Certification of Programs for Secure Information Flow

Dorothy Denning and Peter Denning

What and why … certification?

- An indication whether all possible information flows in the program is in accordance with the information flow policy
- Helps in determining the proof of correctness of the program
- Reduces the need for checking at run-time
- … but does not completely remove the need for run-time checking

More on … Information flow policy

- Information flow policy for a program is a combination of:
  - Security classes
  - Permissible flows between these classes
  - Way to bind program storage objects to these classes
- A security class is just a security ‘rating’. It contains a set of program storage objects.
- A storage object is just anything in a program that hold values ~ variable, array, constant or a file.
- The binding is done (in this case) at the beginning of the program.

Information Flow

- Information is said to flow x $\Longrightarrow$ y if the information in x is transferred so as to derive the value in y.
- The program is said to *specify* a flow x $\Longrightarrow$ y if there is any flow in it that could lead to a transfer of information from x to y.
- Types of flow:
  - Explicit flows happen when the transfer is regardless of the value of x
    - Examples are normal variable assignment, read values from file etc
  - Implicit flow is an indirect flow of information from x to y through an intermediary

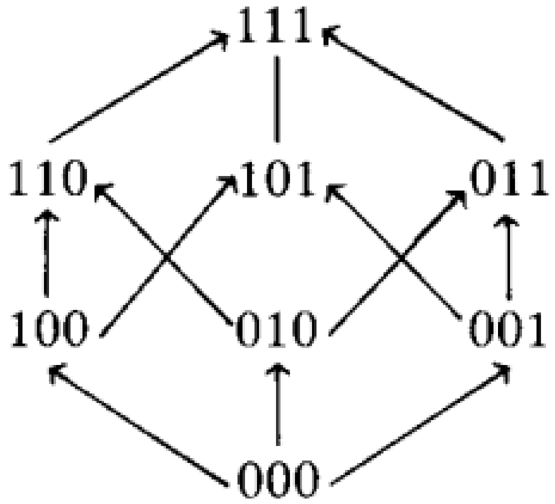
Enter Lattice Theory

- A flow policy is represented by the lattice $\langle S, \rightarrow \rangle$
  - S is the set of security classes (given)
  - $\rightarrow$ represents the set of allowed flows between classes.
- $\underline{x} \rightarrow \underline{y}$ indicates that a flow information from object x to object y is permitted under the given policy.
- $\langle S, \rightarrow \rangle$ is a lattice because it is:
  - Reflexive
  - Transitive
  - Has a Least Upper Bound and Greatest Lower Bound

Lattice Theory in Flow Policy

- Let + and *x* denote the LUB and GLB of a pair of security classes in the flow policy.

$$S = \{000, 001, \ldots, 111\}$$
$$A \rightarrow B \text{ iff } OR(A, B) = B$$
$$A \oplus B = OR(A, B)$$
$$A \otimes B = AND(A, B)$$
$$L = 000,\ H = 111$$



More
Lattice Theory in Flow Policy

- L denotes the greatest lower bound for all the classes
  - All the unnamed constants belong to this class
- H denotes the class that is the greatest lower bound of all the classes.
- In $\underline{x_i} \longrightarrow \underline{y}$ (where i = 1,2 … m), the LUB can be thought as the common security class through which classes x_1, x_2 …. x_m flow through.
- In $\underline{y} \longrightarrow \underline{x_i}$ (where i = 1,2 … n), the GLB can be thought as the common security class through which classes x_1, x_2 …. x_m flow from.
- Help keep track of the origin and destination of flows.

Certification Mechanism

- The paper tries to certify that $x \Rightarrow y$ is specified by p only if $\underline{x} \rightarrow \underline{y}$.
- Determines whether the program specifies *any possible* invalid flows.
- The mechanism is presented in the form of certification semantics.
- Transitive nature of the flow implies that sequence of secure direct flows are secure.
- In particular for a pair of objects, we need only to check their LUB or GLB.

