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

```
1         (func $quadrato (param $x i32) (result i32)
2         (local $temp i32)
3         local.get $x
4         local.get $x
5         i32.mul
6         local.set $temp
7         local.get $temp
8     )
```

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

```
1         (memory (export "mem") 1)
2         (func $leggi_memoria (param $ptr i32) (result i32)
3         local.get $ptr
4         local.get $val
5         i32.store
6         )
```

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

```
1         (func $loop_example (param $x i32)
2         i32.const 0
3         local.set $x
4         (loop $loop
5             local.get $x
6             i32.const 1
7             i32.add
8             local.set $x
9             br $loop
10        )
11      )
```

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

Listing 7: Code Example

```
1     (func $loop_example (param $x i32)
2         i32.const 0
3         local.set $x
4         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. Continuisly 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 -> Execution is a set of State/Event . -> 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 Analysis

Dynamic analysis consists in analyzing the behavior of a program during its execution in several type of enviroment as:

- Sandbox (for WASM);
- Abstract machine (execution in a virtual machine which emulate the behaviorof a real machine);
- Instrumented runtime enviroment(integreted reference monitor);

Which are the limitations of dynamic analysis?

- The Analyzer can not analyze all the possibile paths of the program (i.e a buffer overflow that can occur when an error is triggered might never be detected);
- It depends from the input given to the program;
- By the moment it runs during the execution, it slows down the program;
- Adavance malware can detect dynamic analysis and change the behavior (of the malware);
- Leads to false negative.

A program is 100% safe if all the paths are not unsafe, unfortunately its impossible to analyze all the paths of a program.

3.1 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.

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

RuntTime Structures

- Σ: the ordered sequence of program statements $\Sigma = \mathbb{N} \to \text{Stmt}$
- μ: memory $\mu : \text{Loc} \to \text{Values}$
- ρ: environment $\rho : \text{Var} \to \text{Loc} + \text{Values}$
- **pc**: program counter
- ι : next instruction

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

Intuition: evaluating the expression e in the run-time context provided by the memory μ and the environment ρ produces v as result.

4 Static Analysis

Static analysis aim to understand the behavior and safety of the program before its execution.

Its goals are to avoid memory Leakage, corruptions, injections attack and eventually eliminate dead code.

4.1 Rice's theorem Il teorema di Rice dice che:

Ogni proprietà non banale del linguaggio riconosciuto da una macchina di Turing è indecidibile.

Per proprietà non banale si intende una proprietà che non è vera per tutti i linguaggi né falsa per tutti i linguaggi. Se una proprietà è banale vuol dire che è vera o falsa per tutte le macchine di Touring.

(i.e Dato un programma, riconosce almeno una stringa? → Indecidibile)

Il Teorema di Rice riguarda solo proprietà del linguaggio riconosciuto, quindi non si applica a proprietà sintattiche o strutturali di un programma. (i.e Il programma ha meno di 100 istruzioni? → Decidibile)

1. Halting Problem Afferma che non esiste un algoritmo generale in grado di determinare se un programma (o macchina di Turing) si arresterà o entrerà in un ciclo infinito per un dato input.
2. Teorema di Rice afferma che non esiste un algoritmo generale in grado di verificare se il linguaggio riconosciuto da una macchina di Turing soddisfa una determinata proprietà.
3. La dimostrazione del Teorema di Rice si basa su una riduzione dal problema dell'arresto, mostrando che se potessimo decidere una proprietà non banale del linguaggio, potremmo anche decidere se un programma si arresta (che sappiamo essere impossibile).

4.2 Abstract Interpretation By the moment is undecidable analyze all the program itself, we create an abstraction of it which is decidable avoiding the rice's theorem.

Unfortunately during the analyses we can occur in false positive and false negative:

- **False positive:** is reported an error that is not present; (bad)
- **False negative:** is not reported an error that is present; (really bad)

4.3 Static sign analysis

Concrete Domain (ints)	Abstract Domain (signs)
$v = 1000$	$+$
$v = 1$	$+$
$v = -1$	$-$
$v = 0$	0
$v = e\ ?\ 1 : -1$	$unknown$
$v = w/0$	$undefined$

Lattice: A lattice is a matematical structure (partial order) aiming to organize data in a hierarchical way .

Sign lattice: used in static sign analysis, it is a lattice that organize the data in levels: positive, negative and zero, unkown(top) and undefined(bottom).

