Principled Approaches to Constant-Time Cryptography

Marco Vassena

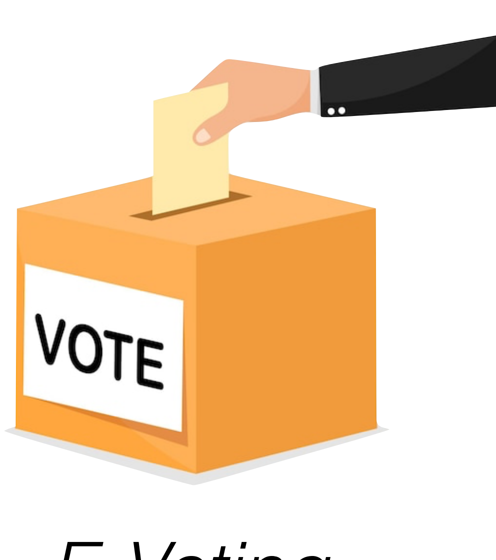
m.vassena@uu.nl

Cryptography

*Cryptography is a **fundamental** mechanism to **secure systems**:*

Digital Currency

Healthcare



E-Voting

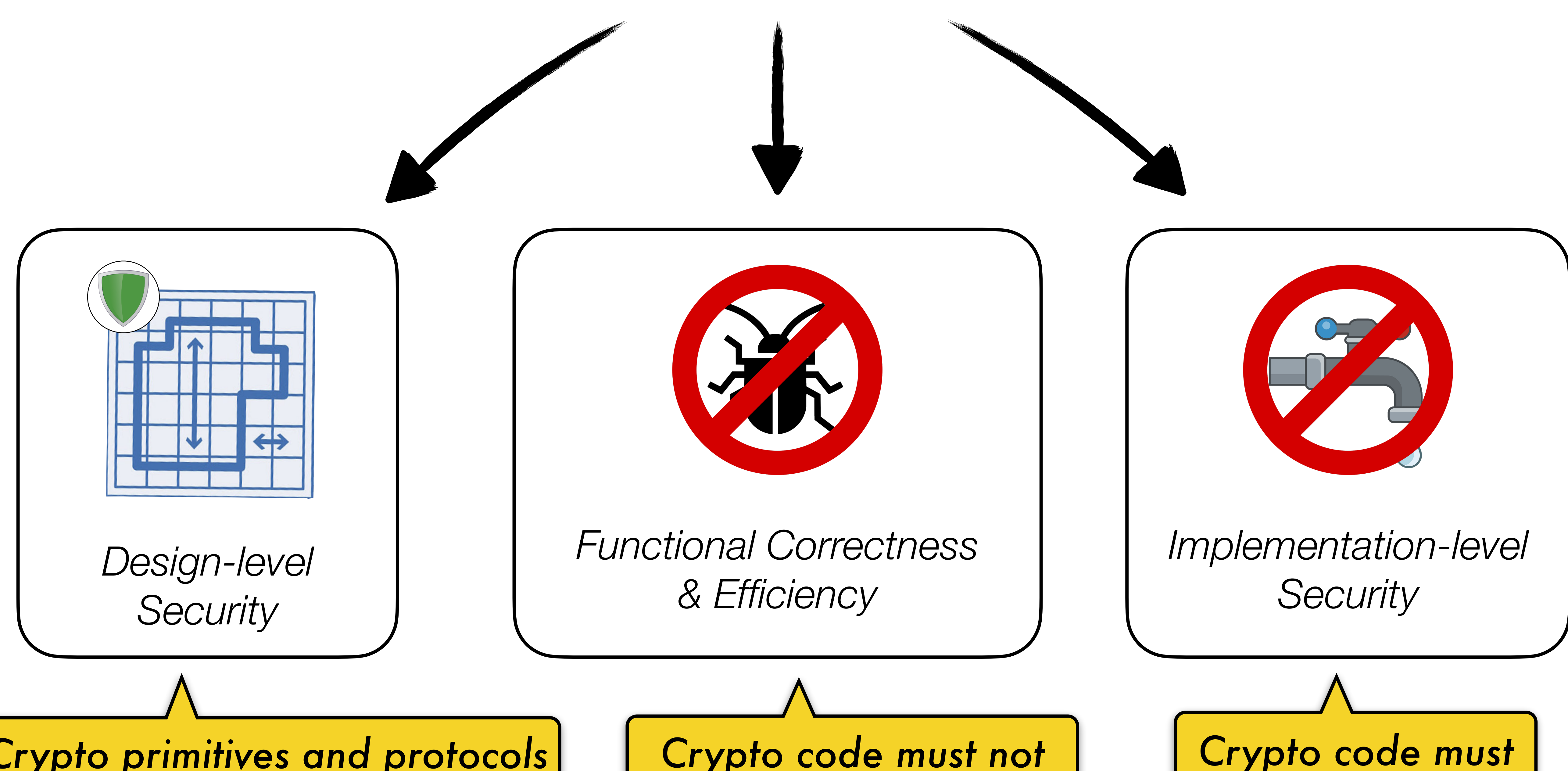
*Cryptographic code runs **everywhere**, on all sorts of devices:*

Smart cards

Internet Browsers

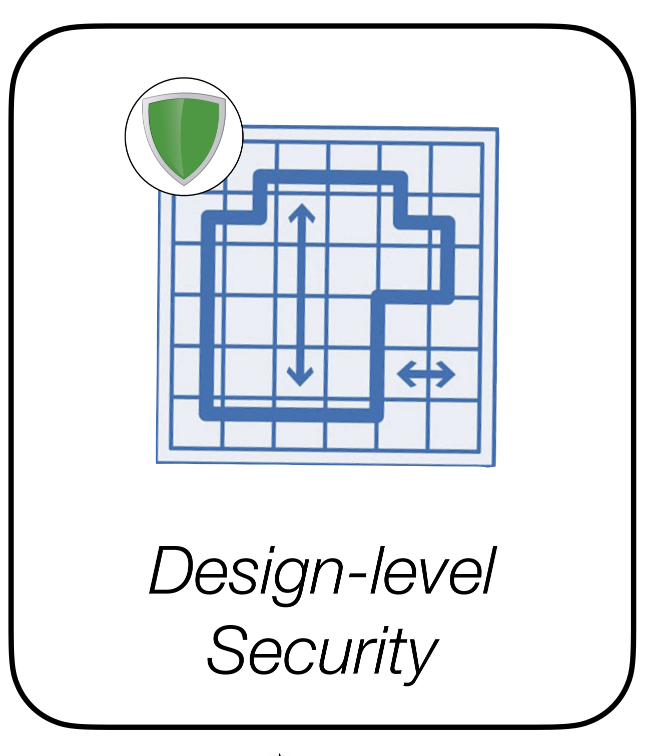
Cloud Servers

Cryptographic mechanisms are ***hard to get right****!*



This Lecture

*Cryptographic mechanisms are **hard to get right**!*

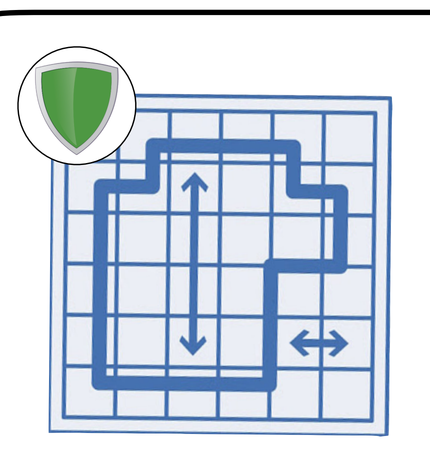


Crypto primitives and protocols must be theoretically secure

Crypto code must not have bugs!

Crypto code must not leak secrets!

