

# Concurrency and security

**Questões de Segurança em Engenharia de Software (QSES)**

Mestrado em Segurança Informática

Departamento de Ciência de Computadores

Faculdade de Ciências da Universidade do Porto

Eduardo R. B. Marques, [edrdo@dcc.fc.up.pt](mailto:edrdo@dcc.fc.up.pt)

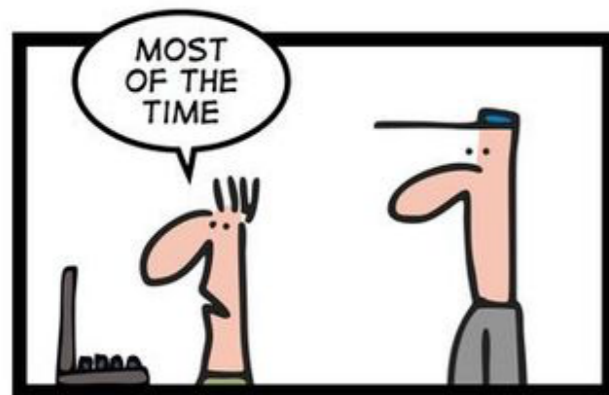
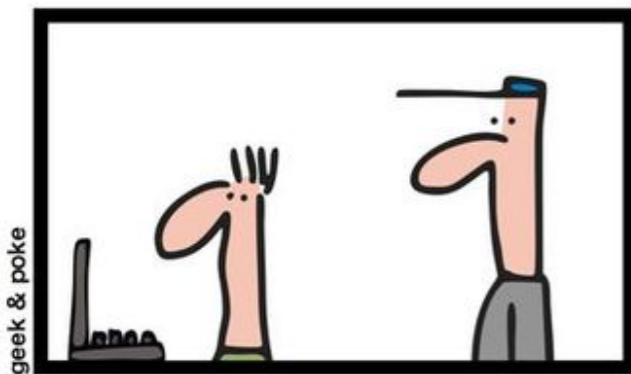
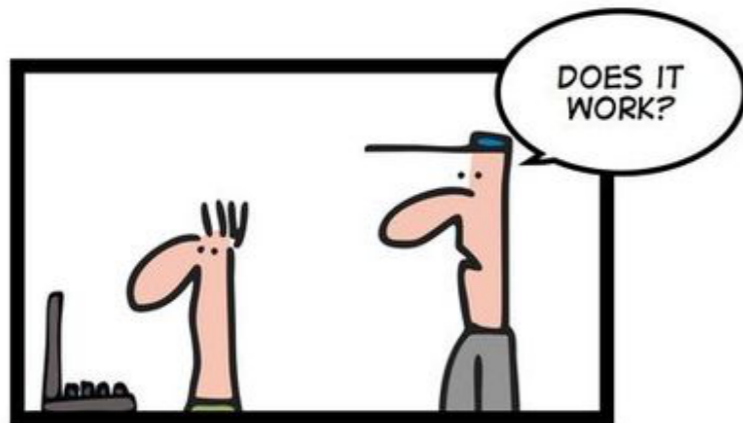


# Concurrency

- Concurrency
  - multiple computational processes that execute at the same time and interact with each other
- Some programmers may only deal with sequential code ...
- ... but even “sequential” code in reality is normally intertwined with several concurrent systems at different levels in the software / hardware stack
- Think of modern-day software and the intricate connection between:
  - data centers in the cloud
  - networks
  - OS processes and the kernel
  - multiple threads within a single process
  - CPUs with multiple cores & complex memory hierarchy
  - I/O handling
  - ...

# Concurrency works “most of the time”

SIMPLY EXPLAINED



CONCURRENCY

- Why only “most of the time”?
- Reasons for non-determinism?
- Security issues ?

Cartoon by Oliver Widder, CC 3.0 license

[geekandpoke.com](http://geekandpoke.com)

# The “time and state” pernicious kingdom

- Concurrency blurs clear notions of **time and state**, one of the [7 pernicious kingdoms](#) in software security.
- [CWE category CWE-361 - Time and State](#)
  - ***“weaknesses related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads”***
  - ***“in order for more than one component to communicate, state must be shared, and all that takes time [...] Defects rush to fill the gap between the programmer's model of how a program executes and what happens in reality. These defects are related to unexpected interactions between threads, processes, time, and information.”***

# Concurrency vulnerabilities

- General
  - [CWE-362](#) — Race Condition
  - [CWE-662](#): Improper Synchronization
  - [CWE-512](#) (and subtypes [385](#) / [515](#)) Covert (Timing / Space) Channel
- Examples of more specific vulnerability classes
  - [CWE-366](#), [CWE-567](#) : Race Condition within a Thread: Unsynchronized Access to Shared Data in a Multithreaded Context
  - [CWE-367](#): Time-of-check Time-of-use (TOCTOU) Race Condition
  - [CWE-377](#): Insecure Temporary File
  - [CWE-1037](#): Processor Optimization Removal or Modification of Security-critical Code - a category introduced in 2018 related to the [Meltdown and Spectre](#) vulnerabilities

# Race condition (CWE-362)

- *“The program contains a **code sequence that can run concurrently with other code, and the code sequence requires temporary, exclusive access to a shared resource, but a timing window exists in which the shared resource can be modified by another code sequence that is operating concurrently.**”*
- *“[...] A race condition **violates these properties, which are closely related:***
  - ***Exclusivity** - the code sequence is given **exclusive access** to the shared resource, i.e., 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**, i.e., no other thread or process can concurrently execute the same sequence of instructions (or a subset) against the same resource.”*

# CWE-362 — example vulnerabilities

## Security Vulnerabilities Related To CWE-362

CVSS Scores Greater Than: [0](#) [1](#) [2](#) [3](#) [4](#) [5](#) [6](#) [7](#) [8](#) [9](#)

