NANYANG TECHNOLOGICAL UNIVERSITY

Tutorial 1: Side Channels and Covert Channels

presented by

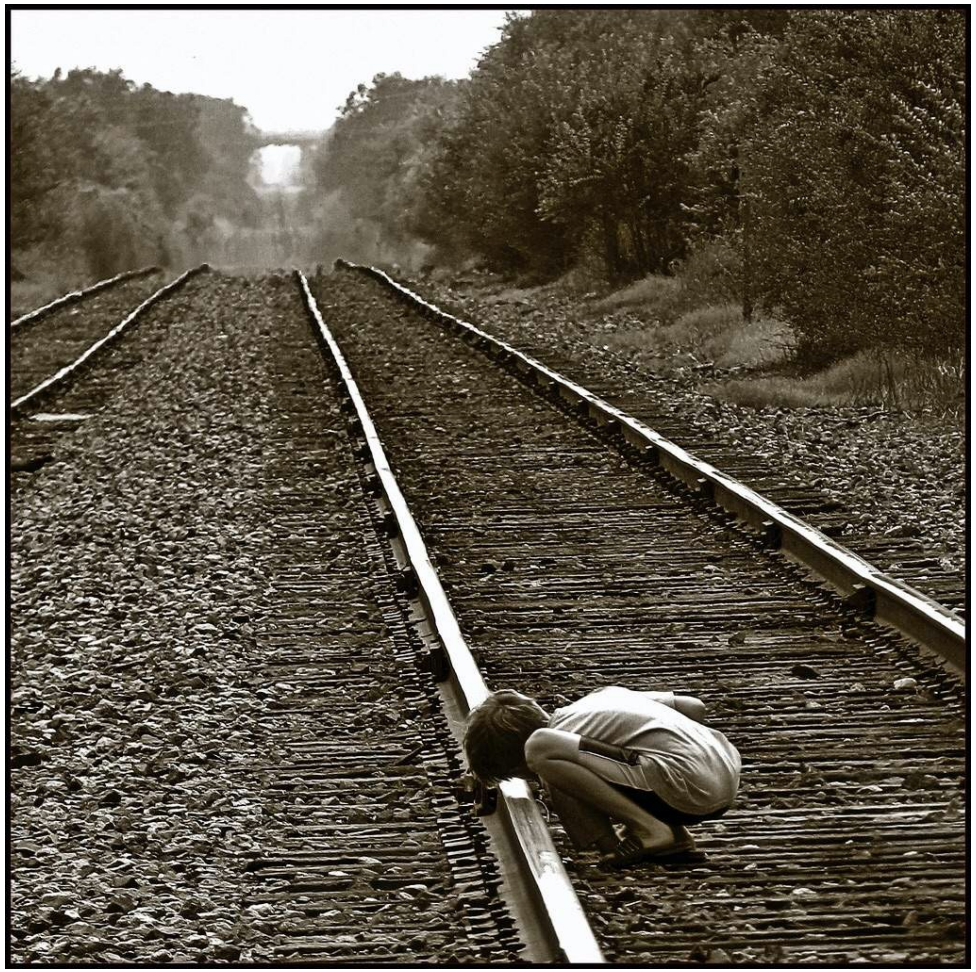
Li Yi
Assistant Professor
SCSE

N4-02b-64
yi_li@ntu.edu.sg

COPYRIGHT STATEMENT

- All course materials, including but not limited to, lecture slides, handout and recordings, are for your own educational purposes only. **All the contents of the materials are protected by copyright, trademark or other forms of proprietary rights.**
- All rights, title and interest in the materials are owned by, licensed to or controlled by the University, unless otherwise expressly stated. **The materials shall not be uploaded, reproduced, distributed, republished or transmitted in any form or by any means, in whole or in part, without written approval from the University.**
- You are also not allowed to take any photograph, film, audio record or other means of capturing images or voice of any contents during lecture(s) and/or tutorial(s) and reproduce, distribute and/or transmit any form or by any means, in whole or in part, without the written permission from the University.
- Appropriate action(s) will be taken against you including but not limited to disciplinary proceeding and/or legal action if you are found to have committed any of the above or infringed the University's copyright.

Side Channel: what is it?



Slide courtesy of David Lie

Covert Channels & Side Channels

- Security models specify how information should flow in the system when communicating explicitly (e.g., read and write)
- A covert/side channel allows implicit information flow, outside of the specified policy
  - e.g., suppose two processes share a lock on the file, then one process can transmit one bit of information to the other process, by holding or not holding the lock
  - e.g., a process can vary the amount of time it takes to execute by causing many page faults, thus transmitting information to another party

Covert Channels & Side Channels

- Covert vs. Side Channels
  - Covert channels usually imply that information is sent intentionally, and the sender wishes to remain undetected
  - With side channels, information is leaked unintentionally

Types of Side Channels

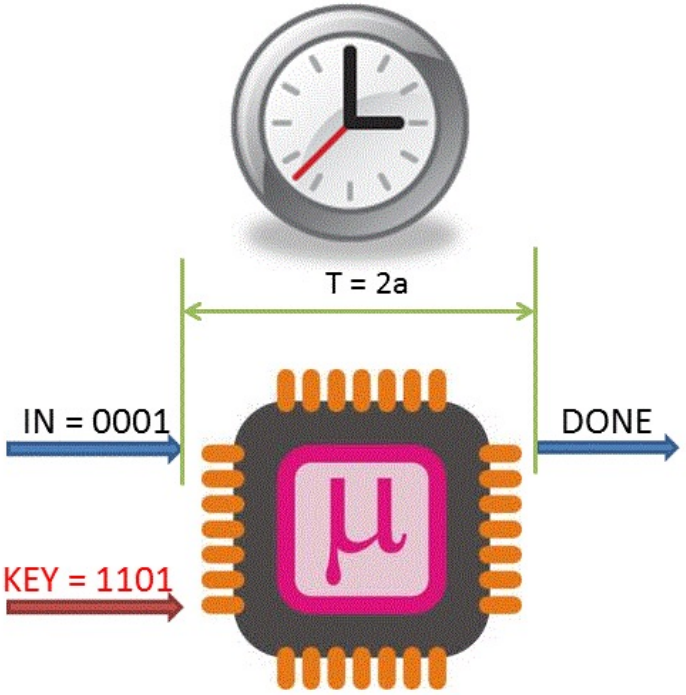
- Side channel tends to exploit the physical implementation of the deployed security mechanism rather than using brute force or algorithmic weakness inherent in the mechanism
- Examples for side-channels:
  - Timing
  - Temperature, radiation, etc.
  - Electromagnetic channel: analysis of radio emissions, etc.
  - Video channel: analysis of reflected light emissions
  - Power consumption (simple, differential…)
  - Acoustic noise (e.g. keyboards..)
  - Cache
  - …

Timing Side Channel

- The computation time depends on the value of secret data, so one can uncover the secret by timing the execution of a particular operation

Algorithm 1

```
MES = IN ⊕ KEY;
FOR EACH b BIT in MES {
   IF (b == 1)
      routine();
}
```



