

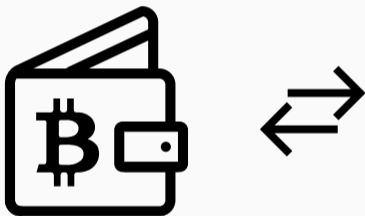
Cash Attacks on SGX

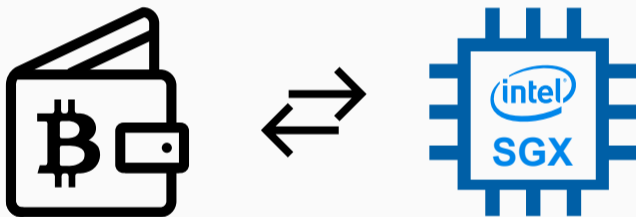
Daniel Gruss, Michael Schwarz

September 9, 2017

Graz University of Technology

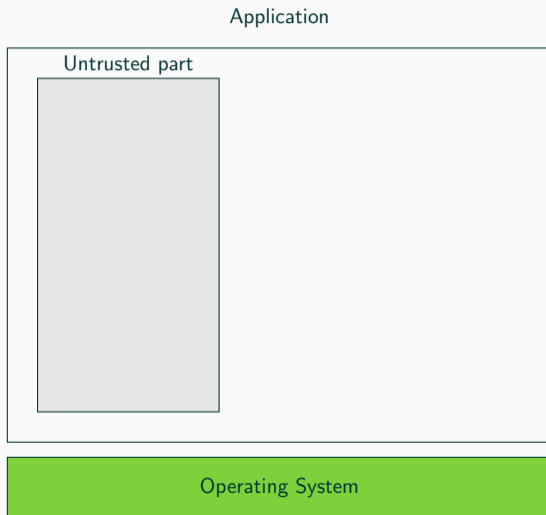


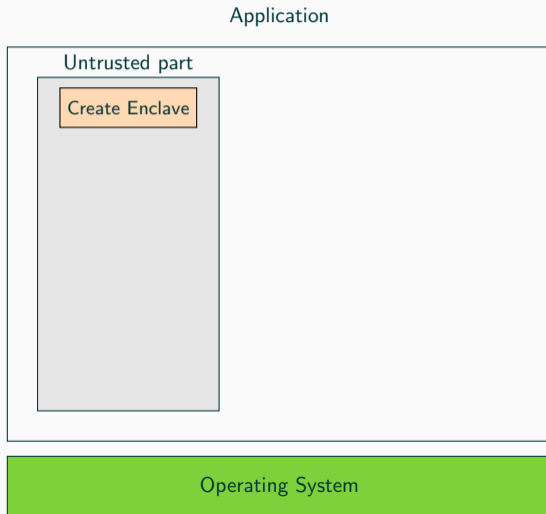


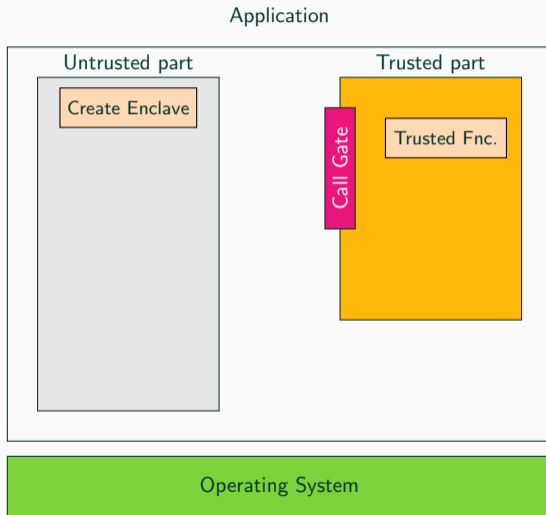


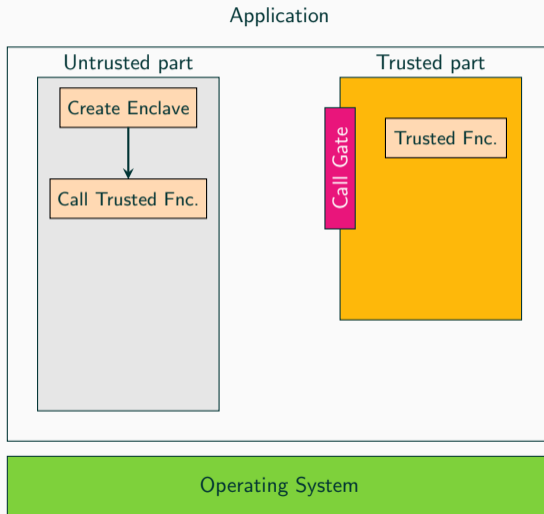


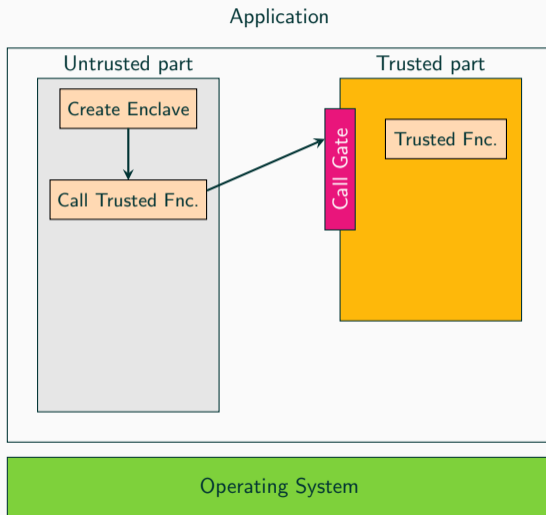


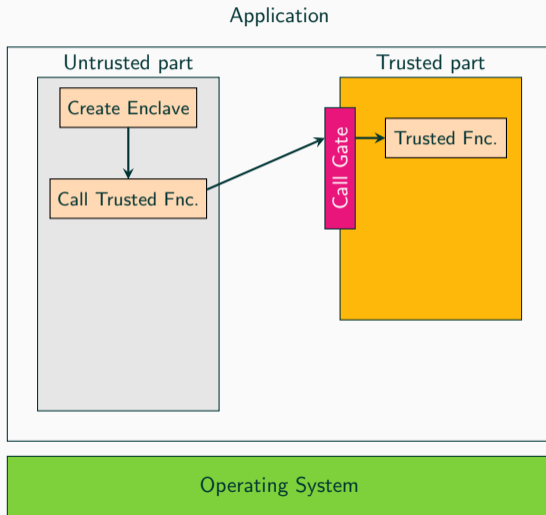


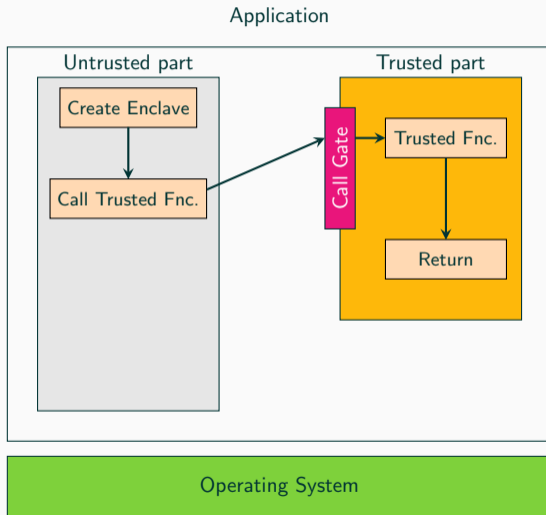


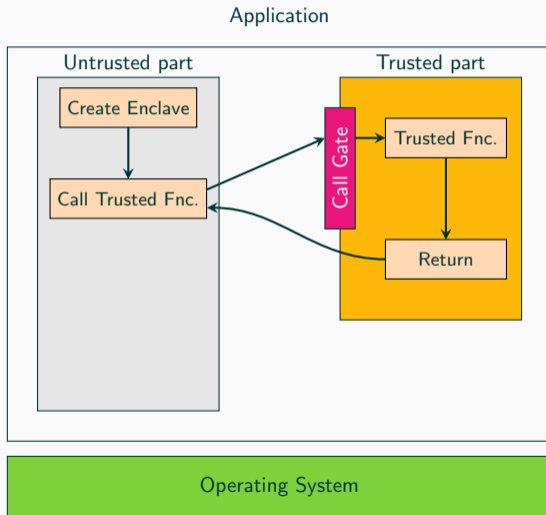


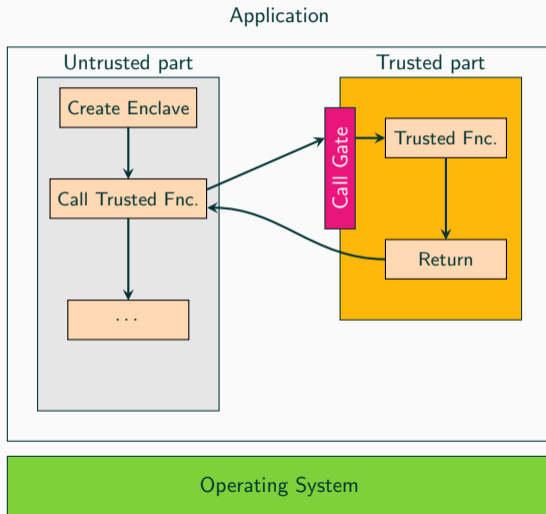


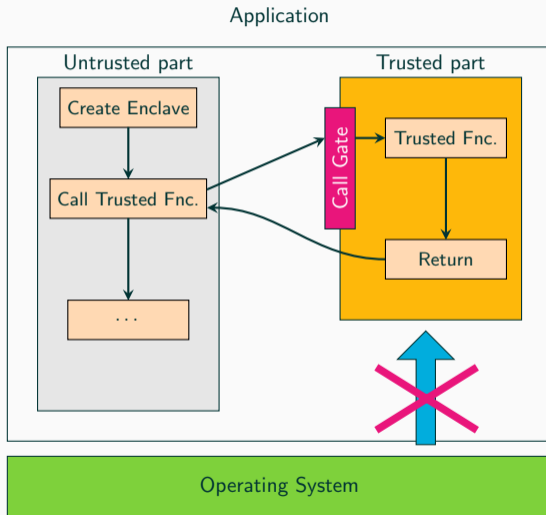














- **Ledger SGX Enclave** for blockchain applications
- **BitPay Copay** Bitcoin wallet
- **Teechain** payment channel using SGX

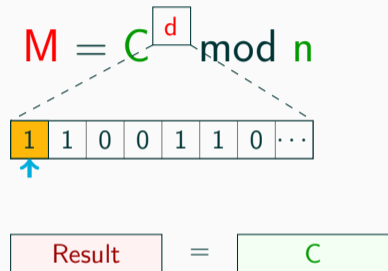


- **Ledger SGX Enclave** for blockchain applications
- **BitPay Copay** Bitcoin wallet
- **Teechain** payment channel using SGX

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. [...]

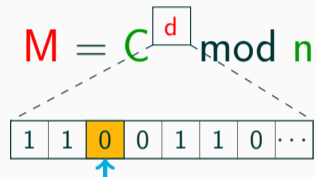
$$M = C^d \text{ mod } n$$



$$M = C^d \pmod n$$

1 1 0 0 1 1 0 ...

$$\text{Result} = \underbrace{\text{Result} \times \text{Result}}_{\text{square}} \times \underbrace{C}_{\text{multiply}}$$

$$M = C^d \pmod n$$


$$\text{Result} = \underbrace{\text{Result} \times \text{Result}}_{\text{square}}$$

$$M = C^d \pmod n$$

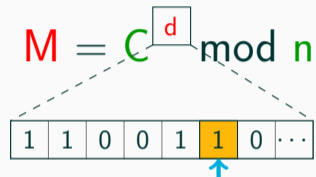
1 1 0 0 1 1 0 ...

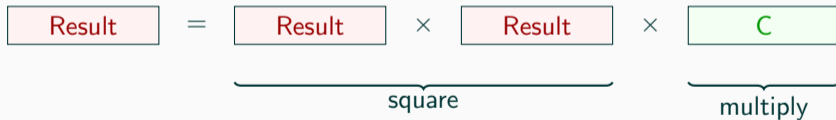
$$\text{Result} = \underbrace{\text{Result} \times \text{Result}}_{\text{square}}$$

$$M = C^d \pmod n$$

1 1 0 0 1 1 0 ...

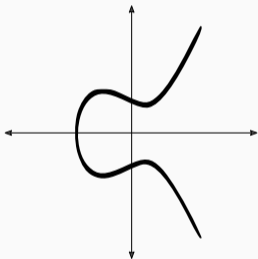
$$\text{Result} = \underbrace{\text{Result} \times \text{Result}}_{\text{square}} \times \underbrace{C}_{\text{multiply}}$$

