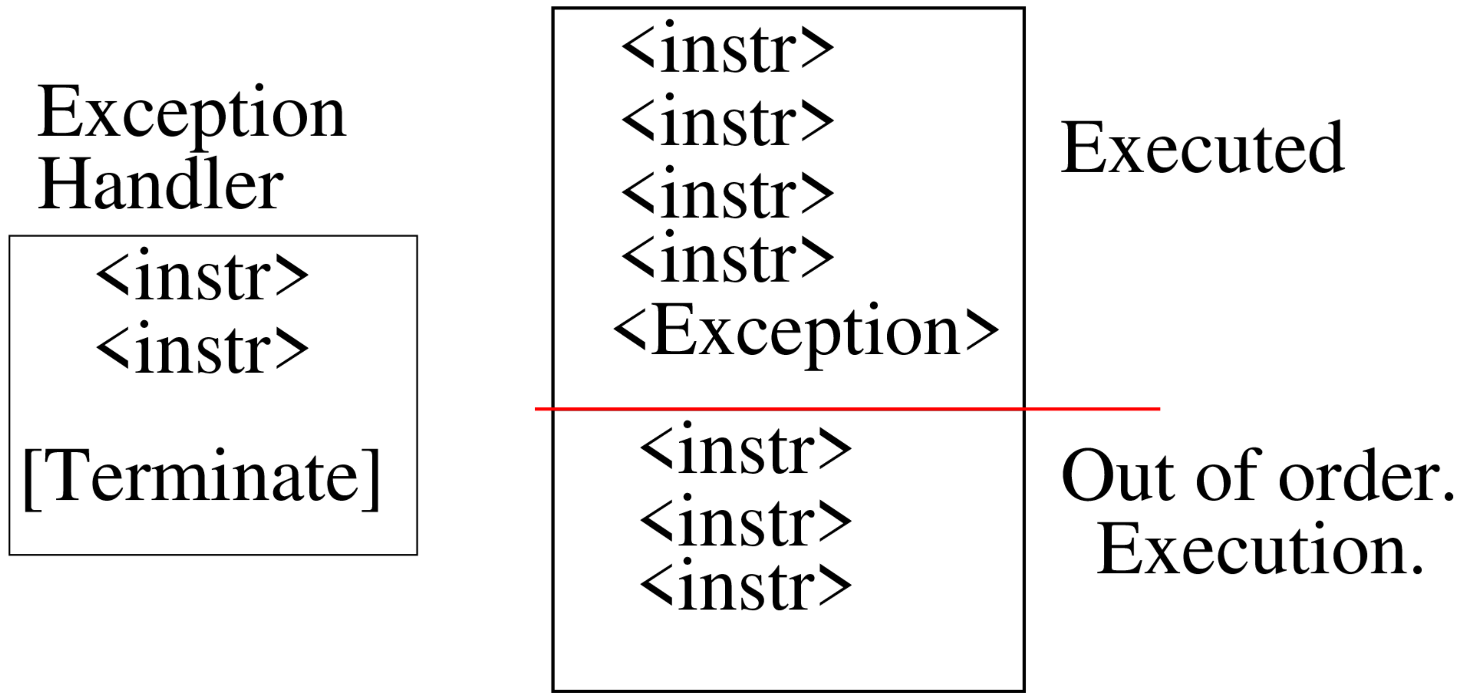
```
if (x < array1_size)
        y = array2[array1[x] * 4096];
```

MeltDown

- Reading Kernel Memory from User Space



MeltDown


Exception
Handler
<instr>
<instr>
[Terminate]
<instr>
<instr>
<instr>
<instr>
<Exception>
<instr>
<instr>
<instr>
Executed
Out of order.
Execution.

