

Software security, secure programming

Information Flow, Non Interference, Sandboxing . . .

Master M2 Cybersecurity

Academic Year 2025 - 2026

Back to password/PIN authentication

```
#define SIZE 4      // public PIN size
#define MAX_TRIES  // maximal tries number

char secretPin[SIZE] = {...} ;      // secret PIN value
unsigned int triesLeft = MAX_TRIES ; // tries counter

boolean checkPIN (char[] inputPin) {
    // No more than triesLeft attempts
    if (triesLeft < 0) return false ; // no authentication
    // Main comparison
    for (short i=0; i < SIZE; i++)
        if (inputPin[i] != secretPin[i]) {
            triesLeft-- ;
            return false ; // no authentication
        }
    // Comparison is successful
    triesLeft = maxTries ;
    return true ; // authentication is successful
}
```

functional property:

$\text{checkPIN}(\text{inputPIN}) \Leftrightarrow \text{inputPin}[0..\text{SIZE} - 1] = \text{secretPin}[0..\text{SIZE} - 1]$

What about **confidentiality** of the secret PIN ?

- ▶ should be protected against **reverse-engineering**
- ▶ should be protected against **(side-channel) information leakage**

Information levels

Several **information levels** “coexist” inside an execution platform:

- ▶ from different users (including root/admin/...)
- ▶ from different processes/threads/applets (e.g., web browser)
- ▶ from different input sources (trusted/untrusted, confidential/public)
- ▶ etc.

⇒ a **lattice**, with **lower** and **higher** information level values

↔ Avoid **unexpected** interferences between **cross level** information flows ?

Security properties to preserve/enforce

confidentiality:

- ↪ no information leakage from higher to lower data
“no write down”, “no read up”

integrity:

- ↪ no information rewriting from lower to higher data
“no write up”, “no read down”

Examples:

- ▶ sensitive shared platform level data (e.g., caches, etc.)
- ▶ sensitive OS level data (e.g., passwords, resource management, etc.)
- ▶ external data, owned by other users/threads
- ▶ sensitive internal application data (e.g., crypto keys, nonces, etc.)
- ▶ sensitive program execution level memory locations (e.g., canaries, return addresses, etc.)

Attacker model

- ▶ knows the **code** (executable → assembly, source ?)
- ▶ **observe** outputs + low variables+ part of the execution platform ...
- ▶ **controls** inputs + low variables
- ▶ may observe other **side-channels**

⇒ may **direct** program execution through controlled inputs

- ▶ to produce/increase **leakage** of higher values
- ▶ to break integrity (of higher data, of code execution, etc).

Rk: could even elaborate interactive/adaptive multi-steps attack strategies !

How information may flow ?

- ▶ Inside a single-threaded application, use/def variable dependencies
 - ▶ **data-flow** (direct/explicit) through assignments
 - ▶ **control-flow** (indirect/implicit) through if, while, ... statements, exceptions, etc.
- ▶ Through side channels
 - ▶ execution time, termination
 - ▶ power consumption
 - ▶ micro-architecture level (shared) resources: caches, instruction pipelines, branch prediction, etc.
 - ▶ others ?
- ▶ Between concurrent/remote processes/threads
 - ▶ sockets, remote calls
 - ▶ shared resources (and race conditions !)

Protection against (unwanted) information flows

- ▶ Hardware mechanisms (enclaves, etc.)
- ▶ OS primitives and access control mechanisms, Virtual Machines
- ▶ Language level facilities and libraries (crypto, etc.)
- ▶ Coding rules (input sanitization, constant-time programming)
- ▶ Compiler options (to enforce protection at the executable code level)
- ▶ some tools ...
 - ▶ static analysis: type systems $\rightsquigarrow$ fix-point computations
 - not decidable, (over-)approximation, not complete
 - ▶ runtime instrumentation/monitoring techniques (taint tags, extra checks)
 - not sound (may miss existing flows)

But:

protection mechanisms always rely on a **TCB (Trusted Computing Base)**

Non Interference: a general definition (1/4)

↪ check information flow **partitions** inside a program

- ▶ more precisely:
no influence of variable/statement of one class to another
influence = read and/or write and/or execute
- ▶ numerous applications in **security**:
 - ▶ confidentiality/integrity
 - ▶ taint analysis (e.g., user-controlled vulnerability exploitability)
 - ▶ side-channels through shared resources (execution time, cache, ...)
 - ▶ no use of uninitialized variables (undefined behavior)
 - ▶ etc.

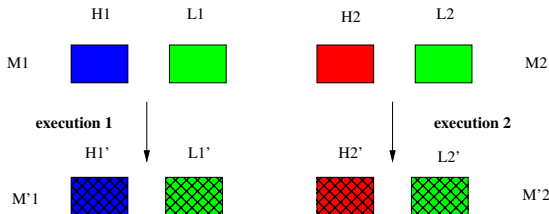
Non Interference: confidentiality (2/4)

No influence **from** data/statement of class H **to** data/statement of class L

Given:

- ▶ a variable partition in 2 classes H and L
- ▶ memory states $M1=(L1, H1)$ and $M2=(L2, H2)$ s.t. $L1 \equiv L2$ and $H1 \neq H2$

Then, any executions from $M1$ and $M2$ lead to memory states $M'1=(L'1, H'1)$ and $M'2=(L'2, H'2)$ s.t. $L'1 \equiv L'2$



Rk:

- ▶ do not take **termination** into account (see later)
- ▶ **hyper property**
(models are **sets** of execution sequences, not single ones ...)