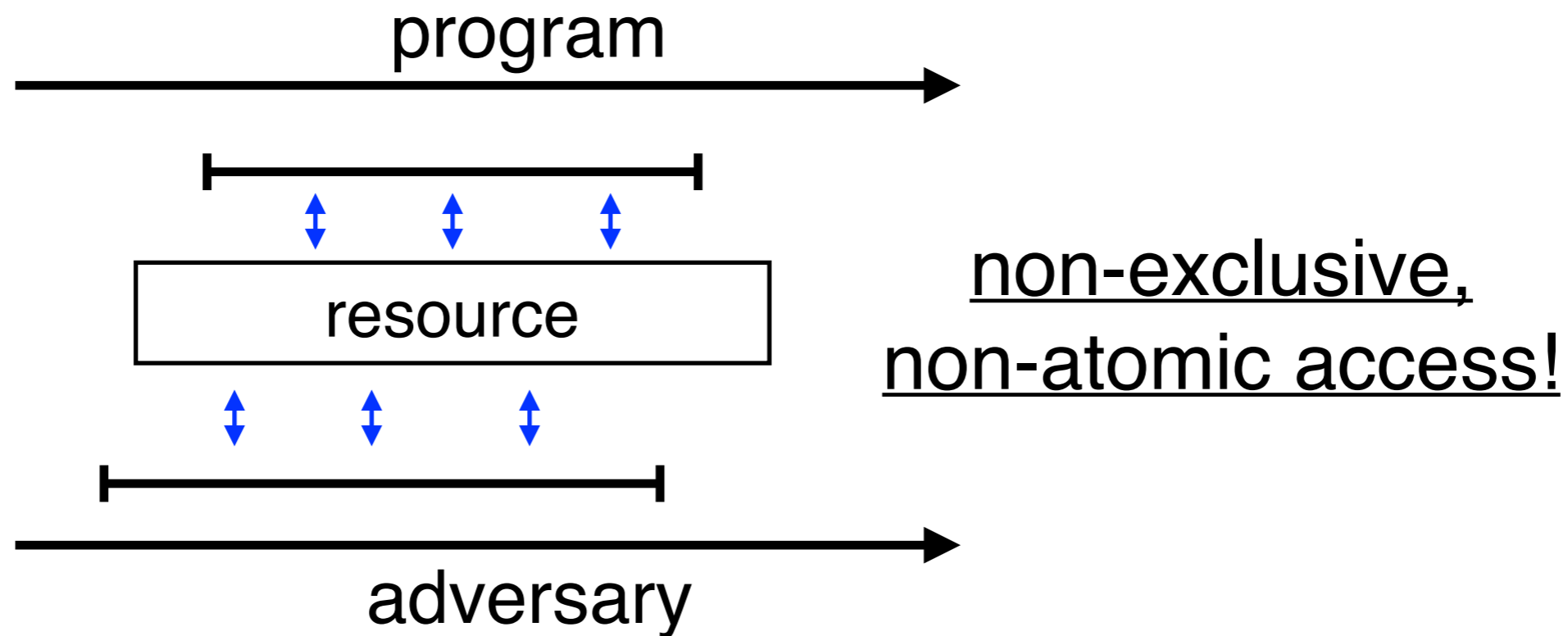
Sort Results By : [CVE Number Descending](#) [CVE Number Ascending](#) [CVSS Score Descending](#) [Number Of Exploits Descending](#)

Total number of vulnerabilities : **383** Page : [1](#) (This Page) [2](#) [3](#) [4](#) [5](#) [6](#) [7](#) [8](#)

[Copy Results](#) [Download Results](#)

#	CVE ID	CWE ID	# of Exploits	Vulnerability Type(s)	Publish Date	Update Date	Score	Gained Access Level
1	<a href="#">CVE-2016-7916</a>	<a href="#">362</a>		+Info	2016-11-16	2016-11-28	7.1	None
Race condition in the environ_read function in fs/proc/base.c in the <b>Linux kernel</b> before 4.5.4 allows local users to obtain sensitive file during a process-setup time interval in which environment-variable copying is incomplete.								
2	<a href="#">CVE-2016-7777</a>	<a href="#">362</a>			2016-10-07	2016-10-11	3.3	None
<b>Xen 4.7.x</b> and earlier does not properly honor CR0.TS and CR0.EM, which allows local x86 HVM guest OS users to read or modify arbitrary tasks on the guest by modifying an instruction while the hypervisor is preparing to emulate it.								
3	<a href="#">CVE-2016-7098</a>	<a href="#">362</a>		Bypass	2016-09-26	2016-11-28	6.8	None
Race condition in <b>wget 1.17</b> and earlier, when used in recursive or mirroring mode to download a single file, might allow remote HTTP connection open.								
4	<a href="#">CVE-2016-6480</a>	<a href="#">362</a>		DoS	2016-08-06	2016-11-28	4.7	None
Race condition in the ioctl_send_fib function in drivers/scsi/aacraid/commctrl.c in the Linux kernel through 4.7 allows local user (crash) by changing a certain size value, aka a "double fetch" vulnerability.								
5	<a href="#">CVE-2016-6156</a>	<a href="#">362</a>		DoS	2016-08-06	2016-11-28	1.9	None
Race condition in the ec_device_ioctl_xcmd function in drivers/platform/chrome/cros_ec_dev.c in the Linux kernel before 4.7 allows local user (denial of service) by changing a certain size value, aka a "double fetch" vulnerability.								
6	<a href="#">CVE-2016-6136</a>	<a href="#">362</a>		Bypass	2016-08-06	2016-11-28	1.9	None
Race condition in the audit_log_single_execve_arg function in kernel/auditsc.c in the Linux kernel through 4.7 allows local user auditing by changing a certain string, aka a "double fetch" vulnerability.								

# Race conditions



- Types of resources:
  - memory, files, databases, ...
- Exploitability / reproducibility
  - exploit tamper the program or its environment to materialize a window of vulnerability



