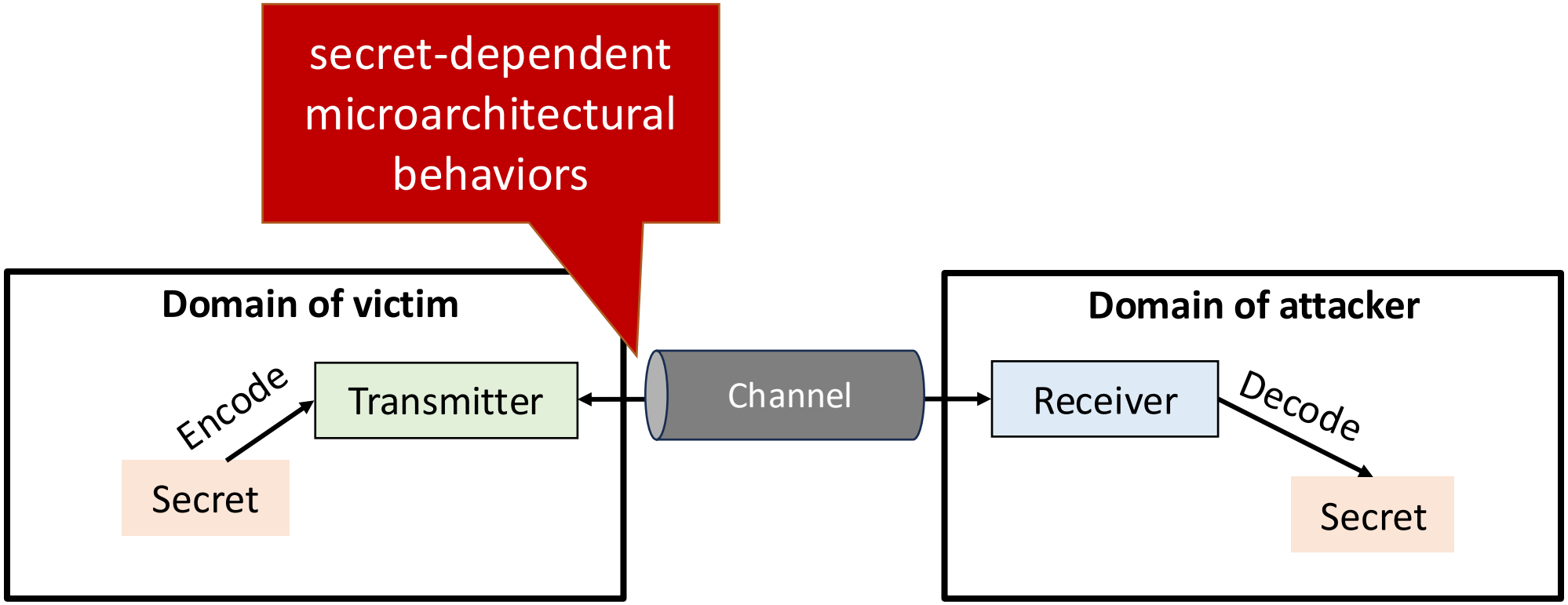
Practical Cache Attacks

Mengjia Yan

Spring 2026

A Communication Model



Kiriansky et al; DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors
Bourgeat et al; CaSA: End-to-end quantitative security analysis of randomly mapped caches

Leak Crypto Library #1: RSA

- Square-and-Multiply Exponentiation

Input :

base b

modulo m

exponent $e = (e_{n-1} \ldots e_0)_2$

Output:

$b^e \bmod m$

When **b = 3, e = 2^{256}**

How many multiplications do you need to do to compute:

$\boldsymbol{b^e}$

Naïve: **2^{256}**

Repeated Squaring: **256**

Leak Crypto Library #1: RSA

- Square-and-Multiply Exponentiation

Input :

base b

modulo m

exponent $e = (e_{n-1} \ldots e_0)_2$

Output:

$b^e \bmod m$

```
r = 1
for i = n-1 to 0 do
        r = sqr(r)
        r  = mod(r, m)
        if e_i == 1 then
                r = mul(r, b)
                r = mod(r, m)
        end
end
```

Leak Crypto Library #2: AES

Input :

Plaintext

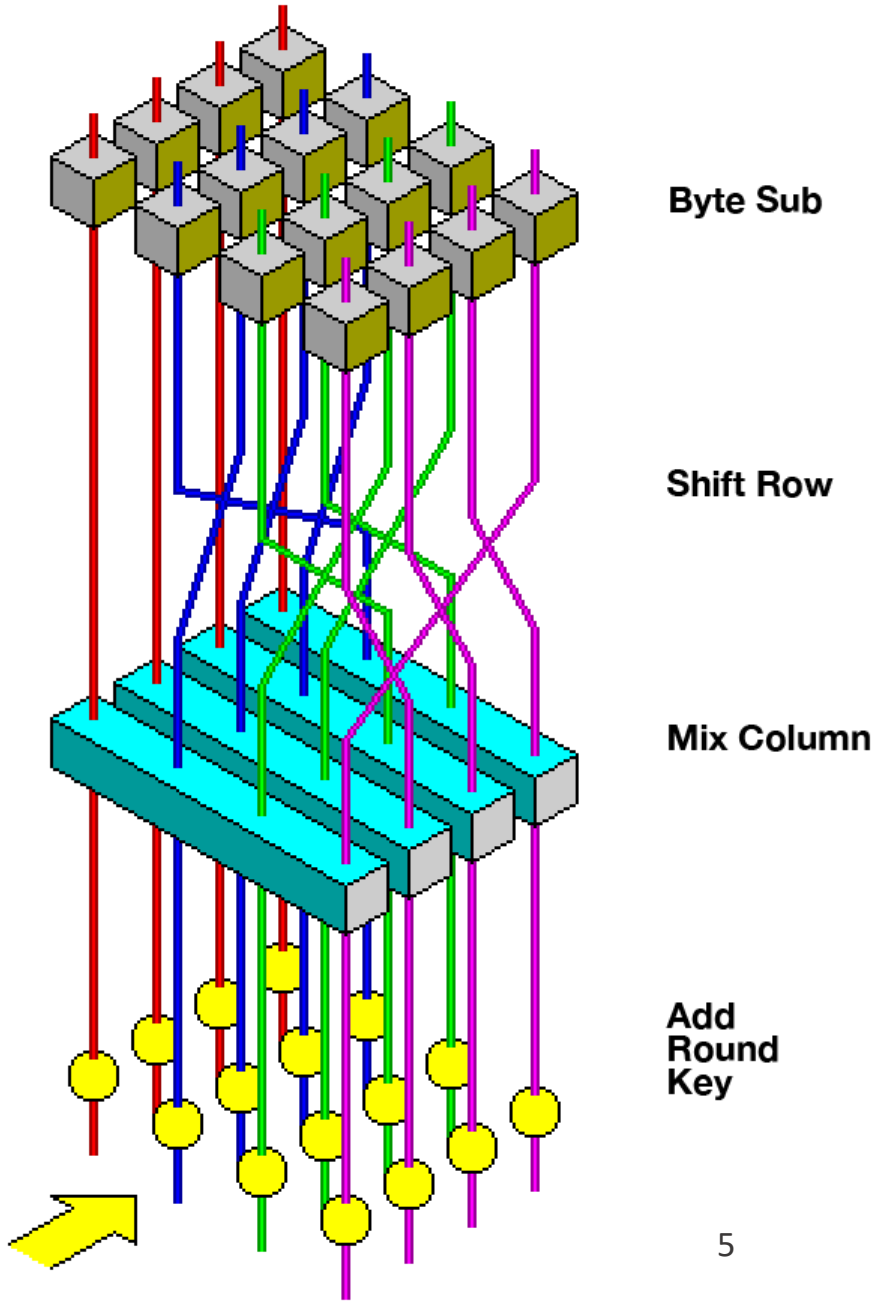
Secret key

Lookup Tables

Output:

Ciphertext

Repeat 10, 12, 14 times depending on key size.



