An Efficient Information Flow Analysis of Recursive Programs based on a Lattice Model of Security Classes

Shigeta Kuninobu[1], Yoshiaki Takata[1], Hiroyuki Seki[1], and Katsuro Inoue[2]

[1] Graduate School of Information Science, Nara Institute of Science and Technology
8916-5 Takayama, Ikoma, Nara, 630-1010, Japan
{shige-ku, y-takata, seki}@is.aist-nara.ac.jp
[2] Graduate School of Engineering Science, Osaka University
1-3 Machikaneyama, Toyonaka, Osaka 560-8531, Japan
inoue@ics.es.osaka-u.ac.jp

Abstract. We present an efficient method for analyzing information flow of a recursive program. In our method, security levels of data can be formalized as an arbitrary finite lattice. We prove the correctness of the proposed algorithm and also show that the algorithm can be executed in cubic time in the size of a program. Furthermore, the algorithm is extended so that operations which hide information of their arguments can be appropriately modeled by using a congruence relation. Experimental results by using a protypic system are also presented.

1 Introduction

In a system used by unspecified people, protecting information from undesirable leaking is essential. One of the ways to protect information from undesirable leaking is an access control technique called *Mandatory Access Control* (MAC). MAC requires that data and users (or processes) be assigned certain security levels represented by a label such as top-secret, confidential and unclassified. A label for a data d is called the security class (SC) of d, denoted as $SC(d)$. A label for a user u is called the clearance of u, denoted as $clear(u)$. In MAC, user u can read data d if and only if $clear(u) \geq SC(d)$. However, it is possible that a program with clearance higher than $SC(d)$ reads data d, creates some data d' from d and writes d' to a storage which a user with clearance lower than $SC(d)$ can read. Hence, an undesirable leaking may occur since data d' may contain some information on data d.

One way to prevent these kinds of information leaks is to conduct a program analysis which statically infers the SC of each output of the program when the SC of each input is given. Several program analyses based on a lattice model of SC have been proposed (see **related works** below); however, some of the program analyses can analyze only relatively simple programs which do not specifically contain a recursive procedure. Also, in some cases, the soundness of the analyses have not been proved.

This paper proposes an algorithm which analyzes information flow of a program containing recursive procedures. The algorithm constructs equations from statements in the program. The equation constructed from a statement represents the information flow caused by the execution of the statement. The algorithm computes the least fixpoint of these equations. We describe the algorithm as an abstract interpretation and

prove the soundness of the algorithm. For a given program *Prog*, the algorithm can be executed in $O(klN)$ time where k is the maximum number of arguments of procedures in *Prog*, l is the number of procedures in *Prog* and N is the total size of *Prog*. Based on the proposed method, a prototypic system has been implemented. Experimental results by using the system are also presented.

In the algorithm proposed in this paper and most of all other existing methods, the SC of the result of a built-in operation θ (e.g., addition) is assumed to be the least upper bound of the SCs of all input arguments of θ. This means that information on each argument may flow into the result of the operation. However, this assumption is not appropriate for some operations such as an aggregate operation and an encryption operation. For these operations, it is practically difficult to recover information on input arguments from the result of the operation. Considering the above discussions, the proposed method is extended so that these operations can be appropriately modeled by using a congruence relation.

The rest of the paper is organized as follows. Section 2 defines the syntax and the operational semantics of a program language which will be the target language of the analysis. In section 3, we formally describe the program analysis algorithm, prove the correctness of the algorithm and show the time complexity of the algorithm. A brief example is also presented in section 3. The method is extended in section 4. Experimental results are briefly presented in section 5.

Related Works [D76] and [DD77] are the pioneering works which proposed a systematic method of analyzing information flow based on a lattice model of security classes. Subsequently, Denning's analysis method has been formalized and extended in a various way by Hoare-style axiomatization [BBM94], by abstract interpretation [O95], and by type theory [VS97,HR98,LR98].

In a type theoretic approach, a type system is defined so that if a given program is well-typed then the program has *noninterference* property such that it does not cause undesirable information flow. [VS97] provides a type system for statically analyzing information flow of a simple procedural program and proves its correctness. The method in [VS97] assumes a program without a recursive procedure while our method can analyze a program which may contain recursive procedures. [HR98] defines a type system for a functional language called Slam calculus to analyze noninterference property. [SV98] showed that their type system in [VS97] is no longer correct in a distributed environment and presented a new type system for a multi-threaded language. How to extend our method to fit a distributed environment is a future study.

A structure of security classes modeled as a finite lattice is usually a simple one such as $\{$top-secret, confidential, unclassified$\}$. [ML98] proposes a finer grained model of security classes called decentralized labels. Based on this model, [M99] proposes a programming language called JFLOW, for which a static type system for information flow analysis as well as a simple but flexible mechanism for dynamically controlling the privileges is provided. However, their type system has not been formally verified.

Recently, control flow analysis of a program which performs dynamic access control such as stack inspection in Java Development kit 1.2 is studied. For example,

[JMT99,NTS01] propose methods of deciding for a given program P and a global security property ψ whether every reachable state of P satisfies ψ.