Transfer functions: define how to evaluate different statement on abstracat values (dati i valori astratti → valutare le espressioni).

$$\begin{array}{ccccc} \boxed{+} & + & \boxed{+} & = & \boxed{+} \\ \boxed{-} & + & \boxed{-} & = & \boxed{-} \\ \boxed{0} & + & \boxed{0} & = & \boxed{0} \\ \boxed{+} & + & \boxed{-} & = & \boxed{T} \end{array}$$

$$\begin{array}{ccccc} \boxed{+} & / & \boxed{+} & = & \boxed{+} \\ \boxed{-} & / & \boxed{-} & = & \boxed{+} \\ \boxed{T} & / & \boxed{0} & = & \boxed{\bot} \\ \boxed{+} & / & \boxed{-} & = & \boxed{-} \end{array}$$

N.B: Arr[unknown] = undefined

N.B: Over approximation causes false positives.

Gli stati del programma sono gli elementi dell abstract domain. (chiave: concrete domain → valore: abstract domain)

4.4 Partial orders Partial order is a set S where are satisfied the following properties:

- Reflexivity: $\forall x \in S, x \leq x$
- Antisymmetry: $\forall x, y \in S, x \leq y \wedge y \leq x \rightarrow x = y$
- Transitivity: $\forall x, y, z \in S, x \leq y \wedge y \leq z \rightarrow x \leq z$

5 Control Flow

Control flow show how the program evolves, so its execution.
We represent all the possible paths of an execution with a directed graph → **CFG** (control flow graph).
The difference beetween CFG and AST is that AST represent the structure and syntax of the program, while CFG represent the runtime behavior of the program..
CFG is composed by:

1. **Node/Basic Block**: a sequence of statements, with one entry point at the beginning od the block and one exit point at the end of the block. Not contain either label (just the first line) or jumps (just the last one to another block).
2. **Edge**: represent the flow of the program, hence the change beetween blocks.

Among basic blocks, we can have a **leader** which is the first line of code identifying the block.

- **Interprocedural analysis:** analyze the whole body of the program → ∀ functions.
- **Intraprocedural analysis:** analyze a single function.

∀ node → ∃ [v] (constraint variable).
∀ [v] → ∃ previous data flow.

Fixed point theorem: Under several conditions, a function will have at least one fixed point: a point where the function will not change anymore. → *f(x) = x.*

5.1 Worklist algorithm: Used to solve problem of data-flow analysis. Propagate result from each node till obtain a fixed point result.
What is a worklist? Is a data structure like stack or queue that track the nodes to process.
How it works?

1. Gather all the nodes → Define starter value: ∅
2. Extract the node → Modify the node if required → Add to the worklist / otherwise not.
3. Reiterate till the worklist = ∅ ↔ fixed point.

5.2 Data Flow Analysis of CFG with Set Theory Compute analysis state at each program point/node → For each statement, define how it affects the analysis → Iterate until fix-point reached

N.B: An expression (non-trivial) is available at certain point ← value computed earlier in the execution.

- Statement : istruzione → assegnazione (x = 5), controllo (x ≥ 5)
- Expression : espressione → aritmetica (a + b), booleana (x ≥ 5)

- Variabili + istruzioni = Espressione
- **GEN():** Available expressions generated by a statement.
  1. The GEN set for a statement (or basic block) contains the expressions that are computed (evaluated) by the statement and **whose values are not subsequently invalidated within the same block**.
  2. (i.e) GEN(x = a * b) = a*b (a*b is set of Set* (Set* = a, b, a op b, $\emptyset$ (all possible set)))
  3. (i.e) GEN(a = a+1) = $\emptyset$ (not available because must be computed earlier)
- **KILL():** Available expressions killed by a statement. return a set whom variable are updated in the current block.

N.B: Kill and Gen are exlusive, or one or the other.

- **IN:** All the expressions incoming the block (initially all $\emptyset$/(bottom))
- **OUT:** All the expressions outgoing from the block.

$$OUT(s) = (IN(s) \setminus KILL(s, exp)) \cup GEN(s)$$

  This formula is called **Flow Equation**
- Check the slide with example

5.3 Data Flow Analysis of CFG with monotone framework :

$\forall$ node $\rightarrow \exists$ [v] (constraint variable).