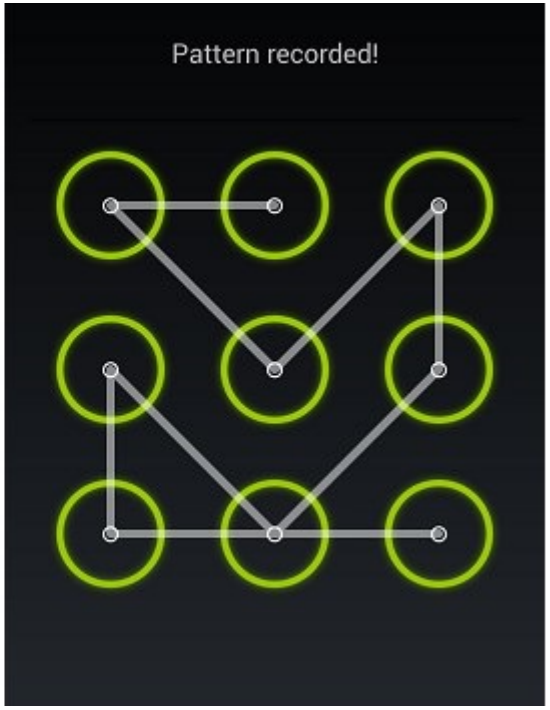
https://www.youtube.com/watch?v=2-zQp26nbY8

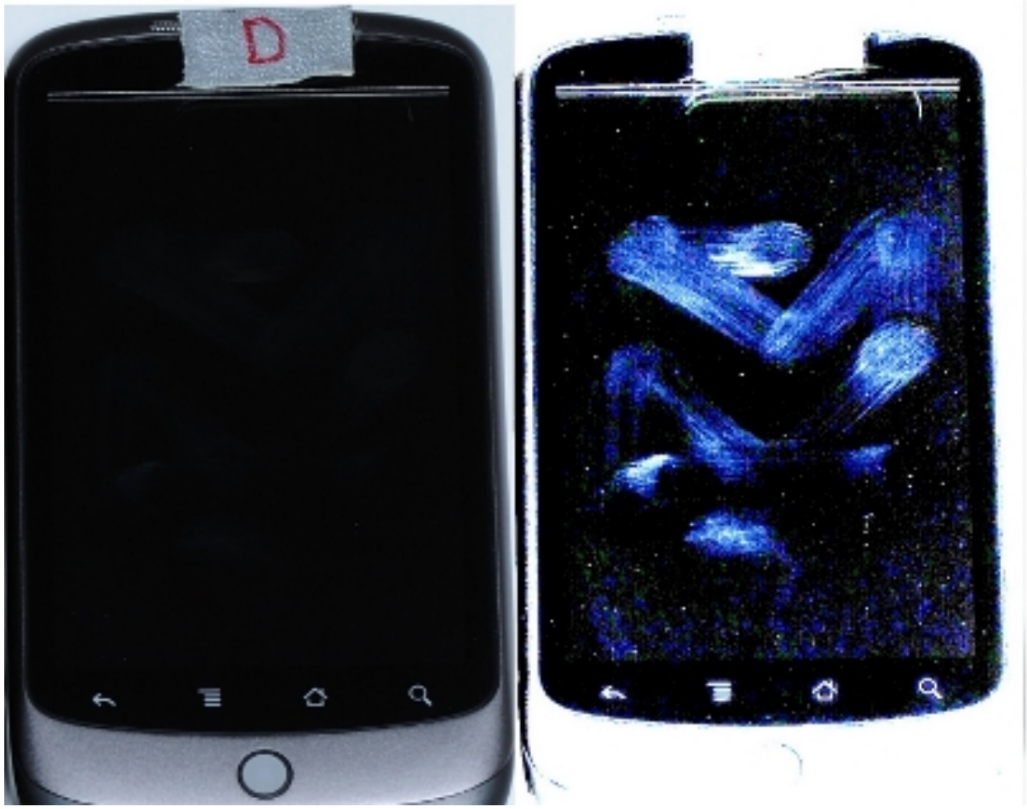
Temperature Side Channel

Quiz: what was the PIN just entered?

https://youtu.be/8Vc-69M-UWk

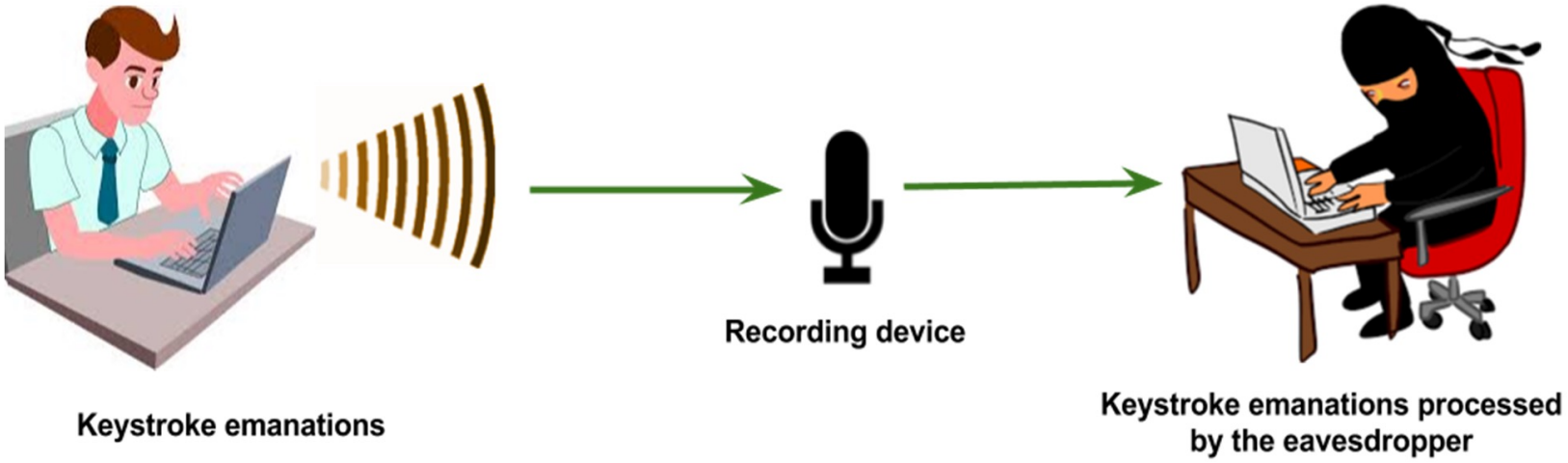
Example: "Smudge Attack"





Smudge Attacks on Smartphone Touch Screens, Aviv et al.
http://www.usenix.org/event/woot10/tech/full_papers/Aviv.pdf