2 Definitions

2.1 Syntax of Program

In this section, we define the syntax and semantics of a programming language which will be the input language to the proposed algorithm. This language is a simple procedural language similar to C.

A program is a finite set of function definitions. A function definition has the following form:

$$f(x_1, \ldots, x_n) \text{ local } y_1, \ldots, y_m \ \{P_f\}$$

where f is a function name, $x_1, \ldots, x_n$ are formal arguments of f, $y_1, \ldots, y_m$ are local variables and P_f is a function body. The syntax of P_f is given below where c is a constant, x is a local variable or a formal argument, f is a function name defined in the program and θ is a built-in operator such as addition and multiplication. Any object generated by $cseq$ can be P_f.

$$\begin{aligned}
cseq &::= cmd \mid cmd_1; cseq \\
cmd &::= \text{if } exp \text{ then } cseq \text{ else } cseq \text{ fi} \mid \text{return } exp \\
cseq_1 &::= cmd_1 \mid cmd_1; cseq_1 \\
cmd_1 &::= x := exp \mid \text{if } exp \text{ then } cseq_1 \text{ else } cseq_1 \text{ fi} \mid \text{while } exp \text{ do } cseq_1 \text{ od} \\
exp &::= c \mid x \mid f(exp, \ldots, exp) \mid \theta(exp, \ldots, exp)
\end{aligned}$$

Objects derived from exp, cmd or cmd_1, $cseq$ or $cseq_1$ are called an expression, a command, a sequence of commands, respectively. An execution of a program $Prog$ is the evaluation of the function named $main$, which should be defined in $Prog$. Inputs for $Prog$ are actual arguments of $main$ and the output of $Prog$ for these inputs is the return value of $main$.

2.2 Semantics of Program

We assume the following types to define the operational semantics of a program. Let $\times$ denote the cartesian product and $+$ denote the disjoint union.

type val **(values)** We assume for each n-ary built-in operator θ, n-ary operation $\theta_{\mathcal{I}} : val \times \cdots \times val \to val$ is defined. Every value manipulated or created in a program has the same type val.

type $store$ There exist two functions

$lookup : store \times var \to val$

$update : store \times var \times val \to store$

which satisfies:

$lookup(update(\sigma, x, v), y) \quad = \quad$ if $x = y$ then v else $lookup(\sigma, y)$.

For readability, we use the following abbreviations:

$\sigma(x) \equiv lookup(\sigma, x), \quad \sigma[x := v] \equiv update(\sigma, x, v).$

Let $\perp_{store}$ denote the store such that $\perp_{store}(x)$ is undefined for every x.

We define a mapping which provides the semantics of a program. This mapping takes a store and one of an expression, a command and a sequence of commands as arguments and returns a store or a value.

$$\models: (store \to exp \to val) + (store \to cmd \to (store + val)) \\ + (store \to cseq \to (store + val))$$

- $\ulcorner \sigma \models M \Rightarrow v \urcorner$ means that a store σ evaluates an expression M to the value v, that is, if M is evaluated by using σ then v is obtained.
- $\ulcorner \sigma \models C \Rightarrow \sigma' \urcorner$ means that a store σ becomes σ' if a command C is executed.
- $\ulcorner \sigma \models C \Rightarrow v \urcorner$ means that if a command C is executed when the store is σ then the value v is returned. This mapping is defined only when C has the form of 'return M' for some expression M.
- Similar for a sequence of commands.

Below we provide axioms and inference rules which define the semantic mapping, where the following meta-variables are used.

$x, x_1, \ldots, y_1, \ldots : var$ $\quad$ $M, M_1, \ldots : exp$ $\quad$ $C : cmd$ or cmd_1

$P, P_1, P_2 : cseq$ or $cseq_1$ $\quad$ $\sigma, \sigma', \sigma'' : store$

(CONST) $\sigma \models c \Rightarrow c_{\mathcal{I}}$

(VAR) $\sigma \models x \Rightarrow \sigma(x)$

(PRIM) $$\frac{\sigma \models M_i \Rightarrow v_i \ \ (1 \le i \le n)}{\sigma \models \theta(M_1, \ldots, M_n) \Rightarrow \theta_{\mathcal{I}}(v_1, \ldots, v_n)}$$

(CALL) $$\frac{\sigma \models M_i \Rightarrow v_i \ \ (1 \le i \le n) \quad \sigma' \models P_f \Rightarrow v}{\sigma \models f(M_1, \ldots, M_n) \Rightarrow v} \quad \begin{pmatrix} f(x_1, \ldots, x_n) \text{ local } y_1, \ldots, y_m \ \ \{P_f\} \\ \sigma' = \perp_{store}[x_1 := v_1] \cdots [x_n := v_n] \end{pmatrix}$$

(ASSIGN) $$\frac{\sigma \models M \Rightarrow v}{\sigma \models x := M \Rightarrow \sigma[x := v]}$$

(IF1) $$\frac{\sigma \models M \Rightarrow \text{true} \quad \sigma \models P_1 \Rightarrow \sigma' \ (rsp.\ v)}{\sigma \models \text{if } M \text{ then } P_1 \text{ else } P_2 \text{ fi} \Rightarrow \sigma' \ (rsp.\ v)}$$

