



University of Pisa

Department of Information Engineering

Language-Based Technology for Security

Nicolò Mariano Fragale
March 2025

Contents

1	Web Assembly	3
1.1	Introduction	3
1.2	Key characteristics	3
1.3	Data-Type	4
1.4	Storing Values	4
1.5	Operations	5
1.5.1	Memory	5
1.5.2	(data-type(i32,i64...)).load	5
1.5.3	Store	6
1.6	Control Flow	6
1.6.1	Program Counter	6
1.6.2	Loop/br -> Break/br_if	6
1.6.3	Call & Call_indirect	7
1.6.4	If & Else	8
1.6.5	Return	8
2	Security Policies	9
2.1	Execution Monitor	9
2.2	Meltdown Virus	10
3	Dynamic Analysis	11
3.1	Dynamic Taint Analysis	12
4	Static Analysis	14
4.1	Rice's theorem	14
4.2	Abstract Interpretation	15
4.3	Set theory and Monotone Framework	15
4.4	May Analysis and Must Analysis	17
4.5	Data Flow Analysis	21
4.5.1	Worklist algorithm:	21
4.5.2	Data Flow Analysis of CFG with Set Theory	22
4.5.3	Data Flow Analysis of CFG with monotone framework	22
4.6	Static Sign Analysis	24
4.7	Liveness Analysis	26
4.8	Dead Store elimination and security	26
4.8.1	Hoare Logic	28
4.8.2	Taint proof system	30
4.9	Static Taint Analysis	33
4.9.1	Monotone framework	33
4.9.2	Static single assignment	33
4.10	Path Sensitive Analysis	34
5	Static Type for information flow	38

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 high level language to run efficiently and properly. It is executed in safe place like browser or other runtime environment.

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, cryptography...
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. Executable in web browser -> 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

```
1  (func $calcola (param $x i32) (result i32)
2      local.get $x
3      local.get $x
4      i32.mul
5      i32.const 2
6      i32.mul
7      i32.const 1
8      i32.add
9  )
```

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 with 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 point 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 constant;
2. Function context -> variable 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 between more functions -> Example:

Listing 3: Code Example

```

1      (global $contatore (mut i32) (i32.const 0))
2
3      (func $incrementa (result i32)
4          global.get $contatore
5          i32.const 1
6          i32.add
7          global.set $contatore
8          global.get $contatore
9      )

```

(**mut i32**) mutable variabile 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 contiguous 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

```

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

```

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 instructions 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(

```

```

5      (loop $loop
6          local.get $x
7          i32.const 1
8          i32.add
9          local.set $x
10
11         local.get $x
12         i32.const 10
13         i32.eq
14         br_if $out
15
16         br $loop
17     )
18 )
19 )

```

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

Listing 8: Code Example

```

1      (block $outer_block ;; label 0
2
3          (block $inner_block ;; label 1
4              ))

```

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

```

1      (func $add (param $a i32) (param $b i32) (result i32)
2          local.get $a
3          local.get $b
4          i32.add
5          )
6
7      (func $main
8          i32.const 5
9          i32.const 10
10         call $add ;; Chiamata della funzione $add
11         )

```


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

1.6.4 If & Else

Listing 10: Code Example