The need for Implementation Security



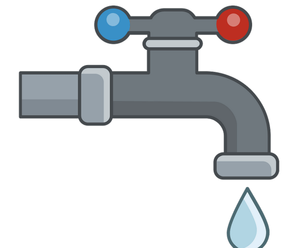
Abstract mathematical functions

AES cipher is ***computationally secure*** *against brute-force attacks*

On average, 10^{18} years for a 128-bit key

Input-output behavior of the code corresponds to AES cipher

The code of OpenSSL version 0.9.7a implements AES cipher ***correctly***

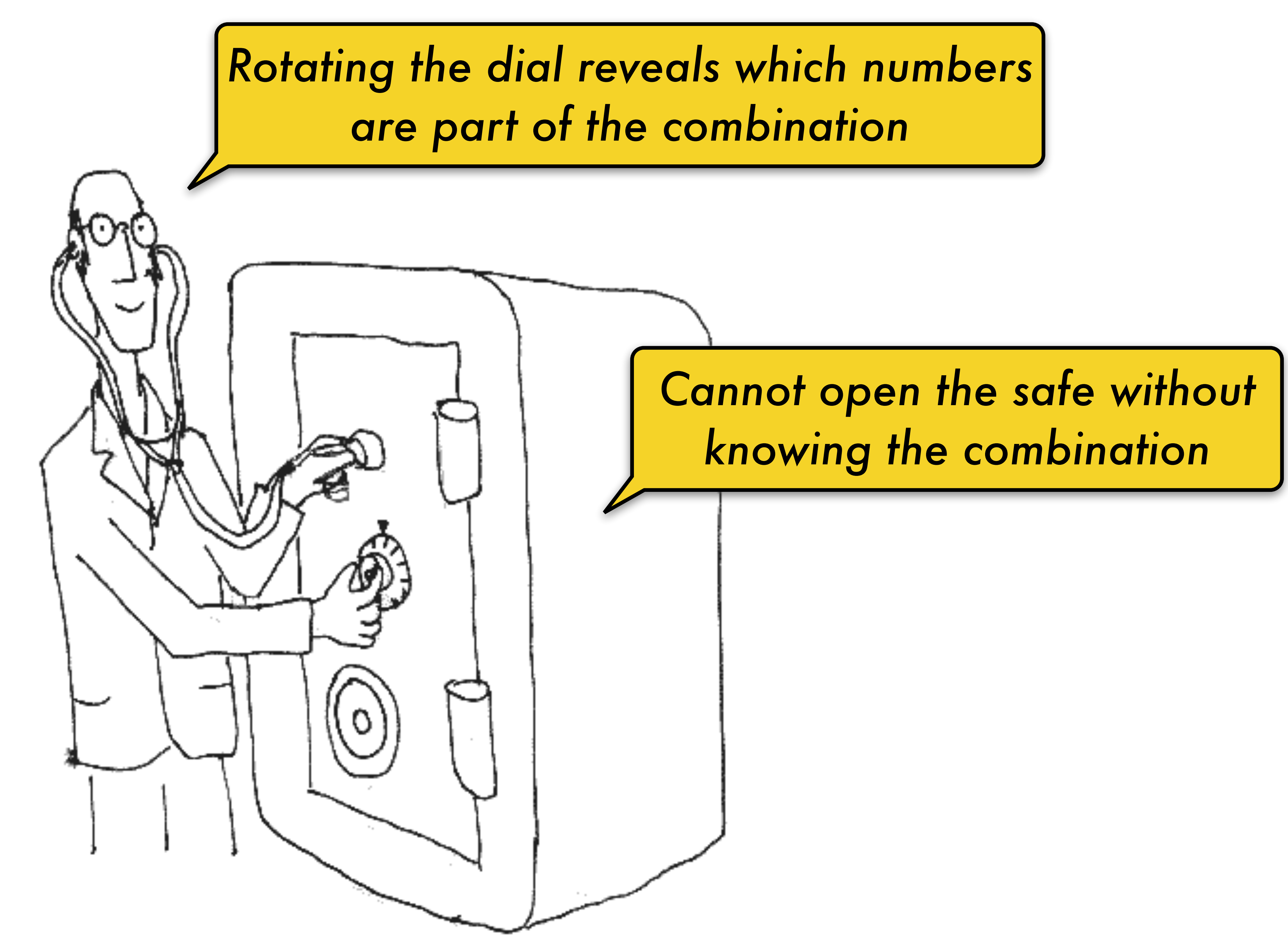


Attacker can observe more than input-output behavior!

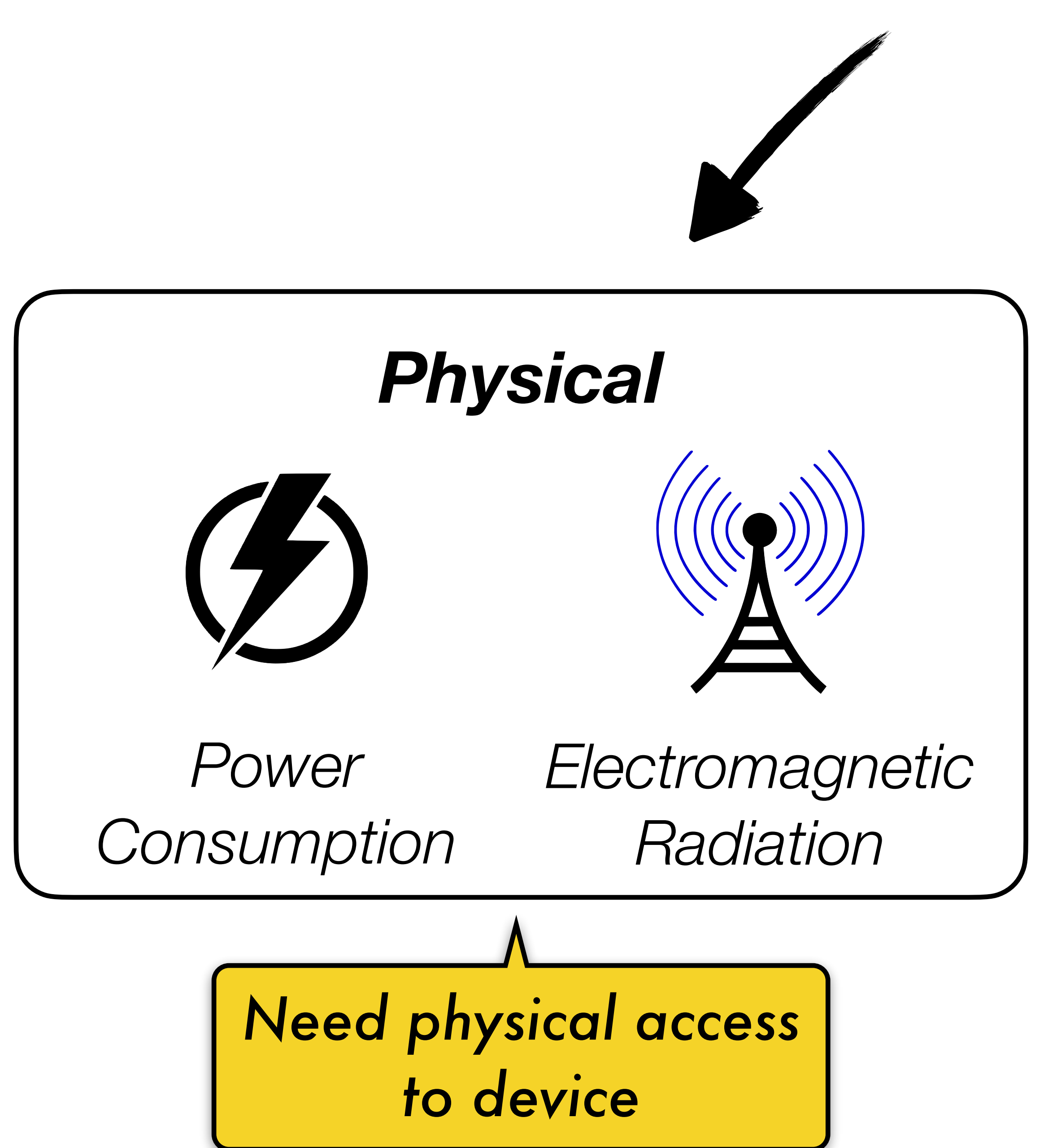
That implementation is vulnerable to ***side-channels attacks***

