CSCI 667: Concepts of Computer Security

Lecture 25

Prof. Adwait Nadkarni

Derived from slides by Dmitry Evtyushkin

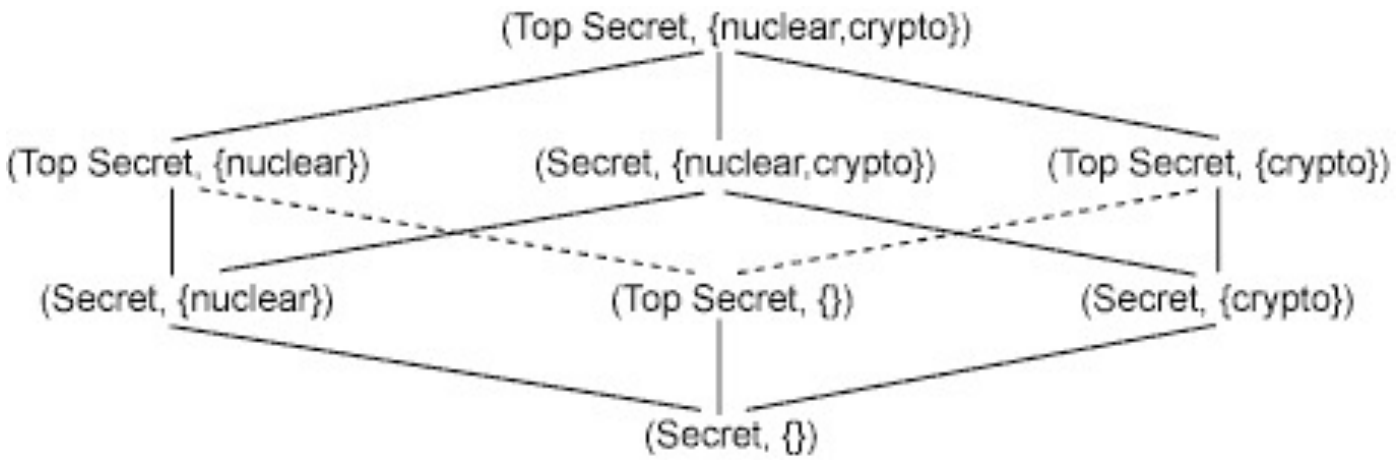
Trusted Processes

- Does it matter if we do not trust some of J's processes?
  - *Trojan Horse*: Attacker controlled code run by J can violate secrecy.
  - *Confused Deputy*: Attacker may trick trusted code to violate integrity

	O_1	O_2	O_3
J	R	RW	RW
S_2	-	R	RW
S_3	-	R	RW

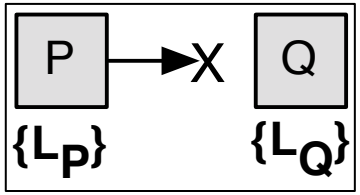
Bell-LaPadula (BLP) Model

- A Confidentiality MLS policy that enforces:
  - *Simple Security Policy*: a subject at specific classification level cannot read data with a higher classification level. This is short hand for "*no read up*".
  - ** (star) Property*: also known as the confinement property, states that subject at a specific classification cannot write data to a lower classification level. This is shorthand for "*no write down*".



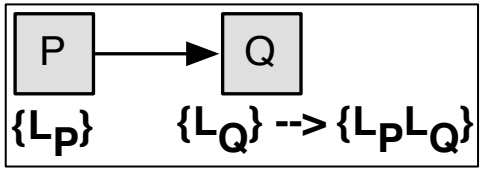
Two ways to enforce IFC

- *Explicit flows:* Predict data flows, enforce rules
  - *Incompatible with dynamic user-directed sharing!*



A. Explicit labels

- *Floating labels* (i.e., taint tracking):
  - Labels follow data
  - Seamless communication



B. Floating labels

```
//'a' contains a secret
a = some_secret;
c = null;
b+=a;
//'b' is tainted as well
```

Side Channels

A simple implicit flow

- Problem:
  - 'a' leaks *implicitly*
  - *i*.e., because of a condition

```
//'a' contains a secret
b = false;
if (a == 0) {
    b = true;
}
```

- Solution?
- If a condition depends on 'a'
  - Propagate labels to all the assignments resulting from it
  - i.e., 'b' gets the 'secret' label

