Automatic Generation of History-Based Access Control from Information Flow Specification

Yoshiaki Takata[1] and Hiroyuki Seki[2]

[1] Kochi University of Technology, Tosayamada, Kochi 782-8502, Japan,
takata.yoshiaki@kochi-tech.ac.jp,
[2] Nara Institute of Science and Technology, Ikoma, Nara 630-0192, Japan,
seki@is.naist.jp

Abstract. This paper proposes a method for automatically inserting check statements for access control into a given recursive program according to a given security specification. A history-based access control (HBAC) is assumed as the access control model. A security specification is given in terms of information flow. We say that a program π satisfies a specification Γ if π is type-safe when we consider each security class in Γ as a type. We first define the problem as the one to insert check statements into a given program π to obtain a program π' that is type-safe for a given specification Γ. This type system is sound in the sense that if a program π is type-safe for a specification Γ, then π has noninterference property for Γ. Next, the problem is shown to be co-NP-hard and we propose an algorithm for solving the problem. The paper also reports experimental results based on our implemented system and shows that the proposed method can work within reasonable time.

1 Introduction

A language-based access control is a promising approach to preventing untrusted modules from accessing confidential data. Stack inspection provided by the Java virtual machine (JVM) and the common language runtime (CLR) is a typical successful example. In a language-based access control, a statement for runtime permission check such as `checkPermission` of JVM (abbreviated as check statement) is placed just before a statement accessing confidential data. Permissions to be checked at each check statement are usually set manually; however, an inappropriate setting of permissions causes either security flaw or unnecessary abortion of execution. Therefore a systematic method for generating appropriate check statements is desired.

This paper assumes a (shallow) history-based access control (HBAC) [1, 12] as an access control model and proposes a method for automatically inserting check statements into a given recursive program according to a given security specification. In this paper, a security specification is given in terms of information flow [8, 9]: a specification is an assignment of a security class (e.g. top_secret, confidential, and unclassified) to each input and output variable (or channel) of a program. The set of security classes is assumed to be a finite semilattice. Since

one of the main purposes of access control (especially, mandatory access control) is to prevent undesirable information leak by deciding which user (or process) can have access to which resource, it is natural to give a security specification using the concept of information flow.

We say that a program π satisfies a specification Γ if π is type-safe when we consider each security class in Γ as a type. This type system is sound in the sense that if a program π is type-safe for a specification Γ, then π has noninterference property for Γ (Theorem 1). Noninterference property is a widely-used semantic (but undecidable in general) criterion for the confidentiality. Intuitively, a program π has noninterference property for a specification Γ if the content of a variable v in π is not affected by the content of any variable in π whose security class specified by Γ is higher than v.

Next, we define the problem as follows: for a given program π including zero or more check statements with permissions to be checked unspecified and a specification Γ, specify permissions to be checked at each check statements in π so that the resultant program is type-safe for Γ. This definition does not lose generality since check statements are usually placed just before access statements and it can be easily done automatically. Then, the problem is shown to be co-NP-hard (Theorem 2) and we propose an algorithm for solving the problem using a model checking method for pushdown systems (PDS). The idea of the proposed method is simple. If we find an execution trace that violates a specification by analyzing the PDS derived from an HBAC program, then we add appropriate permissions to be checked at a check statement nearest to the undesirable access to remove this execution trace. However, adding new permissions may introduce a new violation of the specification (known as a covert channel). This covert channel can be avoided by carefully designed fixpoint operation given in Section 5. The paper also reports experimental results based on our implemented system and shows that the proposed method can generate check statements within reasonable time.

Related Work Static analysis of programs using stack inspection (or stack-based access control, SBAC) has been widely studied, e.g. for safety verification [15, 20], LTL model checking [11], analysis of current permission sets [6, 16]. Pottier et al. [23] and Besson et al. [7] proposed type systems such that type safety implies no violation against runtime access control in an SBAC program. Also safety verification method and its implementation for HBAC programs were reported [26]. Information flow analysis has a long history stemming from [8, 9] and has been extensively studied for recursive programs using Hoare logic [2], abstract interpretation [21], and type systems [14, 17, 25]. The combination of static analysis and dynamic control of information flow has also been studied for usual recursive programs [19, 18, 13]. Information flow analysis has been extended to SBAC [3, 5] and HBAC [4]. The latter showed interesting phenomena that check statements themselves may cause implicit information flow. The work of [22] regarded dynamic permissions as a security class, considered that check statements represent a security specification, and proposed a dynamic control

mechanism of information flow. To the authors' knowledge, however, this paper is the first one to deal with the problem of automatic generation of access control statements from a specification of information flow.

