black hat ASIA 2018

When Good Turns Evil

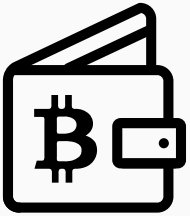
Using Intel SGX to Stealthily Steal Bitcoins

Michael Schwarz
@misc0110

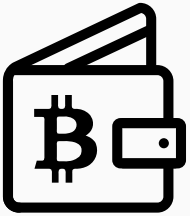
Moritz Lipp
@mlqxyz

Outline

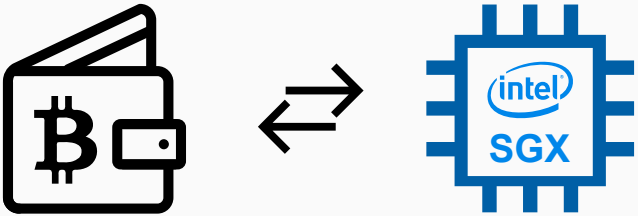
Outline



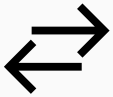
Outline

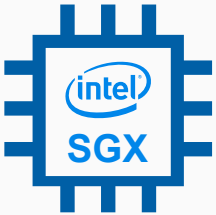


Outline

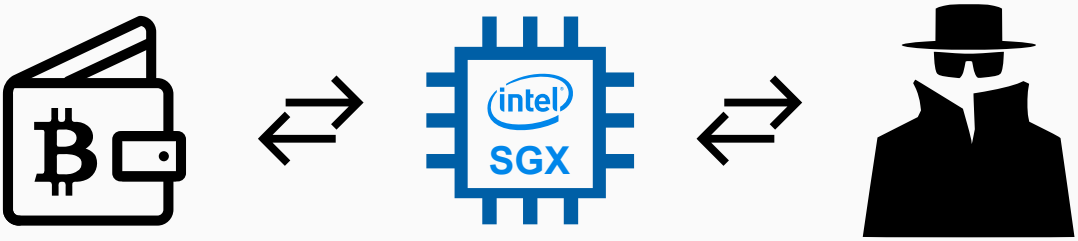


Outline





Outline



- **Michael Schwarz**
- PhD Student, Graz University of Technology
- @misc0110
- michael.schwarz@iaik.tugraz.at

- **Moritz Lipp**
- PhD Student, Graz University of Technology
- @mlqxyz
- moritz.lipp@iaik.tugraz.at

And the team

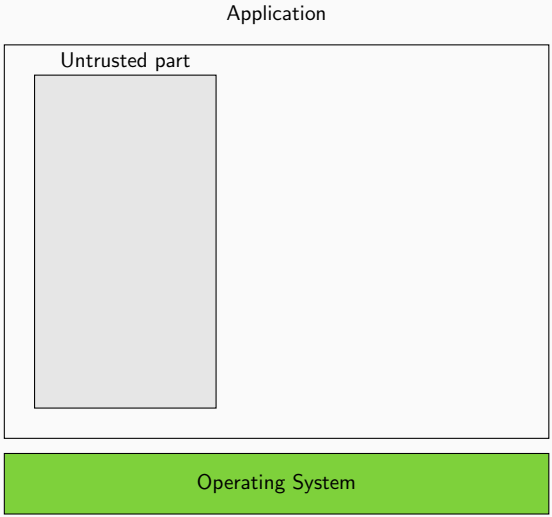
The research team

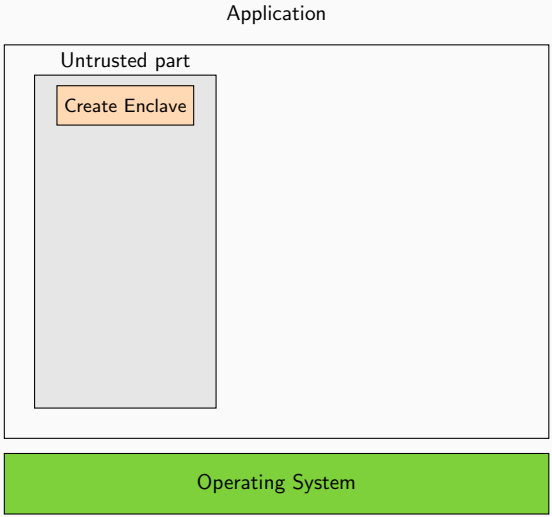
- Daniel Gruss
- Clémentine Maurice
- Samuel Weiser
- Thomas Schuster
- Stefan Mangard

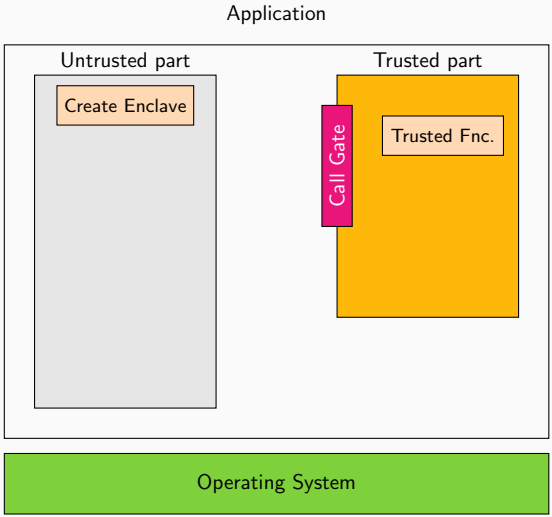
from Graz University of Technology

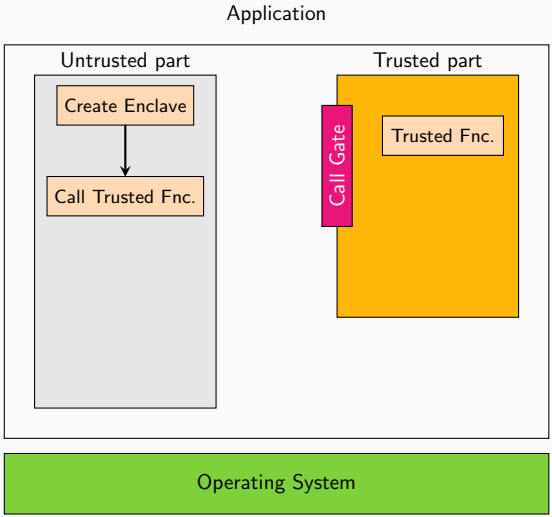
- Anders Fogh

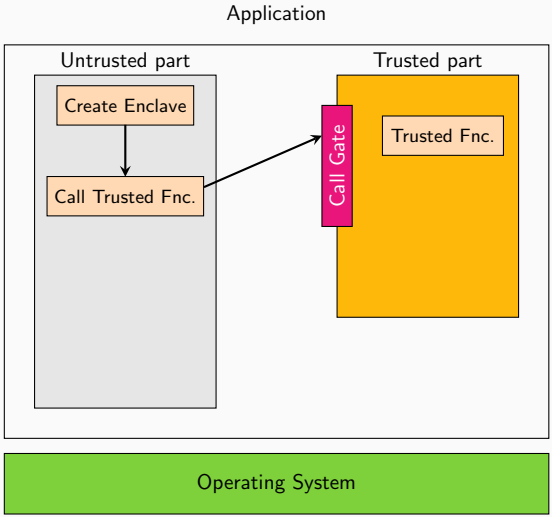
from G DATA Advanced Analytics

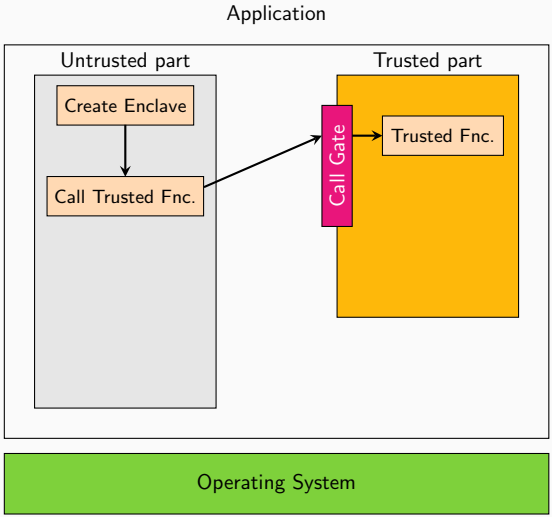

Application
Untrusted part
Operating System

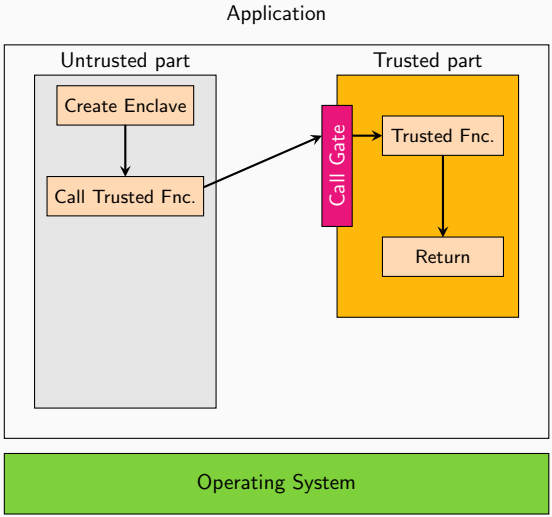

Application
Untrusted part
Create Enclave
Operating System

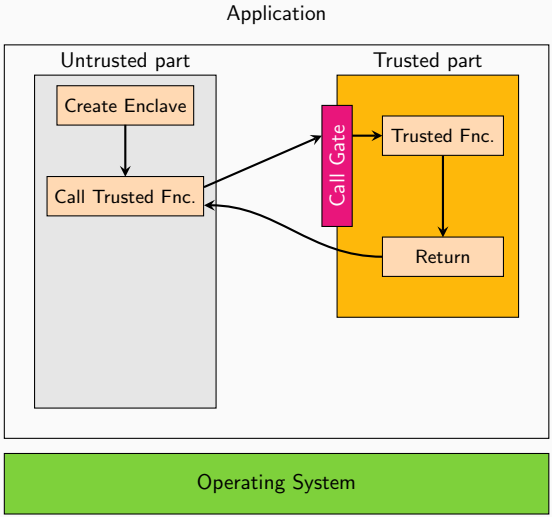

Application
Untrusted part
Create Enclave
Trusted part
Call Gate
Trusted Fnc.
Operating System

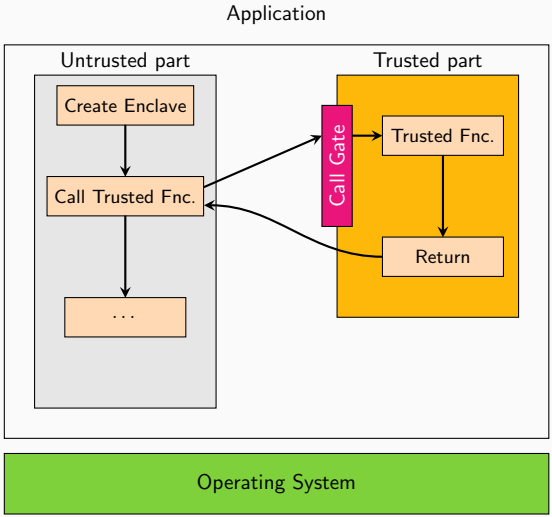

Application
Untrusted part
Create Enclave
Call Trusted Fnc.
Trusted part
Call Gate
Trusted Fnc.
Operating System

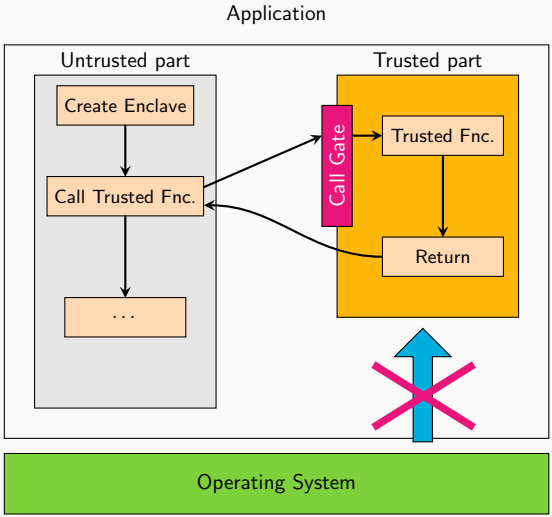

Application
Untrusted part
Trusted part
Create Enclave
Call Trusted Fnc.
Call Gate
Trusted Fnc.
Operating System

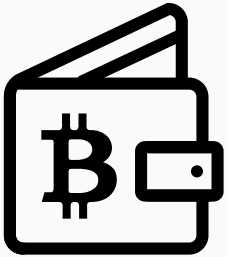

Application
Untrusted part
Trusted part
Create Enclave
Call Trusted Fnc.
Call Gate
Trusted Fnc.
Operating System


Application
Untrusted part
Trusted part
Create Enclave
Call Trusted Fnc.
Call Gate
Trusted Fnc.
Return
Operating System

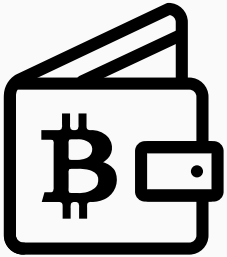

Application
Untrusted part
Trusted part
Create Enclave
Call Trusted Fnc.
Call Gate
Trusted Fnc.
Return
Operating System


Application
Untrusted part
Trusted part
Create Enclave
Call Trusted Fnc.
. . .
Call Gate
Trusted Fnc.
Return
Operating System


Application
Untrusted part
Trusted part
Create Enclave
Call Trusted Fnc.
...
Call Gate
Trusted Fnc.
Return
Operating System



- Ledger SGX Enclave for blockchain applications
- BitPay Copay Bitcoin wallet
- Teechain payment channel using SGX

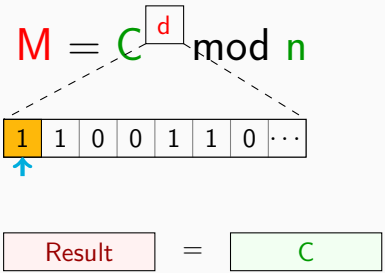


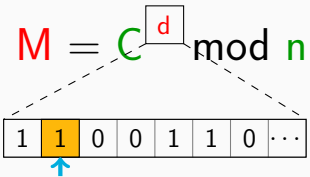
- Ledger SGX Enclave for blockchain applications
- BitPay Copay Bitcoin wallet
- Teechain payment channel using SGX

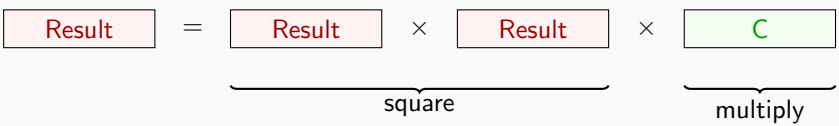
Teechain

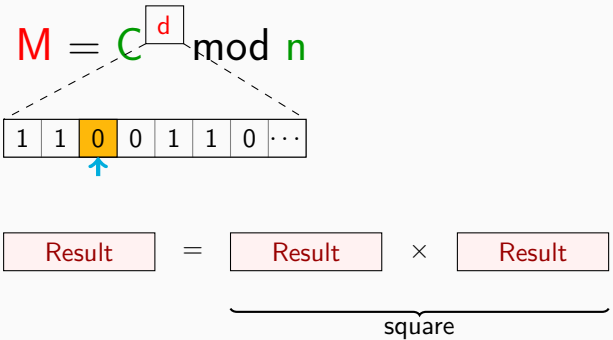
[...] We assume the TEE guarantees to hold and do not consider side-channel attacks [5, 35, 46] on the TEE. Such attacks and their mitigations [36, 43] are outside the scope of this work. [...]