$$M = C^d \pmod n$$


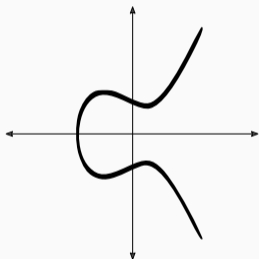
$$\text{Result} = \underbrace{\text{Result} \times \text{Result}}_{\text{square}} \times \underbrace{C}_{\text{multiply}}$$


$$M = C^d \pmod n$$

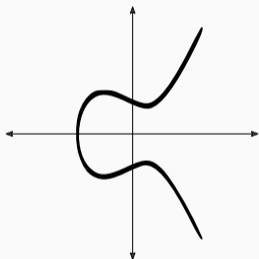
$$\text{Result} = \underbrace{\text{Result} \times \text{Result}}_{\text{square}}$$



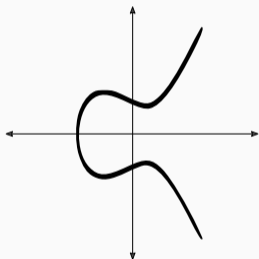
- Used to sign transactions



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



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



- Used to sign transactions
- Point multiplication is similar to RSA exponentiation
- Simplest implementation **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]...

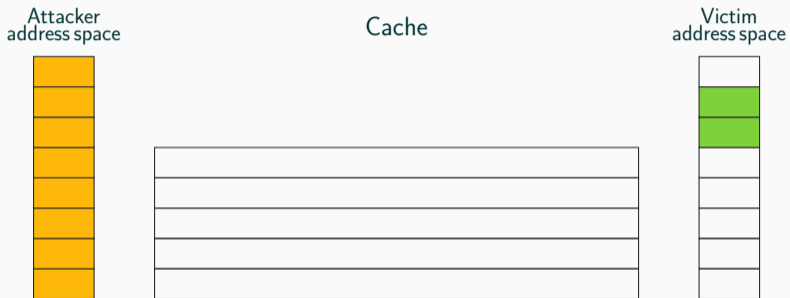
- exploits the **timing difference** when accessing...
 - cached data (fast)
 - uncached data (slow)
- is used to attack **secret-dependent** memory accesses

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

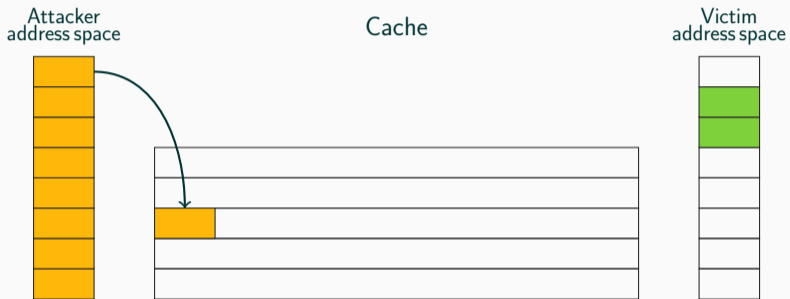
- exploits the **timing difference** when accessing...
 - cached data (fast)
 - uncached data (slow)
- is used to attack **secret-dependent** memory accesses
- is applied to a part of the CPU cache, a cache set

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

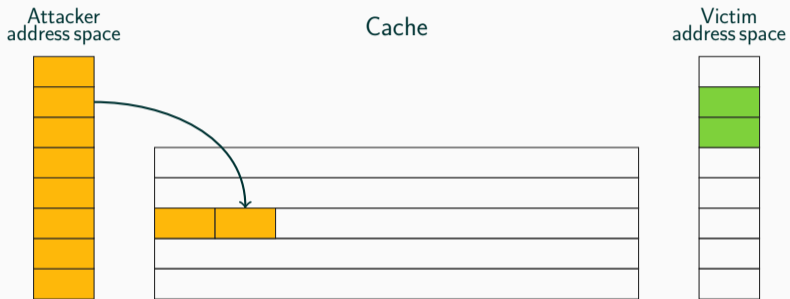
- exploits the **timing difference** when accessing...
 - cached data (fast)
 - uncached data (slow)
- is used to attack **secret-dependent** memory accesses
- is applied to a part of the CPU cache, a cache set
- works **across CPU cores** as the last-level cache is shared



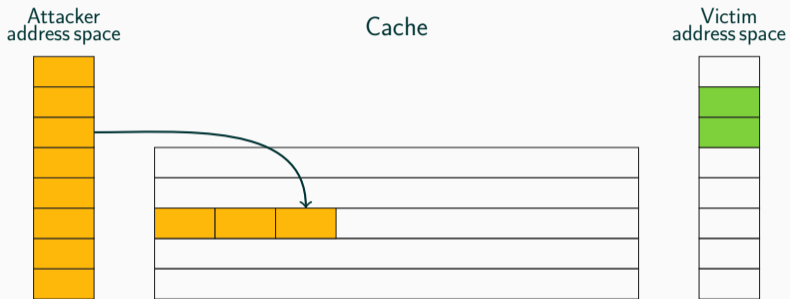
Step 0: Attacker fills the cache (prime)