Leak Crypto Library #2: AES T-Table

```
// Td0, Td1, Td2, Td3 are 4 lookup tables

// s0..s3, t0..t3 are 32-bit integer

for (  ;  ;   ) {
    t0 = Td0[(s0>>24)] ^ Td1[(s3>>16)&0xff] ^ Td2[(s2>>8)&0xff] ^ Td3[s1&0xff] ^ rk[4];
    t1 = Td0[(s1>>24)] ^ Td1[(s0>>16)&0xff] ^ Td2[(s3>>8)&0xff] ^ Td3[s2&0xff] ^ rk[5];
    t2 = Td0[(s2>>24)] ^ Td1[(s1>>16)&0xff] ^ Td2[(s0>>8)&0xff] ^ Td3[s3&0xff] ^ rk[6];
    t3 = Td0[(s3>>24)] ^ Td1[(s2>>16)&0xff] ^ Td2[(s1>>8)&0xff] ^ Td3[s0&0xff] ^ rk[7];


    rk += 8;
    if (--r == 0) {
        break;
    }
    ...
```

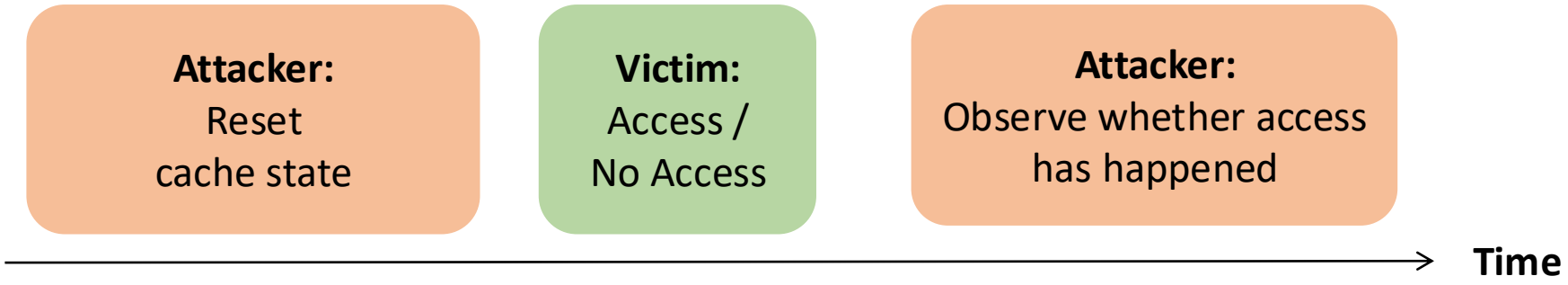
Observation: Secret-dependent memory accesses.

The attacker's goal:
Monitor access patterns at cache line granularity.

Why Cache?

- Large attack surface. Shared across cores/sockets.
- Fast. Can be used to build high-bandwidth channels
- Many states. Can encode secrets spatially to further improve bandwidth and precision.
- There exist many cache-like structures. The same attack concepts and tricks will apply.

Cache Attack Plan



① How to reset cache state?

② How to monitor cache state?

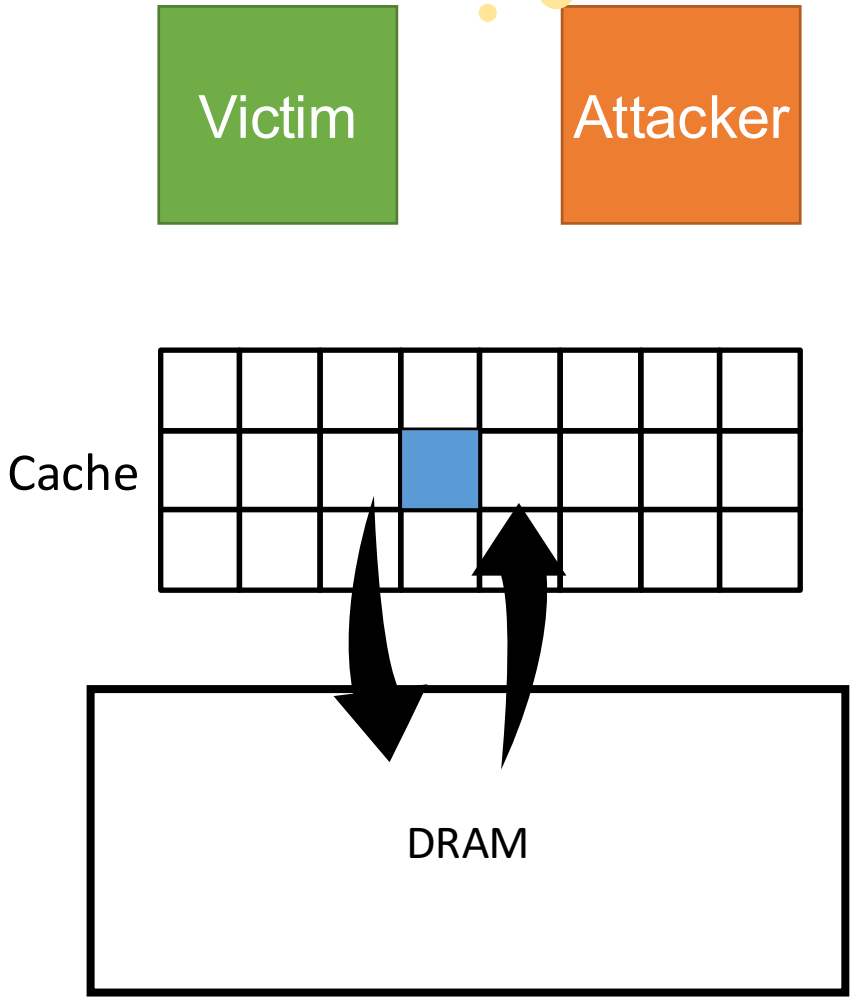
Attack Strategy #1: Flush+Reload

- The flush instructions allow explicit control of cache states
  - In X86, `clflush vaddr`
  - In ARM, `DC CIVAC vaddr`

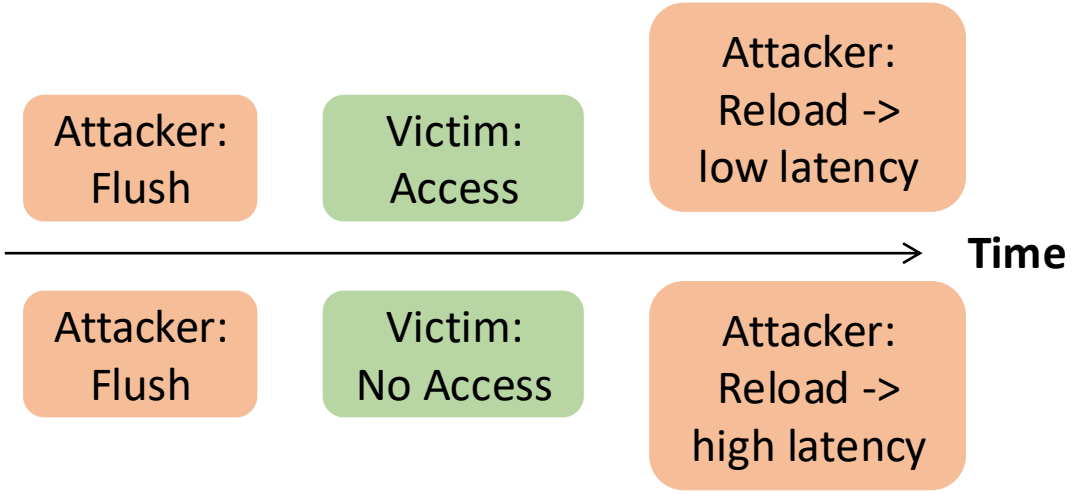
- What are these flush instructions used for except for attacks?
  - For coherence, in the case when the data in the cache is inconsistent with the data in the DRAM.
  - 1) old time, incoherent DMA
  - 2) nowadays, Non-volatile memory for crash recovery