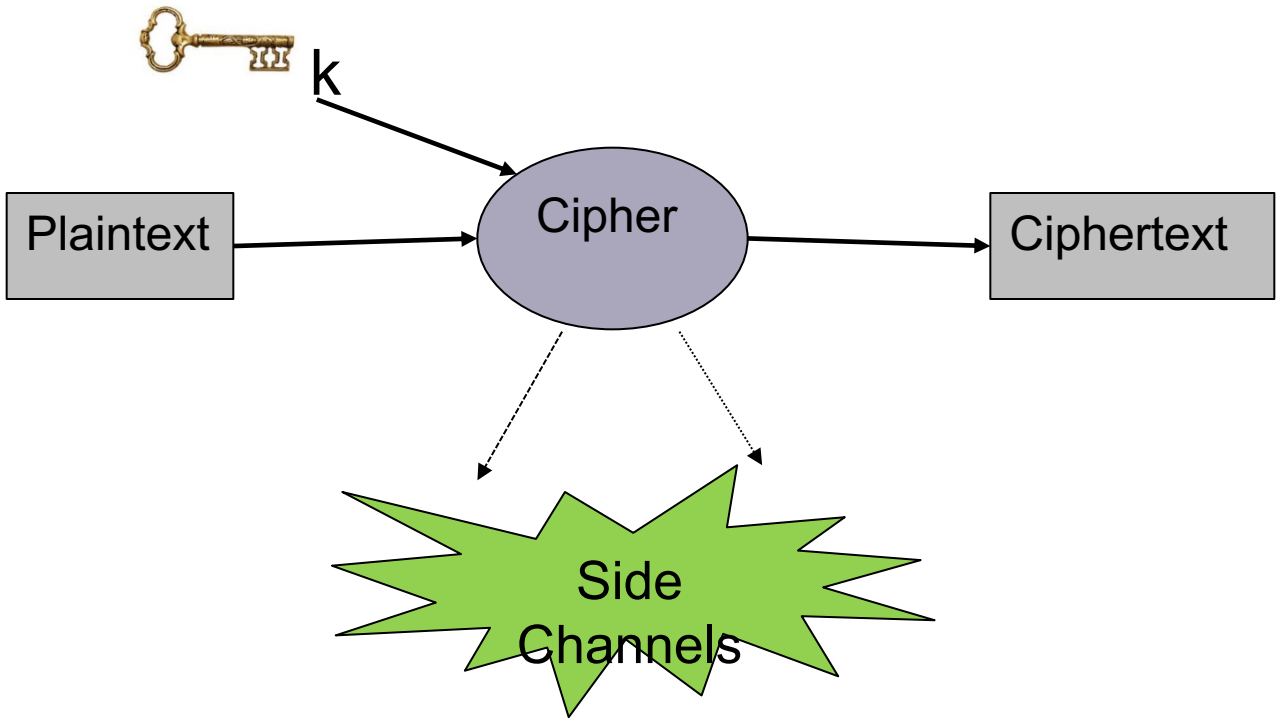
Acoustic (Sound) Side Channel



- It can be done by a smart phone these days
- Attacker may use frequency analysis if the input language is known

Side Channels in Cryptosystem

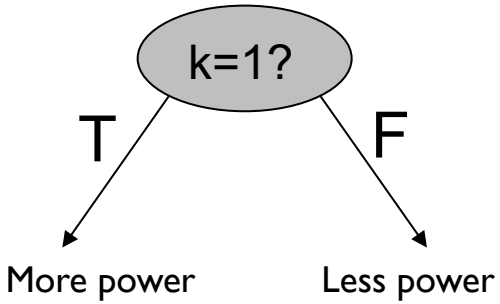
- Any observable information (e.g. key bits) emitted as a byproduct of cryptographic operations.



Example: Differential Power Consumption

- A bit of key information can be discovered by detecting which branch is executed upon a key-dependent conditional branch
- Because the device does different things depending on which branch is actually executed, consuming different amount of power

```
Let R1 = 1
For k = w-1 downto 0:
    If (bit k of x) is 1 then
        Let R0 = R1•y mod n
    Else
        Let R0 = R1
    Let R1 = R0•R0 mod n
EndFor
Return R0
```



Power Side Channel

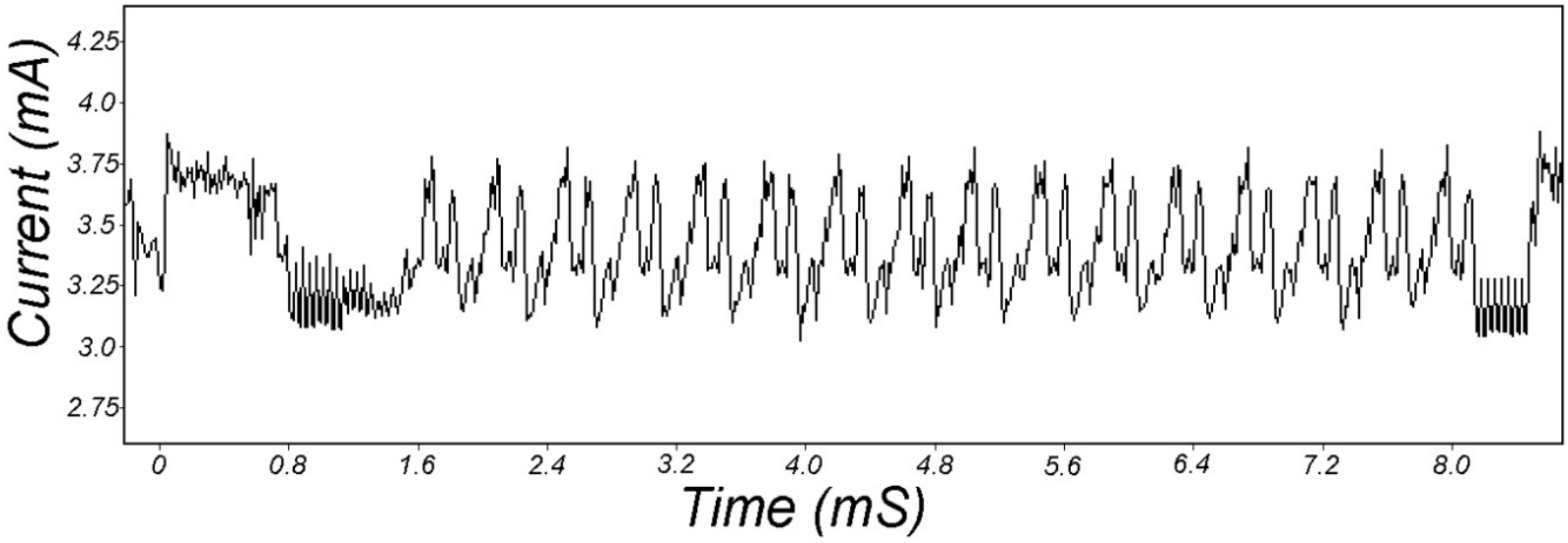


Figure 1: SPA trace showing an entire DES operation.

Paul Kocher, Joshua Jaffe and Benjamin Jun. "Differential Power Analysis". Crypto'99

Type of Covert Channels

Timing channel: Information is transmitted by varying the time it takes to perform an action:

- Example: a process varies the delay between packets. A receiver listening on the network (but not receiving the packets) can then read information. It is not obvious that the process is communicating with the eavesdropper.

Termination channel: Information is transmitted depending on whether a process has terminated

- Example: a process starts and stops a child process at certain times. Another process can observe the list of running processes. Since there is no explicit communication, there is no obvious channel for an OS to detect or prevent.

Type of Covert Channels

Storage channel: Information is transmitted via properties of a storage object (i.e., size of a file, locks on a file, existence of a file, name of a file, etc.)