The
CERTIFIED system variable

- The paper keeps track of a boolean variable called CERTIFIED.
- This variable is initially set to true.
- During the analysis of the program, if the mechanism encounters an invalid flow specification, it sets CERTIFIED to false and returns it.
- This is based on the security condition:

$$x \Rightarrow y \text{ is specified by } p \text{ only if } \underline{x} \rightarrow \underline{y}.$$

Object Security Declarations

begin
i,*n*: **integer security class** *L*;
flag: **Boolean security class** *L*;
f1,*f2*: **file security class** *L*;
x,*sum*: **integer security class** *H*;
f3,*f4*: **file security class** *H*;

Sample program and certification

```
begin
  i := 1;
  n := 0;
  sum := 0;
  while i ≤ 100 do
    begin
      input flag from f1;
      output flag to f2;
      input x from f3;
      if flag then
        begin
          n := n + 1;
          sum := sum + x
        end;
      i := i + 1
    end;

  output n, sum, sum/n to f4
 end
end
```

$\underline{1} \rightarrow \underline{i}\ (L \rightarrow L)$

$\underline{0} \rightarrow \underline{n}\ (L \rightarrow L)$

$\underline{0} \rightarrow \underline{sum}\ (L \rightarrow H)$

$\underline{f1} \rightarrow \underline{flag}\ (L \rightarrow L)$

$\underline{flag} \rightarrow \underline{f2}\ (L \rightarrow L)$

$\underline{f3} \rightarrow \underline{x}\ (H \rightarrow H)$

$\underline{n} \oplus \underline{1} \rightarrow \underline{n}\ (L \rightarrow L)$

$\underline{sum} \oplus \underline{x} \rightarrow \underline{sum}\ (H \rightarrow H)$

$\underline{flag} \rightarrow \underline{n} \otimes \underline{sum}\ (L \rightarrow L)$

$\underline{i} \oplus 1 \rightarrow \underline{i}\ (L \rightarrow L)$

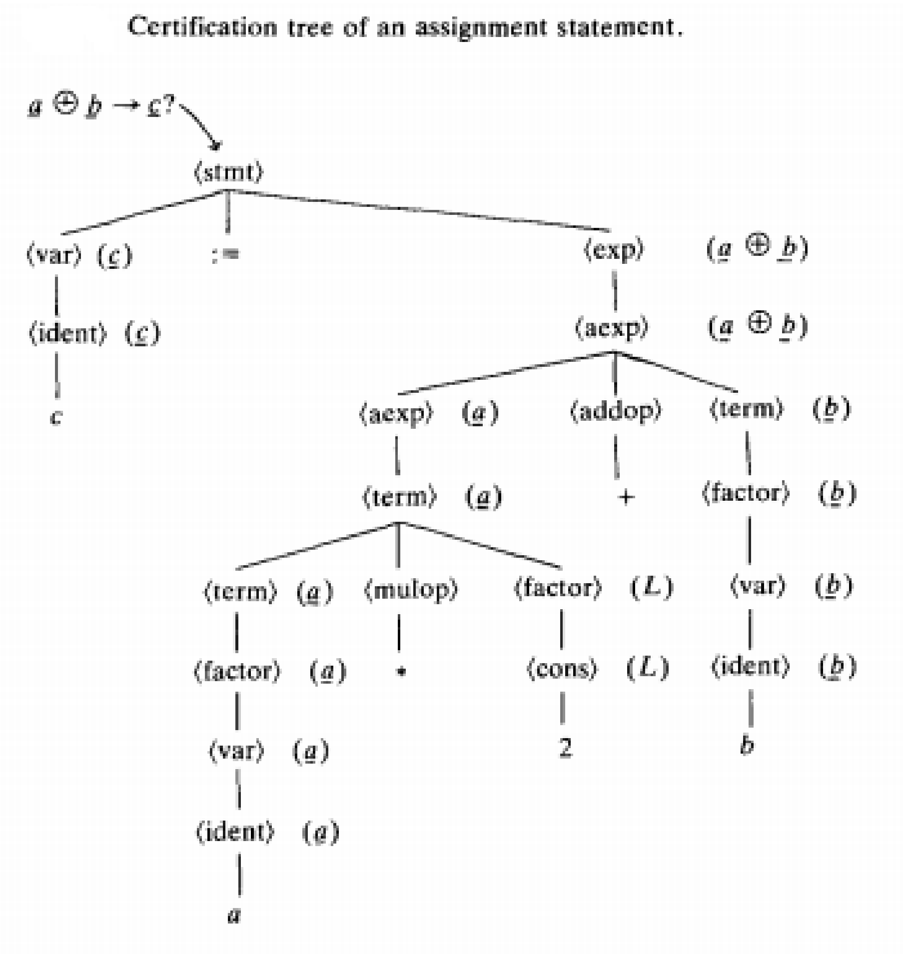
$\underline{i} \oplus \underline{100} \rightarrow \underline{flag} \otimes \underline{f2} \otimes \underline{x} \otimes \underline{n} \otimes \underline{sum} \otimes \underline{i}\ (L \rightarrow L)$

$\underline{n} \oplus \underline{sum} \oplus \underline{sum} \oplus \underline{n} \rightarrow \underline{f4}\ (H \rightarrow H)$