[v] yields the set of expressions that are avaialable at the program point **after**. (reverse then set theory which yields the **before**)

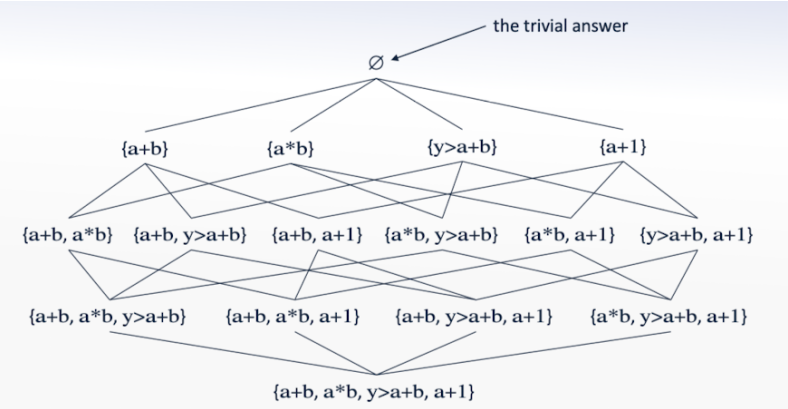


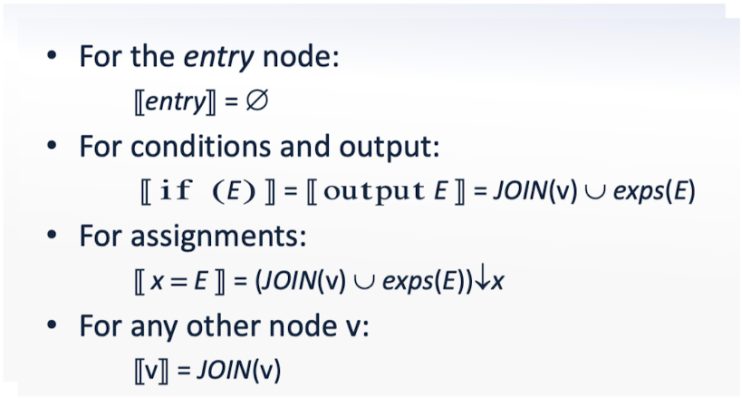
Figure 1: Reverse powerset

N.B: [v] yields the set of expressions available **after** v. After because the powerset is reversed then set theory.

$$JOIN(v) = \bigcap_{w \in PRED(v)} [w]$$

S ↓ x = KILL = S - E ∈ Exp — x ∈ Var(E) remove all the expressions that contain the variable x from the set (Same as KILL in set theory). where S is a subset of set of expressions, E è un espressione e x è una variabile dell espressione E.

- exps(Costante) = ∅
- exps(Variabile) = ∅
- exps(espressione1 operazione espressione2) = espressione1 operazione espressione2 ∪ exps(espressione1) ∪ exps(espressione2)



Controlla slide per esempio finale

6 Bodei lessons

6.1 Liveness analysis set theory

- If variable x is live in a basic block then it is a potential candidate for register allocation.
- Two variables can share the same register if they are not live at the same time.
- Dead code elimination occur when as assignment is redundant.

A variable x is:

- **Dead:** when an istruction assing x.
- **Live:** when an istruction uses it.
- y = x / 2 → y is dead and x is live.

The information propagates **backward** and the information coming from other nodes can be merged with **union**.

Out(v) = $\bigcup_{w \in SUCC(v)} IN(w)$
In(v) = Gen(v) ∪ (Out(v) \ KILL(v))

N.B: Initially there are no live variables → ∅.
Backward analsysis progate live variables in the opposite direction of Control flow.

6.2 Sign analysis monotone framework

- For each CFG node v we assign a constraint variable [[v]] denoting an abstract state that gives the sign values for all variables at the program point immediately after v
- $$\text{JOIN}(v) = \bigcup_{w \in \text{pred}(v)} [\![w]\!]$$

6.3 Dead Store elimination and security Dead store elimination (DSE)

is a possible compiler optimization, built on *liveness analysis*.
DSE is run multiple times.
A (persistent) (state security violation) is triggered when when data remains in memory even if no more necessary.

X becomes skip because is not live (never used).
By the moment there are issues in DSE optimizations, we implement *translation validation.*
Translation validation, given an istance, check its security.

Validating correctness is in PTIME.

1. P input program;
2. Q resulting program;
3. D list of removed stores;
4. Hence, compute Dead store in P and check Dp $\subseteq$ D;
5. Procedures completed in PTIME.

correctness can be defined by considering individual executions

Validating security is UNDECIDABLE.