- Example: a process changes the size of a file. Another process doesn't have the privileges to read the file, but can see the size and thus can communicate with the first process.

Steganography: Information is hidden within some content, such as an encoded text file within a picture

- Example: messages can be hidden in images and videos. The colour or details can be altered in a way that would not be detectable to a person, but can be read and understood by a party that knows the encoding.

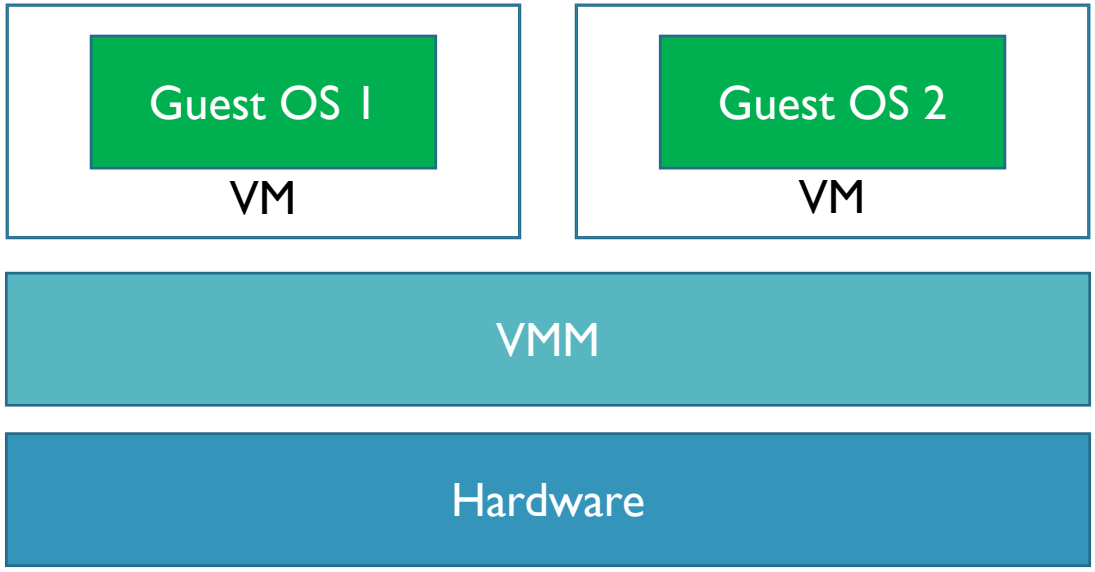
Type of Covert Channels

Power channel: Information about what a circuit is computing is transmitted depending on the current drawn

- Example: a process performs computation running down the battery. The amount of battery can be read by another process even if the two processes are not permitted to communicate.

A Practical Scenario

- In Cloud computing environment two users can share same hardware
- Users running on different cores share the last level cache



Why Are They Important?

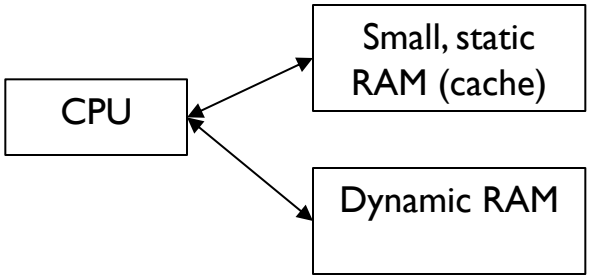
- Difficult to detect
- Can operate for a long time and leak a substantial amount of classified data of low-level privileged processes
- Can compromise an otherwise secure system, including one that has been formally verified!
- Must be considered to achieve high government certification levels

CPU Cache

Recap on Computer Architecture Course

CPU Cache

- A CPU cache is a hardware cache used to reduce the average cost (time or energy) to access data from the main memory
- A cache is a smaller, faster memory, closer to a processor core, which stores copies of the data from frequently used main memory locations