The rest of this paper is organized as follows. In Section 2, the syntax and operational semantics of an HBAC program is defined. In Section 3, we define a security specification as well as the notion of type-safety by deriving a pushdown system (PDS) from a given program π and a specification Γ. This PDS in effect constitutes a type system for π under Γ. Also it is shown that type-safety implies noninterference property. In Section 4, the problem is defined and the problem is shown to be co-NP-hard. In Section 5, an algorithm for solving the problem by reachability analysis of the PDS is given, followed by the experimental results.

2 Input Program

2.1 Syntax

A program consists of a set *Func* of functions, a set *In* of input channels, and a set *Out* of output channels. The body of each function is an element of *cseq* defined by the following BNF specification.

$$
\begin{aligned}
\mathit{cseq} &::= \mathit{cmd} \mid \mathit{cmd}\,\texttt{;}\,\mathit{cseq} \\
\mathit{cmd} &::= \texttt{if}\ \mathit{exp}\ \texttt{then}\ \mathit{cseq}\ \texttt{fi} \\
&\quad\mid \texttt{if}\ \mathit{exp}\ \texttt{then}\ \mathit{cseq}\ \texttt{else}\ \mathit{cseq}\ \texttt{fi} \\
&\quad\mid \mathit{out}\ \texttt{:=}\ x \quad\mid\quad x\ \texttt{:=}\ \mathit{in} \quad\mid\quad x\ \texttt{:=}\ \mathit{exp} \\
&\quad\mid x\ \texttt{:=}\ \mathit{func}\texttt{(}\mathit{exp}\texttt{,}\ldots\texttt{,}\mathit{exp}\texttt{)} \\
&\quad\mid \texttt{check[}P\texttt{]} \\
\mathit{exp} &::= c \mid x \mid \theta\texttt{(}\mathit{exp}\texttt{,}\ldots\texttt{,}\mathit{exp}\texttt{)}
\end{aligned}
$$

In the above BNF specification, *out*, *in*, x, c, P, and θ represent an output channel, an input channel, a variable, a constant, a subset of permissions, and a built-in operator, respectively. The statements in the right-hand side of *cmd* represent an `if` statement, an `if-else` statement, an output statement, an input statement, an assignment statement (without function call), a function call statement, and a permission-check statement (check statement for short), respectively. For a check statement `check[`P`]`, P is called the argument of this check statement. For simplicity, no iteration statement is defined; iteration must be specified by recursive function call. The return value of a function f is stored in a special variable named ret_f.

A program interacts with its environment only through the input and output channels; the starting function (main function) has no arguments or return value.

2.2 Access Control Mechanism

Below we informally describe the HBAC [1, 26], the access control mechanism assumed in this paper. (Formal definition is given in Section 2.3.) HBAC is

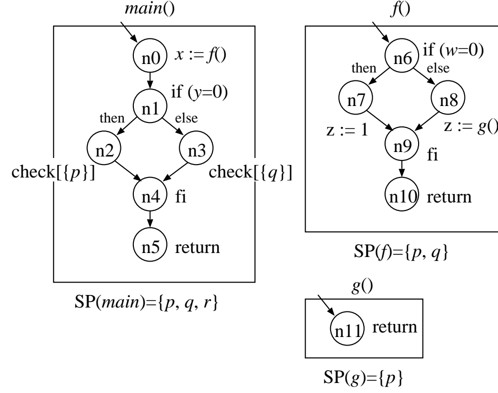

Fig. 1. Program π_0.

proposed to resolve the weakness of the stack inspection such that it ignores the execution history of functions of which execution is finished [1, 4, 26]. (See [1] for the design principles of HBAC.)

A subset of permissions is assigned to each function f before runtime. We call the subset the *static permission set* of f, denoted as $SP(f)$. The runtime system controls another subset of permissions called the *current permission set.* When a program starts, the current permission set is initialized as the static permission set of the starting function. The current permission set is updated when a function is called. The updated current permission set is the intersection of the old current permission set and the static permission set of the callee function; that is, every permission not assigned to the callee function is removed from the current permission set.

When the control reaches a check statement `check[`P`]`, the execution is continued if the current permission set includes all permissions in P, and the execution is aborted otherwise.

Hereafter we assume that a program is represented as a control flow graph (such as Figure 1) in which conditional branches are well-nested. Each node of the graph represents a program point, and each directed edge represents a control flow with in a function.

Example 1. Consider a sample program π_0 in Figure 1. The transition of the call stack and the current permission set of π_0 is depicted in Figure 2.

When function g is called at program point n_8 in function f, permission q is removed from the current permission set. After that, when the control reaches n_3 in function *main*, the execution is aborted because q is not in the current permission set. Abortion at a check statement does not occur if function g is never called or the control never reaches n_3.