# Simple “bank account” example

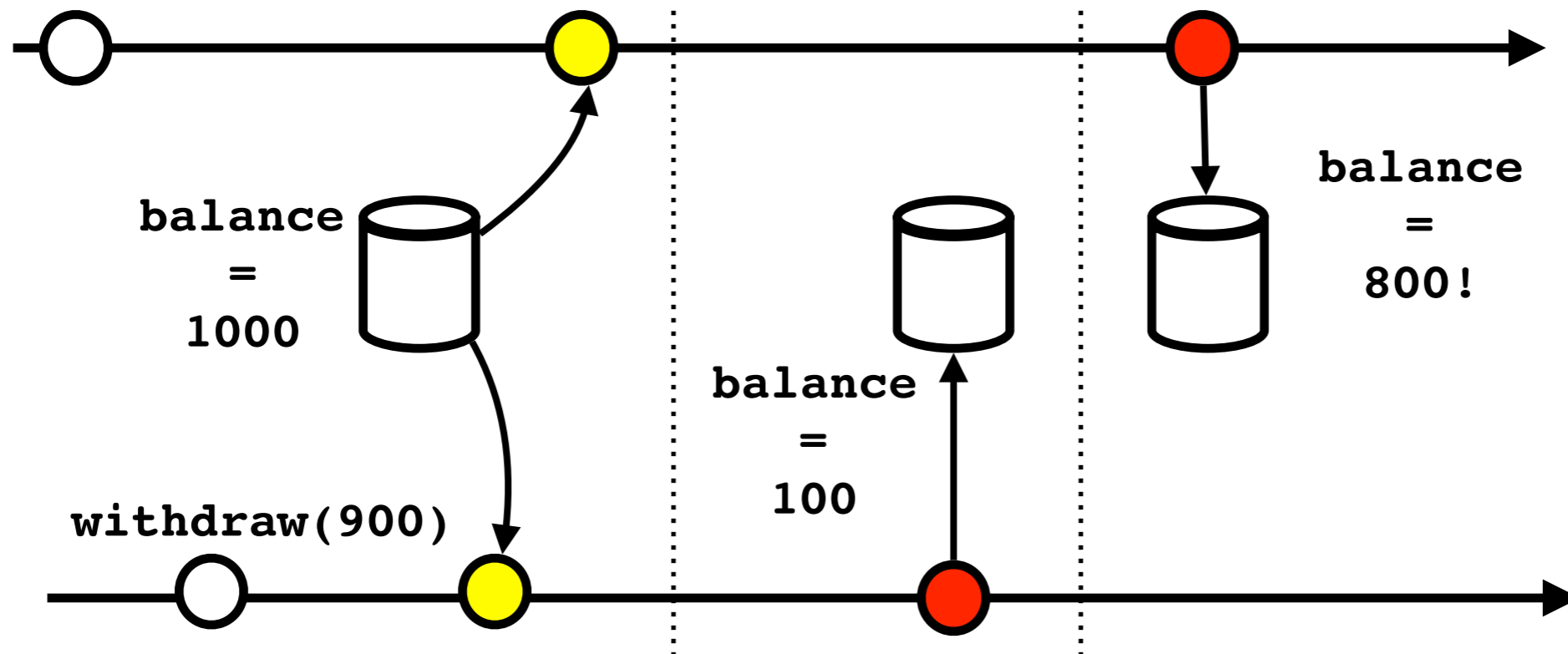
```
# Pseudo-code
withdraw(account, amount) {
    balance = getBalanceInDB(account)
    if balance >= amount
        setBalanceInDB(account, balance - amount)
        return OK
    else
        return NOT_AUTHORIZED
    end
end
end
```

- Assume this is a server-side procedure in your bank that can be executed concurrently, and that **getBalanceInDB** and **setBalanceInDB** execute as individual DB transactions.
- If two (or more client programs) invoke **transfer** concurrently, can you “steal” money from the bank?

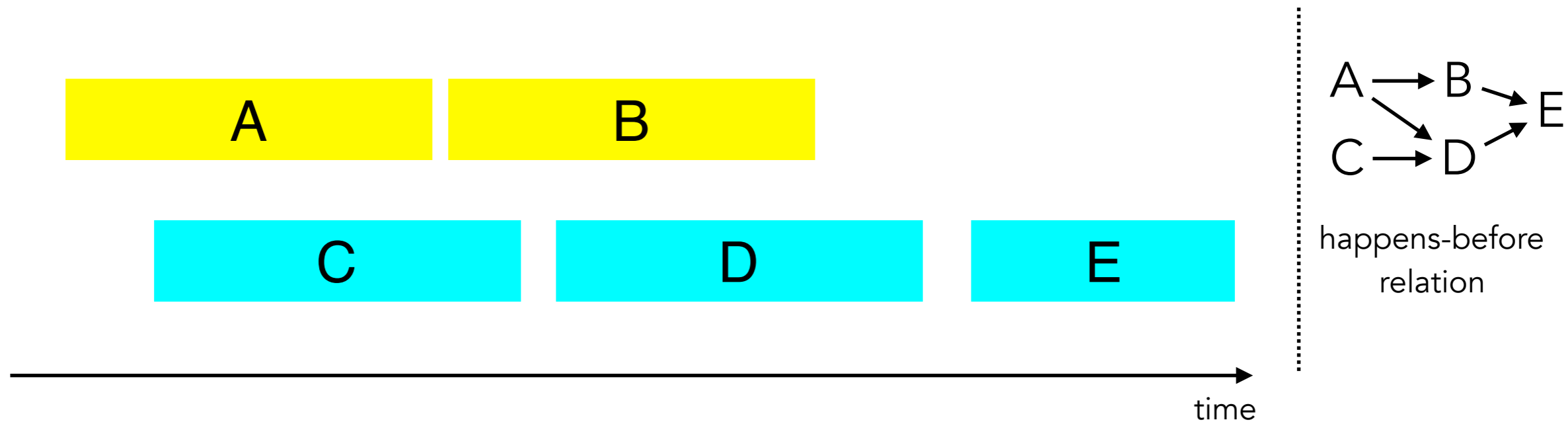
# Simple “bank account” example (2)

```
# Pseudo-code
withdraw(account, amount) {
  balance = getBalanceInDB(account)
  if balance >= amount
    setBalanceInDB(account, balance - amount)
    return OK
  else
    return NOT_AUTHORIZED
  end
end
end
```

withdraw(200)



# Serializability



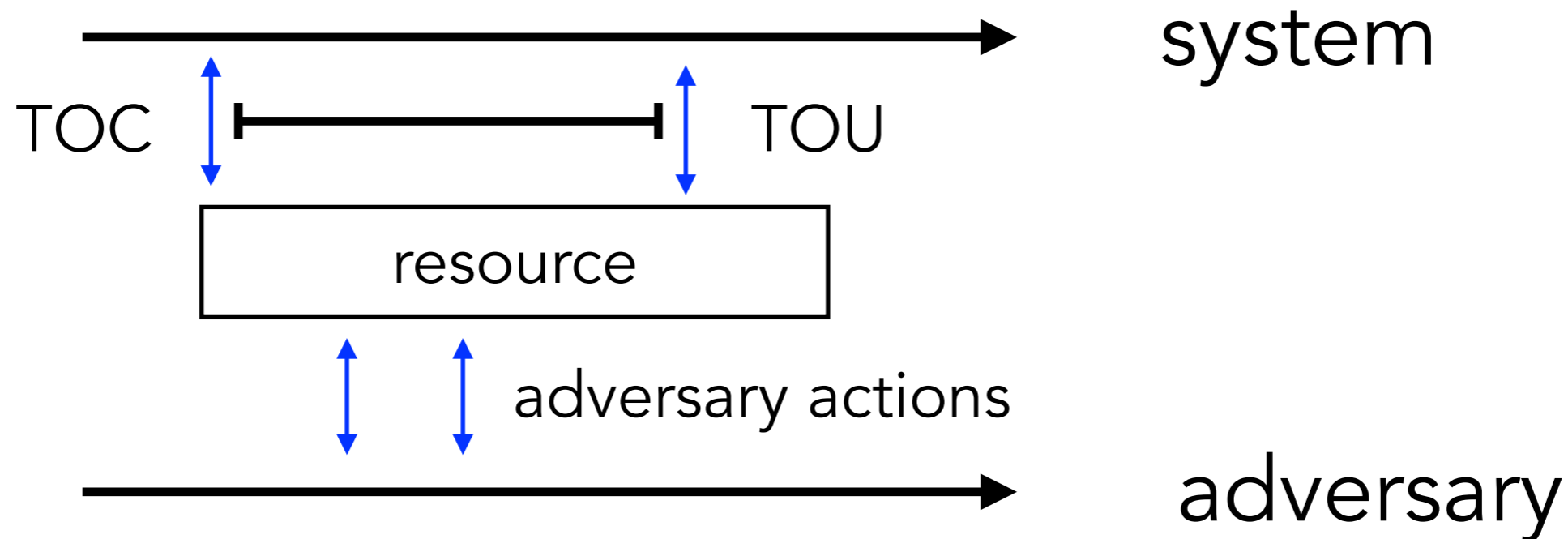
- **Serializability** - the strictest form of isolation
  - The logical effect of a set concurrent transactions must be equivalent to that of a sequential execution of them, i.e., any permutation that preserves the “happens-before” relation.
  - In the example: effect may be equivalent to ABCDE or CADBE for instance but not ABDCE, CDABE, or ABCED
- Relaxed in many cases, due to complexity of implementation and performance / availability trade-offs