Modern CPUs

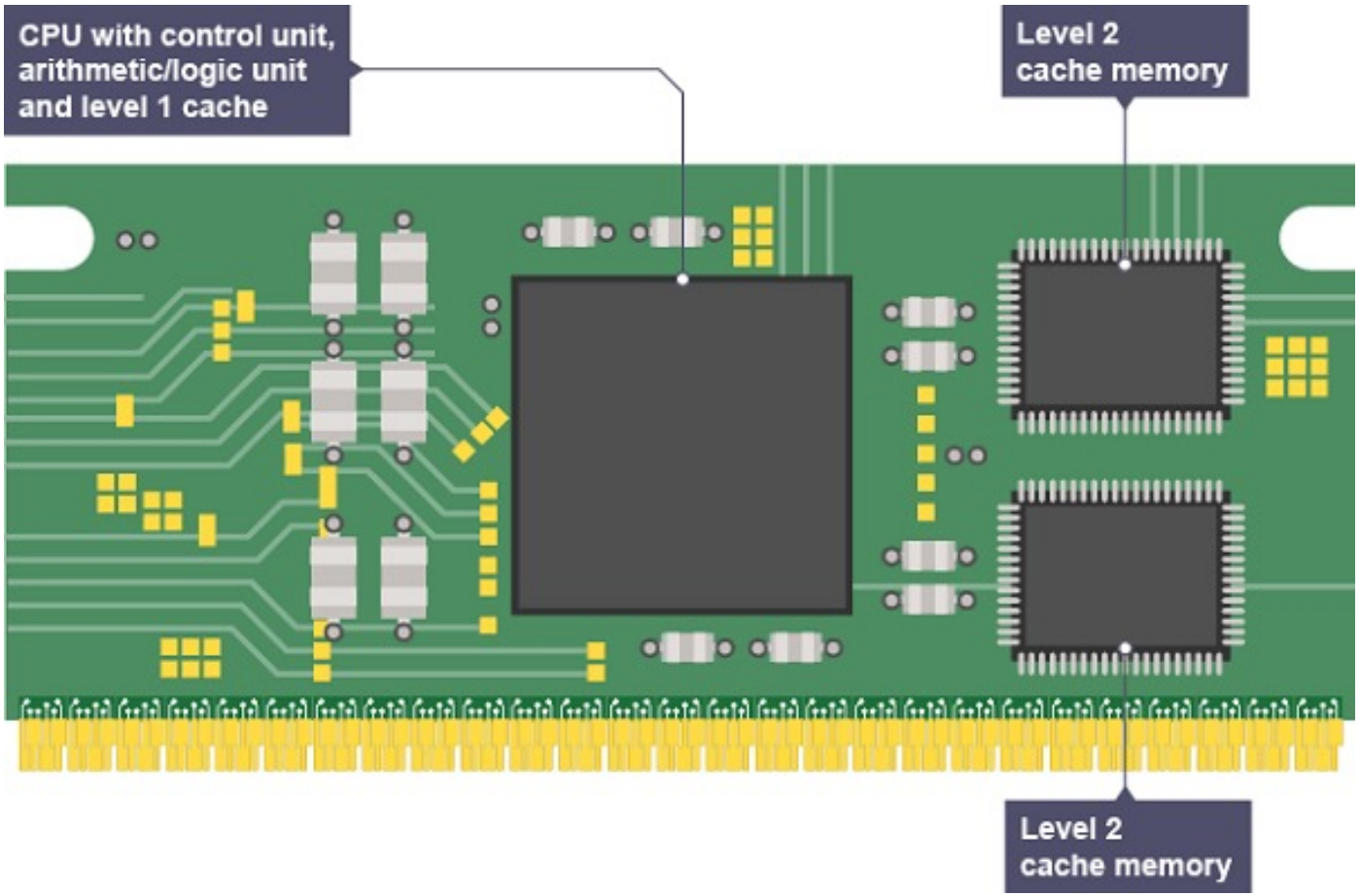


Image source: https://www.guidingtech.com/53366/cpu-cache-explained/

Modern CPUs

- Have at least three independent caches:
  - an instruction cache to speed up executable instruction fetch
  - a data cache to speed up data fetch and store, usually organized as a hierarchy of more cache levels (L1, L2, etc.)
  - a translation lookaside buffer (TLB), a cache used to speed up virtual-to-physical address translation for both executable instructions and data.
- Caches are generally sized in powers of two: 4, 8, 16 etc. KB or MB (for larger non-L1) sizes, although the IBM z13 has a 96 KB L1 instruction cache

Memory Hierarchy

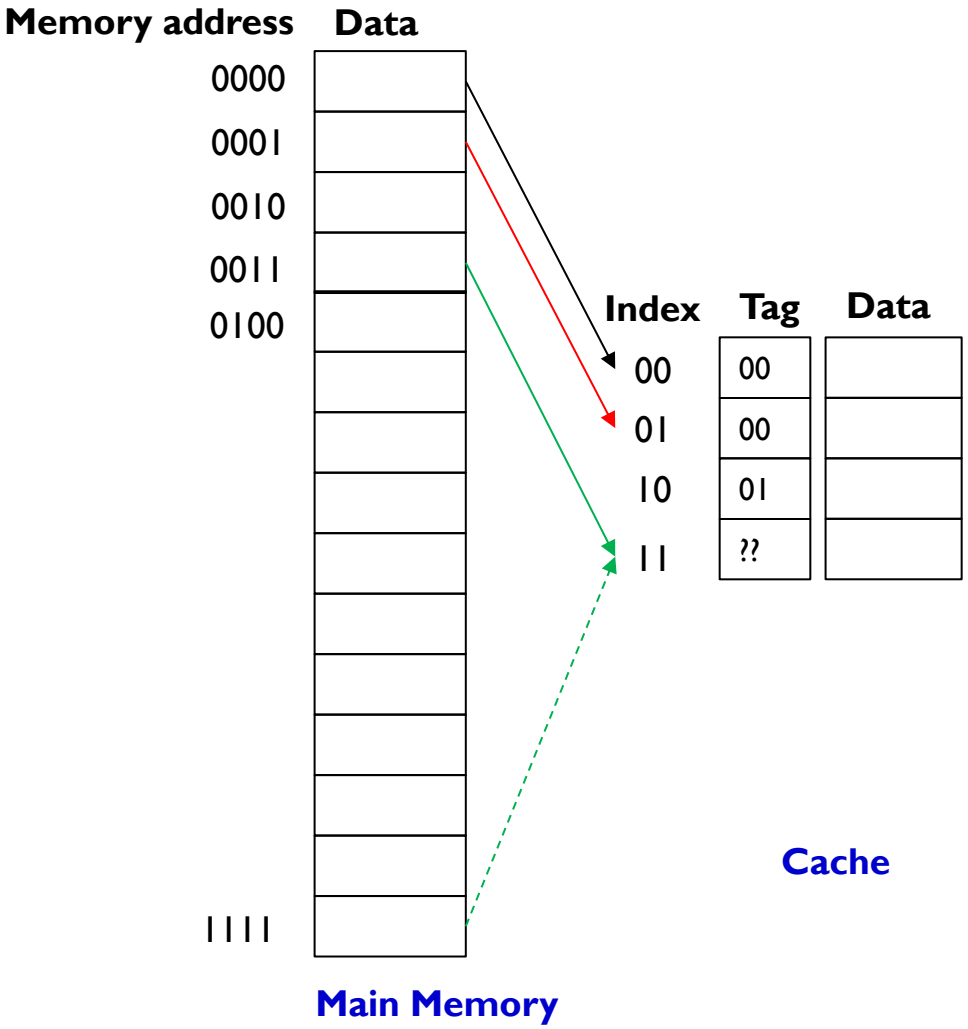
- Memory closer to CPU and smaller provides faster access times
- Larger and further-away memory provides cheaper storage
- Puts frequently-accessed data in small, fast (and expensive) memory, with assumption of non-random data access behaviors

		Access Time	Size
Primary memory	Registers	1 clock cycle	~500 bytes
	Cache	1-2 clock cycles	~10 MB
	Main memory	1-4 clock cycles	~10 GB
Secondary memory	Disk	5-50 msec	TB

Cache Entries

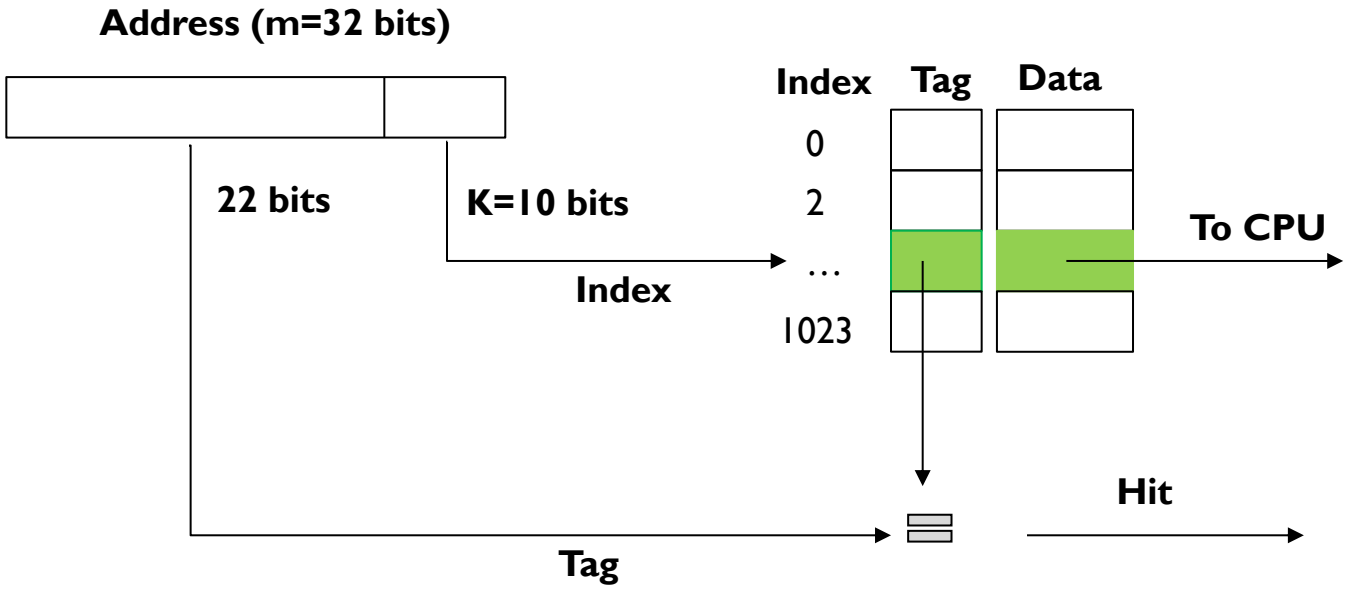
- Data is transferred between memory and cache in blocks of fixed size, called cache lines or cache blocks
- A cache entry is created when a cache line is copied from memory into the cache. It consists of the copied data as well as the requested memory location (called a tag)
- When the processor needs to read or write a location in main memory, it first checks for a corresponding entry in the cache. If any cache line contains that address, a cache hit has occurred. Otherwise, a cache miss has occurred