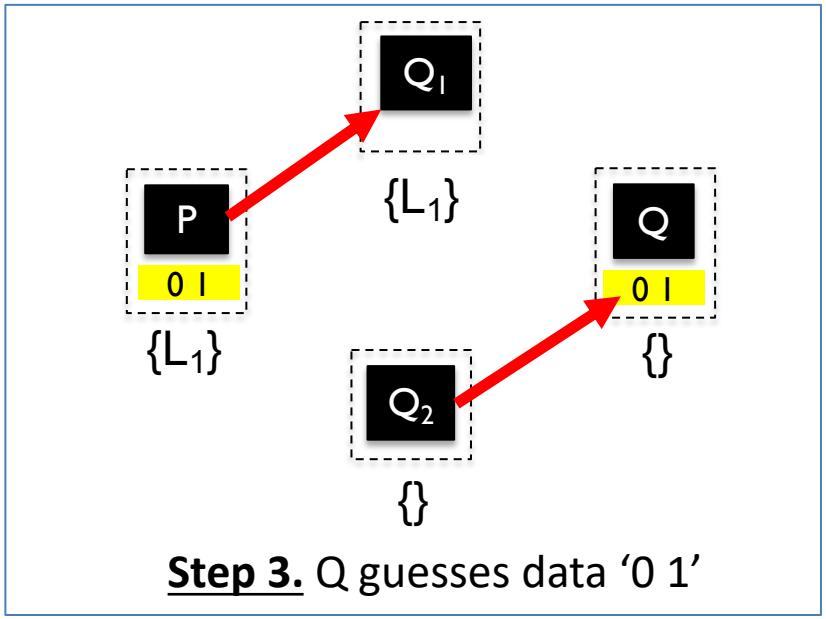
A slightly more complex implicit flow

Denning, 1976

```
1  //a is secret
2  b := c := false
3  if ~a then c := true
4  if ~c then b:=true
```

*a → b without label propagation: because either c:=true or b:=true were **not executed***

Krohn & Tromer, 2009



Step 3. Q guesses data '0 1'

Attack Setup:

- P sends a message to Q_i if the i^{th} bit is '0'
- All the Q_is send Q a message at a fixed time interval, *unless* they have received a message from P

Traditionally

Alice

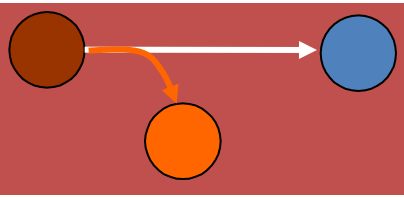
COMMUNICATION CHANNEL

Mallory

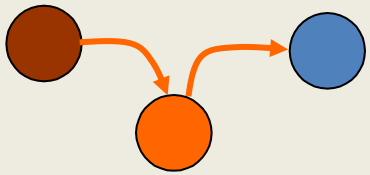
Meta Policy
- Confidentiality
- Integrity
- Authenticity

Bob

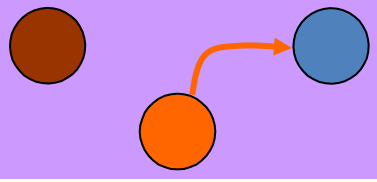
Attacks



Interception (Threat) → Confidentiality (Policy) → Encryption (Mechanism)

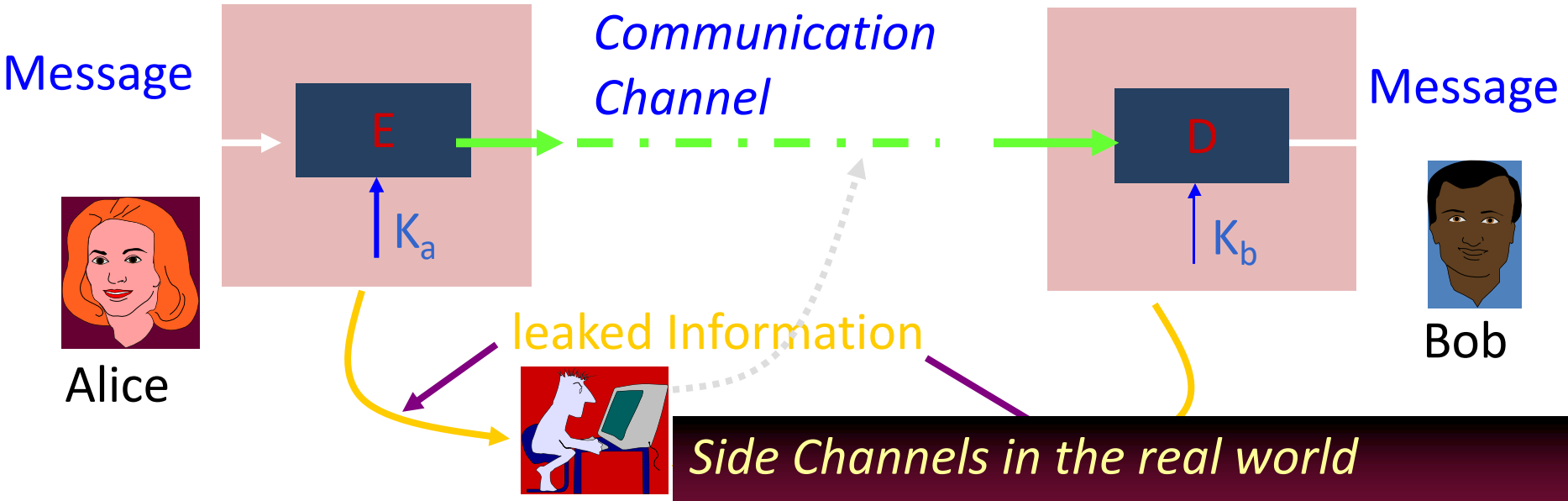


Modification (Threat) → Integrity (Policy) → Hash (Mechanism)



Fabrication (Threat) → Authenticity (Policy) → MAC (Mechanism)

In practice



Side Channels in the real world

Through which a cryptographic module leaks information to its environment unintentionally