# Race condition — multithreaded programs

```
public static class Counter extends HttpServlet {
    static int count = 0;
    protected void doGet(HttpServletRequest in, HttpServletResponse
out) throws ServletException, IOException {
        out.setContentType("text/plain");
        PrintWriter p = out.getWriter();
        count++; // read followed by write implicit
        p.println(count + " hits so far!");
    }
}
```

- Suppose two requests execute simultaneously . As in the previous example, it is easy to see the returned sequence id can be the same for both threads.
- An instance of [CWE-366](#) / [CWE-567](#). More on multithreaded programs later in this class.

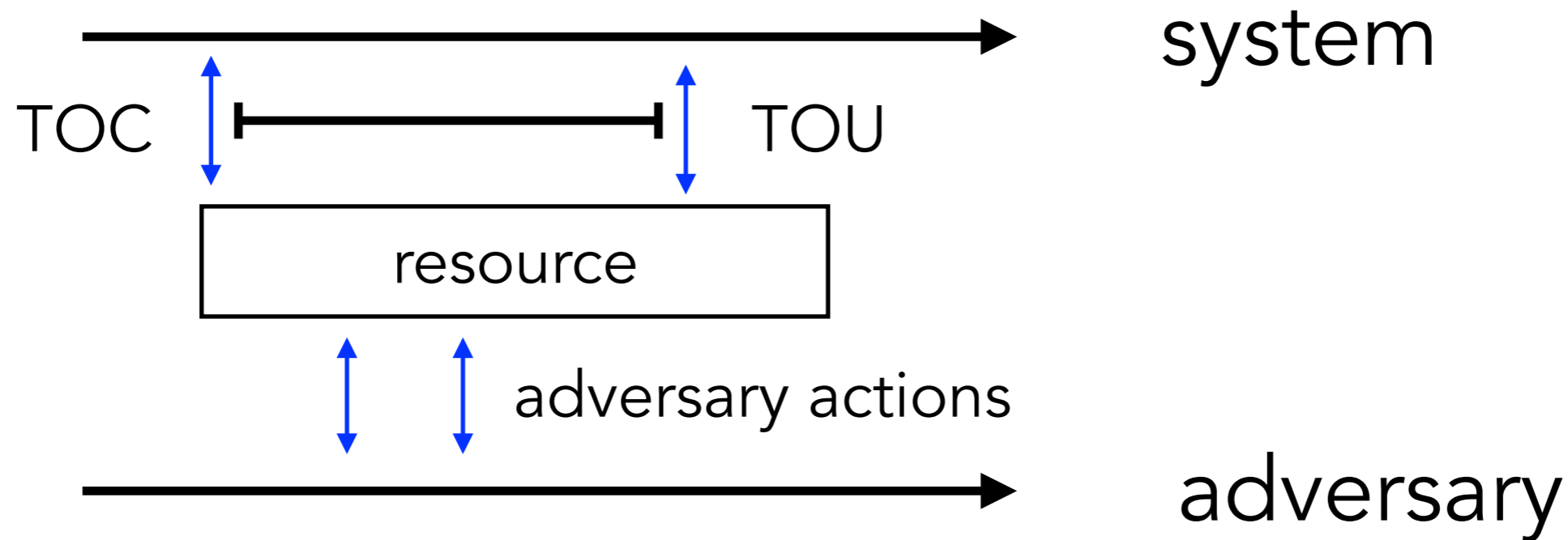
# TOCTOU race conditions (CWE-367)



## ■ TOCTOU: Time-Of-Check, Time-Of-Use - CWE-367

- *“The software **checks the state of a resource before using that resource, but 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.**”*

# TOCTOU race conditions



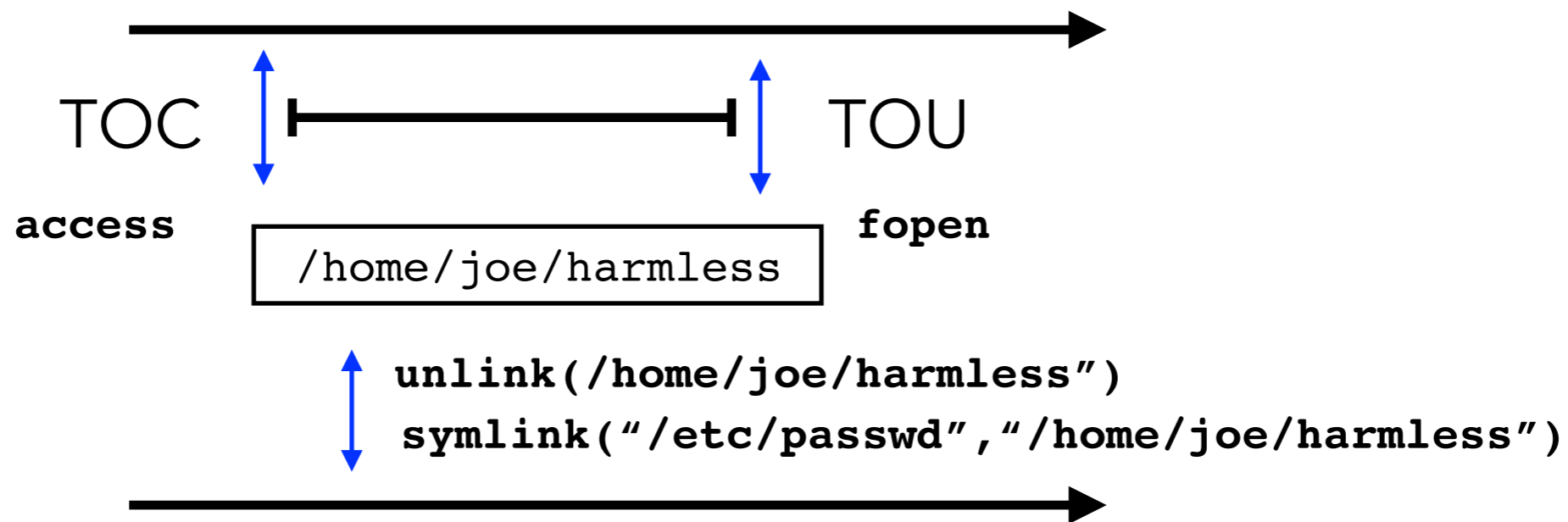
- **The program:**
  - ✦ first checks for availability/safety of the resource - **time-of-check (TOC)**
  - ✦ and subsequently starts using it — **time-of-use (TOU)**
- **Adversary**
  - ✦ in between TOC and TOU, it accesses or modifies the resource state invalidating the TOC assumptions.
  - ✦ system deviates from expected behavior to serve the adversary purposes