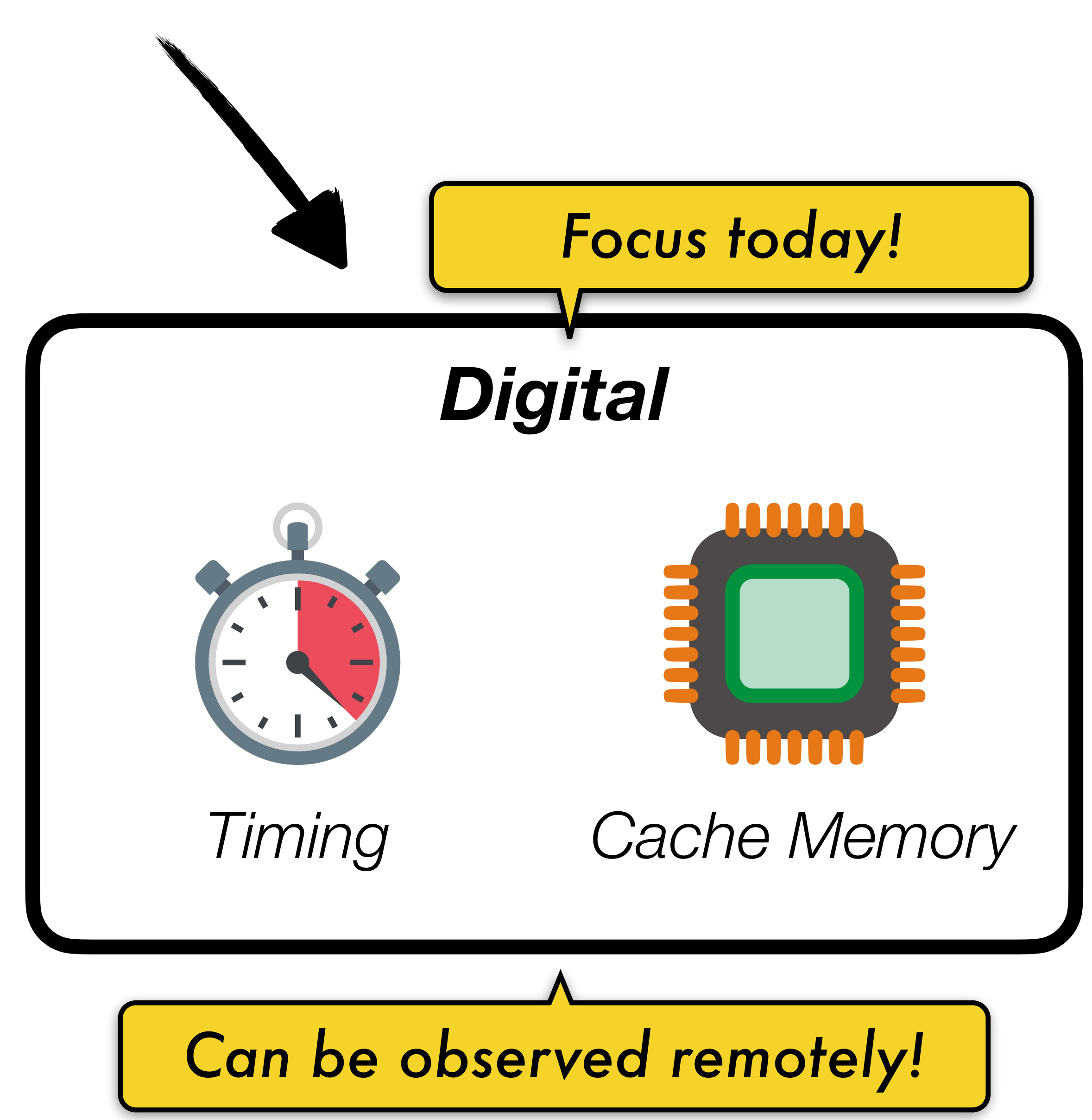
Attacker can reconstruct the secret key in one minute!