Meltdown

1. raise_exception();
2. // the line below is never reached
3. access(probe_array[data * 4096]);

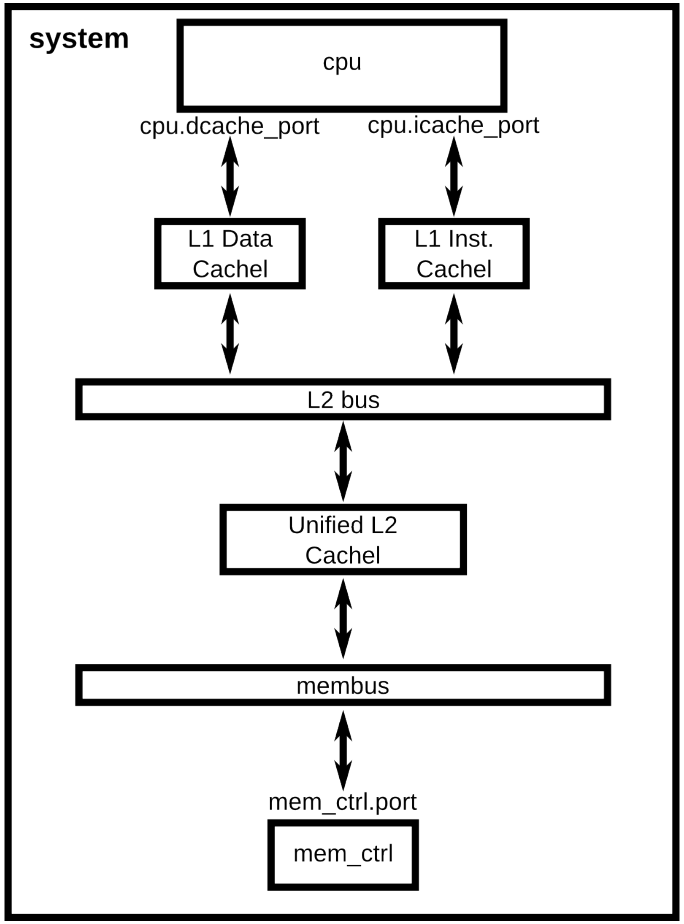
Spill over to the Kernel memory space.

Find the value through Flush+Reload.

MeltDown Mitigations

- KAISER Patch: User space does not have access to kernel memory.
- KASLR (Address space layout randomization): Makes the attack difficult.

TP: CONFIG


system
cpu
cpu.dcache_port
cpu.icache_port
L1 Data Cachel
L1 Inst. Cachel
L2 bus
Unified L2 Cachel
membus
mem_ctrl.port
mem_ctrl

TP: STEP 1

- Clone the repository https://github.com/amusant/micro_archi_attacks

- $source env.sh → sets up environment variables.

- Go to directory hit_miss; look into code hit_miss.c

- Run make to compile the code in hit_miss directory

- Runs $make launch to launch simulation.
-
- We use the gem5 simulator to simulate a basic system with x86 processor and two levels of cache.

- Understand the code used for
  - Flush
  - Acces
  - Reload
- **By changing the acces pattern do you see any difference in the output ?**

- **What is the role of STRIDE, does the code still work after changing STRIDE ?**

TP: STEP 2

- Go to directory flush_reload; look into code flush_reload.c
-
- The function victim(k) does the following:
  - It accesses the **array[secret[desknumber][i]*STRIDE]**
  - Where the secret is a 16 character secret key.
    - **secret[desknumber]=”XXXXXXXXXXXXXXXX”**
  - Your goal is to find the 16 characters of the secret value.
  - The secret value changes with desk number.

- Run make to compile the code in flush_reload directory

- Runs $make launch to launch simulation.

- **Inspire yourself from the hit_miss code.**

TP: STEP 3

- Download the Spectre Example here
- https://gist.github.com/ErikAugust/724d4a969fb2c6ae1bbd7b2a9e3d4bb6
-
- Read and Understand the code.
- Compile the code
  - $**gcc spectre.c**
  - Launch the experiment
    - $ **gem5.opt ../configs/two_level.py ./a.out**
    - Does it work ?
- Change line 99 in **../configs/two_level.py**
  - from DerivO3CPU() to TimingSimpleCPU()
  - - relaunch simulation
  - - Does it work ?