```
1      (func $check_even (param $x i32) (result i32))
2      local.get $x
3      i32.const 2
4      i32.rem_u ;; rem_u : % (module)
5      i32.eqz
6      if
7          i32.const 1 ;; If $x is even, return 1
8      else
9          i32.const 0 ;; else 0
10     end
11 )
```

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 enforcement:** 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) **Enforcement mechanism in VM:** VM use restricted execution environment to enforce security
 - (c) **Reference monitor:** Intercepts security-sensitive operations ensuring they comply with SP.
2. **Static enforcement:** SP is defined since the beginning and applied with no possiblle 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 access to programs that don't comply with some SP to prevent unauthorized resource access;
4. Identify and block malicious activities.
5. Memory safety;
6. Type safety;

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

$$\text{forall } sP(s)$$

(

$$\forall s$$

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.

$$\text{forall } sP(s)$$

and P is called *detector*;

2. P

$$\epsilon$$

holds (means is always True) \rightarrow All the executions before holds too;

3. If the detector rejects an execution (is not True \rightarrow False), the detector declare the rejection in finite time.

If 1,2,3 are complied \rightarrow 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 analyze 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 environment as:

- Sandbox (for WASM);
- Abstract machine (execution in a virtual machine which emulate the behavior of a real machine);
- Instrumented runtime environment (integrated reference monitor);

Which are the limitations of dynamic analysis?

- The Analyzer can not analyze all the possible 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;
- Advanced malware can detect dynamic analysis and change the behavior (of the malware);
- Leads to false negative.
- Behavior depends on the runtime environment.

A program is 100% safe if all the paths are not unsafe, unfortunately it's 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.

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

Runtime Structures

- Σ : the ordered sequence of program statements $\Sigma = \mathbb{N} \rightarrow \text{Stmt}$
- μ : memory $\mu : \text{Loc} \rightarrow \text{Values}$
- ρ : environment $\rho : \text{Var} \rightarrow \text{Loc} + \text{Values}$
- **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 Turing.

(i.e Dato un programma, riconosce almeno una stringa? \rightarrow 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? \rightarrow 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 Set theory and Monotone Framework There are two formal methods to describe:

1. What is analyzed;
2. How the information propagates;
3. How to achieve a fixed point;

Set Theory È l'approccio intuitivo e concreto, usato spesso per esempi didattici. Rappresenta i dati come insiemi finiti.

Usa operazioni di insieme come unione e intersezione.

Ogni nodo ha un $IN[n]$ e $OUT[n]$.

Il lattice nel set theory non viene esplicitato perché il set theory si limita ad effettuare funzioni semplici e banali che hanno un comportamento lineare.

Monotone Framework È un modello teorico più generale e astratto per descrivere intere famiglie di analisi statiche, motivo per cui serve per costruire analizzatori automatici e provare correttezza.

Viene definito da:

1. Lattice;
2. Flow functions, ovvero funzioni monotone $f : D \rightarrow D$ (che descrivono come i dati cambiano nei nodi).
3. Direzione (backward o forward);
4. Metodo per combinare le informazioni;

Il lattice viene definito una volta sola, prima dell'analisi, e non cambia mai durante l'esecuzione.

Nel Monotone Framework, il lattice viene esplicitato perchè lavori con domini astratti più ricchi e funzioni complesse, e il lattice diventa essenziale per garantire il corretto funzionamento dell'analisi.

Quando progetti un'analisi, devi definire:

- Il **dominio** D su cui operi (insiemi, intervalli, segni, booleani, ecc.)
- L'**ordine parziale** tra gli elementi del dominio, denotato con \leq

- L'**operatore di join** \sqcup per combinare le informazioni
- Gli elementi speciali:
 - \perp (informazione minima, ad esempio “nessuna informazione”)
 - \top (informazione massima, ad esempio “massima incertezza”)

Durante l'analisi, non modifichi mai la struttura del lattice, ma i suoi valori.

4.4 May Analysis and Must Analysis

May Analysis Viene applicata quando vogliamo sapere se qualcosa può accadere in almeno un path.

1. Holds in **all** paths,
2. Over approximation (potresti includere anche definizioni che non ci arrivano mai nella pratica),
3. Use \cup as join to combine informations,
4. Lattice order: \subseteq
 - Reaching Definitions, sapere se puoi riutilizzare un valore
 - Live Variables, sapere se una variabile potrebbe essere letta in futuro

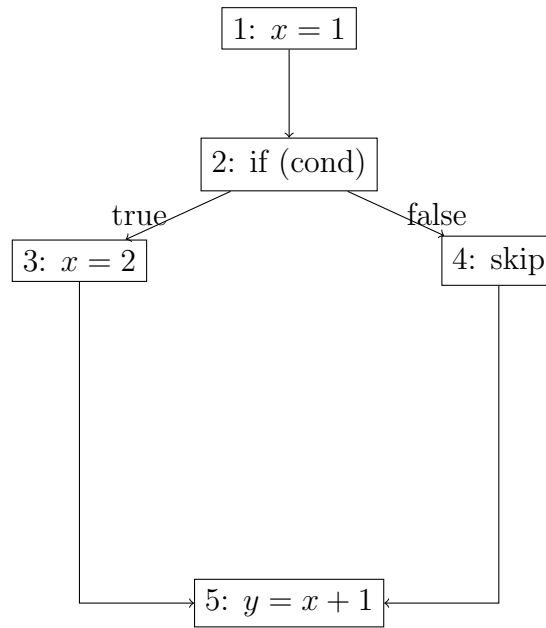
Must Analysis Viene applicata quando vogliamo sapere se qualcosa accade in tutti i path.

1. Holds in **some** paths,
2. Under approximation (devi mantenere solo ciò che è certo),
3. Use \cap as join to combine informations,
4. Lattice order: \supseteq
 - Available expressions, puoi eliminare il ricalcolo solo se è sempre disponibile,
 - Constant propagations, vuoi propagare un valore solo se è sempre costante.

Sia MAY che MUST possono essere:

- **Forward**: l'informazione fluisce dal programma verso il basso, usato in **Sign analysis** e **Available expressions**
- **Backward**: l'informazione risale verso l'alto (tipico per analisi che guardano l'uso futuro), usato in **Live variables analysis**

