Program Testing and Analysis:
Slicing (Part 2)

Dr. Michael Pradel
Software Lab, TU Darmstadt
What does the following code print?

```javascript
var x = 23;
function f() { console.log(this.x); }
var obj = Object.create({ f: f });
obj.x = 42;
f();
f();
obj.f();
```

23  42  23  42
23  42  42  23
Warm-up Quiz

What does the following code print?

```javascript
var x = 23;
function f() { console.log(this.x); }
var obj = Object.create({ f: f });
obj.x = 42;
f();
obj.f();
```

23  42
23  42

obj’s prototype has method f

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>23</td>
</tr>
<tr>
<td>42</td>
<td>42</td>
</tr>
</tbody>
</table>
Warm-up Quiz

What does the following code print?

```javascript
var x = 23;
function f() { console.log(this.x); }
var obj = Object.create({ f: f });
obj.x = 42;
f();
obj.f();
```

- Simple call: global
- Object method: base object
Slicing: Outline

1. Introduction
2. Static Slicing
3. Thin Slicing
4. Dynamic Slicing

Mostly based on these papers:

- *Program Slicing*, Weiser., IEEE TSE, 1984
- *Thin Slicing*, Sridharan et al., PLDI 2007
- *Dynamic Program Slicing*, Agrawal and Horgan, PLDI 1990
Dynamic Slice (Simple Approach)

- **Given:** Execution history
  - Sequence of PDG nodes that are executed

- **Slice for statement** $n$ **and variable** $v$:
  - Keep PDG nodes only if there are in history
  - Use static slicing approach (= graph reachability) on reduced PDG
Example 2

\[
\text{var } n = \text{readInput}(); \\
\text{var } z = 0; \\
\text{var } y = 0; \\
\text{var } i = 1; \\
\text{while } (i \leq n) \{
    z = z + y; \\
    y = y + 1; \\
    i = i + 1;
\}
\text{console.log}(z);
\]

Input: \quad n = 1 \\
History: \quad 1, 2, 3, 4, 5, 6, 7, 8, 5, 9 \\
\circ \quad \text{in history} \\
\circ \quad \text{slice \ (9, 5??) } \\
\quad \quad = \text{all statements} \\
BUT: \\
Statement 7 is not relevant!

\[1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9\]
Limitations of Simple Approach

- **Multiple occurrences** of a single statement are represented as a **single PDG node**

- **Difference occurrences** of a statement may have different dependences
  - All occurrences get **conflated**

- **Slices** may be **larger than necessary**
Dynamic Slice (Revised Approach)

Dynamic dependence graph

- Nodes: Occurrences of nodes of static PDG
- Edges: Dynamic data and control flow dependences

Slice for statement $n$ and variables $V$ that are defined or used at $n$:

- Compute nodes $S_{dyn}$ that can reach any of the nodes that represent occurrences of $n$
- Slice = statements with at least one node in $S_{dyn}$
Example 2 (Revised approach)

```javascript
var n = readInput();
var z = 0;
var y = 0;
var i = 1;

while (i <= n) {
    z = z + y;
    y = y + 1;
    i = i + 1;
}

console.log(z);
```

Input:  \( n = 7 \)

History: 1, 2, 3, 4, 5, 6, 7, 8, 9, 0

\( \ldots \text{slice (9, \{2\})} \)
Discussion: Dynamic Slicing

- May yield a program that, if executed, does not give the same value for the slicing criterion than the original program.

- Instead: Focuses on isolating statements that affect a particular value.
  - Useful, e.g., for debugging and program understanding.

- Other approaches exist, see F. Tip’s survey for an overview.
Slicing: Summary

- **Program slicing**: Extract subset of statements for a particular purpose
  - Debugging, program understanding, change impact analysis, parallelization

- **Various techniques**
  - **Traditional static slicing**: Executable but potentially very large slice
  - **Thin slicing**: Focus on producer statements, reveal explainer statements on demand
  - **Dynamic slicing**: Useful for understanding behavior of particular execution
Program Testing and Analysis: 
Information Flow Analysis

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? ➔ Untrusted place

Possible? ➔ Trusted information

Confidentiality
Integrity
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;
}
```
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;
}
```
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:

Lattice of Security Labels

How to represent different levels of secrecy?

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

High
↓
Low

Top Secret
↓
Secret
↓
Confidential
↓
Public

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

Tuple (S, →, ⊥, ⊤, ⊕, ⊗)

where: S - set of security classes

{ABC, AB, AC, BC, A, B, C, ⊥}

→ partial order S (see figure)

⊥ lower bound : ⊥

⊤ upper bound : ABC

⊕ least upper bound operator, S×S → S

("combine two pieces of information")

union, e.g., AB ⊕ A = AB, ⊥ ⊕ AC = AC

⊗ greatest lower bound operator, S×S → S

intersection, e.g., ABC ⊗ C = C
Quiz: Which of the following is a min. bounded lattice?

1) A
2) Foo → Bar → Baz
3) A → B → C → D → F
4) i

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 subsequent 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 subsequence 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

Some analyses consider only these

```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
Taint Propagation

1) **Explicit flows**

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

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

```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?