# TOCTOU — classic POSIX example

```
// TOC: access call
if (access(file, W_OK) == 0) {
    // TOU: fopen call
    FILE* f = fopen(file, "w");
    writeToFile(f);
    fclose(f);
} else {
    fprintf(stderr, "Cannot write to '%s'\n", file);
}
```

- Program runs with elevated “setuid” privileges
  - ✱ Effective user id (EUID) may be root
  - ✱ Real user id (RUID) typically has less privileges
- **access** call takes into account the real UID
- ... but **fopen** call takes into account the EUID: any file can be written by root ...
- **WHAT CAN GO WRONG?**

# TOCTOU - classic POSIX example



- Recall:
  - **access** call takes into account the real UID
  - ... but **fopen** call takes into account the EUID: any file can be written by root ...
- Adversary changes resource to point to EUID-accessible file like “/etc/passwd”
- A closely related TOCTOU instance — [CWE-363](#) - “Race Condition Enabling Link Following”



# TOCTOU — code auditing

## Flawfinder output

```
toctou.c:16: [4] (race) access:
```

```
  This usually indicates a security flaw. If an attacker can change anything along the path between the call to access() and the file's actual use (e.g., by moving files), the attacker can exploit the race condition (CWE-362/CWE-367). Set up the correct permissions (e.g., using setuid()) and try to open the file directly.
```

```
[...]
```

```
toctou.c:18: [2] (misc) fopen:
```

```
  Check when opening files - can an attacker redirect it (via symlinks), force the opening of special file type (e.g., device files), move things around to create a race condition, control its ancestors, or change its contents? (CWE-362).
```

# TOCTOU – Temporary files

```
if (tmpnam(filename)) { // TOC
    FILE* tmp = fopen(filename, "wb+"); // TOU
    while((recv(sock, recvbuf, DATA_SIZE, 0) > 0) && (amt != 0))
        fwrite(recvbuf, 1, DATA_SIZE, tmp);
}
```

- Other classic examples of TOCTOU vulnerabilities relate to the use of temporary files.
- **tmpnam** (and other functions/variants):
  - generates the name of a temporary file that does not exist *but does not create it! File can be created externally in-between the tmpnam and the fopen call (which was supposed to create it)*
  - in some implementations the file name is also relatively predictable.
- Slightly more safe
  - **tmpfile** : generates name and opens the file by truncating it if it already exists — attacker may create the “original” file with relaxed permissions though
- Safer
  - **mkstemp**: generates name and forces creation of temporary file, otherwise fails.

# TOCTOU [Viega and McGraw, Chap 9]

## advice and programming recipes

- Avoid calls that take filenames to perform a check.
- Use file descriptors instead whenever possible:
  - e.g. open + fstat in place of stat + open
  - Not always feasible: no file-descriptor based alternatives to mkdir, unlink, ...
- In this case take measure to ensure the files at stake are located in a directory owned by the RUID
- File locking may be useful but does not work on file systems like NFS

# Concurrency is hard to get right ...

- There are many explicit concurrency primitives for processes / threads that required careful programming
  - semaphores, message queues, shared memory, locks, monitors, condition variables, ...
- These are easily prone to bad use leading to race conditions, deadlocks, functional errors ...
  - For instance in multithreaded programs: [“Concurrent Bug Patterns and How to Test Them”](#), E. Farchi, Y. Nir, S. Ur, PDPS, 2003

# Alternative concurrency abstractions

- Concurrency can be based on safe abstractions that alleviate the burden from the programmer.
- Brief reference to a few examples:
  - Rust language: memory ownership concept built-in prevents memory races by construction (among other advantages)
  - Go, Erlang, Scala actors, ... : “clean” concurrency based on message-passing / actor model / co-routines
  - Software Transactional Memory (STM): support for memory transactions

# Concurrent program verification

- **Bugs/vulnerabilities are NOT revealed “out-of-the-box”:**
  - **Execution:** interleavings are too many and can be fine-grained. For example, thread/process context switches are non-deterministic, irreproducible and uncontrollable. Only certain (sometimes rare) executions may lead to erroneous behavior.
  - **Debugging** is frequently useless. The execution of a debugger itself affects the schedule of a running program. **“Heisenbugs” are common** (“a bug that disappears or alters its behavior when one attempts to probe or isolate it”).
  - **Standard testing** cannot predict or programmatically control the interleavings.
- **Special-purpose, relatively complex methodologies can be used, but face scalability/state-explosion problems ...**

# Example: exposing a race condition

```
data.value = 0;
Thread t1 = new Thread(() -> { data.value++; }),
        t2 = new Thread(() -> { data.value++; });
t1.start(); t2.start();
t1.join(); t2.join();
assert data.value == 2;
```

```
class Data {
    int value;
}
```

- Multi-threaded Java code — note that:
  - start() : begins thread execution (asynchronously)
  - join() : waits for thread to terminate
- The two threads increment `data.value` that is initially 0
- Is there a race? Is value always 2 after both t1 and t2 end ?