Certification Semantics

Syntax rule	Certification semantics
Declarations	
1 ⟨type⟩ ::= **integer** | **Boolean** | **file**	
2 ⟨idlist⟩ ::= ⟨ident⟩ | ⟨idlist⟩, ⟨ident⟩	
3 ⟨decl⟩ ::= ⟨idlist⟩ : ⟨type⟩ **security class** ⟨security class⟩	for each ⟨ident⟩ in ⟨idlist⟩ associate ⟨security class⟩ with ⟨ident⟩ in the symbol table entry for ⟨ident⟩
4 ⟨declist⟩ ::= ⟨decl⟩ | ⟨declist⟩; ⟨decl⟩	
Expressions	
5 ⟨addop⟩ ::= + | − | ∨	
6 ⟨mulop⟩ ::= * | / | ∧	
7 ⟨relop⟩ ::= < | ≤ | = | ≠ | ≥ | >	
8 ⟨var⟩ ::= ⟨ident⟩	⟨var⟩ ::= ⟨ident⟩
9 ⟨file⟩ ::= ⟨ident⟩	⟨file⟩ := ⟨ident⟩
10 ⟨factor⟩ ::= ⟨var⟩	⟨factor⟩ := ⟨var⟩
11 ⟨factor⟩ ::= ⟨cons⟩	⟨factor⟩ := *L* (the least class)
12 ⟨factor⟩ ::= (⟨exp⟩)	⟨factor⟩ := ⟨exp⟩
13 ⟨factor⟩ ::= ~ $⟨factor⟩_1$	⟨factor⟩ := $⟨factor⟩_1$
14 ⟨term⟩ ::= ⟨factor⟩	⟨term⟩ := ⟨factor⟩
15 ⟨term⟩ ::= $⟨term⟩_1$ ⟨mulop⟩ ⟨factor⟩	⟨term⟩ := $⟨term⟩_1$ ⊕ ⟨factor⟩
16 ⟨aexp⟩ ::= ⟨term⟩	⟨aexp⟩ := ⟨term⟩
17 ⟨aexp⟩ ::= $⟨aexp⟩_1$ ⟨addop⟩ ⟨term⟩	⟨aexp⟩ := $⟨aexp⟩_1$ ⊕ ⟨term⟩
18 ⟨exp⟩ ::= ⟨aexp⟩	⟨exp⟩ := ⟨aexp⟩
19 ⟨exp⟩ ::= $⟨aexp⟩_1$ ⟨relop⟩ $⟨aexp⟩_2$	⟨exp⟩ := $⟨aexp⟩_1$ ⊕ $⟨aexp⟩_2$
Assignment	
20 ⟨stmt⟩ ::= ⟨var⟩ := ⟨exp⟩	⟨stmt⟩ := ⟨var⟩ if not (⟨exp⟩ → ⟨var⟩) then CERTIFIED := **false**
Input	
21 ⟨inlist⟩ ::= ⟨var⟩	⟨inlist⟩ := ⟨var⟩
22 ⟨inlist⟩ ::= $⟨inlist⟩_1$, ⟨var⟩	⟨inlist⟩ := $⟨inlist⟩_1$ ⊗ ⟨var⟩
23 ⟨stmt⟩ ::= **input** ⟨inlist⟩ **from** ⟨file⟩	⟨stmt⟩ := ⟨inlist⟩ if not (⟨file⟩ → ⟨inlist⟩) then CERTIFIED := **false**
Output	
24 ⟨outlist⟩ ::= ⟨exp⟩	⟨outlist⟩ := ⟨exp⟩
25 ⟨outlist⟩ ::= $⟨outlist⟩_1$, ⟨exp⟩	⟨outlist⟩ := $⟨outlist⟩_1$ ⊕ ⟨exp⟩
26 ⟨stmt⟩ ::= **output** ⟨outlist⟩ **to** ⟨file⟩	⟨stmt⟩ := ⟨file⟩ if not (⟨outlist⟩ → ⟨file⟩) then CERTIFIED := **false**
Compound	
27 ⟨stlist⟩ ::= ⟨stmt⟩	⟨stlist⟩ := ⟨stmt⟩
28 ⟨stlist⟩ ::= $⟨stlist⟩_1$; ⟨stmt⟩	⟨stlist⟩ := $⟨stlist⟩_1$ ⊗ ⟨stmt⟩
29 ⟨stmt⟩ ::= **begin** ⟨stlist⟩ **end**	⟨stmt⟩ := ⟨stlist⟩
Selection	
30 ⟨stmt⟩ ::= **if** ⟨exp⟩ **then** $⟨stmt⟩_1$ [**else** $⟨stmt⟩_2$]	⟨stmt⟩ := $⟨stmt⟩_1$ [⊗ $⟨stmt⟩_2$] if not (⟨exp⟩ → ⟨stmt⟩) then CERTIFIED := **false**
Iteration	
31 ⟨stmt⟩ ::= **while** ⟨exp⟩ **do** $⟨stmt⟩_1$	⟨stmt⟩ := $⟨stmt⟩_1$ if not (⟨exp⟩ → ⟨stmt⟩) then CERTIFIED := **false**
Program	
32 ⟨prog⟩ ::= **begin** ⟨declist⟩; ⟨stmt⟩ **end**	if CERTIFIED then certify ⟨prog⟩ else report security violation. (CERTIFIED is initialized to **true** and set to **false** if a violation is detected)