```
x = 1;
if (cond) {
    x = 2;
}
y = x + 1;
```



Analisi MAY:

- Consideriamo **quali definizioni possono raggiungere** il nodo 5.
- Path 1: $x = 1 \rightarrow \text{cond} \rightarrow x = 2 \rightarrow y = x + 1$
- Path 2: $x = 1 \rightarrow \text{cond} \rightarrow \text{skip} \rightarrow y = x + 1$

Al nodo 5, possiamo aver avuto:

$$\{x = 1, x = 2\}$$

perché almeno in uno dei cammini $x = 1$ o $x = 2$ possono raggiungere il punto.

Analisi MUST:

- Consideriamo **quali definizioni devono raggiungere** il nodo 5 su tutti i cammini.
- Path 1: ultima definizione è $x = 2$.
- Path 2: ultima definizione è ancora $x = 1$.

Non c'è una definizione comune a tutti i cammini.

Al nodo 5, l'insieme è:

$$\emptyset$$

Tipo	Valori a [5]	Motivazione
MAY Analysis	$\{x = 1, x = 2\}$	Presente in almeno un cammino
MUST Analysis	\emptyset	Nessuna definizione comune a tutti i cammini

Altro esempio:

```

1: x = 1;
2: y = 2;

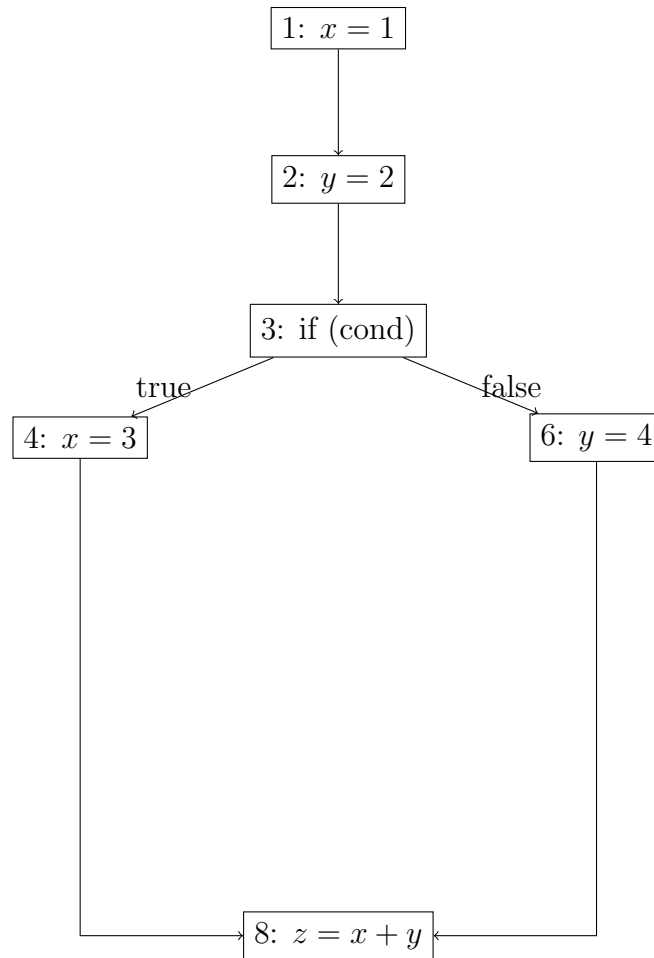
```

```

3: if (cond) {
4:     x = 3;
5: } else {
6:     y = 4;
7: }
8: z = x + y;

```

Control Flow Graph (CFG)



Descrizione dell'Analisi:

Obiettivo: determinare quali variabili possono essere usate in futuro prima di essere sovrascritte.

Tipo: MAY analysis (sovra-approssimazione)

Direzione: Backward (da fine a inizio)

- Nodo [8]: $z = x + y$
 - Usa x e $y \rightarrow \text{Live IN} = \{x, y\}$
- Nodo [4]: $x = 3$
 - Sovrascrive $x \rightarrow \text{Rimuovi } x \text{ dal Live OUT}$

- Live OUT = $\{x, y\} \rightarrow$ Live IN = $\{y\}$
- Nodo [6]: $y = 4$
 - Sovrascrive $y \rightarrow$ Rimuovi y dal Live OUT
 - Live OUT = $\{x, y\} \rightarrow$ Live IN = $\{x\}$
- Nodo [3]: **if** (**cond**)
 - Join dei due rami: $\{y\} \cup \{x\} = \{x, y\}$
 - Live IN = Live OUT = $\{x, y\}$
- Nodo [2]: $y = 2$
 - Sovrascrive $y \rightarrow$ Rimuovi y dal Live OUT
 - Live OUT = $\{x, y\} \rightarrow$ Live IN = $\{x\}$
- Nodo [1]: $x = 1$
 - Sovrascrive $x \rightarrow$ Rimuovi x dal Live OUT
 - Live OUT = $\{x\} \rightarrow$ Live IN = \emptyset

Nodo	Live IN	Live OUT
1	\emptyset	$\{x\}$
2	$\{x\}$	$\{x, y\}$
3	$\{x, y\}$	$\{x, y\}$
4	$\{y\}$	$\{x, y\}$
6	$\{x\}$	$\{x, y\}$
8	$\{x, y\}$	\emptyset

- La Live Variables Analysis è una **MAY analysis**:
 - Se esiste almeno un cammino dove la variabile è viva, la consideriamo viva.
- È una **backward analysis**:
 - I dati fluiscono all'indietro nel CFG.
- Si usa il **join** (unione) nei punti di merge dei cammini.

4.5 Data Flow Analysis Dataflow analysis is a static program analysis technique used to infer properties about program variables (or other program elements) at various points in the control flow.

The reasoning is about data, and dependencies between data.

It systematically tracks how information “flows” through the program based on its structure, without actually executing the code.

