Concurrency
7PK - Time and State

JOÃO PAULO BARRACA

Concurrency

Current operational environments are typically distributed:

- composed by multiple systems
- distributed information/resources
- distributed clock

Algorithms are frequently developed as actions that use those resources

- Commonly mapped to sequential actions

However… interactions/side effects between systems can make some "naive" assumptions false

- Information is available on demand
- Writing to an object makes it available to other components
- Time flows in a single direction

Concurrency

Even if algorithms account for the distributed execution, side effects may be present

- Variable access latency
- Variable execution paths
- Variable object metadata
- Variable locks

Side effects can affect execution or disclose information between domains

- How an algorithm is implemented, what it is doing
- Keys used
- Users currently active

CWE-361 - 7PK - Time and State

.. improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.

Distributed computation is about time and state.

- in order for more than one component to communicate, state must be shared, and all that takes time.
- **… programmers anthropomorphize their work**. They think about one thread of control carrying out the entire program **in the same way they would if they had to do the job themselves**.
- Modern computers switch between tasks very quickly, and in multi-core, multi-CPU, or distributed systems, **two events may take place at exactly the same time**.
- … unexpected interactions between threads, processes, time, and information.
- These interactions happen through shared state: semaphores, variables, the file system, and, **basically, anything that can store information**

Basic Time Related CWEs

CWE-362 - Concurrent Execution using Shared Resource with Improper Synchronization

1. The program contains a code sequence that can run concurrently with other code
2. and the code sequence requires temporary, exclusive access to a shared resource
3. but a timing window exists in which the shared resource can be modified by another code sequence that is operating concurrently.

CWE-662 - Improper Synchronization

1. The software utilizes multiple threads or processes to allow temporary access to a shared resource that can only be exclusive to one process at a time
2. but it does not properly synchronize these actions
3. which might cause simultaneous accesses of this resource by multiple threads or processes.

CWE-362 – Race Condition

A race condition occurs within concurrent environments and is effectively a property of a code sequence.

- Depending on the context, a code sequence may be in the form of a function call, a small number of instructions, a series of program invocations, etc.

A race condition violates basic properties:

- **Exclusivity** - the code sequence is given exclusive access to the shared resource
  - no other code sequence can modify properties of the shared resource before the original sequence has completed execution.
- **Atomicity** - the code sequence is behaviorally atomic
  - no other thread or process can concurrently execute the same sequence of instructions (or a subset) against the same resource.

CWE-362 – Race Condition

Race condition exists when an "interfering code sequence" can still access the shared resource, violating exclusivity.

Programmers may assume that certain code sequences execute too quickly to be affected by an interfering code sequence; when they are not, this violates atomicity.

- “too quickly” may degrade with time to slower execution as functionality is added
- “too quickly” is system dependent

State Related CWEs

CWE-367: Time-of-check Time-of-use (TOCTOU) Race Condition

1. The software checks the state of a resource before using that resource
2. The resource's state can change between the check and the use in a way that invalidates the results of the check.

- This can cause the software to perform invalid actions when the resource is in an unexpected state.

Basic Side Effects Related CWEs (Covert Channel)

CWE-385: Covert Timing Channel

1. Covert timing channels convey information by modulating some aspect of system behavior over time
2. so that the program receiving the information can observe system behavior and infer protected information

Prevalence

Any concurrent hardware or software (aka, almost anything)

https://www.cvedetails.com/vulnerability-list/cweid-362/vulnerabilities.html

- OS Kernels
- XEN
- Libgcrypt20
- GNOME gvfs
- Eclipse OpenJ9
- GitLab Community and Enterprise Edition
- Firefox
- Elasticsearch
- X86 Intel CPUs
- Snapdragon JPG driver
- …

CWE-362 – Example – Banking

```
$transfer_amount = GetTransferAmount();
$balance = GetBalance();

if ($transfer_amount < 0) {
      FatalError("Bad Transfer Amount");
}

$nb = $balance - $transfer_amount;
if (($balance - $transfer_amount) < 0) {
      FatalError("Insufficient Funds");
}

SetNewBalanceToDB($nb);
NotifyUser("Transfer of $transfer_amount succeeded.");
NotifyUser("New balance: $newbalance");
```

Balance could have changed between these instructions

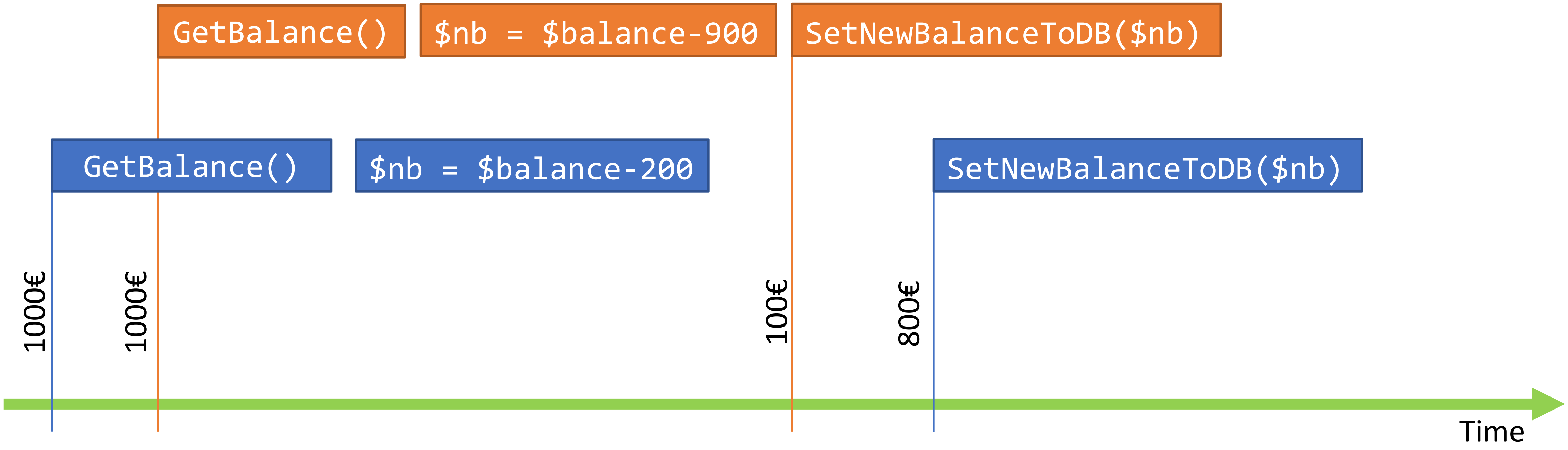
CWE-362 – Example – Banking

```
$transfer_amount = GetTransferAmount();
$balance = GetBalance();

if ($transfer_amount < 0) {
      FatalError("Bad Transfer Amount");
}

$nb = $balance - $transfer_amount;
if (($balance - $transfer_amount) < 0) {
      FatalError("Insufficient Funds");
}

SetNewBalanceToDB($nb);
NotifyUser("Transfer of $transfer_amount succeeded.");
NotifyUser("New balance: $newbalance");
```

Other operations could have happened between these lines. $newbalance is set to a static value

CWE-362 – Example – Banking

Initial balance: 1000€
Total transferred out: 1100€
Final balance: 800€
Result: Bank lost 700€



`GetBalance()` `$nb = $balance-900` `SetNewBalanceToDB($nb)`

`GetBalance()` `$nb = $balance-200` `SetNewBalanceToDB($nb)`

Serializability

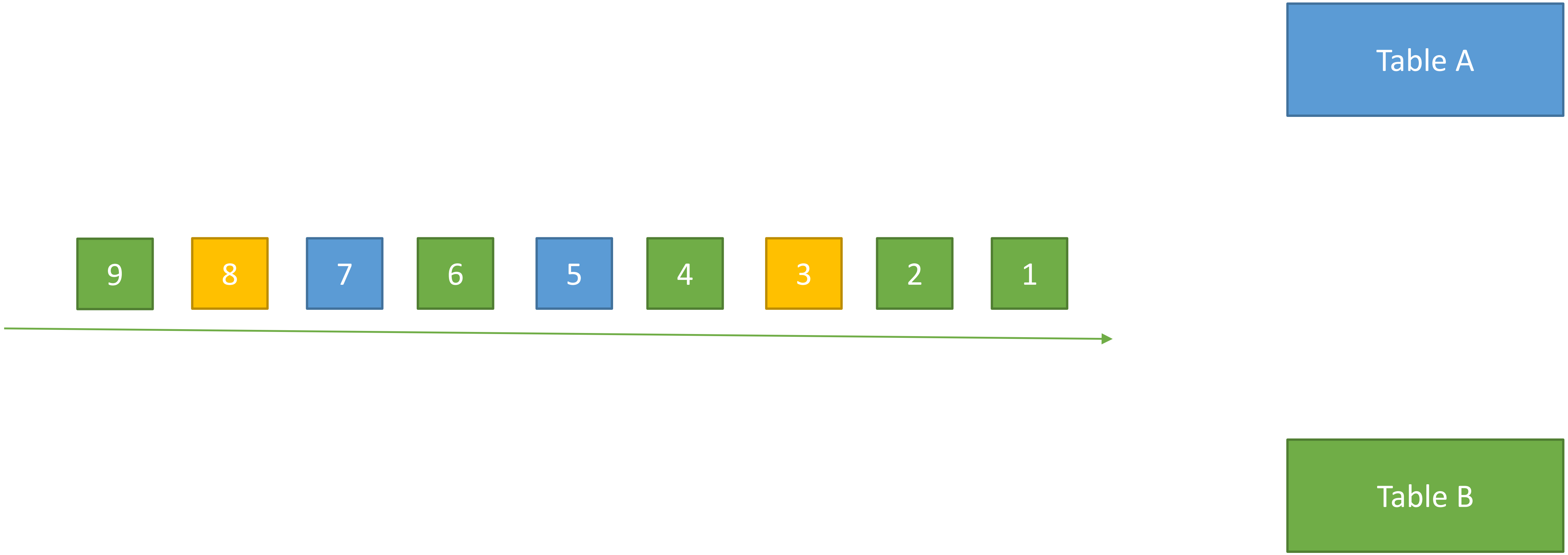
Serializability is the classical concurrency scheme.

