Towards Static Flow-based Declassification for Legacy and Untrusted Programs

Bruno P. S. Rocha
Sruthi Bandhakavi
Jerry den Hartog
William H. Winsborough
Sandro Etalle

IEEE Symposium on Security and Privacy 2010
(Oakland’10)

Introduction

• Language-based information flow aims to analyze programs with respect to flow of information between channels of different security levels

• Non-interference is a formal property for specifying valid flows (Goguen & Meseguer 1982)

Non-interference

Public Input

Private Input

Program

Public Output

Change in the public input can change the output
Change in the private input should not change the output

Declassification

• Non-interference is not enough for most practical applications

• In many occasions, it is necessary to downgrade the security level of specific data i.e., to declassify that data

• Classic examples:
  – Average salary
  – Password verification
  – Encryption

Security-typed languages

• Common approach (Jif)

• Variables have security types

• Compilation-time analysis rejects programs which contain insecure flows

• Declassification is usually associated with specific points in the code, and done explicitly via some declass command

  – Declassification policy is code-dependant

• Problem: implies that programmer is trusted, and understands security labeling of variables

Example: Jif

Static labels + declassification:

```plaintext
int[ Alice;Alice: Bob; ] i; int[ Alice:Bob; ] y = 0;
if (b) {
  // pc is at level {Alice:;Alice←*} at this point.
  declassify ( {Alice:;Alice←*} to {y} ) {
    // at this point, pc has been declassified to the label of the local
    // variable y (that is, {Alice:Bob;Alice←*} ) permitting the
    // assignment to y
    y = 1;
  }
}
```

Dynamic labels:

```plaintext
int[ Alice:Bob; ] i; int[ Alice:Bob; ] j, int[ Alice: ] i;
where [ Alice:Bob ] = lb, Bob actsfor p

// since Bob actsfor p, {Alice:Bob} <= {Alice:Bob}
// and since {Alice:Bob} <= lb, the label of the argument /
// to the function is <= {lb}. Therefore, we can return i+1.
return i+1;
```
Non security-typed approaches

- **Dataflow analysis** are techniques which do not rely on annotated code
  - Do not provide declassification
  - Have issues analyzing control-flow dependencies
- **Taint analysis** also works on unannotated code, but suffers from the same problems of dataflow analysis + is often too restrictive

Our contribution

- We aim for a static flow-based analysis that provides 3 key points:
  1. Analysis of programming code without any security-based annotations (therefore, untrusted code)
  2. Support for user-defined declassification policies
  3. Having code and policy separated and independent from each other
- Individual solutions do exist
  - The true challenge lies in a combined solution
- A first-step in a new direction for information flow analysis

The core analysis process

1. Identify I/O operations
2. Extract expressions on inputs
3. Check if expressions are recognized by policy

Policy Representation

Sets of expressions are represented by graphs

Nodes labels with input channels, constant values or wildcard characters

Certain nodes are marked as final nodes

Loop expressions can also be expressed

Represents the expression set: \( \{0, (0 + \alpha_1), (0 + \alpha_1 + \alpha_2), \ldots \} \)

A separated restriction (not discussed here) guarantees that indexes on \( \alpha \) are unique

Step 1: Preprocessing

Operators are converted to method calls, nested expressions are converted to a series of assignments to temp variables

```
sum := 0; i := 0;
len := length(a);
while (i <= len) do
  val := \( \alpha \);
  sum := sum + val;
  i := i + 1;
avg := sum / len;
y := avg;
```

```
sum := 0; i := 0;
len := length(a);
c := leq(i, len);
while (c) do
  val := \( \alpha \);
  sum := add(sum, val);
  i := add(i, 1);
c := leq(i, len);
avg := div(sum, len);
y := avg;
```

Every variable is defined only once & Phi functions are declared to keep track of assignments in branching statements

Step 2: Static Single Assignment (SSA)

Format

```
sum := 0; i := 0;
len := length(a);
c := leq(i, len);
while (c) do
  val := \( \alpha \);
  sum := add(sum, val);
  i := add(i, 1);
c := leq(i, len);
avg := div(sum, len);
y := avg;
```

```
sum := \( \phi_c^{\phi_i}(sum, \alpha) \);
sum := \( \phi_c^{\phi_i}(sum, \alpha) \);
i := add(i, 1);
c := leq(i, len);
avg := div(sum, len);
y := avg;
```

Every variable is defined only once & Phi functions are declared to keep track of assignments in branching statements

Our contribution

- We aim for a static flow-based analysis that provides 3 key points:
  1. Analysis of programming code without any security-based annotations (therefore, untrusted code)
  2. Support for user-defined declassification policies
  3. Having code and policy separated and independent from each other
- Individual solutions do exist
  - The true challenge lies in a combined solution
- A first-step in a new direction for information flow analysis
Step 3: Program Expression Graph

\[ \text{sum}_1 := 0; \quad i_1 := 0; \]
\[ \text{len}_1 := \text{length}(\alpha); \]
\[ c_1 := \text{leq}(i_1, \text{len}_1); \]
\[ \text{while} \{ c_1 := \phi_1(c_1, c_3); \]
\[ \text{sum}_2 := \phi_2(\text{sum}_1, \text{sum}_3); \]
\[ i_3 := \phi_3(i_1, i_3); \]
\[ c_3 := \phi_3(c_1, c_2); \]
\[ \text{do} \]
\[ \text{val}_1 := \alpha; \]
\[ \text{sum}_2 := \text{add}(\text{sum}_3, \text{val}_1); \]
\[ i_2 := \text{add}(i_3, 1); \]
\[ c_2 := \text{leq}(i_2, \text{len}_1); \]
\[ \text{avg}_1 := \text{div}(\text{sum}_3, \text{len}_1); \]
\[ y := \text{avg}_1; \]

Expression graph for \( y \)

Step 4: Policy Matching

Revisiting the security property

Policy Controlled Release (PCR)

Did I mention “towards” something?

Thank you!

Questions?