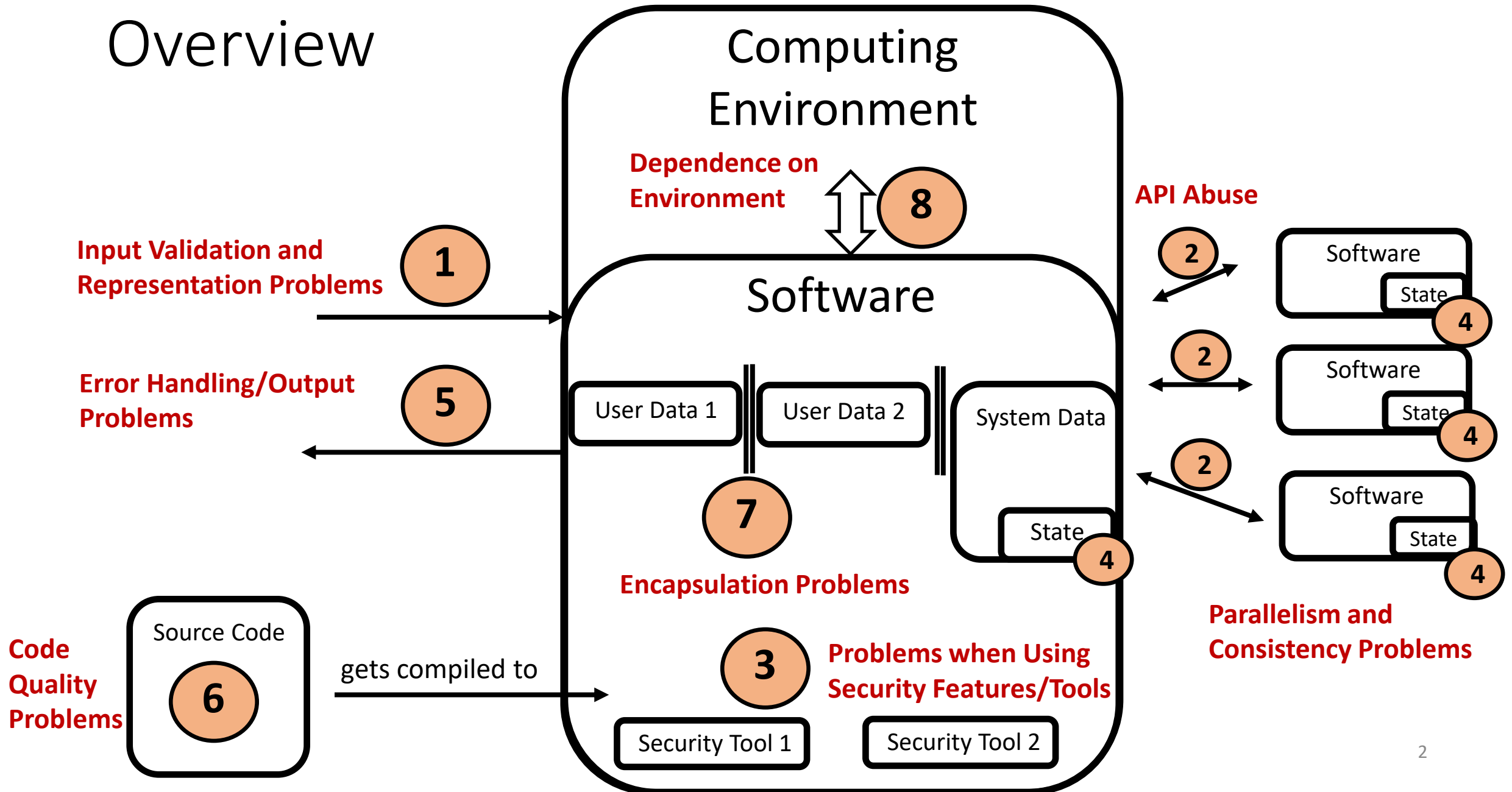


Software Security 12

Sven Schäge

Overview



Learning Goals

- Be able to identify side-channel attacks and propose counter measures.
- Be able to identify fault injection attacks and propose counter measures.
- Be able to identify timing vulnerabilities in code.
- Be able to identify cache timing vulnerabilities in code.
- Understand the practical difficulties of writing constant-time code.
- Be able to identify and reflect on the fundamental costs and trade-offs that protection mechanisms against side-channel attacks involve.
- Be able to reason abstractly about the possibility of side-channel attacks by using a process-oriented or state-oriented abstraction of computing devices.
- Know sources of timing side-channels and identify them in (distributed systems) and reason about their feasibility.
- Know typical observations exploited in side-channel attacks.