- It ensures that a schedule for executing concurrent transactions is equivalent to one that executes the transactions serially in some order.
  - It assumes that all accesses to the database are done using read and write operations.
- A schedule is called ``correct'' if we can find a serial schedule that is ``equivalent'' to it.

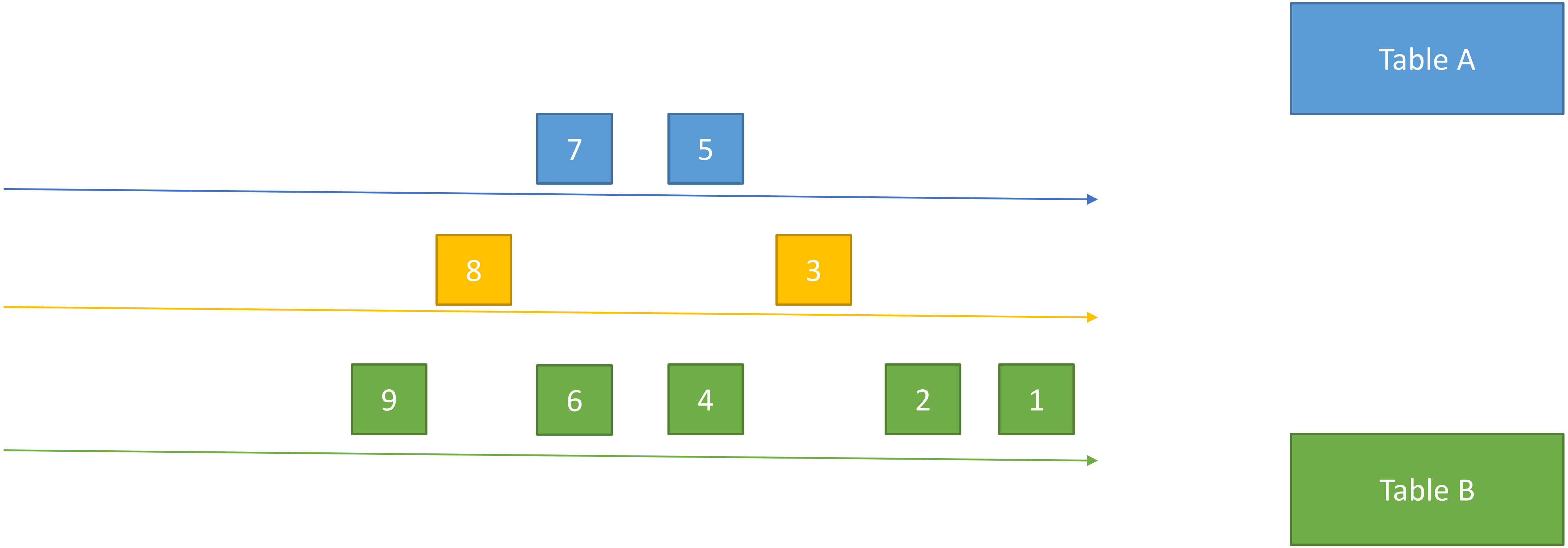
Given a set of transactions, two schedules of these transactions are equivalent if the following conditions are satisfied:

- **Read-Write Synchronization**: If a transaction reads a value written by another transaction in one schedule, then it also does so in the other schedule.
- **Write-Write Synchronization**: If a transaction overwrites the value of another transaction in one schedule, it also does so in the other schedule.

Serializability



Serializability



Database ACID characteristic

Database operation provides ACID characteristics

- Atomicity: All operations either occur or fail and are treated as single instructions
- Consistency: Any rules set (cascades, indexes, triggers) are correctly executed
- Isolation: Concurrent behavior shall be the same as sequential behavior
- Durability: Changes are persisted and shall not be lost, even with a DMBS crash

Caveat:

- In the banking scenario, each access (GetBalance, SetNewBalanceToDB) follows ACID, but the database has no knowledge (or control) over the additional logic

Databases provide notions additional mechanisms to enforce ACID with macro operations

Database ACID characteristic

Locks: DBMS provide the capability for applications to lock the state

- Only a single set of operations may be executed, while others must wait
- There may be a distinction between read locks and write locks

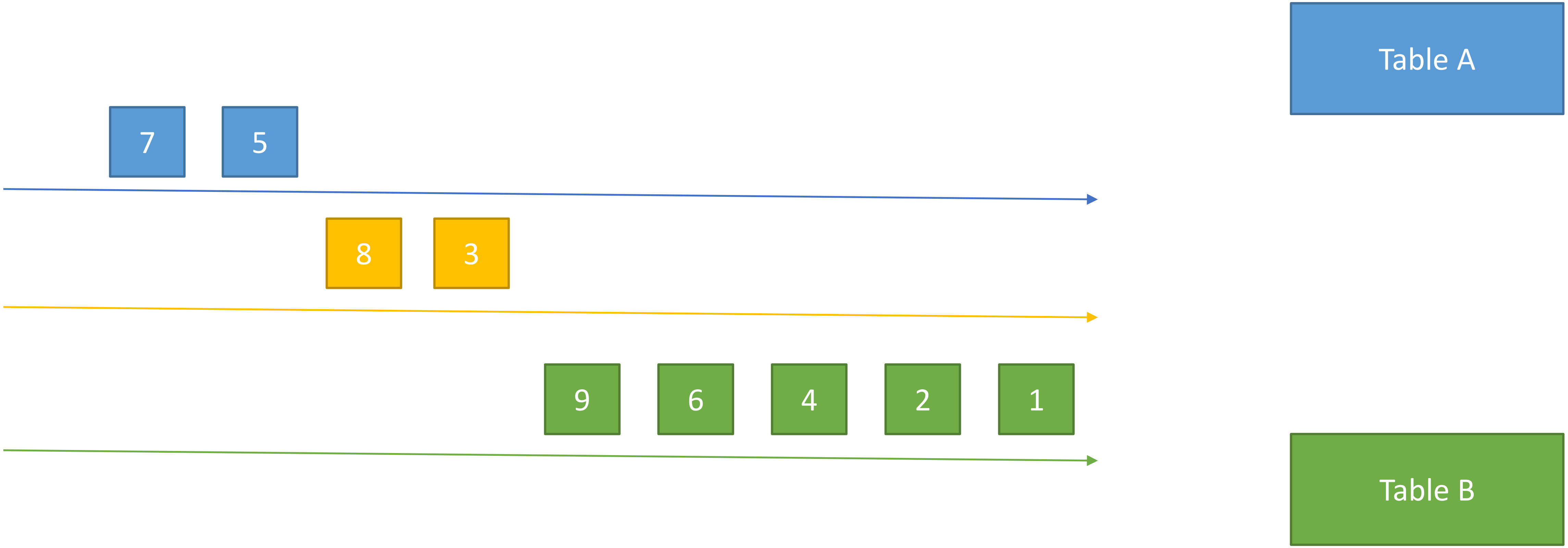
Versioning: DBMS advance DB state as versions and differences

- Read operations do not change state and can be executed
- Write operations can only be executed from the last persisted version
- Concurrent operations may require a refresh before commit