Flush+Reload

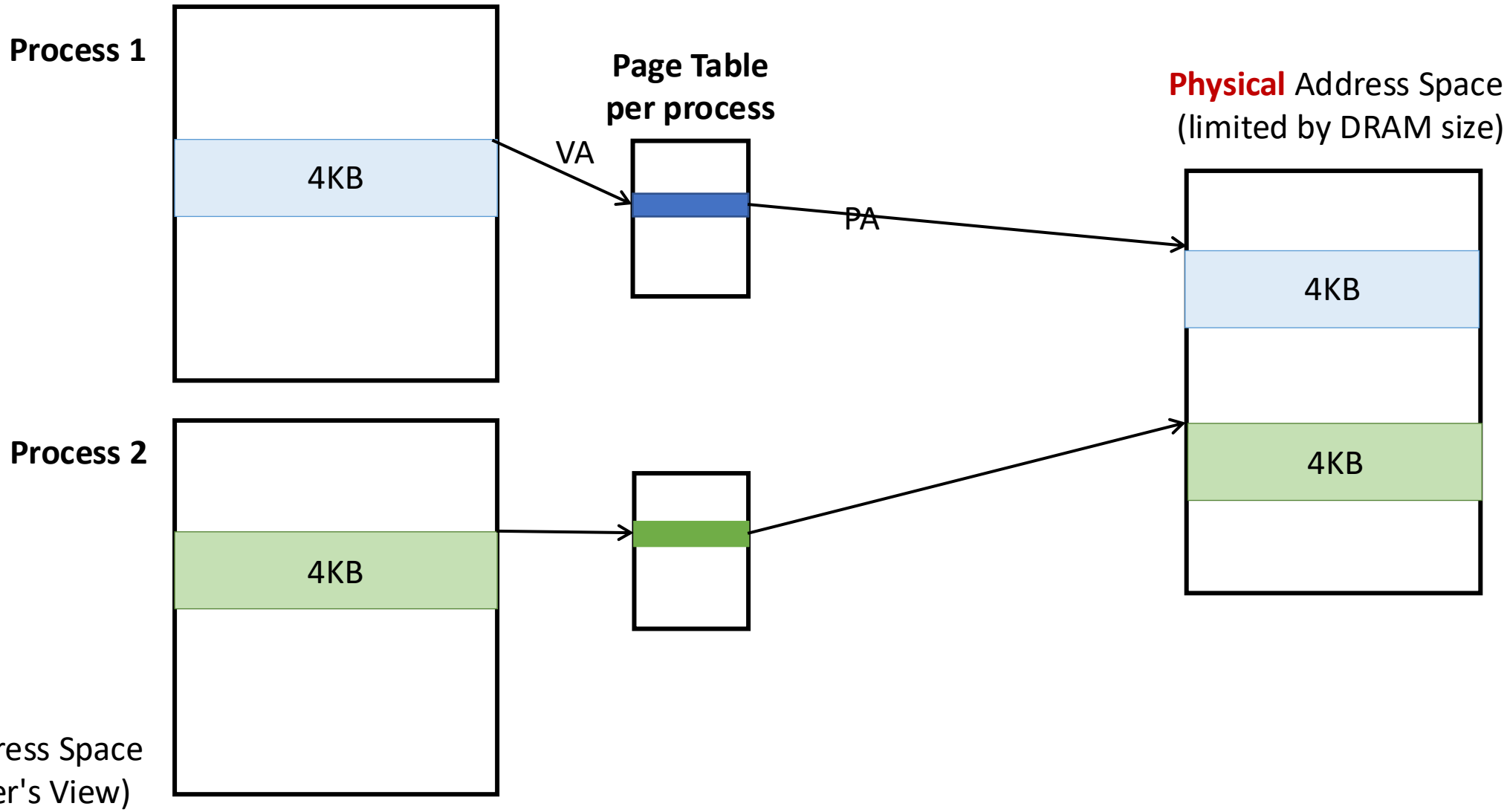
Something might seem weird here… (threat model)



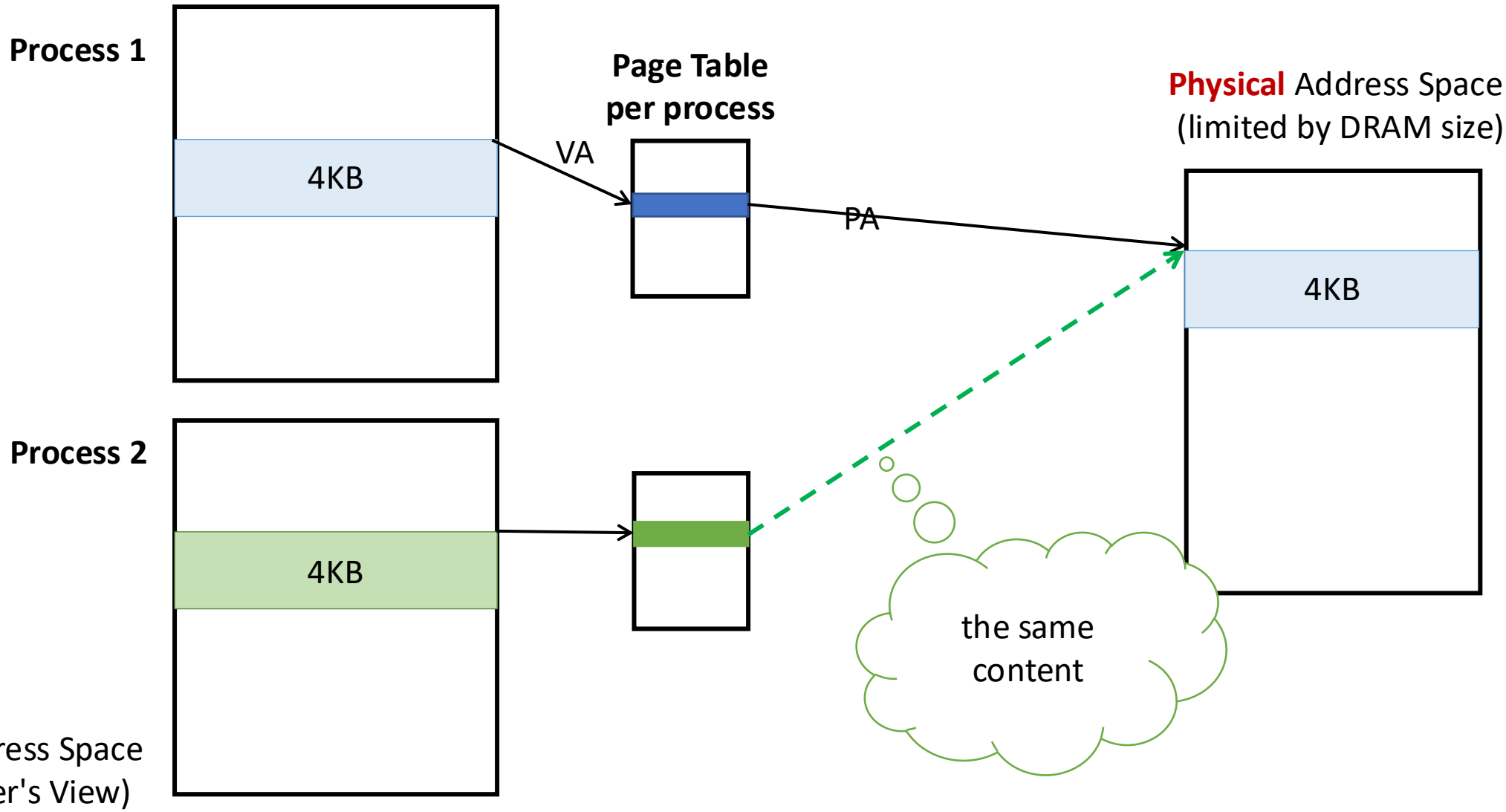
A shared cache line



Page Mapping



Transparent Page Deduplication



The Attack Monitor Code

```
x1 = read cycle;

perform the load;

x2 = read cycle;

latency = x2 – x1;
```

The Attack Monitor Code

```
mfence

rdtsc
mov %eax, %edi


mov (<vaddr>), %rsi

mfence

rdtsc


sub %edi, %eax
```

In x86, 8 GPR:

- `rax, rbx, rcx, rdx`
- `rsp, rbp`
- `rsi, rdi`

"r" means 64-bit

replacing "r" with "e" means the lower 32 bits.

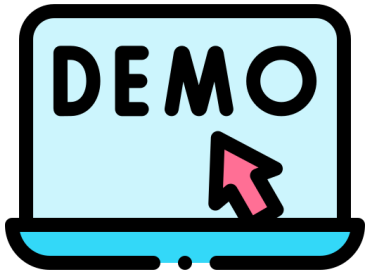
`rdtsc`:

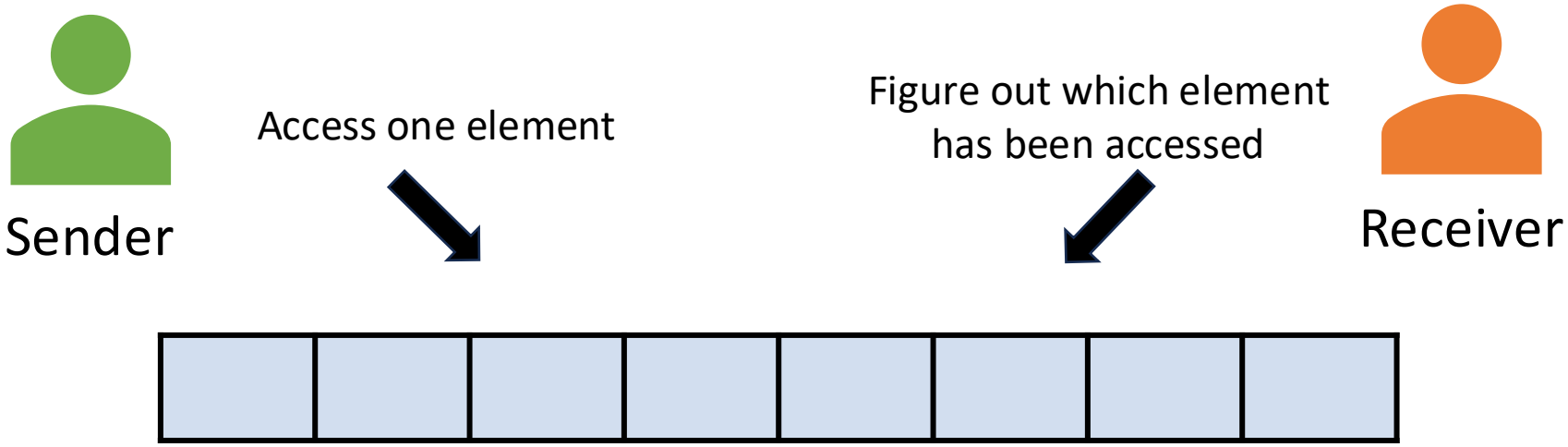
- Read Time-Stamp Counter
- `edx:eax` := TimeStampCounter;