Since the current permission set is uniquely determined by the set of functions that have invoked so far, we assume that $PRM = \{\, p_f \mid f \in \mathit{Func}\,\}$ and $SP(f) =$

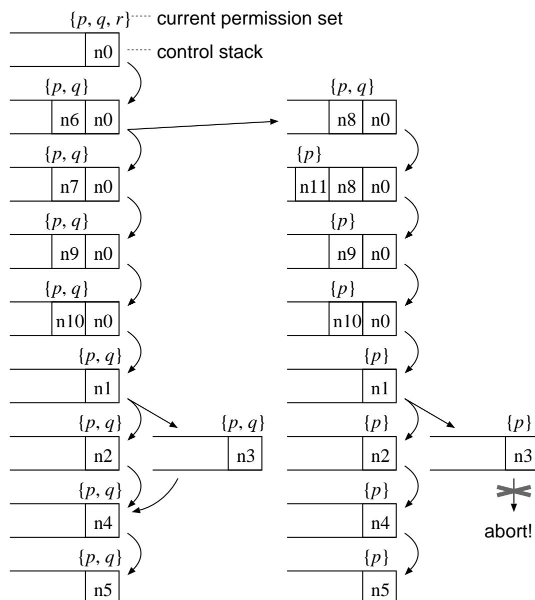

Fig. 2. Transition of the call stack and current permission set of π_0.

$PRM \setminus \{p_f\}$ for each $f \in Func$ without loss of generality[3]. That is, p_f remains in the current permission set if and only if f has never been invoked.

2.3 Operational Semantics

For a program π, we define a transition system M_π that represents the behavior of π. A configuration of M_π is a pair (σ, ξ), in which σ is a state of the input and output channels and ξ is a stack (i.e. a sequence of stack frames). A state of an output channel is a finite sequence of values that have been output so far. A state of an input channel is an infinite sequence of values that are going to be read out. A stack is a finite sequence of stack frames, and a stack frame is a triple $\langle n, \mu, C\rangle$ in which n is a program point, μ is a state of local variables (including formal parameters and the return value variable ret_f), and C is the current permission set. The leftmost stack frame of a stack is the stack top.

The transition relation $\Rrightarrow$ of M_π is the smallest relation that satisfies the following inference rules. As mentioned before, we assume that a program is represented as a control flow graph, and we write $n \rightarrow n'$ if there exists a control

[3] In the original definition of HBAC programs [1, 26], one can specify a *grant set* and an *accept set* for each function call statement, which enable more flexible control of the current permission set. We omit these parameters to simplify the problem. With grant and accept sets, the current permission set is not necessarily determined uniquely by the set of already invoked functions.

flow from a program point n to n'. The program statement of a program point n is denoted as $\lambda(n)$. For the end point n of a function, $\lambda(n) = \texttt{return}$. The formal parameter list of a function f is denoted as $\mathit{param}(f)$, and the initial program point of f is denoted as $\mathit{IT}(f)$. The concatenation of two sequences ξ_1 and ξ_2 is denoted as $\xi_1 : \xi_2$. The state in which every variable is undefined is denoted as $\bot$. For a state μ, $\mu[x \mapsto v]$ is the state that maps x to v and y to $\mu(y)$ for every $y \neq x$. We extend the domain of a state μ to expressions in the usual way, i.e., $\mu(\theta(e_1, \ldots, e_k)) = \theta(\mu(e_1), \ldots, \mu(e_k))$.

$$\frac{\lambda(n) = \mathit{out} := x, \quad n \rightarrow n', \quad \sigma' = \sigma[\mathit{out} \mapsto \sigma(\mathit{out}) : \mu(x)]}{(\sigma, \langle n, \mu, C\rangle : \xi) \Rrightarrow (\sigma', \langle n', \mu, C\rangle : \xi)} \tag{1}$$

$$\frac{\begin{array}{l}\lambda(n) = x := \mathit{in}, \quad n \rightarrow n', \quad \sigma(\mathit{in}) = a : \zeta, \\ \mu' = \mu[x \mapsto a], \quad \sigma' = \sigma[\mathit{in} \mapsto \zeta]\end{array}}{(\sigma, \langle n, \mu, C\rangle : \xi) \Rrightarrow (\sigma', \langle n', \mu', C\rangle : \xi)} \tag{2}$$

$$\frac{\lambda(n) = x := e, \quad n \rightarrow n', \quad \mu' = \mu[x \mapsto \mu(e)]}{(\sigma, \langle n, \mu, C\rangle : \xi) \Rrightarrow (\sigma, \langle n', \mu', C\rangle : \xi)} \tag{3}$$

$$\frac{\begin{array}{l}\lambda(n) = x := f(e_1, \ldots, e_k), \quad \mathit{param}(f) = (x_1, \ldots, x_k), \\ \mu' = \bot[x_1 \mapsto \mu(e_1), \ldots, x_k \mapsto \mu(e_k)], \\ C' = C \cap \mathit{SP}(f)\end{array}}{(\sigma, \langle n, \mu, C\rangle : \xi) \Rrightarrow (\sigma, \langle \mathit{IT}(f), \mu', C'\rangle : \langle n, \mu, C\rangle : \xi)} \tag{4}$$