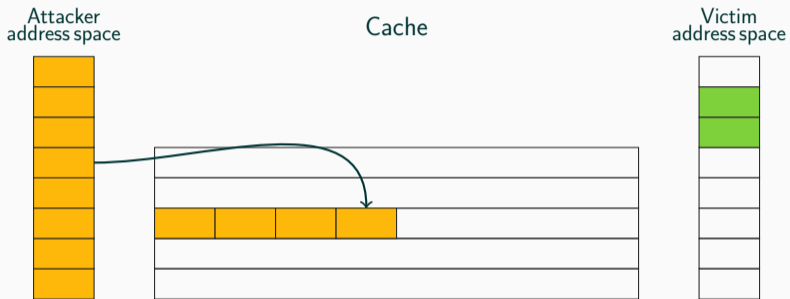
Step 0: Attacker fills the cache (prime)



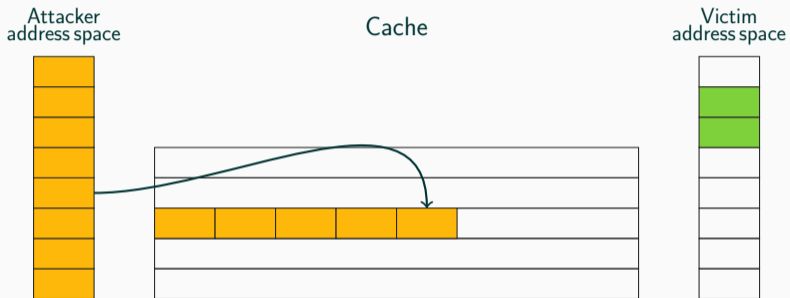
Step 0: Attacker fills the cache (prime)



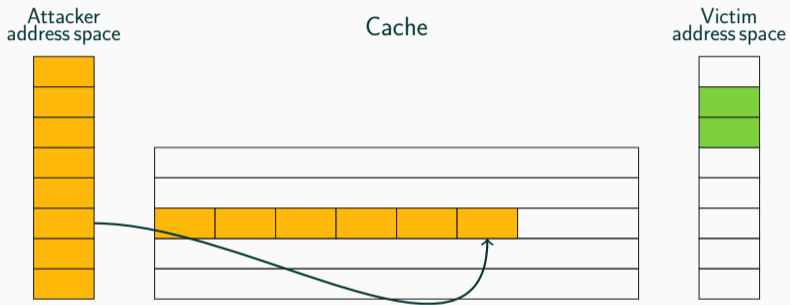
Step 0: Attacker fills the cache (prime)



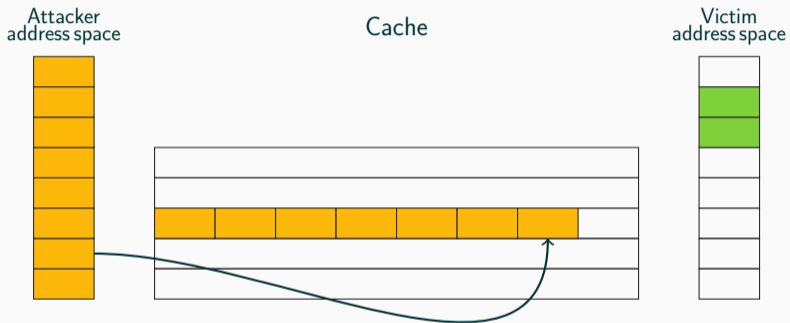
Step 0: Attacker fills the cache (prime)



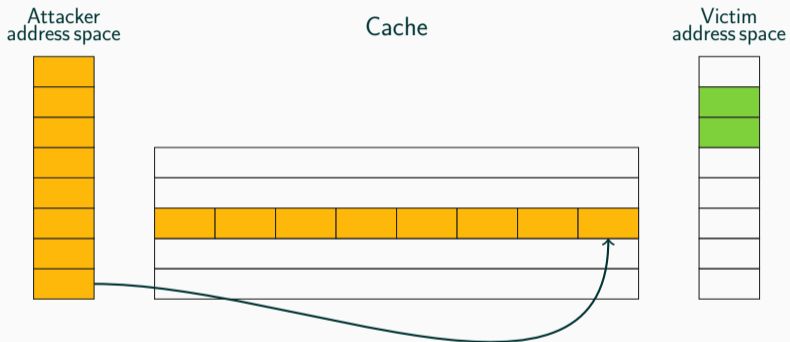
Step 0: Attacker fills the cache (prime)



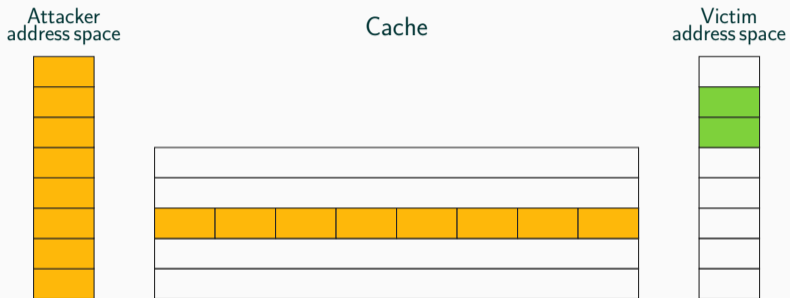
Step 0: Attacker fills the cache (prime)



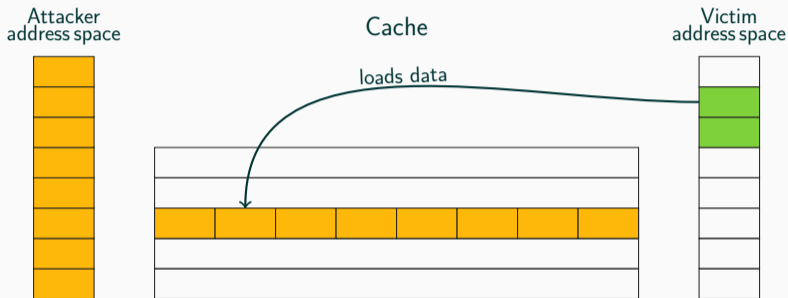
Step 0: Attacker fills the cache (prime)



Step 0: Attacker fills the cache (prime)

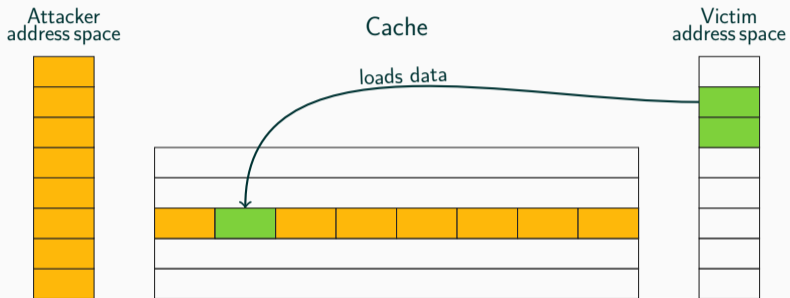


Step 0: Attacker fills the cache (prime)



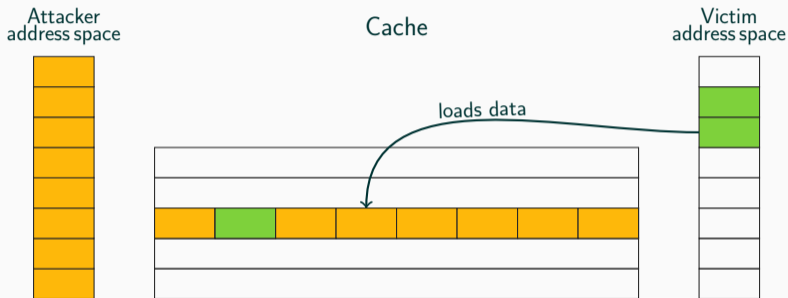
Step 0: Attacker fills the cache (prime)

Step 1: Victim evicts cache lines by accessing own data



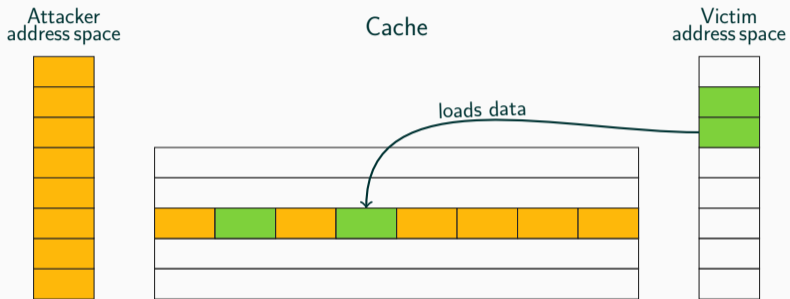
Step 0: Attacker fills the cache (prime)