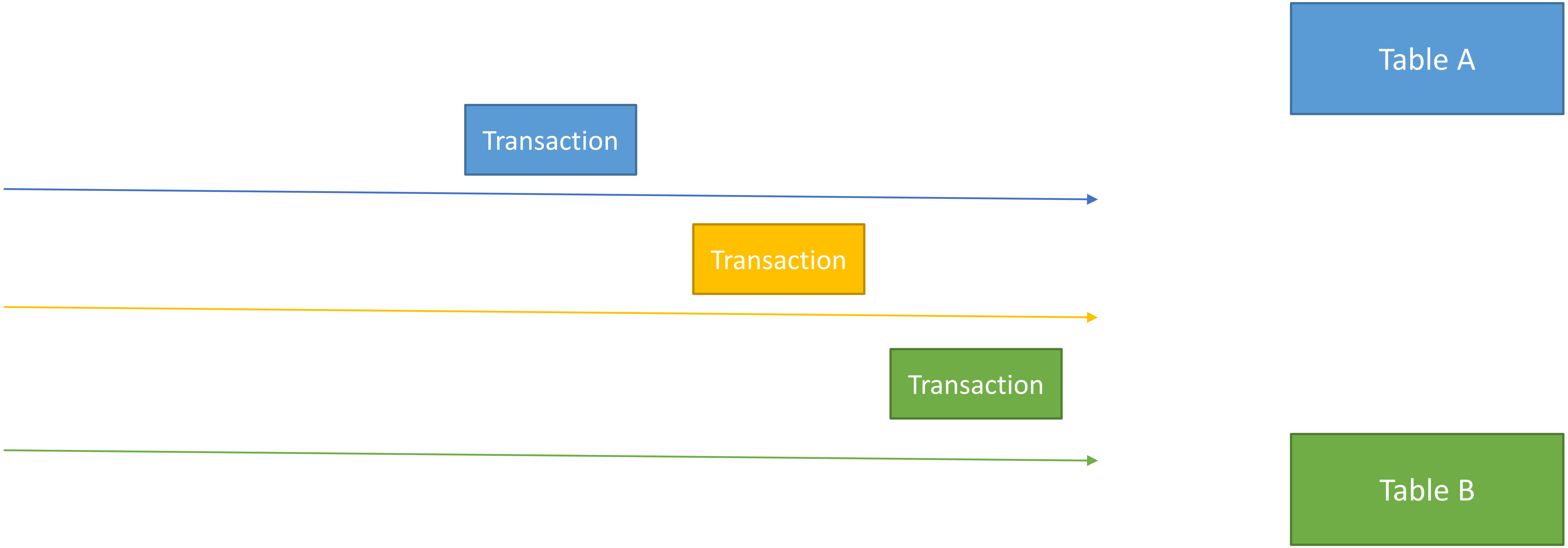
Transactions: DBMS create a context for a set of macro operations

- Software can operate over the context (READ, WRITE, etc…)
- The state then is discarded (rollback if required) or committed
- Commit is atomic.

Locks



Transactions



CWE-362 – Example – Banking - Transaction

```
BeginTransaction();
$transfer_amount = GetTransferAmount();
$balance = GetBalance();

if ($transfer_amount < 0) {
        EndTransaction();
        FatalError("Bad Transfer Amount");
}

$nb = $balance - $transfer_amount;
if (($balance - $transfer_amount) < 0) {
        EndTransaction();
        FatalError("Insufficient Funds");
}

SetNewBalanceToDB($nb);
CommitTransaction();
NotifyUser("Transfer of $transfer_amount succeeded.");
NotifyUser("New balance: $newbalance");
```

DB Operations are queued
Queue is discarded or committed atomically

CWE-362 – Example – Banking - Lock

```
LockDB();
$transfer_amount = GetTransferAmount();
$balance = GetBalance();

if ($transfer_amount < 0) {
        UnLockDB();
        FatalError("Bad Transfer Amount");
}

$nb = $balance - $transfer_amount;
if (($balance - $transfer_amount) < 0) {
        UnLockDB();
        FatalError("Insufficient Funds");
}

SetNewBalanceToDB($nb);
UnLockDB();
NotifyUser("Transfer of $transfer_amount succeeded.");
NotifyUser("New balance: $newbalance");
```

DB is locked.
No other operations take place

CWE-362 – Example – Banking - Versioning

```
GetVersion();
$transfer_amount = GetTransferAmount();
$balance = GetBalance();

if ($transfer_amount < 0) {
        FatalError("Bad Transfer Amount");
}

$nb = $balance - $transfer_amount;
if (($balance - $transfer_amount) < 0) {
        FatalError("Insufficient Funds");
}

SetNewBalanceToDB($nb);
Commit();
NotifyUser("Transfer of $transfer_amount succeeded.");
NotifyUser("New balance: $newbalance");
```

DB version is acquired.
Commit may FAIL if another change took place

CWE-362 – Example – Threads

```
// Global
shared_object_t data;

void update_data(char* cookie) {

     // Manipulate global data object

}
```

CWE-366 – Threads

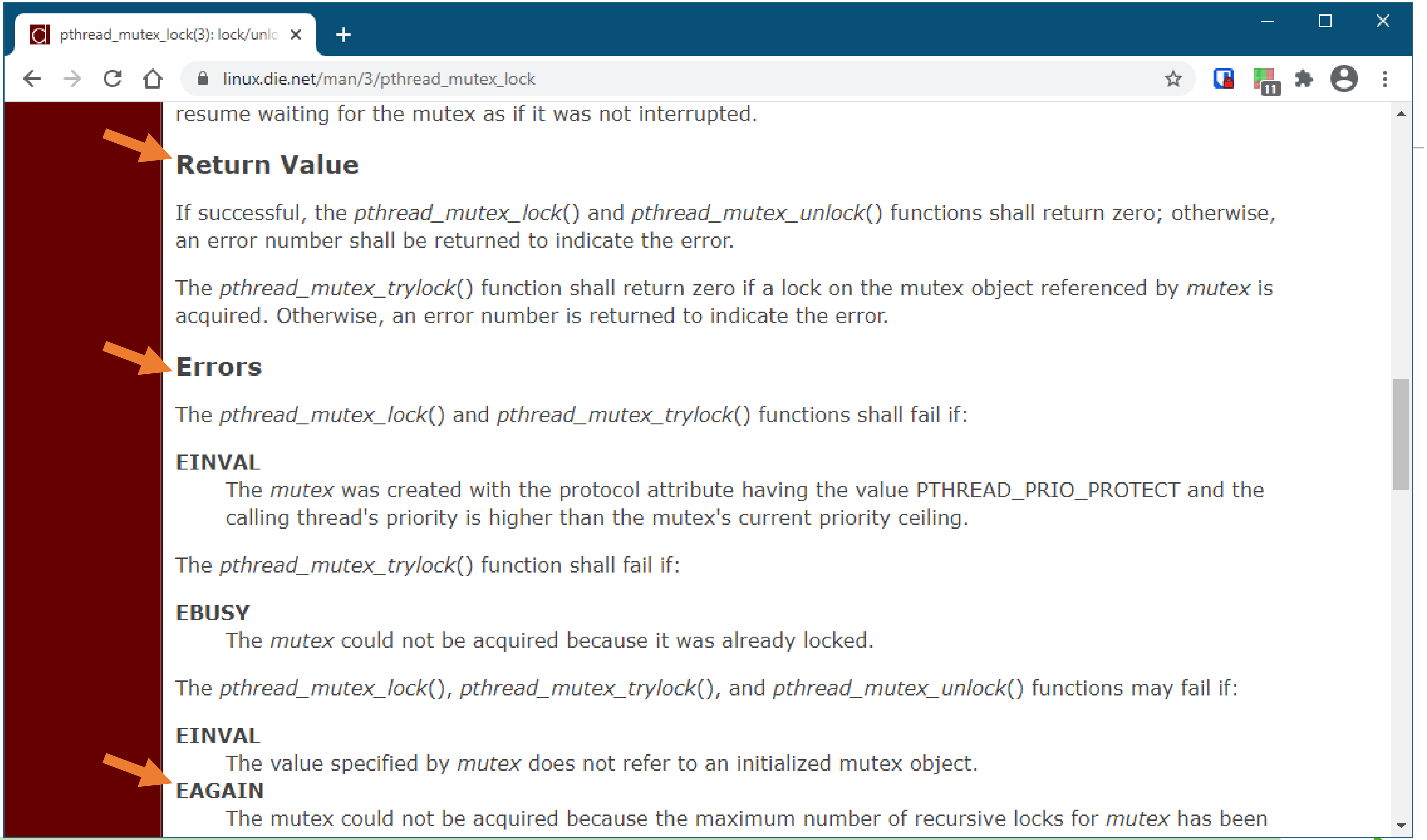
```
// Global
shared_object_t data;

void update_data(char* cookie, pthread_mutex_t * mutex) {
     pthread_mutex_lock(mutex);
     // Manipulate global data object
    pthread_mutex_unlock(mutex);
}
```

Direct solution: protect changes with a mutex

CWE-366 – Threads

```
// Global
shared_object_t data;

void update_data(char* cookie, pthread_mutex_t * mutex) {
     pthread_mutex_lock(mutex);
     // Manipulate global data object
     pthread_mutex_unlock(mutex);
}
```

Developer assumes lock/unlock always work



resume waiting for the mutex as if it was not interrupted.