Intuition for Side-Channel Attacks



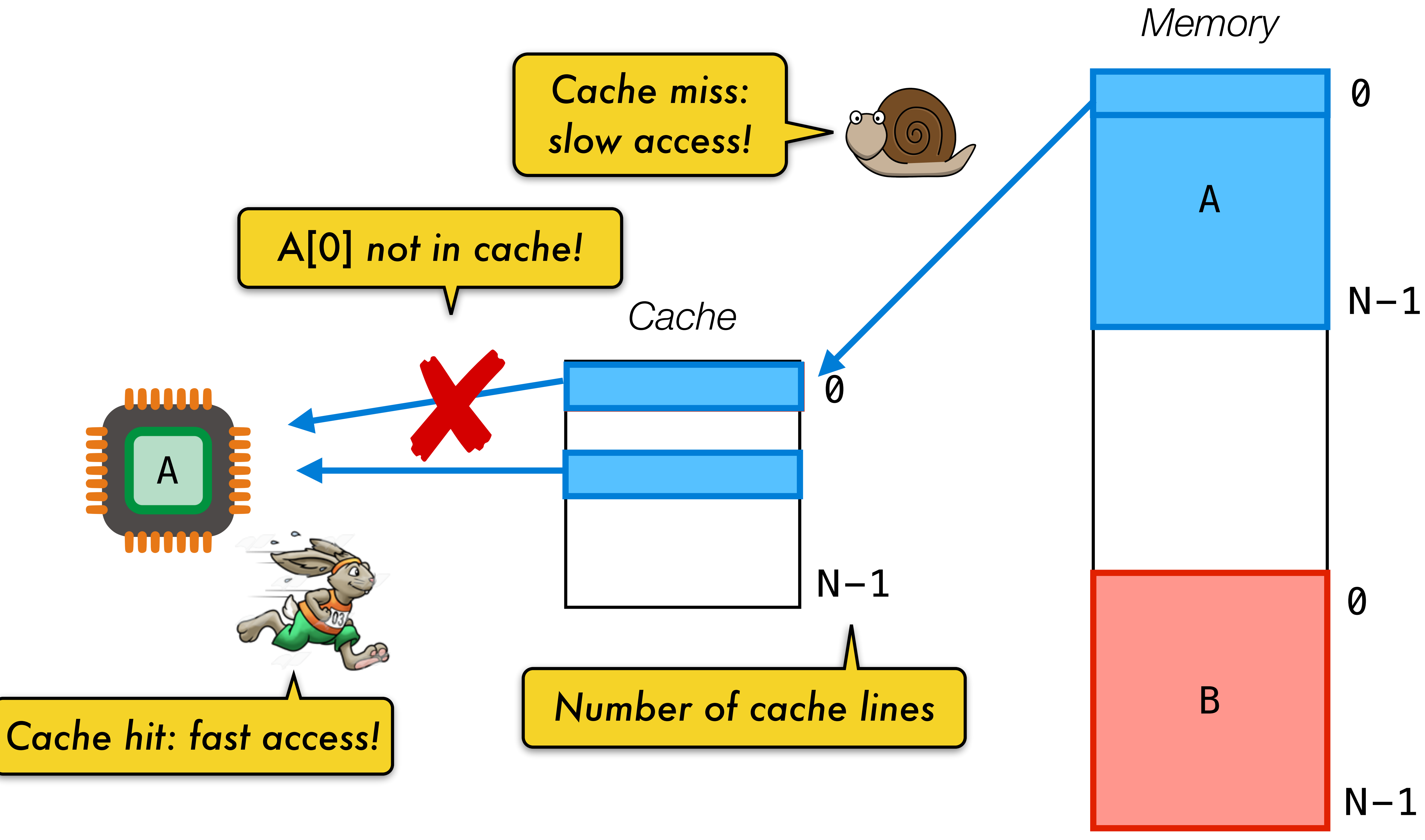
Side Channels in Computer Systems





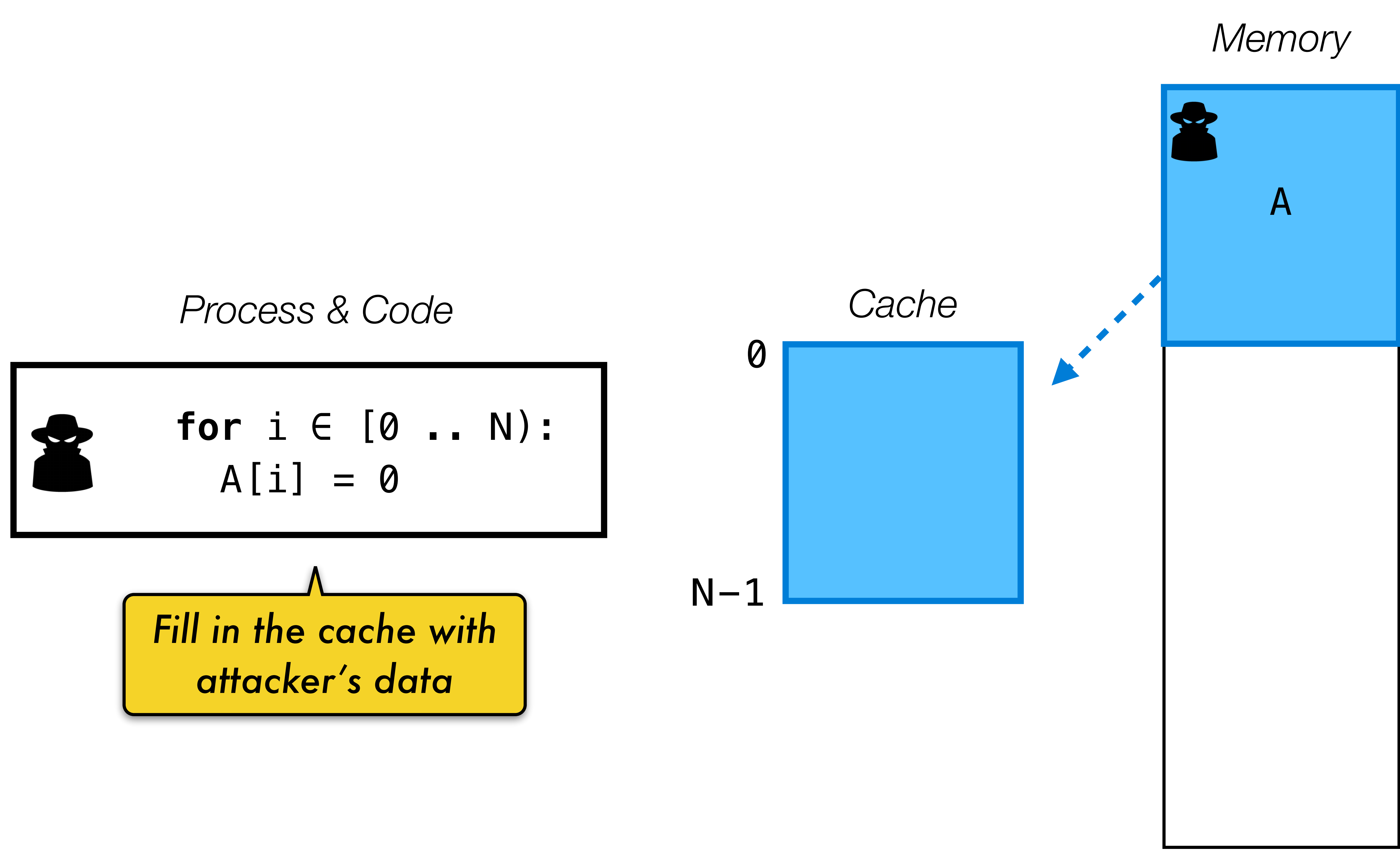
Cache Memory

*Fast memory **shared** between different processes*



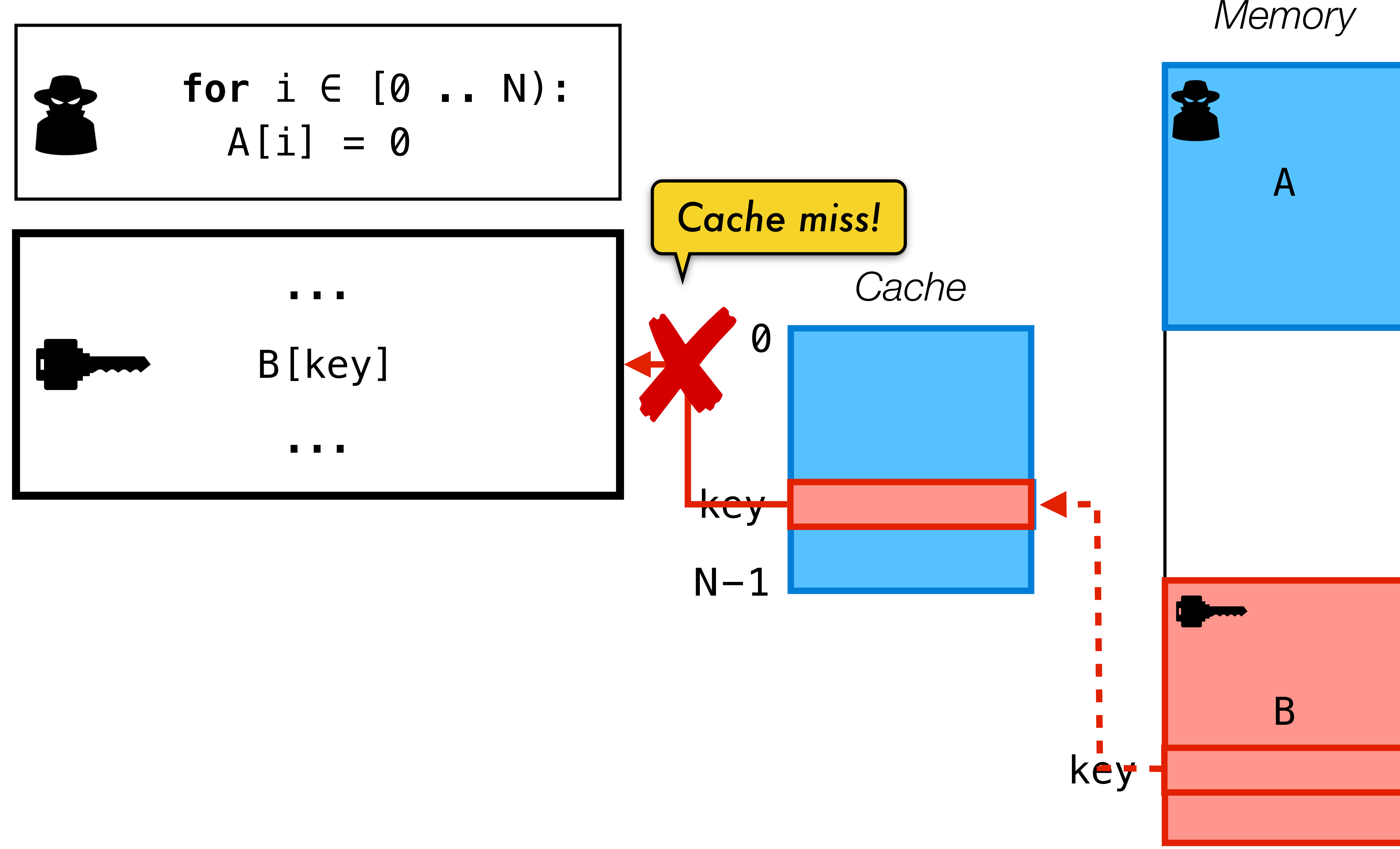
Cache Side-Channel Attacks

*The **memory-access pattern** leaks secret information:*



Cache Side-Channel Attacks

*The **memory-access pattern** leaks secret information:*



Cache Side-Channel Attacks

*The **memory-access pattern** leaks secret information:*

```
for i ∈ [0 .. N):
  A[i] = 0
```

```
...
B[key]
...
```

Evict attacker's data from the cache

Cache

0

key

N−1

Memory

A

B

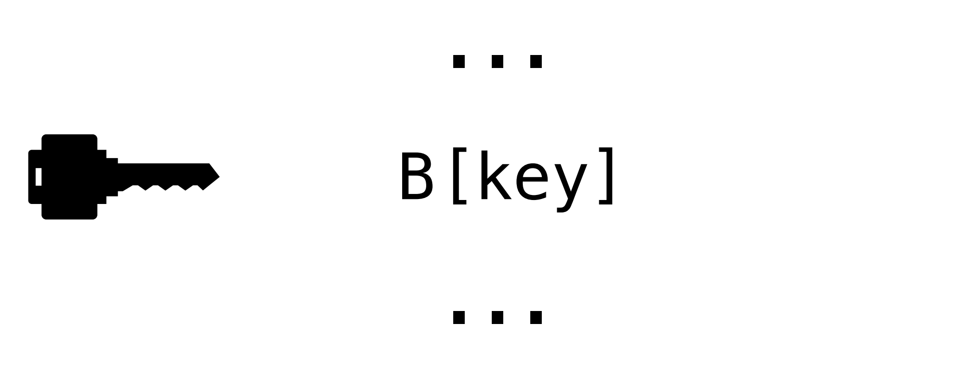
key

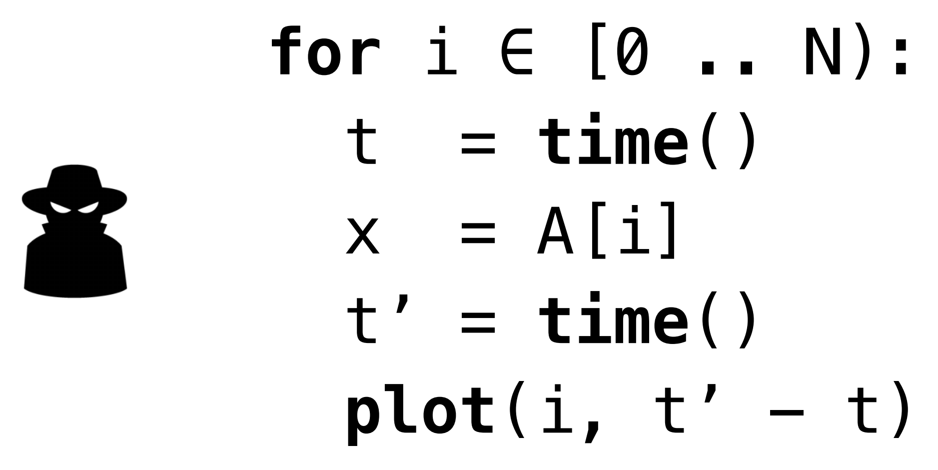
Cache Side-Channel Attacks

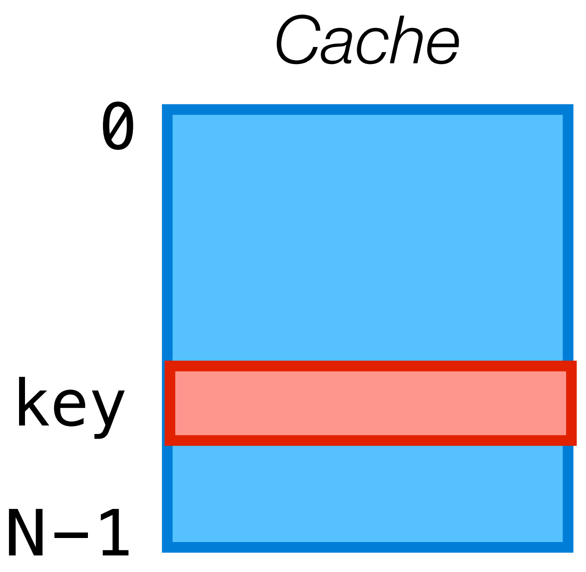
*The **memory-access pattern** leaks secret information:*