Step 1: Victim evicts cache lines by accessing own data



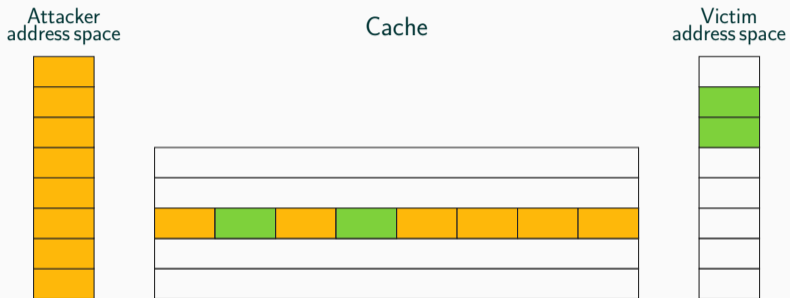
Step 0: Attacker fills the cache (prime)

Step 1: Victim evicts cache lines by accessing own data



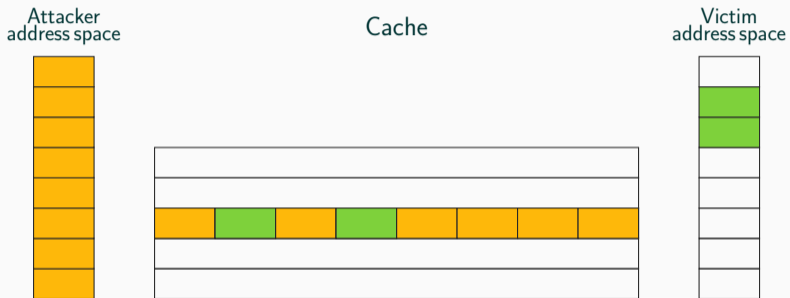
Step 0: Attacker fills the cache (prime)

Step 1: Victim evicts cache lines by accessing own data



Step 0: Attacker fills the cache (prime)

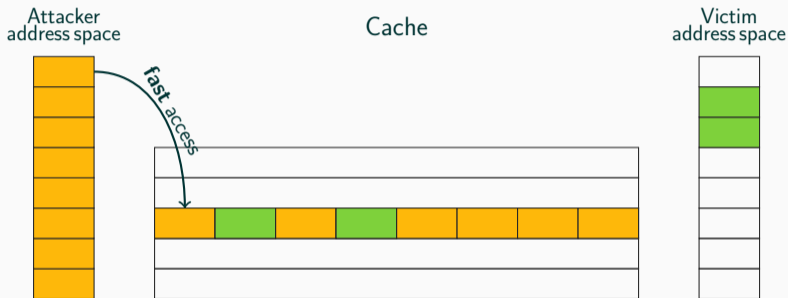
Step 1: Victim evicts cache lines by accessing own data



Step 0: Attacker fills the cache (prime)

Step 1: Victim evicts cache lines by accessing own data

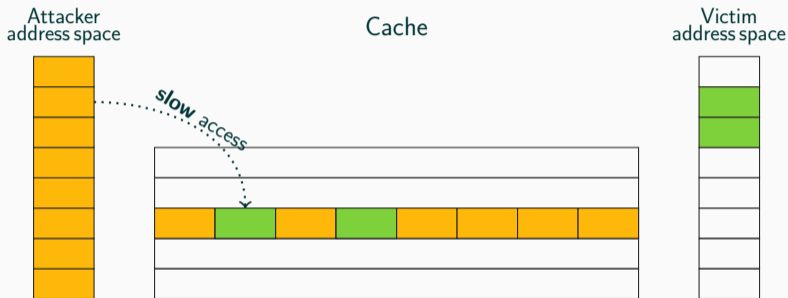
Step 2: Attacker probes data to determine if the set was accessed



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



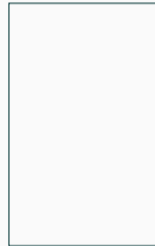
Step 0: Attacker fills the cache (prime)

Step 1: Victim evicts cache lines by accessing own data

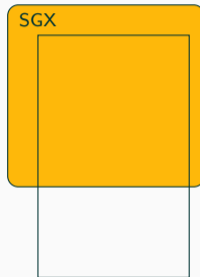
Step 2: Attacker probes data to determine if the set was accessed

Attack

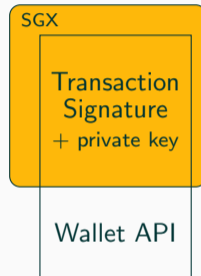
Victim

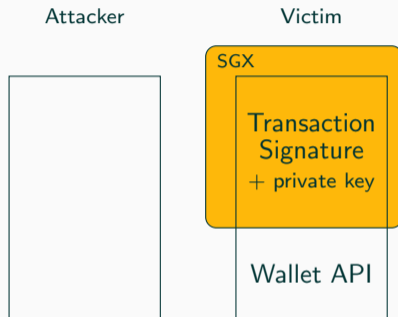


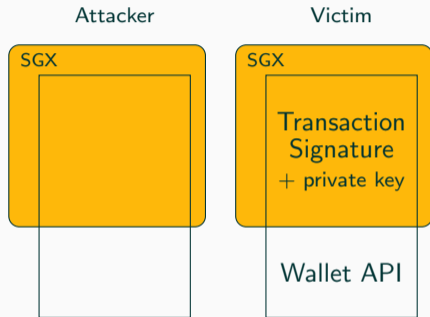
Victim

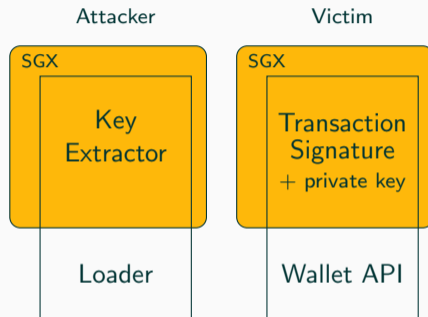


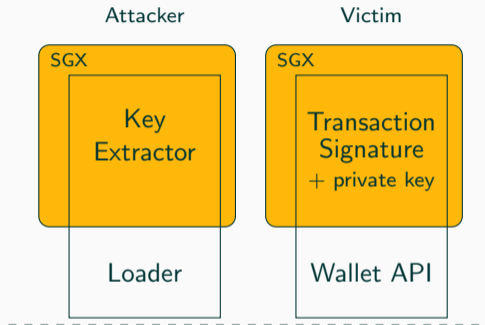
Victim

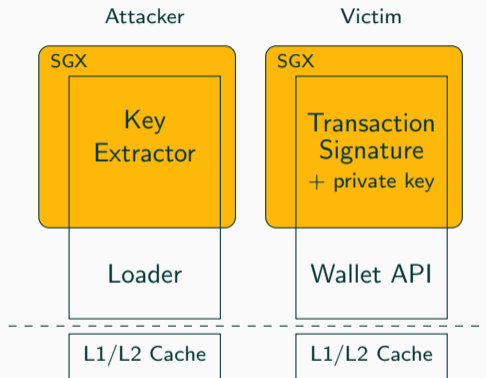


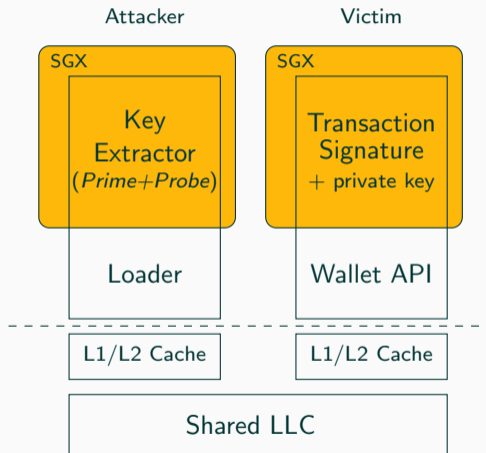












Classical Prime+Probe cannot be mounted within SGX:





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**



- 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**
- This is even **15 % faster** than the native timestamp counter

```
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**



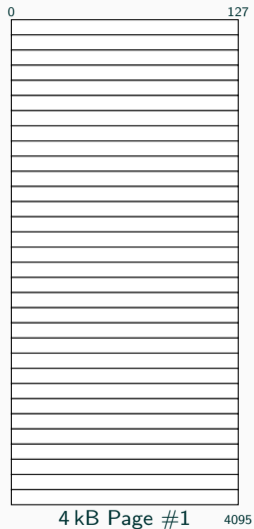
- **Cache set** is determined by part of physical address [Mau+15]
- We have no knowledge of **physical addresses**
- Use the reverse-engineered **DRAM mapping** [Pes+16]



- **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



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



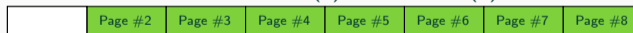
8 kB row x in BG0 (0) and channel (1)

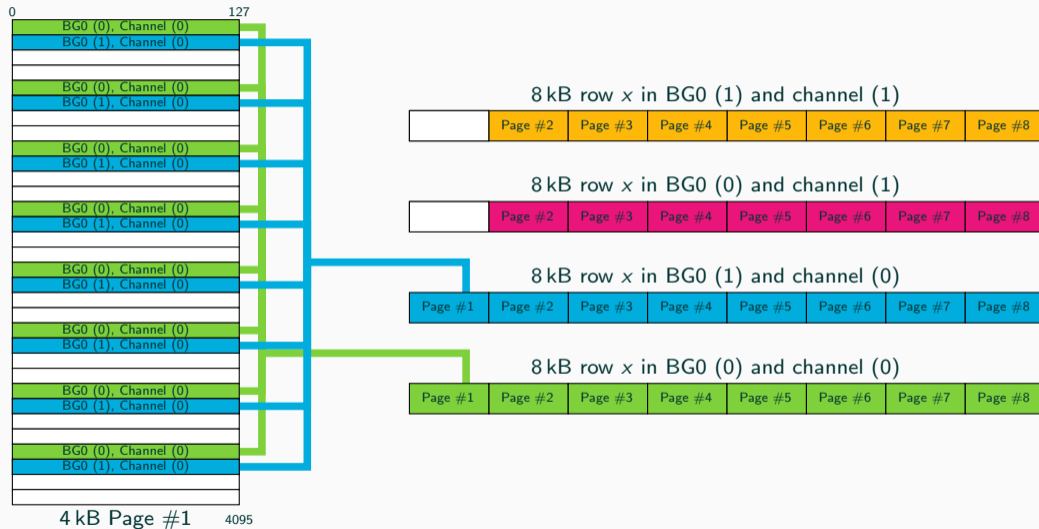


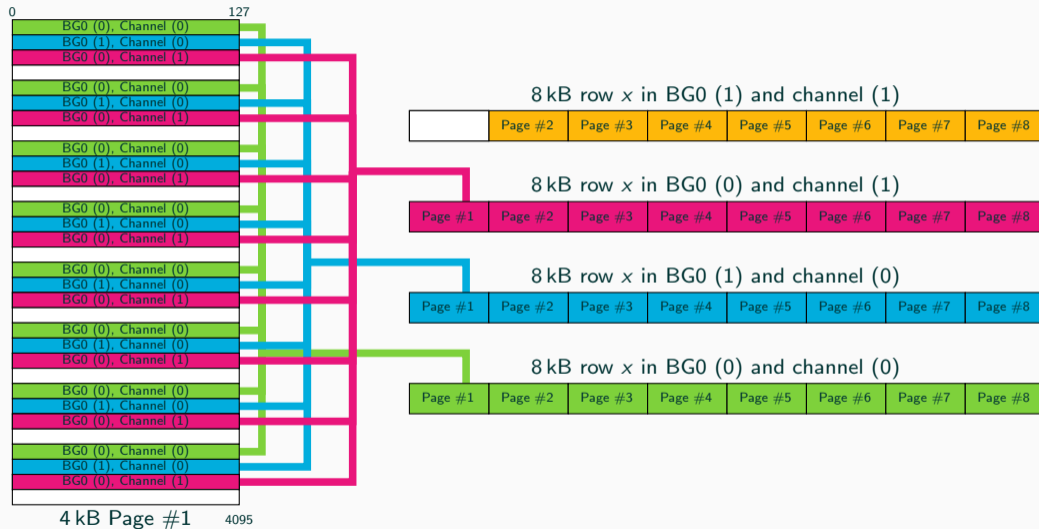
8 kB row x in BG0 (1) and channel (0)

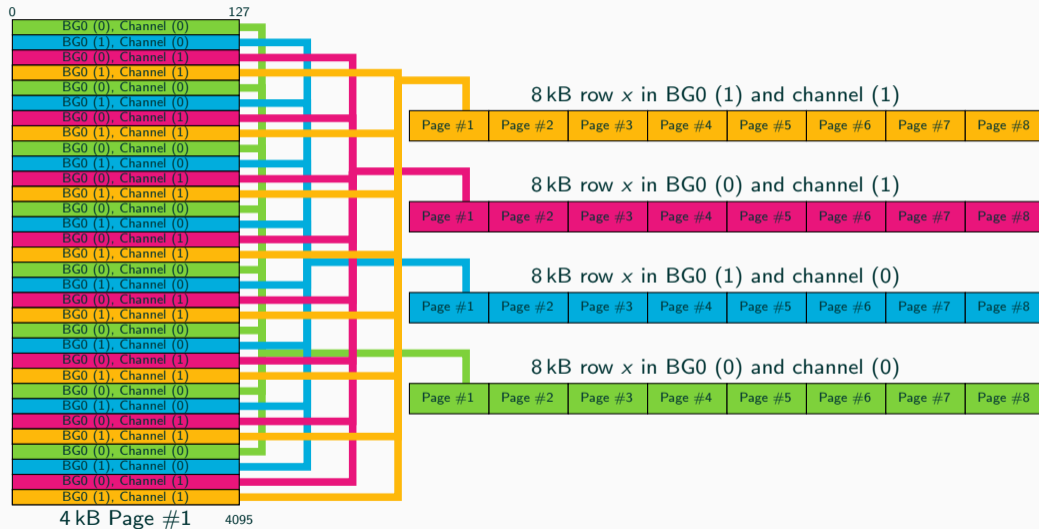


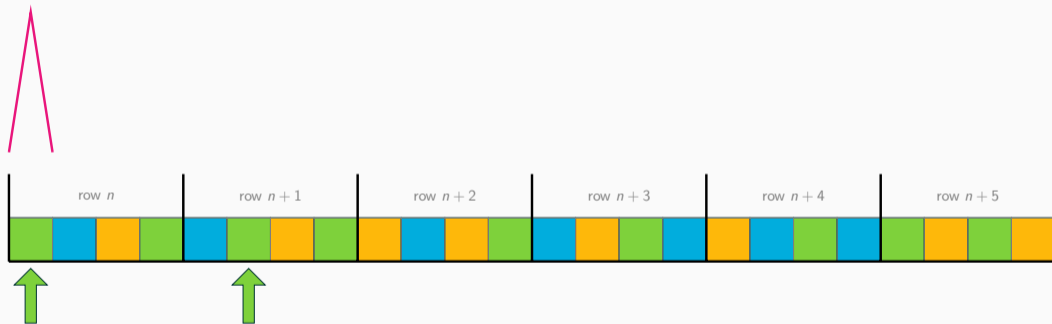
8 kB row x in BG0 (0) and channel (0)

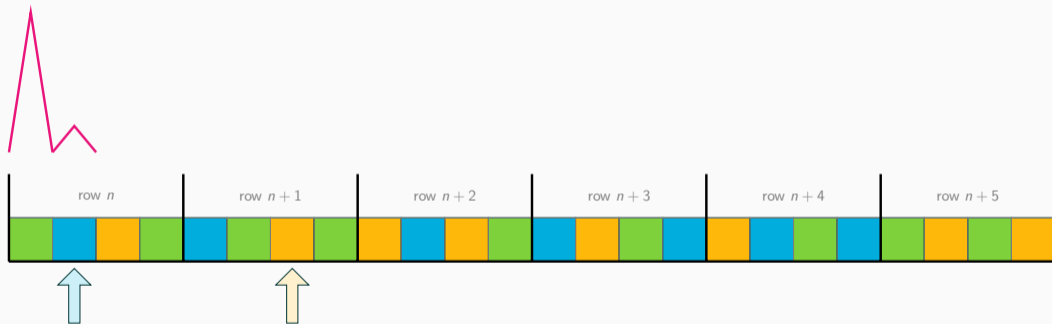


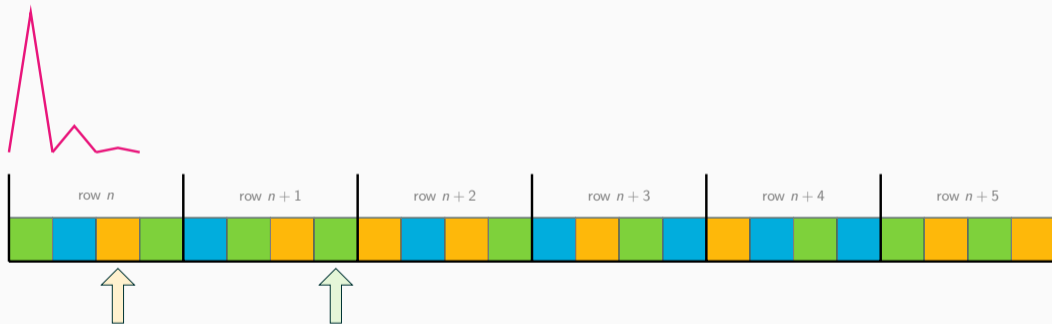


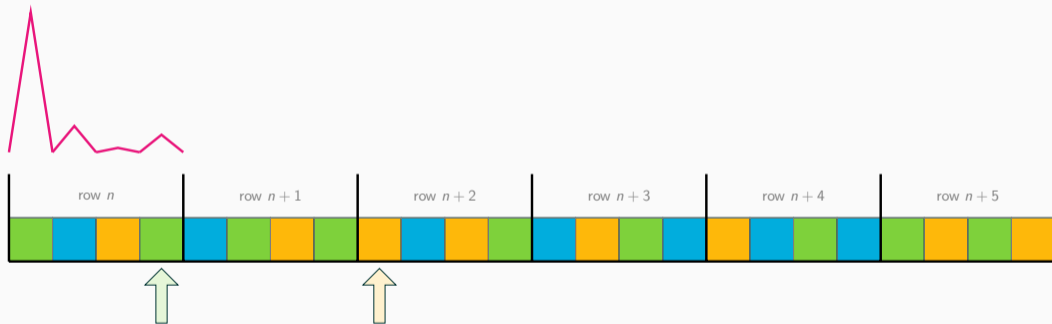


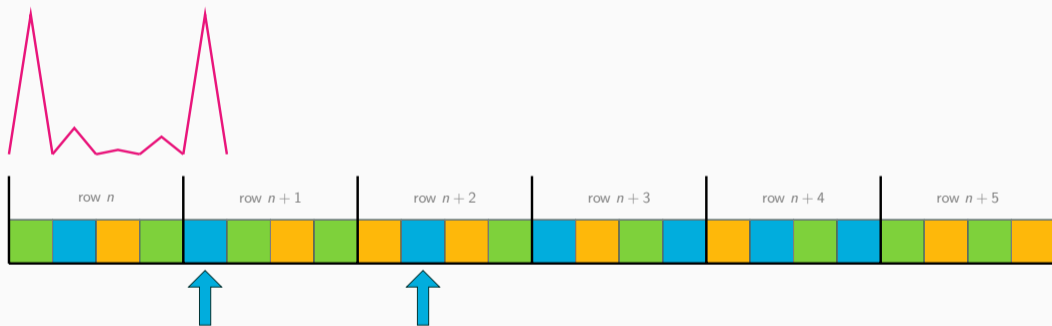


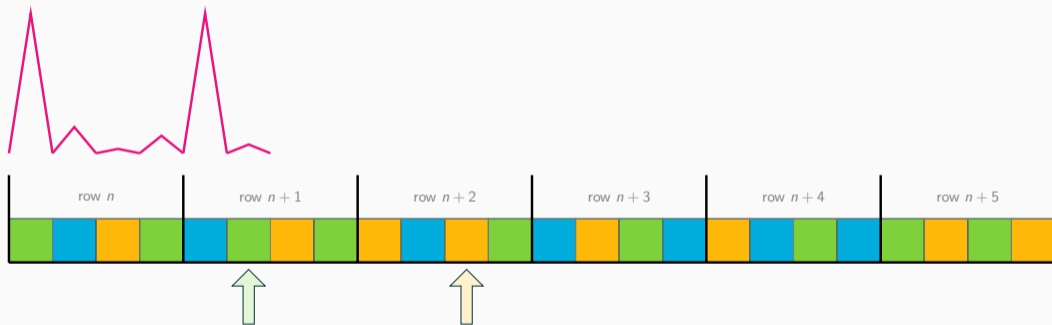


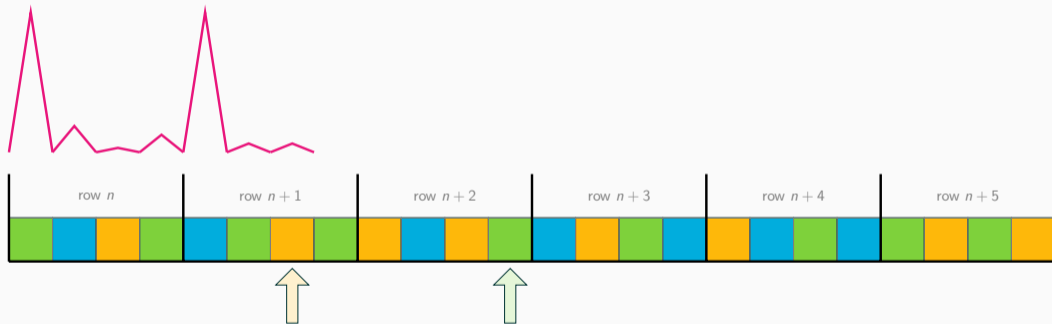


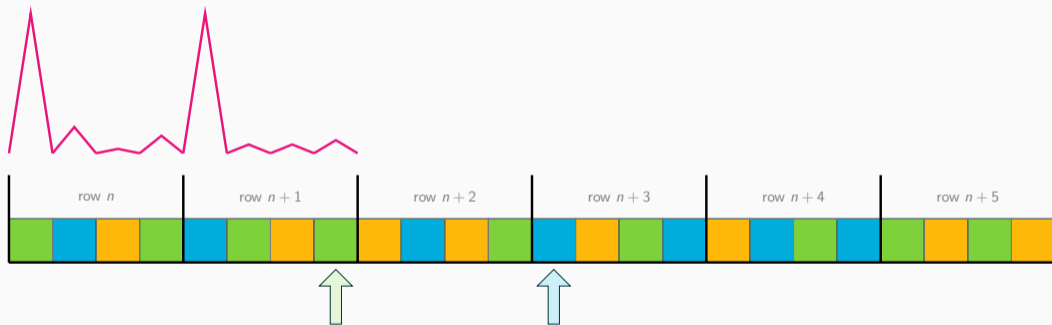


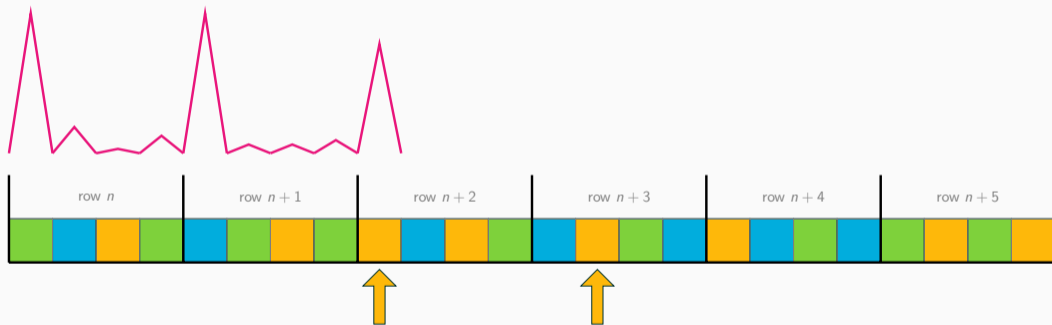


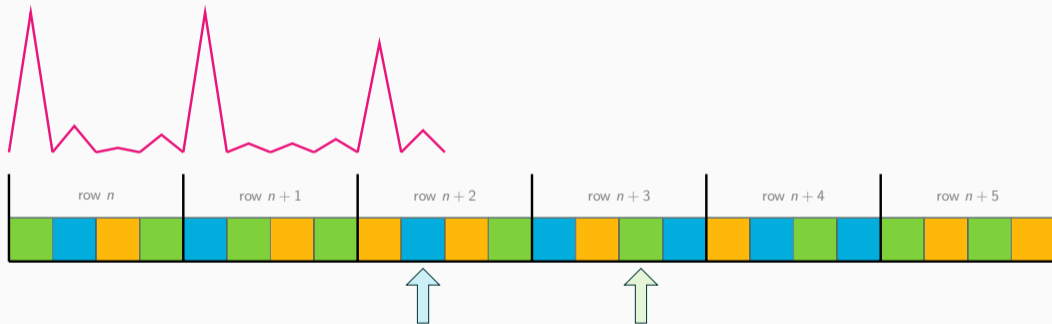


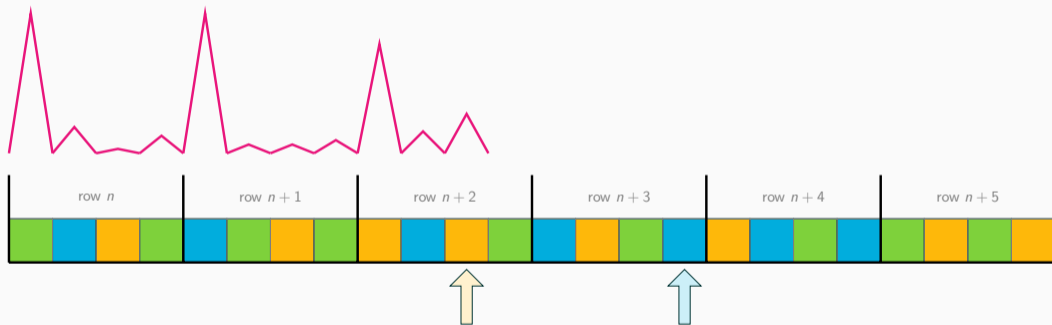


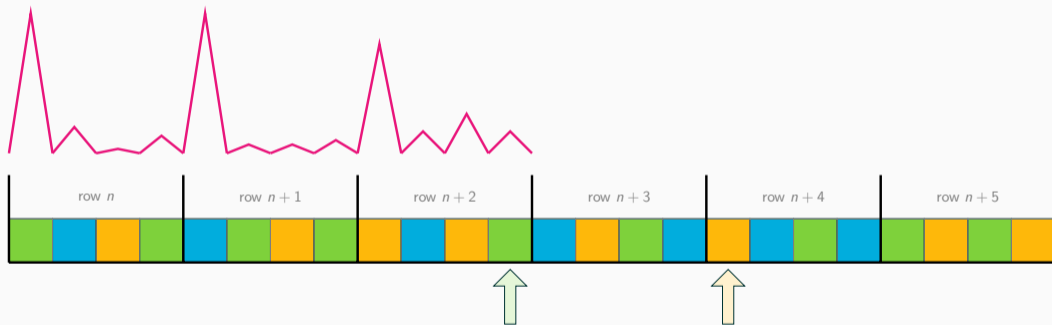




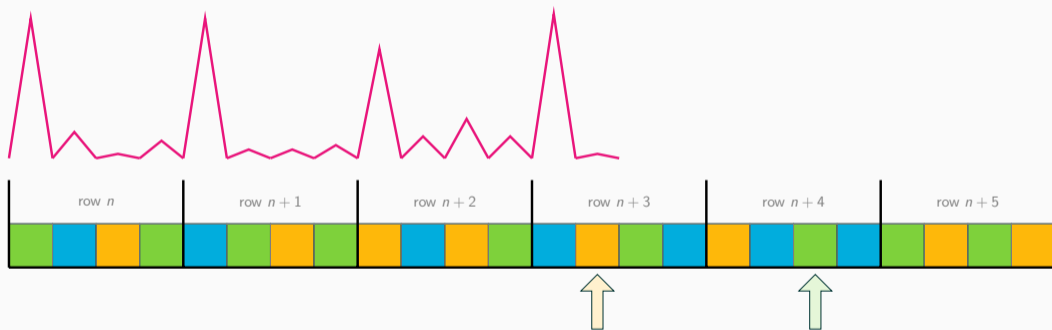






