Analysis and Implementation Method of Program to Detect Inappropriate Information Leak

Reishi Yokomori†, Fumiaki Ohata†, Yoshiaki Takata‡, Hiroyuki Seki‡ and Katsuro Inoue†
†Graduate School of Engineering Science, Osaka University,
1-3 Machikaneyama-cho, Toyonaka, Osaka 560-8531, Japan
{yokomori, oohata, inoue}@ics.es.osaka-u.ac.jp
‡Graduate School of Information Science, Nana Institute of Science and Technology
{y-takata, seki} @is.aist-nara.ac.jp

Abstract

For a program which handles secret information, it is very important to prevent inappropriate information leak from the program with secret data. Denning proposed a mechanism to certify a security of program by statically analyzing information flow, and Kuninobu proposed a more practical analysis framework including recursive procedure handling, although no implementation has been yet made.

In this paper, we propose a method of security analysis implementation, and show a security analysis tool implemented for a procedural language. In this work, we extend Kuninobu's algorithm by devising various techniques for analysis of practical programs that have recursive calls and global variables. This method is validated by applying our tools to a simple credit card program, and we confirm that validation of program security is very useful.

1. Introduction

For a program which handles secret information such as a credit card number, it is very important to prevent inappropriate information leak from program data to the outside of the program (system). Data with secret information should not be printed out and invisible to ordinary users. The values seen by the users with non-privileged access right must be non-secret data, and they have to be independent from the secret data. As there would be some dependency between the system output values and secret data to be protected, a chance to guess and decipher the secret data from the output values.

Denning proposed a mechanism to certify that a given program does not violate a security policy for the information of the program [2, 3]. This work is, however, highly theoretical, and there remains a lot of works before applying it to practical programs. Kuninobu proposed a more practical analysis framework including recursive procedure handling [7], although no implementation has been yet made.

We have been studying various program analyses such as slicing [1, 6], and alias analysis [4, 9]. The approaches for these analyses closely relate to the security analysis, and our tools for program analysis would be extended to the security analysis.

Therefore, based on [7], we will propose a method of security analyses implementation, and will show an implemented tool. In this work, we have devised various techniques for analysis of practical programs that have recursive calls and global variables. We have validated this method by applying our tools to a simple credit card program.

In Section 2, we will briefly overview security analysis. In Section 3, we will present an implementation overview for the analysis. In Section 4, we will present the details of the implementation, and present a simple application of the tool. In Section 5, we will conclude our discussions.

2. Security Analysis

To prevent inappropriate information leak to outside system, following access control mechanism, called *Mandatory Access Control*, is generally used:

(1) Each data and storage area in the system has a Security Class(SC), which represents the degree of secret. The SC for data d is represented by $SC(d)$.

(2) Each user and process in the system also has Clearance(CL), which means the degree of access right. The CL for user u is represented by $CL(u)$. Also, CL for process p is denoted by $CL(p)$.

(3) User u (process p) can access data d, if and only if $CL(u) \geq SC(d)$.

type	= standard-type | array-type .
standard-type	= “integer” | “boolean” | “char”
array-type	= “array”[ ] “of” standard-type .
compound	= “begin” statements “end”.
statement	= basic-statement | “if” expression “then” limited-statement “else” statement | “if” expression “then” statement| “while” expression “do” statement .
limited-statement	= basic-statement | “if” expression “then” limited-statement “ else” limited-statement| “while” expression “do” limited-statement.
basic-statement	= assignment | procedure-call | input-statement|output-statement| compound| empty.
assignment	= left-value “:=” expression.
input-statement	= “readln” [“(” a row of variables “)”]|
output-statement	= “writeln” [“(” a row of outputs “)”].
procedure-call	= procedure-name [“(” a row of expressions “)”].
function-call	= function-name[“(” a row of expressions “)”].

Figure 1. A part of BNF of target language

Under this access control mechanism, however, inappropriate information leak may occur in the following way.