(IF2) $$\frac{\sigma \models M \Rightarrow \text{false} \quad \sigma \models P_2 \Rightarrow \sigma' \ (rsp.\ v)}{\sigma \models \text{if } M \text{ then } P_1 \text{ else } P_2 \text{ fi} \Rightarrow \sigma' \ (rsp.\ v)}$$

(WHILE1) $$\frac{\sigma \models M \Rightarrow \text{true} \quad \sigma \models P \Rightarrow \sigma' \quad \sigma' \models \text{while } M \text{ do } P \text{ od} \Rightarrow \sigma''}{\sigma \models \text{while } M \text{ do } P \text{ od} \Rightarrow \sigma''}$$

(WHILE2) $$\frac{\sigma \models M \Rightarrow \text{false}}{\sigma \models \text{while } M \text{ do } P \text{ od} \Rightarrow \sigma}$$

(RETURN) $$\frac{\sigma \models M \Rightarrow v}{\sigma \models \text{return } M \Rightarrow v}$$

(CONCAT) $$\frac{\sigma \models C \Rightarrow \sigma' \quad \sigma' \models P \Rightarrow \sigma'' \ (rsp.\ v)}{\sigma \models C; P \Rightarrow \sigma'' \ (rsp.\ v)}$$

3 The Analysis Algorithm

A security class (abbreviated as SC) represents the security level of a value in a program. Let $SCset$ be a finite set of security classes. Also assume that a partial order $\sqsubseteq$ is defined on $SCset$ and $(SCset, \sqsubseteq)$ forms a lattice; let $\bot$ denote the minimum element of $SCset$ and let $a_1 \sqcup a_2$ denote the least upper bound of a_1 and a_2 for $a_1, a_2 \in SCset$. Intuitively, $\tau_1 \sqsubseteq \tau_2$ means that τ_2 is more secure than τ_1; it is legal that a user with clearance τ_2 can access a value with SC τ_1. A simple example of $SCset$ is:

$$SCset = \{low, high\}, \quad low \sqsubseteq high.$$

The purpose of the analysis is to infer (an upper bound of) the SC of the output value when an SC of each input is given. Precisely, the analysis problem for a given program $Prog$ is to infer an SC of the output value of $Prog$ which satisfies the soundness property defined in section 3.3.

We first describe the analysis algorithm in section 3.1. The soundness of the proposed algorithm is proved in section 3.3.

3.1 The Algorithm

To describe the algorithm, we use the following types.

type sc **(security class)** .
type $\underline{store}$ **(SC of store)**

$update : \underline{store} \times var \times sc \to \underline{store}$
$lookup : \underline{store} \times var \to sc$

For $\underline{store}$ type, we use the same abbreviations as for $store$ type. If $\underline{\sigma}$ is an element of type $\underline{store}$, then $\underline{\sigma}(x)$ is the SC of variable x inferred by the algorithm. By extending the partial order $\sqsubseteq$ defined on sc to type $\underline{store}$ as shown below, we can provide a lattice structure to $\underline{store}$:

For $\underline{\sigma}$ and $\underline{\sigma}'$ of type $\underline{store}$, $\underline{\sigma} \sqsubseteq \underline{\sigma}' \Leftrightarrow \forall x \in var.\ \underline{\sigma}(x) \sqsubseteq \underline{\sigma}'(x)$.

The minimum element of $\underline{store}$ is $\underline{\sigma}$ satisfying $\forall x \in var.\ \underline{\sigma}(x) = \bot$. We write this minimum element as $\bot_{\underline{store}}$.

type $\underline{fun}$ **(SC of function)** Similarly to type $\underline{store}$, the following functions are defined.

$$lookup : \underline{fun} \times fname \to (sc \times \cdots \times sc \to sc)$$
$$update : \underline{fun} \times fname \times (sc \times \cdots \times sc \to sc) \to \underline{fun}$$

We use the following abbreviations for $F \in \underline{fun}$, $f \in fname$ and $\psi : sc \times \cdots \times sc \to sc$.

$$F[f] \equiv lookup(F, f)$$
$$F[f := \psi] \equiv update(F, f, \psi)$$

For n-ary function f and SCs $\tau_1, \ldots, \tau_n$, $F[f](\tau_1, \ldots, \tau_n)$ is the SC of the returned value of f inferred by the algorithm when the SC of i-th argument is specified as $\tau_i\ (1 \le i \le n)$. Similarly to type $\underline{store}$, we can provide a lattice structure to type $\underline{fun}$. The minimum element of $\underline{fun}$ is denoted as $\bot_{\underline{fun}}$.

type cv-$\underline{fun}$ (covariant $\underline{fun}$) This type consists of every F of type $\underline{fun}$ which satisfies the next condition:

If $\tau_i \sqsubseteq \tau_i'$ for $1 \leq i \leq n$ then $F[f](\tau_1, \ldots, \tau_n) \sqsubseteq F[f](\tau_1', \ldots, \tau_n')$.

We use the following meta-variables as well as the meta-variables introduced in section 2.2.