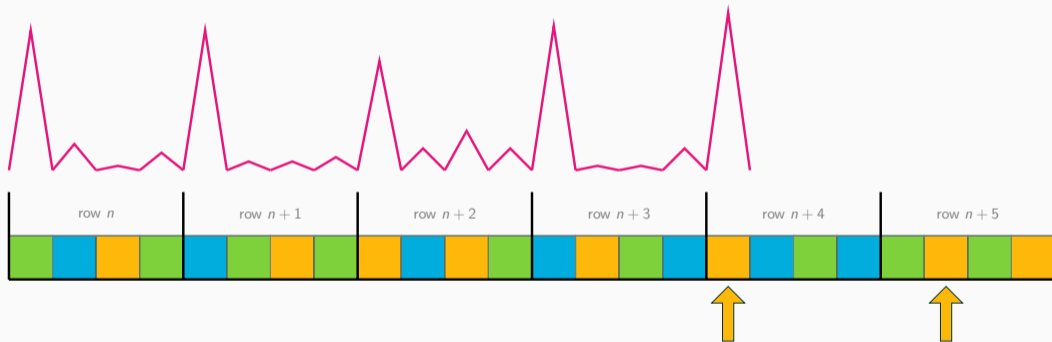


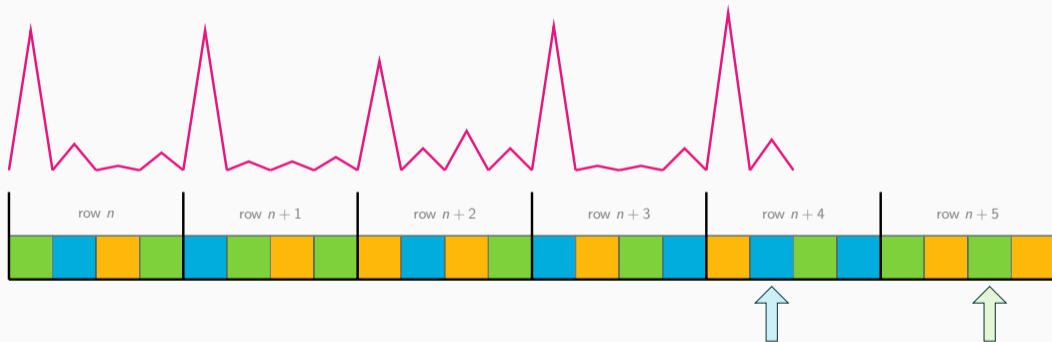


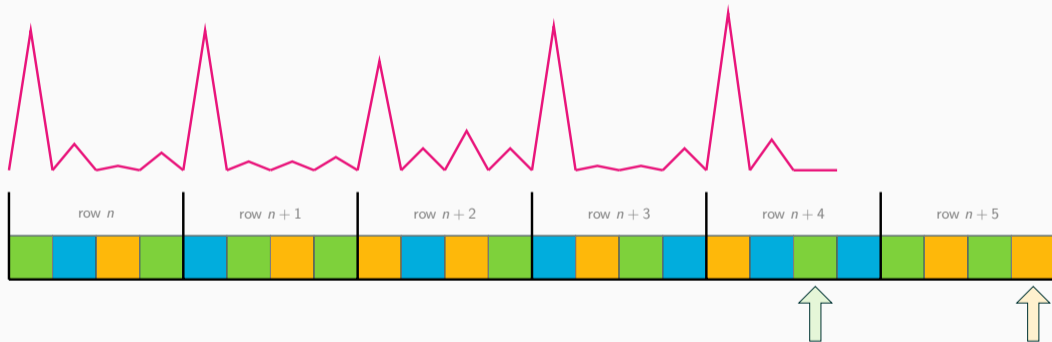




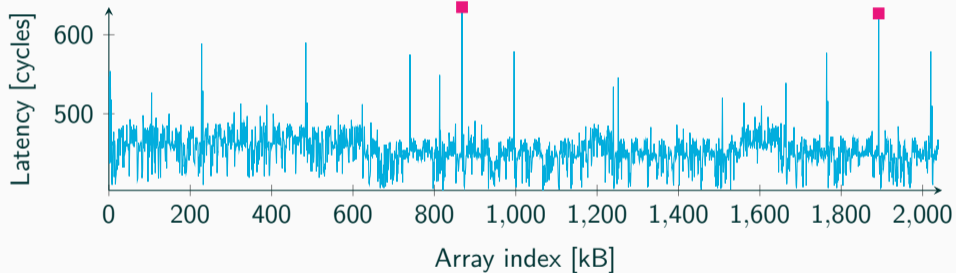


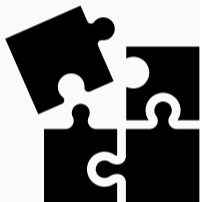




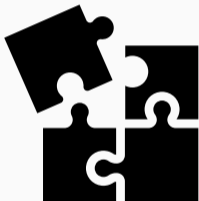


Result on an Intel i5-6200U

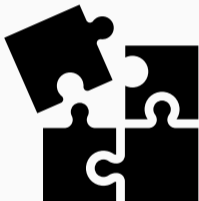




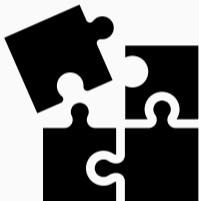
1. Use the **counting primitive** to measure DRAM accesses



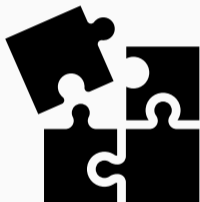
1. Use the **counting primitive** to measure DRAM accesses
2. Through the DRAM side channel, determine the **row borders**



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' → maps to **cache set '0'**



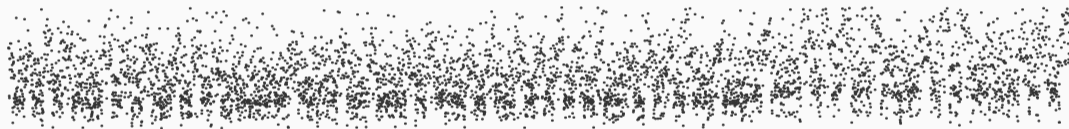
1. Use the **counting primitive** to measure DRAM accesses
2. Through the DRAM side channel, determine the **row borders**
3. Row borders have the 18 LSBs set to '0' → maps to **cache set '0'**
4. Build the **eviction set** for the Prime+Probe attack



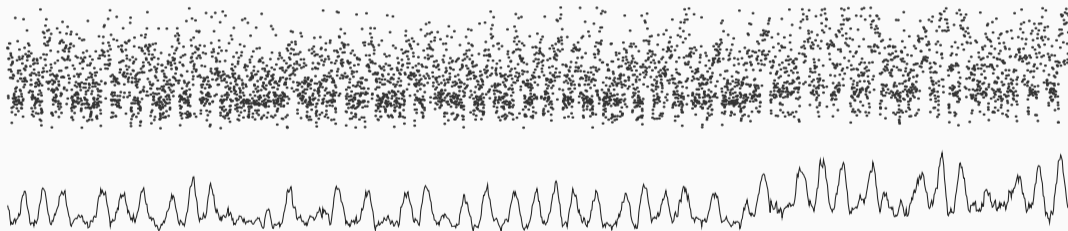
1. Use the **counting primitive** to measure DRAM accesses
2. Through the DRAM side channel, determine the **row borders**
3. Row borders have the 18 LSBs set to '0' → maps to **cache set** '0'
4. Build the **eviction set** for the Prime+Probe attack
5. Mount **Prime+Probe** on the buffer containing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

Results

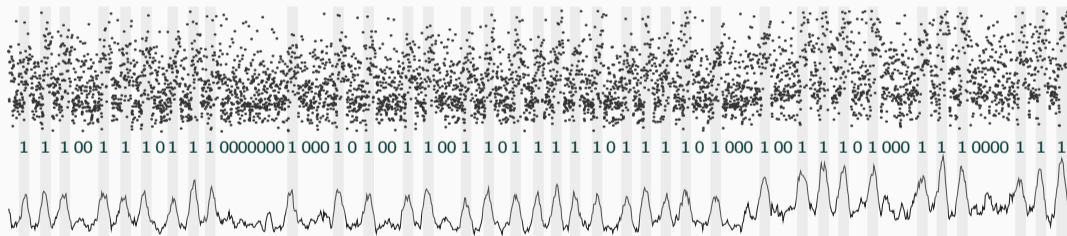
Raw Prime+Probe trace...