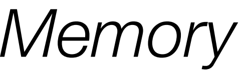
Process & Code

```
for i ∈ [0 .. N):
  A[i] = 0
```

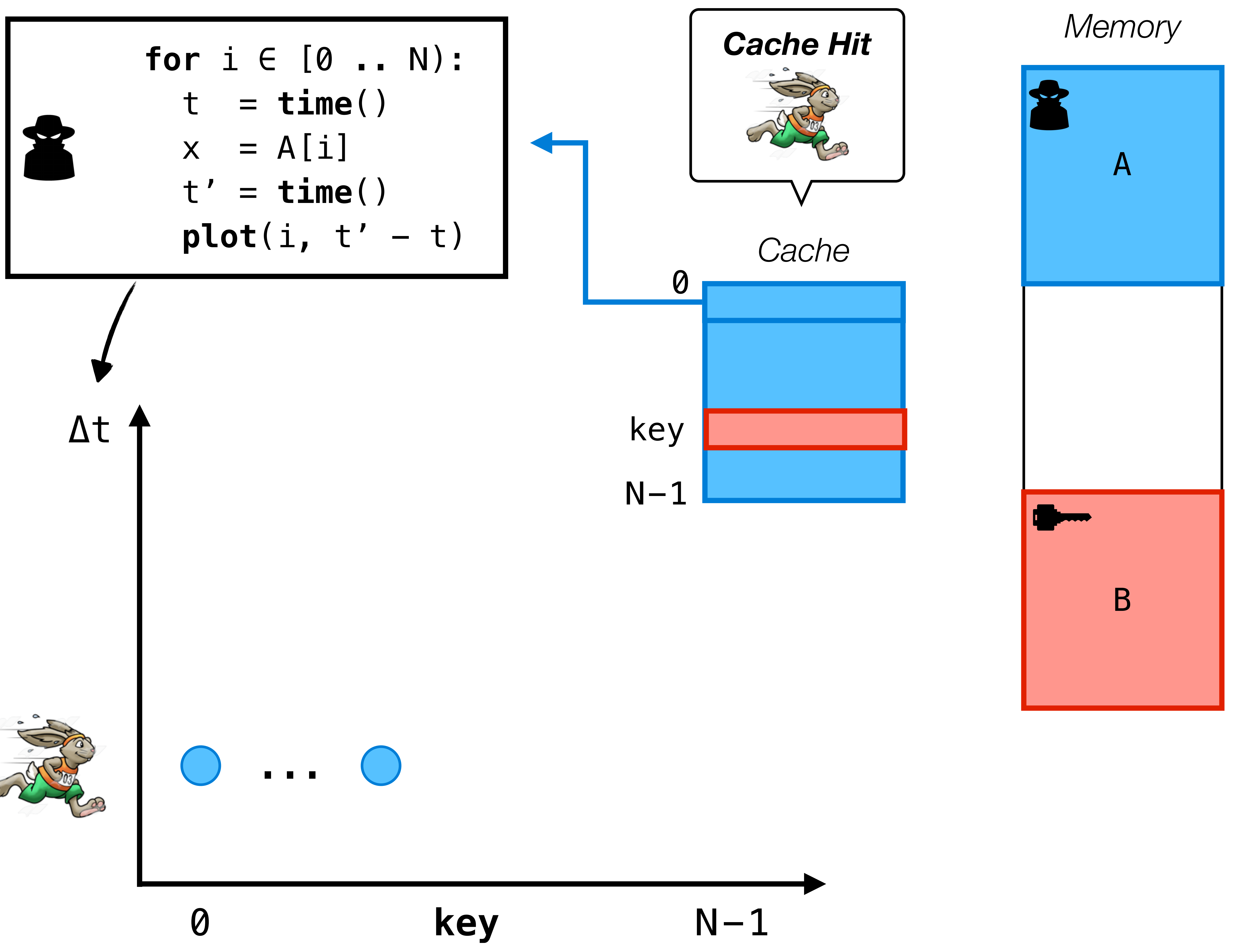


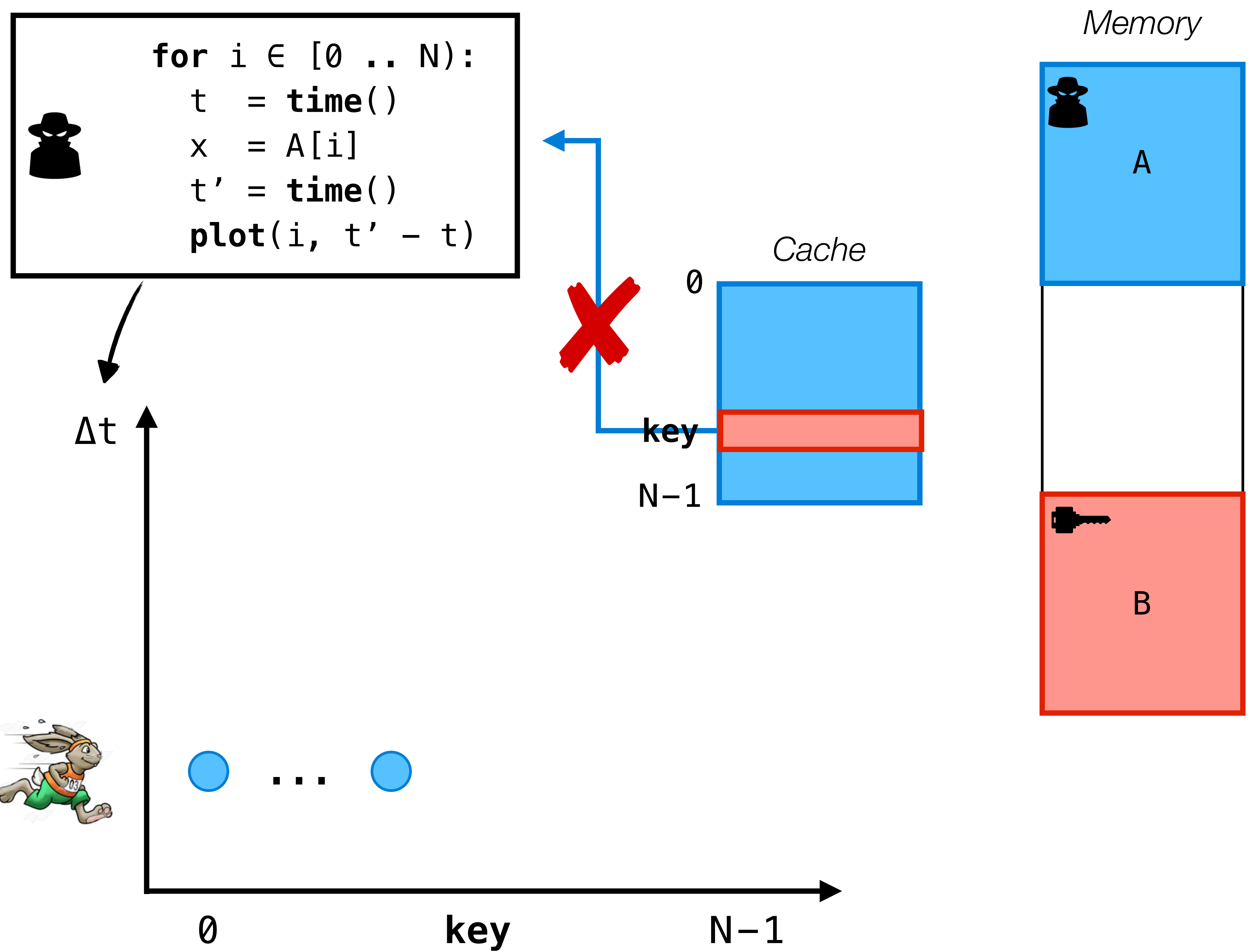


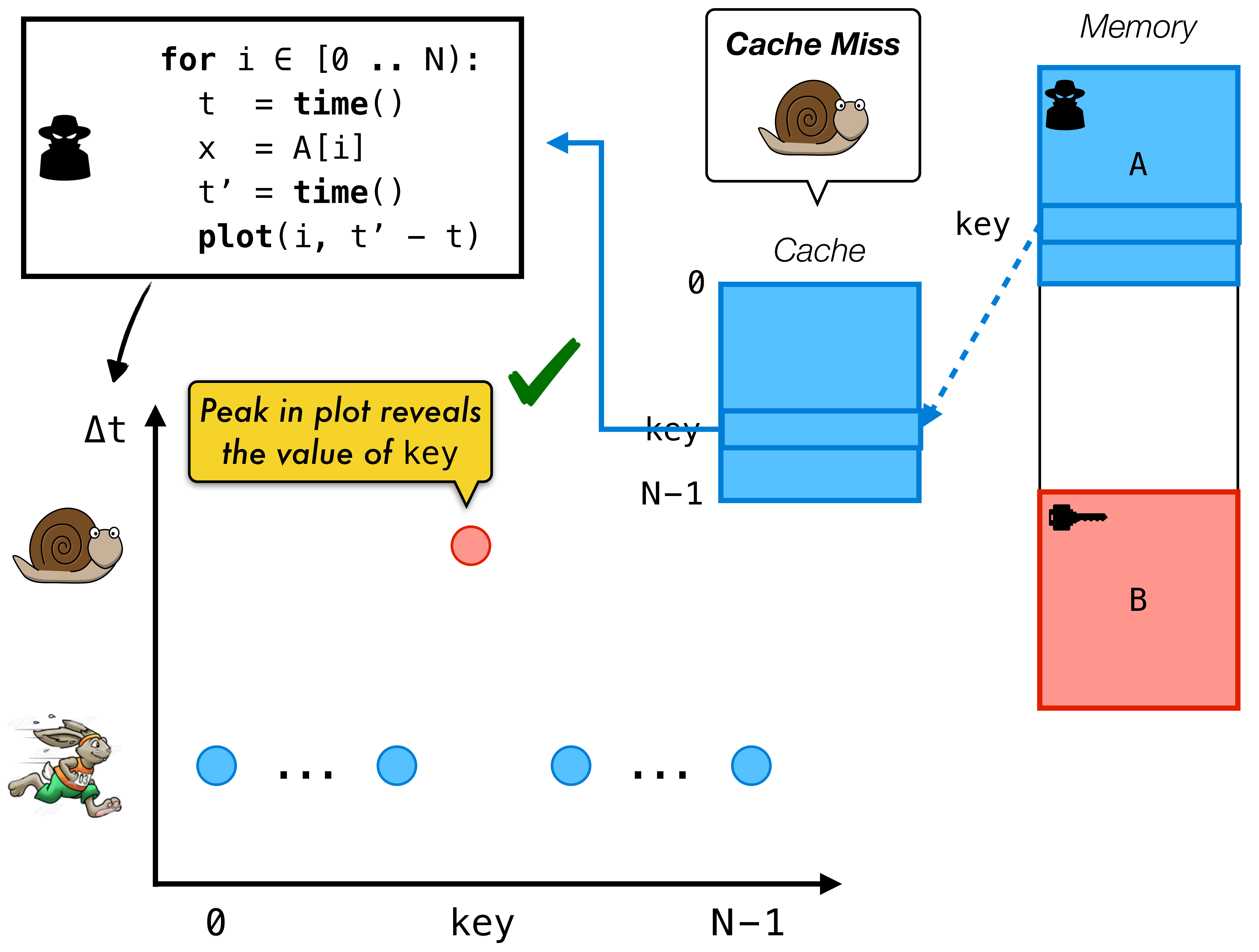




Recover key from memory-access time


for i ∈ [0 .. N):
t = time()
x = A[i]
t' = time()
plot(i, t' − t)
Cache Hit
Memory
A
Cache
0
key
N−1
B
Δt
...
0
key
N−1


for i ∈ [0 .. N):
t = time()
x = A[i]
t’ = time()
plot(i, t’ − t)
Memory
A
Cache
0
key
N−1
B
Δt
...
03
0
key
N−1


for i ∈ [0 .. N):
t = time()
x = A[i]
t' = time()
plot(i, t' − t)
Cache Miss
Memory
A
key
Cache
0
key
N−1
B
Peak in plot reveals the value of key
Δt
...
...
0
key
N−1

Timing Side-Channel Attacks

*The **execution time** of a computation leaks secret information:*

E.g., in user login prompt

```
compare(guess, secret, size) {
  for(i = 0; i < size; i++)
    if guess[i] != secret[i]
      return false;
  return true;
}
```

Observe only input-output behavior

Brute-force attack

$O(2^{size})$ =

*Execution time **correlates** with number of iterations & comparison*