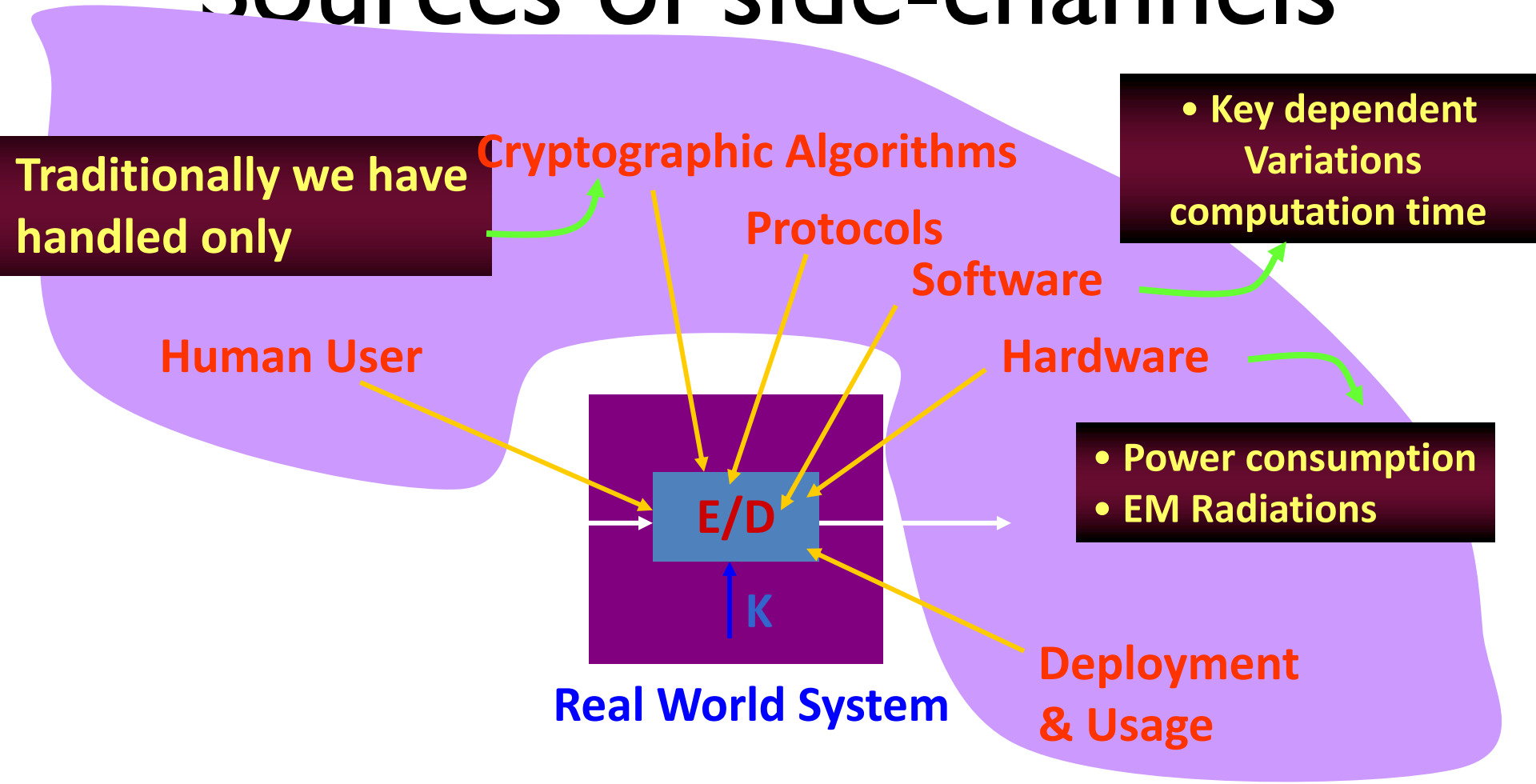
Assumptions

- *Only Alice Knows K_a*
- *Only Bob Knows K_b*
- *Mallory has access to E, D and the Communication Channel but does not know the decryption key K_b*

