Program Analysis

Information Flow Analysis (Part 3)

Prof. Dr. Michael Pradel

Software Lab, University of Stuttgart

Winter 2020/2021

Outline

1. **Introduction**
2. **Information Flow Policy**
3. **Analyzing Information Flows**

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 particular I/O stream

Taint Sources and Sinks

- **Possible sources**:
  - Variables
  - Return values of a particular function
  - 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 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 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" onto S

labels on S are part of input

label(visible) = secret ⊕ label(true)
= secret ⊕ public = 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?

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`

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

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