Program Testing and Analysis: Information Flow Analysis (Part 2)

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

```javascript
function foo(a, a, a) {
    console.log(a);
}

foo("this", "that", "or maybe this");
```

"this"  "that"  "or maybe this"  Something else
What does the following code print?

```javascript
function foo(a, a, a) {
    console.log(a);
}
foo("this", "that", "or maybe this");
```

"this"   "that"   "or maybe this"   Something else
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
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
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
Program Testing and Analysis: Specification Mining

Dr. Michael Pradel
Software Lab, TU Darmstadt

Partly based on slides from Michael Ernst, Andreas Zeller, and Andrzej Wasylkowski
Outline

1. Introduction
2. Program Invariants
3. Finite State Machines
4. Programming Rules

Mostly based on these papers:

- *Dynamically Discovering Likely Program Invariants to Support Program Evolution*, Ernst et al., IEEE TSE, 2001
- *Mining Specifications*, Ammons et al., POPL, 2002
Formal Specifications

- Formal, mathematical description of the intended behavior of a program

Examples:

- Pre- and post-conditions:
  ```javascript
  // pre: typeof(x) === "number"
  function abs(x) { ... }
  // post: typeof(ret) === "number" &&
  //       ret >= 0
  ```

- Finite-state machines:
Uses of Specifications

Traditionally, mainly used for formal verification

- Demonstrate that program is correct w.r.t. its specification
- Mathematical proof
- Ideally, static verification
  - Avoid running an incorrect program
- Also: Runtime verification
  - Detect and potentially prevent problems when they happen
The Problem

So why not formally specify and verify all software?

- Huge effort
- Completely specifying a large system is practically impossible
- Complex specification is likely to have mistakes
- In practice:
  - Used mostly for safety critical systems
  - Used only to specify important properties (e.g., no crash)
Specification Mining

- **Idea:** Infer specifications from existing software
  - No human effort
  - Get specifications "for free"

- **Examples:**
  - Pre- and post-conditions:
    Analyze function and check which properties the inputs and outputs fulfill
  - Finite state machines:
    Analyze code and identify its states and transitions between them
Wait a Minute ...

- How to validate that a program is correct by inferring the specification from the program itself?
- Sounds contradictory, but it works
  - Infer common behavior, report anomalies as potential bugs
  - Infer specifications from one code base, use them to check another
    - Different programs that use the same API
    - Different versions of the same program
  - Detect inconsistencies in the code itself (non-null assumption vs. null check)
Uses of Mined Specifications

- **Software evolution**
  - Understand behavior of program
  - Generate documentation
  - Use as oracle for regression testing

- **Anomaly detection**
  - Outliers are potential bugs

- **Support formal specification of an existing system**
  - Starting point for full specification
Outline

1. Introduction
2. Program Invariants
3. Finite State Machines
4. Programming Rules

Mostly based on these papers:

- *Dynamically Discovering Likely Program Invariants to Support Program Evolution*, Ernst et al., IEEE TSE, 2001
- *Mining Specifications*, Ammons et al., POPL, 2002
Program Invariants

- Invariant = **Data property that holds in all runs**
  - At entry of f(), x is an odd number
  - $0 \leq y \leq 10$

- Useful in software development
  - Protect programmers from making errant changes
  - Verify properties of a program

- Can be explicitly stated in programs
  - Programmers can annotate code with invariants
  - Huge effort
  - Important invariants may be missed
function f(b, n) {
    var i = 0, s = 0;
    while (i !== n) {
        s += b[i];
        i++;
    }
    return s;
}
**Example**

Some invariants from running with 100 randomly generated inputs of length 7-13:

- **Pre-conditions:**
  - \( n = b.length \)
  - \( 7 \leq n \leq 13 \)

- **Post-conditions:**
  - \( n = i = b.length \)
  - \( b = orig(b) \)
  - \( s = sum(b) \)

- **Loop invariants**
  - \( n = b.length \)
  - \( 0 \leq i \leq 13 \)
  - \( s = sum(b[0..i - 1]) \)

```javascript
function f(b, n) {
    var i = 0, s = 0;
    while (i !== n) {
        s += b[i];
        i++;
    }
    return s;
}
```
Daikon Invariant Detector

■ **Dynamic analysis**: Infers invariants from particular execution

■ **Step 1**: *Instrument* source code
  □ Trace variables of interest

■ **Step 2**: *Run* instrumented program using test suite

■ **Step 3**: *Infer invariants* from instrumented and derived variables
Step 1: Instrumentation