Return Value

If successful, the *pthread_mutex_lock*() and *pthread_mutex_unlock*() functions shall return zero; otherwise, an error number shall be returned to indicate the error.

The *pthread_mutex_trylock*() function shall return zero if a lock on the mutex object referenced by *mutex* is acquired. Otherwise, an error number is returned to indicate the error.

Errors

The *pthread_mutex_lock*() and *pthread_mutex_trylock*() functions shall fail if:

EINVAL
The *mutex* was created with the protocol attribute having the value PTHREAD_PRIO_PROTECT and the calling thread's priority is higher than the mutex's current priority ceiling.

The *pthread_mutex_trylock*() function shall fail if:

EBUSY
The *mutex* could not be acquired because it was already locked.

The *pthread_mutex_lock*(), *pthread_mutex_trylock*(), and *pthread_mutex_unlock*() functions may fail if:

EINVAL
The value specified by *mutex* does not refer to an initialized mutex object.

EAGAIN
The mutex could not be acquired because the maximum number of recursive locks for *mutex* has been

CWE-362 – Race Condition – Isolated Ops

X86_64: i++ with gcc

```
add      DWORD PTR [rbp-4], 1
```

X86_64: i++ with clang

```
mov      edi, dword ptr [rbp - 8]
add      edi, 1
mov      dword ptr [rbp - 8], edi
```

ARM: i++

```
ldr      r3, [fp, #-8]
add      r3, r3, #1
str      r3, [fp, #-8]
```

Developer thinks: i++ is a single operation

In reality… it depends, and varies with the architecture

Still (generic behavior)

- Value of "i" must be available (previous logic)
- Value must be fetched from RAM to Cache
  - Page must be addressed and then loaded
  - MMUs and other systems are used
- Value must be fetched from cache to Register
- Register as to be increased
- Result must be stored in Cache
- Result shall be committed to RAM

CWE-367 - TOCTOU

Time-Of-Check, Time-Of-Use

The software **checks the state of a resource (TOC)** before using that resource,

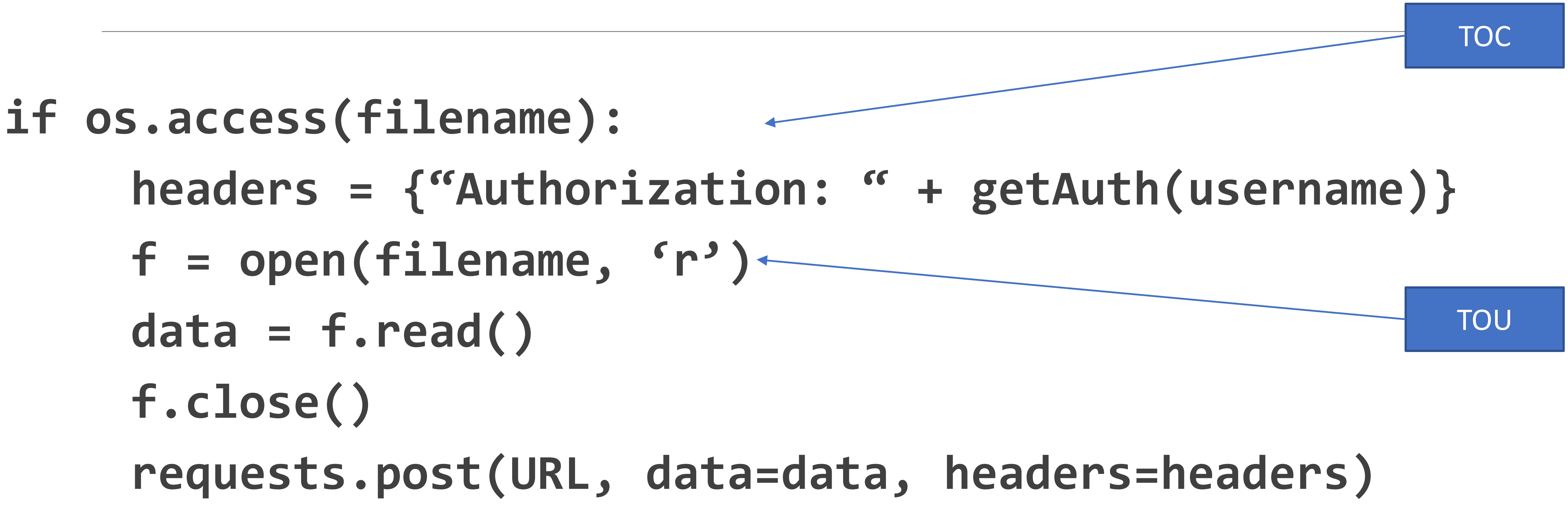
but the **resource's state can change between the check and the use (TOU)** in a way that invalidates the results of the check.

This can cause the software **to perform invalid actions** when the resource is in an unexpected state.

CWE-367 - TOCTOU

```
if os.access(filename):
    headers = {“Authorization: “ + getAuth(username)}
    f = open(filename, ‘r’)
    data = f.read()
    f.close()
    requests.post(URL, data=data, headers=headers)
```

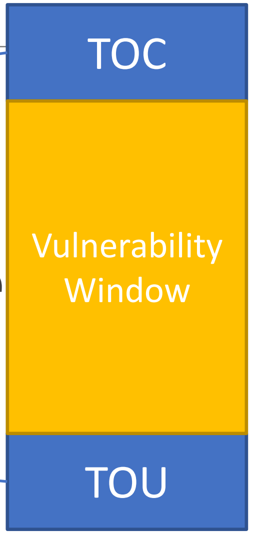
CWE-367 - TOCTOU



```
if os.access(filename):
      headers = {“Authorization: “ + getAuth(username)}
      f = open(filename, ‘r’)
      data = f.read()
      f.close()
      requests.post(URL, data=data, headers=headers)
```

CWE-367 - TOCTOU

```
if os.access(filename):
    headers = {“Authorization: “ + getAuth(username
    f = open(filename, ‘r’)
    data = f.read()
    f.close()
    requests.post(URL, data=data, headers=headers)
```

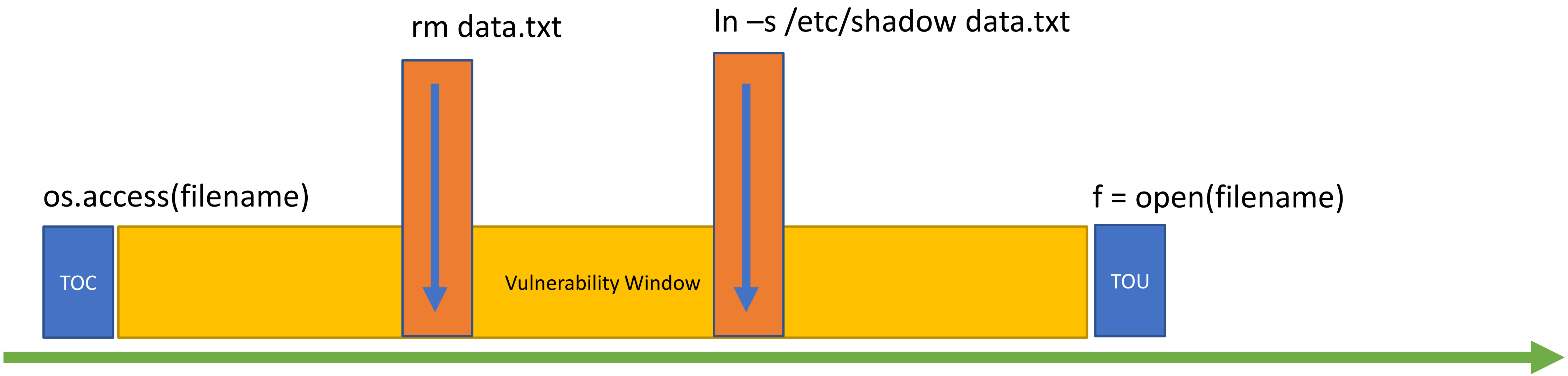


CWE-367 – TOCTOU attack

program run with elevated privileges (setuid)
filename = data.txt

Result
Program will upload /etc/shadow

Access: Use the real uid/gid to test for access to *path*.
Open: Opens file using the effective uid/gid



CWE-367 - TOCTOU

And the list goes on…

```
if user_exists_in_db(username):
        user = get_user(username)
```

Should be

```
user = get_user(username)    #get_user makes a single query
```

CWE-367 – TOCTOU – Bad Logic

Some logic mistakes can create implicit TOCTOU errors

- Not attacks, but software mistakes

```
f = open(“file.txt”, “w”)

# some code that does os.unlink(“file.txt”) by mistake
# or the file is deleted externally

f.write(data)
f.close()
```

Write is works, but nothing will be on disk!