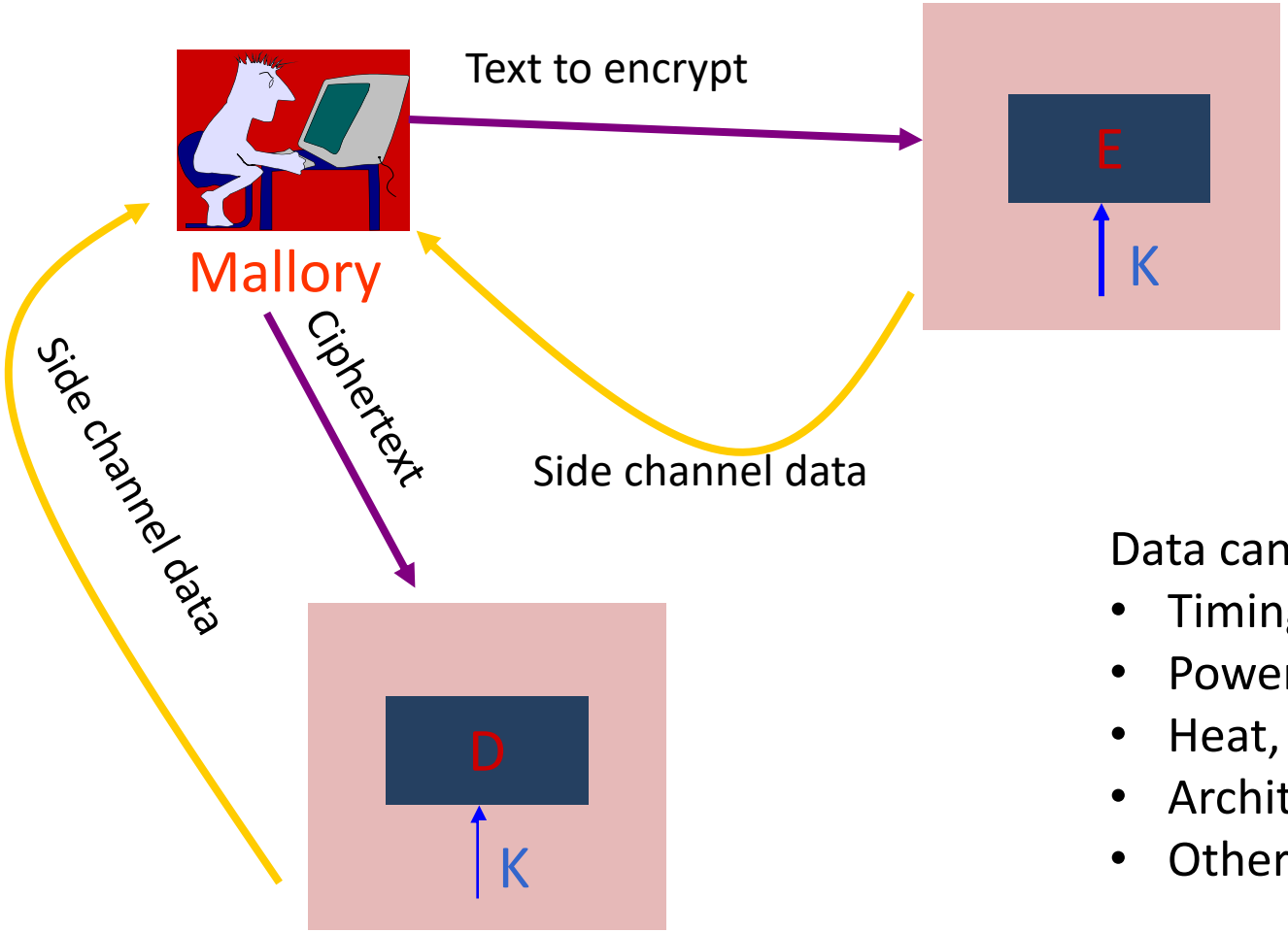
General Idea

- Traditionally it was assumed that data is safe as long as strong crypto is used, however…
- Cryptographic algorithms allow multiple correct implementations (data structures are not specified, etc.)
- Performance of crypto operations can be dependent on secret data (timing attack)
  - E.g. due to CPU caches
- Cryptographic operations can leave traces in shared resources

What all leaks information?: Sources of side-channels



Typical Attack model



Data can include:

- Timings
- Power consumption
- Heat, EM emission
- Architectural events
- Others

Timing Attacks Against RSA

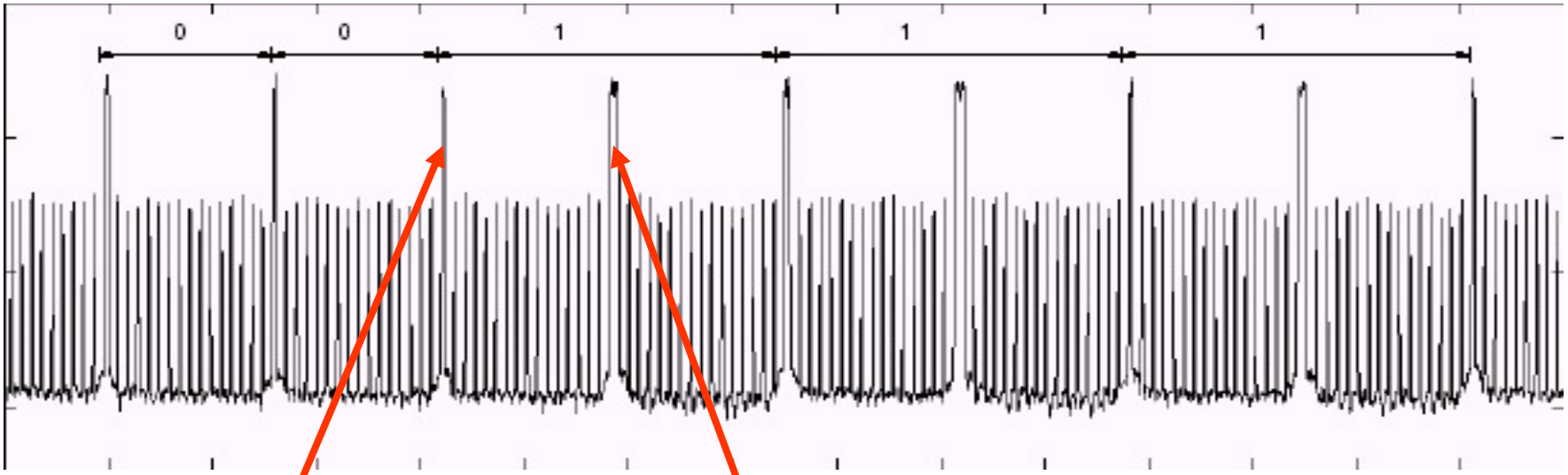
- Recovers the private key from the running time of the decryption algorithm
- Computing $m = c^d \bmod n$ using repeated squaring algorithm:

```
m = 1;
for i = k-1 downto 1
    m = m*m mod n;
    if d_i == 1
        then m = m*c mod n;
return m;
```

