



Breaking the Walls

CPU introspection through
micro-architectural data sampling

by pietroborrello

inspired by: [[CPU Introspection: Intel Load Port Snooping](#)]



Let's play a game

observe
instructions
executed

The
Roman
Xploit



Let's play a game

observe
instructions
executed

The
Roman
Xploit





Let's play a game

observe
instructions
executed

observe
http
interactions

The
Roman
Xploit





Let's play a game



observe
instructions
executed

observe
http
interactions





Let's play a game



observe
instructions
executed

observe
http
interactions

observe
network
packets





Let's play a game



observe
instructions
executed

observe
http
interactions

observe
network
packets





Let's play a game

The
Roman
Xploit

observe
instructions
executed

observe
http
interactions

observe
network
packets

observe
CPU
micro-arch





Let's play a game

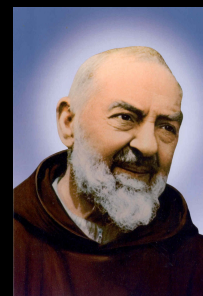
The Roman
Xploit

observe
instructions
executed

observe
http
interactions

observe
network
packets

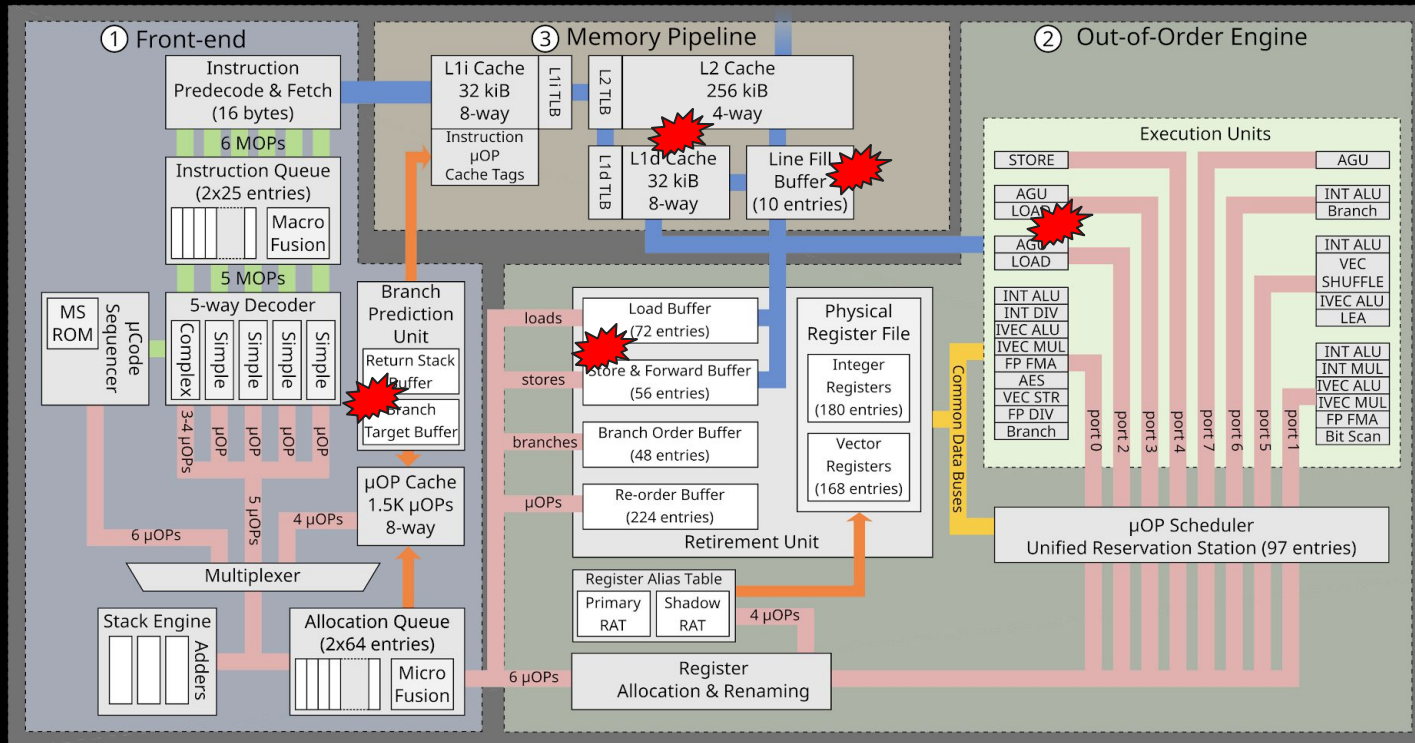
observe
CPU
micro-arch





Modern CPUs are: broken

Meltdown, Spectre, Foreshadow, MDS, ...





Modern CPUs are: undocumented



What do CPUs do:

- during speculation
- during out of order execution
- during microcode assists
- during faulting operations
- during *normal execution*



Modern CPUs are: undocumented



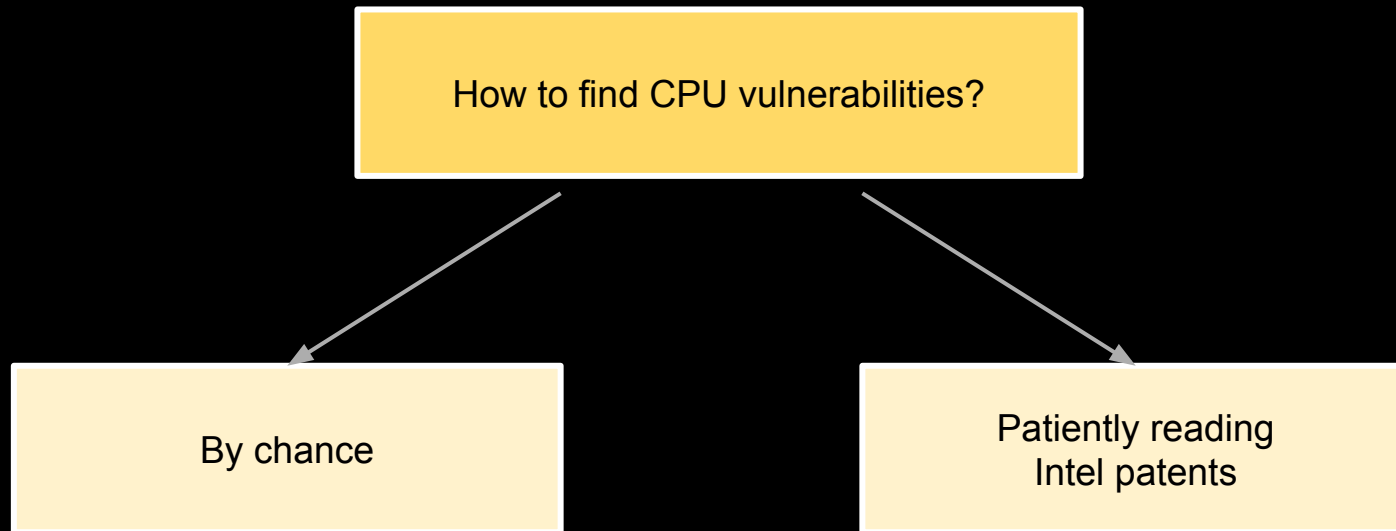
What do CPUs do:

- during speculation
- during out of order execution
- during microcode assists
- during faulting operations
- during *normal execution*





Modern CPUs may improve



Lack of precise CPU introspection utilities



Modern CPUs may improve



How to find CPU vulnerabilities?

How to understand inner CPU behavior?



Modern CPUs may improve



How to find CPU vulnerabilities?