CWE-365: Race Condition in Switch

The switch instruction is inherently dangerous as the expected behavior is very different from the actual behavior

```
switch(a){
   case 0: foo(); break;
   case 1: bar(); break;
  ..
  case n: zed(); break;
}
```

Expected

1. **Evaluate the value of “a”**
2. **Execute the function**

CWE-365: Race Condition in Switch

The switch instruction is inherently dangerous as the expected behavior is very different from the actual behavior

```
switch(a){
    case 0: foo(); break;
    case 1: bar(); break;
   ..
    case n: zed(); break;
}
```

Reality

1. Compare the value of “a” with one option and execute if true
2. Compare the value of “a” with another option and execute if true
3. ….

CWE-365: Race Condition in Switch

The switch instruction is inherently dangerous as the expected behavior is very different from the actual behavior

```
switch(a){
    case 0: foo(); break;
    case 1: bar(); break;
  ..
    case n: zed(); break;
}
```

Issue

1. **“a” can change between comparisons**
2. **“a” may be matched to an incorrect function**
3. **“a” may not be matched!**

CWE-365: Race Condition in Switch

```c
int f(int num) {

    int a = num;

    switch(a){

        case 0: foo(); break;

        case 1: bar(); break;

        case 3: zed(); break;

    }

}
```

```asm
        test    eax, eax
        je      .LBB3_1
        jmp     .LBB3_5
.LBB3_5:
        mov     eax, dword ptr [rbp - 16]
        sub     eax, 1
        je      .LBB3_2
        jmp     .LBB3_6
.LBB3_6:
        mov     eax, dword ptr [rbp - 16]
        sub     eax, 3
        je      .LBB3_3
        jmp     .LBB3_4
.LBB3_1:
        call    foo()
        jmp     .LBB3_4
.LBB3_2:
        call    bar()
        jmp     .LBB3_4
.LBB3_3:
        call    zed()
```

CWE-365: Race Condition in Switch

```c
int f(int num) {

    int a = num;

    switch(a){

        case 0: foo(); break;

        case 1: bar(); break;

        case 3: zed(); break;

    }

}
```

```asm
         test    eax, eax
         je      .LBB3_1
         jmp     .LBB3_5
.LBB3_5:
         mov     eax, dword ptr [rbp - 16]
         sub     eax, 1
         je      .LBB3_2
         jmp     .LBB3_6
.LBB3_6:
         mov     eax, dword ptr [rbp - 16]
         sub     eax, 3
         je      .LBB3_3
         jmp     .LBB3_4
.LBB3_1:
         call    foo()
         jmp     .LBB3_4
.LBB3_2:
         call    bar()
         jmp     .LBB3_4
.LBB3_3:
         call    zed()
```

a is evaluated

TOCTOU

In practice, TOCTOU is extremely prevalent

- dependent on system performance
  - Higher performance will make vuln. windows smaller, but the attacker may have similar resources if running locally
- dependent on target CPU architectures, compilers and flags
  - The code produced may mask the vulnerability
- hard to debug dynamically
  - Behavior under a debugger will be different
  - Subject to small timings

Prevention

- Assert that actions are serialized as expected: may require lower layer knowledge
- Force serialization manually (for DBs and other shared objects)
- If possible, send macro ops to systems (whole transactions) which lock resources at source
- Reduce the use of filenames to a single call, then use File Descriptors

Exercise

Take the bank.zip package available at elearning and install the dependencies:

- `pip3 install --user cherrypy`

Run the server: `python3 server.py`

Run the client: `python3 client.py`

- The client will withdraw $10 from an account initialized with $10000
- 256 clients are started, each withdrawing $10
- If everything is ok: **$2560 will be removed, final balance is $7440**

Check the balance at http://127.0.0.1:8080 and analyze the file log.txt

- Example of a text run: Final balance is $9940 and $2560 was withdrawn. Bank lost $2500.

Can you fix the code? Where is the problem? How can it be fixed?

- Remember: locks, transactions, versioning, etc… try ☺

Covert Channels and Discrepancies

CWE-514: A covert channel is a path that can be used to transfer information in a way not intended by the system's designers.

Some relation to Time and State issues as External State can be determined from observation of side effects, or internal state

CWE-515: Covert Storage Channel

CWE-385: Covert Timing Channel

CWE-203: Observable Discrepancy

Covert Timing Channel

Covert timing channels convey information

- by modulating some aspect of system behavior over time
- so that the program receiving the information can observe system behavior and infer protected information

Covert channels are long used to exfiltrate information from systems

- Modulate system response time, packet interval, etc..

But undesirable Cover Timing Channels can be present due to flaws

- Unknown to the developer/sysadmin
- But perceived to the attacker, allowing attackers to guess state from timing discrepancies

Covert channels can be limited and reduced of usefulness

- Can be prevent in specific cases, especially time based
- Covert channels for malicious purposes can not be avoided altogether

Covert Timing Channel

Code checks if two passwords are the same

- First the length
- Then byte comparison, exiting on first unmatching byte

Provides a covert channel making it possible to guess the password

```
Same password:      0.710 usecs
Different length:   0.147 usecs
First byte wrong:   0.366 usecs
Second byte wrong:  0.401 usecs
Last byte wrong:    0.656 usecs
```

Solutions may consider:

- Different logic
- Making functions time constant
- Adding random delay (delay should be dominant)

```python
def validate_password(actual_pw, typed_pw):

  if len(actual_pw) != len(typed_pw):
    return False

  for i in range(len(actual_pw)):
    if actual_pw[i] != typed_pw[i]:
      return False

  return False
```

```python
def validate_password(actual_pw, typed_pw):

  time.sleep(random() / 100)  # Throw random time

  if len(actual_pw) != len(typed_pw):
    return False

  for i in range(len(actual_pw)):
    if actual_pw[i] != typed_pw[i]:
      return False

  return False
```