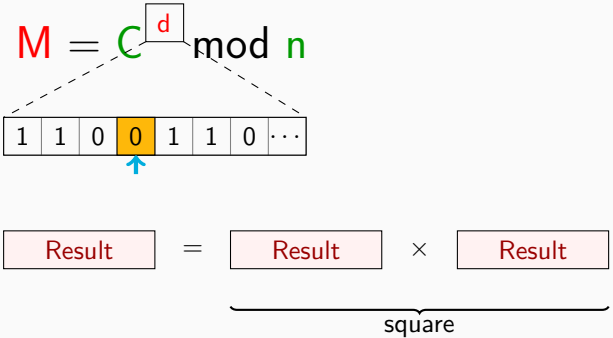
$$M = C^{d} \bmod n$$

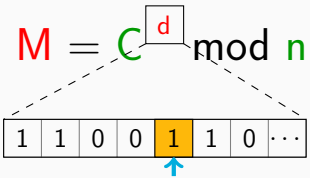

M = C^d mod n
1 1 0 0 1 1 0 ···
Result = C

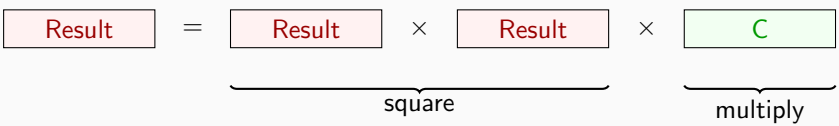

M = C^d mod n
1 1 0 0 1 1 0 ···

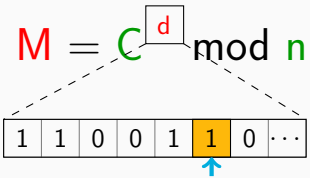

Result = Result × Result × C
square
multiply

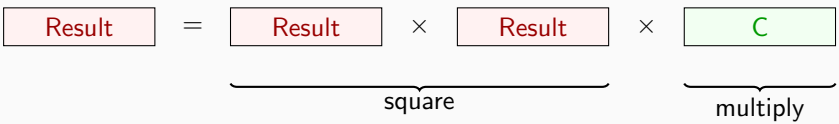

M = C^d mod n
1 1 0 0 1 1 0 ···
Result = Result × Result
square

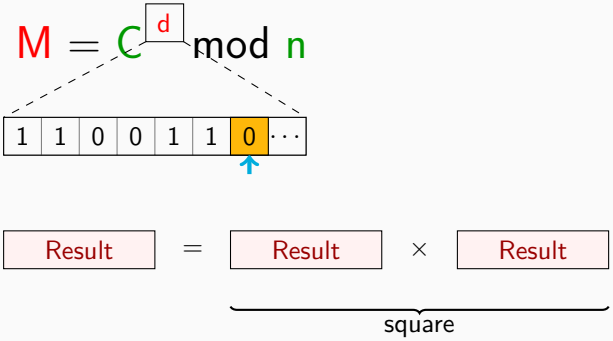

M = C^d mod n
1 1 0 0 1 1 0 ···
Result = Result × Result
square

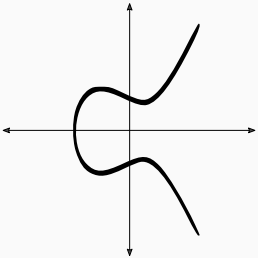

M = C^d mod n
1 1 0 0 1 1 0 ···


Result = Result × Result × C
square
multiply

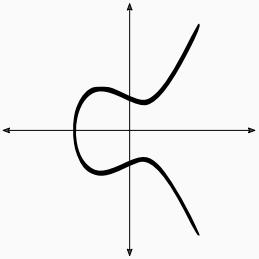

M = C^d mod n
1 1 0 0 1 1 0 ···


Result = Result × Result × C
square
multiply

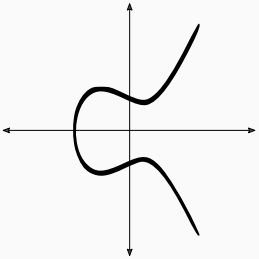

M = C^d mod n
1 1 0 0 1 1 0 ···
Result = Result × Result
square



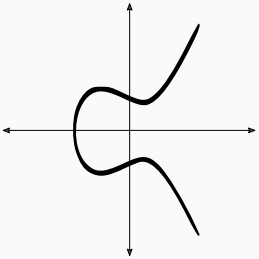
- Used to sign transactions



- Used to sign transactions
- Point multiplication is similar to RSA exponentiation



- Used to sign transactions
- Point multiplication is similar to RSA exponentiation
- Simplest implemention double-and-add or constant-time Montgomery ladder



- Used to sign transactions
- Point multiplication is similar to RSA exponentiation
- Simplest implemention double-and-add or constant-time Montgomery ladder
- Both algorithms have secret-dependent memory accesses

Prime+Probe [OST06; Liu+15; Mau+17]...

Prime+Probe [OST06; Liu+15; Mau+17]...

- exploits the timing difference when accessing...

Prime+Probe [OST06; Liu+15; Mau+17]...

- exploits the timing difference when accessing...
  - cached data (fast)

Prime+Probe [OST06; Liu+15; Mau+17]...

- exploits the timing difference when accessing...
  - cached data (fast)
  - uncached data (slow)

Prime+Probe [OST06; Liu+15; Mau+17]...

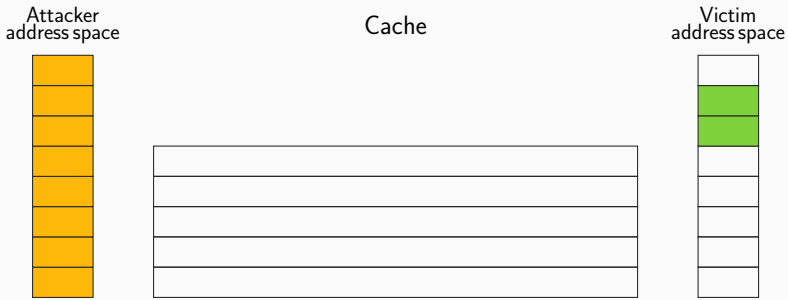
- exploits the timing difference when accessing...
  - cached data (fast)
  - uncached data (slow)
- is used to attack secret-dependent memory accesses

Prime+Probe [OST06; Liu+15; Mau+17]...

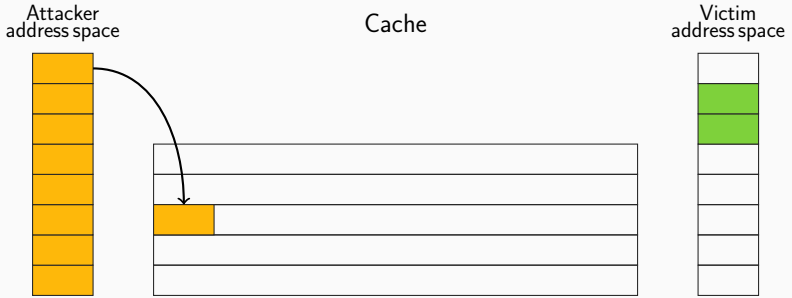
- exploits the timing difference when accessing...
  - cached data (fast)
  - uncached data (slow)
- is used to attack secret-dependent memory accesses
- is applied to a part of the CPU cache, a cache set

Prime+Probe [OST06; Liu+15; Mau+17]...

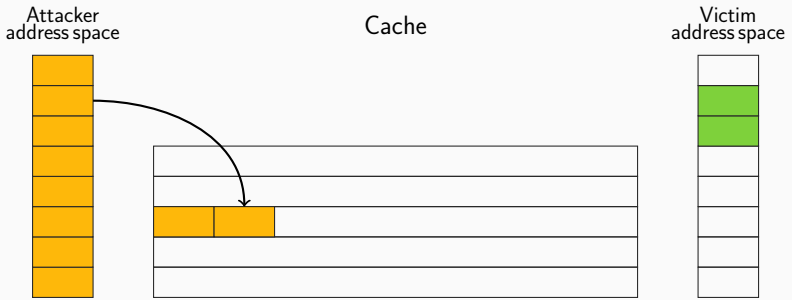
- exploits the timing difference when accessing...
  - cached data (fast)
  - uncached data (slow)
- is used to attack secret-dependent memory accesses
- is applied to a part of the CPU cache, a cache set
- works across CPU cores as the last-level cache is shared



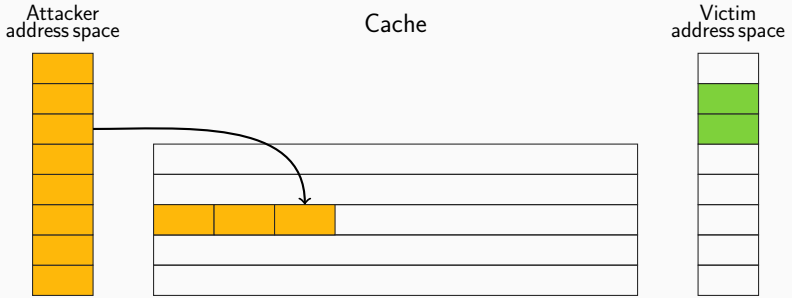
Step 0: Attacker fills the cache (prime)



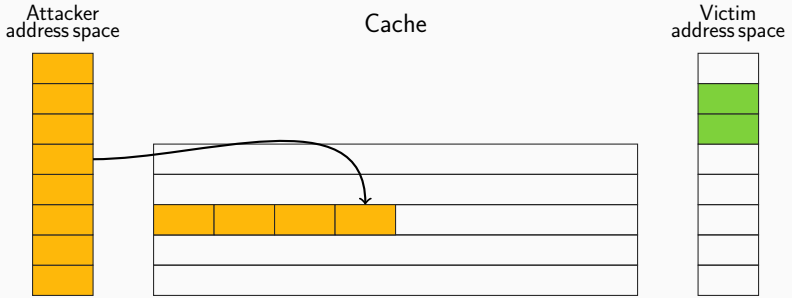
Step 0: Attacker fills the cache (prime)



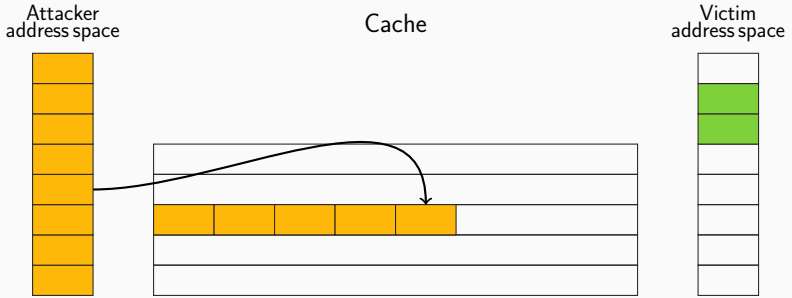
Step 0: Attacker fills the cache (prime)



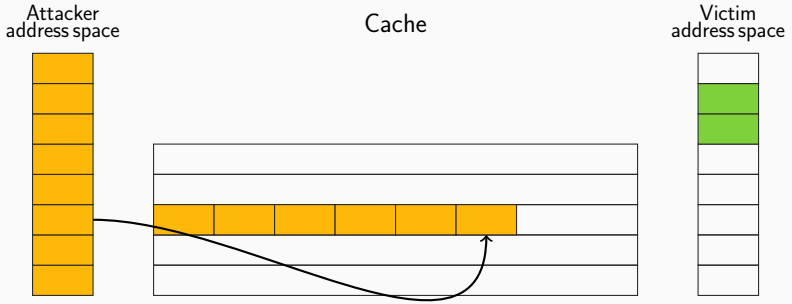
Step 0: Attacker fills the cache (prime)



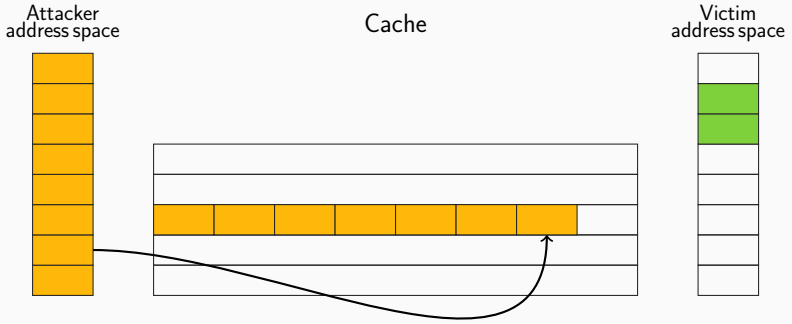
Step 0: Attacker fills the cache (prime)



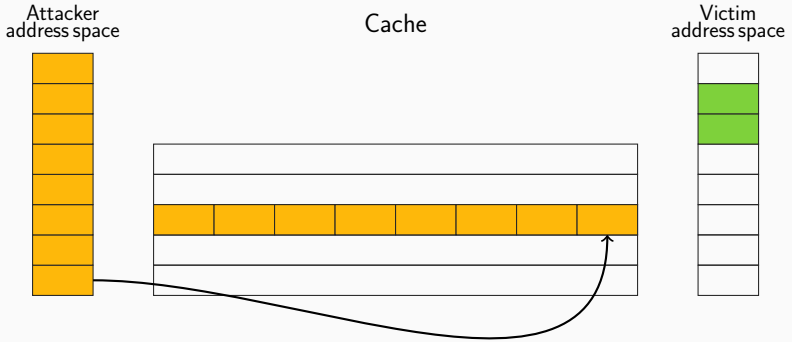
Step 0: Attacker fills the cache (prime)



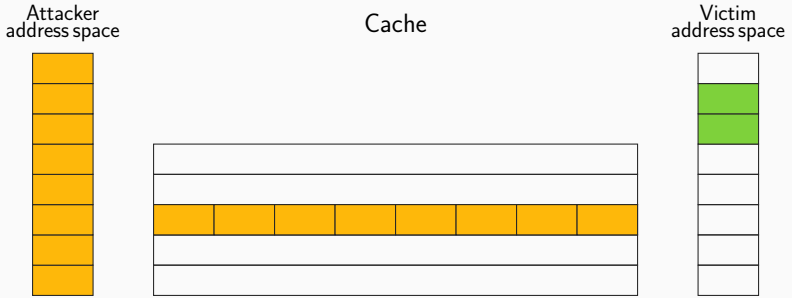
Step 0: Attacker fills the cache (prime)



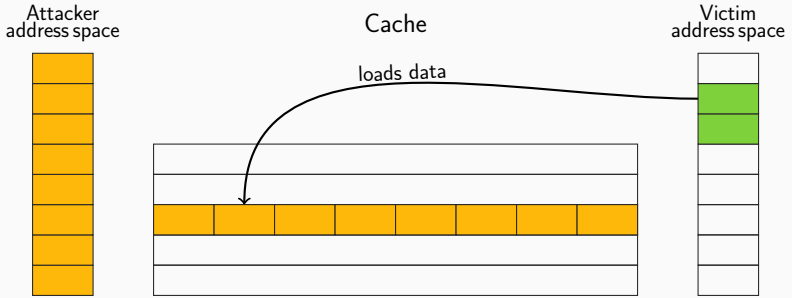
Step 0: Attacker fills the cache (prime)



Step 0: Attacker fills the cache (prime)

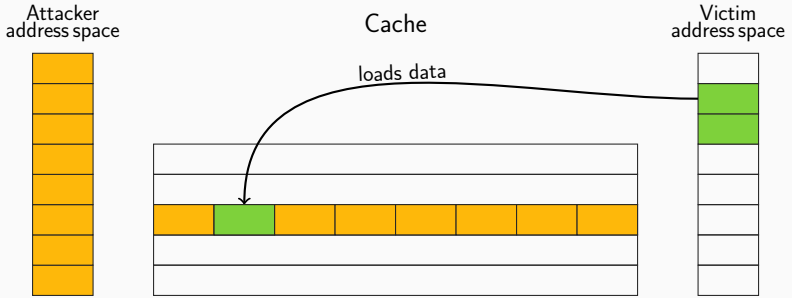


Step 0: Attacker fills the cache (prime)



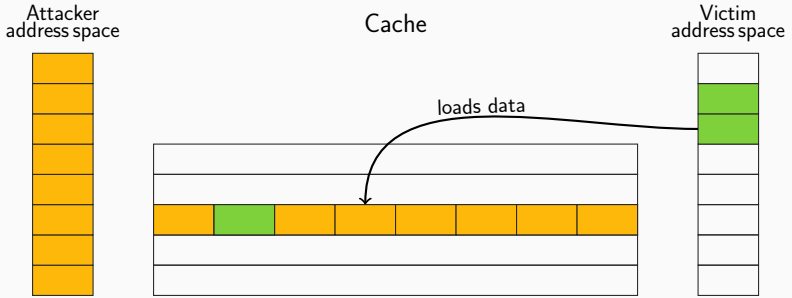
Step 0: Attacker fills the cache (prime)

Step 1: Victim evicts cache lines by accessing own data



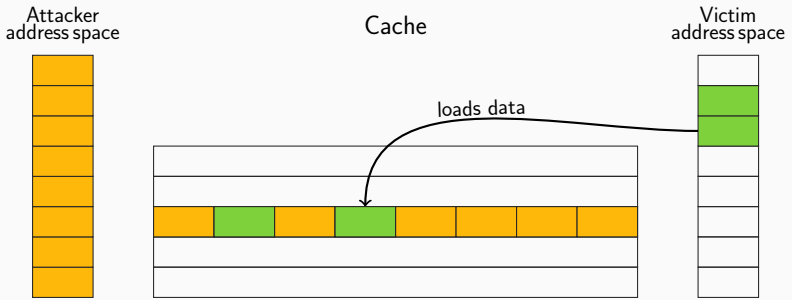
Step 0: Attacker fills the cache (prime)