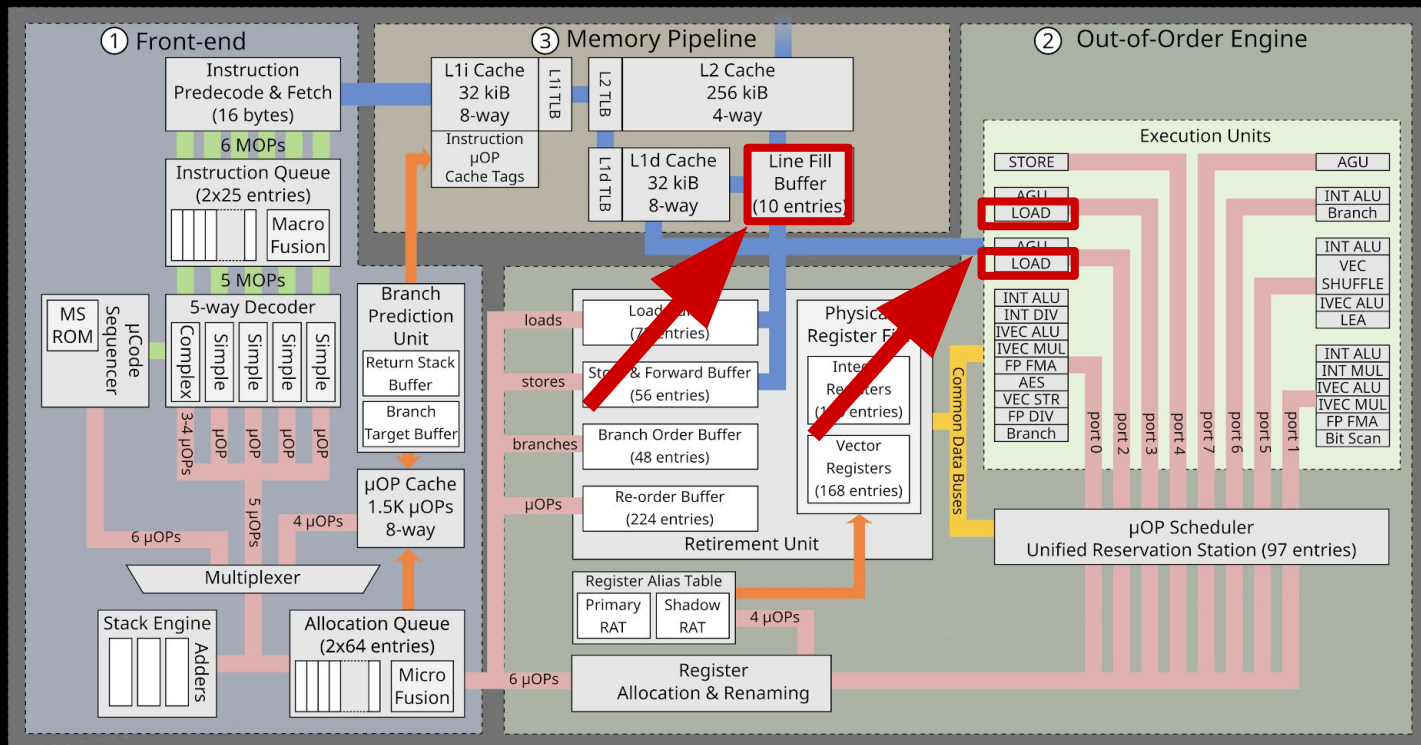
How to understand inner CPU behavior?



Let's leverage CPU vulnerabilities!



Microarchitectural Data Sampling





Microarchitectural Data Sampling



1. Cause a fault during a memory load which requires a microcode assist
2. New data will not be loaded, so the previous value that was filled in the LFB entry (or in the load port) remains unchanged
3. Access the data transiently and leak it through a side channel

```
1. char array[256 * 4096]
2. flush all array cache lines
3. X = *(char *)(ptr)
4. tmp = array[X * 4096]
```

```
1. handle SIGSEGV
2. for(i = 0; i < 256; i++)
    measureTime(array[i*4096])
3. The index with fastest access
    corresponds to X
```



Microarchitectural Data Sampling



So we are able to leak stale data recently transferred in the CPU by sibling threads

⇒ We can observe *any data* is passing through the CPU



Microarchitectural Data Sampling



So we are able to leak stale data recently transferred in the CPU by sibling threads

⇒ We can observe *any data* is passing through the CPU

- any data



Microarchitectural Data Sampling



So we are able to leak stale data recently transferred in the CPU by sibling threads

⇒ We can observe *any data* is passing through the CPU

- any data
 - data loaded speculatively



Microarchitectural Data Sampling



So we are able to leak stale data recently transferred in the CPU by sibling threads

⇒ We can observe *any data* is passing through the CPU

- any data
 - data loaded speculatively
 - data loaded by microcode assists



Microarchitectural Data Sampling



So we are able to leak stale data recently transferred in the CPU by sibling threads

⇒ We can observe *any data* is passing through the CPU

- any data
 - data loaded speculatively
 - data loaded by microcode assists
 - data loaded by faulting instructions



Microarchitectural Data Sampling



So we are able to leak stale data recently transferred in the CPU by sibling threads

⇒ We can observe *any data* is passing through the CPU

- any data
 - data loaded speculatively
 - data loaded by microcode assists
 - data loaded by faulting instructions

WHAT : ✓



Microarchitectural Data Sampling



So we are able to leak stale data recently transferred in the CPU by sibling threads

⇒ We can observe *any data* is passing through the CPU

- any data
 - data loaded speculatively
 - data loaded by microcode assists
 - data loaded by faulting instructions

WHAT : ✓

WHEN: ✗



Timing inference

We leak the data we are interested in:

⇒ We know what data the CPU is actually using, but still not its order

0xbd09
0x4143
0xf178
0xe4e7
0x4036

0xe4e7
0x4143
0xbd09
0x4036
0xf178

0xbd09
0x4036
0xf178
0xe4e7
0x4143



Timing inference



We leak the data we are interested in:

⇒ We know what data the CPU is actually using, but still not its order

0xbd09
0x4143
0xf178
0xe4e7
0x4036

0xe4e7
0x4143
0xbd09
0x4036
0xf178

0xbd09
0x4036
0xf178
0xe4e7
0x4143

Use statistics!



Core synchronization



Monitoring core

Victim Core



Core synchronization



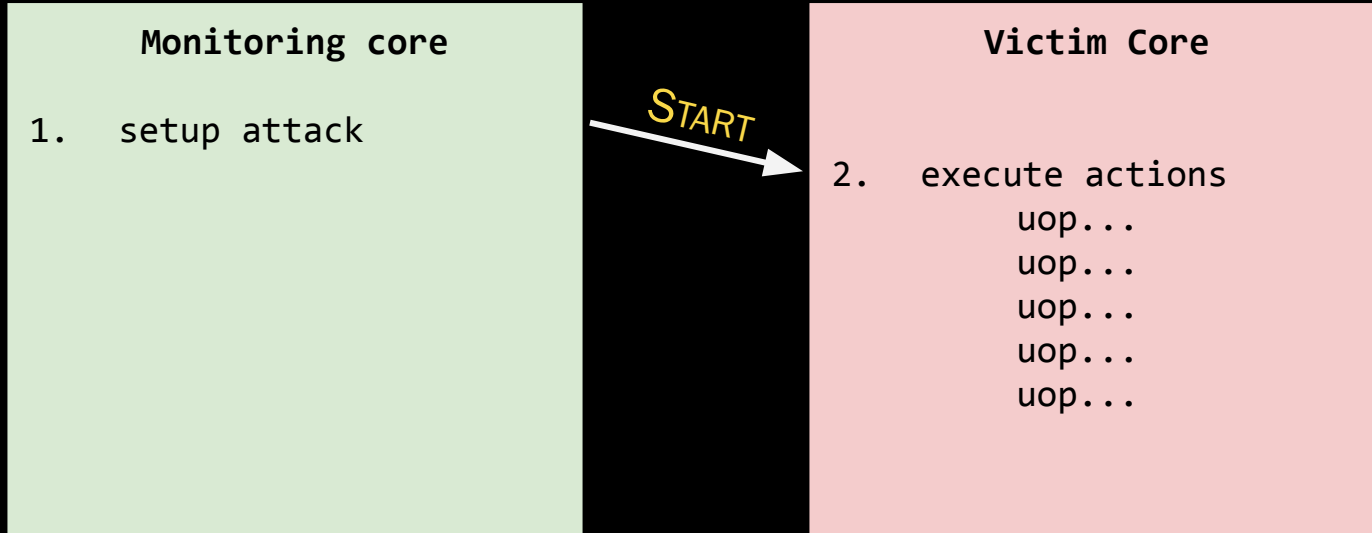
Monitoring core

1. setup attack

Victim Core



Core synchronization





Core synchronization

Monitoring core

1. setup attack
2. wait i clock cycles



Victim Core

2. execute actions
uop...
uop...
uop...
uop...
uop...