(1) User program u, that is $CL(u) \geq SC(d)$, reads a data d.

(2) User program u writes an information d into storage area s, that is $SC(s) < SC(d)$.

(3) User U that is $SC(s) \leq CL(U) < SC(d)$ cannot read data d directly, but U can read information d written in storage area s indirectly. An information leak occurs as this way.

To prevent inappropriate information leak like this, Denning proposed a mechanism to certify that a program does not violate information flow policy [2, 3]. This certification mechanism first sets SC for each variable and input value in the program. After setting SC, the certification mechanism detects inappropriate information leak, based on *Information Flow* which shows control flow and data flow between variables. The certification fails, if there is a statement s such that

- the least upper bound of SC of $used$ variables (variables referred to) at s is higher than SC of $defined$ (variables assigned to) variable at s, or
- the least upper bound of SC of used variables at s is higher than SC of the storage area accessed at s.

In [5], J.Banâtre reexamined this method theoretically, and attempted to generalize the analyze method. However, since these methods do not take account of a recursive call and global variables, applicability is limited. Also these methods analyze only return values and actual parameters in a function (procedure) call statement. Since these methods do not investigate how the return values are calculated, the result is too strict to use practically.

In [7], Kuninobu proposed an information security analysis algorithm that takes account of the analysis of procedure calls including recursive ones. This algorithm defines simultaneous equations from the information flow at each statement. Then, it performs security analysis calculation on these simultaneous equations, and provides SC for each output of the program with respect to given SC for each input. Assume that the formal parameters of the main function of the program are $x_1, \ldots, x_i$, the input files of the program are $infile_1, \ldots, infile_j$, the return value of the main function of the program is y_1, and the output files of the program are $outfile_1, \ldots, outfile_k$. This algorithm calculates $1 + k$ SC's corresponding to the return value and output files from the simultaneous equations, and $i + j$ SC's corresponding to the formal parameters and input files.

In general, the approach of security analysis is classified into following two categories.

- To provide the SC for each output of the program from the SC of each input
- To certify that a given program does not violate the flow policy.

In this paper, we focus the former approach.

3. Implementation of Information Security Analysis Algorithm

Based on [7], we have implemented a prototype of the information security analysis algorithm. In this section, we show an overview of the implementation and a process for the analysis. We will present the details of the tool and an application, in Section 4.

3.1 Overview of Implementation

We have implemented a prototype system for Pascal program. Fig 1 is a part of its BNF.

Our implementation is different from [7]'s algorithm in the following points.

- Global variables handling :

  We will explain it in details later.

- Efficient inter-procedural analysis :

  In [7], when analyzing a procedure, the algorithm needs to hold the analysis results for all combination of parameter's SC. In our implementation, however, we only keep a result for the least upper bound of combination of parameter SC. We consider that this is sufficient for actual use.

- Simplification of SC and input/output file

  To make intuitive understanding easy, we limit SC to $\{high, low\}$, and limit input/output files to only standard input/output files.

3.2 Analysis Procedure

3.2.1 Overview

The security analysis consists of two phases.

Phase 1: setting up initial condition of security analysis

We set up following SC, as the starting point of the security analysis. Other SC's for global variables and local variables are set to low.

input statements: SC's for the input values that are read at input statements

procedure(function) declarations : SC's of the formal parameters of each procedure(function)

Phase 2: information flow analysis

This phase calculates the SC of each output statement of the program with the initial condition and information flow. After this analysis, the algorithm outputs such statements with $high$ SC.

In this paper, we mainly explain Phase 2, especially on the analysis of global variables, intra-procedural analysis, and inter-procedural analysis.

3.2.2 Analysis of Global Variables

Our security analysis algorithm deals with global variables. To deal with global variables in the analysis of each procedure(function), we consider global variables as virtual parameters of each procedure(function), and as virtual return-values of each procedure(function).

This approach works in principle; however, it is not efficient since we have to prepare space for these virtual parameters(return-values) with respect to each procedure even if global variables is not necessarily accessed in the procedure.