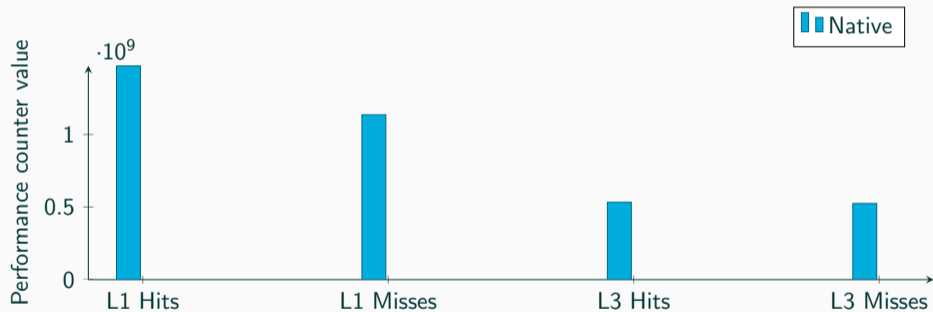


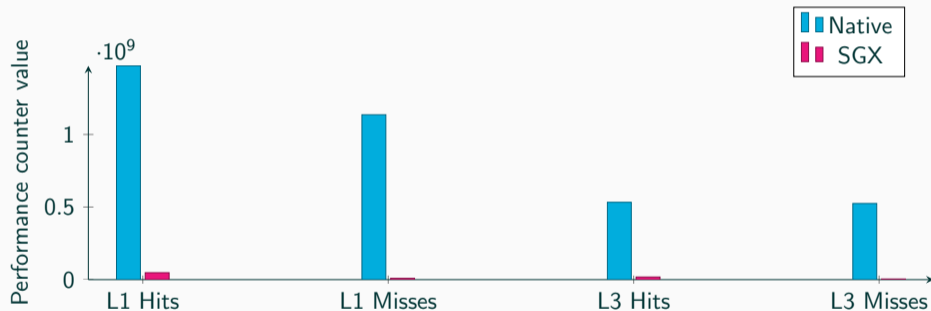
...processed with a simple moving average...



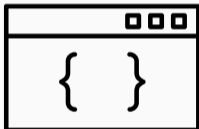
...allows to clearly see the bits of the exponent



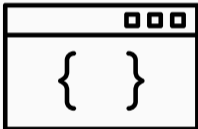




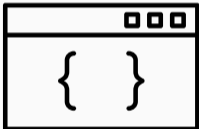
Countermeasures



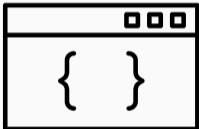
- Cache attacks can be prevented on **source level**



- 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**



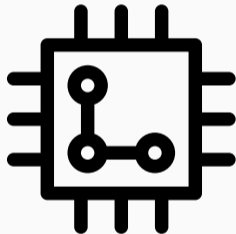
- Trusting the operating system weakens SGX threat model
- Method for the operating system to **inspect enclave code**
- Re-enable certain **performance counters**, such as L3 hits/misses



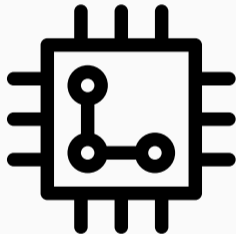
- Trusting the operating system weakens SGX threat model
- Method for the operating system to **inspect enclave code**
- Re-enable certain **performance counters**, such as L3 hits/misses
- **Enclave coloring** to prevent cross-enclave attacks



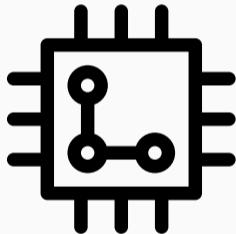
- Trusting the operating system weakens SGX threat model
- Method for the operating system to **inspect enclave code**
- Re-enable certain **performance counters**, such as L3 hits/misses
- **Enclave coloring** to prevent cross-enclave attacks
- **Heap randomization** to randomize cache sets



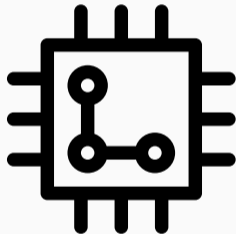
- Intel could prevent attacks by changing the hardware