Core synchronization

Monitoring core

1. setup attack
2. wait i clock cycles
 - i. mds attack

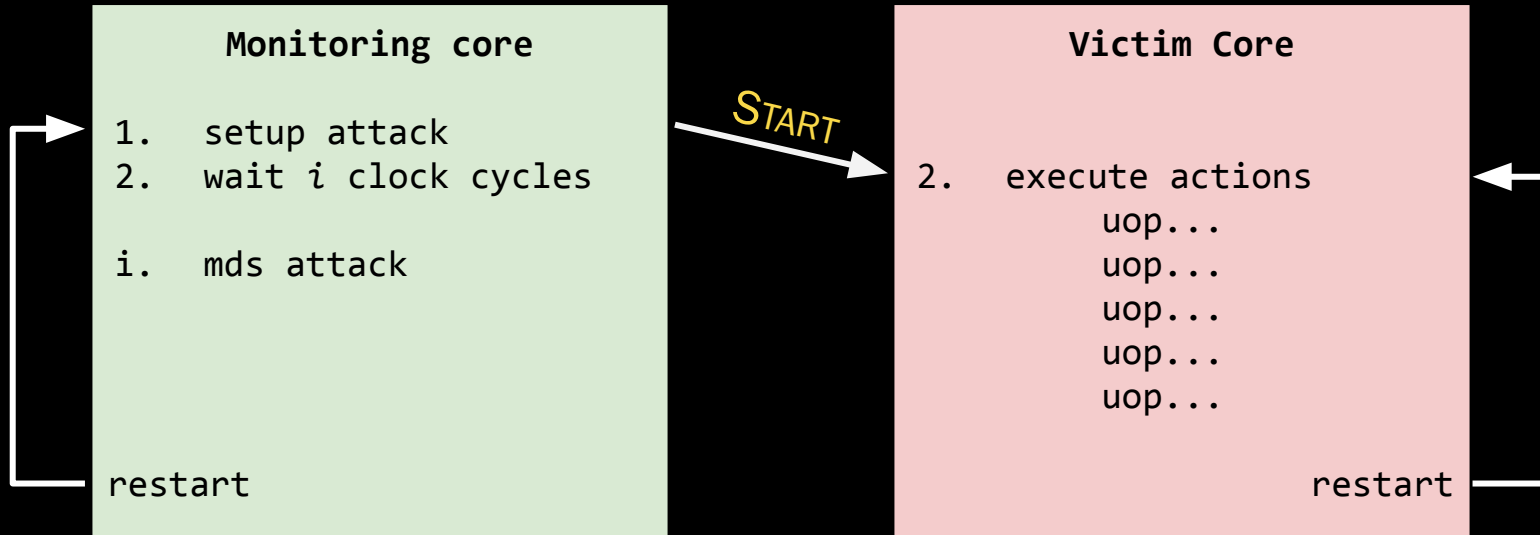
START

Victim Core

2. execute actions
 - uop...
 - uop...
 - uop...
 - uop...
 - uop...

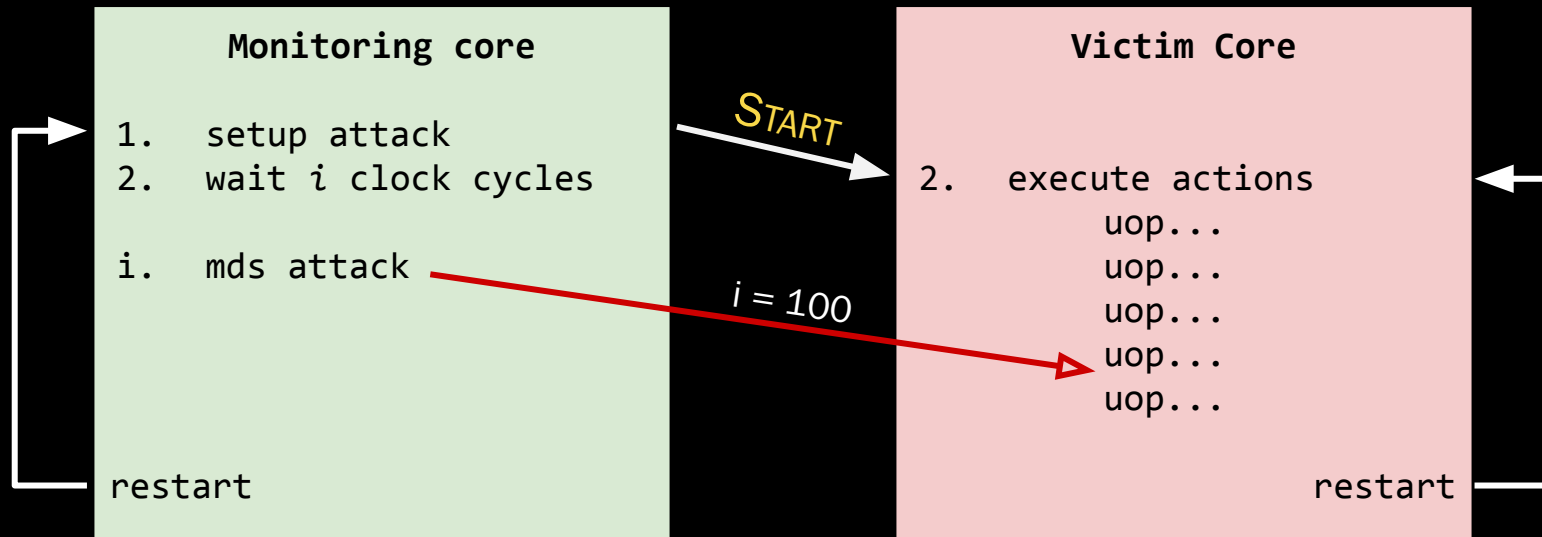


Core synchronization





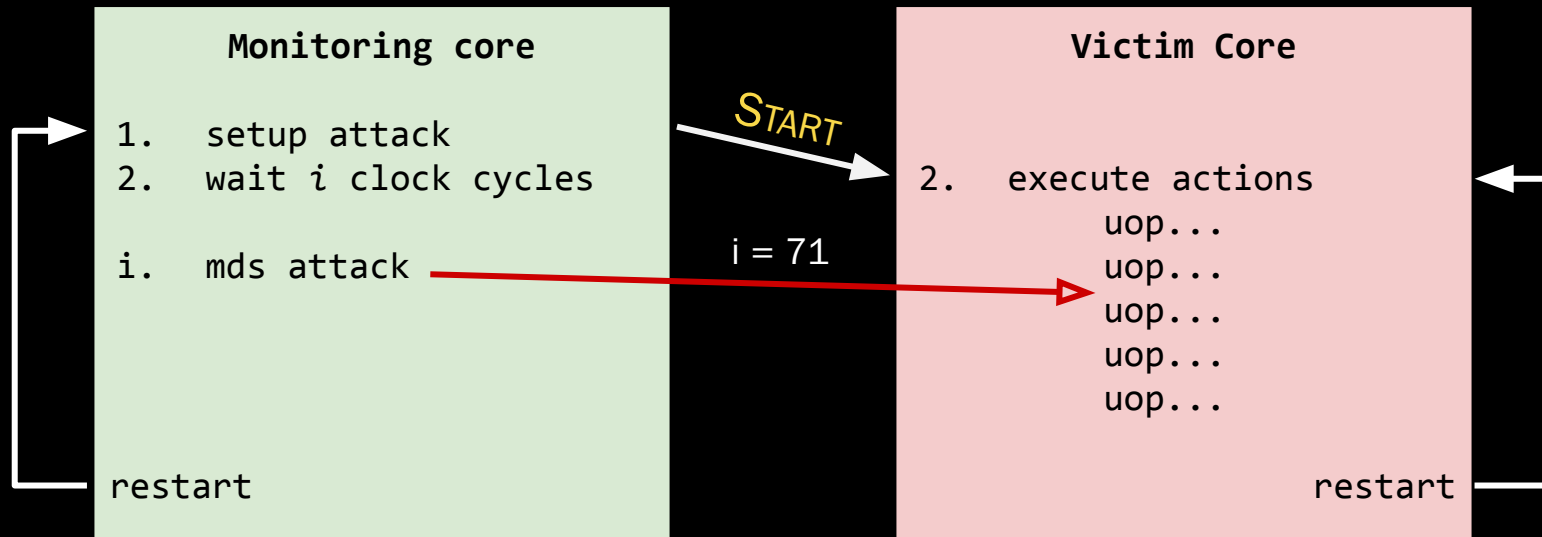
Core synchronization



leaks: (i: 100, 0x1337)



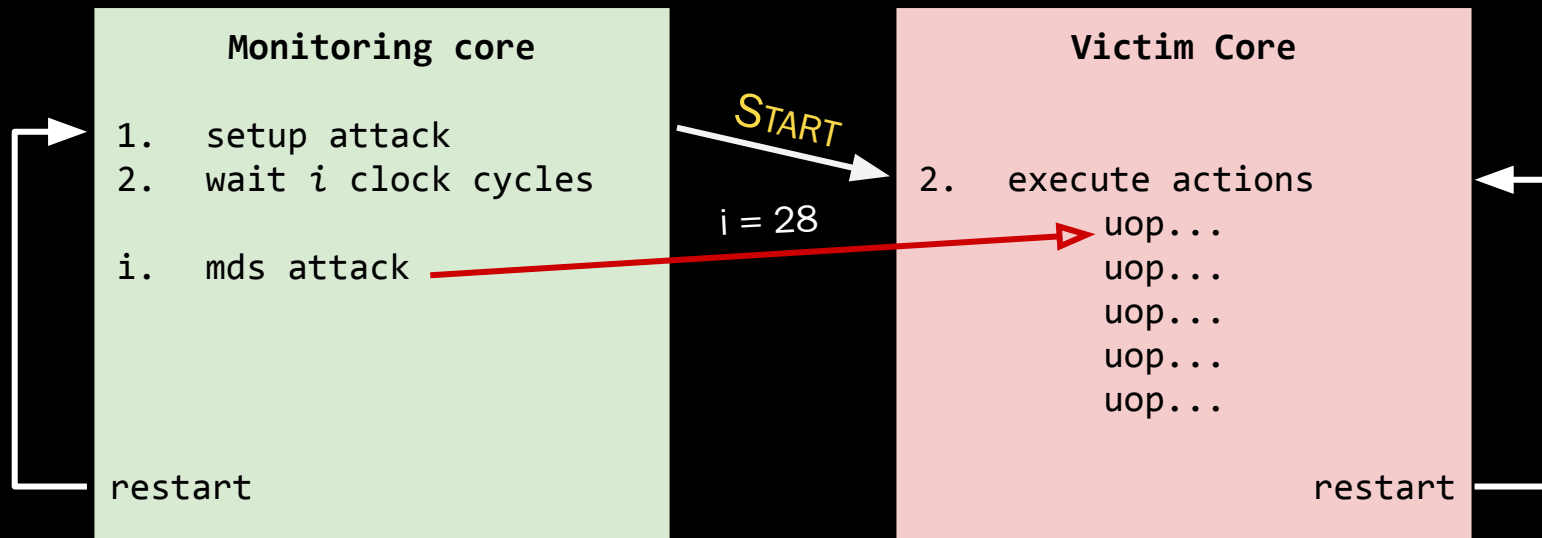
Core synchronization



leaks: (i: 100, 0x1337)
(i: 71, 0xcafe)