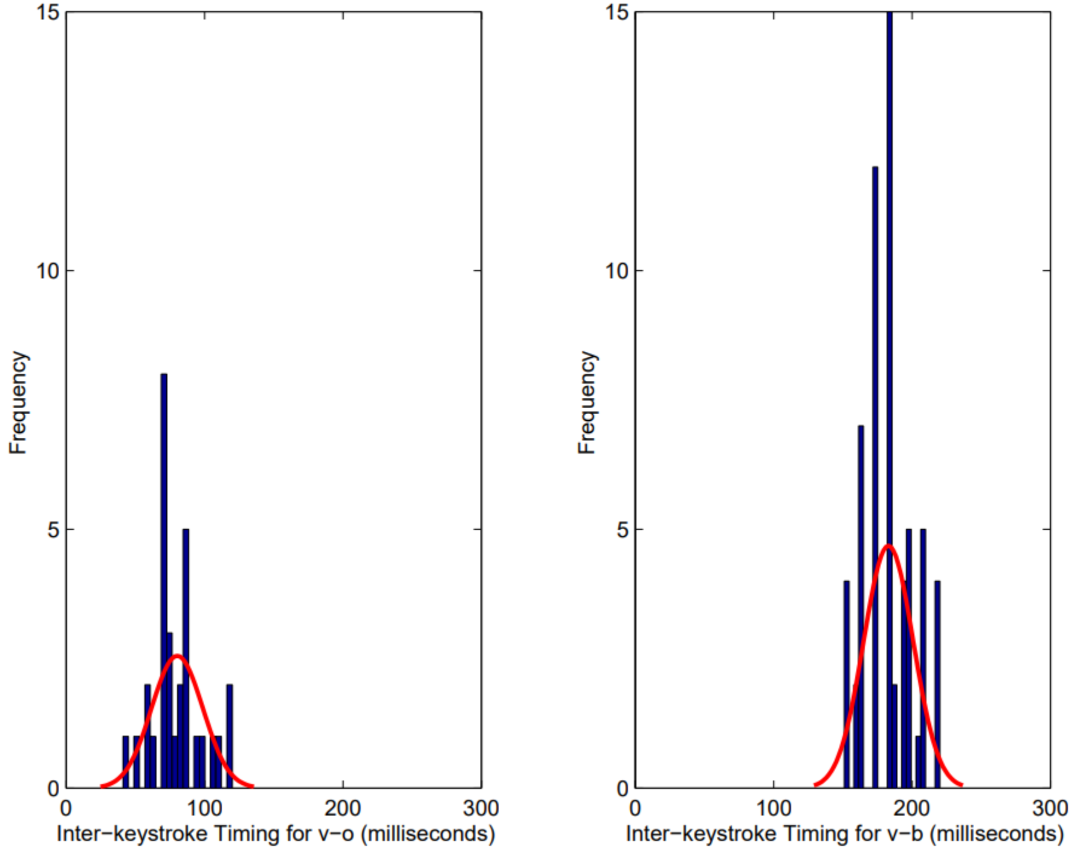
Covert Channel

Some covert channels are created by physical interactions

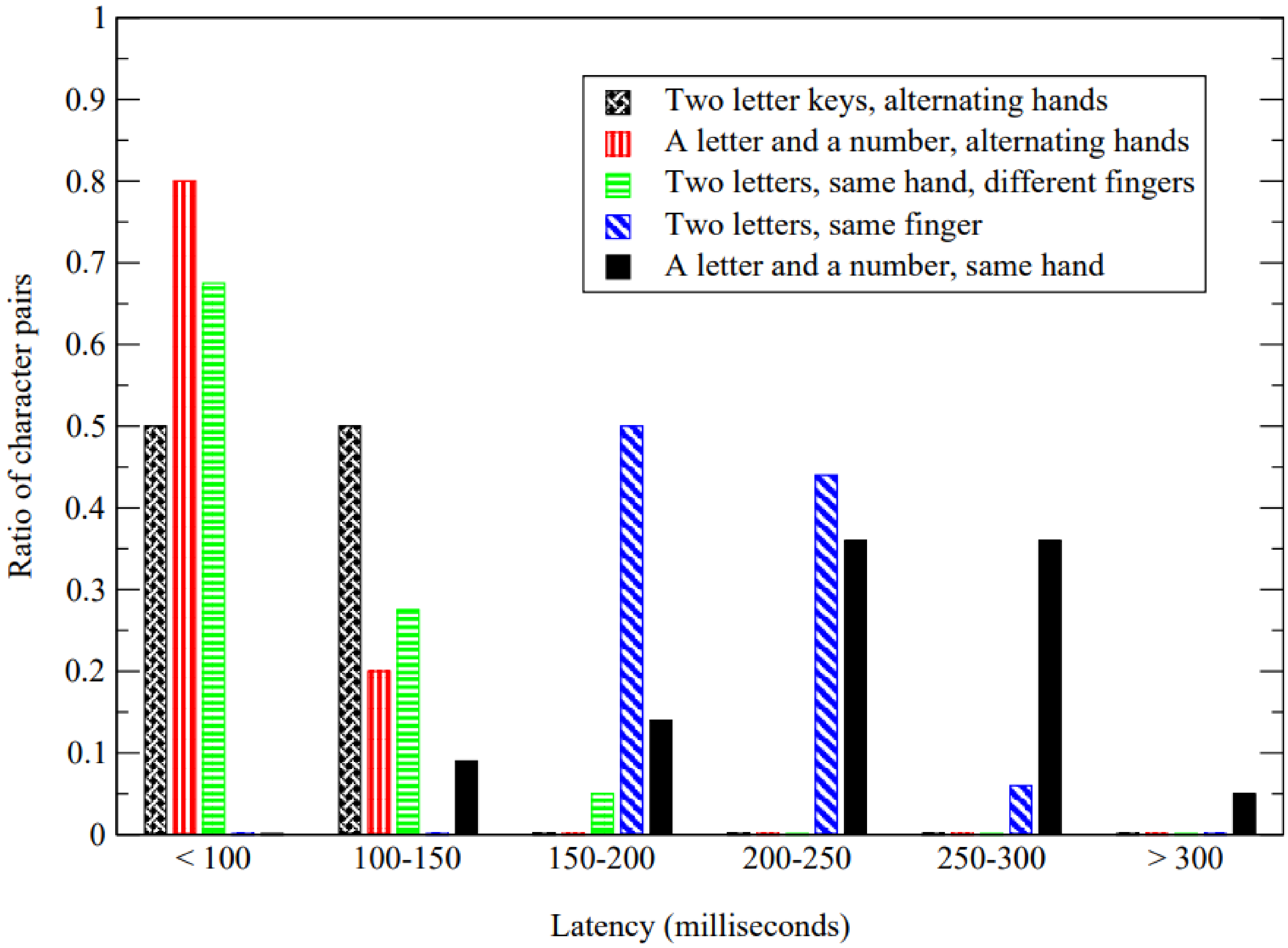
- Keyboards, smartphones
- Typing creates patterns due to hand anatomy and keyboard layout
- Touching a smartphone to enter a code produces small axis rotations

Some covert chanels are inherent to the protocol operation

- Delay between packets can help discriminate a VPN from VoIP
  - VoIP produces packets with constant packet intervals
- A download has a request upstream, many packets with content downstream, and a few Acks upstream



Timing Analysis of Keystrokes and Timing Attacks on SSH


1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
Ratio of character pairs
Two letter keys, alternating hands
A letter and a number, alternating hands
Two letters, same hand, different fingers
Two letters, same finger
A letter and a number, same hand
< 100
100-150
150-200
200-250
250-300
> 300
Latency (milliseconds)

Microarchitectural Covert Channels

Since 2017 a new class of bugs was published which exploits microarchitectural behavioral changes

- Related to the access mechanisms to RAM by the CPU
- Potentiated by speculative and out of order execution mechanisms in present CPUs

General strategy: measure timing differences accessing resources, which will provide information about private data

- Resources are memory pages, memory addresses in the program address space or outside it

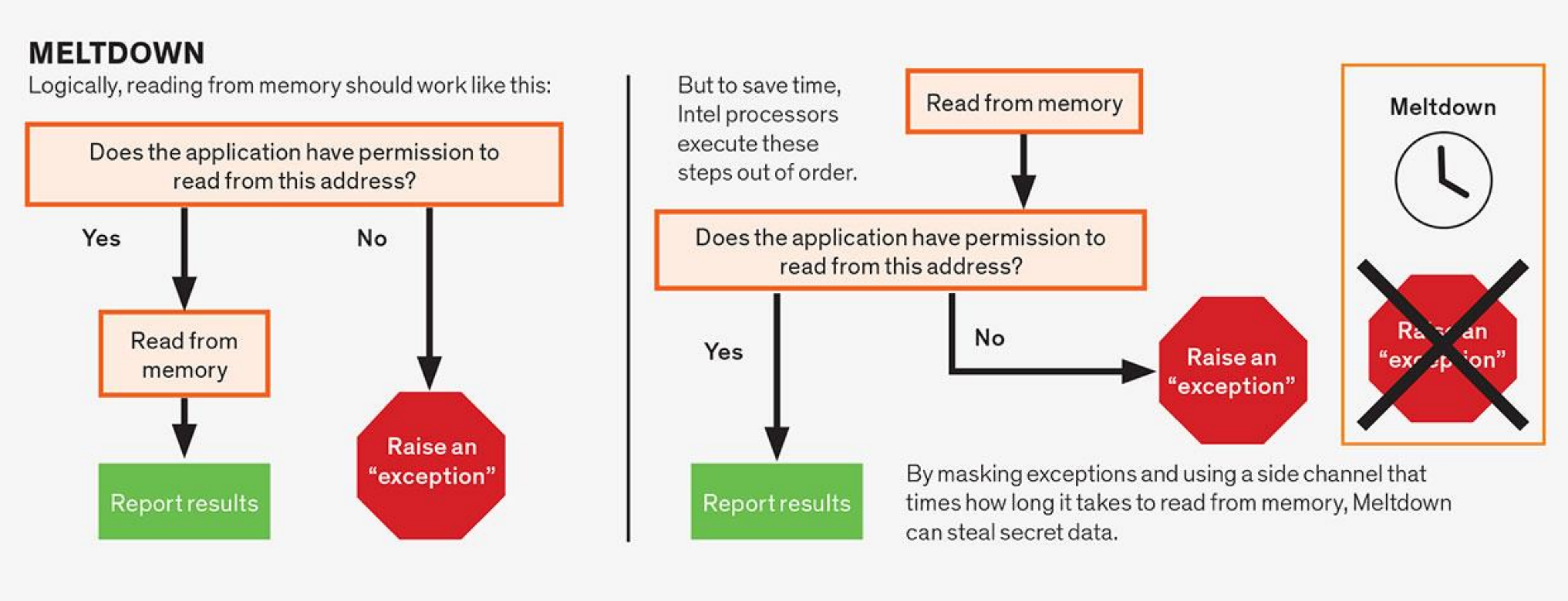
Impact:

- Attacker can read memory content from other parts of process space, or even kernel space
- Attacker can also read memory from other VMs, processes, maybe enclaves…
- Can be explored remotely through network card drivers
  - In the beginning even Javascript engines were vulnerable

Meltdown Type

Affected systems include most Intel CPUs since 1995

- Also some ARM and PowerPC, AMD Phenom, EPYC, ZEN



MELTDOWN

Logically, reading from memory should work like this:

But to save time, Intel processors execute these steps out of order.

By masking exceptions and using a side channel that times how long it takes to read from memory, Meltdown can steal secret data.

IEEE Spectrum

Meltdown

The problem:

- Out of order execution implies that instructions will be executed before they should
- Executing future operations causes side effects to the present

Basic algorithm:

1. Allocate a 256*4096 chunk of memory
   - 256 because the objective is to find the value of a byte, which can have a value from 0 to 256
   - Because pages are not accessed, they exist in RAM but not in cache
   - There is a timing cover channel present as access cache is faster than accessing RAM
2. Create an exception
3. Read byte from the target memory (outside the scope of the program)
4. Multiply byte by 4096
5. Use value to access the memory allocated in 1

Test code: https://github.com/IAIK/meltdown

Meltdown – information sender

(2) will block execution

- (3-5) are not expected to be executed
- But they are….