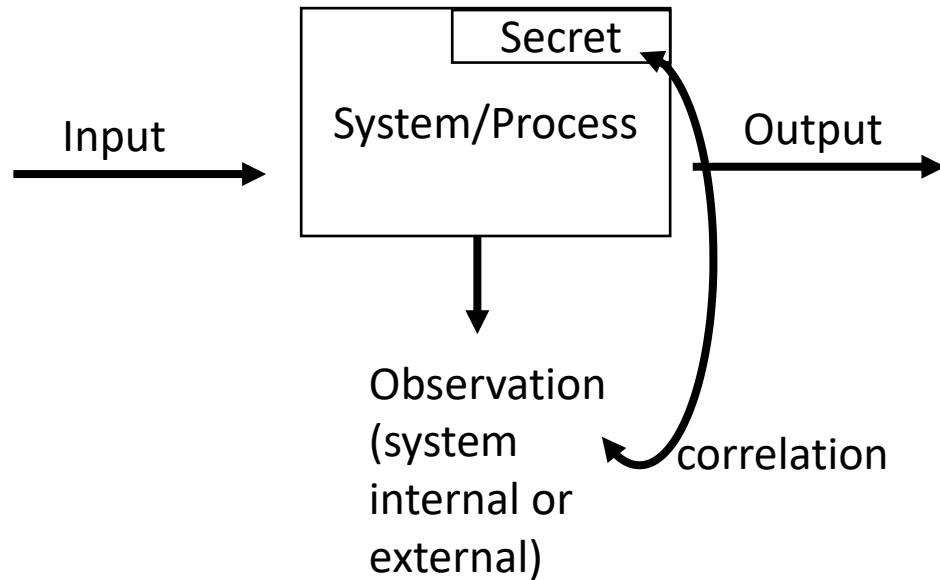
Overview

- Side-Channel Attacks
- Fault-Injection Attacks
- Basic Attack Idea: Computing Secrets Stepwisely
- Timing Attacks
- Cache Timing Attacks

