Program Testing and Analysis: Information Flow Analysis

Cristian-Alexandru Staicu and Dr. Michael Pradel
Software Lab, TU Darmstadt
Outline

1. Introduction
2. Information Flow Policy
3. Analyzing Information Flows
4. Implementation

Mostly based on these papers:

- A Lattice Model of Secure Information Flow, Denning, Comm ACM, 1976
Secure Computing Systems

- Overall goal: Secure the data manipulated by a computing system

- Enforce a security policy
  - Confidentiality: Secret data does not leak to non-secret places
  - Integrity: High-integrity data is not influenced by low-integrity data
Information Flow

■ Goal of information flow analysis:

Check whether information from one "place" propagates to another "place"

☐ For program analysis, "place" means, e.g., code location or variable

■ Complements techniques that impose limits on releasing information

☐ Access control lists
☐ Cryptography
... "Places" in program that hold data

- Secret information
- Possible?
- Trusted information
- Confidentiality
- Integrity

- Untrusted place
Example: Confidentiality

Credit card number should not leak to visible

var creditCardNb = 1234;
var x = creditCardNb;
var visible = false;
if (x > 1000) {
    visible = true;
}

Example: Confidentiality

Credit card number should not leak to visible

```javascript
var creditCardNb = 1234;
var x = creditCardNb;
var visible = false;
if (x > 1000) {
    visible = true;
}
```

Secret information propagates to `x`

Secret information (partly) propagates to `visible`
Example: Integrity

userInput should not influence who becomes president

```javascript
var designatedPresident = "Michael";
var x = userInput();
var designatedPresident = x;
```
Example: Integrity

`userInput` should not influence who becomes president

```javascript
var designatedPresident = "Michael";
var x = userInput();
var designatedPresident = x;
```

Low-integrity information propagates to high-integrity variable
Example: Integrity

userInput should not influence who becomes president

```javascript
var designatedPresident = "Michael";
var x = userInput();
if (x.length === 5) {
    var designatedPresident = "Paul";
}
```
**Example: Integrity**

`userInput` should not influence who becomes president

```javascript
var designatedPresident = "Michael";
var x = userInput();
if (x.length === 5) {
    var designatedPresident = "Paul";
}
```

Low-integrity information propagates to high-integrity variable
Confidentiality vs. Integrity

Confidentiality and integrity are dual problems for information flow analysis

(Focus of this lecture: Confidentiality)
Tracking Security Labels

How to analyze the flow of information?

- Assign to each value some meta information that tracks the secrecy of the value
- Propagate meta information on program operations
Example

```javascript
var creditCardNb = 1234;
var x = creditCardNb;
var visible = false;
if (x > 1000) {
    visible = true;
}
```

---

... = contains secret value
Non-Interference

Property that information flow analysis aims to ensure:

Confidential data does not interfere with public data

- Variation of confidential input does not cause a variation of public output
- Attacker cannot observe any difference between two executions that differ only in their confidential input
Outline

1. Introduction
2. Information Flow Policy
3. Analyzing Information Flows
4. Implementation

Mostly based on these papers:

- A Lattice Model of Secure Information Flow, Denning, Comm ACM, 1976
Lattice of Security Labels

How to represent different levels of secrecy?

- Set of security labels
- Form a universally bounded lattice
Lattice: Examples

High
\downarrow
Low

Top Secret
\downarrow
Secret
\downarrow
Confidential
\downarrow
Public

ABC
\downarrow
AB
\downarrow
A
\downarrow\emptyset
\downarrow
B
\downarrow
BC
\downarrow
C
\downarrow
AC

(Arrows connect more secret classes with less secret classes.)
Universally Bounded Lattice

Tuple \((S, \rightarrow, I, T, \oplus, \otimes)\)

where: \(S\) - set of security classes
\[\{ABC, AB, AC, BC, A, B, C, \emptyset\}\]

\(\rightarrow\) - partial order \(S\) (see figure)

\(I\) - lower bound: \(\emptyset\)

\(T\) - upper bound: \(ABC\)

\(\oplus\) - least upper bound operator, \(S \times S \rightarrow S\)
("combine two pieces of information")
union, e.g., \(AB \oplus A = AB\), \(\emptyset \oplus AC = AC\)

\(\otimes\) - greatest lower bound operator, \(S \times S \rightarrow S\)
intersection, e.g., \(ABC \otimes C = C\)
Quizz: Which of the following is a min. bounded lattice?