`mfence`:

- Memory Fence
- Performs a serializing operation on all memory instructions

A Demo


DEMO


Sender
Access one element
Figure out which element has been accessed
Receiver

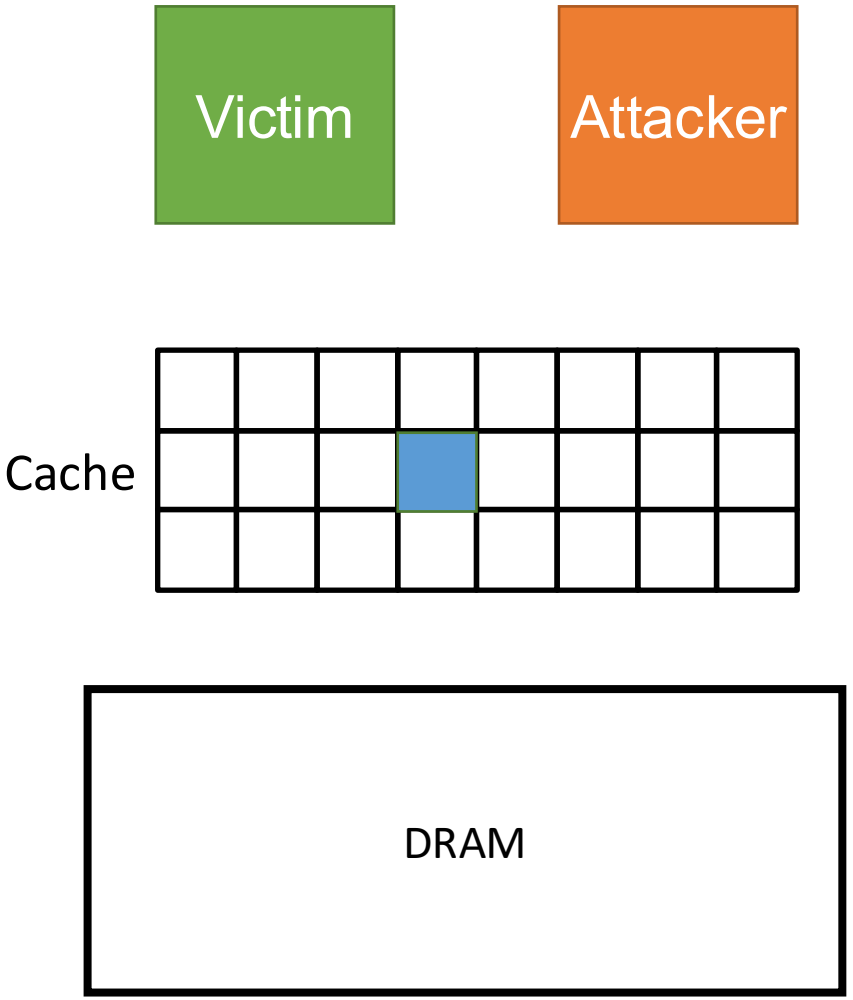
Attack Strategy #2: ?

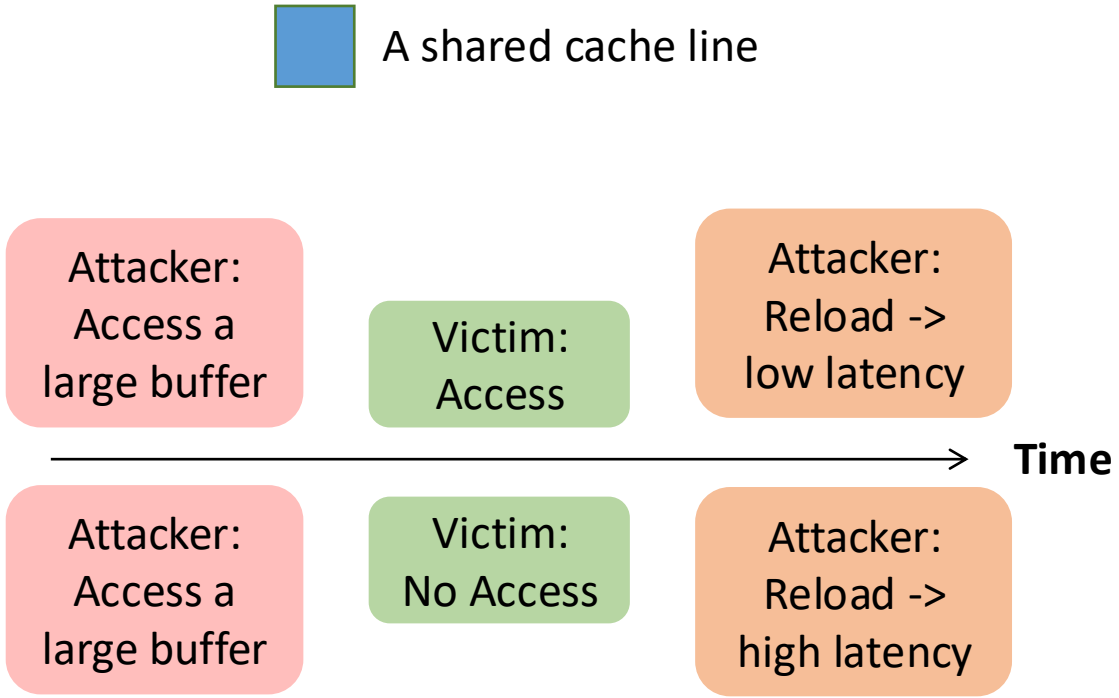
- Cache state manipulation instructions
  - In X86, `clflush vaddr`
  - In ARM, `DC CIVAC vaddr`

- What if these instructions are not available in user space?
  - Apple devices
  - *"Except ARMv8-A CPUs, ARM processors do not support a flush instruction"*

from ARMageddon: Cache Attacks on Mobile Devices (USENIX'16)

Attack Strategy #2: Evict+Reload





Lessons Learnt So F

So the fundamental problem: **shared memory** between different security domains.

Source: https://kb.vmware.com/s/article/2080735

Security considerations and disallowing inter-Virtual Machine Transparent Page Sharing (2080735)

Last Updated: 8/25/2021 **Categories:** Informational **Total Views:** 66593

5 **Language:** 英文 SUBSCRIBE

Details

This article acknowledges the recent academic research that leverages Transparent Page Sharing (TPS) to gain unauthorized access to data under certain highly controlled conditions and documents VMware's precautionary measure of restricting TPS to individual virtual machines by default in upcoming ESXi releases. At this time, VMware believes that the published information disclosure due to TPS between virtual machines is impractical in a real world deployment.