1. P input program;
2. list of dead assignments;
3. Removes these from P $\rightarrow$ Q.

It does not eliminate information leaks from P , it only ensures that no new leaks are introduced in the transformation from P to Q
State Variables are LOW Security.
Input Variables could be LOW or HIGH Security.

Program State A program state is a pair:

- m : CFG node
- p : is a mapping function, map a variable to a value, could be TAINT or UNTAINT

In the initial state, located at the entry node, state variables have a fixed valuation.

Data Flow Information A set of node N in CFG:

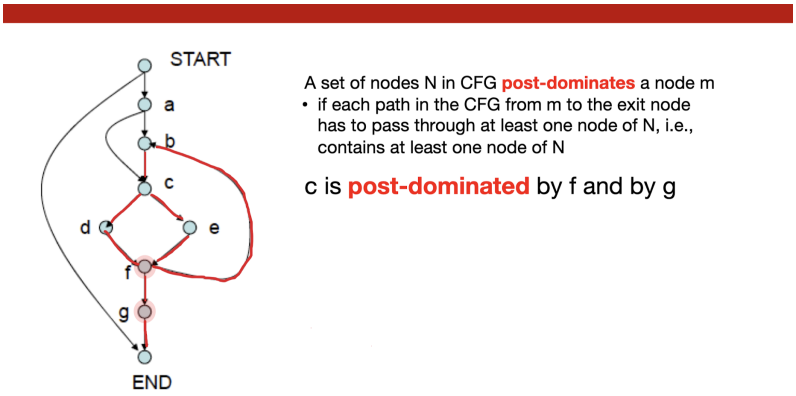
1. **dominates** a node m: if each path in the CFG from the entry node to m has to pass through at least one node of N. It is a **forward analysis**

$$[[v]] = \{v\} \cup \bigcap_{w \in \text{pred}(v)} [[w]] \tag{1}$$

2. **post-dominates** a node m: if each path in the CFG from m to the exit node has to pass through at least one node of N, i.e., contains at least one node of

N. It is a **backward analysis**. (In altre parole: non puoi arrivare alla fine partendo da m senza passare da n.)

$$[[v]] = \{v\} \cup \bigcap_{w \in \text{succ}(v)} [[w]] \tag{2}$$



Dominators and post-dominators tell us which basic block must be executed prior to, of after, a block m.

Information Leakage Program transformations are assumed not to alter the set of input variables, but only state variables.
Una trasformazione di programma prende un programma originale P e lo trasforma in un nuovo programma Q. La trasformazione è detta **corretta** se, per ogni valore di input a, la sequenza degli output prodotti da $P(a)$ e $Q(a)$ è identica. In altre parole, Q deve produrre esattamente gli stessi risultati osservabili di P, indipendentemente da eventuali modifiche interne (ad esempio, ottimizzazioni o ristrutturazioni del codice). Quindi, Q può essere modificato o ottimizzato, ma non deve comportarsi peggio di P in termini di output.
Dal punto di vista della **sicurezza**, si definisce un *leaky triple* come una tripla $(\text{input}, \text{output}, \text{osservazione})$ che rappresenta un possibile canale attraverso cui un attaccante può ottenere informazioni sensibili. Una trasformazione da P a Q è detta **sicura** se l'insieme dei leaky triple di Q è un sottoinsieme di quello di P, ovvero Q non introduce nuovi canali di leakage rispetto a P. Questo garantisce la **sicurezza relativa**, ovvero che Q non è più insicuro di P, anche se non è perfettamente sicuro. L'importante è che la trasformazione non abbia peggiorato la situazione in termini di leakage.

6.3.1 Hoare Logic

Axiomatic semantics P S Q

- **P** is a pre-condition;
- **S** is a statement;

- **Q** is a post-condition, all the state rechable.
- (i.e) b≥0 a = b+1a≥1
- if S is executed in a state where P is true, and the execution terminates with success, then Q is guaranteed to be true afterwards there is a flase positive.

Nel contesto della verifica formale dei programmi, possiamo distinguere tre concetti fondamentali:

- **Semantica**: data una precondizione P e uno statement c, determinare una postcondizione Q tale che l'asserzione $\{P\}c\{Q\}$ descriva correttamente il comportamento del comando c a partire da uno stato che soddisfa P.
- **Specificazione**: dato una precondizione P e una postcondizione Q, determinare un comando c tale che $\{P\}c\{Q\}$, cioè scrivere un programma c che realizzi quanto richiesto dalla specifica determinata da P e Q.