$$\frac{\begin{array}{l}\lambda(n) = x := f(e_1, \ldots, e_k), \quad \lambda(m) = \texttt{return}, \quad n \rightarrow n', \\ \mu'' = \mu[x \mapsto \mu'(\mathit{ret}_f)]\end{array}}{(\sigma, \langle m, \mu', C'\rangle : \langle n, \mu, C\rangle : \xi) \Rrightarrow (\sigma, \langle n', \mu'', C'\rangle : \xi)} \tag{5}$$

$$\frac{\lambda(n) = \texttt{check}[P], \quad P \subseteq C, \quad n \rightarrow n'}{(\sigma, \langle n, \mu, C\rangle : \xi) \Rrightarrow (\sigma, \langle n', \mu, C\rangle : \xi)} \tag{6}$$

$$\frac{\lambda(n) = \texttt{if}\ e, \quad n \overset{\text{then}}{\rightarrow} n', \quad \mu(e) \neq \mathit{false}}{(\sigma, \langle n, \mu, C\rangle : \xi) \Rrightarrow (\sigma, \langle n', \mu, C\rangle : \langle n, \mu, C\rangle : \xi)} \tag{7}$$

$$\frac{\lambda(n) = \texttt{if}\ e, \quad n \overset{\text{else}}{\rightarrow} n', \quad \mu(e) = \mathit{false}}{(\sigma, \langle n, \mu, C\rangle : \xi) \Rrightarrow (\sigma, \langle n', \mu, C\rangle : \langle n, \mu, C\rangle : \xi)} \tag{8}$$

$$\frac{\lambda(n) = \texttt{fi}, \quad n \rightarrow n'}{(\sigma, \langle n, \mu, C\rangle : \langle m, \mu', C'\rangle : \xi) \Rrightarrow (\sigma, \langle n', \mu, C\rangle : \xi)} \tag{9}$$

Rule (3) means that when the control is at the program point n of an assignment statement $x := e$ and $n \rightarrow n'$, the control can move to n' with updating the state of variables to $\mu' = \mu[x \mapsto \mu(e)]$. Rules (1) and (2) are similar to Rule (3). Rule (4) means that when the control is at the program point n of a function call statement $x := f(e_1, \ldots, e_k)$, a new stack frame is pushed into the stack, the control moves to $\mathit{IT}(f)$, and the current values of $e_1, \ldots, e_k$ are assigned to the formal parameters of f. At the same time, the current permission set is updated

to $C \cap SP(f)$. Rule (5) means that when the control is at the end point m of a function, the stack top is removed from the stack and the value of ret_f is returned to the caller. Rule (6) represents the transition for check statements. Rules (7), (8), and (9) represent the transition for `if` statements. Although pushing a stack frame into the stack at Rules (7) and (8) is unnecessary for M_π, we defined the rules so to keep the proof of soundness (Theorem 1) simple.

An initial configuration of a program π is $(\sigma_0, \langle IT(f_0), \bot, SP(f_0)\rangle)$ where σ_0 is a state that maps every output channel to the empty sequence and f_0 is the main function. A stack frame $\langle n, \mu, C\rangle$ is said to be *reachable* if there exists a transitions $cnf_0 \Rightarrow \cdots \Rightarrow (\sigma, \langle n, \mu, C\rangle : \xi)$ for some initial configuration cnf_0 and some σ and ξ. The above sequence is called an *execution trace*. Similarly, a node n is *reachable* if there is a reachable stack frame $\langle n, \mu, C\rangle$ for some μ and C.

3 Information Flow Specification

An *information flow specification* is an assignment of security classes to the input and output channels.

We assume that the set of security classes, denoted as $\mathcal{SC}$, is an arbitrary finite semilattice partially ordered by a relation $\sqsubseteq$. The least element of $\mathcal{SC}$ is denoted as L (= Low), and the least upper bound of $a, b \in \mathcal{SC}$ is denoted as $a \sqcup b$. A simple example of $\mathcal{SC}$ is $\{H, L\}$ (High and Low) such that $L \sqsubseteq H$. For this example, transmitting a value computed using a value from an H input channel into an L output channel is a violation of the information flow specification.

Absence of the violation of an information flow specification can formally be defined in terms of *noninterference*; i.e., no violation exists if values written to every L output channel do not change even when values read out from an H input channel change. However, it is well-known that noninterference is an undecidable property even if access control is absent. Therefore, we define an abstract system $M_\pi^\sharp$ from a program π and a specification Γ and define the notion of type safety by regarding each security class as a type. The soundness of this type system is guaranteed by Theorem 1, which states that type safety of $M_\pi^\sharp$ implies noninterference of π for Γ.