$\underline{\sigma}, \underline{\sigma}', \underline{\sigma}'' : \underline{store} \quad F, F_1, F_2 : \underline{fun}$

Below we define a function $\mathcal{A}[\![\cdot]\!]$ which analyzes the information flow. Before defining the analysis function, we explain implicit flow [D76]. Consider the following command.

if $x = 0$ then $y := 0$ else $y := 1$ fi

In this command, the variable x occurs neither in $y := 0$ nor in $y := 1$. However, after executing this command, we can know whether x is 0 or not by checking whether y is 0 or 1. Therefore, we can consider information on the value stored in the variable x flows into the variable y. In general, information may flow from the conditional clause of a "if" command into "then" and "else" clauses and also it may flow from the conditional clause of a "while" command into "do" clause. Such information flow is called *implicit flow*. The function $\mathcal{A}[\![\cdot]\!]$ infers that the SC of implicit flow caused by a command C or a sequence P of commands is the least upper bound of the SCs of the conditional clauses of all the "if" and "while" commands which contain C or P in their scopes. $\mathcal{A}[\![\cdot]\!]$ takes the SC of implicit flow as its fourth argument.

$\mathcal{A} : (exp \times \underline{fun} \times \underline{store} \to sc) + (cmd \times \underline{fun} \times \underline{store} \times sc \to \underline{store})$
$\quad + (cseq \times \underline{fun} \times \underline{store} \times sc \to \underline{store})$

- 「$\mathcal{A}[\![M]\!](F, \underline{\sigma}) = \tau$」 means that, for SCs F of functions and an SC $\underline{\sigma}$ of a store, the SC of an expression M is analyzed as τ.
- 「$\mathcal{A}[\![C]\!](F, \underline{\sigma}, \nu) = \underline{\sigma}'$」 means that, for SCs F of functions, an SC $\underline{\sigma}$ of a store and an SC ν of implicit flow, the SC of the store after executing a command C is analyzed as $\underline{\sigma}'$.
- Similar for a sequence of commands.

The definition of $\mathcal{A}$ is as follows:

(CONST) $\mathcal{A}[\![c]\!](F, \underline{\sigma}) = \bot$
(VAR) $\mathcal{A}[\![x]\!](F, \underline{\sigma}) = \underline{\sigma}(x)$
(PRIM) $\mathcal{A}[\![\theta(M_1, \ldots, M_n)]\!](F, \underline{\sigma}) = \bigsqcup_{1 \leq i \leq n} \mathcal{A}[\![M_i]\!](F, \underline{\sigma})$
(CALL) $\mathcal{A}[\![f(M_1, \ldots, M_n)]\!](F, \underline{\sigma}) = F[f](\mathcal{A}[\![M_1]\!](F, \underline{\sigma}), \ldots, \mathcal{A}[\![M_n]\!](F, \underline{\sigma}))$
(ASSIGN) $\mathcal{A}[\![x := M]\!](F, \underline{\sigma}, \nu) = \underline{\sigma}[x := \mathcal{A}[\![M]\!](F, \underline{\sigma}) \sqcup \nu]$
(IF) $\mathcal{A}[\![\text{if } M \text{ then } P_1 \text{ else } P_2 \text{ fi}]\!](F, \underline{\sigma}, \nu) = \mathcal{A}[\![P_1]\!](F, \underline{\sigma}, \nu \sqcup \tau) \sqcup \mathcal{A}[\![P_2]\!](F, \underline{\sigma}, \nu \sqcup \tau)$
where $\tau = \mathcal{A}[\![M]\!](F, \underline{\sigma})$
(WHILE) $\mathcal{A}[\![\text{while } M \text{ do } P \text{ od}]\!](F, \underline{\sigma}, \nu) = \mathcal{A}[\![P]\!](F, \underline{\sigma}, \nu \sqcup \mathcal{A}[\![M]\!](F, \underline{\sigma})) \sqcup \underline{\sigma}$
(RETURN) Let ret be a fresh variable which contains a return value of a function.
$\mathcal{A}[\![\text{return } M]\!](F, \underline{\sigma}, \nu) = \underline{\sigma}[ret := \mathcal{A}[\![M]\!](F, \underline{\sigma}) \sqcup \nu]$
(CONCAT) $\mathcal{A}[\![C; P]\!](F, \underline{\sigma}, \nu) = \mathcal{A}[\![P]\!](F, \mathcal{A}[\![C]\!](F, \underline{\sigma}, \nu), \nu)$

Define the function $\mathcal{A}[\![\cdot]\!] : program \to \underline{fun} \to \underline{fun}$, which performs 'one-step' analysis of information flow for each function f defined in a given program as follows:

For $Prog \equiv \{f(x_1, \ldots, x_n) \text{ local } y_1, \ldots, y_m \ \{P_f\}, \ldots\}$,

$$\mathcal{A}[\![Prog]\!](F) = F[f := \lambda\tau_1 \ldots \tau_n.(\mathcal{A}[\![P_f]\!](F, \bot_{\underline{store}}[x_1 := \tau_1] \cdots [x_n := \tau_n], \bot)(ret)) \mid f \text{ is an } n\text{-ary function defined in } Prog] \quad (1)$$