- **Correttezza**: dato una precondizione P, un comando c e una postcondizione Q, dimostrare che l'asserzione $\{P\}c\{Q\}$ è corretta, ovvero che il programma c soddisfa la specifica definita da P e Q. Questa attività corrisponde alla verifica della correttezza di c rispetto alla specifica.
- i.e ? a = b+1 a¿1 → ? = b ¿ 0 (it is the least restrictive requirement g)

Sequential rule

Formally: {seq} $\dfrac{\{P\}\, c_1\, \{R\} \quad \{R\}\, c_2\, \{Q\}}{\{P\}\, c_1; c_2\, \{Q\}}$

{?} y = 3*x + 1; x = y + 3 {x<10}

- {?} x = y + 3 {x<10}
  {x<10[y+3/x]} x = y + 3 {x<10} ➡ {y<7} x = y + 3 {x<10}
- {?} y = 3*x + 1 {y<7}
  {y< 7[3*x+1/y]} y = 3*x + 1 {y<7} ➡ {x<2} y = 3*x + 1 {y<7}

{x<2} y = 3*x + 1; x = y + 3 {x<10}

{skip} $\dfrac{}{\{P\}\ \text{skip}\ \{P\}}$ {assign} $\dfrac{}{\{Q[a/x]\}x := a\{Q\}}$

{seq} $\dfrac{\{P\}\, c_1\, \{R\} \quad \{R\}\, c_2\, \{Q\}}{\{P\}\, c_1; c_2\, \{Q\}}$ {cons} $\dfrac{P \Rightarrow P' \quad \{P'\}\, c\, \{Q'\} \quad Q' \Rightarrow Q}{\{P\}\, c\, \{Q\}}$

{cond} $\dfrac{\{P \wedge b\}\, c_1\, \{Q\} \quad \{P \wedge \neg b\}\, c_2\, \{Q\}}{\{P\}\ \text{if } b \text{ then } c_1 \text{ else } c_2\ \{Q\}}$

{while} $\dfrac{\{P \wedge b\}\, c\, \{P\}}{\{P\}\ \text{while } b \text{ do } c\ \{P \wedge \neg b\}}$

6.3.2 Taint proof system

A Taint Proof System aims at tracking the influence of input variables on program state.
Taint = tainted [true], untainted [false]

$$\frac{}{\{\mathcal{F}[e/x]\}\ x := e\ \{\mathcal{F}\}}$$

Examples

- {x:U, y:U} x = 0; {x:U, y:U}
  the tag of x **directly** depends on the tag of 0, while the tag of y does not change

- {x:U, y:U} x = read_password(); {x:T, y:U}
  the tag of x **directly** depends on the tag of read_password();

Conditional rule: focus on case B

c tainted

$$\frac{\mathcal{E}(c) = \text{true} \quad \{\mathcal{E}\}\ S_1\ \{\mathcal{F}\} \quad \{\mathcal{E}\}\ S_2\ \{\mathcal{F}\} \quad \forall x \text{ in } \text{Assign}(S_1) \cup \text{Assign}(S_2): \mathcal{F}(x)}{\{\mathcal{E}\}\ \text{if c then } S_1 \text{ else } S_2\ \{\mathcal{F}\}}$$