1. A
2. Foo
   Bar
3. A
   B
   C
   D
4. ...

D ⊕ E = 3
three common upper bounds (B, C, A), but none is the least upper bound no upper bound (infinite)
Flow Relation

- Partial order on security classes defines a flow relation
- Program is secure if and only if all information flows are described by the flow relation
- Intuition: No flow from higher to lower security class
Information Flow Policy

Policy specifies secrecy of values and which flows are allowed:

- Lattice of security classes
- Sources of secret information
- Untrusted sinks

Goal:
No flow from source to sink
Information Flow Policy

Policy specifies secrecy of values and which flows are allowed:

- Lattice of security classes
- Sources of secret information
- Untrusted sinks

Goal:
No flow from source to sink

```javascript
var creditCardNb = 1234;
var x = creditCardNb;
var visible = false;
if (x > 1000) {
    visible = true;
}
```
Information Flow Policy

Policy specifies **secrecy of values** and which **flows** are allowed:

- Lattice of security classes
- Sources of secret information
- Untrusted sinks

**Goal:**

No flow from source to sink

```javascript
var creditCardNb = 1234;
var x = creditCardNb;
var visible = false;
if (x > 1000) {
    visible = true;
}
```
■ "No flow from high to low" is impractical

■ E.g., code that checks password against a hash value propagates information to subsequence statements
  
  But: This is intended

```javascript
var password = .. // secret
if (hash(password) === 23) {
  // continue normal program execution
} else {
  // display message: incorrect password
}
```
Declassification

- "No flow from high to low" is impractical
- E.g., code that checks password against a hash value propagates information to subsequent statements
  
  But: This is intended

```javascript
var password = .. // secret
if (hash(password) === 23) {
    // continue normal program execution
} else {
    // display message: incorrect password
}
```

Declassification: Mechanism to remove or lower security class of a value
Outline

1. Introduction
2. Information Flow Policy
3. Analyzing Information Flows
4. Implementation

Mostly based on these papers:

Analyzing Information Flows

Given an information flow policy, analysis **checks for policy violations**

**Applications:**

- Detect **vulnerable code** (e.g., potential SQL injections)
- Detect **malicious code** (e.g., privacy violations)
- Check if program **behaves as expected** (e.g., secret data should never be written to console)
Explicit vs. Implicit Flows

- **Explicit flows**: Caused by data flow dependence
- **Implicit flows**: Caused by control flow dependence
Explicit vs. Implicit Flows

- **Explicit flows**: Caused by data flow dependence
- **Implicit flows**: Caused by control flow dependence

```javascript
var creditCardNb = 1234;
var x = creditCardNb;
var visible = false;
if (x > 1000) {
  visible = true;
}
```
Explicit vs. Implicit Flows

- **Explicit flows**: Caused by data flow dependence
- **Implicit flows**: Caused by control flow dependence

```javascript
var creditCardNb = 1234;
var x = creditCardNb;
var visible = false;
if (x > 1000) {
    visible = true;
}
```

Explicit flow from `creditCardNb` to `x`

Implicit flow from `x > 1000` to `visible`
Explicit vs. Implicit Flows

- **Explicit flows**: Caused by data flow dependence
- **Implicit flows**: Caused by control flow dependence

```javascript
var creditCardNb = 1234;
var x = creditCardNb;
var visible = false;
if (x > 1000) {
    visible = true;
}
```

Explicit flow from `creditCardNb` to `x`

Implicit flow from `x > 1000` to `visible`
Static and Dynamic Analysis

- **Static information flow analysis**
  - Overapproximate all possible data and control flow dependences
  - Result: Whether information "may flow" from secret source to untrusted sink

- **Dynamic information flow analysis**
  - Associate security labels ("taint markings") with memory locations
  - Propagate labels at runtime
Static and Dynamic Analysis

- **Static information flow analysis**
  - Overapproximate all possible data and control flow dependences
  - Result: Whether information "may flow" from secret source to untrusted sink

- **Dynamic information flow analysis**
  - Associate security labels ("taint markings") with memory locations
  - Propagate labels at runtime

Focus of rest of this lecture
Taint Sources and Sinks

- **Possible sources:**
  - Variables
  - Return values of a particular function
  - Data from a type of I/O stream
  - Data from a particular I/O stream
Taint Sources and Sinks

- Possible sources:
  - Variables
  - Return values of a particular function
  - Data from a type of I/O stream
  - Data from a particular I/O stream

- Possible sinks:
  - Variables
  - Parameters given to a particular function
  - Instructions of a particular type (e.g., jump instructions)
Taint Sources and Sinks