(5) will cause a page to be loaded in the cache

- The page will be dependent on the value of the byte accessed (due to (4-5))

The presence of a page on cache constitutes a timing covert channel

- Time to access this page will be lower than the time to access other pages

1. Allocate a 256*4096 chunk of memory
2. Create an exception
3. Read secret byte from the target memory (outside the scope of the program)
4. Multiply byte by 4096
5. Use value to access the memory allocated in 1

Meltdown – information receiver

Attack is dependent on accurate timing

- Reducing the granularity of timers mitigates the attack to some extent
- Other applications executing will add entropy
- May require some training to acquire the most adequate timing

Any memory address can be accessed

- But only a byte can be exfiltrated at once
- Addresses are virtual and not actual RAM addresses

Original work achieved 500kb/s

1. Catch exception
2. Loop through array of allocated pages
3. Measure time to access each page
4. Page with lower access time corresponds to value of secret byte

Meltdown – Information Sender

```
    ; rcx = kernel address, rbx = probe array
    xor rax, rax
retry:
    mov al, byte [rcx]
    shl rax, 0xc
    jz retry
    mov rbx, qword [rbx + rax]
```

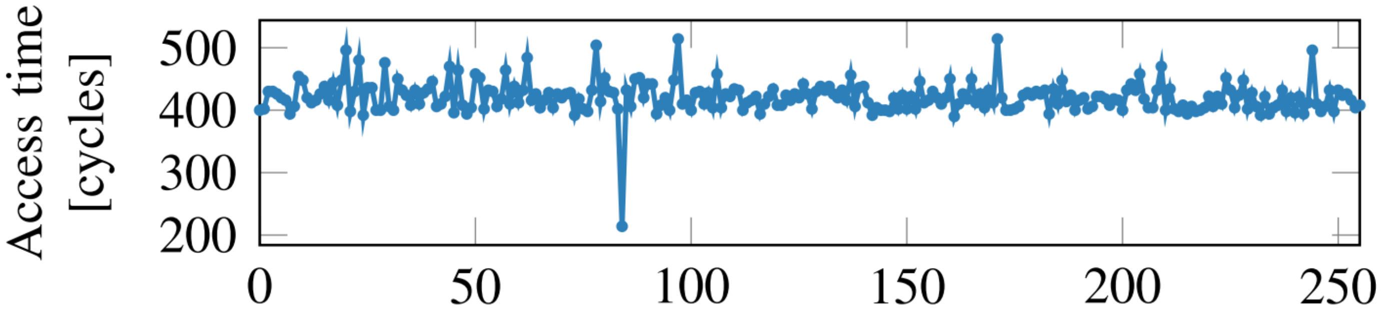
Access a byte at an invalid address (out of of the virtual address space) raising an exception

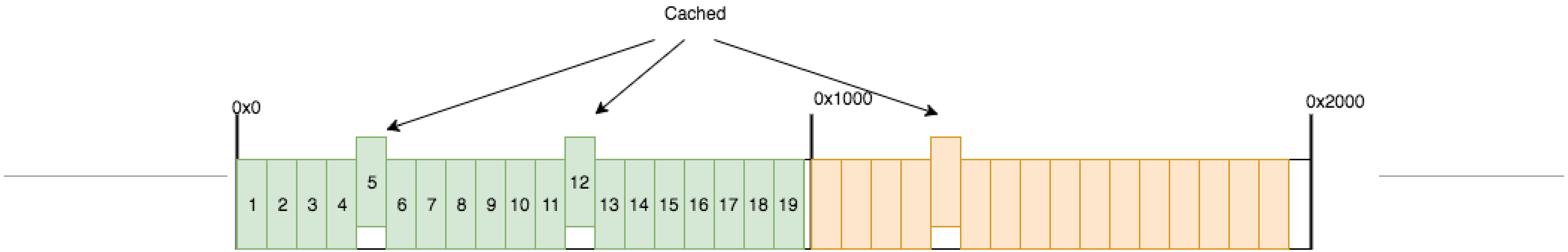
Multiply by 4096 in order to access a specific page

Access a position in our array. Position is a page dependent on the value of read from the target memory

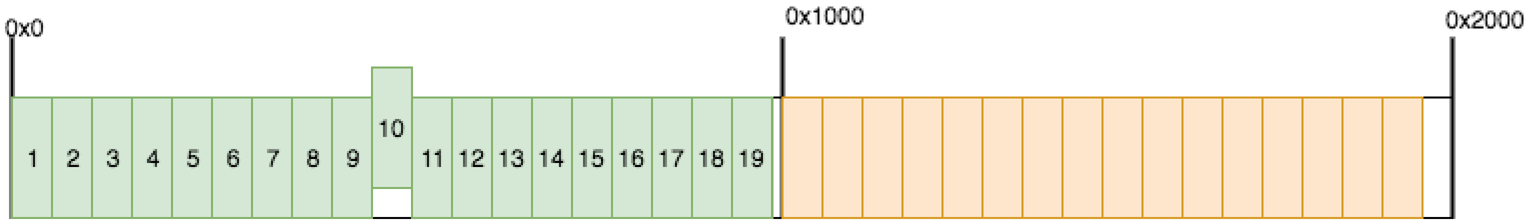
Meltdown – information receiver

```
//Sender:
   char *kernel = 0x1000;
   char secret = kernel[10]; //Will raise an exception and secret is never set

//Receiver
for (int i=0; i < 256; i++) {
    t1 = now();
    char dummy = probe[i * 4096];
    t2 = now();
    accessTimes[i] = t2-t1;
}
```

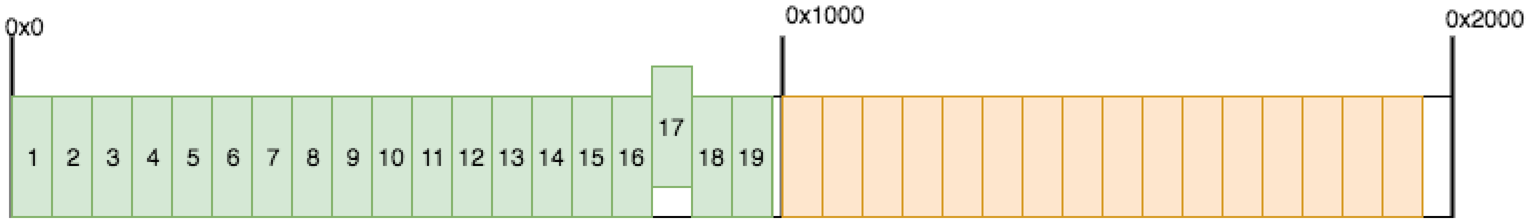




`char dummy = userspace[10 * 4096];`



`char dummy = userspace[kernel[17] * 4096];`



Spectre

Similar to Meltdown but exploring different flaws

- Meltdown explores an exception, expecting that following instructions are still executed, causing side effects which can be measured