For a lattice $(S, \preceq)$ and a function $f : S \to S$, we write the least fix-point of f as $fix(f)$. For a program $Prog$, the function $\mathcal{A}^*[\![Prog]\!]$ which analyzes information flow of $Prog$ is defined as the least fix-point of $\mathcal{A}[\![Prog]\!]$, that is,

$$\mathcal{A}^*[\![Prog]\!] = fix(\lambda F.\mathcal{A}[\![Prog]\!](F)).$$

As will be shown in lemma 1, $\mathcal{A}[\![Prog]\!]$ is a monotonic function on the finite lattice cv-$\underline{fun}$. Therefore,

$$\mathcal{A}^*[\![Prog]\!] = \bigsqcup_{i \geq 0} \mathcal{A}[\![Prog]\!]^i(\bot_{\underline{fun}}) \quad (2)$$

holds [M96] where $f^0(x) = x$, $f^{i+1}(x) = f(f^i(x))$. Hence, $\mathcal{A}^*[\![Prog]\!]$ can be calculated by starting with $\bot_{\underline{fun}}$ and repeatedly applying $\mathcal{A}[\![Prog]\!]$ to the SCs of functions until the SCs of the functions remains unchanged.

3.2 An Example

In this subsection, we show how our analysis algorithm works. The program which we are going to analyze is written below. In this example, we assume $SCset = \{low, high\}$, $low \sqsubseteq high$.

```
main(x) {                          f(x) {
    while x > 0 do                     if x > 0
        y := x + 1; x := y − 4             then return x * f(x − 1)
    od;                                    else return 0
    return f(x)                        fi
}                                  }
```

In order to analyze this program, we continue updating F using the following relation until F does not change any more.

$$F = F[main := \lambda\tau.(\mathcal{A}[\![P_{main}]\!](F, \bot_{\underline{store}}[x := \tau], \bot)(ret))] \\ [f := \lambda\tau.(\mathcal{A}[\![P_f]\!](F, \bot_{\underline{store}}[x := \tau], \bot)(ret))]$$

The table below shows how F changes. The SCs of the i-th column are calculated by using the SCs of the $(i-1)$th column.

	0	1	2	3
$F[main]$	$\lambda\tau.\bot$	$\lambda\tau.\bot$	$\lambda\tau.\tau$	$\lambda\tau.\tau$
$F[f]$	$\lambda\tau.\bot$	$\lambda\tau.\tau$	$\lambda\tau.\tau$	$\lambda\tau.\tau$

From this table, we can know that $\mathcal{A}^*[\![Prog]\!][main](\tau) = \tau$, that is, the SC of the return value of the main function is low when the SC of the actual argument is low and the SC of the return value of the main function could be $high$ when the SC of the actual argument is $high$.

3.3 Soundness of the Algorithm

As mentioned in section 3.1, the analysis algorithm is a function of the following type:

$$\mathcal{A}^*[\![\cdot]\!] : program \to fname \to (sc \times \cdots \times sc \to sc).$$

$\mathcal{A}^*[\![Prog]\!][f](\tau_1, \ldots, \tau_n) = \tau$ means that for an n-ary function f defined in $Prog$ and for SCs $\tau_1, \ldots, \tau_n$ of arguments of f, $\mathcal{A}^*[\![\cdot]\!]$ infers that the SC of f is τ.

Definition 1. An analysis algorithm $\mathcal{A}^*[\![\cdot]\!]$ is **sound** if the following condition is satisfied.

Assume $Prog$ is a program and $main$ is the main function of $Prog$. If

$$\mathcal{A}^*[\![Prog]\!][main](\tau_1, \ldots, \tau_n) = \tau,$$
$$\bot_{store} \models main(v_1, \ldots, v_n) \Rightarrow v, \ \ \bot_{store} \models main(v'_1, \ldots, v'_n) \Rightarrow v',$$
$$\forall i\, (1 \leq i \leq n) : \tau_i \sqsubseteq \tau.\ v_i = v'_i$$

then $v = v'$ holds. □

By the above definition, an analysis algorithm is sound if and only if the following condition is satisfied: assume that the analysis algorithm answers "the SC of the returned value of the main function is τ if the SC of the i-th argument is τ_i." If every actual argument with SC equal to or less than τ remain the same then returned values of the main function also remains the same even if an actual argument with SC higher than or incomparable with τ changes. Intuitively, this means that if the analysis algorithm answers "the SC of the main function is τ," then information contained in each actual argument with SC higher than or incomparable with τ does not flow into the return value of the main function.

The following lemma guarantees the validity of the equation (2).

Lemma 1. *(a)* If F is of type cv-$\underline{fun}$ then $\mathcal{A}[\![Prog]\!](F)$ is also of type cv-$\underline{fun}$.
(b) (*monotonicity*) Assume F_1 and F_2 are of type cv-$\underline{fun}$. If $F_1 \sqsubseteq F_2$ then $\mathcal{A}[\![Prog]\!](F_1) \sqsubseteq \mathcal{A}[\![Prog]\!](F_2)$. □

The next two lemmas are used to show that the algorithm presented in section 3.1 is sound in the sense of definition 1.

