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
- *Dytan: A Generic Dynamic Taint Analysis Framework*, Clause et al., ISSTA 2007

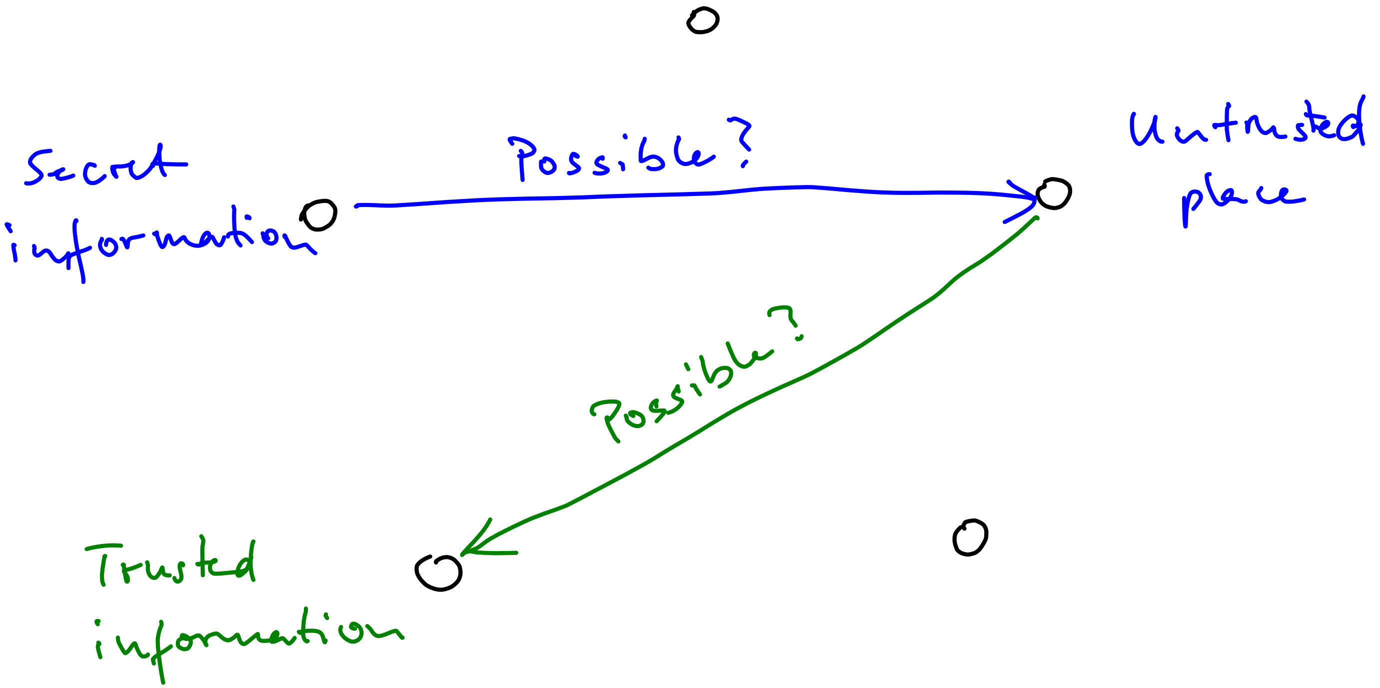
Secure Computing Systems

- **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



Confidentiality

Integrity

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`

```
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**

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

Example: Integrity

`userInput` should not influence who becomes president

```
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

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

Example: Integrity

`userInput` should not influence who becomes president

```
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

secret value

```
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
- *Dytan: A Generic Dynamic Taint Analysis Framework*, Clause et al., ISSTA 2007

Lattice of Security Labels

How to represent different **levels of secrecy**?

- Set of security labels
- Form a **universally bounded lattice**

Lattice: Examples

$$\begin{array}{c} \text{High} \\ \downarrow \\ \text{Low} \end{array}$$

$$\begin{array}{c} \text{Top Secret} \\ \downarrow \\ \text{Secret} \\ \downarrow \\ \text{Confidential} \\ \downarrow \\ \text{Public} \end{array}$$

$$\begin{tikzcd}
& ABC \arrow[dl] \arrow[d] \arrow[dr] & \\
AB \arrow[d] \arrow[dr] & BC \arrow[d] \arrow[dr] & AC \arrow[dll] \arrow[d] \\
A \arrow[dr] & B \arrow[d] & C \arrow[dl] \\
& \emptyset &
\end{tikzcd}$$

(Arrows connect more secret class with less secret class.)

Universally Bounded Lattice

Tuple $(S, \rightarrow, \perp, \top, \oplus, \otimes)$

where: S .. set of security classes

$\{ABC, AB, AC, BC, A, B, C, \emptyset\}$

$\rightarrow$ .. partial order S (see figure)

$\perp$ .. lower bound : $\emptyset$

$\top$ .. 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. $ABC \otimes C = C$

Quiz: Which of the following is a univ. bounded lattice?

①
$$\begin{tikzcd} & A \arrow[dl] \arrow[dr] & \\ B \arrow[dr] & & C \arrow[dl] \\ & D & \end{tikzcd}$$

②
$$\begin{tikzcd} Foo \arrow[d] \\ Bar \arrow[d] \\ Baz \end{tikzcd}$$

③
$$\begin{tikzcd} & A \arrow[dl] \arrow[dr] & \\ B \arrow[d] \arrow[dr] & & C \arrow[d] \arrow[dl] \\ D \arrow[dr] & & E \arrow[dl] \\ & F & \end{tikzcd}$$

$D \oplus E = ?$

three common upper bounds (B, C, A), but none is the least upper bound

④
$$\begin{tikzcd} \dots \arrow[d] \\ 3 \arrow[d] \\ 2 \arrow[d] \\ 1 \arrow[d] \\ 0 \end{tikzcd}$$

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

```
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

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

Goal:

No flow from source to sink

Declassification

- ”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

```
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 subsequence statements

  But: This is intended

```
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:

- *A Lattice Model of Secure Information Flow*, Denning, Comm ACM, 1976
- *Dytan: A Generic Dynamic Taint Analysis Framework*, Clause et al., ISSTA 2007

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

```
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**

```
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

Some analyses consider only these

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

```
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

Taint Propagation

1) **Explicit flows**

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

$$label(result) \leftarrow label(inp_1) \oplus ... \oplus label(inp_2)$$

Taint Propagation (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 $label(x)$ on S
- When control flow reaches immediate post-dominator of loc, pop $label(x)$ 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"

```
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
- *Dytan: A Generic Dynamic Taint Analysis Framework*, Clause et al., ISSTA 2007

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?

Example 2

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

label (x) = secret

label (y) = label (x) ⊕ label (5)

= secret

label (z) = public

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

push secret

label (z) = secret ⊕ 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

```
// 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

```
// 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

```
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:

- *A Lattice Model of Secure Information Flow*, Denning, Comm ACM, 1976
- *Dytan: A Generic Dynamic Taint Analysis Framework*, Clause et al., ISSTA 2007

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