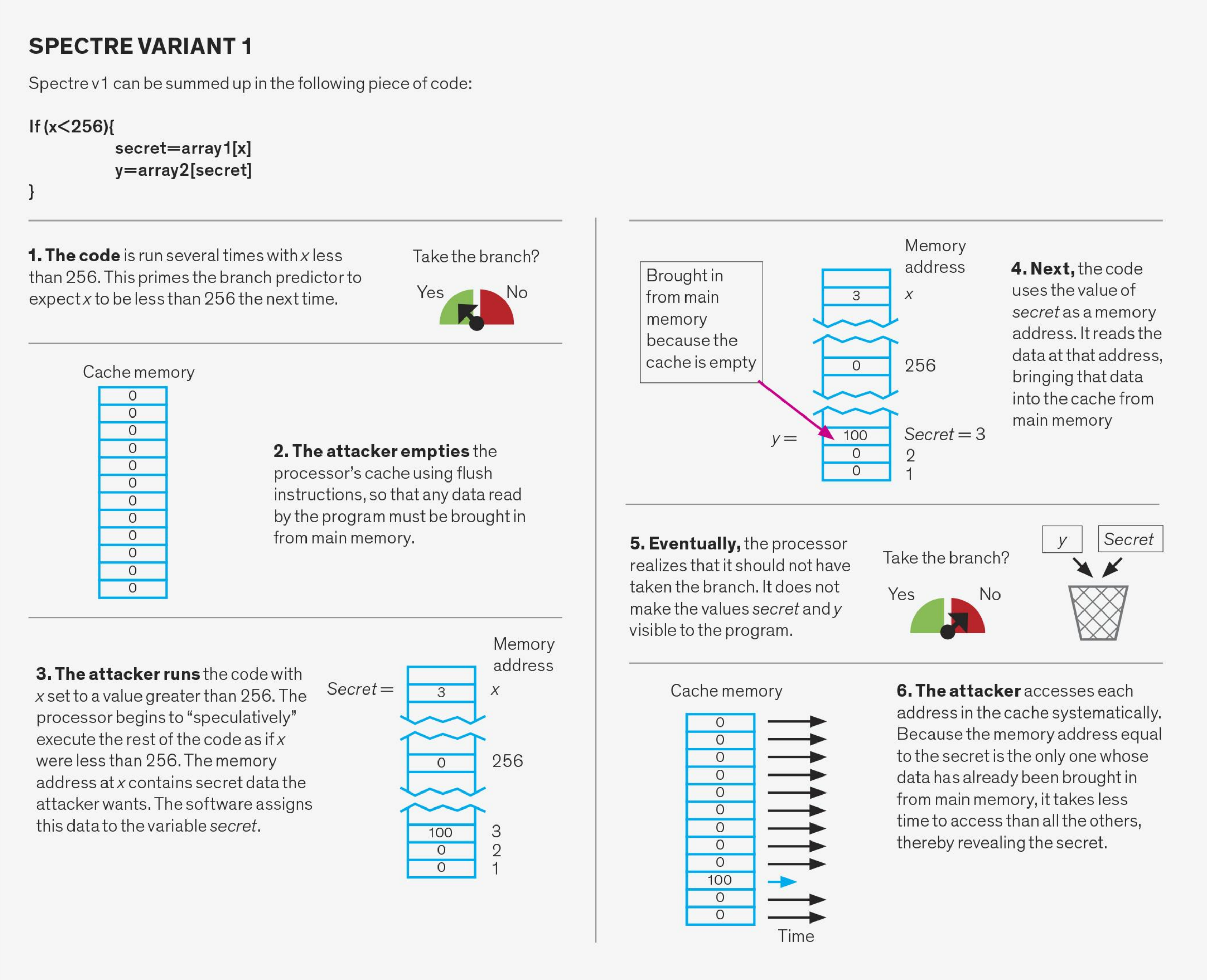
Spectre explores branch predictors

1. Train branch predictor so that CPU predict a positive branch (that is, doesn’t branch)
2. Execute a condition that will fail
    1. Code inside that condition will be executed speculatively and result will be discarded
    2. Timing side effects will be present in the cache lines
3. Proceed as with meltdown

Doesn’t generate any exception, can be explore by remote attackers

- Javascript in browsers
- Network drivers when processing packets

Spect



SPECTRE VARIANT 1

Spectre v1 can be summed up in the following piece of code:

```
If (x<256){
        secret=array1[x]
        y=array2[secret]
}
```

1. The code is run several times with *x* less than 256. This primes the branch predictor to expect *x* to be less than 256 the next time.

2. The attacker empties the processor's cache using flush instructions, so that any data read by the program must be brought in from main memory.

3. The attacker runs the code with *x* set to a value greater than 256. The processor begins to "speculatively" execute the rest of the code as if *x* were less than 256. The memory address at *x* contains secret data the attacker wants. The software assigns this data to the variable *secret*.

4. Next, the code uses the value of *secret* as a memory address. It reads the data at that address, bringing that data into the cache from main memory

5. Eventually, the processor realizes that it should not have taken the branch. It does not make the values *secret* and *y* visible to the program.

6. The attacker accesses each address in the cache systematically. Because the memory address equal to the secret is the only one whose data has already been brought in from main memory, it takes less time to access than all the others, thereby revealing the secret.

Mitigating Spectre and Meldown

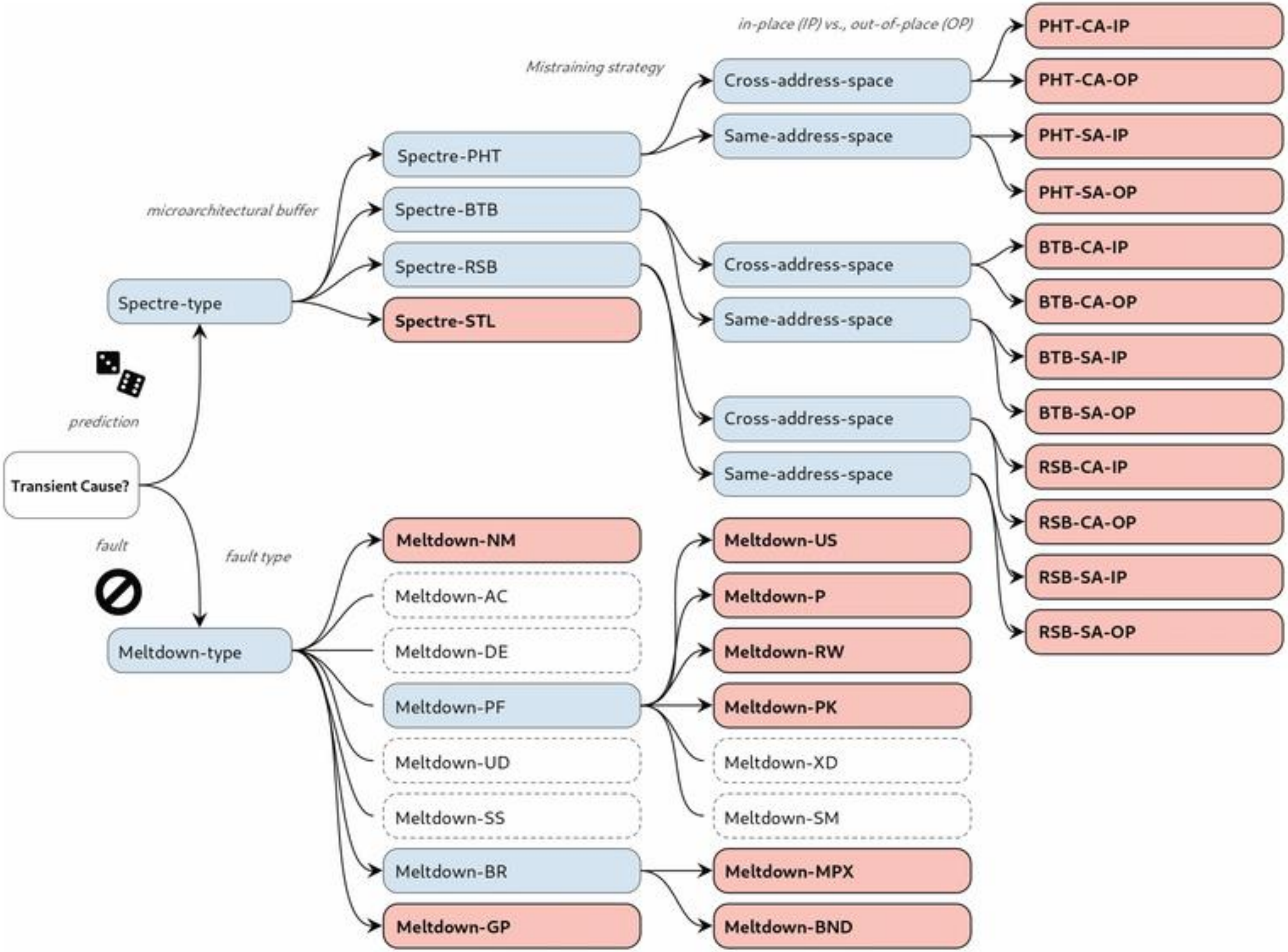
For remotely exposed systems (browsers, network), limiting the accuracy of timers is a quick solution

- Although the vulnerability exists, data exfiltration will not be possible

For local systems, microcode and kernel updates are required

- Adding barriers to exceptions, preventing speculative execution
- Generating bytecode not presenting an attack potential

Problem… new variants are being presented, exploring an ever increasing surface



Transient Cause?

prediction

fault

Spectre-type

Meltdown-type

microarchitectural buffer

Spectre-PHT

Spectre-BTB

Spectre-RSB

Spectre-STL

Mistraining strategy

Cross-address-space

Same-address-space

Cross-address-space

Same-address-space

Cross-address-space

Same-address-space

in-place (IP) vs., out-of-place (OP)

PHT-CA-IP

PHT-CA-OP

PHT-SA-IP

PHT-SA-OP

BTB-CA-IP

BTB-CA-OP

BTB-SA-IP

BTB-SA-OP

RSB-CA-IP

RSB-CA-OP

RSB-SA-IP

RSB-SA-OP

fault type

Meltdown-NM

Meltdown-AC

Meltdown-DE

Meltdown-PF

Meltdown-UD

Meltdown-SS

Meltdown-BR

Meltdown-GP

Meltdown-US

Meltdown-P

Meltdown-RW

Meltdown-PK

Meltdown-XD

Meltdown-SM

Meltdown-MPX

Meltdown-BND

Mitigating Spectre and Meldown