We represent all the possible paths of an execution with a directed graph \rightarrow **CFG** (control flow graph).

The difference between 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 of 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 between 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 $\rightarrow \forall$ functions.
- **Intraprocedural analysis**: analyze a single function.

\forall node $\rightarrow \exists [v]$ (constraint variable).

$\forall [v] \rightarrow \exists$ 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. $\rightarrow f(x) = x$.

4.5.1 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 \rightarrow Define starter value: \emptyset
2. Extract the node \rightarrow Modify the node if required \rightarrow Add to the worklist / otherwise not.
3. Reiterate till the worklist $= \emptyset \leftrightarrow$ fixed point.

4.5.2 Data Flow Analysis of CFG with Set Theory

Compute analysis state at each program point/node \rightarrow For each statement, define how it affects the analysis \rightarrow Iterate until fix-point reached

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

- Statement : istruzione \rightarrow assegnazione ($x = 5$), controllo ($x \geq 5$)
- Expression : espressione \rightarrow aritmetica ($a + b$), booleana ($x \geq 5$)
- 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^* ($Set^* = a, b, a \text{ op } b, \emptyset$ (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 (what is made untainted or overwritten by blocks). return a set whom variable are updated in the current block.

N.B: Kill and Gen are exclusive, 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

Flow Equation capture flow sensitivity.

- 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.

4.5.3 Data Flow Analysis of CFG with monotone framework

:

\forall node $\rightarrow \exists [v]$ (constraint variable).

$[v]$ yields the set of expressions that are available at the program point **after**.

(reverse then set theory which yields the **before**)

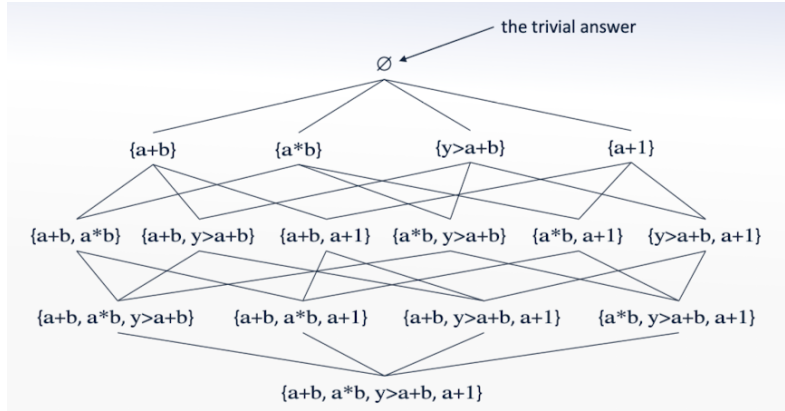


Figure 1: Reverse powerset

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

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

$S \downarrow x = \mathbf{KILL} = S - E \in \mathbf{Exp} \text{ — } x \in \mathbf{Var}(E)$ 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 .

- $\text{exps}(\text{Costante}) = \emptyset$
- $\text{exps}(\text{Variabile}) = \emptyset$
- $\text{exps}(\text{espressione1 operazione espressione2}) = \text{espressione1 operazione espressione2} \cup \text{exps}(\text{espressione1}) \cup \text{exps}(\text{espressione2})$

- For the **entry** node:

$$\llbracket \text{entry} \rrbracket = \emptyset$$

- For conditions and output:

$$\llbracket \text{if } (E) \rrbracket = \llbracket \text{output } E \rrbracket = JOIN(v) \cup \text{exps}(E)$$

- For assignments:

$$\llbracket x = E \rrbracket = (JOIN(v) \cup \text{exps}(E)) \downarrow x$$

- For any other node v :

$$\llbracket v \rrbracket = JOIN(v)$$

Controlla slide per esempio finale

4.6 Static Sign Analysis

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

Lattice: A lattice is a mathematical 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 \rightarrow valutare le espressioni).

$\boxed{+}$	$+$	$\boxed{+}$	$=$	$\boxed{+}$
$\boxed{-}$	$+$	$\boxed{-}$	$=$	$\boxed{-}$
$\boxed{0}$	$+$	$\boxed{0}$	$=$	$\boxed{0}$
$\boxed{+}$	$+$	$\boxed{-}$	$=$	\boxed{T}
$\boxed{+}$	$/$	$\boxed{+}$	$=$	$\boxed{+}$
$\boxed{-}$	$/$	$\boxed{-}$	$=$	$\boxed{+}$
\boxed{T}	$/$	$\boxed{0}$	$=$	$\boxed{\perp}$
$\boxed{+}$	$/$	$\boxed{-}$	$=$	$\boxed{-}$

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 \rightarrow valore: abstract domain)

$$\text{JOIN}(i) = \bigsqcup_{j \in \text{prec}(i)} X_j$$

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: $\forall x, y, z \in S, x \leq y \wedge y \leq z \rightarrow x \leq z$

Static sign analysis with monotone framework

- For each CFG node v we assign a constraint variable $\llbracket v \rrbracket$ denoting an abstract state that gives the sign values for all variables at the program point immediately after v

-

$$\text{JOIN}(v) = \bigcup_{w \in \text{pred}(v)} \llbracket w \rrbracket$$

4.7 Liveness Analysis

- 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 an assignment is redundant \rightarrow Variables reupdates are dead variables.

A variable x is:

- **Dead:** when an instruction assigns x .
- **Live:** when an instruction uses it.
- $y = x / 2 \rightarrow y$ is dead and x is live.

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

$$\text{Out}(v) = \bigcup_{w \in \text{SUCC}(v)} \text{IN}(w)$$

$$\text{In}(v) = \text{Gen}(v) \cup (\text{Out}(v) \setminus \text{KILL}(v))$$

N.B: Initially there are no live variables $\rightarrow \emptyset$.

Backward analysis propagate live variables in the opposite direction of Control flow.

N.B: Live analysis use MAY analysis

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

$$\text{JOIN}(v) = \bigcup_{w \in \text{SUCC}(v)} [[w]]$$

4.8 Dead Store elimination and security is a possible compiler optimization, built on *liveness analysis*.

A (persistent) (state security violation) is triggered when data remains in memory even if no more necessary.



x becomes skip because it is not live (never used).

By the moment there are issues in DSE optimizations, we implement *translation*

validation.

Translation validation, given an instance, 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 $D_p \subseteq D$;