# Example: exposing a race condition (2)

```
data.value = 0;  
Thread t1 = new Thread(() -> { data.value++; } ),  
        t2 = new Thread(() -> { data.value++; } );  
t1.start(); t2.start();
```

data.value++;  
equivalent to

```
data.value = data.value + 1
```

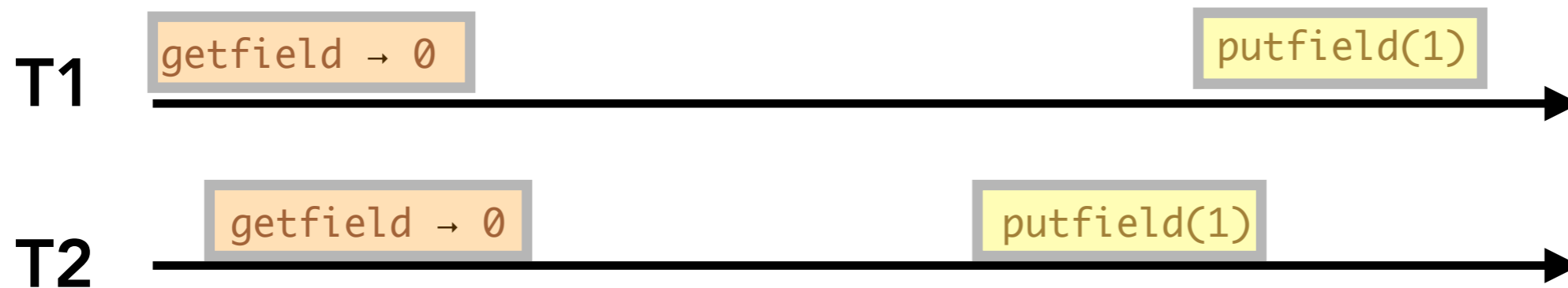
```
class Data {  
    int value;  
}
```

```
0: aload_0  
1: dup  
2: getfield      #34 // Field qses/Data.value:I  
5: iconst_1  
6: iadd  
7: putfield     #34 // Field qses/Data.value:I
```

JVM  
bytecode



# Example: exposing a race condition (3)



```
synchronized(data) {  
    data.value++;  
}
```

Code that  
fixes the race  
condition

- Result can be 1 if both reads precede the writes.
- Fix: synchronize access to data / ensure mutual exclusion during update
- But if you can run the incorrect code > a million times ... and the value will be 2
- **Small segment of code in each thread tends to run in sequence without interleaving. Context switches are too coarse-grained ...**

# Inducing noise to expose bugs...

```
int v = data.value;  
Thread.yield();  
data.value = v + 1;
```

- “Noise” = calls that lead to context switches like `Thread.yield()` / `sleep()`
- Noise at thread interference points increases the probability of inducing more fine-grained schedules / exposing bugs
  - For instance see “[Testing Concurrent Java Programs using Randomized Scheduling](#)”, S.D. Steller, RV’02

# Testing approaches

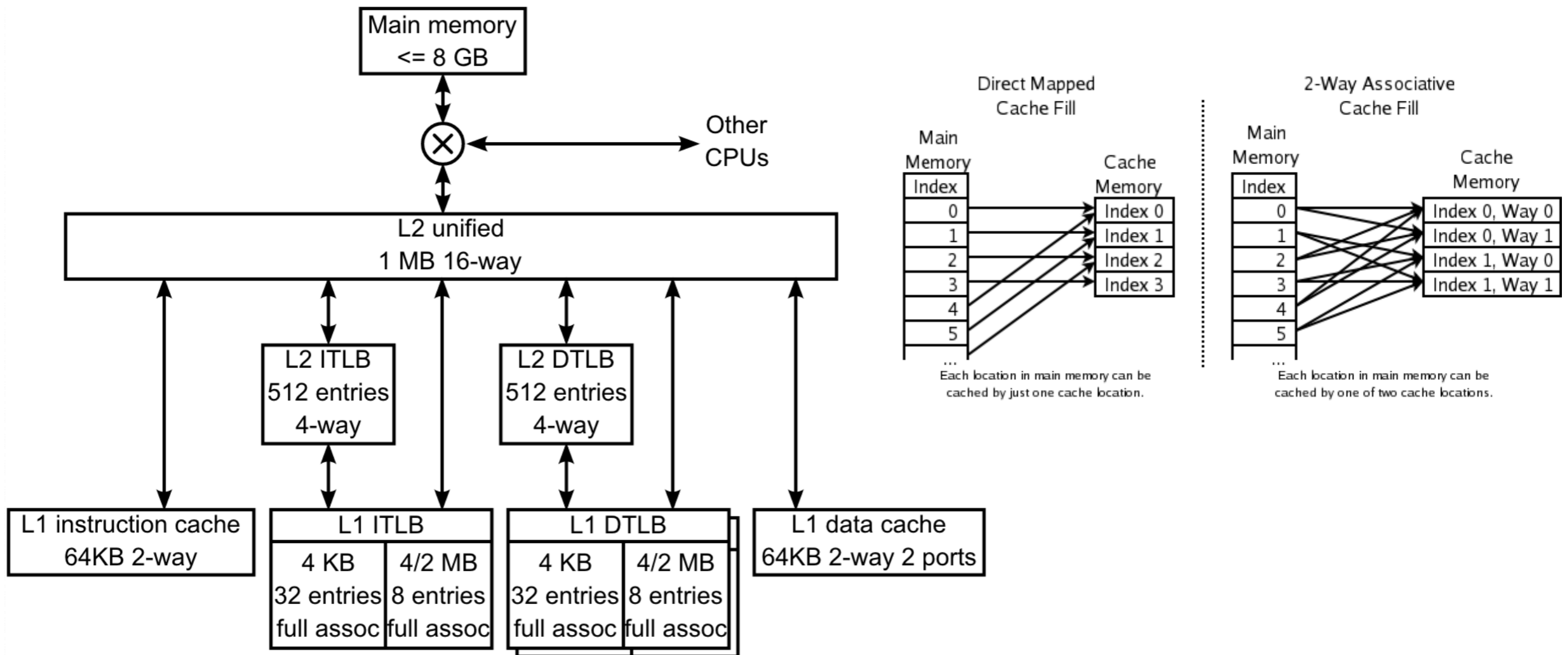
- Possible thread interleavings can be explored by systematically intercepting thread interference points on
  - shared data access
  - use of multi-threading primitives
- Example systems
  - CHES — “[Finding and Reproducing Heisenbugs in Concurrent Programs](#)”, Musuvathi et al., OSDI’08
  - “[Cooperari: A Tool for Cooperative Testing of Multithreaded Java Programs](#)”, Marques et al, PPPJ’14
  - We’ll go through an overview of Cooperari next (overview slides [here](#)).