Therefore, it is necessary to investigate global variables that used or defined directly or indirectly at each procedure before the security analysis. Global variables that defined directly or indirectly at the procedure P are regarded as the virtual return-values of the procedure P. Global variables that used directly or indirectly at the procedure P are regarded as the virtual parameters of the procedure P. "Directly defined(used) in procedure P" means that it is defined(used) in procedure P. "Indirectly defined(used) in procedure P" means that it is defined(used) in procedures that are called by procedure P.

3.2.3 Intra-procedural Analysis

First, the security analysis algorithm prepares *Security Class Set(SCset)*, that is a collection of SC of each variable at each program point. An element of SCset is a pair (variable, SC). An initial state of SCset consists of elements of global variables that are used in the procedure, local variables, and formal parameters. Each SC is initialized as follows.

local variable: low

formal parameter: the least upper bound of actual parameter's SC of the corresponding procedure-call statements

global variable: the least upper bound of global variable's SC at the imitation of the corresponding procedure-call statements

After setting up SCset, we analyze the procedure from the first statement following the order of execution of the program. Fig 2 shows the calculation of SCset. Let P_{start} be the first statement in the procedure P. This algorithm starts with ALGORITHM(P_{start}, $\emptyset$). SCset directly before analysis of statement s is denoted by $SCset(s)$, and one after the analysis is denoted by $SCset(s')$.

```
(assignment statement)
    cl := { c | x∈Ref(s)∧(x, c)∈SCset } ∪ imp;
    kill := { (x, c) | x∈Def(s)∧(x, c)∈SCset };
    gen := { (x, ⊔_{c∈cl} c) | x∈Def(s)};
  ------------------------------------------------
    SCset := SCset − kill ∪ gen

(input statement)
    kill := { (x, c) | x∈Def(s)∧(x, c)∈SCset };
    gen := SCset_input(s)
        (* { x | x∈SCset_input(s) } = Def(s) *)
  ------------------------------------------------
    SCset := SCset − kill ∪ gen

(output statement)
    cl := { c | x∈Ref(s)∧(x, c)∈SCset } ∪ imp
  ------------------------------------------------
    SC_output := ⊔_{c∈cl} c; SCset := SCset

(if statement)(if E then B_then else B_else)
    cl := { c | x∈Ref(E)∧(x, c)∈SCset } ∪ imp;
    SCset_pre := SCset;
    ALGORITHM(B_then, ⊔_{c∈cl} c); SCset_then := SCset;
    SCset := SCset_pre;
    ALGORITHM(B_else, ⊔_{c∈cl} c); SCset_else := SCset;
  ------------------------------------------------
    SCset := unite(SCset_then, SCset_else)

(while statement)(while E do B)
    SCset_pre := ∅;
    while SCset <> SCset_pre begin
      cl := { c | x∈Ref(E)∧(x, c)∈SCset } ∪ imp;
      ALGORITHM(B, ⊔_{c∈cl} c);
      SCset := unite(SCset, SCset_pre)
    end
  ------------------------------------------------
    SCset := SCset_pre

(block statement)(being B_1; . . . B_n; end)
    ALGORITHM(B_1, ⊔_{c∈cl} c);
    . . .
    ALGORITHM(B_n, ⊔_{c∈cl} c)
  ------------------------------------------------
    SCset := SCset

(procedure call)
  statement s calls a procedure P.
    SCset_next := ∅
    for i := 0 to |s_actuals| begin
      cl := { c | (s_actuals[i], c)∈SCset };
      SCset_next := SCset_next ∪ { (P_formals[i], cl) };
    end;
    foreach x∈Ref'(P) begin
      SCset_next := SCset_next ∪
                                { (x, c) | (x, c)∈SCset }
    end;
    SCset := SCset_next;
    analysis of procedure P;
    kill := ∅;
    for i := 0 to |s_actuals| begin
      kill := kill ∪
        { (P_formals[i], c) | (P_formals[i], c)∈SCset }
    end
  ------------------------------------------------
    SCset := SCset − kill
```