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:

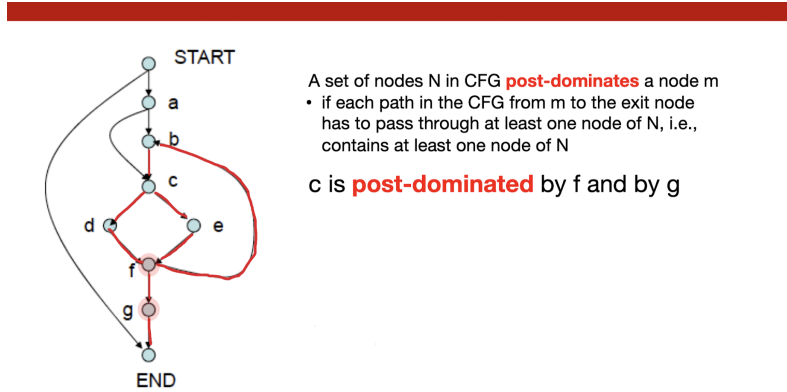
1. **dominates** a node m: if each path in the CFG from the entry node to m has to pass through at least one node of N. It is a **forward analysis**

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

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.)

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



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 (input, output, 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.

4.8.1 Hoare Logic

Axiomatic semantics P S Q

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

- **Q** is a post-condition, all the state reachable.
- (i.e) $\{b \geq 0\} \ a = b+1 \ \{a \geq 1\}$
- 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 false 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 $\{?\} \ a = b+1 \ \{a > 1\} \rightarrow ? = b > 0$ (it is the least restrictive requirement g)

Sequential rule

Formally:

$$\{\text{seq}\} \frac{\{P\} c_1 \{R\} \ \{R\} c_2 \{Q\}}{\{P\} c_1; c_2 \{Q\}}$$

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

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

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

La direzione con cui leggere è da sinistra verso destra.

$$\begin{array}{c}
\{\text{skip}\} \frac{}{\{P\} \text{skip} \{P\}} \qquad \{\text{assign}\} \frac{}{\{Q[a/x]\} x := a \{Q\}} \\
\{\text{seq}\} \frac{\{P\} c_1 \{R\} \{R\} c_2 \{Q\}}{\{P\} c_1; c_2 \{Q\}} \quad \{\text{cons}\} \frac{P \Rightarrow P' \quad \{P'\} c \{Q'\} \quad Q' \Rightarrow Q}{\{P\} c \{Q\}} \\
\{\text{cond}\} \frac{\{P \wedge b\} c_1 \{Q\} \quad \{P \wedge \neg b\} c_2 \{Q\}}{\{P\} \text{if } b \text{ then } c_1 \text{ else } c_2 \{Q\}} \\
\{\text{while}\} \frac{\{P \wedge b\} c \{P\}}{\{P\} \text{while } b \text{ do } c \{P \wedge \neg b\}}
\end{array}$$

4.8.2 Taint proof system

A Taint Proof System aims at tracking the influence of input variables on program state.

Taint = tainted [true], untainted [false]

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

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

Conditional rule: focus on case B

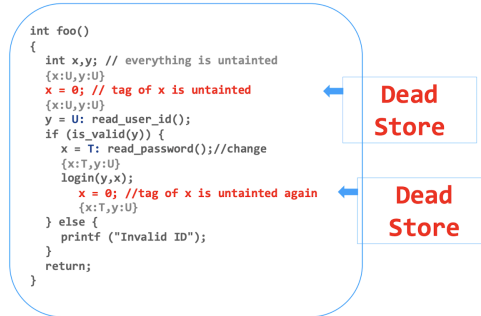
$$\frac{\text{c tainted} \quad \mathcal{E}(c) = \text{true} \quad \{E\} S_1 \{T\} \quad \{E\} S_2 \{T\} \quad \forall x \text{ in } \text{Assign}(S_1) \cup \text{Assign}(S_2): T(x)}{\{E\} \text{if } c \text{ then } S_1 \text{ else } S_2 \{T\}}$$

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

Example

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

Taint analysis at work



Secure DSE procedure

Consider a
candidate dead store
to variable x

It may be removed if:

1) the store is post-dominated by other stores

any leak through x must arise from the dominating stores