# Covert channels / micro-architectural attacks

## ■ [CWE-385](#): 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.”*
- For instance, password timing attacks are based on the fact that password checking may not occur in constant time / independently of password length.
- Some relevant covert channels inherently rely on **concurrency at the hardware micro-architectural level**:
  - **memory hierarchy, in particular the shared use of caches by multiple cores/processors**
  - **Processor features: hyper-threading, branch-prediction, out-of-order execution**

# Memory hierarchy / caches



- Example: AMD Athlon K8

- Image source: [Wikipedia entry for "CPU cache"](#)

# Covert-channel cache attacks

## ■ Outline

- Cache hits and misses have significantly different latencies  $\Leftrightarrow$  memory access time is correlated with cache hits/misses.
  - A “spy program” races the cache by flushing it altogether or accessing memory that leads to the eviction of cache lines by other programs.
  - The execution of the “victim programs” will incur cache misses.
  - Spy can “walk” through the cache again and know what cache lines have been accessed (if it also has misses)
- There are several variants, e.g., check the [libflush](#) library.

Hyperthread logical processor 1

Hyperthread logical processor 2

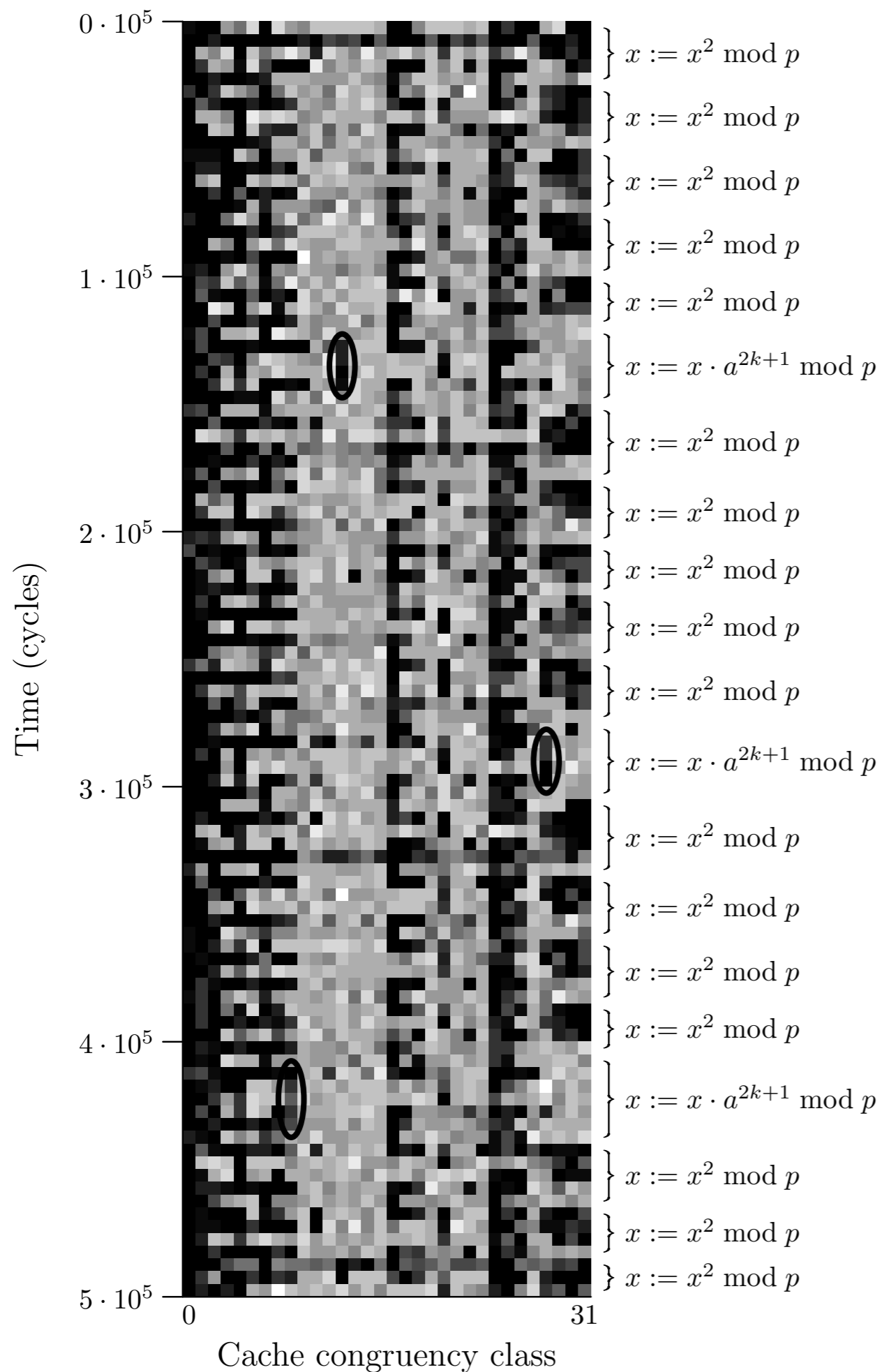
OpenSSL performs RSA crypto operations leaving cache miss usage trail revealing branches taken while executing

Malicious program walks cache in loop dirtying cache lines and detecting which miss and reflect code paths in logical processor 1

Shared level 1 cache

System memory

- Image source: “[Concurrency and security](#)”, R. Watson, Security course at Cambridge



Source: "Cache Missing for Fun and Profit"  
Colin Percival

FIGURE 2. Part of a 512-bit modular exponentiation in OpenSSL 0.9.7c. The shading of each block indicates the number of cycles needed to access all the lines in a cache set, ranging from 120 cycles (white) to over 170 (black). The circled regions reveal information about the multipliers  $a^{2k+1}$  being used.



# Meltdown

```
1 raise_exception();  
2 // the line below is never reached  
3 access(probe_array[data * 4096]);
```

Listing 1: A toy example to illustrate side-effects of out-of-order execution.

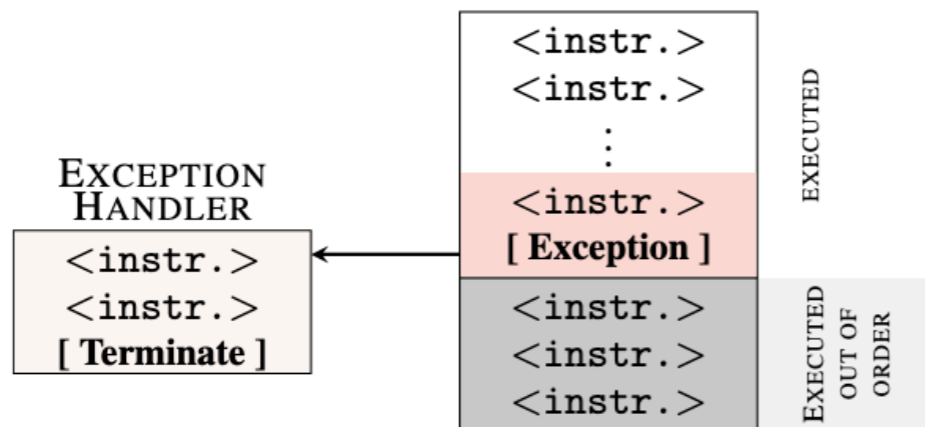


Figure 3: If an executed instruction causes an exception, diverting the control flow to an exception handler, the subsequent instruction must not be executed. Due to out-of-order execution, the subsequent instructions may already have been partially executed, but not retired. However, architectural effects of the execution are discarded.