Cache Entries (Direct Mapping)



On a Cache Hit

- When the CPU tries to read from memory, the address will be sent to a cache controller
  - The lowest k bits of the address will index a block in the cache
  - If the block is valid and the tag matches the upper (m - k) bits of the m-bit address, then that data will be sent to the CPU

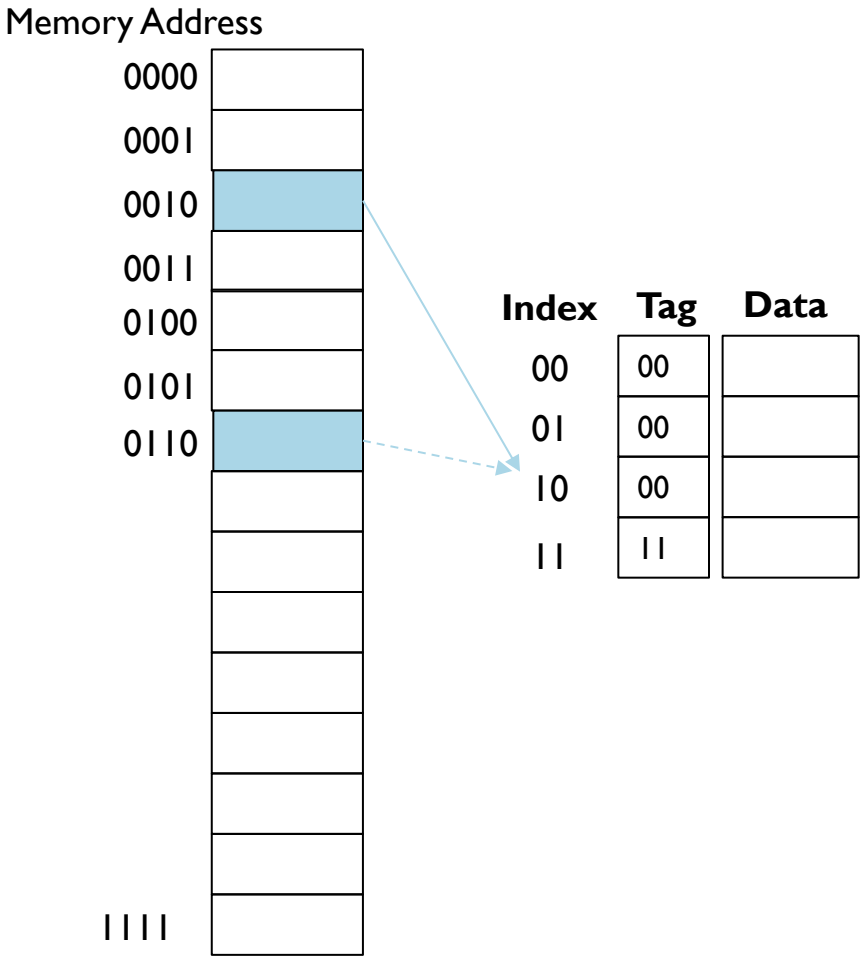


On a Cache Miss

- For a cache miss, the cache allocates a new entry and copies data from main memory, then the request is fulfilled from the contents of the cache
- To make room for the new entry on a cache miss, the cache may have to evict one of the existing entries, e.g., replacing least recently used entries
- On a cache miss, the CPU access time would be much slower (larger cycle times), compared with a cache hit
- "The simplest thing to do is to stall the pipeline until the data from main memory can be fetched and also copied into the cache"
  - But modern CPUs are not that simple…

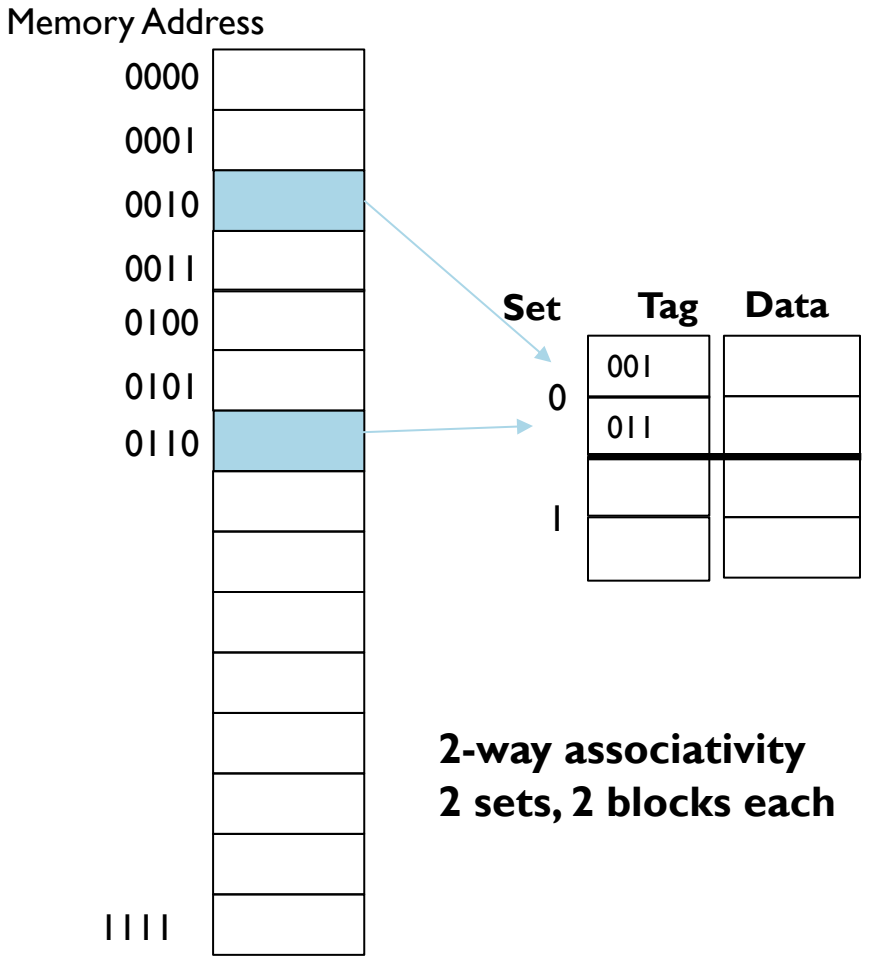
Disadvantage of Direct Mapping

- The direct mapped cache is easy: indices can be computed with bit operators or simple arithmetic
- Because each memory address belongs in exactly one block
- But, what happens if a program uses address 2, 6, 2, 6, 2, …?