Figure 2. ALGORITHM(**s**, imp)

Ref(s): a collection of variables that used(referred to) at statement s

Def(s): a collection of variables that defined at statement s

Ref'(P): a collection of global variables that used in procedure P

Def'(P): a collection of global variables that defined in procedure P

$\mathbf{s}_{actuals}$**:** a collection of actual parameters in procedure-call statement s

$P_{formals}$**:** a collection of formal parameters in procedure P

SCset: SCset immediately before the analysis of statement s

SCset$_{input}$**:** a collection of elements (variable, SC) that determined at the input statement s as initial condition.

SC$_{output}$**(s):** SC for output statement s

$\sqcup$**:** operator for the least upper bound

unite(A, B): unify SCset A and SCset B.
SC of each variables is the least upper bound of the SC that the variable has in SCset A and the SC that the variable has in SCset B.

Figure 3. Terms and symbols in Figure 2

According to the kinds of statements, the SCset $SCset(s)$ is updated and $SCset(s')$ is defined as shown in Fig 2,

Fig 2 represents

$$\frac{\text{inner transaction of analysis of statement } s}{\text{SCset of directly after analysis of statement } s \text{ or SC of output statement } s}$$

Fig 3 explains terms and symbols that appear in Fig 2.

3.2.4 Inter-procedural Analysis

An usual procedural program consists of more than one procedure, and generally more than one procedure-call statement exist for each procedure. A procedure may call itself recursively, or two or more procedures may call each other.

Therefore, the analysis result of procedure P may be influenced (through parameters, return values, or global variables) the analysis result of procedure P' that calls P. Security analysis algorithm must continue until the analysis results become stabilized. The security analysis algorithm prepares the following elements for the inter-procedural analysis.

analysis list :
The analysis list is a list of analysis order of procedures. This is based on the invocation of procedure-calls of the program. The analysis list is updated during the analysis, and the analysis terminates if the analysis list becomes empty. The details of the algorithm of updating the analysis list is explained in [8].

SCset $\mathcal{C}$ at the beginning of procedure :
Each procedure has one $\mathcal{C}$. Consider a procedure-call statement s that calls procedure P. In analyzing s, the algorithm calculates the least upper bound $\bar{u}$ of $\mathcal{C}$ for P and SCset $\mathcal{C}'$ for s. If $\bar{u}$ is greater than $\mathcal{C}$, $\mathcal{C}$ is redefined with the values of $\bar{u}$, and the analysis goes into P. On the other hand, if $\bar{u}$ is the same SCset as $\mathcal{C}$, analysis of P is known to be unnecessary.

SCset $\mathcal{D}$ at the end of procedure :
Consider at the end of the analysis of procedure P, where the SCset for P was initially D and is updated to $\mathcal{D}'$. The algorithm calculates the least upper bound $\bar{w}$ of SCset $\mathcal{D}$ and SCset $\mathcal{D}'$. If $\bar{w}$ is greater than $\mathcal{D}$, $\mathcal{D}$ is redefined with the values of $\bar{w}$. Also, all procedures calling P are added in the analysis list. On the other hand, if $\bar{w}$ is the same SCset as $\mathcal{D}$, nothing is done.

3.3 Example of Analysis

As an example, consider a function f in Fig 4. At first, we define the following SCset from the global variable, local variable, and parameter.