2) variable x is untainted before the store and untainted at the exit from the program

the taint proof is unchanged, so a leak cannot arise from x ; other flows are preserved

3) variable x is untainted before the store, other stores to x are unreachable, and this store post-dominates the entry node

• the taint proof is unchanged, so a leak cannot arise from x ; other flows are preserved

The algorithm only ensures that no new leaks are added during the transformation, i.e., the transformation is secure.

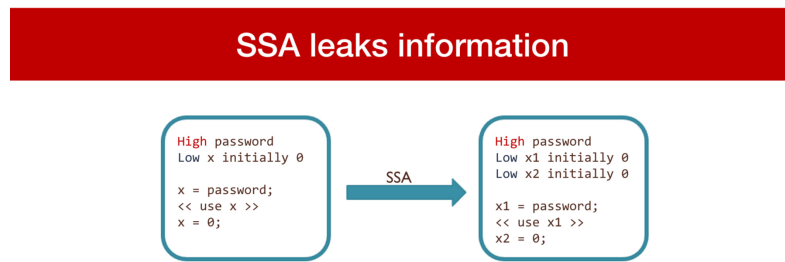
Correctness is assumed: focus is on information leakage.

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

Static Single Assignment (SSA) is insecure.

SSA is a way of structuring the intermediate representation (IR) of programs so that every variable is assigned exactly once and every variable is defined before it is used.

This simplifies register allocation by splitting the live range of variables, but may expose all intermediate values of variables, which may lead to further leaks.



The SSA transform introduces fresh names `x1` and `x2` for the assignments to `x`, with different registers. The `secret password` leaks out through `x1`.

Possible solutions:

- clear all potentially tainted variables before register allocation: inefficient?
- modify SSA to carry auxiliary information about leakage: how?

Compiler optimizations may be correct and yet be not secure.

Ensuring security of DSE through translation validation is difficult.

A probably secure DSE transform based on taint propagation + domination.

4.9 Static Taint Analysis 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 than 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!**

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.

Without GEN/KILL, we would have to assume taints never disappear.

4.9.1 Monotone framework

Lattice $L = \text{untainted} \leq \text{tainted}$

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

Esempi in slide.

4.9.2 Static single assignment

SSA requires that:

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

ϕ : is a function that propose to choose either y1 or y2 depending on the control flow.

SSA Example

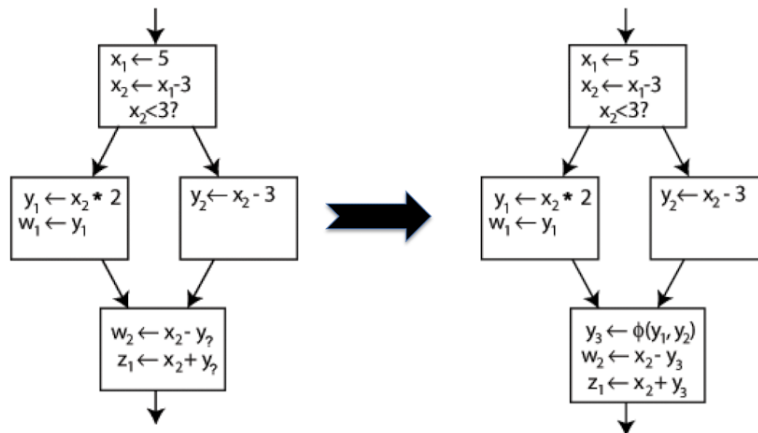


Figure 2:

4.10 Path Sensitive Analysis La **Path Sensitive Analysis** è un tipo di analisi statica in cui si tiene conto dei diversi cammini di esecuzione possibili di un programma, distinguendoli uno ad uno.

In pratica:

- L'analisi non si limita a considerare uno stato unico dopo una diramazione (`if`, `while`, ecc.).
- Mantiene informazioni separate per ogni cammino, ricordando quali condizioni erano vere o false lungo l'esecuzione.
- Questo porta a una precisione molto più alta rispetto alle analisi non-path-sensitive.

Esempio:

```

if (x > 0) {
    y = 1;
} else {
    y = -1;
}
z = 10 / y;
  
```

Con un'analisi *non path-sensitive*, dopo l'`if` sapremmo solo che y può essere 1 o -1, senza ulteriori distinzioni.

Con un'analisi *path-sensitive*, distinguiamo:

- Cammino 1: $x > 0 \Rightarrow y = 1$
- Cammino 2: $x \leq 0 \Rightarrow y = -1$

Quindi possiamo concludere che non esiste alcun rischio di divisione per zero in nessun cammino, grazie alla maggiore precisione.

Differenze principali rispetto alla non-path-sensitive analysis:

Aspetto	Non-Path-Sensitive	Path-Sensitive
Tratta i cammini	Insieme	Singolarmente
Precisione	Più bassa	Più alta
Complessità computazionale	Bassa	Alta (può esplodere)
Uso tipico	Ottimizzazioni veloci	Bug hunting, sicurezza

Quando conviene usarla:

- Quando si cercano bug critici come race condition, tainted data, memory leaks, ecc.
- In analisi di sicurezza di sistemi sensibili (embedded, safety critical).
- Per ridurre falsi positivi.
- Identificare tainted data per ogni paths.

Quando NON conviene:

- Se si analizzano programmi di grandi dimensioni dove serve un'analisi veloce e generale.

Path sensitivity with Taint Values

The analysis exploits path-sensitive predicates (represented by ϕ) in order to associate taint facts with symbolic conditions representing the execution paths.

These paths predicate are used to:

- Distinguish between different execution paths: $\phi_1, \phi_2 \dots$
- Track conditional taint facts of variables
- Evaluate whether taint values may reach the sink under specific conditions
- $(\phi, x) \rightarrow$ variable x is tainted under condition ϕ .

Some auxiliary routines

S = Basic block
 φ = predicate / proprietà
FACTS = variabili TAINTED

TUTTO CIO CHE E GENERATO *TUTTO CIO CHE E KILLATO*

gen_kill(s, φ , facts) = $\langle \text{gen}(s, \varphi, \text{facts}), \text{kill}(s, \varphi, \text{facts}) \rangle$:
 Computes which taint facts are added or removed at node s, under condition φ

SONO I NODI DEL CFG

edgeCondition(n1, n2):
 Returns the condition required to take the conditional branch
 (e.g., φ_1, φ_2)

Restrict(facts, φ):
 Keeps only facts that are consistent with the current path predicate
è UN FILTRO

Consideriamo il seguente pseudocodice:

```
if (x > 0) {
    x = 5;
} else {
    x = x + 1;
}
```

```
if (x > 0) {
    y = 0;
} else {
    y = x;
}
```

```
sink(y);
```

Ogni ramo genera una coppia (path condition, taint values), ad esempio:

- Se $x > 0$, otteniamo $(x > 0, \phi)$
- Se $x \leq 0$, otteniamo $(x \leq 0, \{x, y\})$

La collezione finale delle coppie è:

$$T = \{(x > 0, \phi), (x \leq 0, \{x, y\})\}$$

Operazione di restrict:

- Vogliamo restringere T in base alla condizione $x > 0$
- Si definisce:

$$\text{restrict}(T, x > 0) = \{(x > 0, \phi)\}$$

- Formalmente, **restrict** prende tutte le coppie di T il cui path condition implica $x > 0$.

Nota:

- $(x \leq 0, \{x, y\})$ non soddisfa $x > 0$, quindi viene scartato.

In breve:

- **restrict** filtra le coppie rispetto a una condizione specifica.
- Il risultato è un sottoinsieme del set iniziale T .

The notation **taint**(**x**) is used to indicate that variable x belongs to the set of tainted variables.

The notation **!taint**(**x**) is used to indicate that variable x does not belong to the set of tainted variables (i.e. it has to be removed from that set).

SMT, SAT, UNSAT**Sensitive analysis with monotone framework****Context sensitive Analysis**

5 Static Type for information flow

Type Checking