Lemma 2. (*property of implicit flow*)

(a) If $\mathcal{A}[\![P]\!](F, \underline{\sigma}, \nu) = \underline{\sigma}'$, $\sigma \models P \Rightarrow \sigma'$ and $\nu \not\sqsubseteq \underline{\sigma}'(y)$ then $\sigma(y) = \sigma'(y)$.
(b) If $\mathcal{A}[\![P]\!](F, \underline{\sigma}, \nu) = \underline{\sigma}'$, $\sigma \models$ while M do P od $\Rightarrow \sigma'$ and $\nu \not\sqsubseteq \underline{\sigma}'(y)$ then $\sigma(y) = \sigma'(y)$. □

Lemma 3. (*noninterference property*) Let $F = \mathcal{A}^*[\![Prog]\!]$.

(a) If $\mathcal{A}[\![M]\!](F, \underline{\sigma}) = \tau$, $\sigma_1 \models M \Rightarrow v_1$, $\sigma_2 \models M \Rightarrow v_2$ and $\forall x : \underline{\sigma}(x) \sqsubseteq \tau.\ \sigma_1(x) = \sigma_2(x)$, then $v_1 = v_2$.

(b) If $\mathcal{A}[\![P]\!](F, \underline{\sigma}, \nu) = \underline{\sigma}'$, $\sigma_1 \models P \Rightarrow \sigma_1'$, $\sigma_2 \models P \Rightarrow \sigma_2'$, $\underline{\sigma}'(y) = \tau$ and $\forall x : \underline{\sigma}(x) \sqsubseteq \tau$. $\sigma_1(x) = \sigma_2(x)$, then $\sigma_1'(y) = \sigma_2'(y)$.
(c) If $\mathcal{A}[\![P]\!](F, \underline{\sigma}, \nu) = \underline{\sigma}'$, $\sigma_1 \models P \Rightarrow v_1$, $\sigma_2 \models P \Rightarrow v_2$ and $\forall x : \underline{\sigma}(x) \sqsubseteq \underline{\sigma}'(ret)$. $\sigma_1(x) = \sigma_2(x)$, then $v_1 = v_2$.

(Proof Sketch) By using Lemmas 1 and 2, the lemma is proved by induction on the application number of inference rules for $\mathcal{A}[\![\cdot]\!]$. □

Theorem 1. The algorithm $\mathcal{A}^*[\![\cdot]\!]$ is sound.

(Proof) By lemma 3*(c)*. □

3.4 Time Complexity

In this subsection, the time complexity of the algorithm $\mathcal{A}^*[\![\cdot]\!]$ presented in section 3.1 is examined. Let *Prog* be an input program and let k, l and N be the maximum number of arguments of each function in *Prog*, the number of functions in *Prog* and the total size of *Prog*, respectively. Since the only operations which appear in the algorithm are $\bot$ and $\sqcup$, for each n-ary function f in *Prog*, $\mathcal{A}[\![Prog]\!](F)[f]$ can be written as

$$\mathcal{A}[\![Prog]\!](F)[f](\tau_1, \ldots, \tau_n) = \tau_{i_1} \sqcup \cdots \sqcup \tau_{i_m}$$

where τ_i $(1 \leq i \leq n)$ is an arbitrary SC and $\{i_1, \ldots, i_m\} \subseteq \{1, \ldots, n\}$. The worst case is that for each execution of $\mathcal{A}[\![Prog]\!](F)$, only one τ_j is added to $\mathcal{A}[\![Prog]\!](F)[f_1](\tau_1, \ldots, \tau_n)$ for only one function f_1 and $\mathcal{A}[\![Prog]\!](F)[f](\tau_1, \ldots, \tau_n)$ remains unchanged for every function f other than f_1. For example, $\mathcal{A}[\![Prog]\!](F)[f_1](\tau_1, \tau_2, \tau_3) = \tau_1$ becomes $\mathcal{A}[\![Prog]\!](\mathcal{A}[\![Prog]\!](F))[f_1](\tau_1, \tau_2, \tau_3) = \tau_1 \sqcup \tau_3$ while $\mathcal{A}[\![Prog]\!](\mathcal{A}[\![Prog]\!](F))[f](\tau_1, \ldots, \tau_n) = \mathcal{A}[\![Prog]\!](F)[f](\tau_1, \ldots, \tau_n)$ for every function f other than f_1. Thus, the maximum number of iterations of $\mathcal{A}[\![Prog]\!]$ is kl. On the other hand, it is not difficult to see that one iteration of $\mathcal{A}[\![Prog]\!]$ takes $O(N)$ time. Hence, we obtain the following theorem:

Theorem 2. Let *Prog* be a program. The algorithm $\mathcal{A}^*[\![Prog]\!]$ can be executed in $O(klN)$ time where k is the maximum number of arguments of each function in *Prog*, l is the number of functions in *Prog* and N is the total size of *Prog*, respectively. □

4 An Extended Model

The algorithm $\mathcal{A}$ in the previous section has been defined for any built-in operator θ as:

(PRIM) $\mathcal{A}[\![\theta(M_1, \ldots, M_n)]\!](F, \underline{\sigma}) = \bigsqcup_{1 \leq i \leq n} \mathcal{A}[\![M_i]\!](F, \underline{\sigma})$.