SCset(8) := $\{$(a, low), (x, low), (y, low)$\}$

Then, we analyze from the first statement(line 8) of the function following the order of execution. We get from Fig 2,

$kill$:= $\{$(y, low)$\}$; gen := $\{$(y, $high$)$\}$

SCset(8') := SCset(8) $-$ $kill \cup gen$
:= $\{$(a, low), (x, low), (y, $high$)$\}$.

```
1:  program sample;
2:  var a : integer;
3:  . . .
4:  function f(x : integer) : integer;
5:  (* assume a, x ← low *)
6:  var y : integer;
7:  begin
8:     readln(y); (* ← high *)
9:     if a > 0 then
10:    begin
11:       a := y + 1;
12:       y := x − 1;
13:    end;
14:
15:    writeln(y);
16:    writeln(x);
17:
18:    f := y;
19:    end
20: . . .
21: end.
```

Figure 4. a sample program

Next, SCset(9) is defined as SCset(8'), and statement 9 is analyzed . In the same way, analysis continues until at the end of the statement 18. We obtain

SCset(18') := $\{$(a, $high$), (x, low), (y, $high$), (f, $high$)$\}$.

After this intra-procedural analysis, we compute the unite Q of SCset(18') and initial $\mathcal{D}$ for f. If Q is greater than $\mathcal{D}$, $\mathcal{D}$ is redefined with the values of Q, and we add all procedures calling f into the analysis list. On the other hand, if Q is the same SCset as $\mathcal{D}$, nothing is done.

4. Security Analysis Tool and Its Application

In this section, we will present the details of the tool which realizes the security analysis algorithm.

We have developed the security analysis tool by adding the security analysis routines to Osaka Slicing System[8], which is a tool for computing program slices.

4.1 Overview

Fig 5 shows an overview of our security analysis tool. The input of this tool is a source code in Pascal. First, this tool analyzes the program syntax and semantics(Fig 5, Step1). Next, the user sets up an initial condition of security analysis(Fig 5, Step2). Then, the tool performs the security analysis (Fig 5, Step3). Finally the tool outputs statements whose SC are $high$ (Fig 5, Step4).

4.2 Example of Application

We consider that this tool is useful for validating the security of a program, in the sense that we can detect in ad-

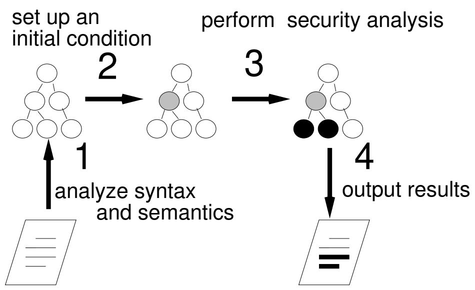

Figure 5. Overview of Security Analysis Tool

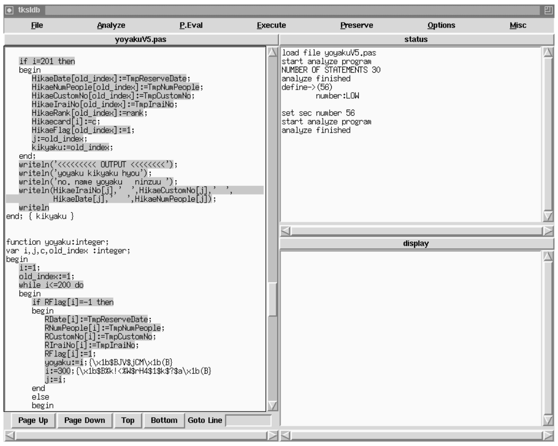

Figure 6. Analysis Result Output of Tool

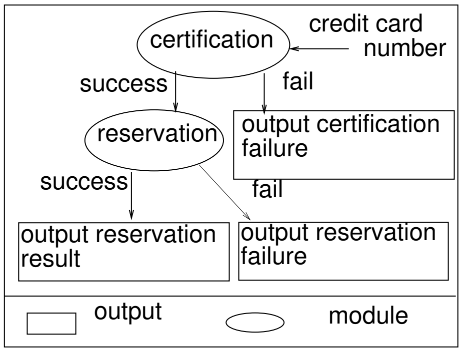

Figure 7. Overview of ticket reservation system

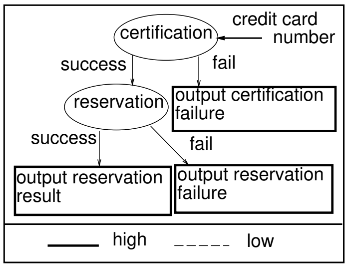

Figure 8. Analysis Result of Original Program

vance unexpected information leak by checking the statements whose SC are high before practical use of the program. By showing possible information leak, we can reorganize program structure, and reduce statements whose SC are high, i.e., the ones with possible information leak.