***Measure** the execution time with **known** `secret` and `guess`*

```
compare(guess, secret, size) {
  for(i = 0; i < size; i++)
    if guess[i] != secret[i]
      return false;
  return true;
}
```

Execution time is proportional to number of guessed characters

guess	secret	
"0000"	"1234"	$T_0 = C$
"1000"	"1234"	$T_1 = I + C$
"1200"	"1234"	$T_2 = 2I + C$
"1230"	"1234"	$T_3 = 3I + C$

With enough measurements, the attacker can determine parameters **I** *and* **C**

*Now we can extract **unknown secrets** from timing alone:*

Execution time

= N * I + C

Number of iterations & guessed characters

For an arbitrary guess and secret,
compute the number of guessed character as:

Measured execution time for guess

$$N = \frac{⏱ - C}{I}$$

The first N characters of guess and secret are the same!

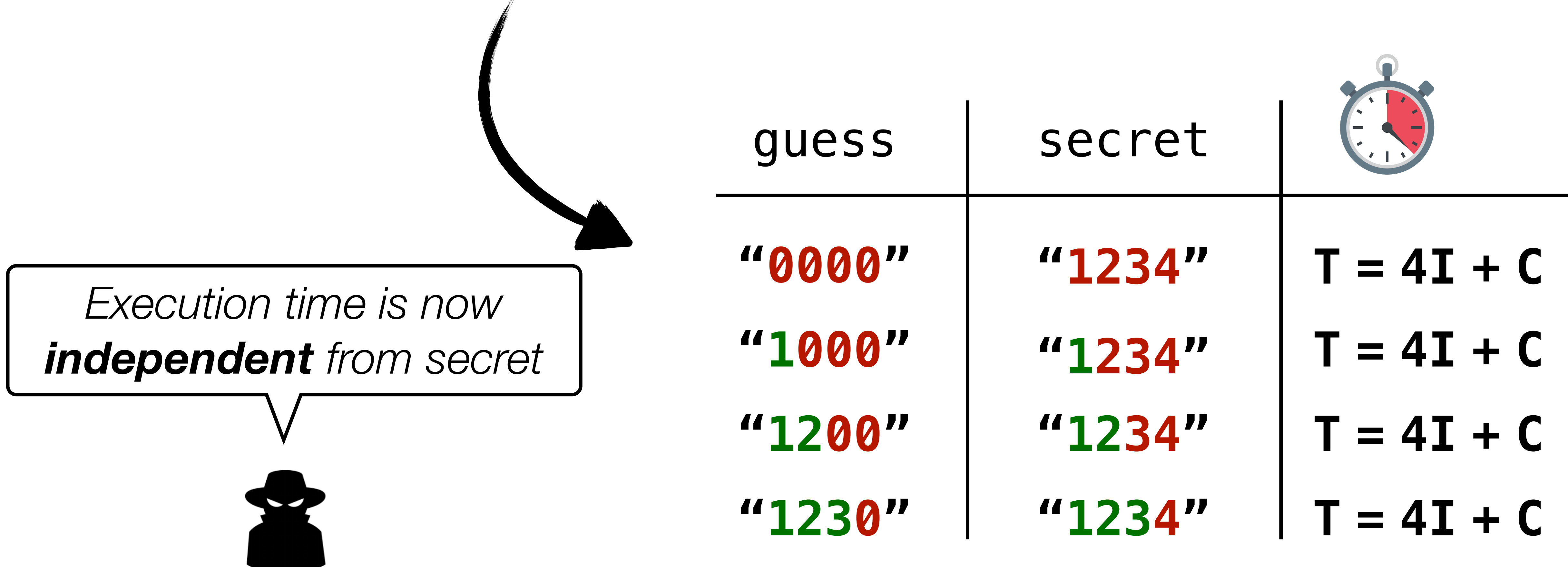
Timing attack

O(size) =

To avoid timing channels, ***control-flow*** *should not depend on secrets!*

```
ct_compare(guess, secret, size) {
  result = false;
  for(i = 0; i < size; i++)
    result = result || (guess[i] != secret[i])
  return result;
}
```

Always executes size iterations!