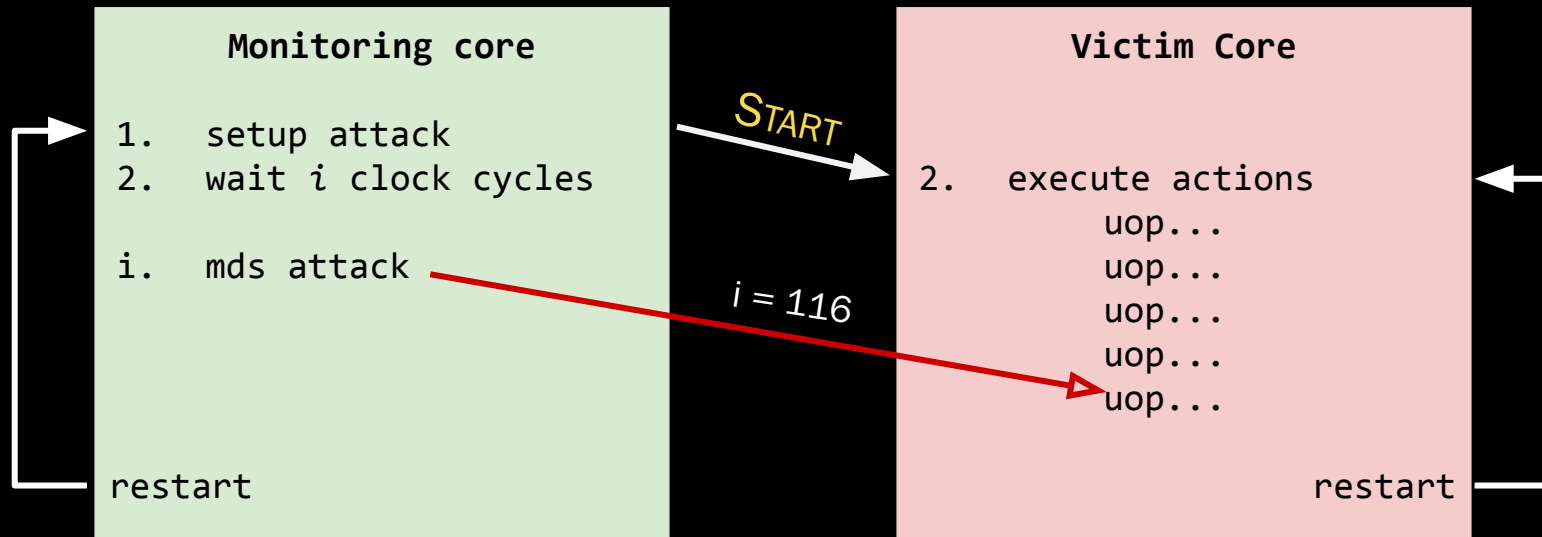
Core synchronization



leaks: (i: 100, 0x1337)
(i: 71, 0xcafe)
(i: 28, 0x1234)



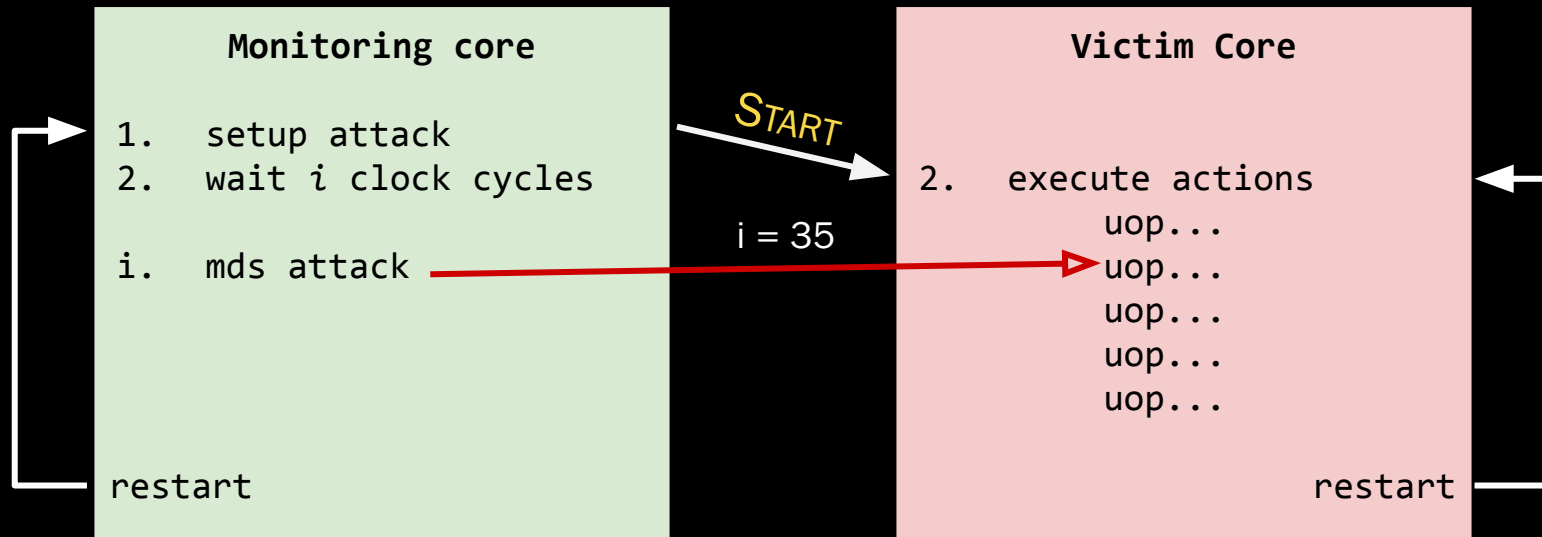
Core synchronization



leaks: (i: 100, 0x1337)
(i: 71, 0xcafe)
(i: 28, 0x1234)
(i: 116, 0x1337)



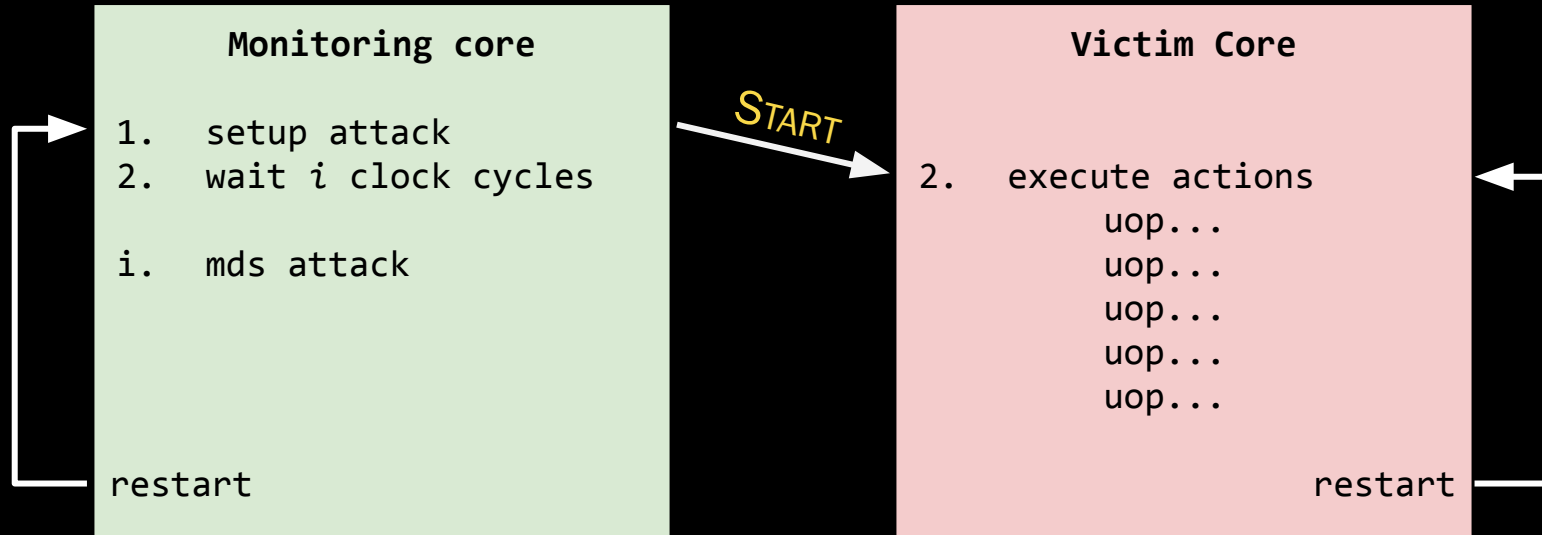
Core synchronization



leaks: (i: 100, 0x1337)
(i: 71, 0xcafe)
(i: 28, 0x1234)
(i: 116, 0x1337)
(i: 35, 0x1234)