Example: Power Analysis

Idea: During switching, CMOS gates draw spiked current

Trace of Current drawn - RSA Secret Key Computation



Result: Many smart cards leak secret keys

A simple timing attack example

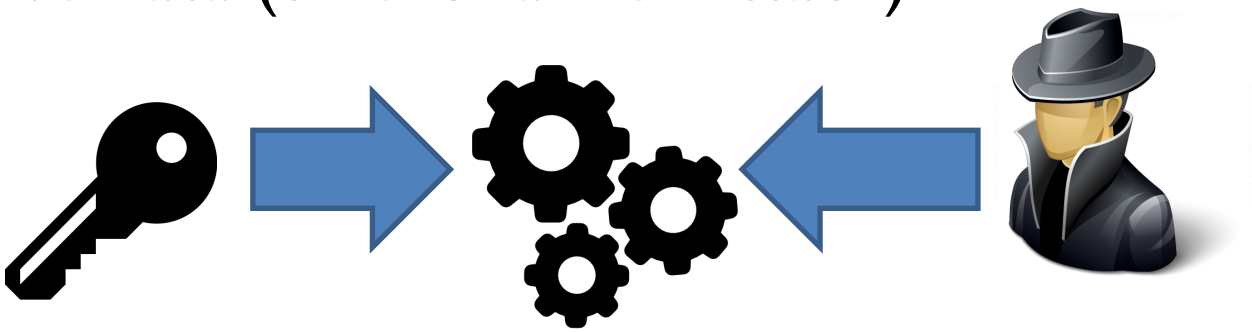
```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

char pass[] = "MYLONGPASSWORD";

int main(){
  char input [100];
  scanf("%100s", input);

  int pass_len = strnlen(pass, 100);
  int recived_len = strnlen(input, 100);

  if(pass_len!=recived_len) exit(1);

  for (int i=0; i<pass_len; i++){
    if(pass[i]!=input[i]){
      exit(1);
    }
  }
  printf("Hello!\n");
}
```

Side vs Covert Channels

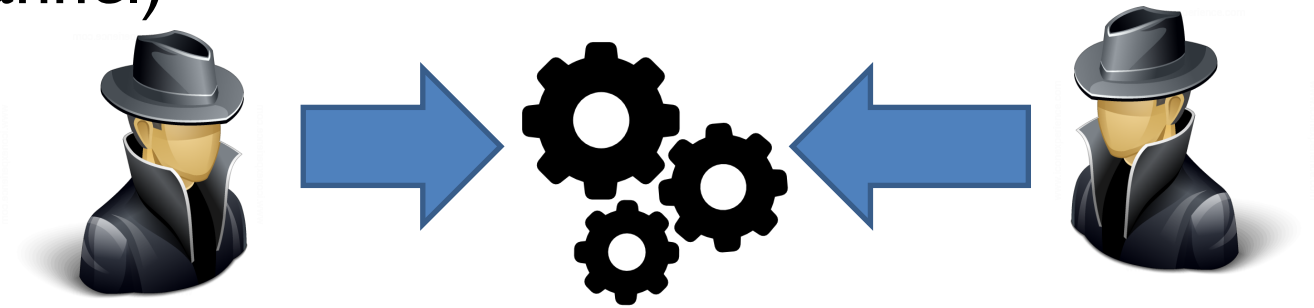
- Both are based on *extracting* information from media not designed for it
- Side channels: Spying on program activity using side effects of its execution
  - Finding if someone's home by looking at their lights
  - *Q: How to find out if car was driven or not an hour ago?*
- Covert channel: Intentionally communicating using execution artefacts
  - Prisoners communicating by banging on pipes
  - *Q: How to send a secret message (e.g. number) to your friend if you are only allowed to send empty emails?*

Side and Covert Channel Attacks through Shared Hardware

- Shared hardware resources e.g. CPU caches can leak sensitive data (Side Channel Attack)



- Malicious entities transfer information by manipulations with shared resource (Covert Channel)



Why is this important?

- Completely stealthy, *passive* attacks
  - Do not require crashing program (unlike memory corruption attacks)
  - Often completely unnoticeable
- Many side channels require physical access
  - The spy has to be able to measure
- Today: Architecture based side channels
  - Victim and spy run on the same system
    - Often not a problem with prevalence of Cloud Computing, JIT compiled 3rd party scripts, etc.
  - Spy uses the shared architecture components as a side channel

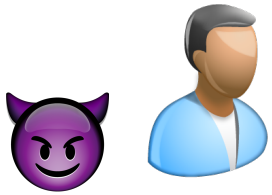
Why is this important?

- Completely stealthy, *passive* attacks
  - Do not require crashing program (unlike memory corruption attacks)
  - The spy has to be able to measure
- Today: Architecture based side channels
  - Victim and spy run on the same system
    - Often not a problem with prevalence of Cloud Computing, JIT compiled 3rd party scripts, etc.
  - Spy uses the shared architecture components as a side channel

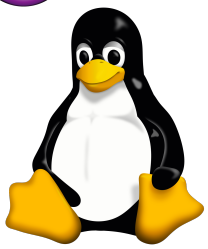
What is the *fundamental* reason why side-channels exist?