This means that we assume information contained in each argument may flow into the result of the operation $\theta_{\mathcal{I}}$. However, this assumption is too conservative for a certain operation. For example, if an operation $\theta_{\mathcal{I}}$ is defined as $\theta_{\mathcal{I}}(x, y) = x$, then it is clear that information in the second argument does not flow into the result of the operation. Another example is an encryption. Assume that for a plain text d and an encryption

key k, the result of the operation $E_{\mathcal{I}}(d,k)$ is the cipher text of d with key k. We may consider that the SC of $E(x,y)$ is *low* even if the SCs of x and y are both *high*.

To express the above mentioned properties of particular built-in operations, we generalize the above definition as:

(PRIM) $\mathcal{A}[\![\theta(M_1,\ldots,M_n)]\!](F,\underline{\sigma}) = \mathcal{B}[\![\theta]\!](\mathcal{A}[\![M_1]\!](F,\underline{\sigma}),\ldots,\mathcal{A}[\![M_n]\!](F,\underline{\sigma}))$,

where $\mathcal{B}[\![\theta]\!]$ is an arbitrary monotonic total function on sc:

$$\mathcal{B}[\![\theta]\!] : sc \times \cdots \times sc \rightarrow sc.$$

In particular, $\mathcal{B}[\![\theta]\!](\tau_1,\ldots,\tau_n) = \bigsqcup_{1\leq i\leq n}\tau_i$ for the original definition of $\mathcal{A}$.

However, the generalized algorithm is no longer sound in the sense of definition 1. Suppose that we define $\mathcal{B}[\![E]\!](\tau_1,\tau_2) = low$, and consider a program

$$Prog = \{main(x,y)\ \{\ \texttt{return}\ E(x,y)\ \}\ \}.$$

$\mathcal{A}^*[\![Prog]\!][main](high,high) = low$ holds while for distinct plain texts d_1, d_2 and a key k, $E_{\mathcal{I}}(d_1,k) \neq E_{\mathcal{I}}(d_2,k)$. Hence $\mathcal{A}^*[\![\cdot]\!]$ is not sound. Intuitively, the fact that the SC of expression $E(x,y)$ is inferred as *low* means that we cannot recover information contained in the arguments x,y from the result of the encryption. In other words, $E_{\mathcal{I}}(d_1,k)$ and $E_{\mathcal{I}}(d_2,k)$ are indistinguishable with respect to the information in the arguments. To express this indistinguishability, we introduce the following notions.

A relation R on type *val* is called a congruence relation if R is an equivalence relation which satisfies:

> for each n-ary built-in operator θ, if $c_i\ R\ c_i'$ for $1 \leq i \leq n$
> then $\theta_{\mathcal{I}}(c_1,\ldots,c_n)\ R\ \theta_{\mathcal{I}}(c_1',\ldots,c_n')$.

In the following, we assume that a particular congruence relation $\sim$ is given. For v, v' of type *val*, if $v \sim v'$ then we say that v and v' are indistinguishable. By the definition, if v_i and v_i' for $1 \leq i \leq n$ are indistinguishable then for any built-in operator θ, $\theta_{\mathcal{I}}(c_1,\ldots,c_n)$ and $\theta_{\mathcal{I}}(c_1',\ldots,c_n')$ are also indistinguishable. This implies that once v and v' become indistinguishable, we cannot obtain any information to distinguish v and v' through any operations.

Next, we require $\mathcal{B}[\![\cdot]\!]$ to satisfy the following condition.

Condition 4 Assume $\mathcal{B}[\![\theta]\!](\tau_1,\ldots,\tau_n) = \tau$ for an n-ary built-in operator θ. Let c_i, c_i' be of type *val* ($1 \leq i \leq n$). If $c_j \sim c_j'$ for each j ($1 \leq j \leq n$) such that $\tau_j \sqsubseteq \tau$, then $\theta_{\mathcal{I}}(c_1,\ldots,c_n) \sim \theta_{\mathcal{I}}(c_1',\ldots,c_n')$. □

The above condition states that:

> Let $\mathcal{B}[\![\theta]\!](\tau_1,\ldots,\tau_n) = \tau$. Assume that arguments of θ are changed from $c_1,\ldots,c_n$ to $c_1',\ldots,c_n'$. As long as c_j and c_j' are indistinguishable for each argument position j such that $\tau_j \sqsubseteq \tau$, $\theta_{\mathcal{I}}(c_1,\ldots,c_n)$ and $\theta_{\mathcal{I}}(c_1',\ldots,c_n')$ remain indistinguishable.