3.1 Derived Pushdown System and Type Safety

For a given program π and an information flow specification Γ, we define a transition system $M_\pi^\sharp$ as follows. Intuitively, $M_\pi^\sharp$ represents the behavior of a program that is the same as π except that each variable x keeps a security class instead of a value stored in x. Since the set $\mathcal{SC}$ of security classes is finite, $M_\pi^\sharp$ is a pushdown system (PDS), and we can compute the reachable set of stack frames of $M_\pi^\sharp$ [10].

A configuration of $M_\pi^\sharp$ is a stack. While a stack frame of M_π is a triple $\langle n, \mu, C\rangle$, that of $M_\pi^\sharp$ is $\langle n, sc, C\rangle$ in which sc is an assignment of security classes to local variables and permissions. The transition relation $\Rightarrow$ of $M_\pi^\sharp$ is the smallest relation that satisfies the following inference rules, in which $e^\sqcup$ is the expression obtained from an expression e by substituting $\sqcup$ for every built-in operator in e.

$$\frac{\lambda(n) = out := x, \quad n \rightarrow n'}{\langle n, sc, C\rangle : \xi \Rightarrow \langle n', sc, C\rangle : \xi} \tag{10}$$

$$\frac{\lambda(n) = x := in, \quad n \rightarrow n', \quad sc' = sc[x \mapsto \Gamma(in) \sqcup sc(\nu_{\text{if}})]}{\langle n, sc, C\rangle : \xi \Rightarrow \langle n', sc', C\rangle : \xi} \tag{11}$$

$$\frac{\lambda(n) = x := e, \quad n \rightarrow n', \quad sc' = sc[x \mapsto sc(e^{\sqcup} \sqcup \nu_{\text{if}})]}{\langle n, sc, C\rangle : \xi \Rightarrow \langle n', sc', C\rangle : \xi} \tag{12}$$

$$\frac{\begin{array}{l}\lambda(n) = x := f(e_1, \ldots, e_k), \quad param(f) = (x_1, \ldots, x_k), \\ C' = C \cap SP(f), \\ sc' = \bot[x_1 \mapsto sc(e_1^{\sqcup} \sqcup \nu_{\text{if}}), \ldots, x_k \mapsto sc(e_k^{\sqcup} \sqcup \nu_{\text{if}}), \nu_{\text{if}} \mapsto sc(\nu_{\text{if}})] \\ \qquad\quad [p \mapsto sc(p \sqcup \nu_{\text{if}}) \mid p \in C \setminus C'] \\ \qquad\quad [p \mapsto sc(p) \qquad\quad \mid p \in \overline{C \setminus C'}]\end{array}}{\langle n, sc, C\rangle : \xi \Rightarrow \langle IT(f), sc', C'\rangle : \langle n, sc, C\rangle : \xi} \tag{13}$$

$$\frac{\begin{array}{l}\lambda(n) = x := f(e_1, \ldots, e_k), \quad \lambda(m) = \texttt{return}, \quad n \rightarrow n' \\ sc'' = sc[x \mapsto sc'(ret_f) \sqcup sc(\nu_{\text{if}})][p \mapsto sc'(p) \mid p \in PRM]\end{array}}{\langle m, sc', C'\rangle : \langle n, sc, C\rangle : \xi \Rightarrow \langle n', sc'', C'\rangle : \xi} \tag{14}$$

$$\frac{\lambda(n) = \texttt{check}[P], \quad P \subseteq C, \quad n \rightarrow n'}{\langle n, sc, C\rangle : \xi \Rightarrow \langle n', sc, C\rangle : \xi} \tag{15}$$

$$\frac{\lambda(n) = \texttt{if}\ e, \quad (n \stackrel{\text{then}}{\rightarrow} n' \text{ or } n \stackrel{\text{else}}{\rightarrow} n'), \quad sc' = sc[\nu_{\text{if}} \mapsto sc(e^{\sqcup} \sqcup \nu_{\text{if}})]}{\langle n, sc, C\rangle : \xi \Rightarrow \langle n', sc', C\rangle : \langle n, sc, C\rangle : \xi} \tag{16}$$

$$\frac{\lambda(n) = \texttt{fi}, \quad n \rightarrow n', \quad sc'' = sc[\nu_{\text{if}} \mapsto sc'(\nu_{\text{if}})]}{\langle n, sc, C\rangle : \langle m, sc', C'\rangle : \xi \Rightarrow \langle n', sc'', C\rangle : \xi} \tag{17}$$

An initial configuration and reachable stack frames of $M^{\sharp}_{\pi}$ are defined in the same way as M_{π}.

In the above definition of $M^{\sharp}_{\pi}$, each assignment statement is replaced with a calculation on security classes. For example, an assignment statement $z := x + y$ in π is replaced with $z := x \sqcup y$ in $M^{\sharp}_{\pi}$, because the security class of z after this assignment is the maximum of the security classes of x and y.