Step 1: Victim evicts cache lines by accessing own data



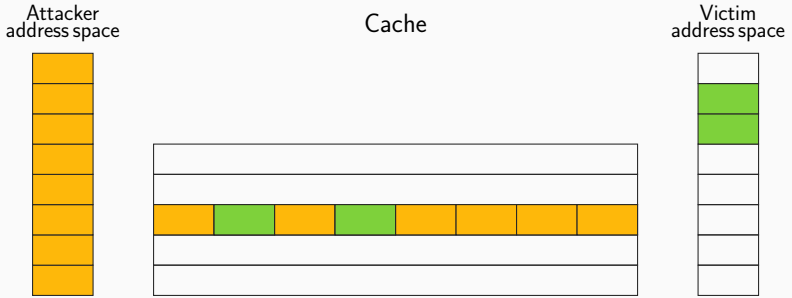
Step 0: Attacker fills the cache (prime)

Step 1: Victim evicts cache lines by accessing own data



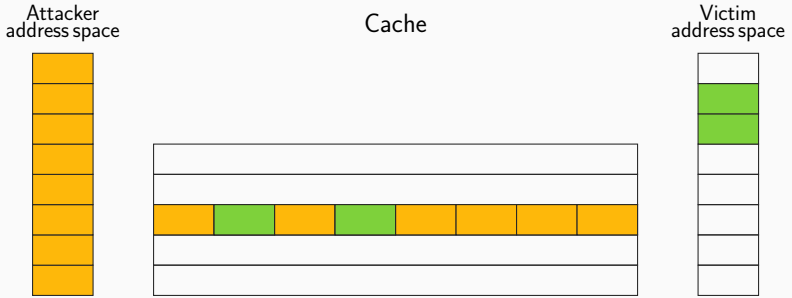
Step 0: Attacker fills the cache (prime)

Step 1: Victim evicts cache lines by accessing own data



Step 0: Attacker fills the cache (prime)

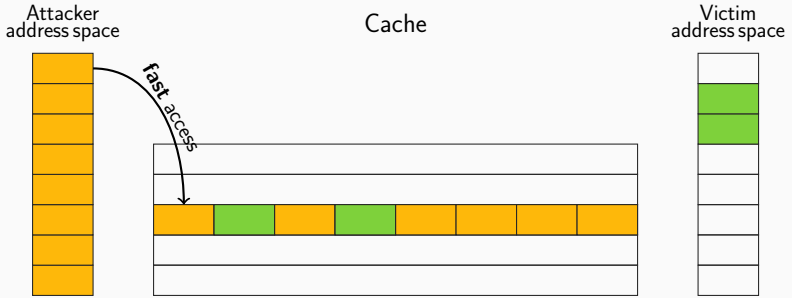
Step 1: Victim evicts cache lines by accessing own data



Step 0: Attacker fills the cache (prime)

Step 1: Victim evicts cache lines by accessing own data

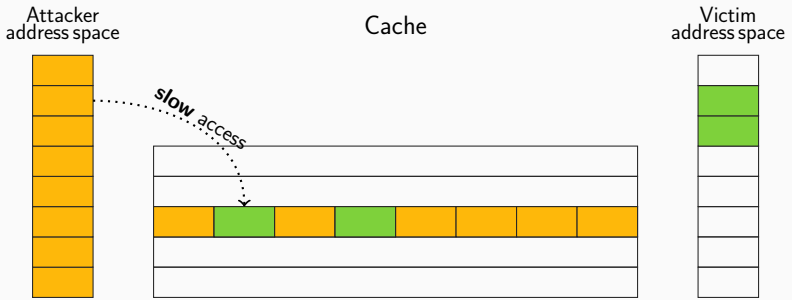
Step 2: Attacker probes data to determine if the set was accessed



Step 0: Attacker fills the cache (prime)

Step 1: Victim evicts cache lines by accessing own data

Step 2: Attacker probes data to determine if the set was accessed



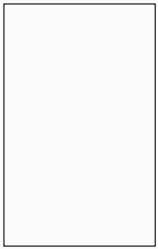
Step 0: Attacker fills the cache (prime)

Step 1: Victim evicts cache lines by accessing own data

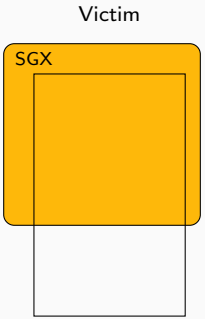
Step 2: Attacker probes data to determine if the set was accessed

Attack

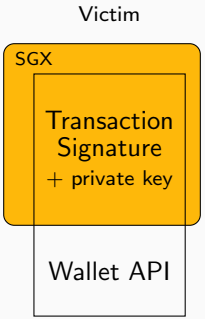
Victim

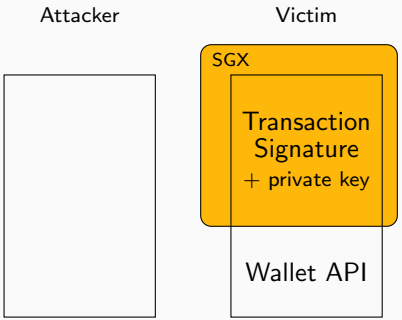


Attack Settings

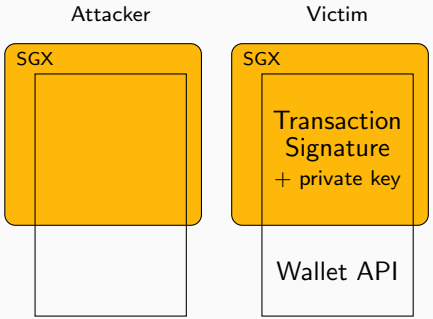


Attack Settings

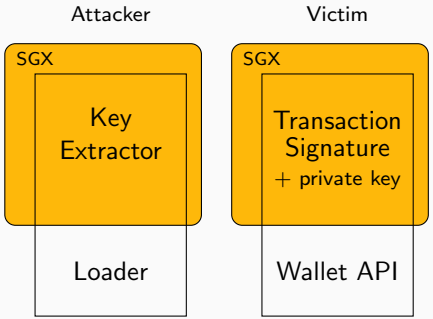



Attacker
Victim
SGX
Transaction
Signature
+ private key
Wallet API

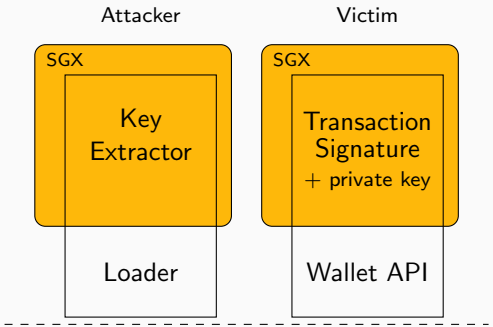
Attack Settings

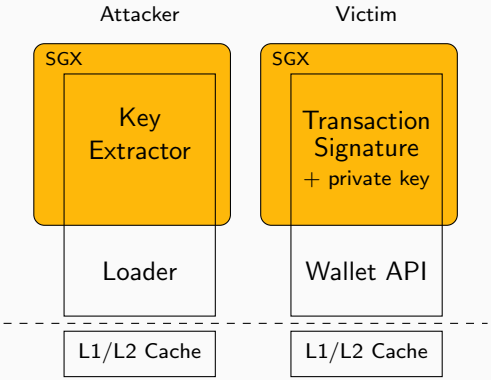


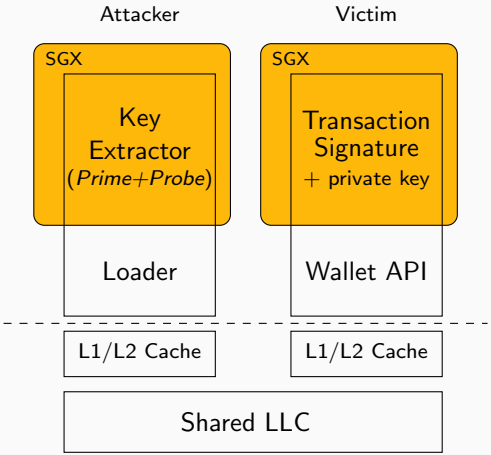
Attack Settings



Attack Settings




Attacker
Victim
SGX
Key Extractor
Loader
SGX
Transaction Signature
+ private key
Wallet API
L1/L2 Cache
L1/L2 Cache


Attacker
Victim
SGX
Key Extractor (*Prime+Probe*)
SGX
Transaction Signature + private key
Loader
Wallet API
L1/L2 Cache
L1/L2 Cache
Shared LLC

Classical Prime+Probe cannot be mounted within SGX:

Classical Prime+Probe cannot be mounted within SGX:

- No access to high-precision timer (`rdtsc`)

Classical Prime+Probe cannot be mounted within SGX:

- No access to high-precision timer (`rdtsc`)
- No syscalls

Classical Prime+Probe cannot be mounted within SGX:

- No access to high-precision timer (`rdtsc`)
- No syscalls
- No shared memory

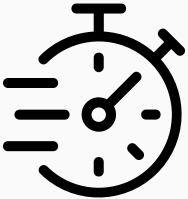
Classical Prime+Probe cannot be mounted within SGX:

- No access to high-precision timer (`rdtsc`)
- No syscalls
- No shared memory
- No physical addresses

Classical Prime+Probe cannot be mounted within SGX:

- No access to high-precision timer (`rdtsc`)
- No syscalls
- No shared memory
- No physical addresses
- No 2 MB large pages

- We have to build our own timer



- We have to build our own timer
- Timer resolution must be in the order of cycles

- We have to build our own timer
- Timer resolution must be in the order of cycles
- Start a thread that continuously increments a global variable

- We have to build our own timer
- Timer resolution must be in the order of cycles
- Start a thread that continuously increments a global variable
- The global variable is our timestamp

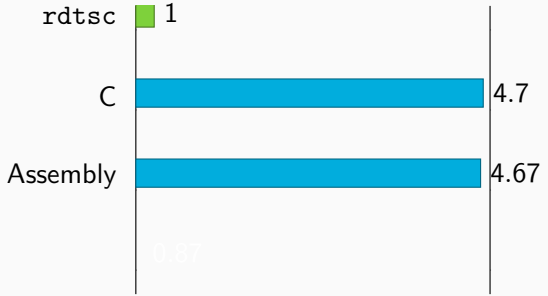
CPU cycles one increment takes


```
1 timestamp = rdtsc();
```

CPU cycles one increment takes

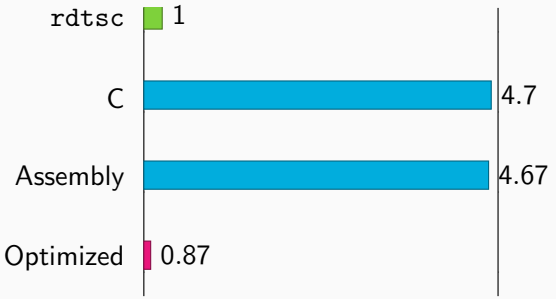

```
1 while(1) {
2   timestamp++;
3 }
```

CPU cycles one increment takes

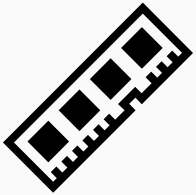



```
1 mov &timestamp, %rcx
2 1: incl (%rcx)
3 jmp 1b
```

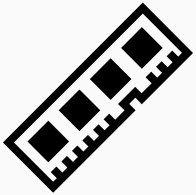
CPU cycles one increment takes



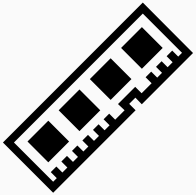

```
1 mov &timestamp, %rcx
2 1: inc %rax
3 mov %rax, (%rcx)
4 jmp 1b
```



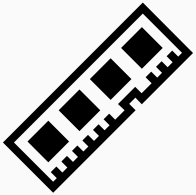
- Cache set is determined by part of physical address [Mau+15]



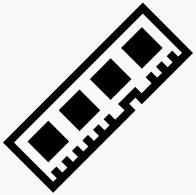
- Cache set is determined by part of physical address [Mau+15]
- We have no knowledge of physical addresses



- Cache set is determined by part of physical address [Mau+15]
- We have no knowledge of physical addresses
- Use the reverse-engineered DRAM mapping [Pes+16]

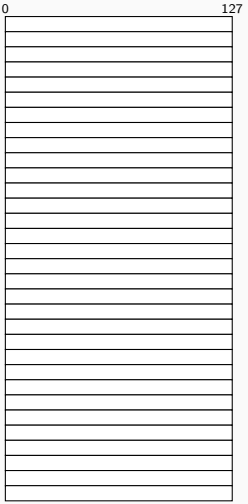


- Cache set is determined by part of physical address [Mau+15]
- We have no knowledge of physical addresses
- Use the reverse-engineered DRAM mapping [Pes+16]
- Exploit timing differences to find DRAM row borders



- Cache set is determined by part of physical address [Mau+15]
- We have no knowledge of physical addresses
- Use the reverse-engineered DRAM mapping [Pes+16]
- Exploit timing differences to find DRAM row borders
- The 18 LSBs are '0' at a row border

Physical Addresses

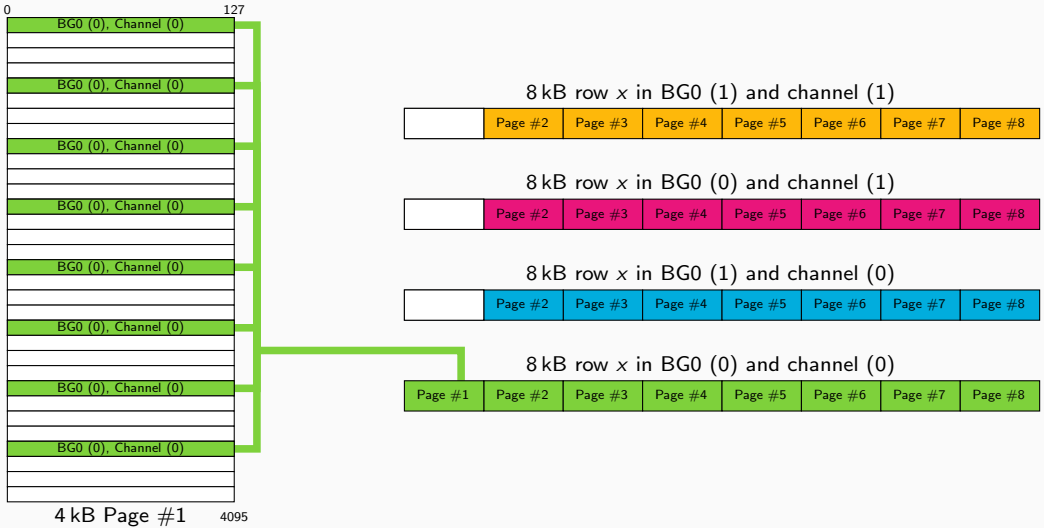


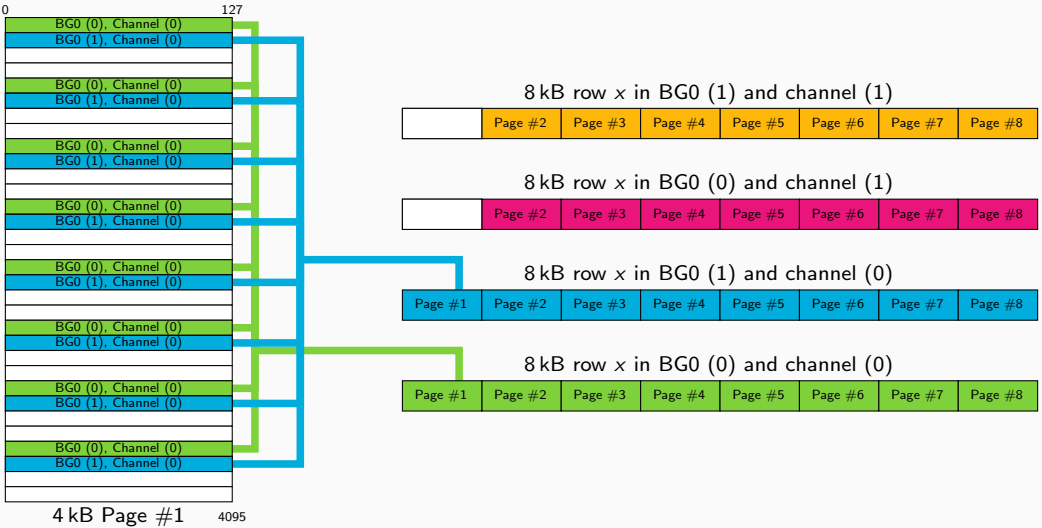
8 kB row x in BG0 (1) and channel (1)

8 kB row x in BG0 (0) and channel (1)