Example 1 (nonstrict function). Assume that $\theta_{\mathcal{I}}(x,y) = x$ and $\mathcal{B}[\![\theta]\!](\tau_1,\tau_2) = \tau_1$. For any values c_1, c_1', c_2 and c_2', $c_1 \sim c_1'$ implies $\theta_{\mathcal{I}}(c_1,c_2) = c_1 \sim c_1' = \theta_{\mathcal{I}}(c_1',c_2')$. Hence, condition 4 is met for any congruence relation $\sim$. □

Example 2 (declassification). Let *mk-rpt* be an operator which takes a patient record and produces a doctor's report. Assume that no information in the argument of *mk-rpt* flows into the result of the operator. In this case, we can define $\mathcal{B}[\![\textit{mk-rpt}]\!](high) = low$ with $\textit{low} \sqsubseteq \textit{high}$. Condition 4 requires that for any patient records c, c', $\textit{mk-rpt}_{\mathcal{I}}(c) \sim \textit{mk-rpt}_{\mathcal{I}}(c')$. Intuitively, this means that we cannot discover information on a particular patient's record by reading a doctor's report. □

Example 3 (encryption). Let *E* be an encryption function which takes a plain text and an encryption key as arguments. Assume that no information in the plain text can be discovered by manipulating the encrypted text. In this case, we can define $\mathcal{B}[\![E]\!](high, high) = low$. Condition 4 requires that for any plain texts d, d' and keys k, k', $E_{\mathcal{I}}(d, k) \sim E_{\mathcal{I}}(d', k')$. □

Now we can define the soundness by using the notion of indistinguishability as follows:

Definition 2 (generalized soundness). Let $\sim$ be a congruence relation. We say that an algorithm $\mathcal{A}^*[\![\cdot]\!]$ is sound (with respect to $\sim$) if the following condition holds:

If $\mathcal{A}^*[\![Prog]\!][main](\tau_1, \ldots, \tau_n) = \tau$,
$\perp_{store} \models main(v_1, \ldots, v_n) \Rightarrow v$, $\perp_{store} \models main(v'_1, \ldots, v'_n) \Rightarrow v'$, and
$\forall i\, (1 \leq i \leq n) : \tau_i \sqsubseteq \tau.\, v_i \sim v'_i$
then $v \sim v'$ holds. □

It is not difficult to prove the following theorem in a similar way to the proof of theorem 1.

Theorem 3. If condition 4 is satisfied, then the generalized algorithm $\mathcal{A}^*[\![\cdot]\!]$ is sound in the sense of definition 2. □

5 Conclusion

In this paper, we have proposed an algorithm which can statically analyze the information flow of a procedural program containing recursive definitions. It has been shown that the algorithm is sound and that the algorithm can be executed in polynomial time in the size of an input program. In [Y01], the proposed algorithm is extended to be able to analyze a program which may contain global variables and a prototypic analysis system has been implemented. Table 1 shows the execution time to analyze sample programs by the implemented system.

Table 1. Analysis time

Program	Number of lines	Average analysis time (sec)
Ticket reservation system	419	0.050
Sorting algorithm	825	0.130
A program library	2471	2.270

Extending the proposed method so that we can analyze a program which has pointers and/or object-oriented features is a future study.

Acknowledgments

The authors sincerely thank Fumiaki Ohata and Reishi Yokomori of Osaka University for their valuable comments and discussions.

References

[BBM94] J. Banâtre, C. Bryce and D. Le Métayer: Compile-time detection of information flow in sequential programs, 3rd ESORICS, LNCS 875, 55–73, 1994.

[D76] D. E. Denning: A lattice model of secure information flow, Communications of the ACM, 19(5), 236–243, 1976.

[DD77] D. E. Denning and P. J. Denning: Certification of programs for secure information flow, Communications of the ACM, 20(7), 504–513, 1977.

[HR98] N. Heintze and J. G. Riecke: The SLam calculus: Programming with secrecy and integrity, 25th ACM Symp. on Principles of Programming Languages, 365–377, 1998.

[JMT99] T. Jensen, D. Le Métayer and T. Thorn: Verification of control flow based security properties, 1999 IEEE Symp. on Security and Privacy, 89–103, 1999.

[LR98] X. Leroy and F. Rouaix: Security properties of typed appletes, 25th ACM Symp. on Principles of Programming Languages, 391–403, 1998.

[M96] J. Mitchell: *Foundations of Programming Languages*, The MIT Press, 1996.

[M99] A. C. Myers: JFLOW: Practical mostly-static information flow control, 26th ACM Symp. on Principles of Progmming Languages, 228–241, 1999.

[ML98] A. C. Myers and B. Liskov: Complete, safe information flow with decentralized labels, 1998 IEEE Symp. on Security and Privacy, 186–197.

[NTS01] N. Nitta, Y. Takata and H. Seki: Security verification of programs with stack inspection, 6th ACM Symp. on Access Control Models and Technologies, 31–40, 2001.

[O95] P. Ørbæk: Can you trust your data? TAPSOFT '95, LNCS 915, 575–589.

[SV98] G. Smith and D. Volpano: Secure information flow in a muti-threaded imperative language, 25th ACM Symp. on Principles of Programming Languages, 355–364, 1998.

[VS97] D. Volpano and G. Smith: A type-based approach to program security, TAPSOFT '97, LNCS 1214, 607–621.

[Y01] R. Yokomori: Security analysis algorithm for object-oriented programs, Master's Thesis, Osaka University, 2001.