Problem: Resource Sharing

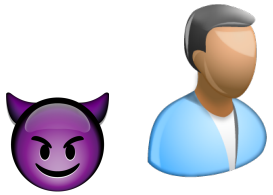
- Trusted and Untrusted code execute on same hardware using same resources



Attacker: Virtual machine
Victim: Virtual machine



Attacker: Unprivileged process
Victim: Privileged process



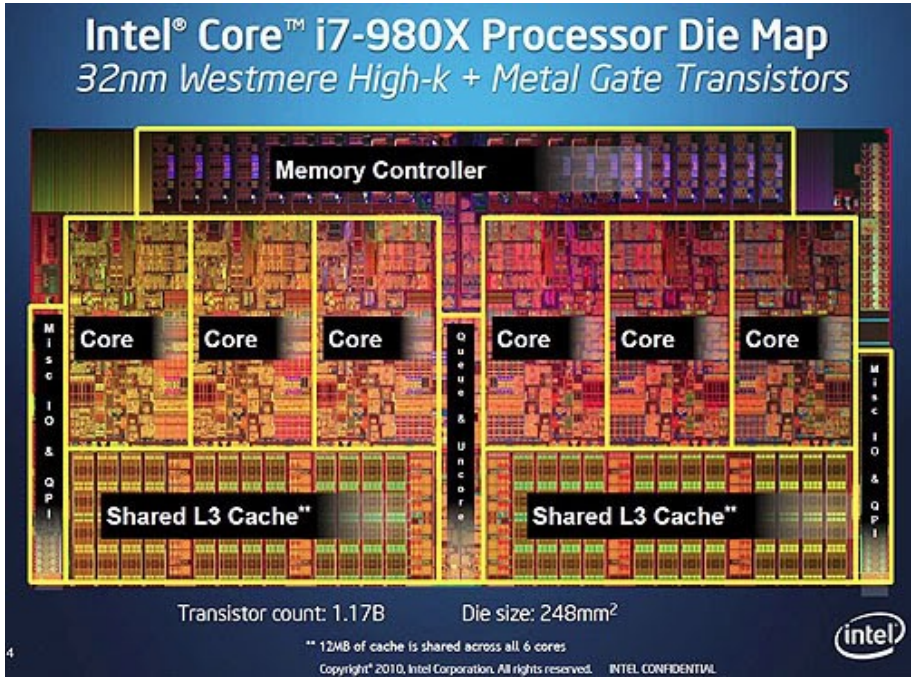
Attacker: Javascript code
Victim: Browser process with access to stored passwords

Attacker Presence

- Remote – attacker located over network
- Near the machine – attacker in physical proximity, is able to monitor "whole system" side channels, e.g. EM-emission
- Another CPU – On multi-socket machine, attacker runs on another physical CPU to victim
- Another Core – Same physical CPU, but different core
- Another Hyperthread – Same core, different virtual core
- Same Context – e.g. sandboxed code within trusted process

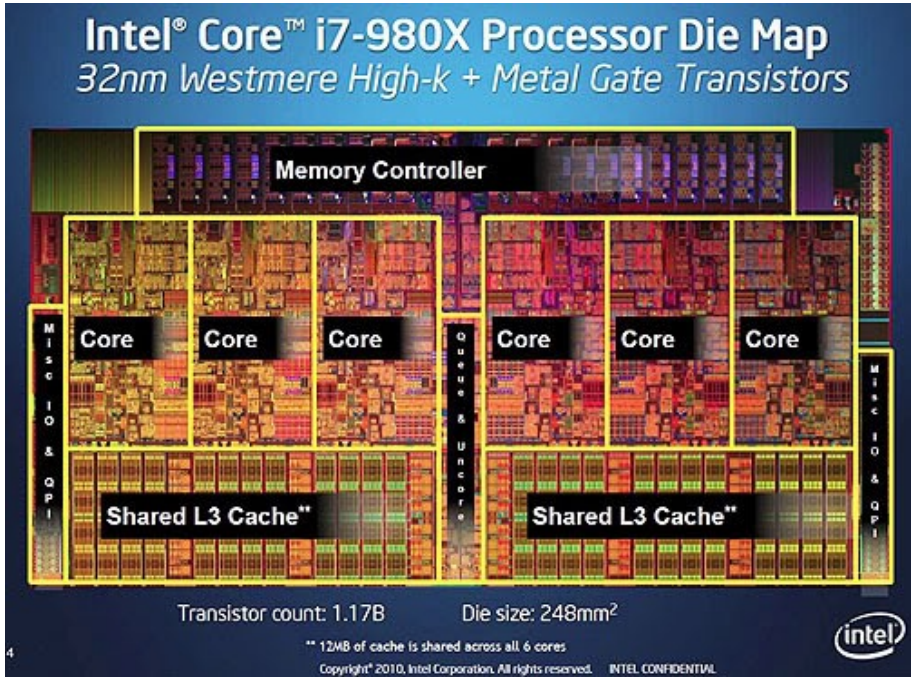
Microarchitectural Side Channels

- Modern processors support multiple programs running at the same time
  - On same core or on different cores
- Many shared resources
  - What one process does can affect others
  - Q: Examples?



Desired Properties

- What makes a good side channel?
  - The behaviour is observable by the attacker and depends on the victim's sensitive data
  - The observations (and behaviour) are deterministic.