Parse of the syntax tree



Certifying General Control Structures

- The steps for certifying statements like **repeat**, **for** and **case**:
  1. Basic blocks are found out
  2. A Control-flow graph is constructed with transitions
  3. Expression e_i selects the successor for block b_i.
  4. The Immediate Forward Dominator IFD(b_i) is determined for each block b_i.
     - It is the block closest to b amongst all the blocks that lie on every path from b to the exit
  5. Find B_i
     - It is the set of all blocks between b_i and IFD(b_i).
  6. Security class $\underline{B}_i$ for a block b_i is the GLB of all the blocks in B_i.
  7. Check whether $\underline{e}_i \longrightarrow \underline{B}_i$

- We don't really need *goto*, do we?

Certifying Data Structures

- Arrays:
  - Assumption: Security classes of all the elements in the array is the same.
  - When an array reference is processed, classes of subscript and array identifier are joined together.
  - If the array is being assigned to, need to check <array ref> = <ident>

- Records: A record is structure comprising of *m* fields, i.e. till $r.y_m$
  - Copying a record r from file f is secure only if f $\longrightarrow$ *x* r
  - Copying a record r into file f is secure only if f $\longrightarrow$ + r

Procedure calls

- Let *q* be a procedure with input arguments $x_1, x_2 \ldots. x_m$ and output parameters $y_1, y_2 \ldots. y_{n.}$
- **call** $q(x_1, x_2 \ldots. x_m ; y_1, y_2 \ldots. y_n)$ is secure only when:
  - The call to procedure *q* from P is secure.
  - The mappings between the corresponding variables is secure
- If the call occurs inside a series of conditional expressions $e_1, e_2 \ldots. e_k$ and $c_1, c_2 \ldots. c_l$ are all the objects that q specifies, then need to verify:

$$\underline{e}_1 \oplus \cdots \oplus \underline{e}_k \rightarrow \underline{c}_1 \otimes \cdots \otimes \underline{c}_l$$

- Problem with handling arbitrary classes

Exception Handling

- Invalid flows can be caused by traps (exceptions).

```
p:  begin
    i: integer security class L;
    e: Boolean security class L;
    f: file security class L;
    x, sum: integer security class H;
    begin
      sum := 0;
      i := 0;
      e := true;
      while e do
        begin
          sum := sum + x;
          i := i + 1;
          output i to f
        end
    end
    end
```

- Can be avoided by not prohibiting all non-handled traps.

Certifying the certifier – Basis step

- **Theorem**: A program is certified true only if it is secure.
  - Proof through induction
- There are three atomic statements for the base step:
  - <var> := exp (secure based on rule 20)
  - **input** <inlist> **from** <file> (secure based on rule 23)
  - **output** <inlist> **to** <file> (secure based on rule 26)

Certifying the certifier – Induction step

- Induction step: Assuming that the program is certified and secure up to statement J.
- Need to certify for:
  - **begin** <stlist> **end**
  - **if** <exp> **then** <stmt>$_1$ [**else** <stmt>$_2$]
  - **while** <exp> **do** <stmt>$_1$

Limitations

- This paper can’t handle leak of secure information through *covert channels.*
  - Not a big issue, because work by Lipner has shown that guarding information leak through covert channels might be impossible.
- This paper does guard against information leak through legitimate channels and storage channels.

Applications

- Confinement problem:
  - A service is totally confined if user information can never be stored at all.
  - A service is selectively confined if confidential user information can never be stored.
  - This paper can verify varying levels of these confinements.
- State variables
- Data Bank Confidentiality
  - DQL statements can be verified through the LUB of all columns.
  - DML statements cam be verified through the GLB of all the columns.


Questions?