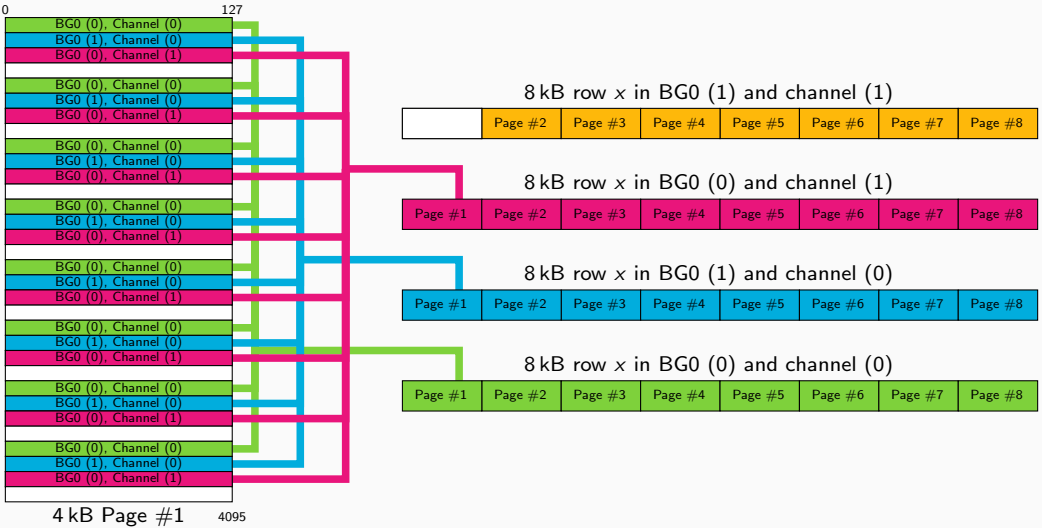
8 kB row x in BG0 (1) and channel (0)

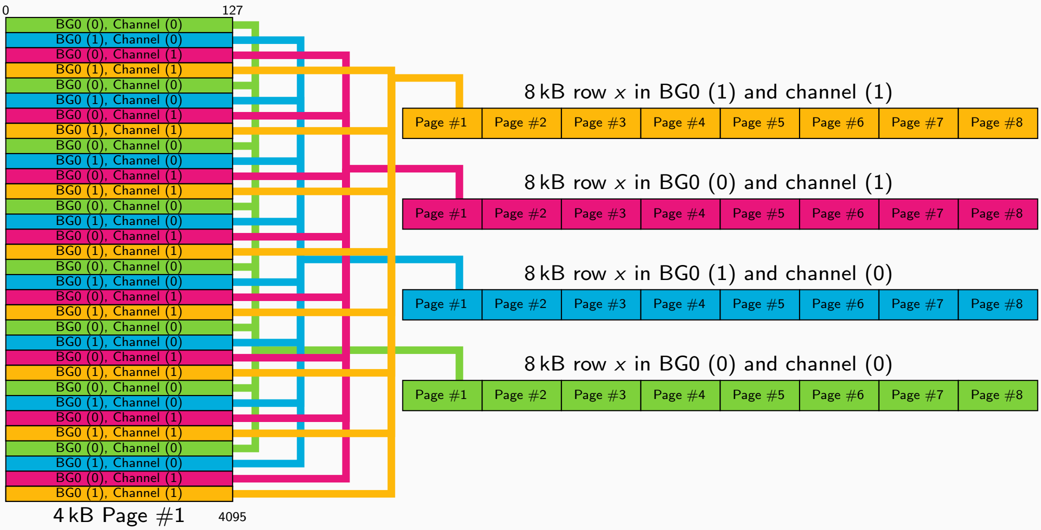
8 kB row x in BG0 (0) and channel (0)


0
127
BG0 (0), Channel (0)
BG0 (0), Channel (0)
BG0 (0), Channel (0)
BG0 (0), Channel (0)
BG0 (0), Channel (0)
BG0 (0), Channel (0)
BG0 (0), Channel (0)
BG0 (0), Channel (0)
4 kB Page #1
4095
8 kB row x in BG0 (1) and channel (1)
Page #2
Page #3
Page #4
Page #5
Page #6
Page #7
Page #8
8 kB row x in BG0 (0) and channel (1)
Page #2
Page #3
Page #4
Page #5
Page #6
Page #7
Page #8
8 kB row x in BG0 (1) and channel (0)
Page #2
Page #3
Page #4
Page #5
Page #6
Page #7
Page #8
8 kB row x in BG0 (0) and channel (0)
Page #1
Page #2
Page #3
Page #4
Page #5
Page #6
Page #7
Page #8


0
127
BG0 (0), Channel (0)
BG0 (1), Channel (0)
4 kB Page #1
4095
8 kB row x in BG0 (1) and channel (1)
Page #2
Page #3
Page #4
Page #5
Page #6
Page #7
Page #8
8 kB row x in BG0 (0) and channel (1)
8 kB row x in BG0 (1) and channel (0)
Page #1
8 kB row x in BG0 (0) and channel (0)

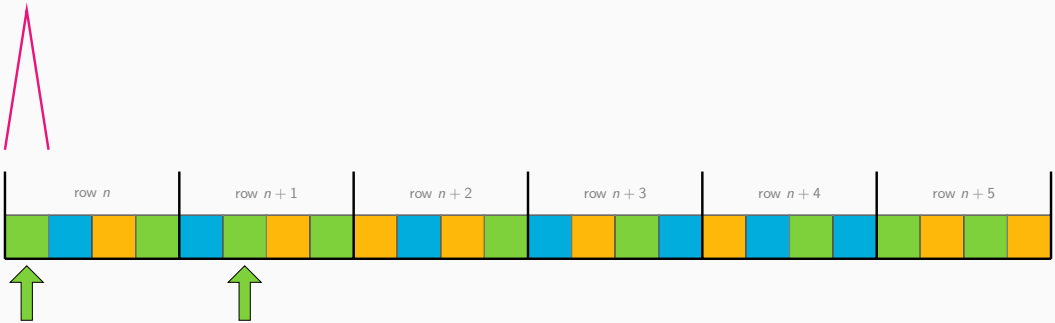
Physical Addresses



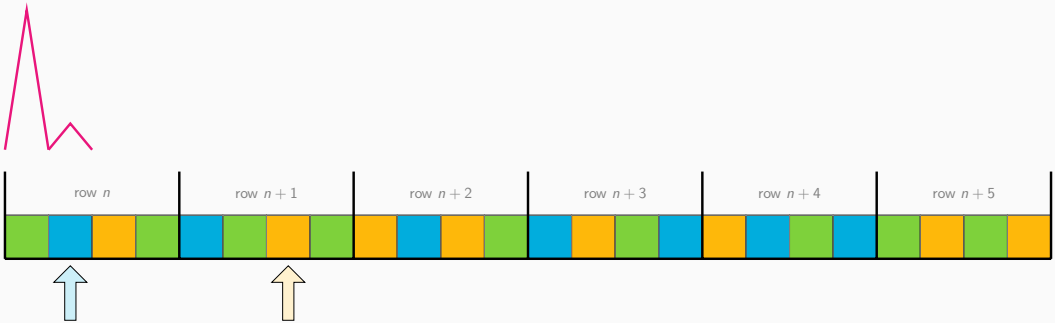


0
127
BG0 (0), Channel (0)
BG0 (1), Channel (0)
BG0 (0), Channel (1)
BG0 (1), Channel (1)
4 kB Page #1
4095
8 kB row x in BG0 (1) and channel (1)
Page #1 Page #2 Page #3 Page #4 Page #5 Page #6 Page #7 Page #8
8 kB row x in BG0 (0) and channel (1)
Page #1 Page #2 Page #3 Page #4 Page #5 Page #6 Page #7 Page #8
8 kB row x in BG0 (1) and channel (0)
Page #1 Page #2 Page #3 Page #4 Page #5 Page #6 Page #7 Page #8
8 kB row x in BG0 (0) and channel (0)
Page #1 Page #2 Page #3 Page #4 Page #5 Page #6 Page #7 Page #8

