Towards Static Flow-based Declassification for Legacy and Untrusted Programs

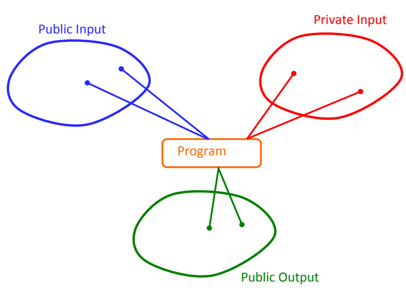
Bruno P. S. Rocha
Sruthi Bandhakavi
Jerry den Hartog
William H. Winsborough
Sandro Etalle

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



Change in the [illegible] 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:

```
int{Alice:;Alice←*} b;
int{Alice:Bob;Alice←*} 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:

```
int{*lbl} m{*lbl}(label{*lbl} lbl, principal{*lbl} p, int{Alice:p} i)
    where {Alice:Bob} <= lbl, Bob actsfor p
{
    // since Bob actsfor p, {Alice:p} <= {Alice:Bob},
    // and since {Alice:Bob} <= lbl, the label of the argument I
    // is <= {*lbl}. Therefore, we can return i+1.
    return i+1;
}
```

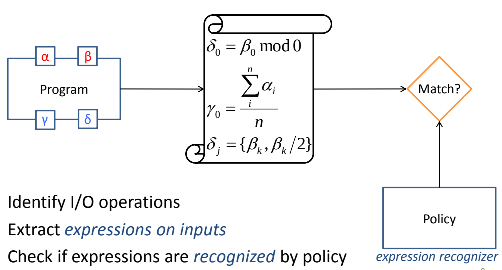
Non security-typed approaches

- *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



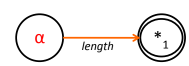
1. Identify I/O operations
2. Extract *expressions on inputs*
3. Check if expressions are *recognized* by policy

Policy Representation

Sets of expressions are represented by graphs

Nodes labels with input channels, constant values or wildcard characters

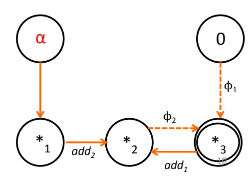
Certain nodes are marked as final nodes



Loop expressions can also be expressed

Represents the expression set:
{0, (0 + α_1), (0 + α_1 + α_2), ...}

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



Code analysis – 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( α );

while (i <= len) do
    val := α ;
    sum := sum + val;
    i := i + 1;

avg := sum / len ;
γ := avg;
```

```
sum := 0; i := 0;
len := length( α );
c := leq(i,len);
while (c) do
    val := α ;
    sum := add(sum,val);
    i := add(i,1);
    c := leq(i,len);
avg := div(sum,len);
γ := avg;
```


Step 2: Static Single Assignment (SSA) Format

```
sum := 0;  i := 0;
len := length( α );
c := leq(i, len) ;
while (c) do
    val := α ;
    sum := add(sum, val);
    i := add(i, 1);
    c := leq(I,len);
avg := div(sum, len) ;
γ := avg;
```

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

```
sum_1 := 0;  i_1 := 0;
len_1 := length( α );
c_1 := leq(i_1, len_1);
while ( c_3 := ϕ_c3(c_1, c_2);
        sum_3 := ϕ_c3(sum_1, sum_2);
        i_3 := ϕ_c3(i_1, i_2);
        c_3) do
    val_1 := α ;
    sum_2 := add(sum_3, val_1);
    i_2 := add(i_3, 1);
    c_2 := leq(i_2, len_1);
avg_1 := div(sum_3, len_1);
γ := avg_1;
```

Step 3: Program Expression Graph

$sum_1 := 0;\ i_1 := 0;$

$len_1 :=$ length(α);

$c_1 := leq(i_1, len_1);$

while ($c_3 := \phi_{c3}(c_1, c_2);$

$sum_3 := \phi_{c3}(sum_1, sum_2);$

$i_3 := \phi_{c3}(i_1, i_2);\ c_3)$ **do**

$val_1 := \alpha\ ;$

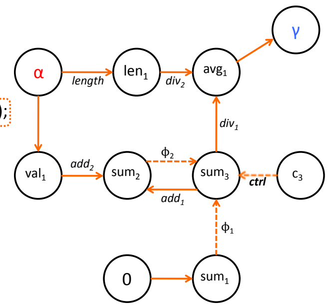
$sum_2 := add(sum_3, val_1);$

$i_2 := add(i_3, 1);$

$c_2 := leq(i_2, len_1);$

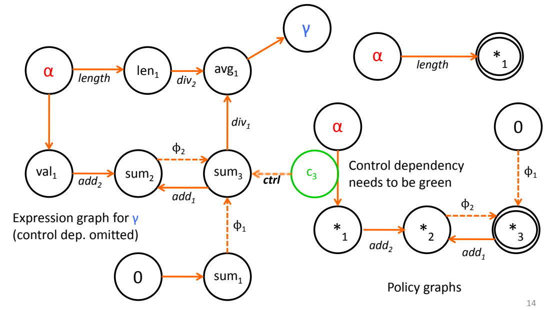
$avg_1 := div(sum_3, len_1);$

$\gamma := avg_1;$

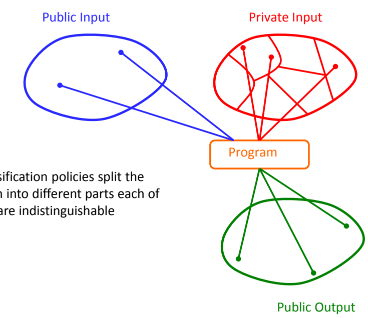


Expression graph for γ

Step 4: Policy Matching



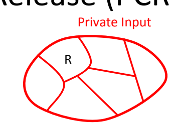
Revisiting the security property



Declassification policies split the domain into different parts each of which are indistinguishable

Policy Controlled Release (PCR)

Revealed Knowledge(R) : Given an environment, π, and a declassification policy, *d*, **R** is the set of all environments for which the value of the declassifiable expressions is the same.



Observed Knowledge(K) : Given an environment, π, and declassification policy, *d*, **K** is the set of all environments which produce the same sequence of visible outputs as π. The observer cannot distinguish between environments in **K**.

Policy Controlled Release Theorem: If the program satisfies PCR, then the knowledge obtained from observing the program (**K**) is bound by the information released by the declassification policy (**R**). That is, $K \supseteq R$.

Did I mention "towards" something?

- Assumptions that can be relaxed:
  - We use a simple toy language, that lacks:
    - More control-flow commands such as **break**, **case**, etc. (easy)
    - User-defined functions, object orientation (currently working on it)
    - Arrays and pointers (SSA solution for these helps a lot)
  - Matching mechanism is currently defined mathematically: algorithms are under way
  - Some operational issues still untreated:
    - Enforcing how many times a given loop runs
    - Algebraic equivalence of expressions denoted by the graphs i.e., *add(x,y) = add(y,x)*

Thank you!

Questions?