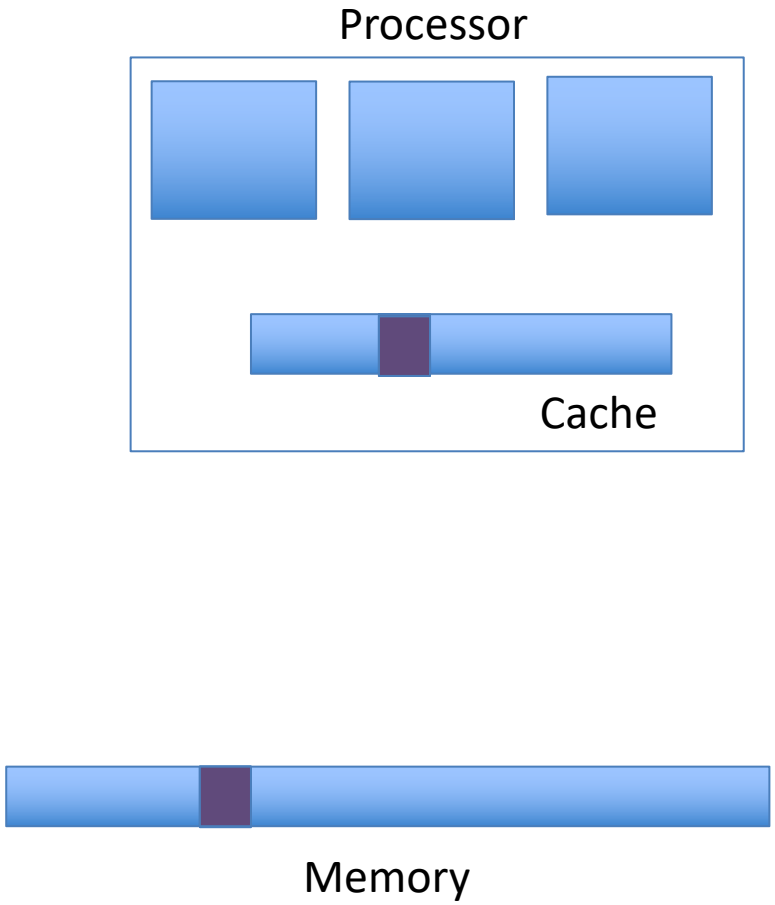
Broad types of resource sharing

- **Contention based:** *A* can see resource has been used by *B*, e.g., cache
- **Data (state) reuse based:** data loaded by *A* is used by *B*, e.g., branch predictor

	Shared Between Cores	Shared Hyperthreads	Contention Channel	Data (State) Reuse Channel
Local Caches (L1 & L2)	No	Yes	Yes	No
Last Level Cache (L3)	Yes	Yes	Yes	No
Branch Predictor	No	Yes	Yes	Yes
Prefetchers	No	Yes	Yes	Yes?
Execution Units	No	Yes	Yes	No

The x86 cache

- Memory is slower than the processor
- The cache utilizes locality to bridge the gap
  - Divides memory into *lines*
  - Stores recently used lines
- Shared caches improve performance for multi-core processors



Types of interference

- Assume process A and process B execute on a single core and share caches

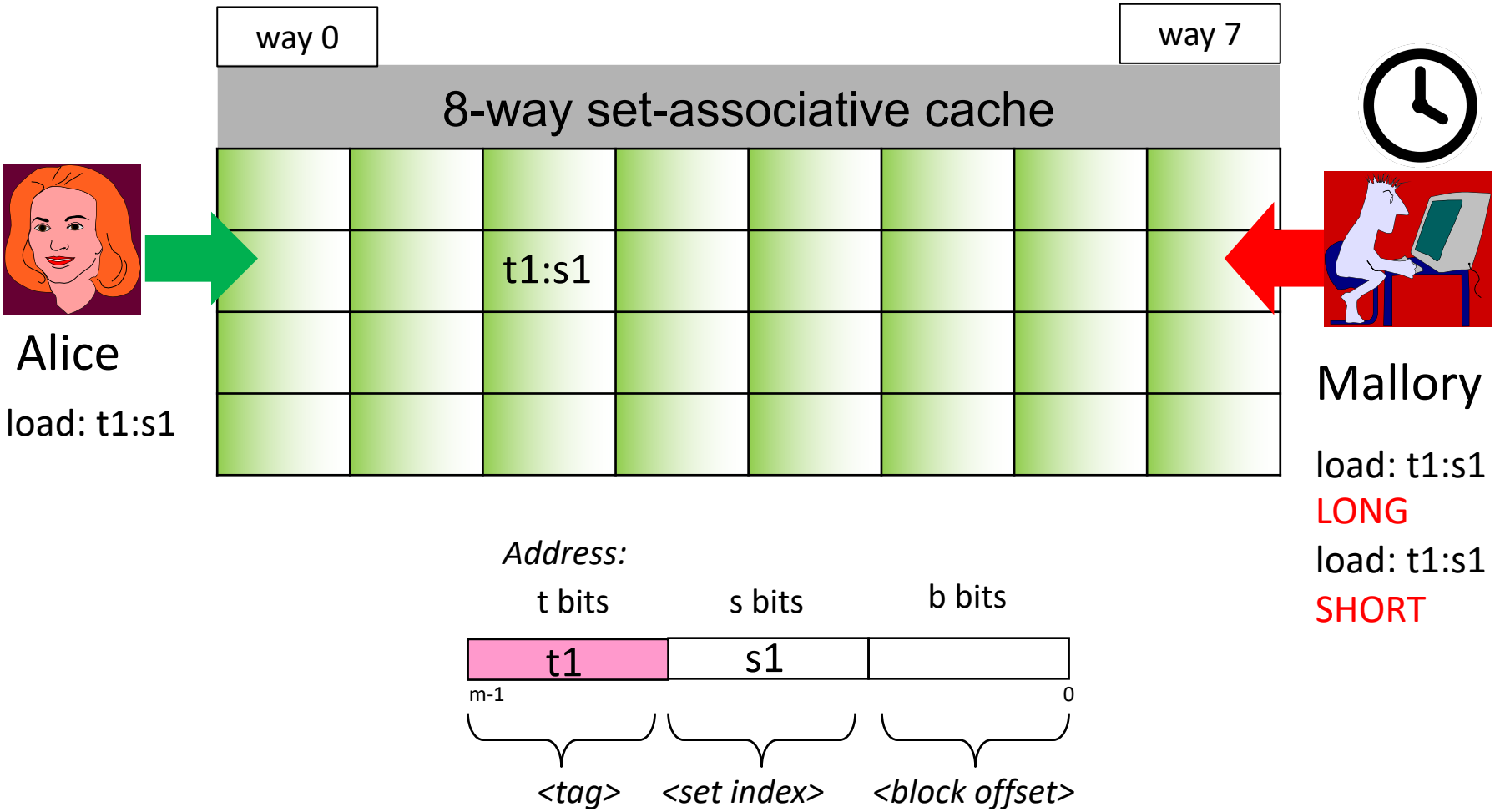
- If data is *shared*
  - Data loads performed by A will be visible to B
  - Removing data from cache by A will affect execution time of B