Insert instrumentation points

- Function entry
- Function exit
- Loop heads

Write to a file values for

- all variables in scope
- global variables
- function arguments
- local variables
- function’s return value
Step 2: Execution

- Instrumented program writes file with runtime values
- Result: Trace of execution
Step 3: Inference

Daikon has library of invariant patterns over variables (e.g., $x, y, z$) and constants (e.g., $a, b, c$), e.g.:

- Check for each variable:
  - Constant or small number of values

- Check for numeric variables:
  - Range: $a \leq x \leq b$

- Check for multiple numbers:
  - Set of functions, e.g., $x = \text{abs}(y)$
  - Comparisons, e.g., $x < y$

- Check for sequences:
  - Sortedness
Daikon has library of invariant patterns over variables (e.g., \(x, y, z\)) and constants (e.g., \(a, b, c\)), e.g.:

- Check for each variable:
  - Constant or small number of values

- Check for numeric variables:
  - Range: \(a \leq x \leq b\)

- Check for multiple numbers:
  - Set of functions, e.g., \(x = \text{abs}(y)\)
  - Comparisons, e.g., \(x < y\)

- Check for sequences:
  - Sortedness

Only matching patterns are preserved
What post-conditions could Daikon infer from tests \(g(1)\) and \(g(3)\)?

```javascript
function g(n) {
    var x = n * 2;
    var y = 0;
    for (var i = 0; i < x; i++) {
        y += i;
    }
    return y;
}
```

\[x < y \quad i \geq 2 \quad 1 \leq y \leq 15 \quad n = 1\]
Quiz

What post-conditions could Daikon infer from tests \( g(1) \) and \( g(3) \)?

```javascript
function g(n) {
    var x = n * 2;
    var y = 0;
    for (var i = 0; i < x; i++) {
        y += i;
    }
    return y;
}
```

\[ x < y \quad i \geq 2 \quad 1 \leq y \leq 15 \quad n = 1 \]
Outline

1. Introduction
2. Program Invariants
3. Finite State Machines
4. Programming Rules

Mostly based on these papers:

- *Dynamically Discovering Likely Program Invariants to Support Program Evolution*, Ernst et al., IEEE TSE, 2001
- *Mining Specifications*, Ammons et al., POPL, 2002
Example: Socket API

```c
int s = socket(AF_INET, SOCK_STREAM, 0);
bind(s, &serv_addr, sizeof(serv_addr));
listen(s, 5);
while (true) {
    int ns = accept(s, &addr, &len);
    if (ns < 0) break;
    do {
        read(ns, buffer, 255);
        write(ns, buffer, size);
        if (cond1) return;
    } while (cond2)
    close(ns);
}
close(s);
```
Automata Mining

- Many APIs impose usage protocols on its clients
  - Not formally specified, but implicit in implementation/documentation

- Idea: Dynamically analyze API usage of clients and infer protocols
  - Assumption: Most API usages are correct

- Approach:
  - Instrument and execute program → Trace
  - Extract scenarios = small sequences of dependent API calls
  - Learn finite state machine
Execution Trace

socket(domain = 2, type = 1, proto = 0, return = 7)
bind(so = 7, addr = 0x400120, addr_len = 6, return = 0)
listen(so = 7, backlog = 5, return = 0)
accept(so = 7, addr = 0x400200, addr_len = 0x400240, return = 8)
read(fd = 8, buf = 0x400320, len = 255, return = 12)
write(fd = 8, buf = 0x400320, len = 12, return = 12)
read(fd = 8, buf = 0x400320, len = 255, return = 7)
write(fd = 8, buf = 0x400320, len = 7, return = 7)
close(fd = 8, return = 0)
accept(so = 7, addr = 0x400200, addr_len = 0x400240, return = 10)
read(fd = 10, buf = 0x400320, len = 255, return = 13)
write(fd = 10, buf = 0x400320, len = 13, return = 13)
close(fd = 10, return = 0)
close(fd = 7, return = 0)
socket(domain = 2, type = 1, proto = 0, return = 7)
bind(so = 7, addr = 0x400120, addr_len = 6, return = 0)
listen(so = 7, backlog = 5, return = 0)
accept(so = 7, addr = 0x400200, addr_len = 0x400240, return = 8)
read(fd = 8, buf = 0x400320, len = 255, return = 12)
write(fd = 8, buf = 0x400320, len = 12, return = 12)
read(fd = 8, buf = 0x400320, len = 255, return = 7)
write(fd = 8, buf = 0x400320, len = 7, return = 7)
close(fd = 8, return = 0)
accept(so = 7, addr = 0x400200, addr_len = 0x400240, return = 10)
read(fd = 10, buf = 0x400320, len = 255, return = 13)
write(fd = 10, buf = 0x400320, len = 13, return = 13)
close(fd = 10, return = 0)
close(fd = 7, return = 0)
Execution Trace