Side-Channel Attacks

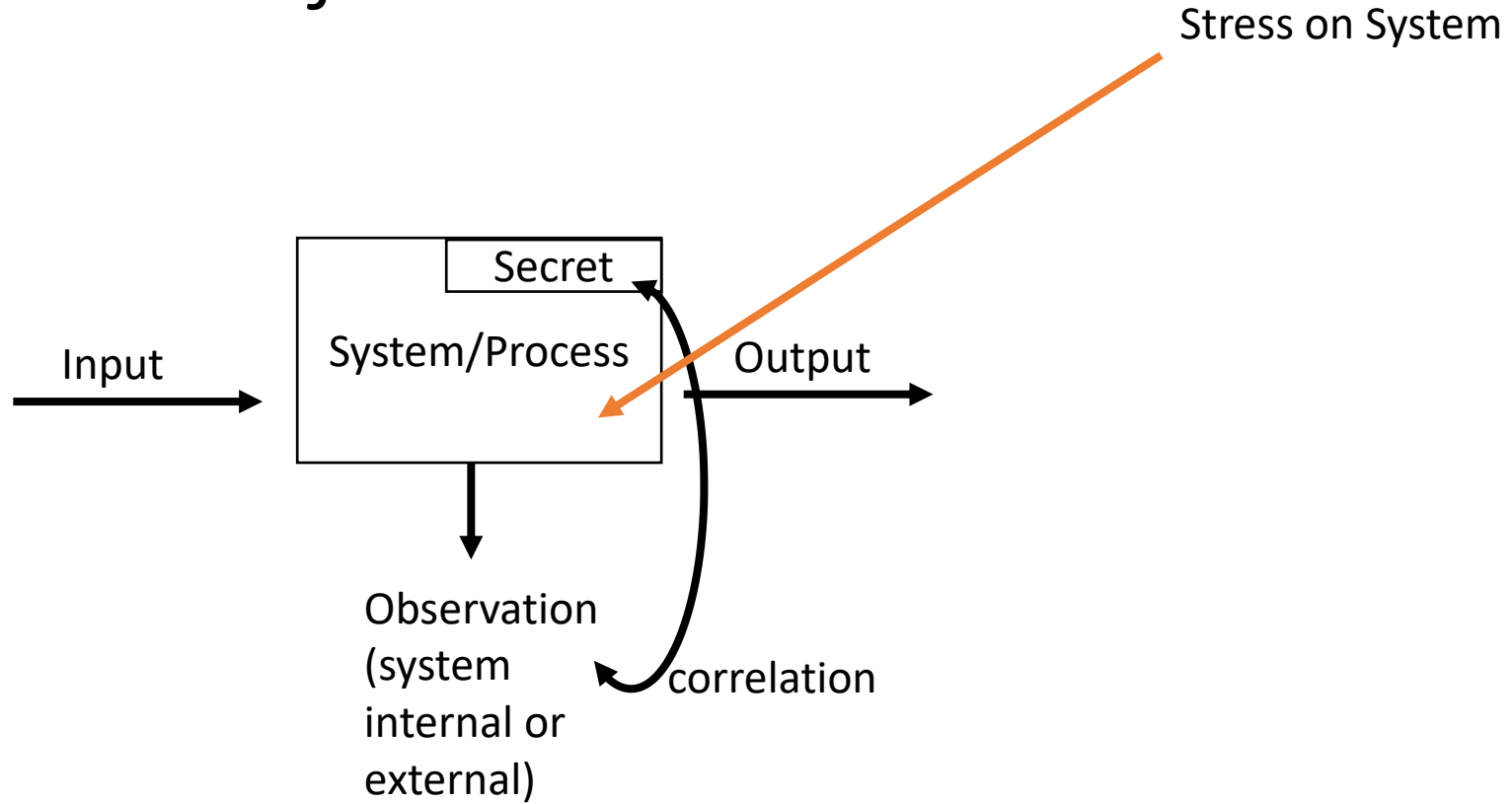
- Wikipedia:
 - In computer security, a side-channel attack is any attack based on **extra information** that can be gathered **because of the fundamental way a computer protocol or algorithm is implemented**, rather than flaws in the design of the protocol or algorithm itself (e.g. flaws found in a cryptanalysis of a cryptographic algorithm) or minor, but potentially devastating, mistakes or oversights in the implementation.
 - Conceptually, when studying side-channel attacks we consider a scenario where the **extra information can reveal information on a secret**.

Side-Channel Attacks



- Typical Goal: Violate Confidentiality
- Observation
 - Timing
 - Power
 - Electromagnetic Analysis
 - Cache Attacks
 - Acoustic Analysis
 - Fault Injection Attacks
 - Error Signal

Fault Injection Attacks



Basic Attack

- Assume secret s has 128 characters, each coming from a set of 26 letters.
- Brute-forcing the secret will require $(26)^{128}$ worst-case.
- However, if we can check the correctness of each character (which is the same as being able to check the correctness of each prefix for each prefix length from lower length to higher length) separately, the complexity is reduced to $26 \cdot 128$

Observable Side-Channels

- Side-channels may not directly leak information on the secret but often after a statistical analysis of many runs
- Simple Power-Analysis
 - In a single power trace, cryptographic keys can be read out immediately.
- Differential Power-Analysis
 - Many power traces with differing input data are created.
 - A subkey is guessed.
 - According to this guess, the traces are typically partitioned into two classes C1 and C2.
 - These classes are then compared such that:
 - If the guess is correct, the comparison will show a clear statistical indication (think averaging the traces of the classes and subtracting them)
 - Otherwise, no indication will be visible.
 - In both classes all other probabilistic processes act as noise that is believed to be the same for each class.

