Università di Pisa

**University of Pisa**

*Department of Information Engineering*

# Language-Based Technology for Security

*Nicolò Mariano Fragale*

**March 2025**

# Contents

## Information

These notes are intended for educational purposes only and cover essential concepts in the field of data systems and security. The aim is to provide a comprehensive understanding of topics such as system vulnerabilities, protection techniques, and defense strategies in cybersecurity.

This document includes topics related to access control, authentication mechanisms, database security, cryptographic methods, and advanced persistent threats, with a particular focus on practical applications in real-world scenarios.

# 1 Web Assembly

**1.1 Introduction** WASM is not a programming language, but a binary format generated from other language like C, C++ or Rust. WASM permit this hig level language to run efficiently and properly. It is executed in safe place like browser or other runtime enviroment.

**It is safe because it runs in isolated sandbox.**

It is used to increase performance in web application as:

1. 3D games in Browser;
2. Figma etc…
3. Editing image/video software online…
4. Ai, ML, blockchain, criptoghraphy…
5. Allow to execute C, C++, Rust online;
6. It can be used on server.

## 1.2 Key characteristics

1. Stack-Based (push and pop) `<->` Does not use registers; `->` Operations;
2. Executabel in web broser `->` using WebAssembly JavaScript API. `->` API is the only way to communicate from sandbox to outside;
3. Secure `->` Sandbox and permission denied to access system resources;
4. Platform-independent `->` runs on any device that has WASM runtime.

Listing 1: Code Example

```
(func $calcola (param $x i32) (result i32)
    local.get $x
    local.get $x
    i32.mul
    i32.const 2
    i32.mul
    i32.const 1
    i32.add
)
```

Analyze the example:

1. *func $calcola (param $x i32) (result i32)* :
   (a) *func* it is the key word declaring the function;
   (b) *$calcola* function's name;
   (c) *param* it is the key word declaring the parameter;

(d) *$x* parameter's name;

(e) *i32* indicates the data-type (32 bit integer);

(f) *result i32* indicates the result will be a i32 data type.

(g) **if $ is omitted the code will still work.**

2. *local.get $x* push X in stack with index 0 (Func Starts wih stack empty);
3. *local.get $x* push X in stack with index 1;
4. *i32.mul* pop 0 and 1 mul, then mul them (both x) and push temporary result in index 0;
5. *i32.const 2* push in stack the value 2 as type i32 and index 1;
6. *i32.mul* pop 0 and 1, then mul them and push as temporary result as index 0;
7. *i32.const 1* add 1 as i32 in index 1 ;
8. *i32.add* pop 0 and 1, add index 0 and 1, result is pushed in index 0.

### 1.3 Data-Type

1. **i32** integer with or without sign in 32 bit -> (from 0 to 4.294.967.295) or (from -2.147.483.648 to 2.147.483.647);
2. **i64** integer with or without sign in 64 bit;
3. **f32** floating poin in 32 bit;
4. **f64** floating point in 64 bit.

### 1.4 Storing Values

1. Stack -> push and pop (for operations) of the parameter and costant;
2. Function context -> variabale passed as parameter or declared inside the function -> Example: *local $temp i32*; ->

Listing 2: Code Example

```
        (func $quadrato (param $x i32) (result i32)
        (local $temp i32)
        local.get $x
        local.get $x
        i32.mul
        local.set $temp
        local.get $temp
    )
```

3. Single global memory -> Linear memory to handle complex data structure -> Used by many functions to store data in long term or to share data beetween more functions -> Example:

Listing 3: Code Example

```
        (global $contatore (mut i32) (i32.const 0))

        (func $incrementa (result i32)
            global.get $contatore
            i32.const 1
            i32.add
            global.set $contatore
            global.get $contatore
        )
```

**(mut i32)** mutable variabale type i32 (otherwise immutable during the execution), **(i32.const 0)** initialized at 0.

## 1.5 Operations

1. *local.get $x:* Push x onto the stack;
2. *local.set $x:* Assign the value in top stack to x;

### 1.5.1 Memory

There is a single linear memory built as a contigous array of byte where u can read and write data. The Address Memory is a number -> offset.