- If data is *not shared*
  - Cache is still shared, thus activity of A can *evict data placed by B*, affecting B's execution time

Shared data/code between victim and attacker? 🤔

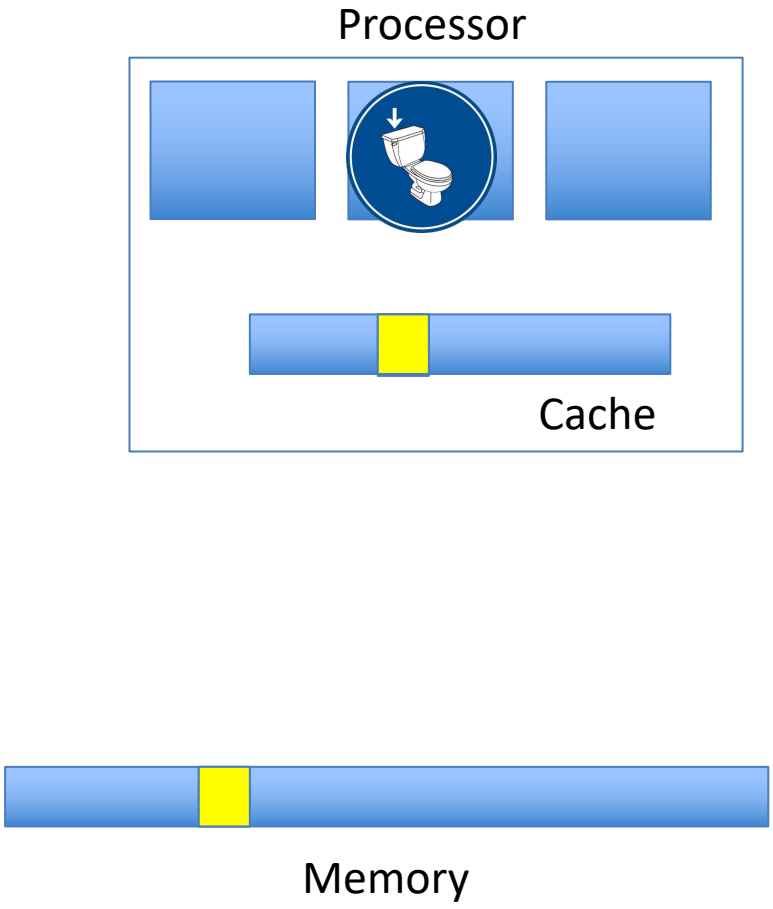
- Very common!
  - E.g. different programs use same libraries (both code and read only data)
  - OS/VMM tries to save physical memory my doing memory deduplication
- As a result:
  - Encrypt/Decrypt operation code is shared (RSA side channel)
  - Precomputed lookup tables (AES side channel)

Shared Data Attack



Cache Consistency

- Memory and cache can be in inconsistent states
  - Rare, but possible
- Solution: Flushing the cache contents
  - Ensures that the next load is served from the memory
  - CLFLUSH instruction
  - The invalidation is broadcast throughout the cache coherence domain



The Flush + Reload Technique

- Exploits cache behavior to leak information on victim access to shared memory.
  - Shared text segments
  - Shared libraries
  - Memory de-duplication

- Spy monitors victim's access to shared code
  - Spy can determine what victim does
  - Spy can infer the data the victim operates on

FLUSH+RELOAD

- **FLUSH** memory line
- Wait a bit
- Measure time to **RELOAD** line
  - slow-> no access
  - fast-> access
- Repeat