General Countermeasures Side-Channel Attacks

- Suppress extra information
 - Better isolation, shielding
 - Introduce noise
 - Constant-time (isochronous) computations, PC-security (program counter security model - program path does not rely on secret data), branch-free code
 - Compiler optimization can be problematic
- Disturb correlation between extra information and data without destroying functionality
 - Often exploits some kind of homomorphisms to implement masking/blinding
 - Randomized executions
 - Artificial aborts

Timing Attacks

checkAPIKey

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>

// Function to check if the inputKey is correct, returns 1 if
// correct, 0 otherwise
int checkApiKey(const char *inputKey, const char *correctKey) {
    // Get the length of the inputKey
    size_t inputLen = strlen(inputKey);
    // Get the length of the correctKey
    //(assuming it has the same length as inputKey)
    size_t correctLen = strlen(correctKey);
```

```
    // Check characters in a loop
    for (size_t i = 0; i < inputLen; ++i) {
        // If a mismatch is found, return 0
        if (inputKey[i] != correctKey[i]) {
            return 0;
        }
    }
    // All characters matched, return 1
    return 1;
} //end checkApiKey
```

checkAPIKey

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>

// Function to check if the inputKey is correct, returns 1 if
// correct, 0 otherwise
int checkApiKey(const char *inputKey, const char *correctKey) {
    // Get the length of the inputKey
    size_t inputLen = strlen(inputKey);
    // Get the length of the correctKey
    //(assuming it has the same length as inputKey)
    size_t correctLen = strlen(correctKey);
```

memcmp comparison style
'early exit'



```
// Check characters in a loop
for (size_t i = 0; i < inputLen; ++i) {
    // If a mismatch is found, return 0
    if (inputKey[i] != correctKey[i]) {
        return 0;
    }
}

// All characters matched, return 1
return 1;
} //end checkApiKey
```

Sources of Timing Side-Channels

- Non-Local memory access
 - Cache
- Conditional Jumps
- Mathematical Operations
 - Integer Division
 - Shifts and Rotations (as Loops->No Barrel Shifter)

Random Waiting Times?

- set finishTime = now + ESTIMATED_DURATION_OF_COMPUTATION
- checkApiKey(inputKey, correctKey);
- sleep until finishTime

Finish Time

- `checkApiKey(inputKey, correctKey);`
- sleep for random amount of time
- continue with rest of program...

Problems

- How to choose ESTIMATED_DURATION_OF_COMPUTATION? Too high a value is a problem for performance
- Even single-core computers execute multiple tasks concurrently using context switches. While the program is sleeping, the CPU is free to work on other tasks. In other words, the amount of time the code spends sleeping affects how quickly other tasks finish... which is something a skilled attacker can measure.

Worst-Case Execution and Accumulation

```
bool insecureStringCompare(const void *a, const void *b, size_t length) {  
    const char *ca = a, *cb = b;  
    for (size_t i = 0; i < length; i++)  
        if (ca[i] != cb[i])  
            return false;  
    return true;  
}
```

```
bool constantTimeStringCompare(const void *a, const void *b, size_t length) {  
    const char *ca = a, *cb = b;  
    bool result = true;  
    for (size_t i = 0; i < length; i++)  
        result &= ca[i] == cb[i];  
    return result;  
}
```

Busy Loop?

- Replace sleep with a busy loop? Now the CPU will be busy from the time the checkApiKey function is started until finishTime.
- Modern CPUs have multiple physical cores and features like simultaneous multithreading (Intel calls it “hyper-threading”). Even if one logical core is 100% occupied by a thread, other tasks might still be executed in parallel on other cores. Depending on what it’s doing exactly, your code will produce a subtle but measurable pattern of interference in the performance of other threads and processes on the same hardware.

Competition for Resources Leaks!

- Processes/threads will compete for space in the same CPU caches, evicting each others' entries. This leaks information about what locations in memory are being accessed.
- Competition for particular execution units and time can reveal what kinds of computations are occurring (e.g. integer arithmetic/logic vs floating-point).