Core synchronization



```
leaks: (i: 100, 0x1337)
       (i: 71, 0xcafe)
       (i: 28, 0x1234)
       (i: 116, 0x1337)
       (i: 35, 0x1234) ... millions of tries
```



Data analysis

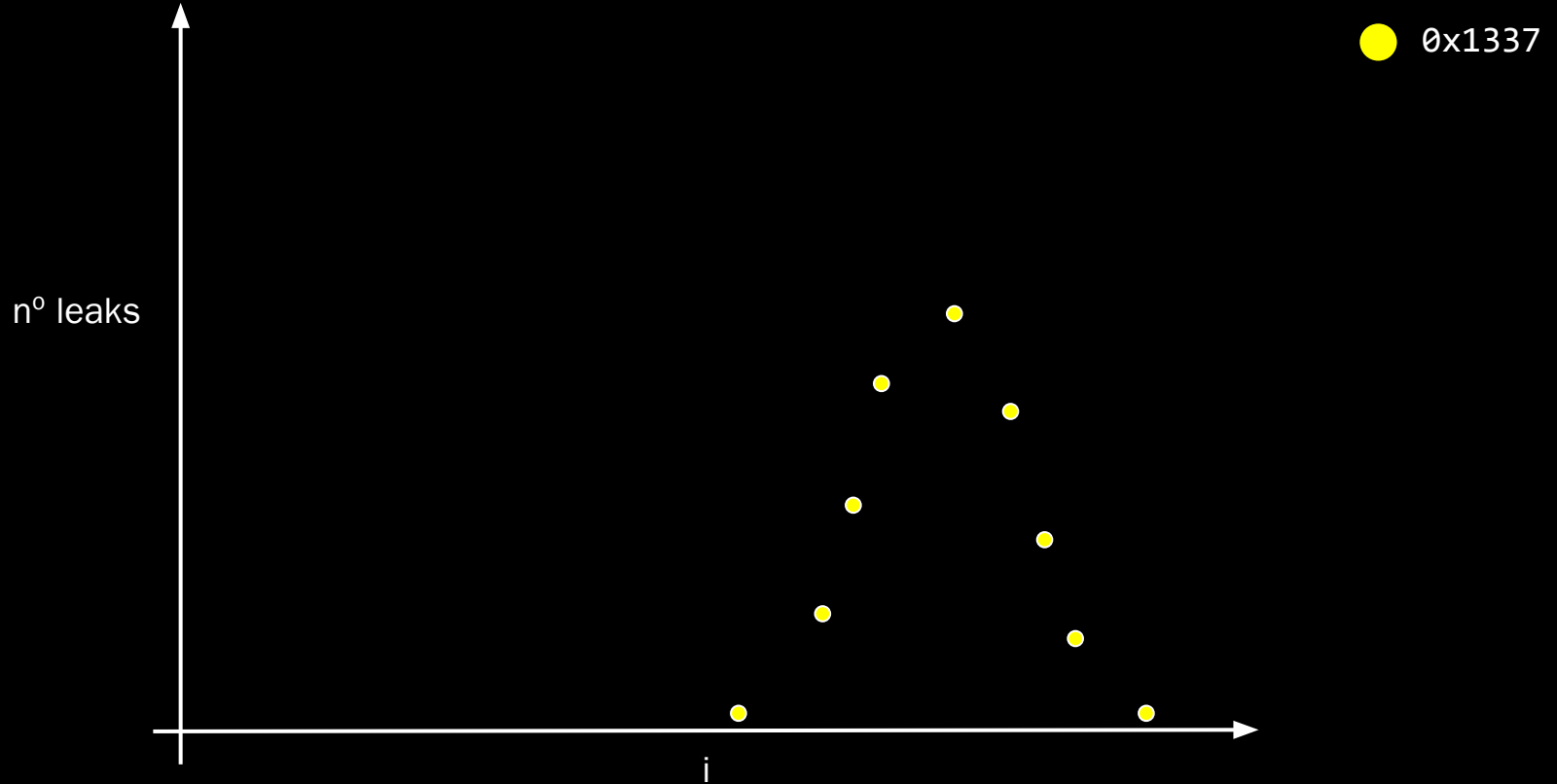


```
leaks: (i: 100, 0x1337)      (i: 30, 0x1234)      (i: 70, 0xcafe)
        (i: 71, 0xcafe)      (i: 104, 0x1337)     (i: 38, 0x1234)
        (i: 28, 0x1234)      (i: 116, 0x1337)    (i: 68, 0xcafe)
        (i: 116, 0x1337)     (i: 71, 0xcafe)     (i: 35, 0x1234)
        (i: 35, 0x1234)     (i: 68, 0xcafe)     (i: 65, 0xcafe)
        (i: 104, 0x1337)    (i: 35, 0x1234)     (i: 98, 0x1337)
        (i: 68, 0xcafe)     (i: 28, 0x1234)     (i: 71, 0xcafe)
        (i: 38, 0x1234)     (i: 38, 0x1234)     (i: 28, 0x1234)
        (i: 98, 0x1337)     (i: 105, 0x1337)    (i: 109, 0x1337)
        (i: 40, 0x1234)     (i: 40, 0x1234)     (i: 116, 0x1337)
        (i: 105, 0x1337)    (i: 121, 0x1337)    (i: 40, 0x1234)
        (i: 70, 0xcafe)     (i: 98, 0x1337)     (i: 100, 0x1337)
        (i: 68, 0xcafe)     (i: 100, 0x1337)    (i: 30, 0x1234)
        (i: 121, 0x1337)    (i: 68, 0xcafe)     (i: 121, 0x1337)
        (i: 30, 0x1234)     (i: 70, 0xcafe)     (i: 104, 0x1337)...
```