We also have to consider the *implicit flow* from the condition of an `if` statement to each branch of the statement. For example, although variable y does not appear in the right-hand side of each assignment statement for x in the following `if` statement, one can decide whether $y = 0$ or not if he or she can see the value of x just after this `if` statement. That is, information of y implicitly flows to x.

$$\texttt{if}\ \ y = 0\ \ \texttt{then}\ \ x := 0\ \ \texttt{else}\ \ x := 1\ \ \texttt{fi}$$

We use a special variable ν_{if} in $M^{\sharp}_{\pi}$ for representing the security class of information that implicitly flows. The security class of ν_{if} increases at the beginning of a conditional branch, and the increase is canceled at the end of the branch

(see Rules (16) and (17)). To save the security class of ν_{if} before a conditional branch, a new stack frame is pushed into the stack in Rule (16).

We have to consider another kind of implicit flow caused by a check statement. For example, when the following compound statement is executed and $p \notin SP(f)$ for the callee function f, one can know whether $y = 0$ or not because if $y = 0$ then permission p is removed from the current permission set and the execution is aborted at the check statement.

$$\texttt{if } y = 0 \texttt{ then } x := f() \texttt{ fi; check}[\{p\}]$$

In this case, the information on whether $y = 0$ or not flows into the current permission set, and then the information flows outside by the check statement (even when the execution is not aborted at the check statement). Hence we take the security class of each permission p into account as well as that of each variable. The security class of p represents the security class of information on whether or not p remains in the current permission set. Moreover, we consider that information on each permission contained in the argument of a check statement flows into an insecure output channel; that is, a type error exists if a permission p whose security class is not L is contained in the argument of a check statement (cf. type error E3 described below).

Type Error We say that there exists a type error in $M^\sharp_\pi$ if there exists a reachable stack frame $\langle n, sc, C\rangle$ that satisfies any of the following conditions E1 to E4. If no type error exists in $M^\sharp_\pi$, then we say that π is *type-safe*.

E1) $\lambda(n) = out := x$ for an output channel out and $sc(x \sqcup \nu_{\text{if}}) \not\sqsubseteq \Gamma(out)$.
E2) $\lambda(n) = x := in$ for an input channel in and $sc(\nu_{\text{if}}) \not\sqsubseteq \Gamma(in)$.
E3) $\lambda(n) = \texttt{check}[P]$ and $sc(p) \neq L$ for some $p \in P$.
E4) $\lambda(n) = \texttt{check}[P]$ and $P \not\subseteq C$ and $sc(\nu_{\text{if}}) \neq L$.

E1 represents a situation in which a value of a security class higher than an output channel out is written to out. E2 represents an information leak through an input channel. We assume that an attacker can be aware of reading out a value from an input channel, and thus an implicit flow occurs if the reading out is performed in a branch of an `if`-statement. Therefore we consider that a type error exists if $sc(\nu_{\text{if}}) \not\sqsubseteq \Gamma(in)$ for an input statement $x := in$. E3 and E4 represent an information leak through a check statement. E3 represents a situation in which information on the current permission set flows out. E4 represents a situation in which an attacker is aware of the current program point because of the abortion at the check statement.

Another kind of information flow may occur when a program does not terminate. For example, when functions $main$ and f are defined as follows, this program does not terminate if and only if the value read out from in is zero.

$$main()\ \{\ y := in;\ \texttt{if } y = 0 \texttt{ then } x := f() \texttt{ fi}\ \}$$
$$f()\ \{\ z := f()\ \}$$

For simplicity, we ignore this kind of information flow in this paper (termination insensitivity). Although we can modify our type-error detection method as follows so that it becomes sound even for the above kind of information flow, the modified method may report more false positives because it considers that every loop may not terminate.

(1) Find all functions in $M^\sharp_\pi$ that may not terminate. This can be performed by finding cycles in the call graph.
(2) For a function call statement that calls a function that may not terminate, if $sc(\nu_{\mathrm{if}}) \neq L$ then we regard it as a type error.

3.2 Soundness

The above type-error detection method using $M^\sharp_\pi$ is sound in the sense that π satisfies noninterference if $M^\sharp_\pi$ is type-safe (and if π always terminates). This soundness is guaranteed by Theorem 1 shown below. In the following, $sc_1 \sqcup sc_2$ is the assignment of security classes such that $(sc_1 \sqcup sc_2)(x) = sc_1(x) \sqcup sc_2(x)$ for all x. The reflexive transitive closures of $\Rrightarrow$ and $\Rightarrow$ are denoted as $\Rrightarrow^*$ and $\Rightarrow^*$, respectively.