Solution: Constant-Time

- We can't really hide what kinds of computations we're doing, but we can write our code so that secret information has no influence on how we use hardware resources. In other words, even if an attacker can see exactly what hardware resources our program uses and when, they still won't learn anything secret.
- There is no need for the execution time to be constant. Variations in execution time have no impact on security as long as those variations have no relation to any secret information.
- Execution time is not the only way for information to leak. What the code does during that time is also quite important.
- Desired: Secret-independent resource usage.

Golden Rule of Constant-Time Code

- Secret information may only be used in an input to an instruction if that input has no impact on what resources will be used, how the resources will be used, and specifically for how long.

Implications

- In practice, this golden rule has a few implications:
 - Secret values cannot be used to decide what code to execute next (i.e. it cannot be used as a condition to a branch instruction)
 - Secret values cannot be used to decide what memory address to access.
 - Secret values cannot be used as input to variable-time instructions like DIV on x86 (smaller numbers divide faster).

Problem

- Whether a particular instruction is “constant-time” can vary by architecture and by processor model, and there isn’t any official documentation for this behavior.

Practical Problem

- So how can we **write** code in high-level languages that will still follow the golden rule when compiled?
- This is challenging: Today's languages and compilers weren't really built for this. They are built for good efficiency.
- The usual approach is to write code that looks constant-time.
- Document which variables you want to keep secret. Avoid using those variables in the conditions of if-statements and loops. Avoid using those variables when calculating an array index (since the array index determines what memory address in the array to access). Avoid using those variables in operations that probably compile into variable-time instructions, like division or modulus.



Beware of Compilers!

- But this doesn't always work. The compiler might decide that our code would be faster if it used variable-time instructions.
- Moreover, an optimizing compiler might observe that we are trying to avoid an if-statement, and the compiler puts the if-statement back in because it knows it will be faster.

The Problem is Widespread and also works in Remote Settings: Token-based Authentication

- Token-based authentication schemes (i.e. how you would typically implement 'remember me' cookies or password reset URLs) typically suffer from a design constraint can leave applications vulnerable to timing attacks.

SQL Example

- When you login with a username and a password, typically you do something like this:
 - Run a SQL query like `SELECT userid, password_hash FROM user_accounts WHERE username = :username`
 - Validate the password e.g. `password_verify($password, $storedPasswordHash)` special function: no early exit
 - If the password is valid, you would then associate the current session with the user's ID
- Contrast this with a traditional token-based authentication scheme (tokens act as secrets).
 - Run a SQL query like `SELECT tokenid, userid FROM password_reset_tokens WHERE token = :token AND NOW() < expire_time`
 - If you get a result, the token was valid

Comparison via SQL engine, typically implemented via memcmp: early exit timing vulnerability

Cache Timing Attacks

Cache

- A cache is a small amount of fast 'intermediate' memory; when you read from memory, the contents are placed in this fast memory (possibly along with adjacent locations); if you read from the location again, you read it from the fast memory (which, of course, proceeds much faster).
- Hence, if you read a location that you've read before, it goes much faster than if you read a location which you haven't; hence, you're not constant time.
- Modern CPUs can process data much faster than memory can respond (perhaps by a factor of 100), hence caches are pretty much ubiquitous. Some lower end microcontrollers might not have them; but they're an exception.

Computer

- CPUs can compute very fast as compared to the speed of delivering data from and to RAM.
- Memory operations are an efficiency bottleneck in modern architectures.
- RAM is large but cheap, register are small and expensive.
- Caches represent additional intermediate stages of this memory hierarchy.
 - With increasing x Level x caches have more storage capacity than on Level $x-1$ but are cheaper per memory unit. However, Level x caches are faster.
- Having operands be present in caches of Level x with low x can greatly reduce the overall execution time!
 - They can very quickly get loaded into the CPU's registers.
- Example: Intel Skylake architecture (2-28 cores)
 - L1 cache 64 KB per core
 - L2 cache 256 KB per core (1 MB per core for Skylake-X)
 - L3 cache Up to 38.5 MB shared
 - L4 cache 128 MB of eDRAM (on Iris Pro models)

Example: Intel Skylake architecture (2-28 cores)