As an example of such validation of program, we consider a ticket reservation system. The system contains a credit card certification module as shown in Fig 7.

As an initial condition, we assume that only the input statement that reads a credit card number has $high$ SC, and other input statements have low SC. We analyze this system by our tool, and the result is that 35 output statements of 36 output statements have high SC, as shown with highlight lines in Fig 6). These statements with high SC are widely embedded not only in the credit card verification module but in the reservation module as shown in Fig 8. This is because the system handles a reservation after a success of the credit card certification. In this situation, "any possible action in the reservation" implies "a success of credit card certification." We can guess the status of the credit card certification through the reservation result, and this is unexpected security hole.

We have changed the structure of the program, so that we handle the reservation before the certification of a credit card as shown in Fig 9. We have analyzed this system by our tool under the same condition. The result is that only 13 output statements of 36 output statements have high SC. Now all the output statements in the reservation module are low, and all output statements with high SC are in the credit card certification module.

As shown in this example, changing program structure can cause a change of information flow, leading to chance of possible information leak. Thus validation of program security is very important.

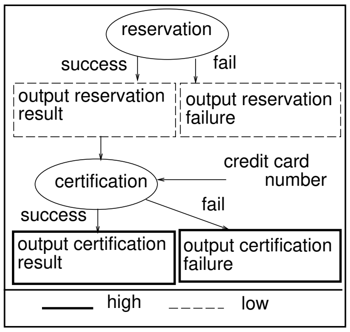

Figure 9. Analysis Result of Reorganized Program

5. Conclusion and Future Work

In this paper, we have proposed an implementation of a security analysis algorithm that was originally proposed in [7], and shown an application of this method. In this implementation, the algorithm originally in [7] has been extended to deal with global variables and procedural calls.

We are planning extensions of our tool as follows:

- Extension of security analysis algorithm to apply object-oriented programs
- Realization of security analysis with a general lattice model

References

[1] D. Atkison and W.G.Griswold. The design of whole-program analysis tools. *in Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany*, pages 16–27, 1996.

[2] D.E.Denning. A lattice model of secure information flow. *Communication of the ACM*, 19(5):236–243, 1976.

[3] D.E.Denning and P.J.Denning. Certification of programs for secure information flow. *Communication of the ACM*, 20(7):504–513, 1977.

[4] F.Ohata and K.Inoue. Alias analysis for object-oriented programs. *Technical report of IPSJ 2000-SE-126*, pages 57–64, 2000. (In Japanese).

[5] J.Banâtre, C.Bryce, and D. Mêtayer. Compile-time detection of information flow in sequential programs. *Proc.3rd ESORICS, LNCS 875*, pages 55–73, 1994.

[6] M.Weiser. Program slicing. *in Proceedings of the 5th International Conference on Software Engineering, San Diego, USA*, pages 439–449, 1981.

[7] S.Kuninobu, Y.Takata, H.Seki, and K.Inoue. An information flow analysis of programs based on a lattice model. *Technical report of IEICE SS 2000-30*, pages 25–21, 2000. (In Japanese).

[8] S.Sato, H.Iida, and K.Inoue. Software debug supporting tool based on program dependence analysis. *Transaction on IPSJ*, 37(4):536–545, 1996. (In Japanese).

[9] P. Tonella, G. Antoniol, R. Fiutem, and E. Merlo. Flow insensitive c++ pointers and polymorphism analysis and its application to slicing. *in Proceedings of the 19th International Conference on Software Engineering*, pages 433–443, 1997.