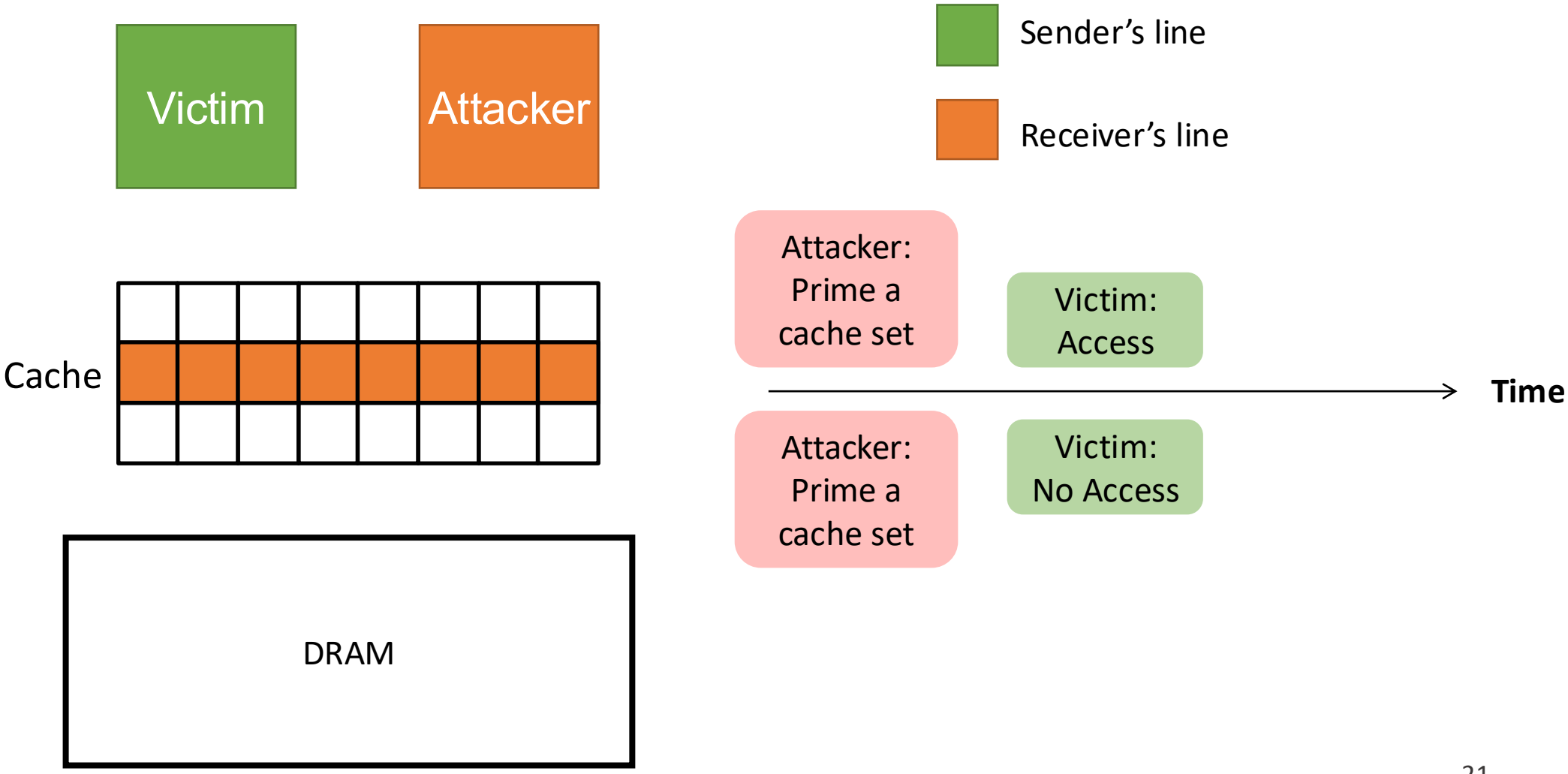
Published academic papers have demonstrated that by forcing a flush and reload of cache memory, it is possible to measure memory timings to try and determine an AES encryption key in use on another virtual machine running on the same physical processor of the host server if Transparent Page Sharing is enabled between the two virtual machines. This technique works only in a highly controlled system configured in a non-standard way that VMware believes would not be recreated in a production environment. .

Even though VMware believes information being disclosed in real world conditions is unrealistic, out of an abundance of caution upcoming ESXi Update releases will no longer enable TPS between Virtual Machines by default (TPS will still be utilized within individual VMs).

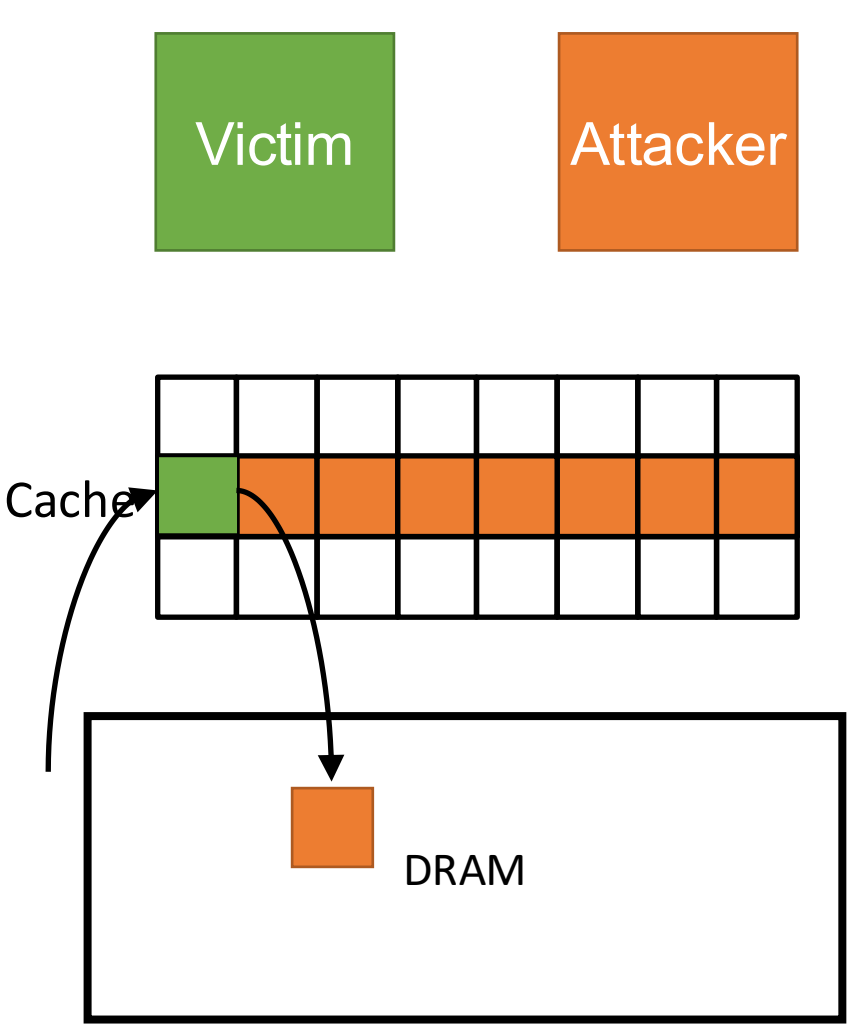
No more shared memory.

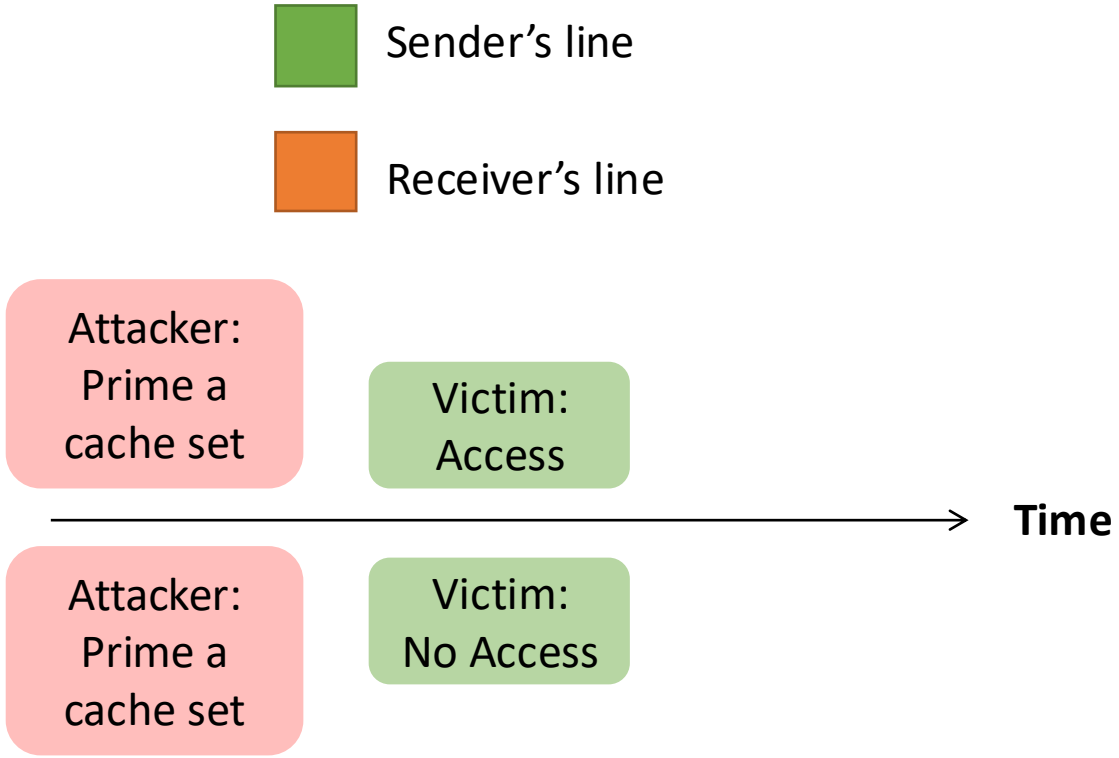
Can we still attack?