- Sizes
 - L1 cache 64 KB per core
 - L2 cache 256 KB per core (1 MB per core for Skylake-X)
 - L3 cache Up to 38.5 MB shared
 - L4 cache 128 MB of eDRAM (on Iris Pro models)
- Intel i7-6700 (Skylake), 4.0 GHz (Turbo Boost), 14 nm. RAM: 16 GB, dual DDR4-2400 CL15 (PC-19200).
- L1 Data cache = 32 KB, 64 B/line, 8-WAY.
- L1 Instruction cache = 32 KB, 64 B/line, 8-WAY.
- L2 cache = 256 KB, 64 B/line, 4-WAY
- L3 cache = 8 MB, 64 B/line, 16-WAY
- L1 Data Cache Latency = 4 cycles for simple access via pointer
- L1 Data Cache Latency = 5 cycles for access with complex address calculation.
- L2 Cache Latency = 12 cycles
- L3 Cache Latency = 42 cycles (core 0) (i7-6700 Skylake 4.0 GHz)
- L3 Cache Latency = 38 cycles (i7-7700K 4 GHz, Kaby Lake)
- RAM Latency = 42 cycles + 51 ns (i7-6700 Skylake)
- Note: It's possible that L2 Cache Latency can be 11 cycles in some cases. But dependency chain workload shows 12 cycles.

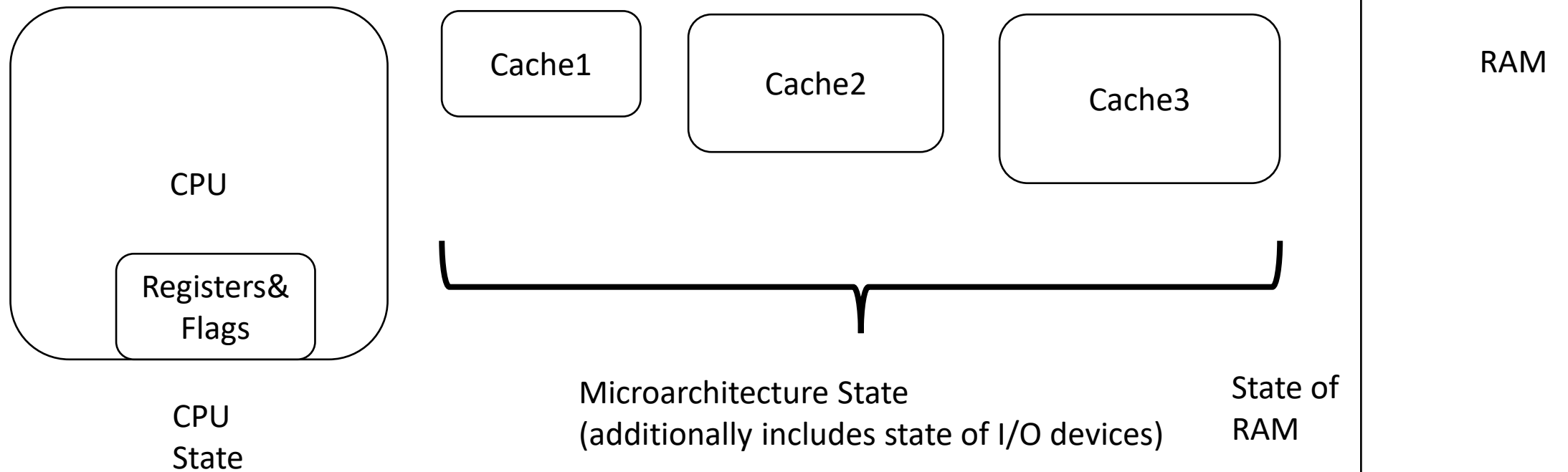
Cache

- The cache consists of memory blocks that have been loaded from the RAM, so called cache lines that have size 32, 64, 128 kB.
- If the cache is full, a cache replacement policy determines which cache entry to overwrite. Typically, this is the oldest entry.
- Thus, cache lines usually contain a time indication of how old entries are.
- According to this system addresses are not sorted and checking whether an address is already present in the cache is more complex.
- To find entries quickly nevertheless, caches use surjective hash function to map input addresses to an index set which in turn holds several lines of memory.
- Checking if an address is present in the cache thus amounts to determining the SetIndex via a hash function and then searching through the set.
- $\text{SetIndex} = \text{hash}(\text{Address})$