If the condition is tainted each variable assigned in the condition branches becomes tainted

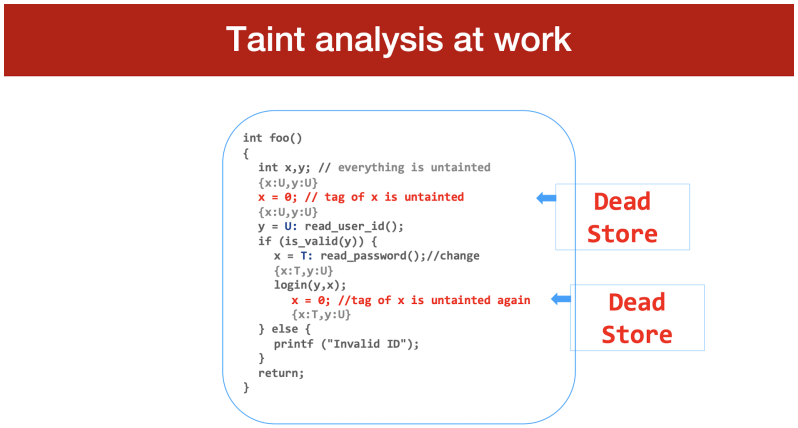
Example

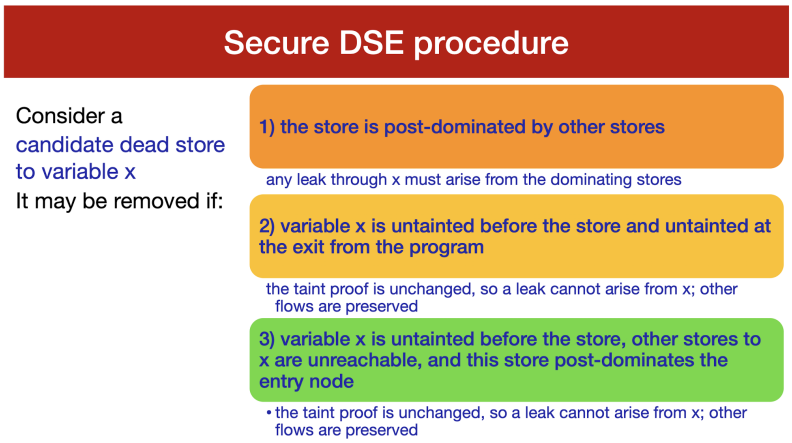
- {c:T, x:U, y:U} if c then x = y else x = z {c:T, x:T, y:U}
  the tag of x indirectly depends on the tag of c

Teorema: for any transformation with a restrict refinement proof, correctness implies security.

6.4 Static taint tracking The flow from state T to state X is ensured only if the type of value of both statuses is the same.
In the lattice: untaint is less defined then taint.

- Sink(untainted string s): is an untainted method (e.g print)
- Source(): is a tainted method (e.g getsFromNetwork)
- Example:
  - string name = source()
  - string x = name;
  - sink(x)
  - **A tainted value reached sink!**





Comparison:

- **May analysis**: we care about possibile behavior (e.g possible vulnerabilities)

  Lattice structure: $\subseteq$
  Combining constraints(JOIN functions): $\cup$

- **Must analysis**: we need guaranted facts (e.g safe optimizations)

  Lattice structure: $\supseteq$
  Combining constraints(JOIN functions): $\cap$

The analysis tracks how taint change over time through the control flow graph (considering the order of statements in the program).

Exists GEN(s) and KILL(s) with the same role of previous paragraph, IN(s), OUT(s) and flow equation too.

- If a variable was tainted earlier (IN(s)), but gets overwritten or sanitized (KILL(s)), it will no longer be tainted in OUT(s).
- If a taint is introduced (GEN(s)), it appears in OUT(s) even if it was not previously present.

6.4.1 Monotone framework

:

Lattice L = untainted $\leq$ tainted

JOIN(i) = $\cup$[j] j $\in$ predecessors(i)

6.4.2 Static single assignment

:

SSA requires that:

- Every variable is assigned exactly once;
- Every variable is defined before it is used.

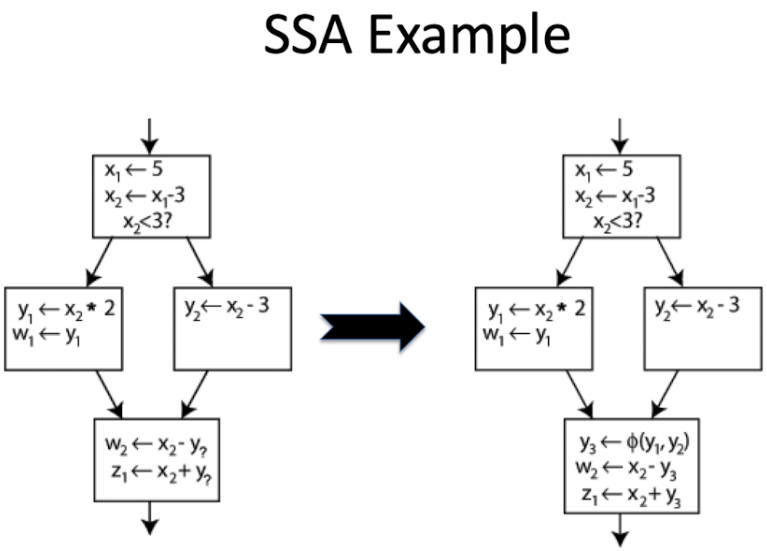


Figure 2:

ϕ: choose either y1 or y2 depending on the control flow.