Alternative: Set Associativity

- Fully-associative cache permits data to be stored in any cache block, instead of forcing each memory address into one particular block; but is expensive to implement because tag field must now hold the full address of memory
- Set-Associative cache: the cache is divided into groups of blocks, called sets
- Each memory address maps to exactly one set in the cache, but data may be placed in any block within that set



Cache-Based Side Channel

Percival's Attack on RSA

- Colin Percival, "Cache Miss for Fun and Profit"

- Modern microprocessors, such as Simultaneous Multi-Threading (SMT) processors (Intel Hyper-threading), allow multiple threads to run simultaneously, sharing part of the cache subsystem
- This gives an attacker process the ability to *directly observe* other concurrent threads' cache accesses and obtain a relatively accurate trace
- In 2005, Percival demonstrated an attack against the popular OpenSSL implementation of the RSA algorithm using this approach

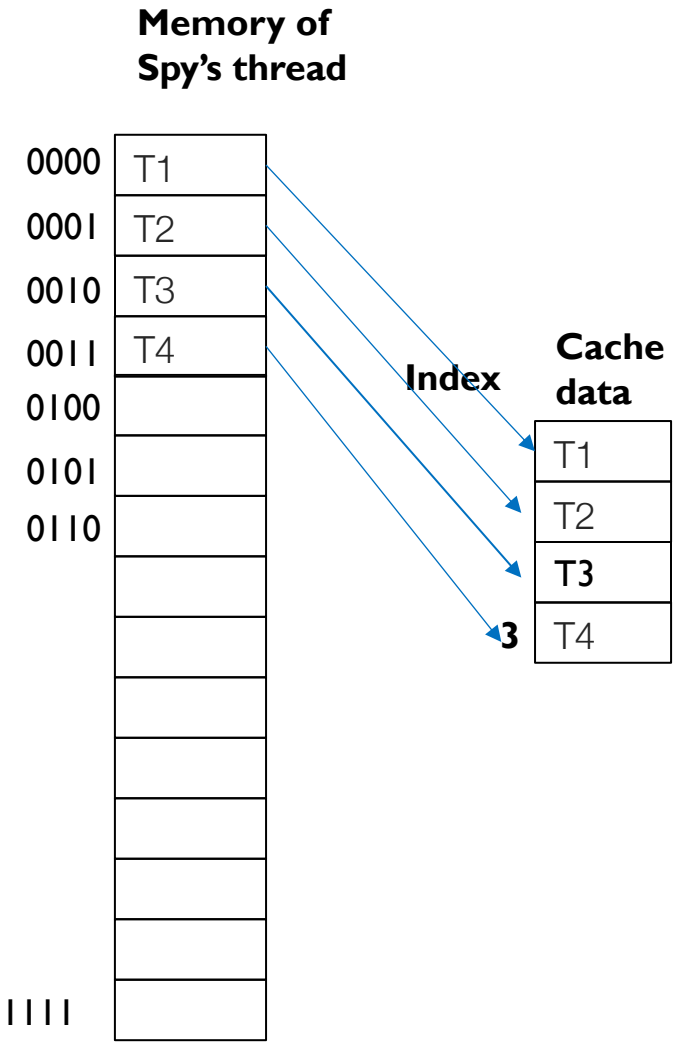
L1 Cache Missing

- Hyper-Threading permits two threads to execute on a single Pentium 4 core
  - Covert channel: Trojan (highly-privileged process) runs one thread, spy runs another
  - Side channel: victim runs one thread, spy runs another

- L1 data cache: 128 cache lines of 64 bytes each (8KB), organized into 32 4-way associative sets
  - Cache is shared between threads
  - But they cannot read each other's cache data

L1 Cache Missing ctd.

- Spy allocates 8KB array; Trojan allocates 2KB array
- The threads can communicate via a timing channel by forcing each other's data out of the (L1) cache
- Spy repeatedly reads its 8KB array, filling L1 cache



Prime + Probe Technique

Prime + probe technique consists of 3 stages:

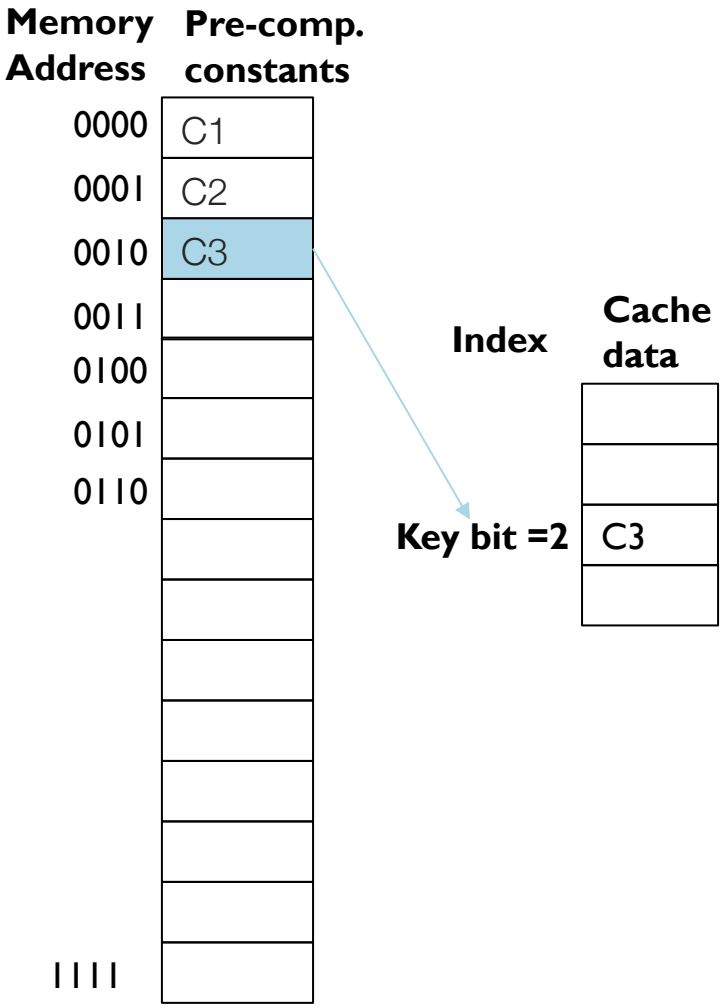
- Prime stage: the attacker fills the cache with his own cache lines
- Victim accessing stage: the victim process runs
- Probing stage: the attacker accesses the priming data again. If the victim process evicts the primed data, the reloading will incur cache miss

OpenSSL RSA Key Theft

- The attacker manages to run simultaneously with the victim process which is performing RSA encryption
- His goal is to discover bits of the private encryption key used by the victim
- The attacker sequentially and repeatedly accesses an array, thus loading in his own data to occupy all cache lines; at the same time he measures the delay for each access to detect cache misses, e.g., using the `rdtsc` instructions to read a timer in Intel x86 processors
- The victim's cache accesses will evict the attacker's data, causing the attacker to miss on these cache lines, enabling detection by the attacker