### 1.5.2 (data-type(i32,i64...)).load

Using ().load, u can interact with the memory -> U read 4 byte (if i32) in the memory starting from the offset -> Convert in number (i32 in this case) and push onto the stack. It is used when u want read data from the memory.

Example:

Listing 4: Code Example

```
        (memory (export "mem") 1)
        (func $leggi_memoria (param $ptr i32) (result i32)
        local.get $ptr
        i32.load
        )
```

The function will read 4 byte from the value stored in ptr (this value represent the offset). If ptr = 100 (offset), then from 100 the function will read 4 bytes (100-101-102-103).

**(memory (export ”mem”) 1):** Before use load and store u must declare the memory from whom u are going to read data with the commands **memory**.

**N.B:** 1 : 64 Kb (1 page is 64 kb -> 2 pages are 128 kb and so on.)

**N.B:** If u try to ask for a offset+value ¿ 56 536 byte -> *RuntimeError: invalid memory access out of bounds* -> WASM permit dynamic memory allocation.

### 1.5.3 Store

Store allow to write in memory.

Listing 5: Code Example

```
        (memory (export "mem") 1)
        (func $leggi_memoria (param $ptr i32) (result i32)
        local.get $ptr
        local.get $val
        i32.store
        )
```

Work the same way as before, while *local.get $val* specify the value to write in memory.

## 1.6 Control Flow

### 1.6.1 Program Counter

It is a register that keep track about the next istructions to execute, increment each time by 1. It acts like a pointer.

### 1.6.2 Loop/br -> Break/br_if

**Loop** create a label (code of block) which runs infinitely if **br** is at the end of the label.

**N.B:** You cant use **br** outside the scope of the label.

Listing 6: Code Example

```
        (func $loop_example (param $x i32)
        i32.const 0
        local.set $x
        (loop $loop
            local.get $x
            i32.const 1
            i32.add
            local.set $x
            br $loop
        )
      )
```

To block the infinite loop we need a condition -> br_if and a label to call -> Break:

Listing 7: Code Example

```
    (func $loop_example (param $x i32)
        i32.const 0
        local.set $x
        block $out(
```

```
        (loop $loop
            local.get $x
            i32.const 1
            i32.add
            local.set $x

            local.get $x
            i32.const 10
            i32.eq
            br_if $out

            br $loop
        )
    )
)
```

U can indent block, the outest gets label 0, the second enclosing block gets 1 and so on...

Listing 8: Code Example

```
(block $outer_block ;; label 0

    (block $inner_block ;; label 1
    ))
```

**N.B:** We can call br 0 in the inner_block and in this case it will jump directly outside label 0.

**N.B:** U can omit the name of the label. -> br ¡label_name¿ is replace with br 0.

**N.B:** U can omit the name of the variable. Instead use indeces for references (not reccomended).

### 1.6.3 Call & Call_indirect

1. **Call** a function:

Listing 9: Code Example

```
(func $add (param $a i32) (param $b i32) (result i32)
local.get $a
local.get $b
i32.add
)

(func $main
i32.const 5
i32.const 10
call $add ;; Chiamata della funzione $add
)
```

2. **Call_indirect** a function using its index stored in a table containing functions: *(call_indirect (type ¡type¿) ¡index¿)*

### 1.6.4 If & Else

Listing 10: Code Example

```
    (func $check_even (param $x i32) (result i32)
  local.get $x
  i32.const 2
  i32.rem_u ;; rem_u : % (module)
  i32.eqz
  if
    i32.const 1  ;; If $x is even, return 1
  else
    i32.const 0  ;; else 0
  end
)
```

### 1.6.5 Return

A function's return value is **implicitly** the value at the top of the stack. -> U dont need to write it explixitely at the end of the function.

# 2 Security Policies

Security policies defines the rules and constraints about how and when the programs can access to data. Examples of SP are: CIA.

SP can be applied with dynamic techniques or static techniques.

1. **Dynamic enforcment:** SP can change due to events, threats or unexpected changes.
   (a) **Runtime monitoring:** SP monitors each execution of the program checkinf if they respect SP;
   (b) **Enforcment mechanism in VM:** VM use restricted execution enviroment to enforce security
   (c) **Reference monitor:** Intercepts security-sensitive operations ensuring they comply with SP.
2. **Static enforcment:** SP is defined since the beginning and applied with no possibile future changes.
   (a) **Type System & Type safety:** Variables and functions insterted correspond to expected ones;
   (b) **Static Analysis:** Analyze control and data flow before the execution to detect violation;
   (c) **Formal Verification:** Matematical verification applied on states to ensure SP.