Data analysis

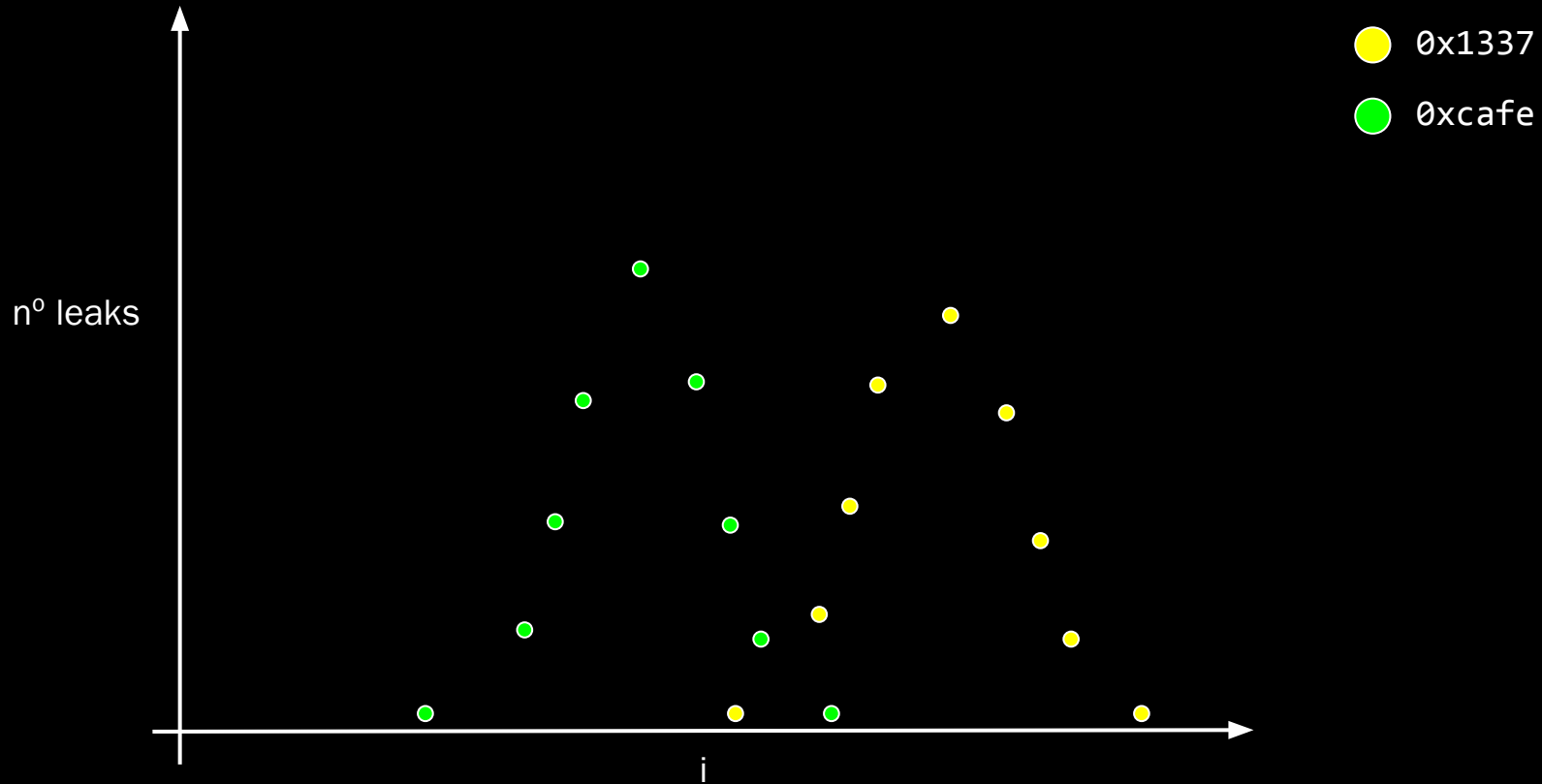
The
Roman
Xploit





Data analysis

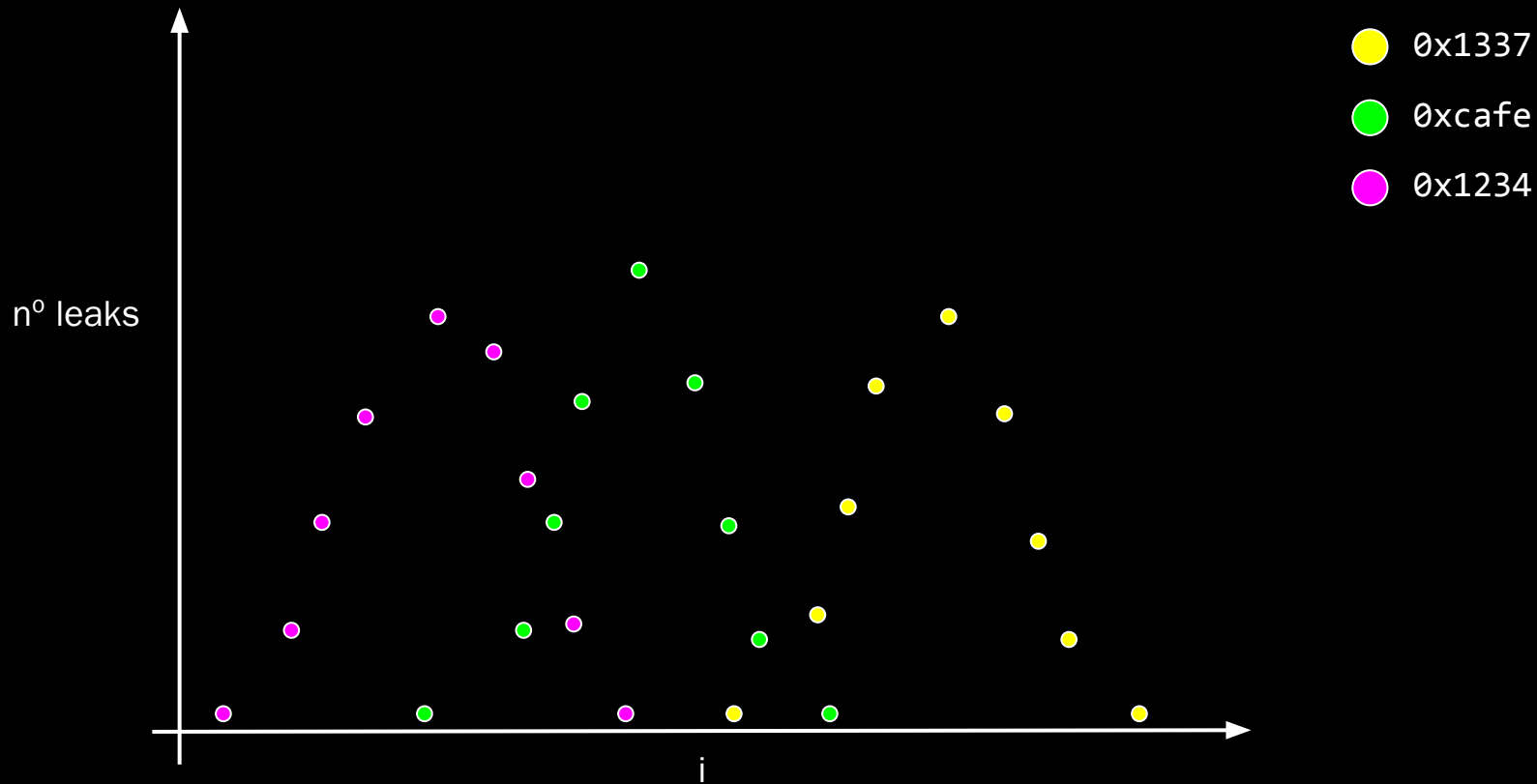
The
Roman
Xploit





Data analysis

The
Roman
Xploit

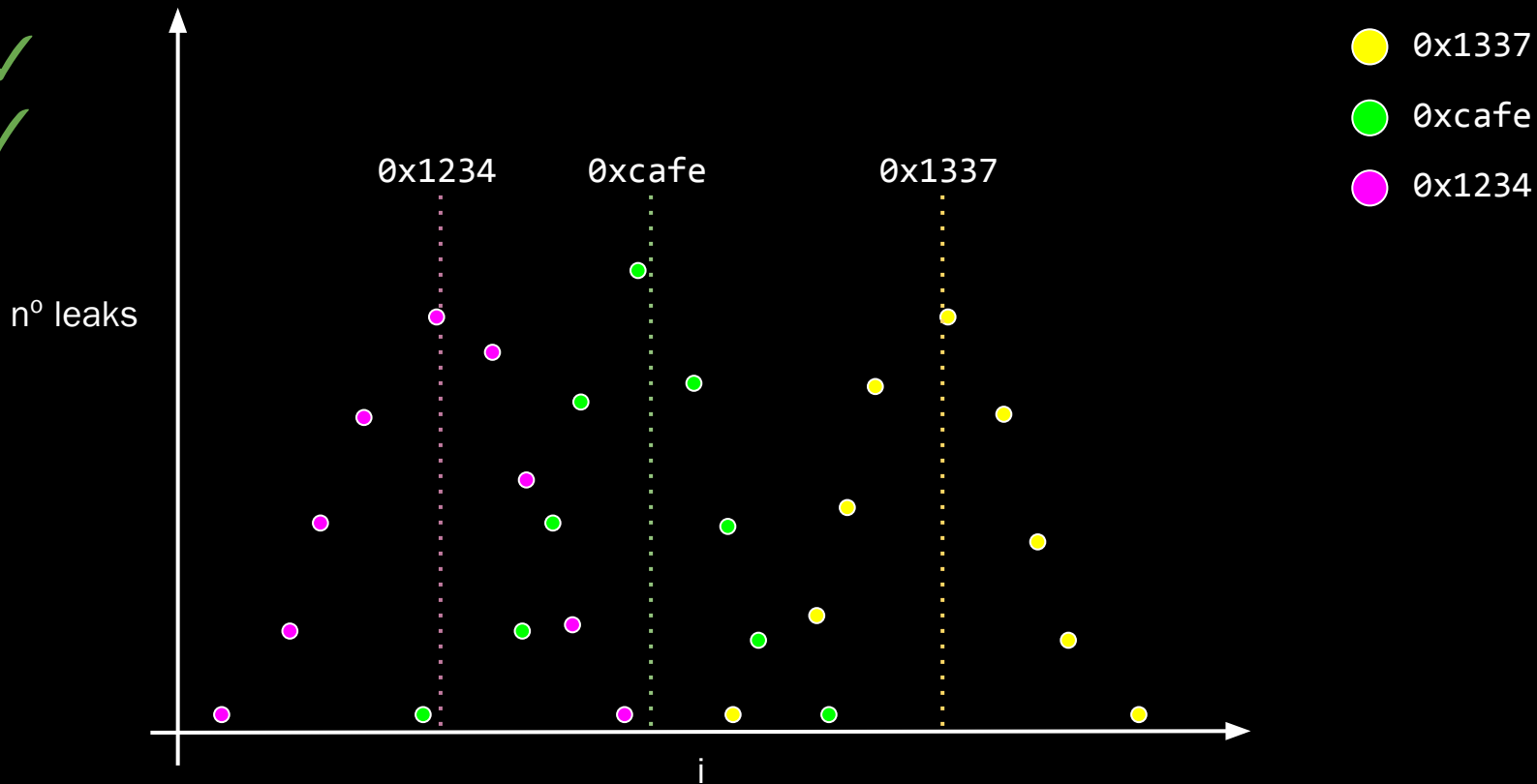




Data analysis

WHAT : ✓

WHEN: ✓





And now what?