Attack Strategy #3: Prime+Probe

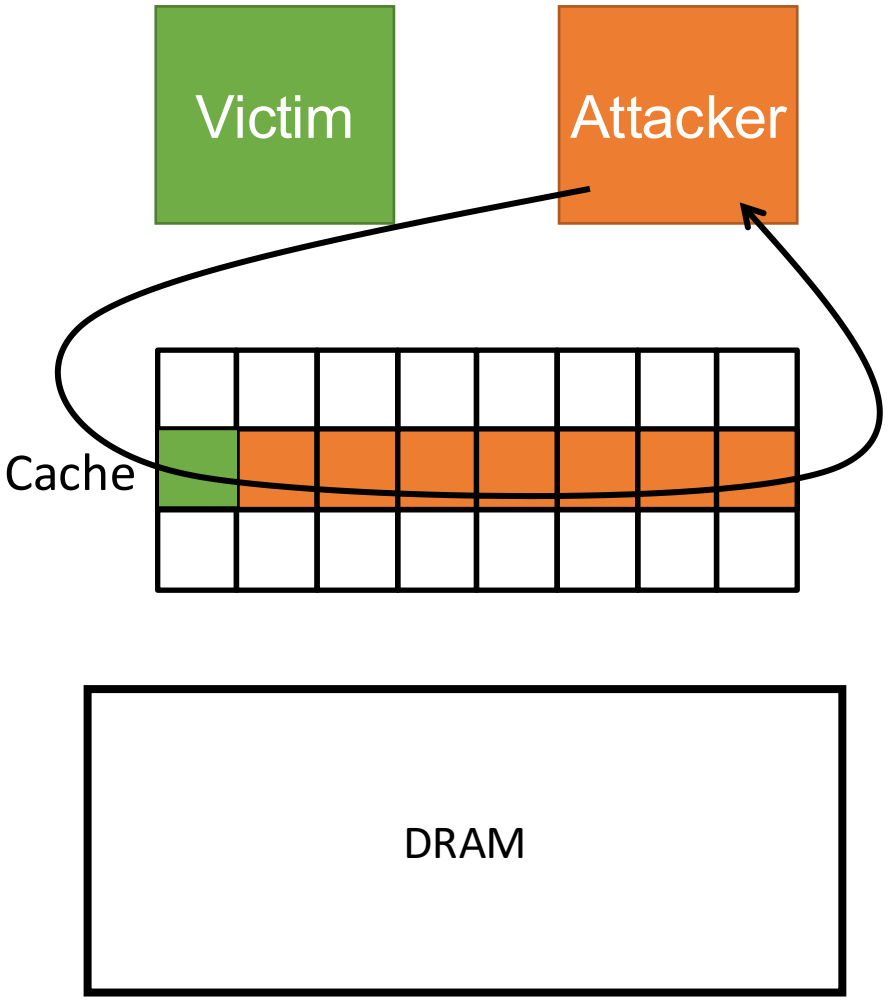


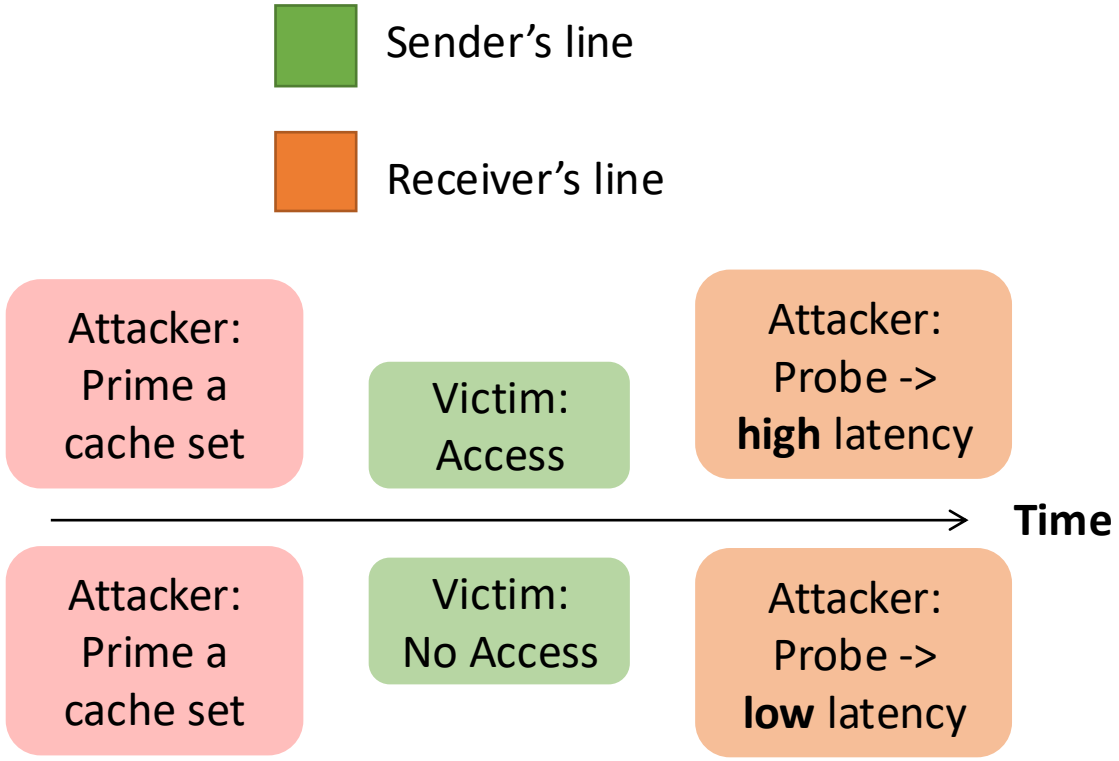
Attack Strategy #3: Prime+Probe





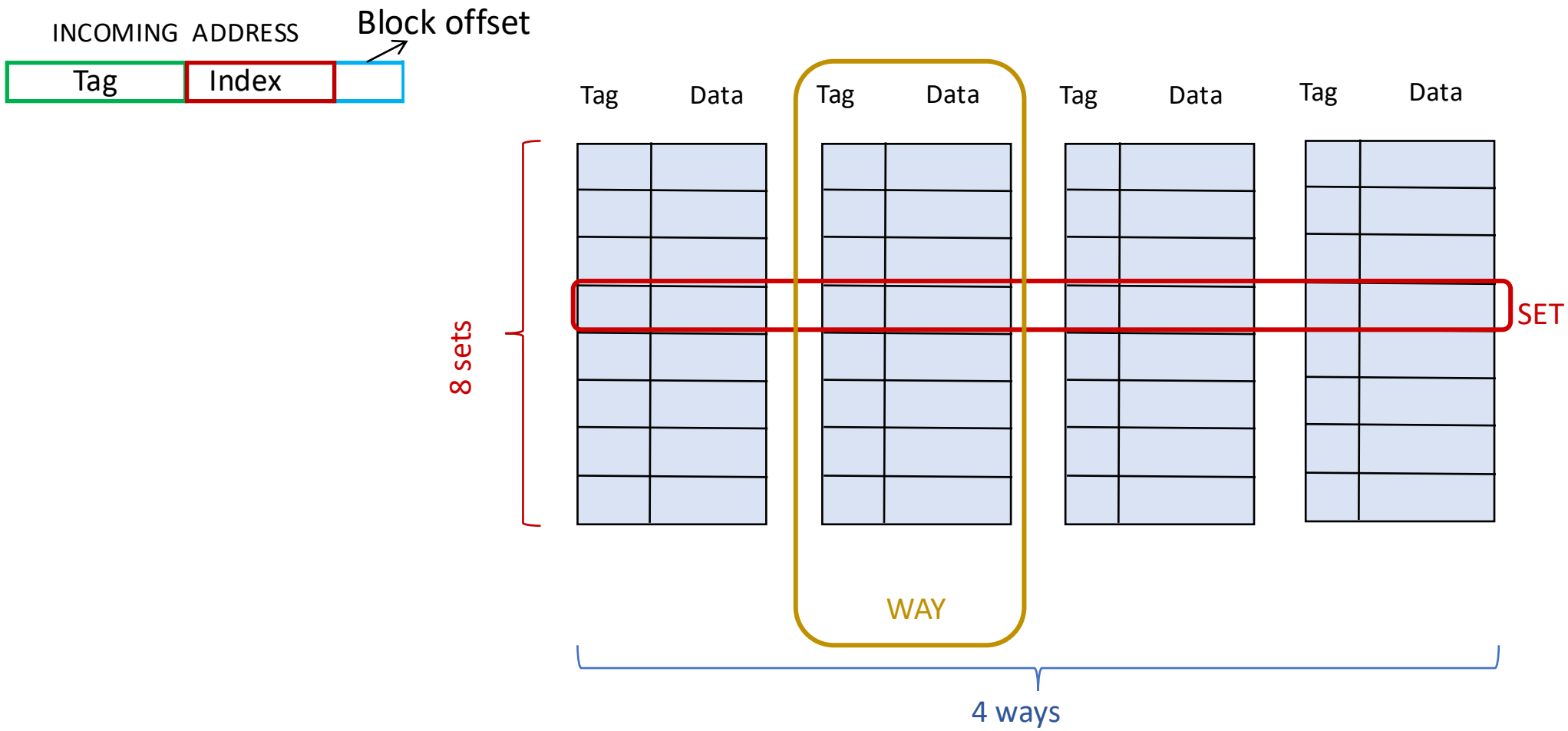
Attack Strategy #3: Prime+Probe





Question: How to know which cache set an address maps to?