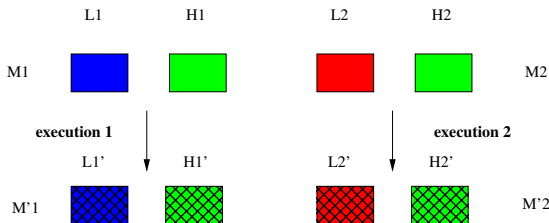
Non Interference: integrity (3/4)

No influence **from** data/statement of class L **to** data/statement of class H

Given:

- ▶ a variable partition in 2 classes H and L
- ▶ memory states $M1=(L1, H1)$ and $M2=(L2, H2)$ s.t. $H1 \equiv H2$ and $L1 \neq L2$

Then, any executions from M1 and M2 lead to
memory states $M'1=(L'1, H'1)$ and $M'2=(L'2, H'2)$ s.t. $H'1 \equiv H'2$



Rk:

- ▶ do not take **termination** into account (see later)
- ▶ **hyper property**
(models are **sets** of execution sequences, not single ones ...)

Non Interference: taint (4/4)

A variable/statement is **tainted** at a program location if its value/execution is influenced by a **user input**

- ▶ **taint source** = “user input channel” (keyboard, network, filesystem, etc.)
- ▶ **taint sink** = (unwanted) **user-controlled vulnerable** variable/statement

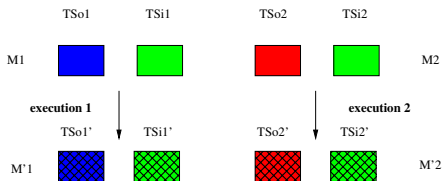
↪ no influence **from** taint source (L) **to** taint sink (H) $\equiv$ **integrity property**

Given:

- ▶ some variables/statements labelled as TSo (taint source) or TSi (taint sink)
- ▶ $M1=(TSo1, TSi1)$ and $M2=(TSo2, TSi2)$ s.t. $TSi1 \equiv TSi2$ and $TSo1 \neq TSo2$

Then, any executions from M1 and M2 lead to

$M'1=(TSo'1, TSi'1)$ and $M'2=(TSo'2, TSi'2)$ s.t. $TSi'1 \equiv TSi'2$



Rk:

- ▶ not need to take **non-termination** into account . . .
- ▶ **hyper property**
(models are **sets** of execution sequences, not single ones . . .)

In the following . . .

1. Information flow within single-threaded applications (see E. Poll' slides)
2. Side channels (next slides below)
3. Sandboxing and access control (see E. Poll' slides)

Information leakage through side channels

Are these programs secure?

In both cases,

- ▶ some additional information about the secret is leaked by the time or the instruction cache
- ▶ by interacting iteratively with the application, the adversary is able to improve his knowledge

```
void compare(int l, int s){  
    if (s<l)  
        {write_log('too large');}    // 1 sec.  
    else  
        {some_computation();}        // 2 sec.  
}
```

- ▶ Attacker can binary search on s using l and the leaked output

```
int pwdCheck(char *l, char* pwd){  
    unsigned i;  
    for (i=0; i<B_Size; i++)  
        if (l[i]!=pwd[i])  
            {return 0;}  
    return 1;  
}
```

- ▶ Segment Oracle Attack : Attacker can brute-force individual characters

Side channels

Information leakage through:

- ▶ (implicit) shared resources: caches, hidden registers, etc.
- ▶ physical observations: time, power consumption, etc.

A same cause: the use of **high** variables to control

- ▶ (global) memory accesses (e.g, arrays) $\rightsquigarrow$ data cache attacks
- ▶ execution control flow $\rightsquigarrow$ instruction cache or branch prediction attacks
- ▶ time-dependent (assembly level) instructions

constant-time¹ programming:

a set of coding rules to protect against such attacks ...

See for instance:

- ▶ A beginner's guide to constant-time cryptography
- ▶ (Intel) Guidelines for Mitigating Timing Side Channels Against Cryptographic Implementations
- ▶ Some Cryptocoding rules

¹not in the sense of **time complexity**!

Some research directions regarding side channels

- ▶ **Quantifying** the information leakage (Quantitative Information Flow)
*is **always** leaking one single (same!) bit of a crypto key less critical than leaking **only once** the whole key?*
- ▶ **Quantifying** the “control level” of an attacker
how much can she/he influence the execution, at which cost ?
- ▶ Distinguish **regular** vs **unwanted** outputs when computing the leakage
e.g., password checking may (at least!) return a boolean value
- ▶ Improve **automatic detection** of side channel information flows
(hyper-property checking)
- ▶ Automatic **code transformation** to constant time mode, dedicated programming languages, etc.