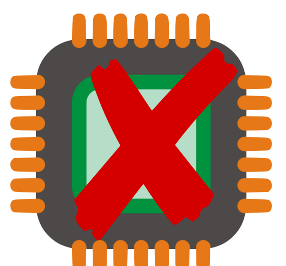
guess	secret	⏱
"0000"	"1234"	T = 4I + C
"1000"	"1234"	T = 4I + C
"1200"	"1234"	T = 4I + C
"1230"	"1234"	T = 4I + C

Constant-Time Discipline

*To avoid side-channels, write code following the **constant-time discipline***

*Constant-time code has **no secret-dependent***

No leaks via the cache



*1) **Memory accesses***

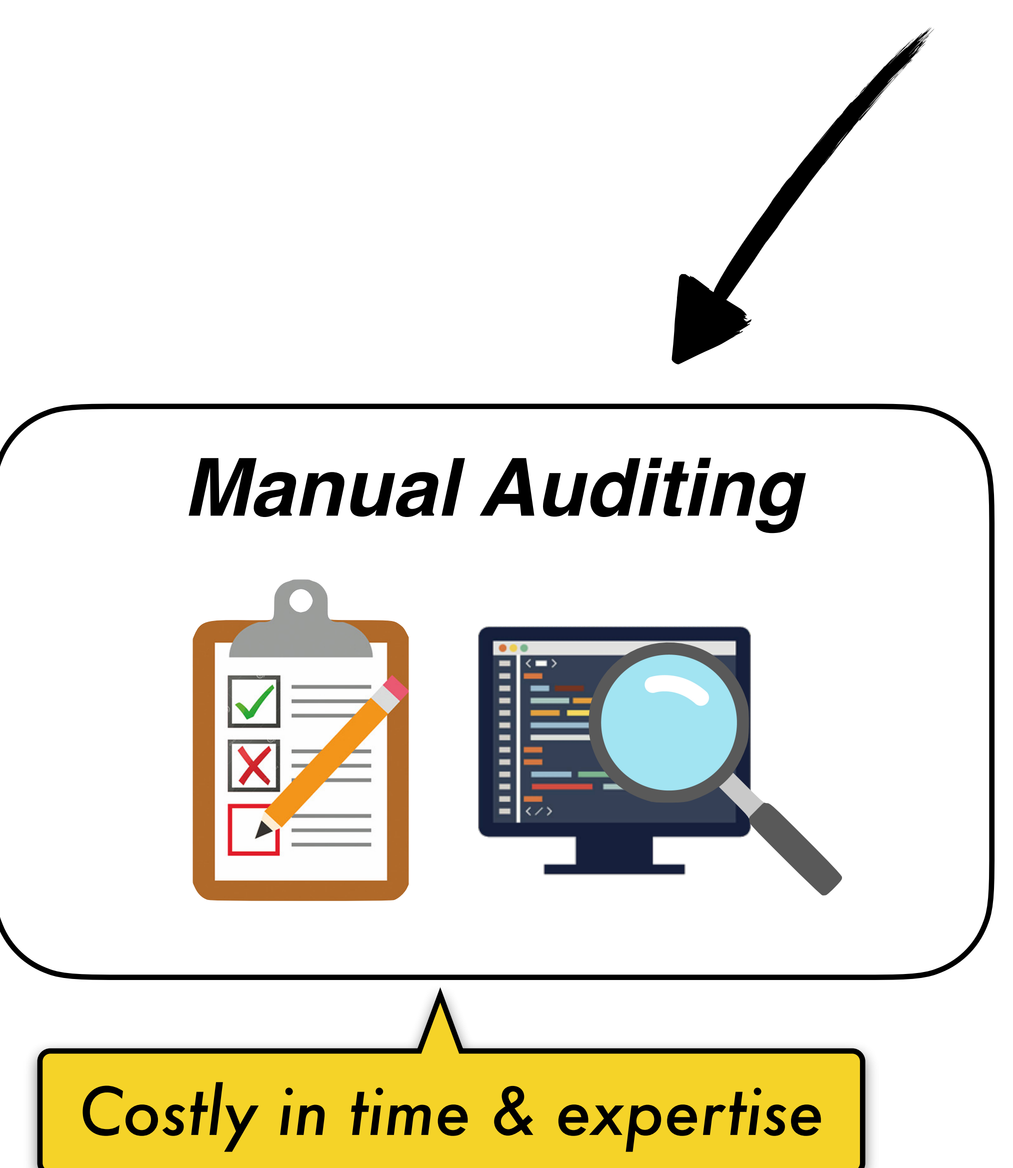
*2) **Control-flow instructions***

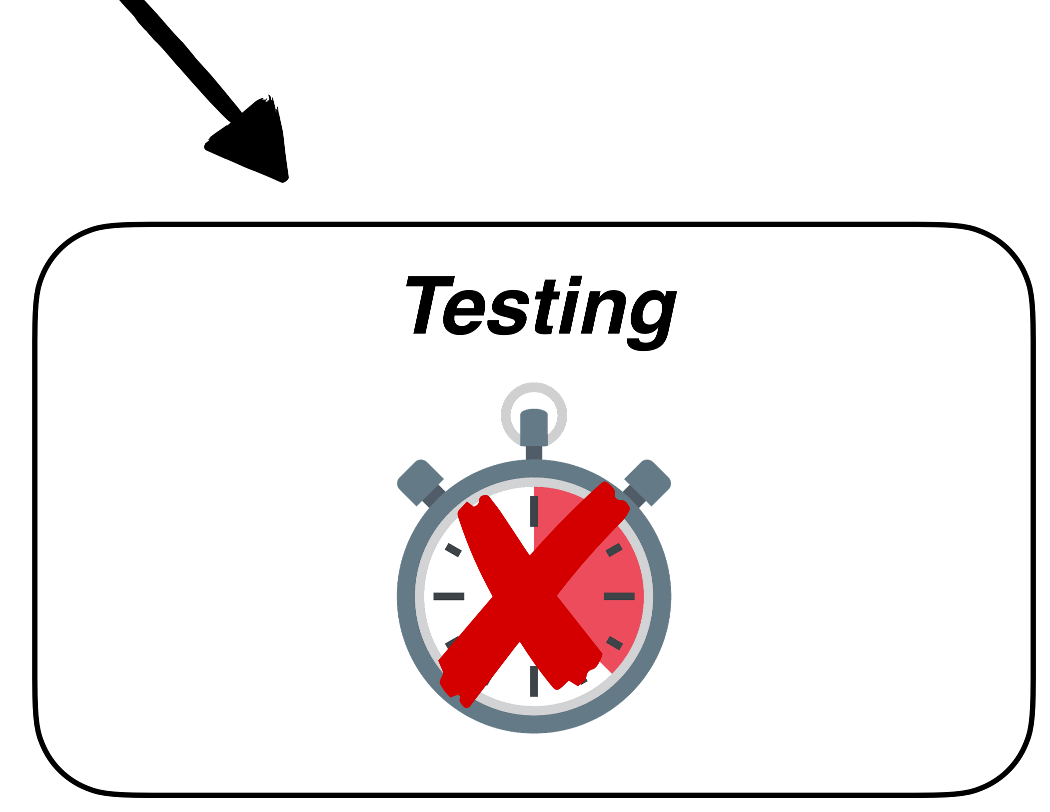
*3) **Variable-time operations***

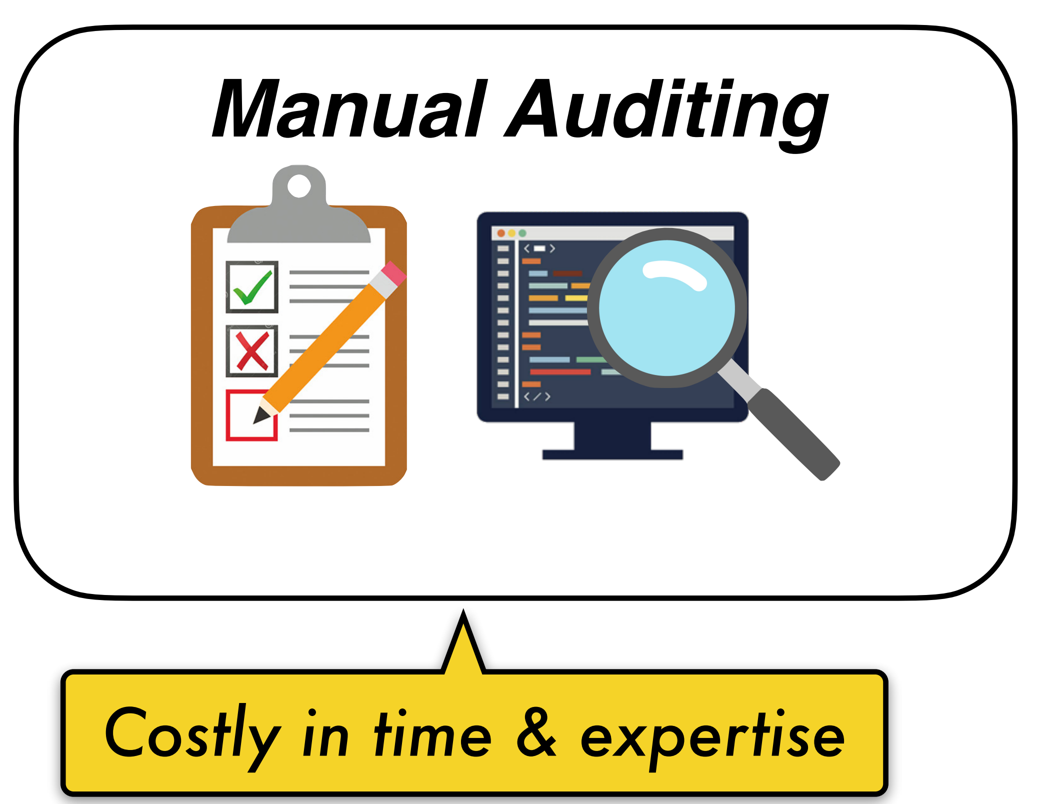
No leaks via timing

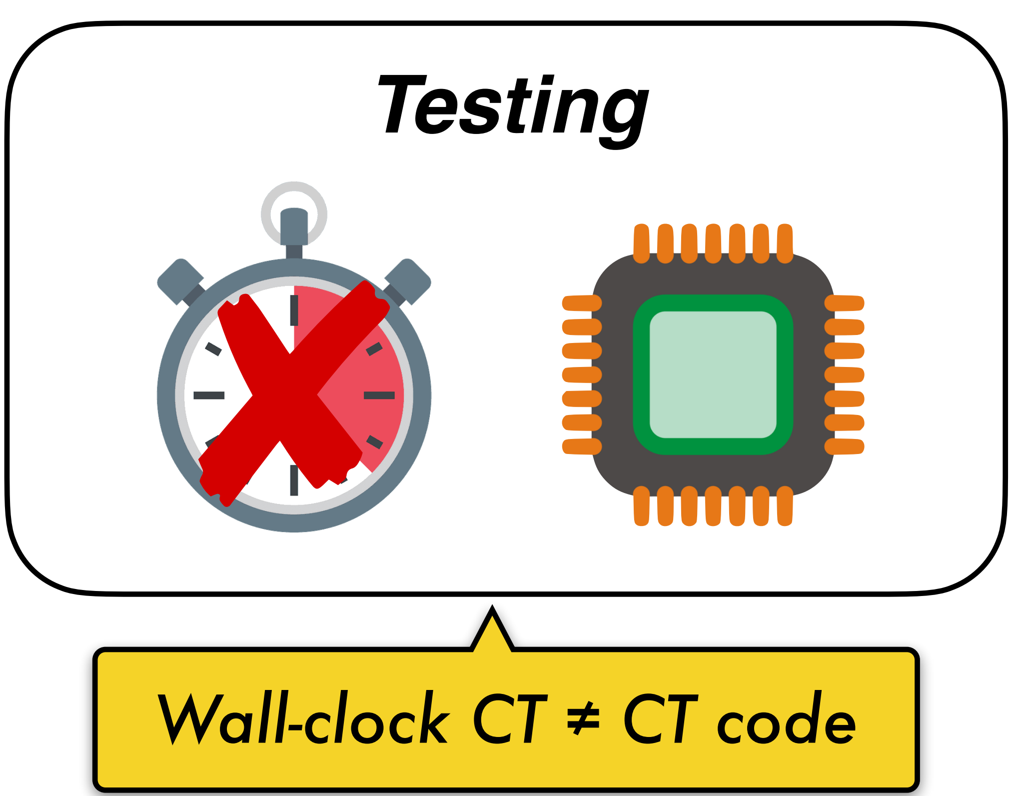
E.g., Floating point operations, division & modulus

Current approaches to CT









These approaches are ***inadequate:***

Vulnerabilities *in* ***TLS 1.0***

CVE

Patch

Have we patched all of them?

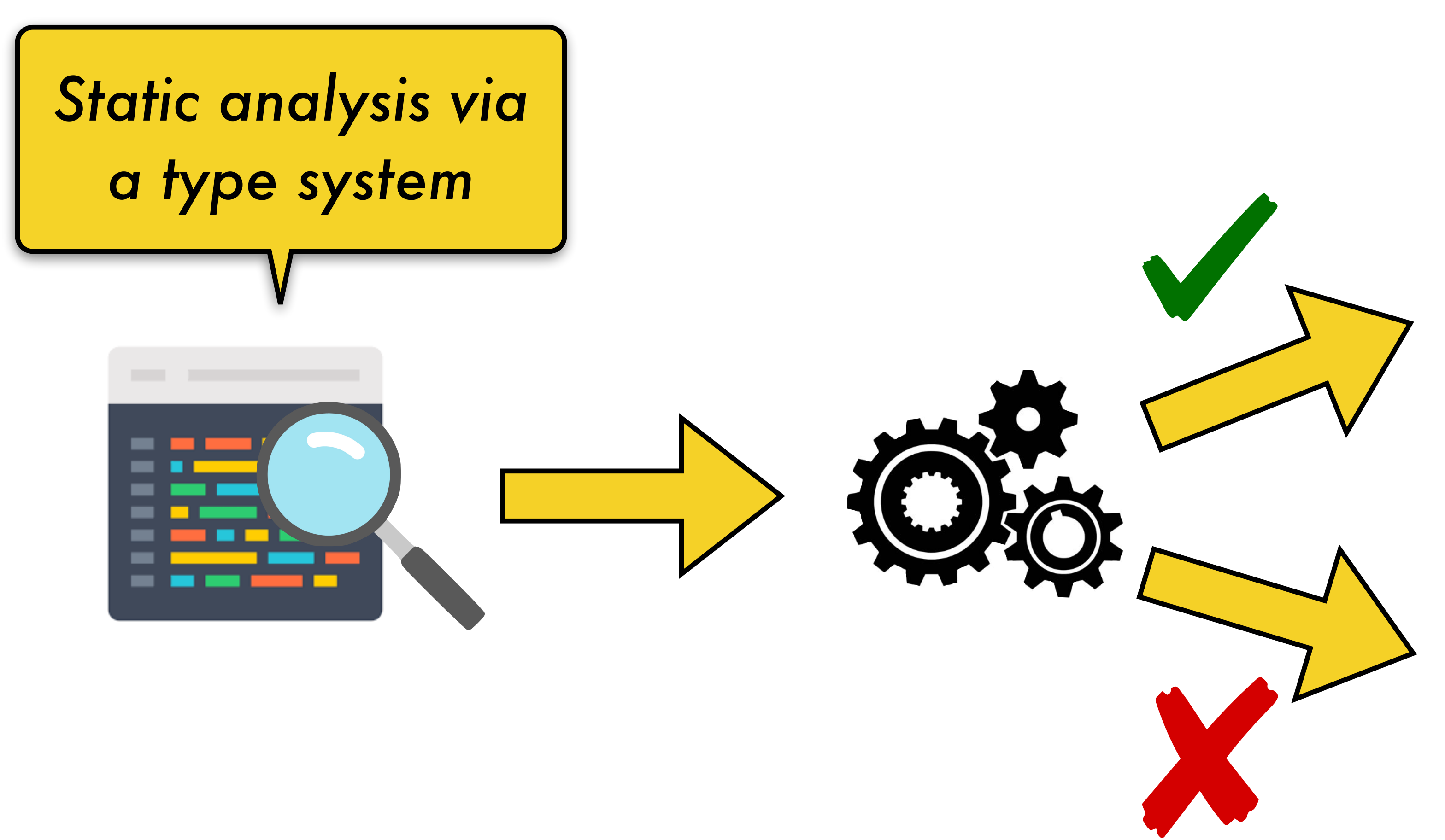
Timing Vulnerability (2004)

Lucky 13 Attack (2013)

Timing Vulnerability (2017)

Program Analysis to Rescue

Detect code that violates the constant-time discipline ***automatically***



Code is ***not*** *constant-time and* ***may*** *leak*

Bibliography

- SoK: Computer-Aided Cryptography, Barbosa et al., IEEE Symposium on Security and Privacy, 2021 (esp. Section IV).
- Crypto implementations are vulnerable to side-channel attacks:
  - Remote timing attacks are practical, Brumley & Boneh, USENIX Security, 2003.
  - Cache-timing attacks on AES, Daniel J. Bernstein, 2005.
- Recent surveys on cache- and timing-side channel attacks
  - A Survey of Microarchitectural Side-channel Vulnerabilities, Attacks and Defenses in Cryptography, Lou et al., ACM Computing Surveys, 2022.
  - Winter is here! A decade of cache-based side-channel attacks, detection & mitigation for RSA, Mushtaq et al., Information Systems 2020.
  - A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware, Ge et al., Journal of Cryptographic Engineering, 2018.
- System-level Non-interference for Constant-time Cryptography, Barthe et al., SIGSAC Conference on Computer and Communications Security, 2014.
- Secure Compilation of Side-Channel Countermeasures: The Case of Cryptographic “Constant-Time”, Barthe et al., IEEE Computer Security Foundations Symposium 2018.