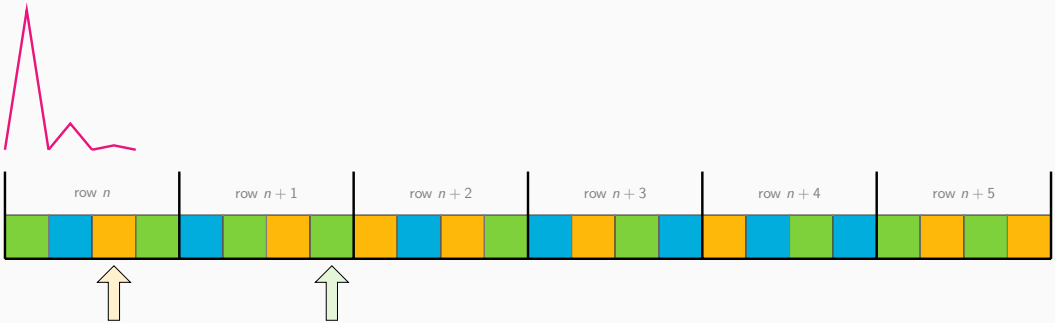
Physical Addresses



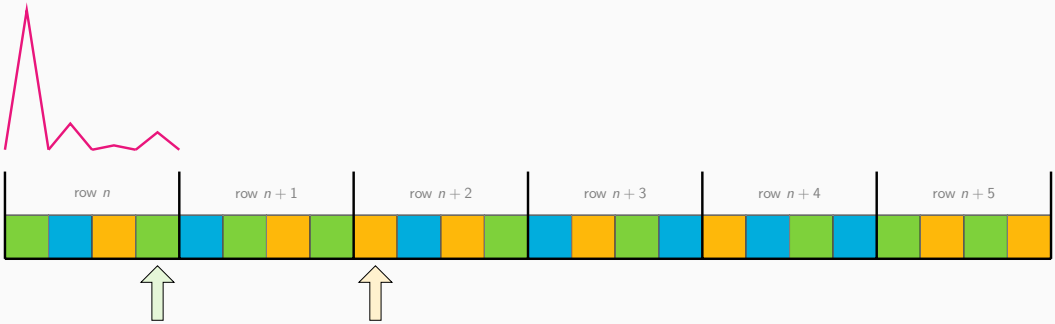
Physical Addresses



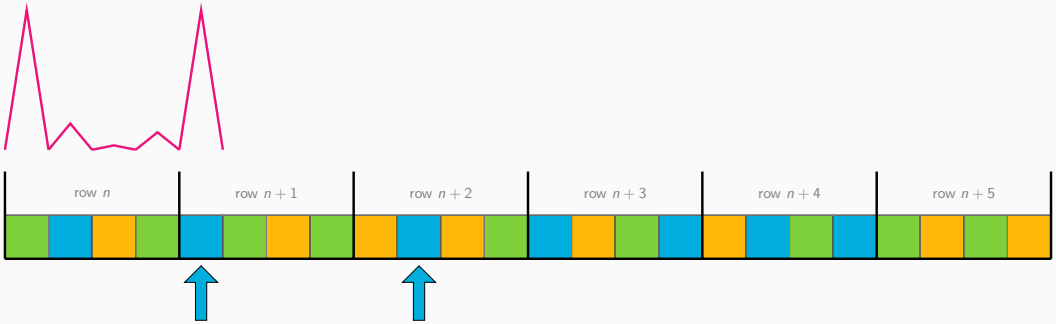
Physical Addresses

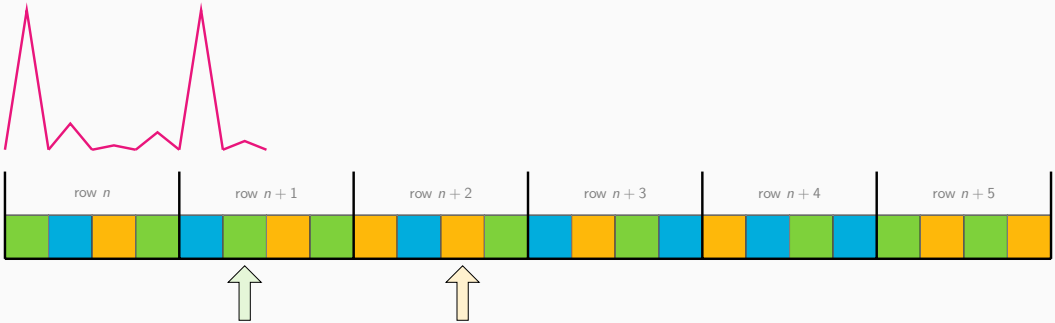


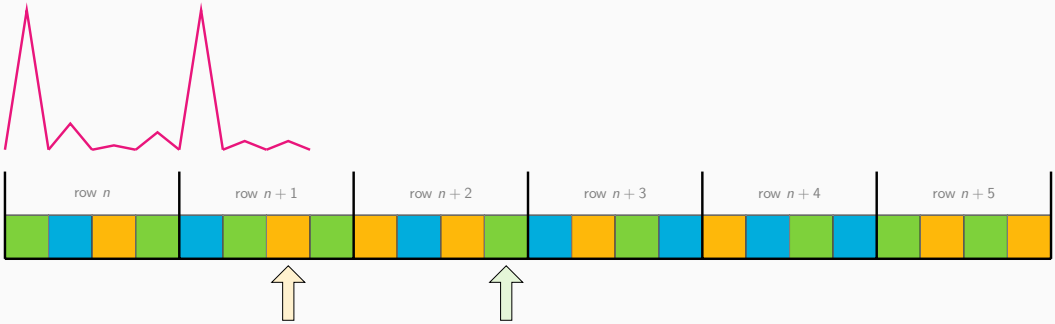
Physical Addresses



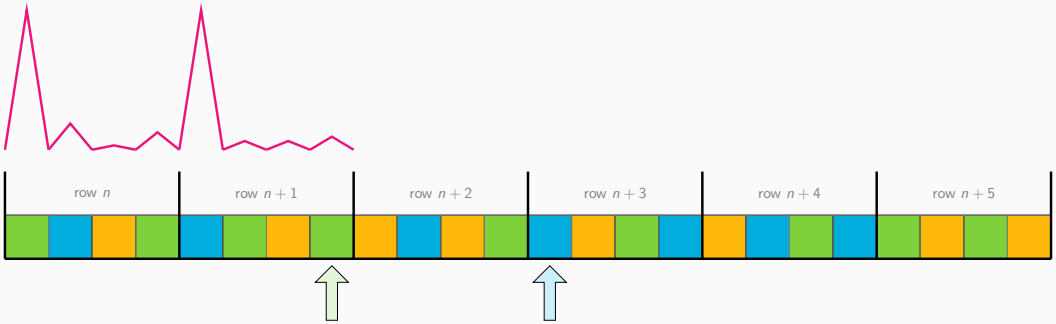
Physical Addresses

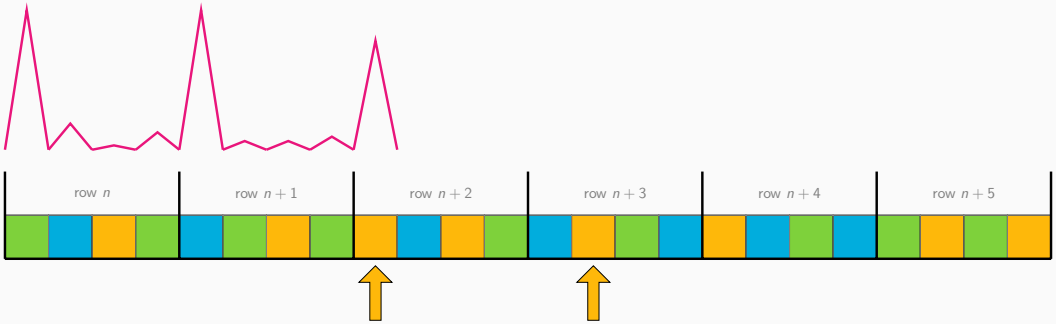



row n
row $n+1$
row $n+2$
row $n+3$
row $n+4$
row $n+5$

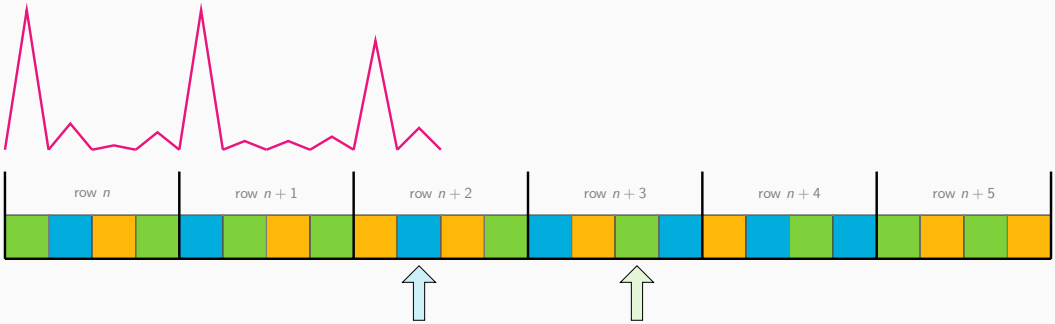

row n
row $n+1$
row $n+2$
row $n+3$
row $n+4$
row $n+5$

Physical Addresses

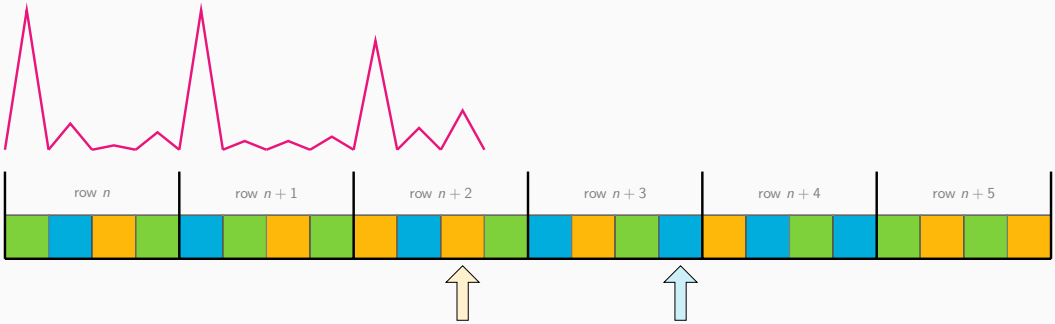



row n
row $n+1$
row $n+2$
row $n+3$
row $n+4$
row $n+5$

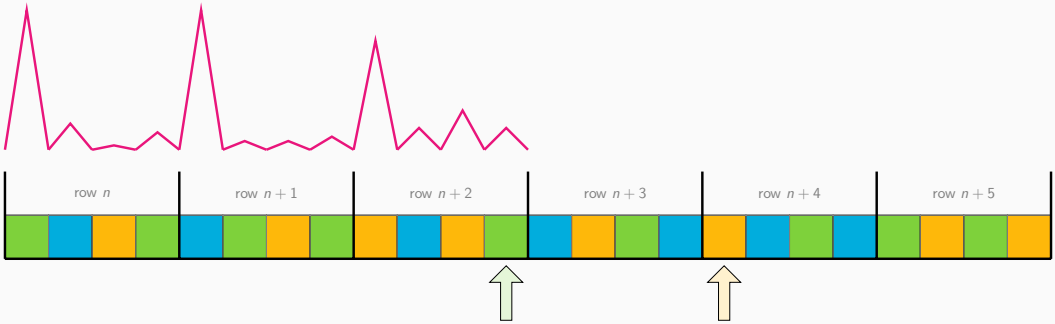
Physical Addresses



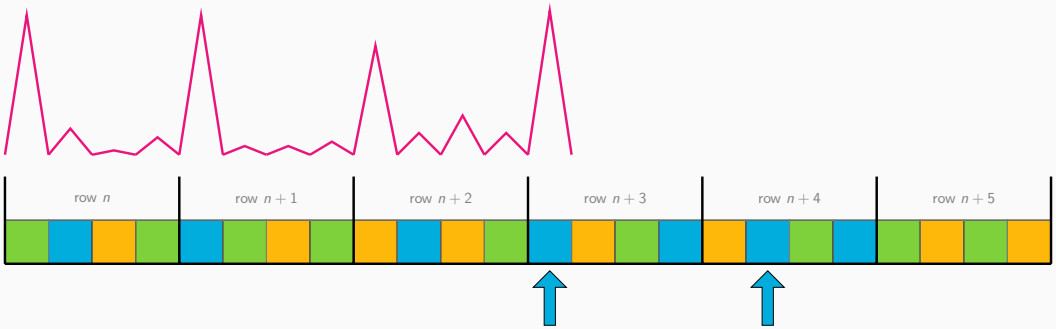
Physical Addresses



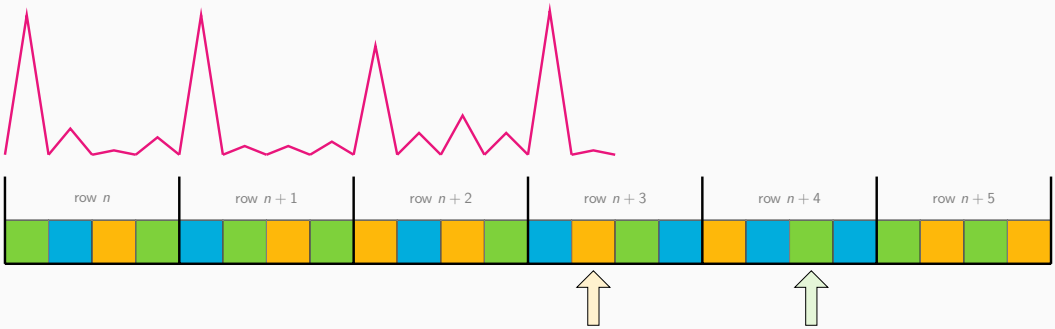
Physical Addresses



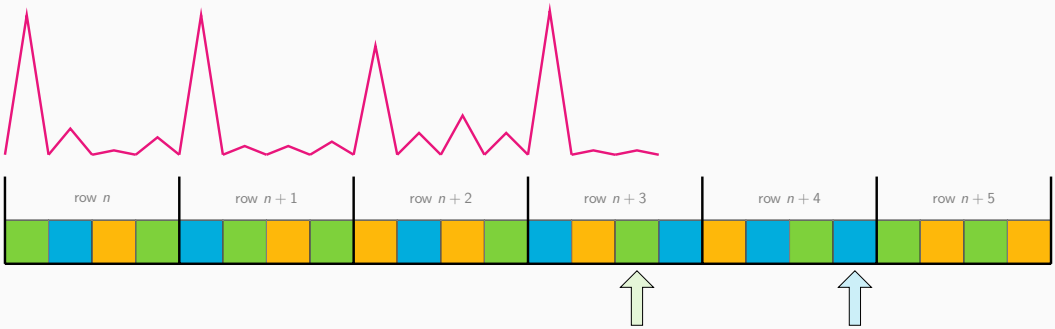
Physical Addresses

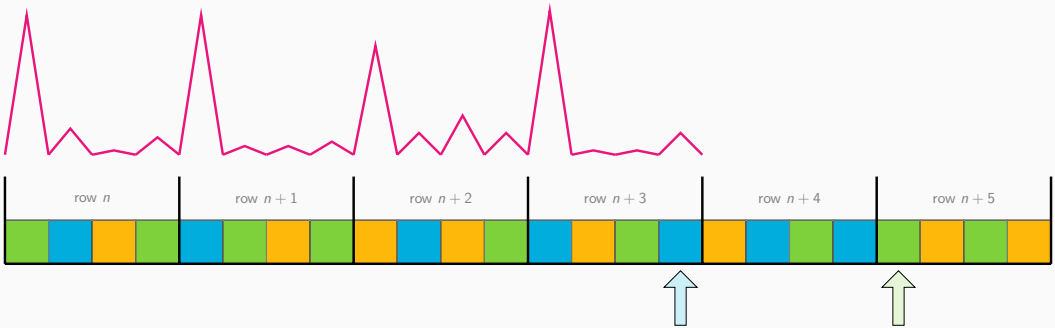


Physical Addresses

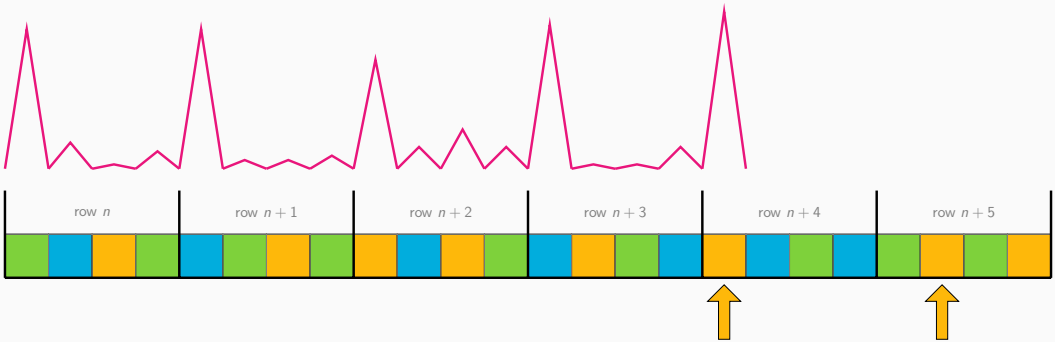


Physical Addresses

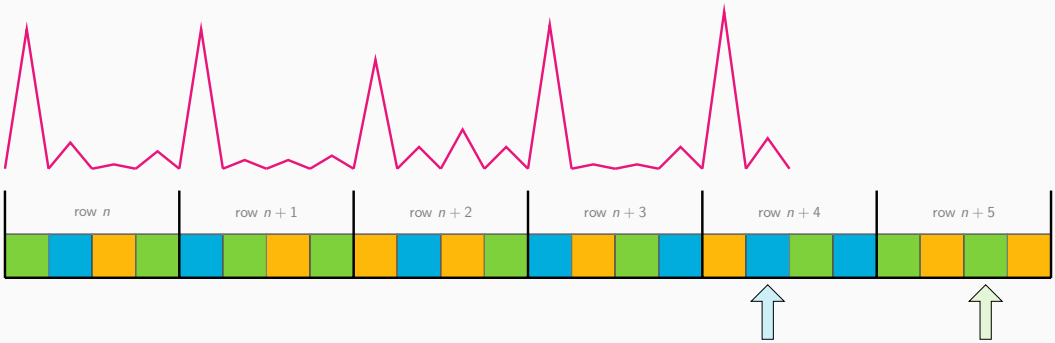



row n
row $n+1$
row $n+2$
row $n+3$
row $n+4$
row $n+5$

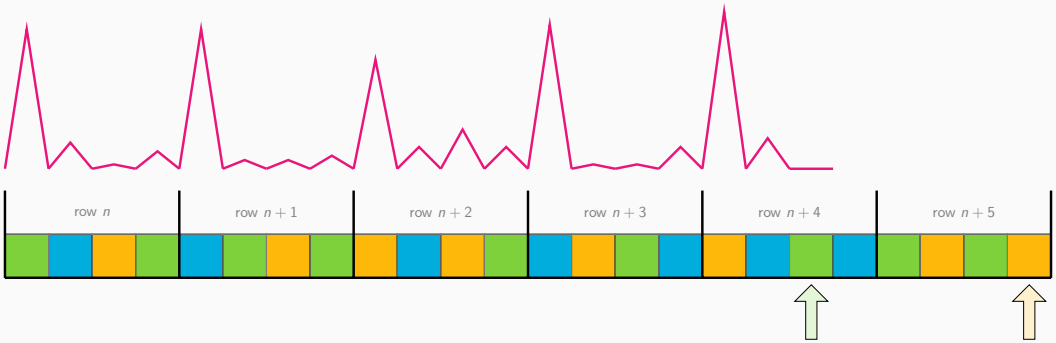
Physical Addresses



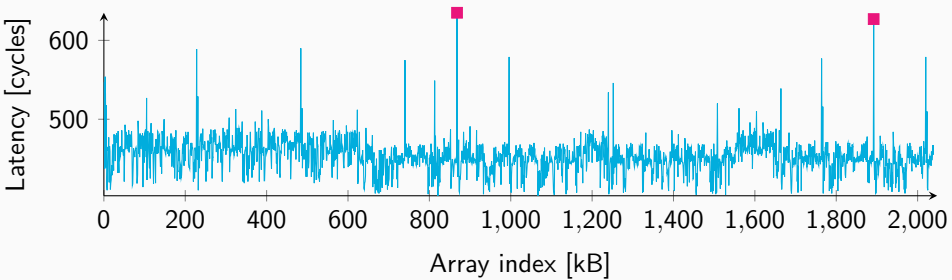
Physical Addresses

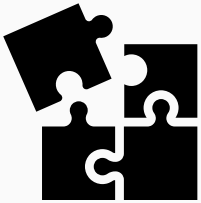


Physical Addresses



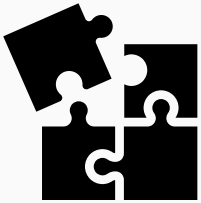
Result on an Intel i5-6200U





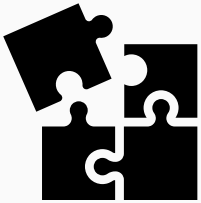
1. Use the counting primitive to measure DRAM accesses

Combining Everything

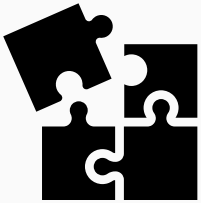


1. Use the counting primitive to measure DRAM accesses
2. Through the DRAM side channel, determine the row borders

Combining Everything

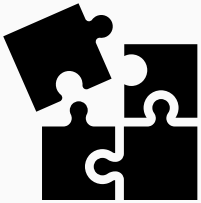


1. Use the counting primitive to measure DRAM accesses
2. Through the DRAM side channel, determine the row borders
3. Row borders have the 18 LSBs set to '0' $\rightarrow$ maps to cache set '0'



1. Use the counting primitive to measure DRAM accesses
2. Through the DRAM side channel, determine the row borders
3. Row borders have the 18 LSBs set to '0' $\rightarrow$ maps to cache set '0'
4. Build the eviction set for the Prime+Probe attack

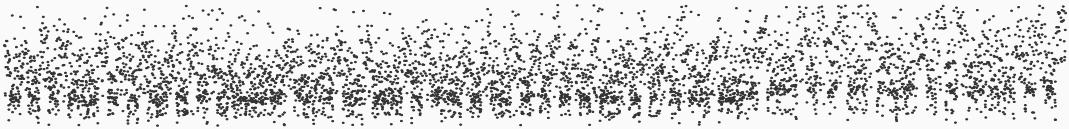
Combining Everything



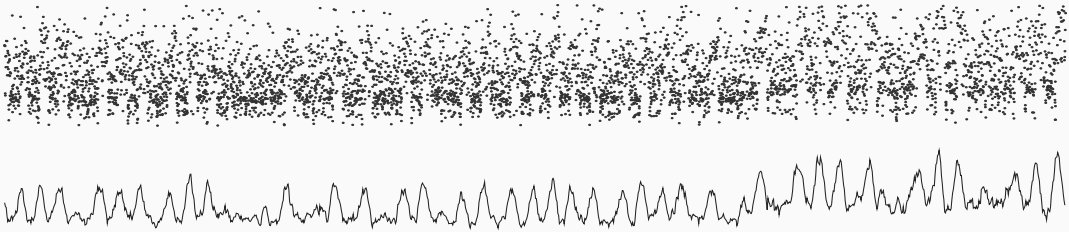
1. Use the counting primitive to measure DRAM accesses
2. Through the DRAM side channel, determine the row borders
3. Row borders have the 18 LSBs set to '0' $\rightarrow$ maps to cache set '0'
4. Build the eviction set for the Prime+Probe attack
5. Mount Prime+Probe on the buffer containing the multiplier [Sch+17]

Results

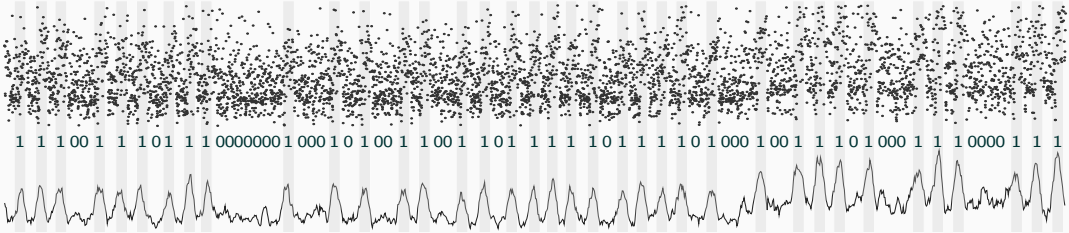
Raw Prime+Probe trace...

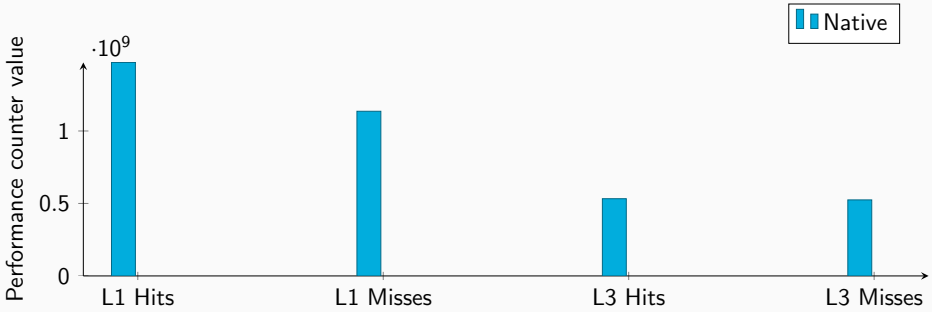


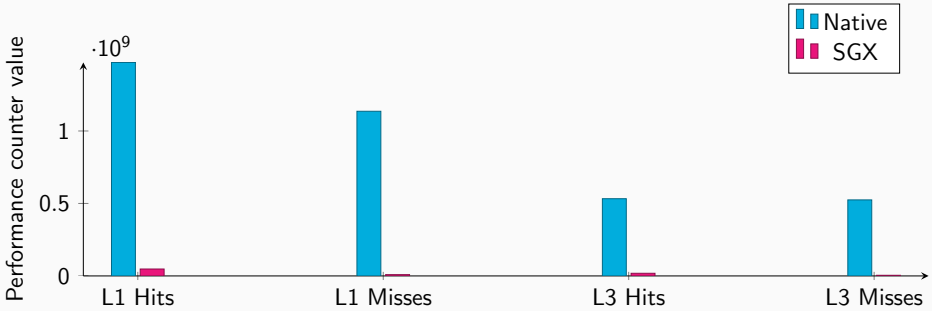
...processed with a simple moving average...