N-way Set-Associative Cache



A 6.191/6.004 Quiz Question

- I have a virtual address: 0xAAAA

- The cache parameter is as below
  - Cache size: 64KB
  - Line size/Block size: 64B
  - Associativity: 8

Question 1:
What is the cache set index?

Question 2:
What is the next address that map to **the same cache set** as this one but not the same cache line?

A 6.191/6.004 Quiz Question

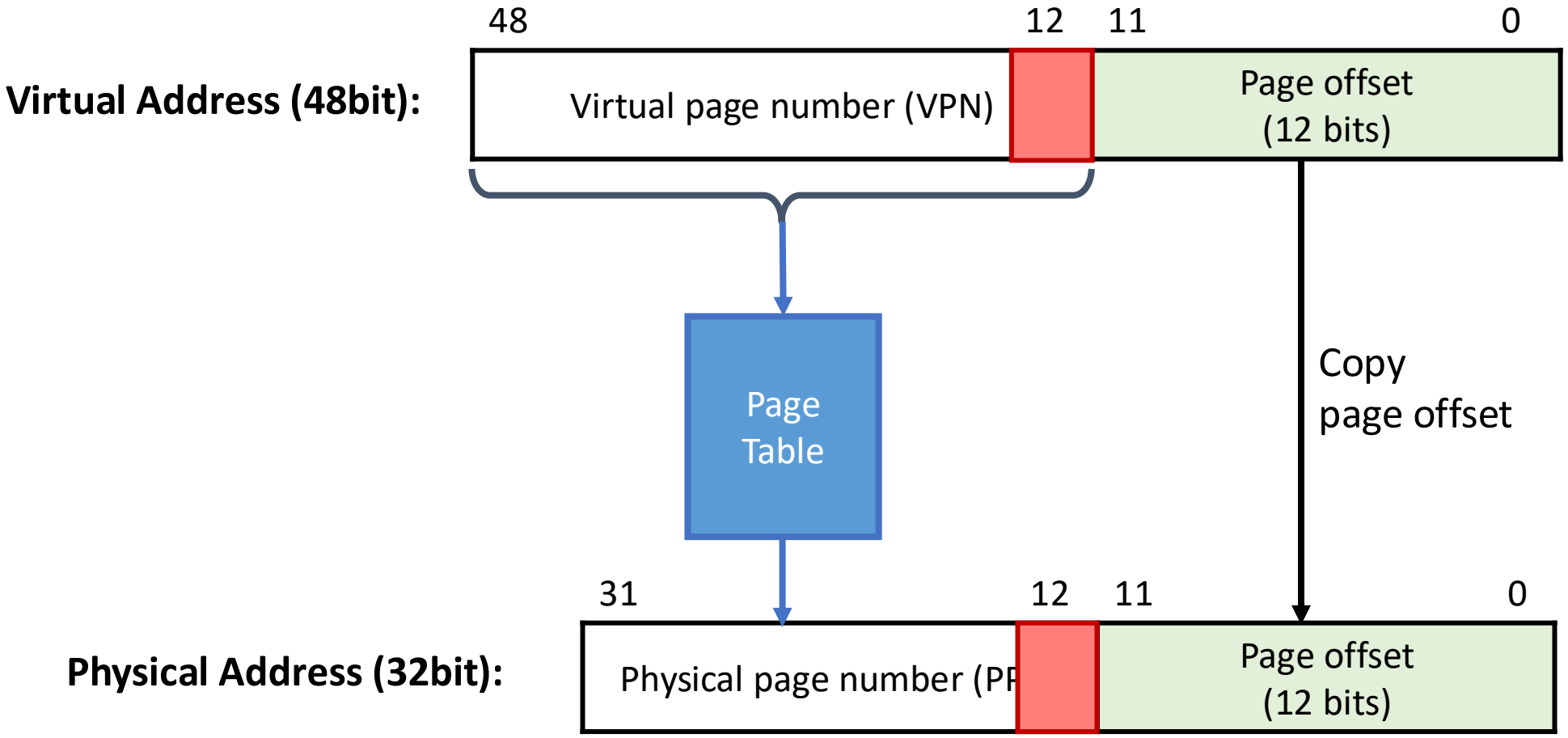
- I have a **virtual address**: 0xAAAA

- The cache parameter is as below
  - Cache size: 64KB
  - Line size/Block size: 64B
  - Associativity: 8

Question 1:
What is the cache set index?

Question 2:
What is the next address that map to **the same cache set** as this one but not the same cache line?

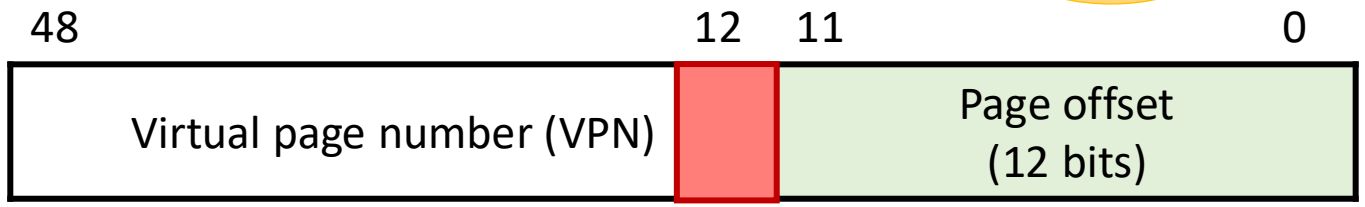
Address Translation (4KB page)



Using Huge Pages

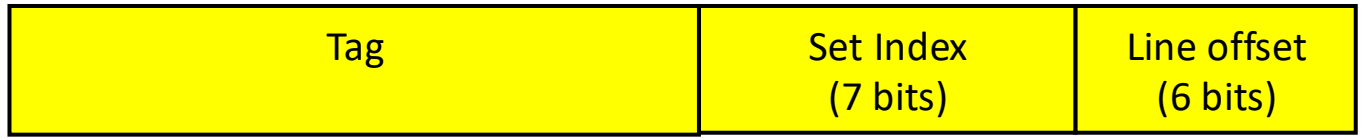
Why system designers introduce huge pages?

Virtual Address (48bit):

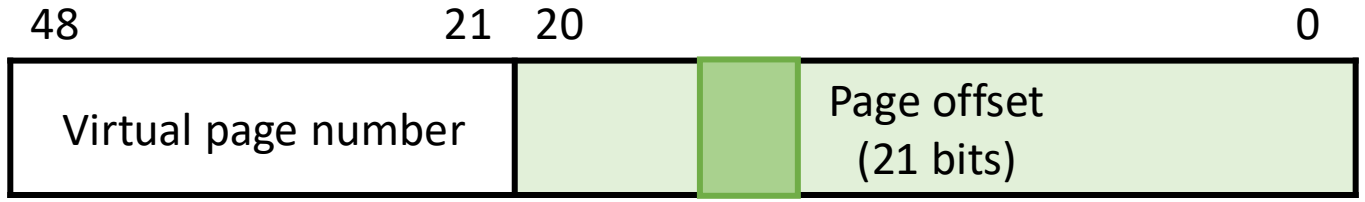


The red bit is NOT under attacker's control

Cache mapping:
(256 sets)



Virtual Address :
2MB page



This green bit IS under attacker's control

There are still many other practical challenges.

If you tackle all of them, you can ...