We can precisely sequence what do load ports contain:

- during speculation
- during out of order execution
- during microcode assists
- during faulting operations

WHAT : ✓

WHEN: ✓





And now what?

We can precisely sequence what do load ports contain:

- during speculation
- during out of order execution
- during microcode assists
- during faulting operations

WHAT : ✓

WHEN: ✓

Let's see some examples!





Spectre attack

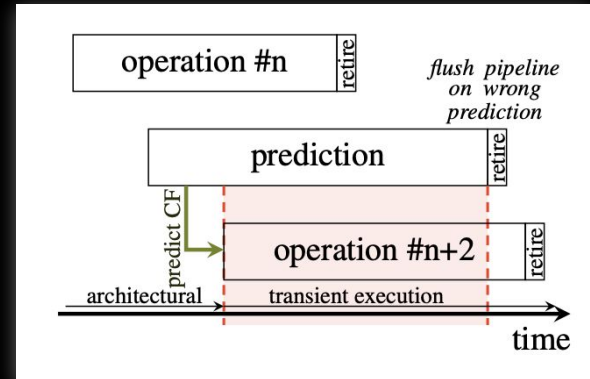


The CPU executes predicted instructions transiently

⇒ There is a small window of instructions that shouldn't be executed, due to misprediction

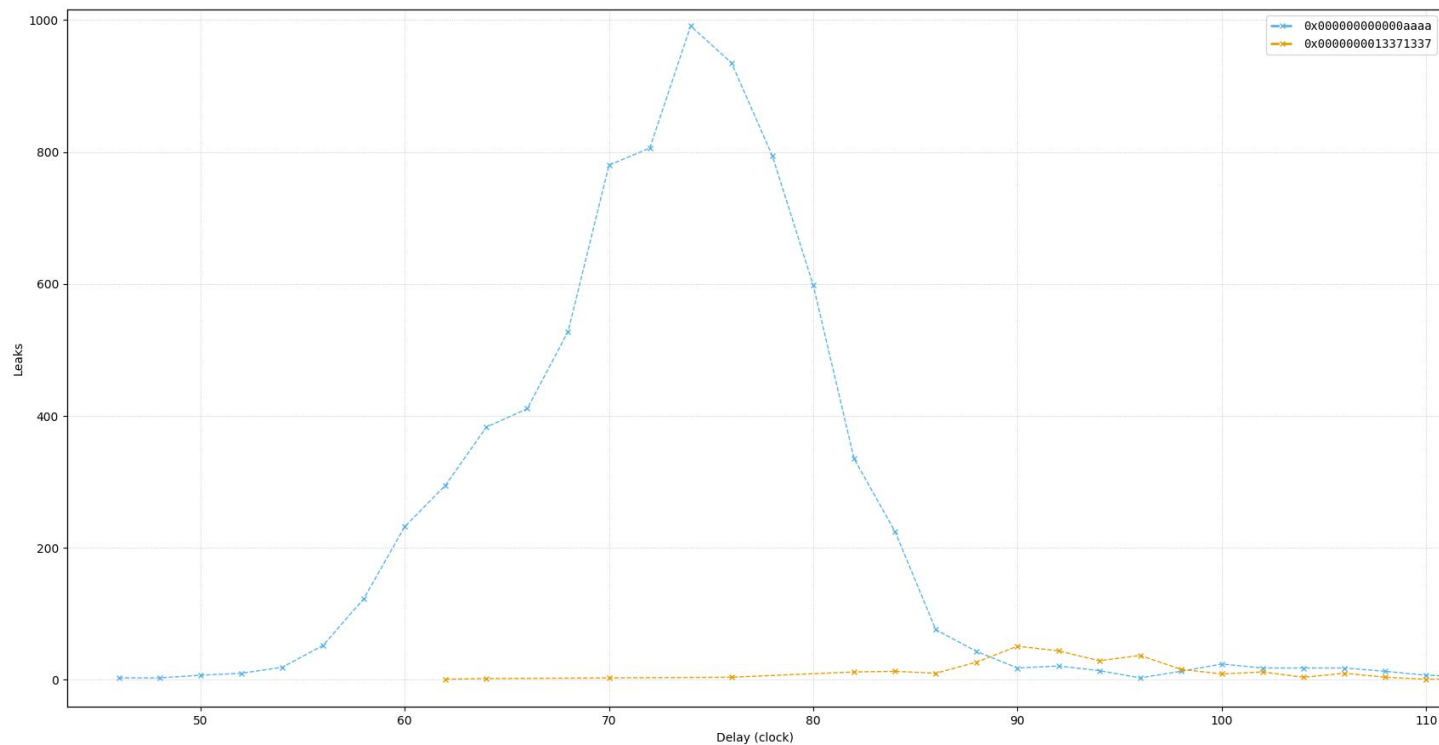
- CPU may execute unexpected instructions on unexpected data

```
for (i = 0; i < len(array); i++) {  
    y += array[i]; // 0xaaaa  
}
```





Spectre attack





CPU Reordering



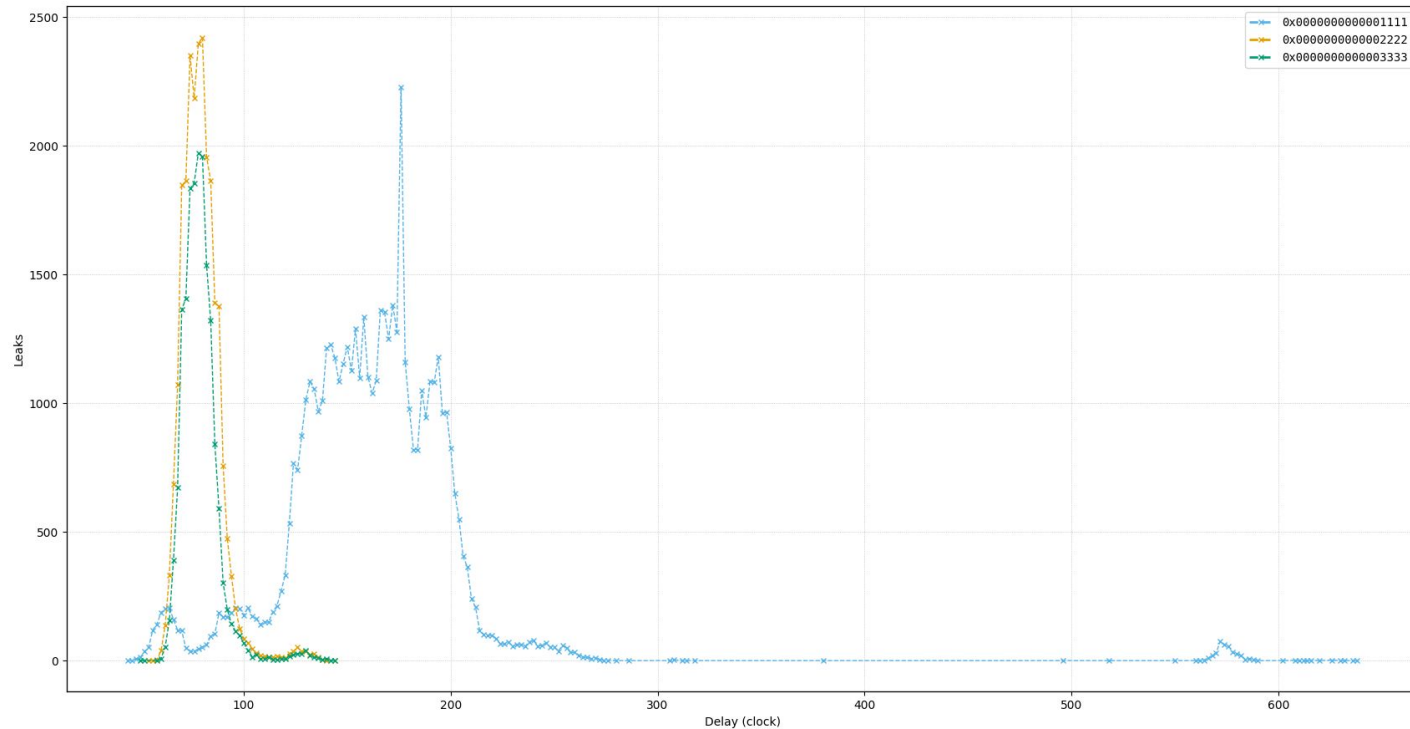
The CPU may reorder instructions that are not ready to execute

```
1. a = uncached_address[0]; // 0x1111
2. b = cached_address[0];   // 0x2222
3. c = cached_address[1];   // 0x3333
```

Line 2 and 3 may execute before line 1



CPU Reordering





Processor Faults

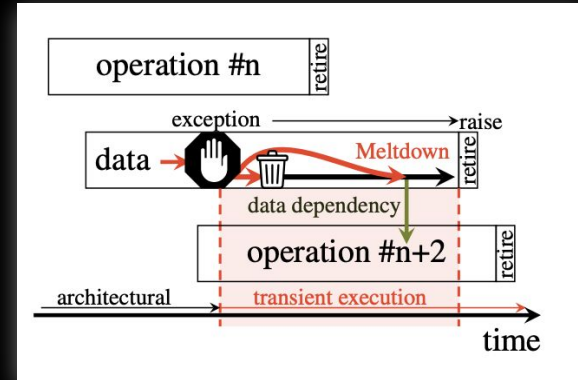


CPU exceptions are enforced lazily:

⇒ After a faulty instruction, there is a small window of instructions that are still executed

```
1. a = *(NULL);  
2. b = array[0];  
3. c = array[1];
```

Even if instruction 1 generates a fault, instructions 2 and 3 are still executed transiently

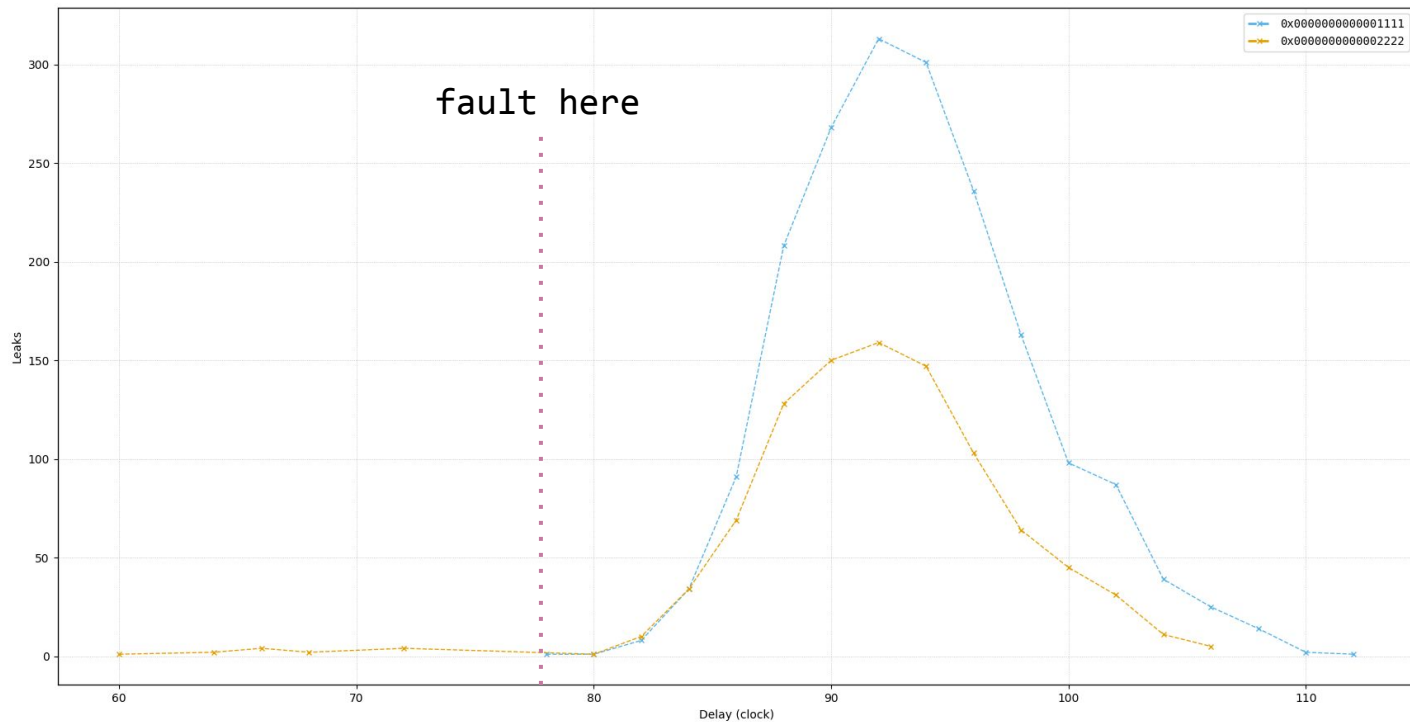




Processor Faults



The
Roman
Xploit



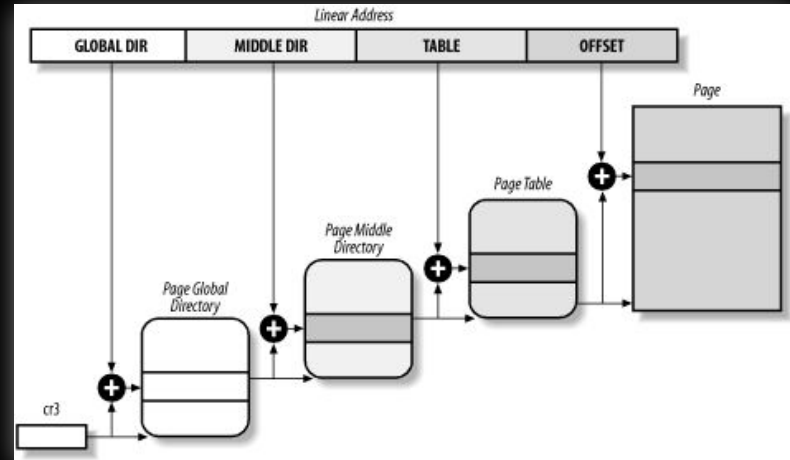


Microcode Assists

There are different cases where the CPU automatically dispatches additional micro operations to be executed to deal with complex situations.

e.g. Page Table Updates:

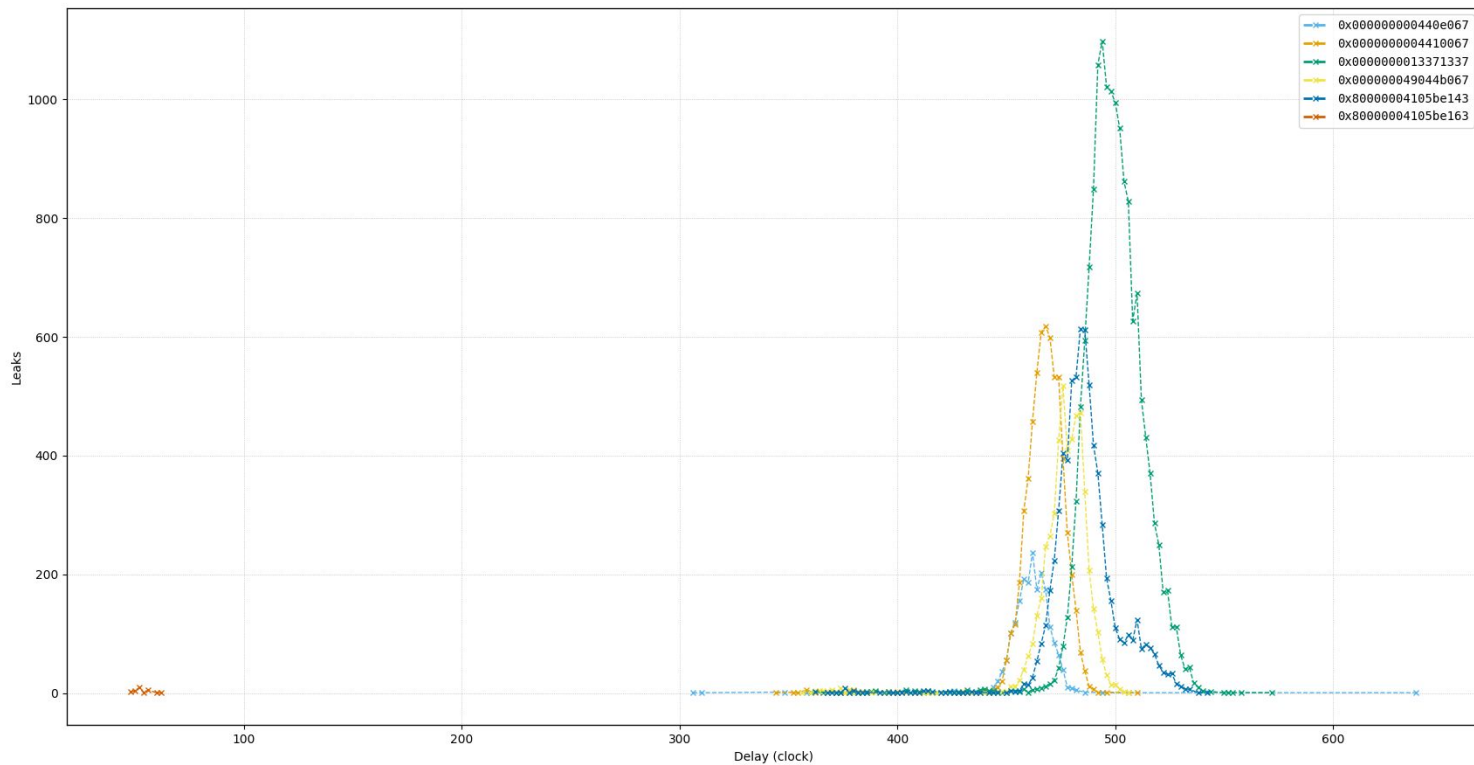
```
a = *never_accessed_page;
```



⇒ the Page Table must be updated to set the *accessed* bit:



Microcode Assists





Thank you!



Questions?