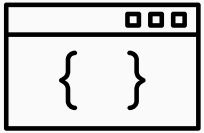
...allows to clearly see the bits of the exponent



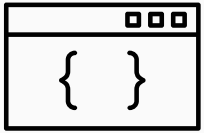

Native
Performance counter value
$\cdot 10^9$
1
0.5
0
L1 Hits
L1 Misses
L3 Hits
L3 Misses


Native
SGX
Performance counter value
$\cdot 10^9$
1
0.5
0
L1 Hits
L1 Misses
L3 Hits
L3 Misses

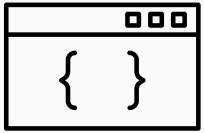
Countermeasures



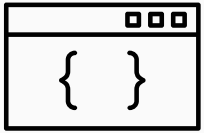
- Cache attacks can be prevented on source level



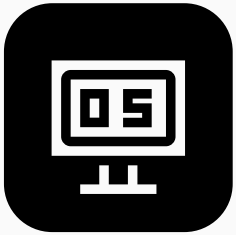
- Cache attacks can be prevented on source level
- Use side-channel resistant crypto implementations



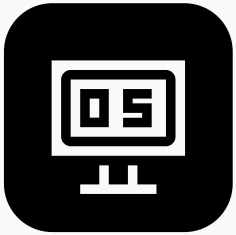
- Cache attacks can be prevented on source level
- Use side-channel resistant crypto implementations
- Exponent blinding for RSA prevents multi-trace attacks



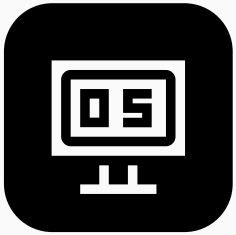
- Cache attacks can be prevented on source level
- Use side-channel resistant crypto implementations
- Exponent blinding for RSA prevents multi-trace attacks
- Bit-sliced implementations are not vulnerable to cache attacks



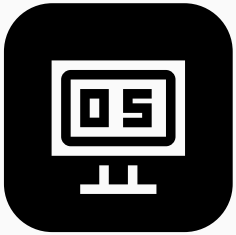
- Trusting the operating system weakens SGX threat model



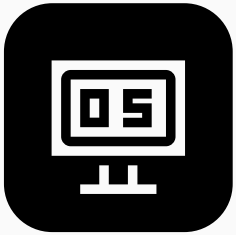
- Trusting the operating system weakens SGX threat model
- Method for the operating system to inspect enclave code



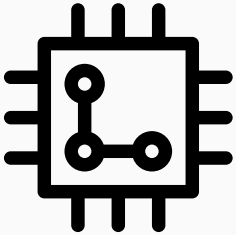
- Trusting the operating system weakens SGX threat model
- Method for the operating system to inspect enclave code
- Re-enable certain performance counters, such as L3 hits/misses



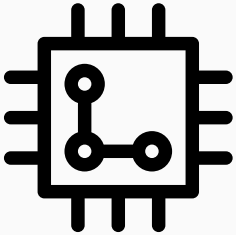
- Trusting the operating system weakens SGX threat model
- Method for the operating system to inspect enclave code
- Re-enable certain performance counters, such as L3 hits/misses
- Enclave coloring to prevent cross-enclave attacks



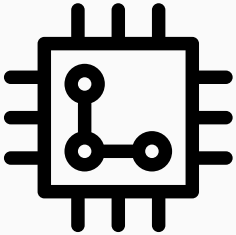
- Trusting the operating system weakens SGX threat model
- Method for the operating system to inspect enclave code
- Re-enable certain performance counters, such as L3 hits/misses
- Enclave coloring to prevent cross-enclave attacks
- Heap randomization to randomize cache sets



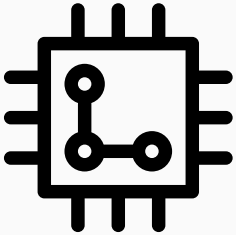
- Intel could prevent attacks by changing the hardware



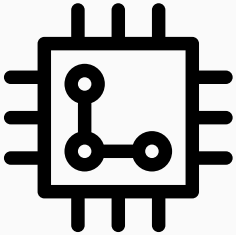
- Intel could prevent attacks by changing the hardware
- Combine Cache Allocation Technology (CAT) with SGX



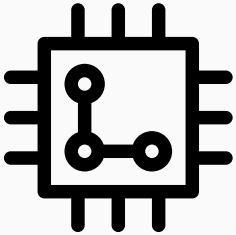
- Intel could prevent attacks by changing the hardware
- Combine Cache Allocation Technology (CAT) with SGX
  - Instead of controlling CAT from the OS, combine it with `eenter`



- Intel could prevent attacks by changing the hardware
- Combine Cache Allocation Technology (CAT) with SGX
  - Instead of controlling CAT from the OS, combine it with `eenter`
  - Entering an enclave would automatically activate CAT for this core

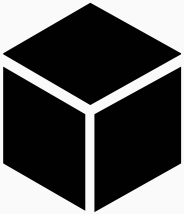


- Intel could prevent attacks by changing the hardware
- Combine Cache Allocation Technology (CAT) with SGX
  - Instead of controlling CAT from the OS, combine it with `eenter`
  - Entering an enclave would automatically activate CAT for this core
  - L3 is then isolated from all other enclaves and applications

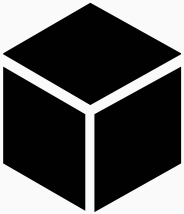


- Intel could prevent attacks by changing the hardware
- Combine Cache Allocation Technology (CAT) with SGX
  - Instead of controlling CAT from the OS, combine it with `eenter`
  - Entering an enclave would automatically activate CAT for this core
  - L3 is then isolated from all other enclaves and applications
- Provide a non-shared secure memory element which is not cached

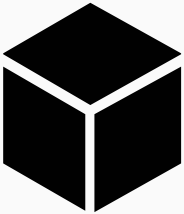
Exploiting Non-Side-Channel Bugs using Side-Channel Attacks



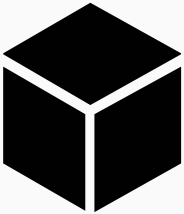
- Assuming a side-channel resistant crypto implementation



- Assuming a side-channel resistant crypto implementation
- Everything is inside the enclave



- Assuming a side-channel resistant crypto implementation
- Everything is inside the enclave
- No source or (unencrypted) binary available



- Assuming a side-channel resistant crypto implementation
- Everything is inside the enclave
- No source or (unencrypted) binary available
- A blackbox with an API


DRM
Video
Player

- A classical digitial rights management scenario

- A classical digitial rights management scenario
- Encrypted video file with a signed header

- A classical digitial rights management scenario
- Encrypted video file with a signed header
- The header contains the validity period

- A classical digitial rights management scenario
- Encrypted video file with a signed header
- The header contains the validity period
- Checks whether current time is within period and signature is valid

- A classical digitial rights management scenario
- Encrypted video file with a signed header
- The header contains the validity period
- Checks whether current time is within period and signature is valid
- Plays the file, e.g., using Intel's Protected Audio Video Path (PAVP)

- We do not like DRM

- We do not like DRM
- And we want to keep our stolen Bitcoins...

- We do not like DRM
- And we want to keep our stolen Bitcoins...
- ...not pay for the same movie over and over

- We do not like DRM
- And we want to keep our stolen Bitcoins...
- ...not pay for the same movie over and over
- No cache attack on crypto this time

A Glance at the Code

```
void play_file(DRMFile* file) {
  if(check_signature(file->header) == ERR_INVALID) {
    // invalid signature
    return;
  }

  int now = get_timestamp_secure();

  if(now < file->header.from || now > file->header.to) {
    // not allowed to watch it now
    return;
  }

  void* data = decrypt(file->content);
  play_secure(data);
}
```

DRM-protected file

```
void play_file(DRMFile* file) {
  if(check_signature(file->header) == ERR_INVALID) {
    // invalid signature
    return;
  }

  int now = get_timestamp_secure();

  if(now < file->header.from || now > file->header.to) {
    // not allowed to watch it now
    return;
  }

  void* data = decrypt(file->content);
  play_secure(data);
}
```

A Glance at the Code

DRM-protected file

Tampering detected

```
void play_file(DRMFile* file) {
  if(check_signature(file->header) == ERR_INVALID) {
    // invalid signature
    return;
  }

  int now = get_timestamp_secure();

  if(now < file->header.from || now > file->header.to) {
    // not allowed to watch it now
    return;
  }

  void* data = decrypt(file->content);
  play_secure(data);
}
```

A Glance at the Code


```
void play_file(DRMFile* file) {
  if(check_signature(file->header) == ERR_INVALID) {
    // invalid signature
    return;
  }

  int now = get_timestamp_secure();

  if(now < file->header.from || now > file->header.to) {
    // not allowed to watch it now
    return;
  }

  void* data = decrypt(file->content);
  play_secure(data);
}
```

DRM-protected file

Tampering detected

Expired

A Glance at the Code

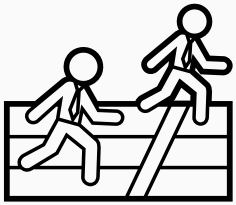

```
void play_file(DRMFile* file) {
  if(check_signature(file->header) == ERR_INVALID) {
    // invalid signature
    return;
  }

  int now = get_timestamp_secure();

  if(now < file->header.from || now > file->header.to) {
    // not allowed to watch it now
    return;
  }

  void* data = decrypt(file->content);
  play_secure(data);
}
```

DRM-protected file

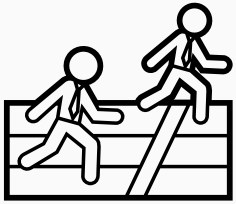
Tampering detected

Expired

Everything is fine

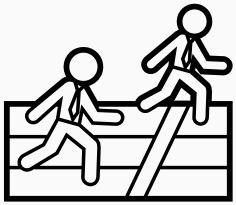


- Although the code looks correct, there is a problem



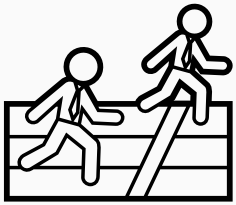
- Although the code looks correct, there is a problem
- SGX enclaves can access non-enclave memory

A Race Condition



- Although the code looks correct, there is a problem
- SGX enclaves can access non-enclave memory
- For convenience, memory is often shared between enclave and loader

A Race Condition



- Although the code looks correct, there is a problem
- SGX enclaves can access non-enclave memory
- For convenience, memory is often shared between enclave and loader
- This can create race conditions

A Glance at the Code - Again

```
void play_file(DRMFile* file) {
  if(check_signature(file->header) == ERR_INVALID) {
    // invalid signature
    return;
  }

  int now = get_timestamp_secure();

  if(now < file->header.from || now > file->header.to) {
    // not allowed to watch it now
    return;
  }

  void* data = decrypt(file->content);
  play_secure(data);
}
```

Shared memory

```
void play_file(DRMFile* file) {
  if(check_signature(file->header) == ERR_INVALID) {
    // invalid signature
    return;
  }

  int now = get_timestamp_secure();

  if(now < file->header.from || now > file->header.to) {
    // not allowed to watch it now
    return;
  }

  void* data = decrypt(file->content);
  play_secure(data);
}
```

A Glance at the Code - Again

Shared memory

First access of shared memory

```
void play_file(DRMFile* file) {
  if(check_signature(file->header) == ERR_INVALID) {
    // invalid signature
    return;
  }

  int now = get_timestamp_secure();

  if(now < file->header.from || now > file->header.to) {
    // not allowed to watch it now
    return;
  }

  void* data = decrypt(file->content);
  play_secure(data);
}
```

A Glance at the Code - Again


```
void play_file(DRMFile* file) {
  if(check_signature(file->header) == ERR_INVALID) {
    // invalid signature
    return;
  }

  int now = get_timestamp_secure();

  if(now < file->header.from || now > file->header.to) {
    // not allowed to watch it now
    return;
  }

  void* data = decrypt(file->content);
  play_secure(data);
}
```

Shared memory

First access of shared memory

Second access of shared memory

A Glance at the Code - Again

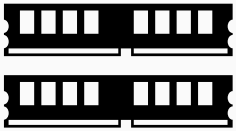

```
void play_file(DRMFile* file) {
  if(check_signature(file->header) == ERR_INVALID) {
    // invalid signature
    return;
  }

  int now = get_timestamp_secure();

  if(now < file->header.from || now > file->header.to) {
    // not allowed to watch it now
    return;
  }

  void* data = decrypt(file->content);
  play_secure(data);
}
```

Shared memory

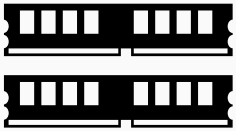
First access of shared memory

Headers can be changed here!

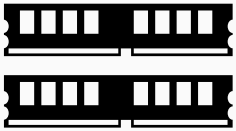
Second access of shared memory



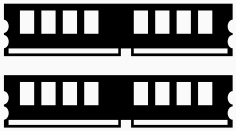
- A time-of-check-to-time-of-use (TOCTTOU) bug



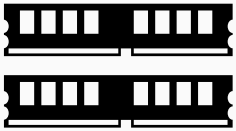
- A time-of-check-to-time-of-use (TOCTTOU) bug
- After the signature check, the memory might change



- A time-of-check-to-time-of-use (TOCTTOU) bug
- After the signature check, the memory might change
- Adversary can abuse this to set arbitrary validity period



- A time-of-check-to-time-of-use (TOCTTOU) bug
- After the signature check, the memory might change
- Adversary can abuse this to set arbitrary validity period
- Caused by accessing the shared memory twice



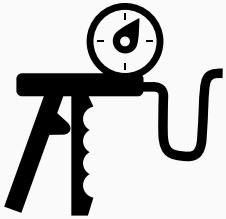
- A time-of-check-to-time-of-use (TOCTTOU) bug
- After the signature check, the memory might change
- Adversary can abuse this to set arbitrary validity period
- Caused by accessing the shared memory twice
- Also called double-fetch bugs

- We cheated by looking at the code

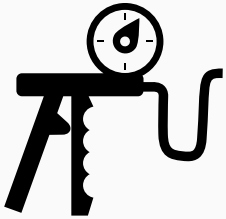
- We cheated by looking at the code
- What if we have neither code nor binary?