Review RSA Vulnerability

- Square-and-Multiply Exponentiation

Input :

base b

modulo m

exponent $e = (e_{n-1} \ldots e_0)_2$

Output:

$b^e \bmod m$

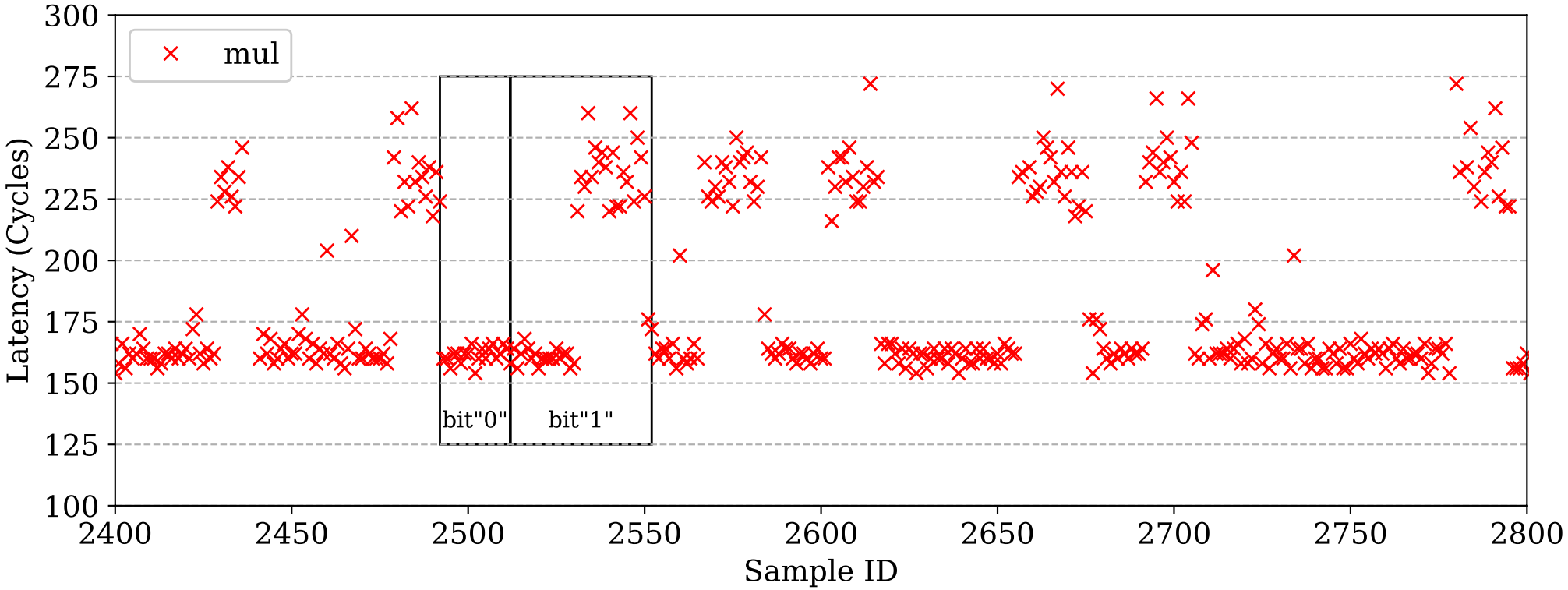
```
r = 1
for i = n-1 to 0 do
        r = sqr(r)
        r  = mod(r, m)
        if e_i == 1 then
                r = mul(r, b)
                r = mod(r, m)
        end
end
```

The Multiply Function

```
471 mpi_limb_t
472 mpihelp_mul( mpi_ptr_t prodp, mpi_ptr_t up, mpi_size_t usize,
473                   mpi_ptr_t vp, mpi_size_t vsize)
474 {
475     mpi_ptr_t prod_endp = prodp + usize + vsize - 1;
476     mpi_limb_t cy;
477     struct karatsuba_ctx ctx;
478
479     if( vsize < KARATSUBA_THRESHOLD ) {
480     mpi_size_t i;
481     mpi_limb_t v_limb;
482
483     if( !vsize )
484         return 0;
485
486     /* Multiply by the first limb in V separately, as the result can be
487      * stored (not added) to PROD.  We also avoid a loop for zeroing.  */
488     v_limb = vp[0];
489     if( v_limb <= 1 ) {
490         if( v_limb == 1 )
491         MPN_COPY( prodp, up, usize );
492         else
493         MPN_ZERO( prodp, usize );
494         cy = 0;
495     }
496     else
497         cy = mpihelp_mul_1( prodp, up, usize, v_limb );
498
499     prodp[usize] = cy;
500     prodp++;
```

```
501
502     /* For each iteration in the outer loop, multiply one limb from
503      * U with one limb from V, and add it to PROD.  */
504     for( i = 1; i < vsize; i++ ) {
505         v_limb = vp[i];
506         if( v_limb <= 1 ) {
507         cy = 0;
508         if( v_limb == 1 )
509            cy = mpihelp_add_n(prodp, prodp, up, usize);
510         }
511         else
512         cy = mpihelp_addmul_1(prodp, up, usize, v_limb);
513
514         prodp[usize] = cy;
515         prodp++;
516     }
517
518     return cy;
519     }
520
521     memset( &ctx, 0, sizeof ctx );
522     mpihelp_mul_karatsuba_case( prodp, up, usize, vp, vsize, &ctx );
523     mpihelp_release_karatsuba_ctx( &ctx );
524     return *prod_endp;
525 }
```

Raw Trace



Access latencies measured in the probe operation in Prime+Probe on a cache line inside the multiplication function.
A sequence of "01010111011001" can be deduced as part of the exponent.

Takeaways

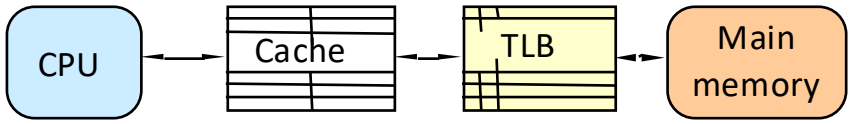
- **Practical** challenges in implementing a reliable cache attack
  - Page sharing
  - Noise due to prefetchers
  - Uncertainty due to page mapping
  - Replacement policy
  - etc.

- **Hardware and software optimizations** make attacks easier
  - Transparent page sharing
  - Copy-on-write
  - Huge pages
  - Virtually-indexed and physically-tagged caches

Next:
Cache Attack Recitation

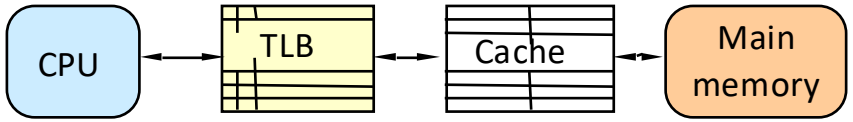
Using Caches with Virtual Memory

Virtually-Addressed Cache



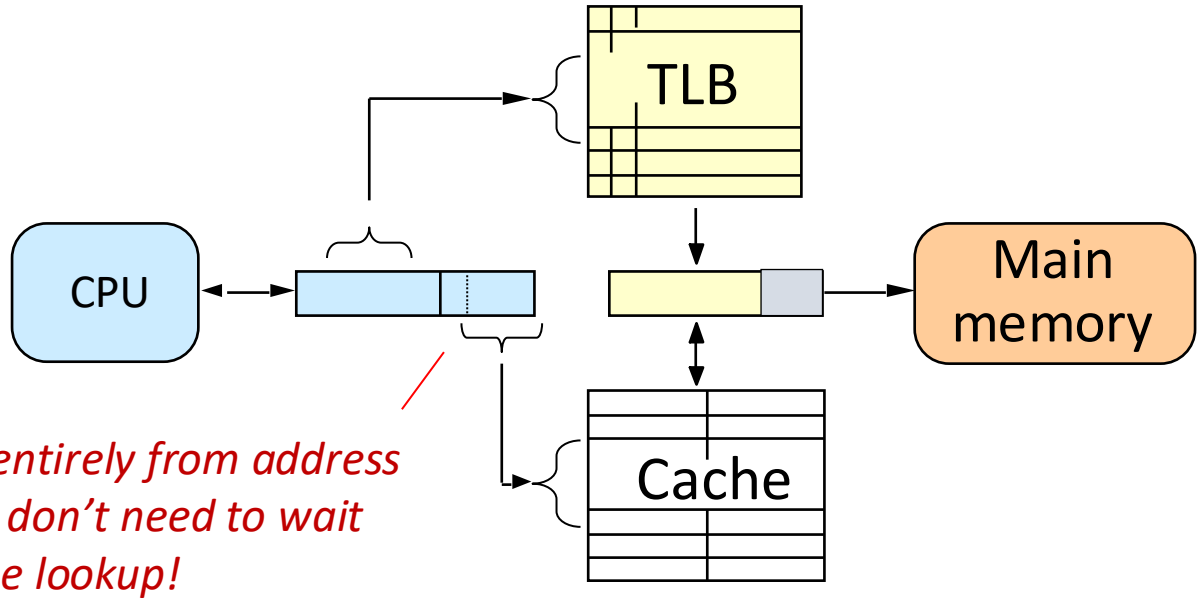
- FAST: No virtual→physical translation on cache hits
- Problem: Must flush cache after context switch

Physically-Addressed Cache



- Avoids stale cache data after context switch
- SLOW: virtual→physical translation before every cache access

Best of Both Worlds (L1 Cache): Virtually-Indexed, Physically-Tagged Cache (VIPT)



Cache index comes entirely from address bits in page offset – don't need to wait for TLB to start cache lookup!