Lemma 1. *Assume that $sc'(y) \sqsubseteq \tau$ for every sc' such that $\langle n, sc_i, C\rangle \Rightarrow^* \langle n', sc', C'\rangle$ $(i = 1, 2)$. Then, $sc''(y) \sqsubseteq \tau$ holds for every sc'' such that $\langle n, sc_1 \sqcup sc_2, C\rangle \Rightarrow^* \langle n', sc'', C'\rangle$.*

Proof. Define a subset V of variables as $V = \{\, x \mid$ The security class of x at n "affects" the security class of x at $n' \,\}$. Because $M^\sharp$ performs no calculation other than $\sqcup$, $x \in V$ implies $sc_i(x) \sqsubseteq \tau$ $(i = 1, 2)$ by the assumption. Moreover, $sc''(y)$ depends only on $(sc_1 \sqcup sc_2)(x)$ for $x \in V$. Thus, again, $M^\sharp$ performs no calculation other than $\sqcup$, $sc''(y) \sqsubseteq \bigsqcup_{x\in V}(sc_1 \sqcup sc_2)(x)$ holds. By the definition of $sc_1 \sqcup sc_2$, $\bigsqcup_{x\in V}(sc_1 \sqcup sc_2)(x) \sqsubseteq \tau$. □

Lemma 2. *If $M^\sharp$ is type-safe when the initial configuration is $\langle n, sc_1, C\rangle$ and when that is $\langle n, sc_2, C\rangle$, then $M^\sharp$ is type-safe when the initial configuration is $\langle n, sc_1 \sqcup sc_2, C\rangle$.*

Proof. By Lemma 1. □

Below we show the soundness theorem (Theorem 1). Intuitive meaning of each item stated in the theorem is as follows. Item (1) states that $M^\sharp_\pi$ is type-safe. Items (2) and (3) state an assumption such that for two execution traces of π, the value of every variable whose security class is less than or equal to τ at the beginning of those traces is the same. Items (4) and (5) state that if the security class of a variable y at the end of every trace is less than or equal to τ, then the value of y at the end of every trace in π is the same; i.e. π satisfies noninterference. Item (6) is the same property for input and output channels.

Theorem 1 (Soundness). *Let π be an HBAC program, Γ be a specification with $\mathcal{SC}$ as the set of security classes, n be a program point in π, $sc \in \mathcal{SC}$, and C_1 and C_2 be subsets of permissions in π such that*

(1) there exists no type error in $M^\sharp_\pi$ if the initial configuration of $M^\sharp_\pi$ is either $\langle n, sc, C_1\rangle$ or $\langle n, sc, C_2\rangle$.

Assume the folloing three conditions hold.

(2) $(\sigma_i, \langle n, \mu_i, C_i\rangle) \Rrightarrow^* (\sigma'_i, \langle n', \mu'_i, C'_i\rangle)\quad (i = 1, 2)$.
(3) For every variable x, $sc(x) \sqsubseteq \tau$ implies $\mu_1(x) = \mu_2(x)$. For every $io \in In \cup Out$, $\Gamma(io) \sqsubseteq \tau$ implies $\sigma_1(io) = \sigma_2(io)$. For every permission p, $sc(p) \sqsubseteq \tau$ implies $p \in C_1 \Leftrightarrow p \in C_2$.
(4) For any sc' such that $\langle n, sc, C_i\rangle \Rightarrow^* \langle n', sc', C'_i\rangle$ ($i = 1$ or 2), $sc'(y) \sqsubseteq \tau$.

Then, the following two conditions hold.

(5) $\mu'_1(y) = \mu'_2(y)$ if y is a variable. $y \in C'_1 \Leftrightarrow y \in C'_2$ if y is a permission.
(6) For every $io \in In \cup Out$, $\Gamma(io) \sqsubseteq \tau$ implies $\sigma'_1(io) = \sigma'_2(io)$.

Proof. Let $\alpha_i = ((\sigma_i, \langle n, \mu_i, C_i\rangle) \Rrightarrow \cdots \Rrightarrow (\sigma'_i, \langle n', \mu'_i, C'_i\rangle))$ $(i = 1, 2)$. This theorem can be proved by induction on the length of α_1.

(Basis) Assume that the length of α is zero. This implies $n = n'$. Since no iteration statement exists, the length of α_2 is also zero, and thus $\sigma_i = \sigma'_i,\ \mu_i = \mu'_i$, and $C_i = C'_i$ $(i = 1, 2)$. Moreover, $n = n'$ and $C_1 = C'_1$ imply $\langle n, sc, C_1\rangle \Rightarrow^* \langle n, sc, C_1\rangle = \langle n', sc, C'_1\rangle$, and thus $sc(y) \sqsubseteq \tau$ by Assumption (4). Therefore (5) and (6) hold by Assumption (3).

(Induction step) Let the length of α_1 be $\ell > 0$ and assume that this theorem holds whenever the length of α_1 is less than ℓ.