- We cheated by looking at the code
- What if we have neither code nor binary?
- Memory accesses are visible in the cache

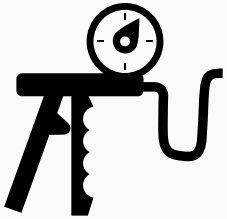
- We cheated by looking at the code
- What if we have neither code nor binary?
- Memory accesses are visible in the cache
- Observe memory accesses via cache attack



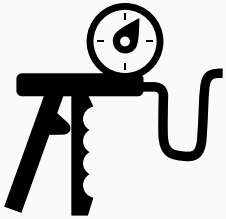
- Shared memory makes it a lot easier



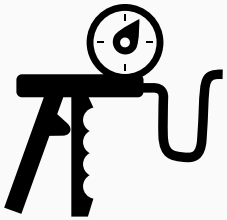
- Shared memory makes it a lot easier
- Constantly flush value from cache using `clflush` instruction



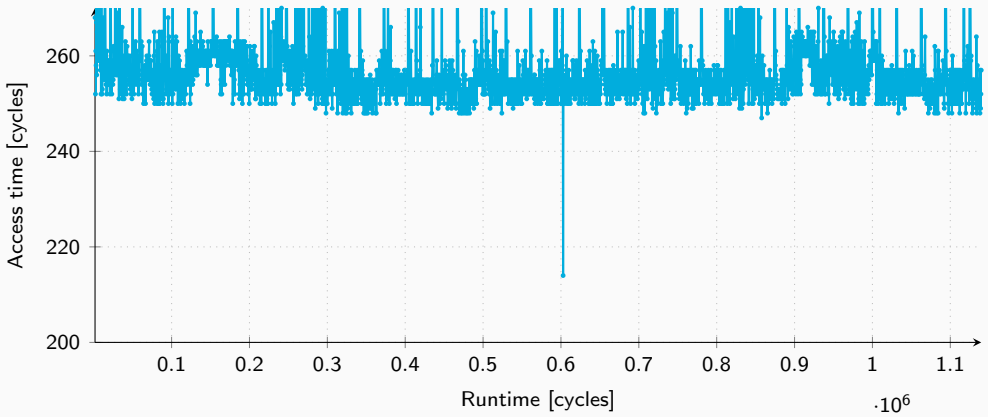
- Shared memory makes it a lot easier
- Constantly flush value from cache using `clflush` instruction
- Schedule victim

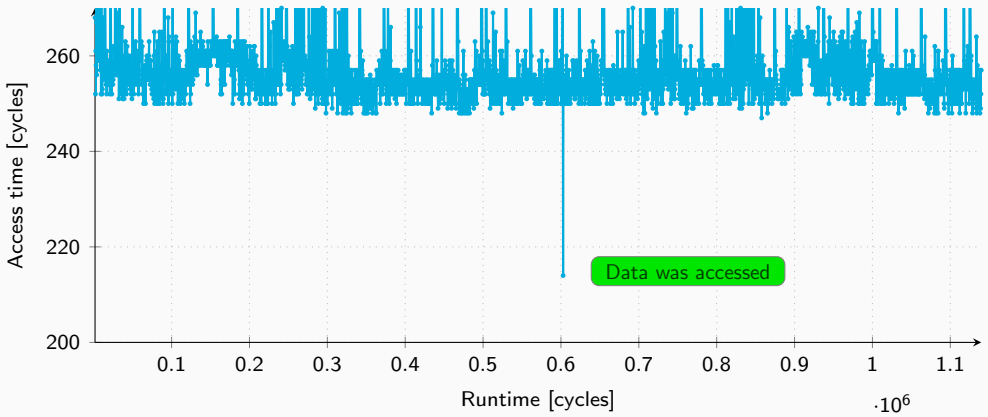


- Shared memory makes it a lot easier
- Constantly flush value from cache using `clflush` instruction
- Schedule victim
- Measure access time to value

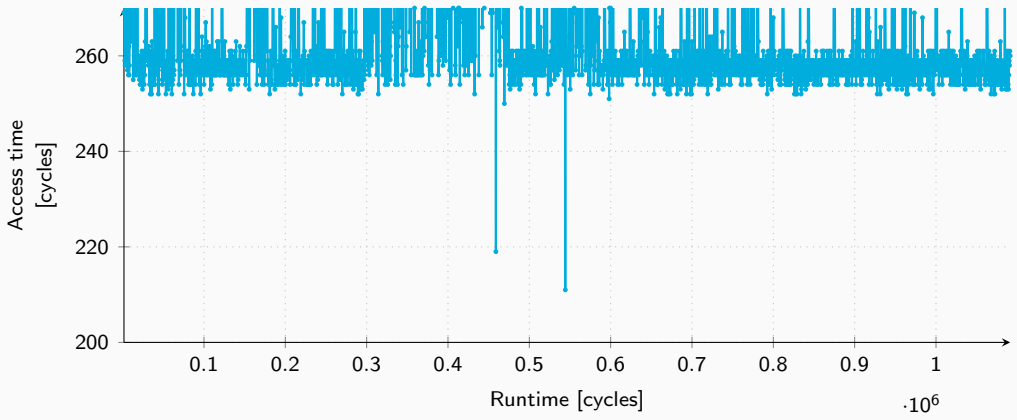


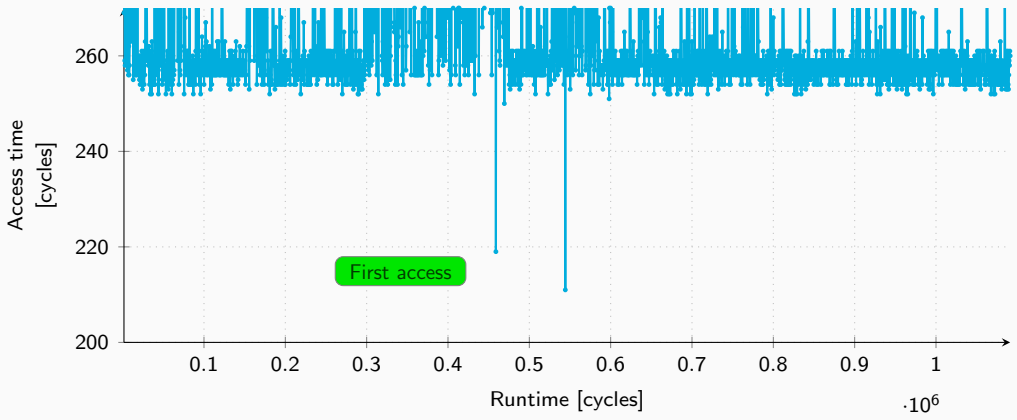
- Shared memory makes it a lot easier
- Constantly flush value from cache using `clflush` instruction
- Schedule victim
- Measure access time to value
- If access is fast, victim accessed (and cached) the data

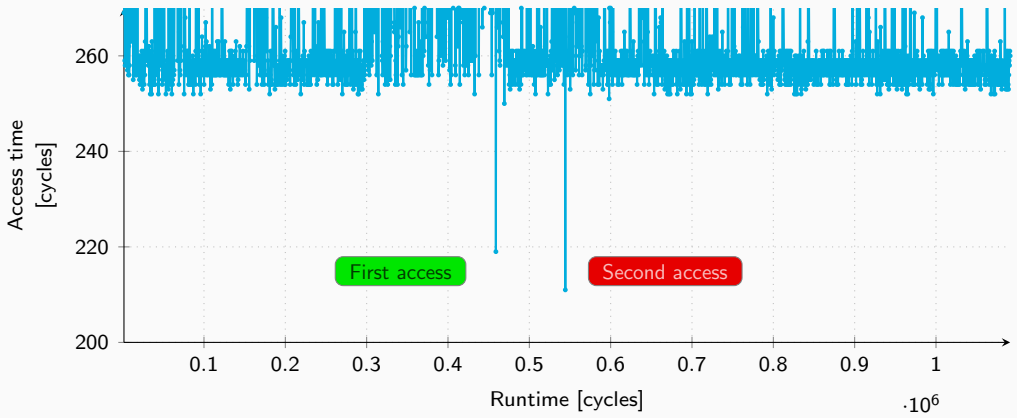

Access time [cycles]
260
240
220
200
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
1.1
Runtime [cycles]
·10^6

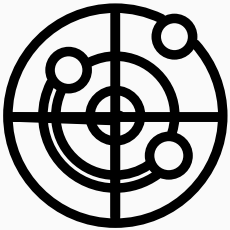

Access time [cycles]
260
240
220
200
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
1.1
Runtime [cycles]
·10^6
Data was accessed

Double-fetch Detection

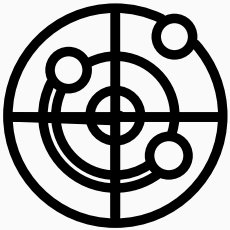



Access time [cycles]
260
240
220
200
First access
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
Runtime [cycles]
·10^6

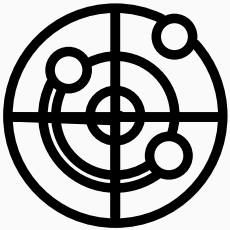

Access time [cycles]
260
240
220
200
First access
Second access
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
Runtime [cycles]
·10^6



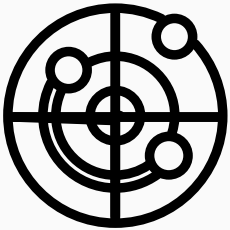
- Cache attacks dynamically detect double fetches in black boxes



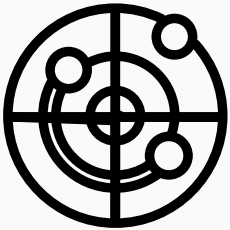
- Cache attacks dynamically detect double fetches in black boxes
- The further apart the two fetches, the higher the probability



- Cache attacks dynamically detect double fetches in black boxes
- The further apart the two fetches, the higher the probability
- Minimum time between accesses is approximately 600 cycles



- Cache attacks dynamically detect double fetches in black boxes
- The further apart the two fetches, the higher the probability
- Minimum time between accesses is approximately 600 cycles
- If the time is $\geq$ 3 000 cycles, detection rate is close to 100 %



- Cache attacks dynamically detect double fetches in black boxes
- The further apart the two fetches, the higher the probability
- Minimum time between accesses is approximately 600 cycles
- If the time is $\geq$ 3 000 cycles, detection rate is close to 100 %
- Allows automated detection of double fetches in many scenarios

- After detecting a double fetch, we want to exploit it

- After detecting a double fetch, we want to exploit it
- There is only a small time window to modify the value

- After detecting a double fetch, we want to exploit it
- There is only a small time window to modify the value
- State-of-the-art exploitation:
  - Flip the value as fast as possible between two values
  - At some point, we hit the time window
  - Probability is not very high

- After detecting a double fetch, we want to exploit it
- There is only a small time window to modify the value
- State-of-the-art exploitation:
  - Flip the value as fast as possible between two values
  - At some point, we hit the time window
  - Probability is not very high
- We need a trigger

Cache-based Trigger

- Use the cache side channel as trigger

- Use the cache side channel as trigger
- We already detect the memory accesses

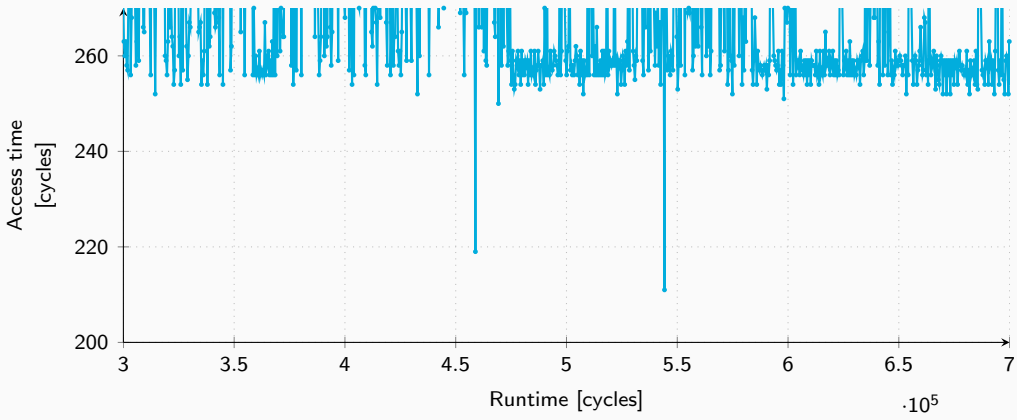
Cache-based Trigger

- Use the cache side channel as trigger
- We already detect the memory accesses
- The first memory access acts as our trigger

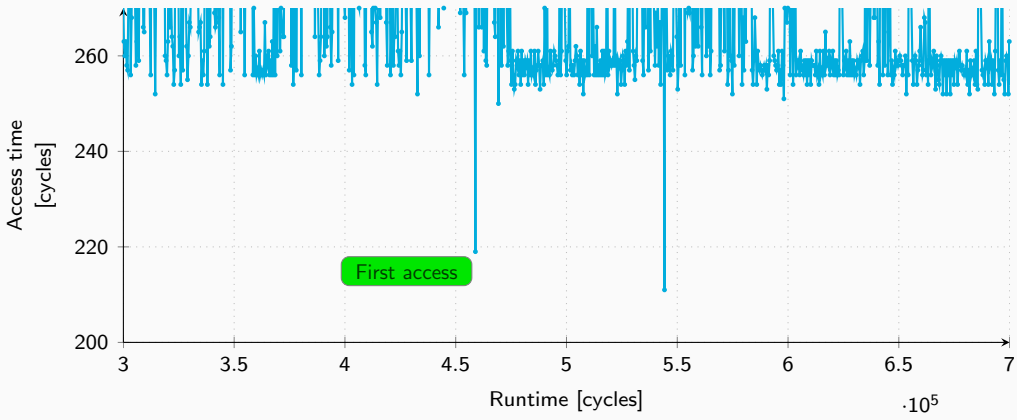
Cache-based Trigger

- Use the cache side channel as trigger
- We already detect the memory accesses
- The first memory access acts as our trigger
- Simply change the value immediately after the first access

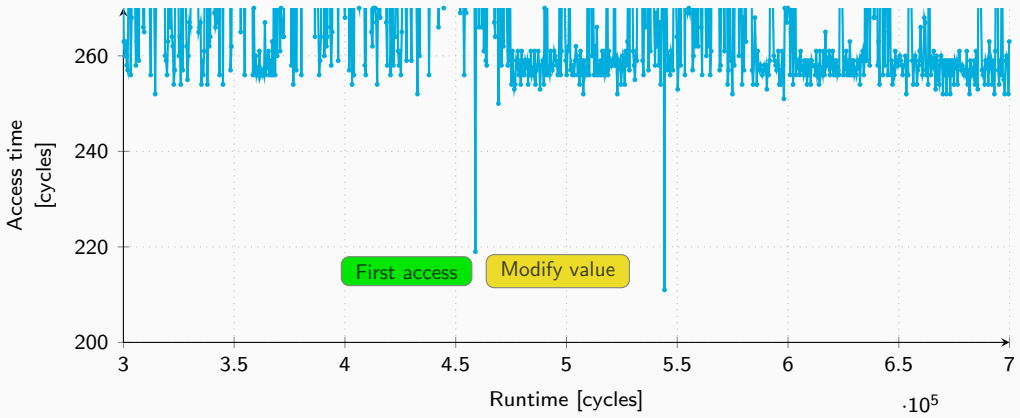
Cache-based Trigger



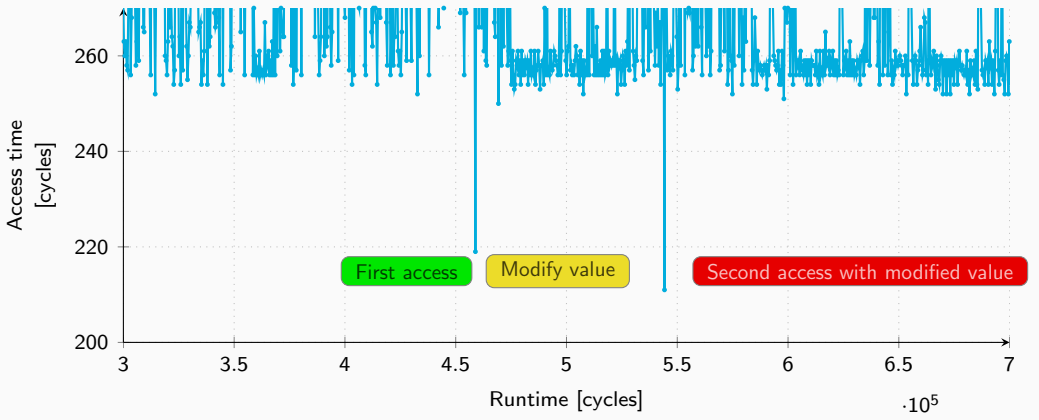
Cache-based Trigger



Cache-based Trigger



Cache-based Trigger




Live
DEMO

Preventing Exploitation of Double-fetch Bugs

- Double fetches are not bad by itself

- Double fetches are not bad by itself
- Sometimes they are necessary

- Double fetches are not bad by itself
- Sometimes they are necessary
- Only problematic if a modified value leads to an exploit

- Double fetches are not bad by itself
- Sometimes they are necessary
- Only problematic if a modified value leads to an exploit
- Even frameworks to exploit such race conditions [Wei+16]

- Double fetches are not bad by itself
- Sometimes they are necessary
- Only problematic if a modified value leads to an exploit
- Even frameworks to exploit such race conditions [Wei+16]
- With cache trigger very reliable [Sch+18]

- Idea: Ensure that both accesses are atomic...