■ Possible sources:
  □ Variables
  □ Return values of a particular function
  □ Data from a type of I/O stream
  □ Data from a particular I/O stream

■ Possible sinks:
  □ Variables
  □ Parameters given to a particular function
  □ Instructions of a particular type (e.g., jump instructions)

Report illegal flow if taint marking flows to a sink where it should not flow
1) Explicit flows

For every operation that produces a new value, propagate labels of inputs to label of output:

\[ \text{label(result)} \leftarrow \text{label(inp}_1) \oplus \ldots \oplus \text{label(inp}_2) \]
2) Implicit flows

- Maintain security stack $S$: Labels of all values that influence the current flow of control.
- When $x$ influences a branch decision at location $loc$, push $\text{label}(x)$ on $S$.
- When control flow reaches immediate post-dominator of $loc$, pop $\text{label}(x)$ from $S$.
- When an operation is executed while the $S$ is non-empty, consider all labels on $S$ as input to the operation.
Example 1

Policy:
- security classes: public, secret
- source: variable "creditCardNb"
- sink: variable "visible"

```javascript
var creditCardNb = 1234;
var x = creditCardNb;
var visible = false;
if (x > 1000) {
    visible = true;
}
```

label (creditCardNb) = secret

explicit flow: label (x) = secret

label (visible) = public

produce intermediate value b,
label (b) = label (x) ⊕ label (1000)
= secret ⊕ public = secret

push secret on S

labels on S are part of input
label (visible) = secret ⊕ label (true)
= secret

⇒ violation of policy
Example 2: Quiz

```
var x = getX();
var y = x + 5;
var z = true;
if (y === 10)
    z = false;
foo(z);
```

Policy:
- Security classes: public, secret
- Source: `getX`
- Sink: `foo()`

Suppose that `getX` returns 5. Write down the labels after each operation. Is there a policy violation?
Outline

1. Introduction
2. Information Flow Policy
3. Analyzing Information Flows
4. Implementation

Mostly based on these papers:

- A Lattice Model of Secure Information Flow, Denning, Comm ACM, 1976
Example 2: Quiz

```javascript
var x = getX();
var y = x + 5;
var z = true;
if (y === 10) {
    z = false;
}
foo(z);
```

Policy:
- Security classes: public, secret
- Source: `getX`
- Sink: `foo()`

Suppose that `getX` returns 5. Write down the labels after each operation.

Is there a policy violation?
Example 2

```javascript
var x = getX();
var y = x + 5;
var z = true;
if (y === 10) {
    z = false;
}
foo(z);
```

label(x) = secret

label(y) = label(x) \oplus (label(5))
          = secret

label(z) = public

yields "b", label(b) = secret,
push secret ...,

label(z) = secret \oplus public = secret

pop secret

violation because z is secret
Hidden Implicit Flows

- Implicit flows may happen even though a branch is not executed
- Approach explained so far will miss such "hidden" flows

```javascript
// label(x) = public, label(secret) = private
var x = false;
if (secret)
    x = true;
```
Hidden Implicit Flows

- Implicit flows may happen even though a branch is not executed
- Approach explained so far will miss such "hidden" flows

```javascript
// label(x) = public, label(secret) = private
var x = false;
if (secret)
  x = true;

Copies secret into x
But: Execution where secret is false does not propagate anything
Hidden Implicit Flows (2)

Approach to reveal hidden flows:

For every conditional with branches $b_1$ and $b_2$:

- Conservatively overapproximate which values may be defined in $b_1$
- Add spurious definitions into $b_2$
Hidden Implicit Flows (2)

Approach to reveal hidden flows:
For every conditional with branches $b_1$ and $b_2$:

- Conservatively overapproximate which values may be defined in $b_1$
- Add spurious definitions into $b_2$

```javascript
var x = false;
if (secret)
    x = true;
else
    x = x; // spurious definition
```

All executions propagate "secret" label to $x$
Outline

1. Introduction
2. Information Flow Policy
3. Analyzing Information Flows
4. Implementation

Mostly based on these papers:

Implementation in Dytan

Dynamic information flow analysis for x86 binaries

- Taint markings stored as bit vectors
- One bit vector per byte of memory
- Propagation implemented via instrumentation (i.e., add instructions to existing program)
- Computes immediate post-dominators via static control flow graph
Information Flow: Summary

- Information flow analysis:
  Track secrecy of information handled by program

- Goal: Check information flow policy
  - Security classes, sources, sinks

- Various applications
  - E.g., malware detection, check for vulnerabilities

- There exist channels missed by information flow analysis
  - E.g., power consumption, timing