socket(domain = 2, type = 1, proto = 0, return = 7)
bind(so = 7, addr = 0x400120, addr_len = 6, return = 0)
listen(so = 7, backlog = 5, return = 0)
accept(so = 7, addr = 0x400200, addr_len = 0x400240, return = 8)
read(fd = 8, buf = 0x400320, len = 255, return = 12)
write(fd = 8, buf = 0x400320, len = 12, return = 12)
read(fd = 8, buf = 0x400320, len = 255, return = 7)
write(fd = 8, buf = 0x400320, len = 7, return = 7)
close(fd = 8, return = 0)
accept(so = 7, addr = 0x400200, addr_len = 0x400240, return = 10)
read(fd = 10, buf = 0x400320, len = 255, return = 13)
write(fd = 10, buf = 0x400320, len = 13, return = 13)
close(fd = 10, return = 0)
close(fd = 7, return = 0)
Inferred Specification

socket (return=x)
  └── bind (so=x)
  │   └── listen (so=x)
  │       └── accept (so=x, return=y)
  │           └── read (fd=y)
  │               └── write (fd=y)
  │                   └── close (fd=y)
  └── close (fd=x)
Outline

1. Introduction
2. Program Invariants
3. Finite State Machines
4. Programming Rules

Mostly based on these papers:

- *Dynamically Discovering Likely Program Invariants to Support Program Evolution*, Ernst et al., IEEE TSE, 2001
- *Mining Specifications*, Ammons et al., POPL, 2002
PR-Miner

- **Static** specification mining technique
- Idea: Find *implicit programming rules* and warn about paths where rules are violated
  - E.g., using one variable/function implies using another
- **Main steps:**
  - Extract *symbols names* from each function
  - Use *frequent itemset mining* to identify symbols that often occur together → Programming rules
  - Search paths where one symbol of is used but not another → *Violations of rules*
// function source code
getRelationDescription (...) { 
    HeapTuple relTup;
    ...
    relTup = SearchSysCache (...);
    if (!HeapTupleIsValid (relTup))
       elog (...);
    relForm = ...;
    ...
    ReleaseSysCache (relTup);
}
Example

// function source code
getRelationDescription (...) {
    HeapTuple relTup;
    ...
    relTup = SearchSysCache (...);
    if (!HeapTupleIsValid (relTup))
        elog (...);
    relForm = ...;
    ...
    ReleaseSysCache (relTup);
}

Itemset:
HeapTuple
SearchSysCache
HeapTupleIsValid
elog
ReleaseSysCache
...
Example (2)

More itemsets:

<table>
<thead>
<tr>
<th>HeapTuple</th>
<th>StringInfoData</th>
<th>Form_pg_class</th>
</tr>
</thead>
<tbody>
<tr>
<td>SearchSysCache</td>
<td>getObjectClass</td>
<td>SearchSysCache</td>
</tr>
<tr>
<td>HeapTupleIsValid</td>
<td>HeapTuple</td>
<td>elog</td>
</tr>
<tr>
<td>elog</td>
<td>SearchSysCache</td>
<td>RelationIsVisible</td>
</tr>
<tr>
<td>ReleaseSysCache</td>
<td>NameStr</td>
<td>ReleaseSysCache</td>
</tr>
<tr>
<td>...</td>
<td>Relation</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>ReleaseSysCache</td>
<td></td>
</tr>
</tbody>
</table>
Example (2)

More itemsets:
- HeapTuple
- SearchSysCache
- HeapTupleIsValid
- elog
- ReleaseSysCache
- StringInfoData
- getObjectClass
- HeapTuple
- SearchSysCache
- NameStr
- Relation
- ReleaseSysCache
- ... 
- Form_pg_class
- SearchSysCache
- elog
- RelationIsVisible
- ReleaseSysCache
- ... 
- ... 

- Frequent itemset:
  SearchSysCache, ReleaseSysCache

- Missing items point to potential bugs
Summary

- **Specification mining:**
  Extract formal specification from existing program
  - Often builds on data mining techniques

- Useful for **bug detection, program understanding, documentation, etc.**

- **Main challenge:** Keep false positive warnings at reasonable level

- **Active research topic** with many interesting, open questions