Address	DataContent

SetIndex	Line within Set	Data Content	Time

State of Computers



Isolation->Separation of Memory

- Typically, caches are **shared resources** that are used by all processes on a computer.
- Caches cannot be accessed directly by programs but loading and storing is done by hardware.
- Isolation via memory separation is often simply done by reserving distinct RAM address ranges for processes.
 - Kernel processes that are highly trusted by operation systems have a memory range assigned to them that user space processes cannot access.
 - To isolate different users from each other, user space processes should also be separated so that UserX is not able to address the data of UserY in RAM.
 - The operating system checks before a memory access whether the calling process has the required authorization.
- Using a hierarchical system, complex access structures can be defined that say who should be able to access some data.
- This is particularly important when relying on computer systems that host processes from different users like in cloud computing.

Basic Cache Timing Attack Example

- Assume the attacker runs as a user process on a machine (e.g. in cloud computing). Assume the kernel regularly uses some secret to perform an operation e.g. disk encryption. Assume the attacker tries to obtain that secret.
- Init: The attacker fills up the cache by loading a set of unused values `initArray` from RAM. (Example output values of an S-Box in a block cipher. The S-Box is implemented via a simple table lookup.)
- The attacked process P loads a lot of data into the cache. This, however, secret-dependent data from the RAM into the cache. (Only specific outputs of the S-Box that depend on the secret.)
- The attacker tries to fill up the cache again with all the values from `initArray`.
- The attacker measures for each entry that is loaded the loading time.
 - If the time is small, the value is already in cache. It has been loaded by the attacked process. This holds for any process P, even those in the kernel space. (This gives information on which S-Box output is in the cache and thus which secret input to the S-Box was used.)
 - If the time is long, it needs to be loaded from RAM. This value has not been used by the attacked process.

Cache Timing Attacks

- These attacks exploit that the state of the cache can reveal to ProcessX what ProcessY has done although the memory ranges are separated.

Speculative Execution

- The ability to issue instructions past branches that are yet to resolve.
- Essentially tasks are computed that may not be needed.
- Commands are executed speculatively so that the CPU is kept busy.
- In case the speculative execution is actually needed, the CPU commits the speculative computations. This results in a considerable speed up.
- In case the work done by speculative execution is not needed, the CPU discards the speculative computations and the old CPU state should be restored.

Branch Prediction

- Using Branch Prediction, the CPU can predict which branch of a branching/decision point (if statement) is taken in the code and then additionally speculatively execute it (while waiting for some multi-cycle commands to finish).
- Before the branching point, the CPU state is stored.
- **Static Branch Prediction**
 - predictions based on statistical information gathered during the compiler's analysis of the code
- **Dynamic Branch Prediction**
 - relies on the actual runtime behavior of the program
- In case the branch was predicted correctly, the CPU commits the speculative computations. This results in a considerable speed up.
- In case of misprediction, the CPU discards the speculative computations and the old CPU state should be restored.

Exploiting Branch Prediction and Speculative Encryption: Towards Spectre

```
char secret[15];
if (index < secret_size) //index should be smaller than secret_size
{
    v=array1[index];      //if index is too large, this will
                          //overflow and assign secret to
v
    w=array2[v];
}
```

Assumptions
The attacker makes it such that:

- the attacker can manipulate index.
- secret_size is not in the cache, so it needs to be loaded from memory, spanning several cycles.
- the attacker can train the branch prediction to compute the if-block speculatively before secret_size is available.
- array1 is in the cache.
- array2 is not in the cache.
- the attacker can measure the time to read from array2.

Attack:

- w=array2[v] is executed speculatively as a consequence of the training of branch prediction.
- The cache holds w, even after the operations performed by speculative execution is discarded. Speculative encryption only resets the CPU state not the microarchitecture state.
- The attacker can find input index of array2 in cache by loading all array2[i] for all possible characters i and comparing the loading times.
- If the time is small, array2[i] is in the cache and character i must have been index to array2. Otherwise, character i has not been index to array2.