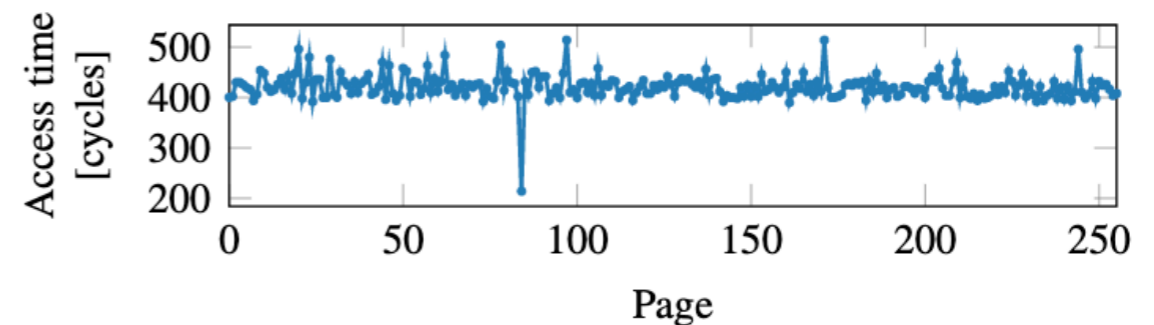


Figure 4: Even if a memory location is only accessed during out-of-order execution, it remains cached. Iterating over the 256 pages of probe\_array shows one cache hit, exactly on the page that was accessed during the out-of-order execution.

Lipp et al., “Meltdown: Reading Kernel Memory from User Space”, 27th USENIX Security Symposium, 2018

# Spectre

```
if (x < array1_size)
    y = array2[array1[x] * 4096];
```

Listing 1: Conditional Branch Example

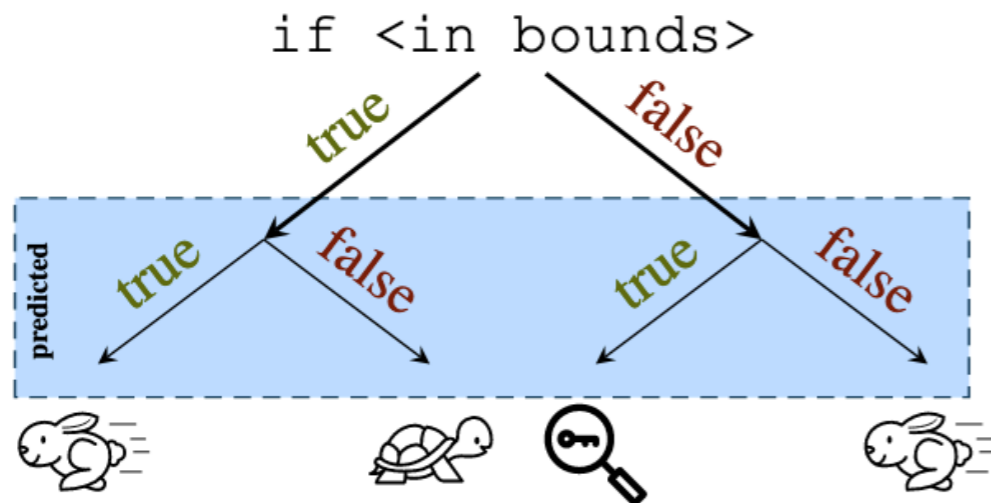


Fig. 1: Before the correct outcome of the bounds check is known, the branch predictor continues with the most likely branch target, leading to an overall execution speed-up if the outcome was correctly predicted. However, if the bounds check is incorrectly predicted as true, an attacker can leak secret information in certain scenarios.

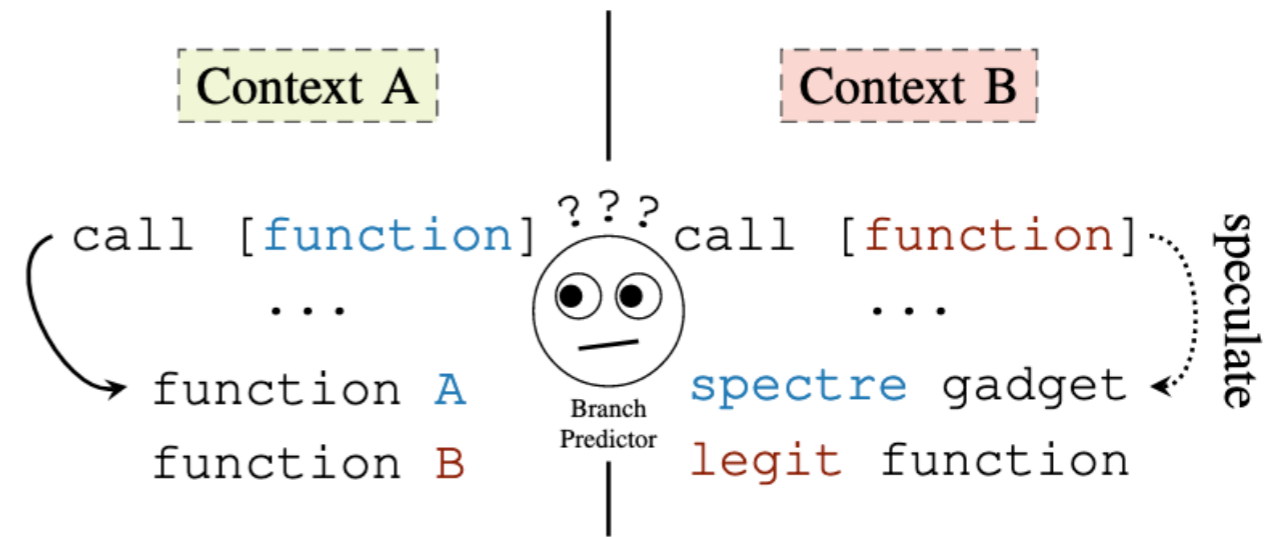


Fig. 2: The branch predictor is (mis-)trained in the attacker-controlled context A. In context B, the branch predictor makes its prediction on the basis of training data from context A, leading to speculative execution at an attacker-chosen address which corresponds to the location of the Spectre gadget in the victim's address space.

Rocher et al., “Spectre Attacks: Exploiting Speculative Execution”, 40th IEEE Symposium on Security and Privacy, 2019