- Idea: Ensure that both accesses are atomic...
- ...or at least look atomic from an attacker's perspective

- Idea: Ensure that both accesses are atomic...
- ...or at least look atomic from an attacker's perspective
- We have such a mechanism on modern Intel CPUs

- Idea: Ensure that both accesses are atomic...
- ...or at least look atomic from an attacker's perspective
- We have such a mechanism on modern Intel CPUs
- Intel TSX provides exactly this functionality in hardware

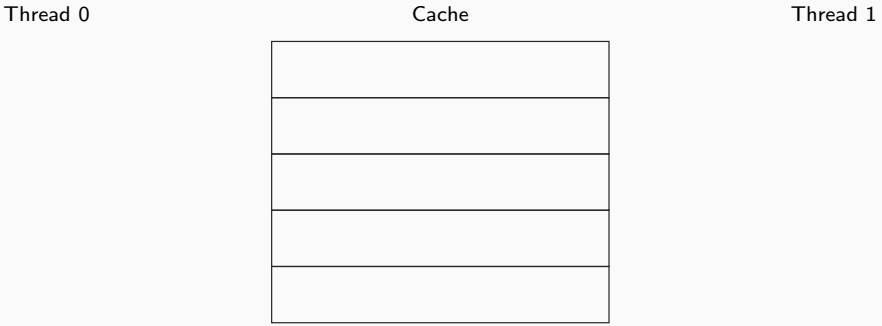
- Intel TSX is an implementation of hardware transactional memory

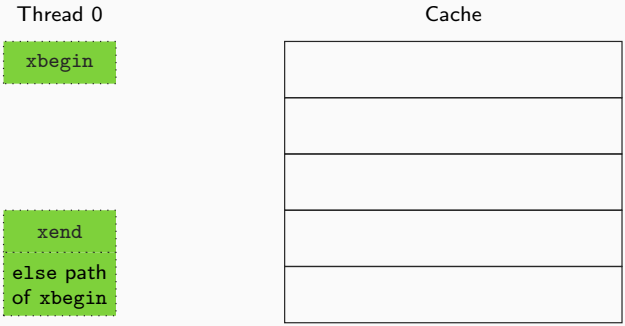
- Intel TSX is an implementation of hardware transactional memory
- Ensures that multiple reads and writes are atomic

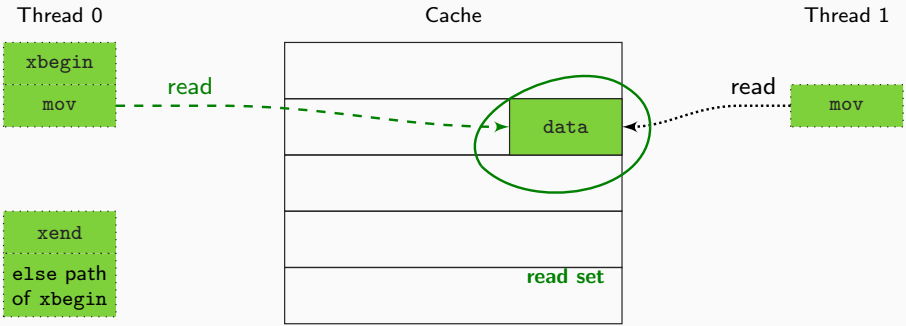
- Intel TSX is an implementation of hardware transactional memory
- Ensures that multiple reads and writes are atomic
- Operations are wrapped in a transaction

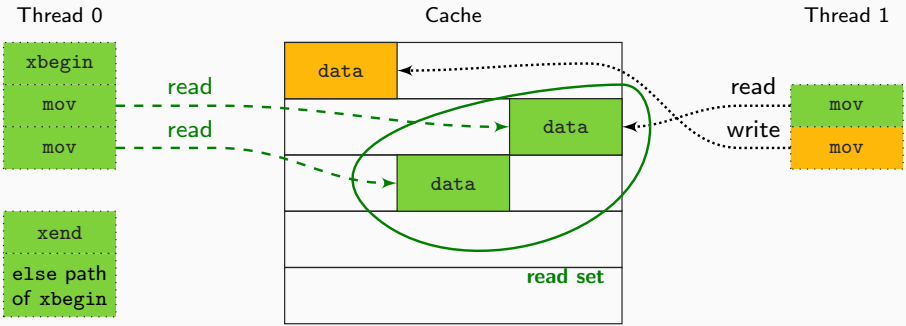
- Intel TSX is an implementation of hardware transactional memory
- Ensures that multiple reads and writes are atomic
- Operations are wrapped in a transaction
- If something conflicts, transaction is rolled back

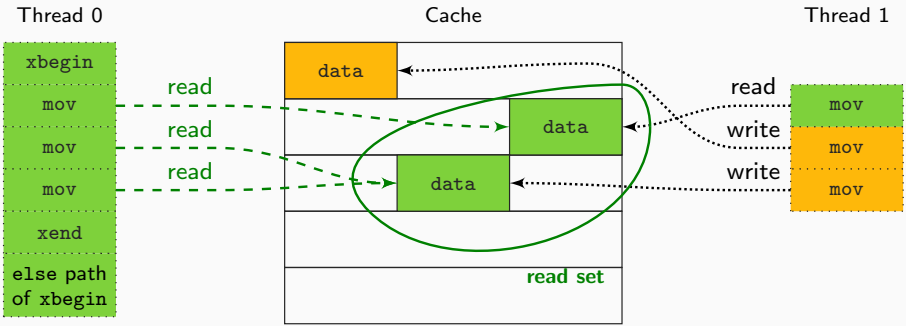
- Intel TSX is an implementation of hardware transactional memory
- Ensures that multiple reads and writes are atomic
- Operations are wrapped in a transaction
- If something conflicts, transaction is rolled back
- Implemented using the cache

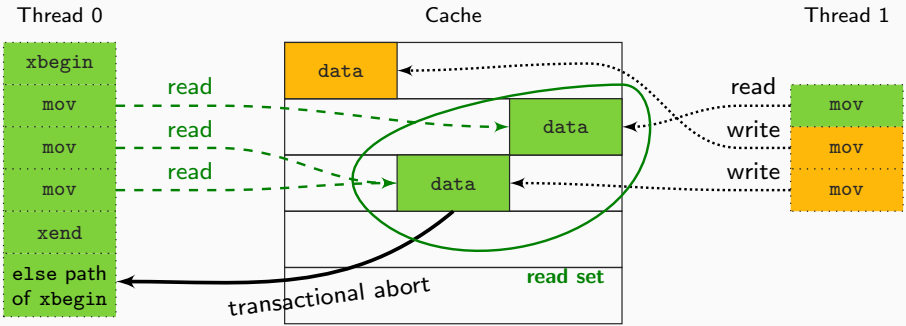

Thread 0
Cache
Thread 1

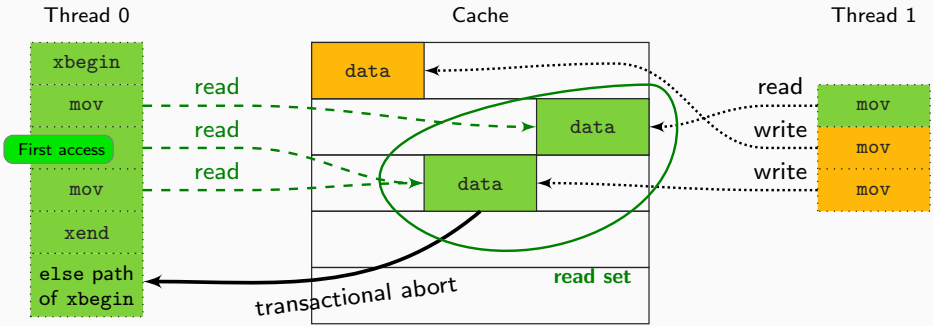

Thread 0
Cache
Thread 1
xbegin
xend
else path
of xbegin

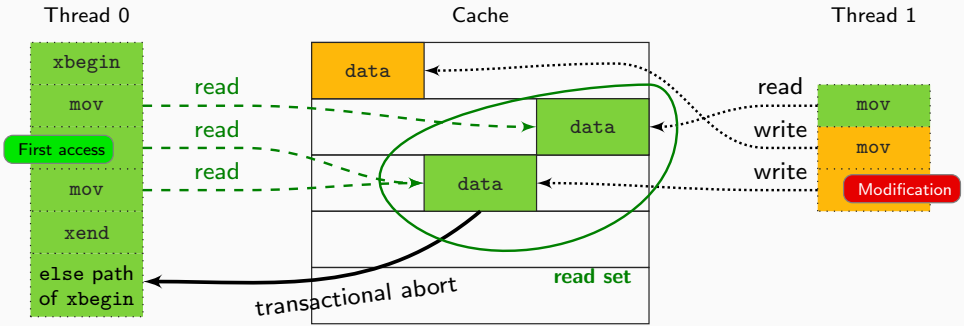

Thread 0
Cache
Thread 1
xbegin
mov
read
data
read
mov
xend
else path
of xbegin
read set

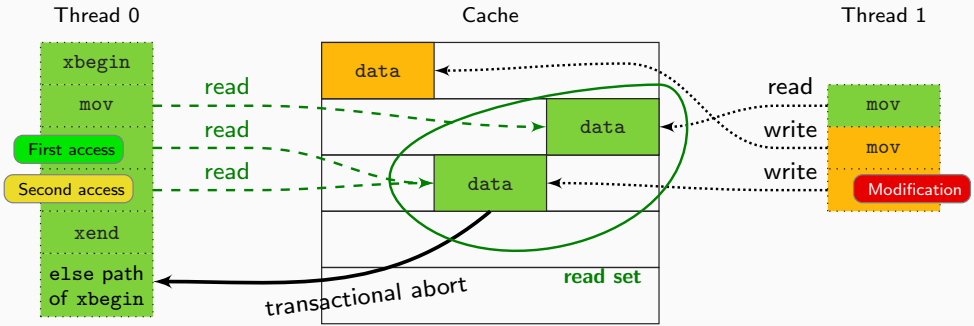

Thread 0
Cache
Thread 1
xbegin
mov
mov
xend
else path
of xbegin
read
read
data
data
data
read set
read
write
mov
mov

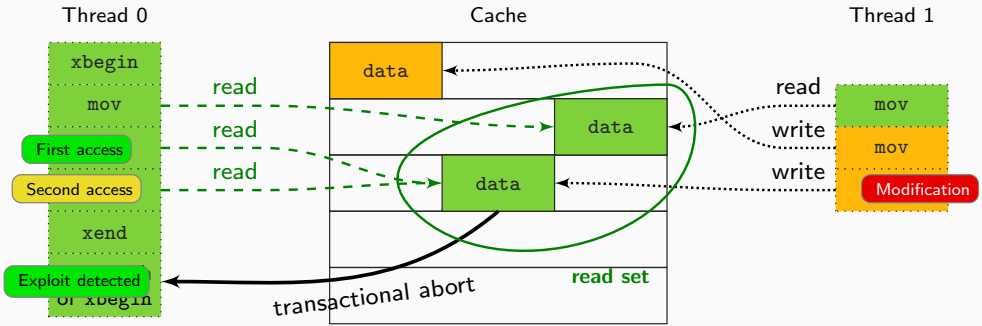

Thread 0
Cache
Thread 1
xbegin
mov
mov
mov
xend
else path of xbegin
read
read
read
data
data
data
read set
read
write
write
mov
mov
mov


Thread 0
Cache
Thread 1
xbegin
mov
mov
mov
xend
else path of xbegin
read
read
read
data
data
data
read
write
write
mov
mov
mov
transactional abort
read set


Thread 0
Cache
Thread 1
xbegin
mov
First access
mov
xend
else path
of xbegin
read
read
read
data
data
data
read
write
write
mov
mov
mov
transactional abort
read set


Thread 0
Cache
Thread 1
xbegin
mov
First access
mov
xend
else path
of xbegin
read
read
read
data
data
data
read
write
write
mov
mov
Modification
transactional abort
read set


Thread 0
Cache
Thread 1
xbegin
mov
First access
Second access
xend
else path
of xbegin
read
read
read
data
data
data
read
write
write
mov
mov
Modification
transactional abort
read set


Thread 0
Cache
Thread 1
xbegin
mov
First access
Second access
xend
Exploit detected
or xbegin
read
read
read
data
data
data
read set
transactional abort
read
write
write
mov
mov
Modification

- We created DropIt, a tiny library

- We created DropIt, a tiny library
- It prevents exploitation of double fetches using TSX

- We created DropIt, a tiny library
- It prevents exploitation of double fetches using TSX
- Open source: `https://github.com/IAIK/libdropit`

- We created DropIt, a tiny library
- It prevents exploitation of double fetches using TSX
- Open source: `https://github.com/IAIK/libdropit`
- Easy to use, 3 additional lines of code are sufficient

```
void play_file(DRMFile* file) {
  doublefetch_t config = doublefetch_init(10);
  doublefetch_start(config);
  if(check_signature(file->header) == ERR_INVALID) {
    // invalid signature
    return;
  }

  int now = get_timestamp_secure();

  if(now < file->header.from || now > file->header.to) {
    // not allowed to watch it now
    return;
  }
  doublefetch_end(config, { printf("Exploit detected!\n"); exit(-1);});

  void* data = decrypt(file->content);
  play_secure(data);
}
```

The Protected Code

```
void play_file(DRMFile* file) {
  doublefetch_t config = doublefetch_init(10);
  doublefetch_start(config);
  if(check_signature(file->header) == ERR_INVALID) {
    // invalid signature
    return;
  }

  int now = get_timestamp_secure();

  if(now < file->header.from || now > file->header.to) {
    // not allowed to watch it now
    return;
  }
  doublefetch_end(config, { printf("Exploit detected!\n"); exit(-1);});

  void* data = decrypt(file->content);
  play_secure(data);
}
```

Retry maximum 10 times

The Protected Code

Start protection

Retry maximum 10 times

```
void play_file(DRMFile* file) {
  doublefetch_t config = doublefetch_init(10);
  doublefetch_start(config);
  if(check_signature(file->header) == ERR_INVALID) {
    // invalid signature
    return;
  }

  int now = get_timestamp_secure();

  if(now < file->header.from || now > file->header.to) {
    // not allowed to watch it now
    return;
  }
  doublefetch_end(config, { printf("Exploit detected!\n"); exit(-1);});

  void* data = decrypt(file->content);
  play_secure(data);
}
```

The Protected Code

```
void play_file(DRMFile* file) {
  doublefetch_t config = doublefetch_init(10);
  doublefetch_start(config);
  if(check_signature(file->header) == ERR_INVALID) {
    // invalid signature
    return;
  }

  int now = get_timestamp_secure();

  if(now < file->header.from || now > file->header.to) {
    // not allowed to watch it now
    return;
  }
  doublefetch_end(config, { printf("Exploit detected!\n"); exit(-1);});

  void* data = decrypt(file->content);
  play_secure(data);
}
```

Retry maximum 10 times

Start protection

End protection

The Protected Code

```
void play_file(DRMFile* file) {
  doublefetch_t config = doublefetch_init(10);
  doublefetch_start(config);
  if(check_signature(file->header) == ERR_INVALID) {
    // invalid signature
    return;
  }

  int now = get_timestamp_secure();

  if(now < file->header.from || now > file->header.to) {
    // not allowed to watch it now
    return;
  }
  doublefetch_end(config, { printf("Exploit detected!\n"); exit(-1);});

  void* data = decrypt(file->content);
  play_secure(data);
}
```

Retry maximum 10 times

Start protection

Everything in between is atomic

End protection

- Efficient and easy solution to a complex problem

- Efficient and easy solution to a complex problem
- Faster than traditional locking ($\approx 18\,\%$)

- Efficient and easy solution to a complex problem
- Faster than traditional locking ($\approx 18\,\%$)
- Unfortunately limited to new Intel CPUs

- Efficient and easy solution to a complex problem
- Faster than traditional locking ($\approx 18\,\%$)
- Unfortunately limited to new Intel CPUs
- Many CPUs with SGX also have TSX

Conclusion

Black Hat Sound Bytes.

- Side channels can cost you money

Black Hat Sound Bytes.

- Side channels can cost you money
- Do not consider side channels out-of-scope

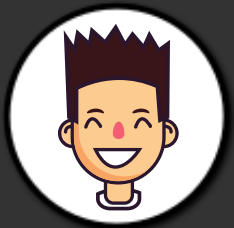
Black Hat Sound Bytes.

- Side channels can cost you money
- Do not consider side channels out-of-scope
- Exploitable code + SGX = exploitable SGX enclave

Thank you!

When Good Turns Evil

Using Intel SGX to Stealthily Steal Bitcoins



Michael Schwarz
@misc0110

Moritz Lipp
@mlqxyz


F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In: S&P. 2015.


C. Maurice, N. Le Scouarnec, C. Neumann, O. Heen, and A. Francillon. Reverse Engineering Intel Complex Addressing Using Performance Counters. In: RAID. 2015.


C. Maurice, M. Weber, M. Schwarz, L. Giner, D. Gruss, C. A. Boano, S. Mangard, and K. Römer. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In: NDSS. 2017.


D. A. Osvik, A. Shamir, and E. Tromer. Cache Attacks and Countermeasures: the Case of AES. In: CT-RSA. 2006.


P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In: USENIX Security Symposium. 2016.


M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In: DIMVA. 2017.


M. Schwarz, D. Gruss, M. Lipp, C. Maurice, T. Schuster, A. Fogh, and S. Mangard. Automated Detection, Exploitation, and Elimination of Double-Fetch Bugs using Modern CPU Features. In: AsiaCCS'18. 2018.


N. Weichbrodt, A. Kurmus, P. Pietzuch, and R. Kapitza. AsyncShock: Exploiting synchronisation bugs in Intel SGX enclaves. In: ESORICS'16. 2016.