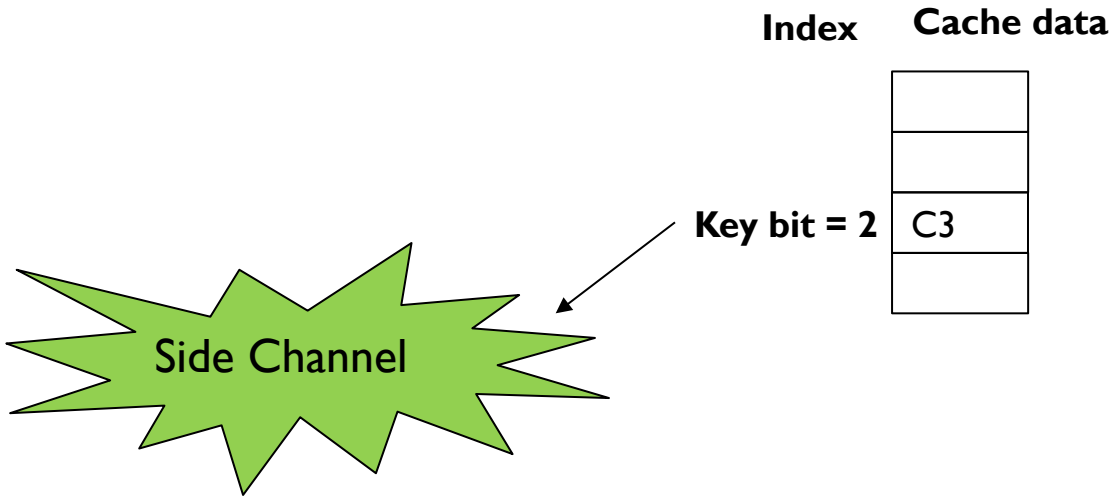
OpenSSL RSA Key Theft ctd.

- The core operation used in RSA is modulo exponentiation
- It is often implemented with a series of squarings and multiplications. The encryption key is also divided into a series of segments
- $x := a^d \ mod \ p$ →
  - $x := x^2$, ... (if key bit is not set)
  - $x := x \cdot a^{2k+1}$ (if key bit is set)
- For each multiplication, a multiplier is selected from a set of pre-computed constants stored in a table: $\{a, a^3, a^5, \ldots, a^{31}\} \ mod \ p$
- During the table lookup, a segment of the encryption key is used to index the table

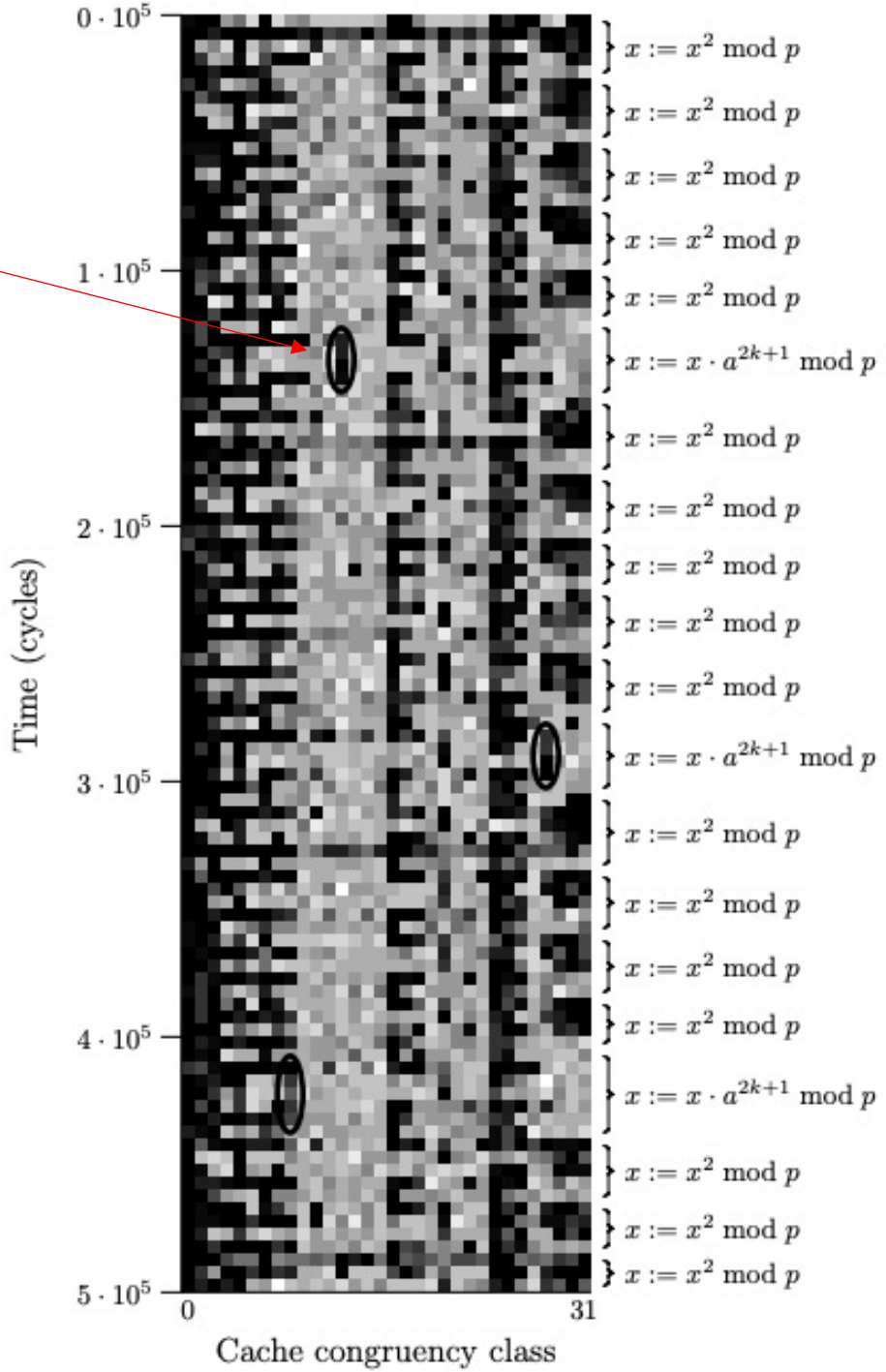


OpenSSL RSA Key Theft ctd.

- As the table is stored in memory, the attacker can detect the cache evictions caused by the victim (the encrypting process) for the table lookup
- Based on which line is evicted, the attacker can infer which table entry is accessed. This tells the attacker the index used for this table lookup, which is a segment of the encryption key



Reveals a^{2k+1}



120 cycles (white)

>170 cycles (black)

Noise

- Challenge: "noise" from the modular multiplications
  - Eviction may also result from other operations if they are mapped to the same cache set
  - Even the correct cache set is identified, the multiplier is often not uniquely determined
- But… able to identify within two possibilities in 50% of the modular multiplications
  - 310 out of 512 bits

Countermeasures

- Architecture-level:
  - Don't share caches between threads
    - More expensive, slower
  - Change cache eviction strategy to enforce fair sharing between threads, i.e., only allow thread A to evict a cache line "owned" by thread B if thread B currently "owns" more than its fair share
    - Performance penalty
- OS-level:
  - Make sure low- and high-level processes never share the processor simultaneously