**2.1 Execution Monitor** EM is the third part that permit the program to access system resources if SP is True.

EM monitor untrested program, if an execution dont respect SP `->` Violation `->` Alert!

EM are run-time modules that runs in parallel with application.

EM is inside the OS or embedded in Program (if/else and functions wrote in program itself to respect SP);

**Real EM:**

1. Sees most event in a program, not all.
2. Prevent disruptive action in case of violation.
3. Limits the damage in case of violation.

**EM OS:**

1. Ensures the program comply with OS SP (AC rules and roles);
2. Continiusly monitors system;

3. Restrict acces to programs dont comply with some SP to prevent anauthorized resource access;
4. Identify and block malicious activities.
5. Memory safety;
6. Type safety;

**N.B:** Program is a set of executions$s$ -> Execution is a set of State/Event$e$ . -> SP is a predicate, similar a function applied to most execution (call it $P$):

$$forallsP(s)$$

(

$$\forall s$$

can omitted).

**N.B:** The empty sequence

$$\epsilon$$

is an execution.

**N.B:** Security Policy P is a *property* of the program.

**N.B:** A program is secure if all the executions are True, so comply with SP.

**EM enforceable policies:**

1. $$forallsP(s)$$
   and P is called *detector*;
2. P
   $$\epsilon$$
   holds (means is always True) -> All the executions before holds too;
3. If the detector rejects an execution (is not True -> False), the detector declare the rejection in finite time.

If 1,2,3 are complied -> Safety Policy

**EM in Programm:**

The idea is to implement the logic of the SP in the code, handling the flow e the compliance in the programm itself (i.e use condition statement).

(Professor anaylize Automata e FSM in slide LBT25-11-EM)

## 2.2 Meltdown Virus

# 3 Dynamic Taint Analysis

Dynamic taint analysis is a technique used to track the flow of sensitive data in a program. The idea is to mark the data as *tainted* when it comes from an untrusted source, and propagate this information through the program ensuring it does not interact with sensitive data. This way, it is possible to monitor the flow of sensitive data and check if it is handled correctly.

In fact by *Confidentiality Policy* informations can flow only from less to more secure level. ( public `->` private `->` secret `->` top secret).

And by *Integrity Policy* informations can flow only from more to less secure level. ( top secret `->` secret `->` private `->` public).

**DTA** is used in programming language and **especially at runtime** to prevent several type of attacks as: injection, buffer overflow and data leakage and how the informations flow accross the memory, registers, variables and network. If a tainted data interact with a sensitive data, the the result is tainted too.

**DTA** works in several granularities:

1. **Byte-level:** Taint is propagated at byte level;
2. **Bit-level:** Taint is propagated at bit level;
3. **Function-level:** Taint is propagated at function level;
4. So many others . . .

Tainted data can occur in several ways:

- **User Input:** Data provided by the user through forms, command-line arguments, or other input methods.
- **File Input:** Data read from files, especially those from untrusted sources.
- **Network Input:** Data received over the network, such as HTTP requests or socket communication.
- **Environment Variables:** Data retrieved from the system's environment variables.
- **Inter-process Communication:** Data exchanged between processes, which may come from untrusted sources.

**Notice:** To prevent tainted data infect sensitive data, we can solve sanitazing the input data.

### 3.1 Operational Semantics

**Operational Semantics** are used to define how the taint information is propagated through the program.

**RuntTime Structures**

- $\Sigma$: the ordered sequence of program statements $\quad \Sigma = \mathbb{N} \rightarrow \text{Stmt}$

- $\mu$: memory $\quad \mu : \text{Loc} \rightarrow \text{Values}$
- $\rho$: environment $\quad \rho : \text{Var} \rightarrow \text{Loc} + \text{Values}$
- **pc**: program counter
- $\iota$ : next instruction

$$\mu, \rho \Vdash e \Downarrow v$$

**Intuition:** evaluating the expression $e$ in the run-time context provided by the memory $\mu$ and the environment $\rho$ produces $v$ as result.