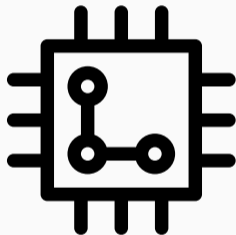
- Intel could prevent attacks by changing the hardware
- Combine Cache Allocation Technology (CAT) with SGX



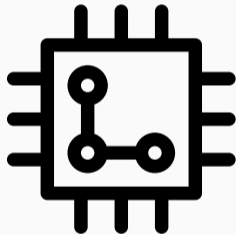
- Intel could prevent attacks by changing the hardware
- Combine Cache Allocation Technology (CAT) with SGX
 - Instead of controlling CAT from the OS, combine it with eenter



- 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

Conclusion

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







Thank you!

Cash Attacks on SGX

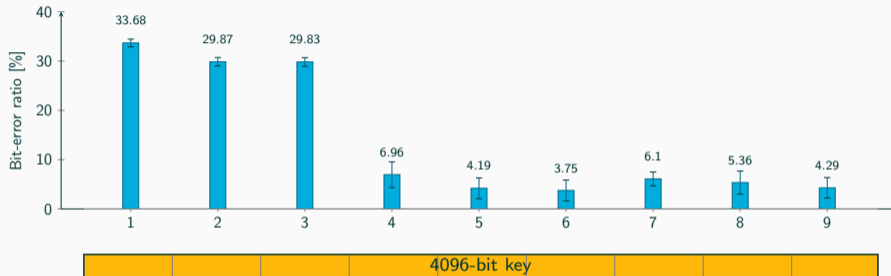
Daniel Gruss, Michael Schwarz

September 9, 2017

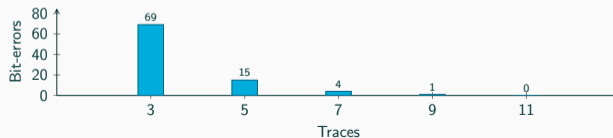
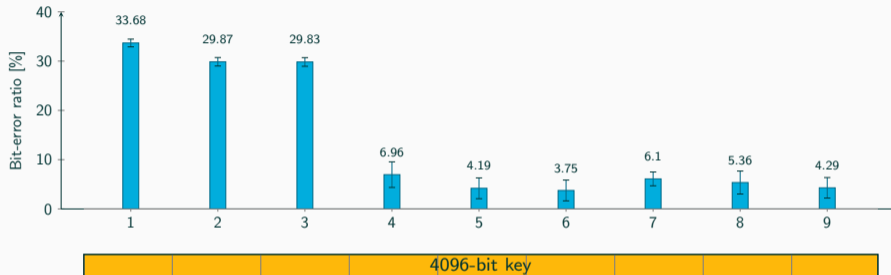
Graz University of Technology

-  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.

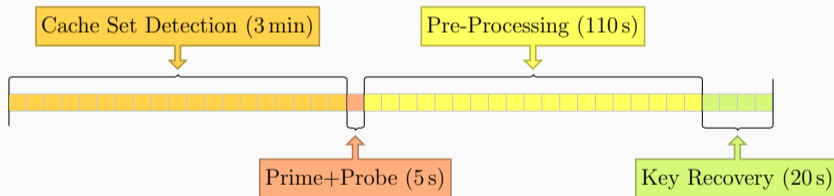
Error probability depends on which cache set of the key we attack



Error probability depends on which cache set of the key we attack



Full recovery of a 4096-bit RSA key in approximately 5 minutes



CPU cycles one increment takes

```
rdtsc 1
```

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

CPU cycles one increment takes

rdtsc 1

C 4.7

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


CPU cycles one increment takes

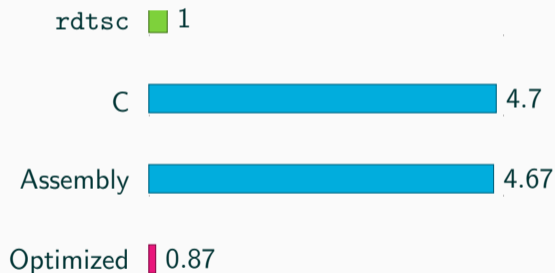
rdtsc 1

C 4.7

Assembly 4.67

```
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
```