(A) If $\lambda(n)$ is a function call statement and α_1 ends in the return from the callee function, or $\lambda(n)$ is an `if` statement and α_1 ends in the end of the `if` statement, or $\lambda(n)$ is another statement and α_1 is just the execution of the statement, then we can show (5) and (6) by the definition of M_π and $M^\sharp_\pi$.

Note that since the stack of the last configuration of α_1 contains only one stack frame, if $\lambda(n)$ is a function call statement then the end of α_1 never be in the middle of the callee function. In the same way, if $\lambda(n)$ is an `if` statement then the end of α_1 never be in the middle of that statement.

(B) Execution traces α_i $(i = 1, 2)$ can be decomposed into $(\sigma_i, \langle n, \mu_i, C_i\rangle) \Rrightarrow \cdots \Rrightarrow (\sigma''_i, \langle n'', \mu''_i, C''_i\rangle) \Rrightarrow \cdots \Rrightarrow (\sigma'_i, \langle n', \mu'_i, C'_i\rangle)$ and we can show that this theorem holds for the first part of α_i to $(\sigma''_i, \langle n'', \mu''_i, C''_i\rangle)$ by Case (A).

Let S be the set that consists of every sc'' such that $\langle n, sc, C_i\rangle \Rightarrow^* \langle n'', sc'', C''_i\rangle$ $(i = 1$ or $2)$. Since this theorem holds for the first part of α_i $(i = 1, 2)$, $(\sqcup S)(x) \sqsubseteq \tau$ implies $\mu''_1(x) = \mu''_2(x)$ for every variable x and $(\sqcup S)(p) \sqsubseteq \tau$ implies $p \in C''_1 \Leftrightarrow p \in C''_2$ for every permission p.

For every $sc'' \in S$, $\langle n'', sc'', C''_i\rangle \Rightarrow^* \langle n', sc', C'_i\rangle$ $(i = 1$ or $2)$ implies $\langle n, sc, C_i\rangle \Rightarrow^* \langle n', sc', C'_i\rangle$ and thus $sc'(y) \sqsubseteq \tau$ by Assumption (4). Thus $\langle n'', \sqcup S, C''_i\rangle \Rightarrow^* \langle n', sc', C'_i\rangle$ implies $sc'(y) \sqsubseteq \tau$ by Lemma 1. Moreover, by Lemma 2, $M^\sharp$ is type-safe when the initial configuration is $\langle n'', \sqcup S, C''_i\rangle$.

Therefore we can show (5) and (6) by applying the induction hypothesis to the second part (from n'' to n') of α_i $(i = 1, 2)$ and the execution trace $\langle n'', \sqcup S, C''_i\rangle \Rightarrow^* \langle n', sc', C'_i\rangle$ of $M^\sharp$. $\square$

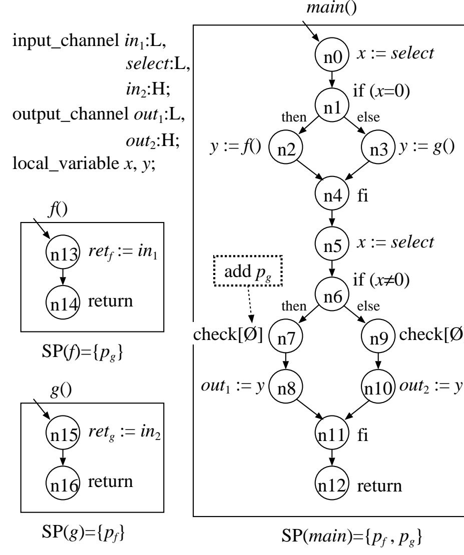

Fig. 3. Program π_1.

4 Permission-Check Statement Insertion Problem

4.1 Problem Statement

The permission-check statement insertion problem is defined as follows.

Input A program π and an information flow specification Γ. All check statements in π must be `check[∅]`.

Output A type-safe program π' that is obtained from π by modifying the arguments of arbitrary number of check statements.

Example 2. Consider the program π_1 and the information flow specification (i.e. assignment of security classes to input and output channels) in Figure 3. If input channel *select* always gives a non-zero value at program points n_0 and n_5, then the execution trace of the program is (a prefix of) the following transition sequence of stacks, where the three components of each stack frame represent a program point, a state of variables and permissions, and the current permission set, respectively. The state of variables and permissions is a tuple of the security classes of variables x, y, ν_{if}, ret_f, and ret_g, and permissions p_f and p_g.

$$
\begin{aligned}
&\langle n_0, (\bot, \bot, \bot, \bot, \bot, \bot, \bot), \{p_f, p_g\}\rangle \\
&\Rightarrow \langle n_1, (L, \bot, \bot, \bot, \bot, \bot, \bot), \{p_f, p_g\}\rangle \\
&\Rightarrow \langle n_3, (L, \bot, L, \bot, \bot, \bot, \bot), \{p_f, p_g\}\rangle : \langle n_1, (\ldots), \{\ldots\}\rangle
\end{aligned}
$$

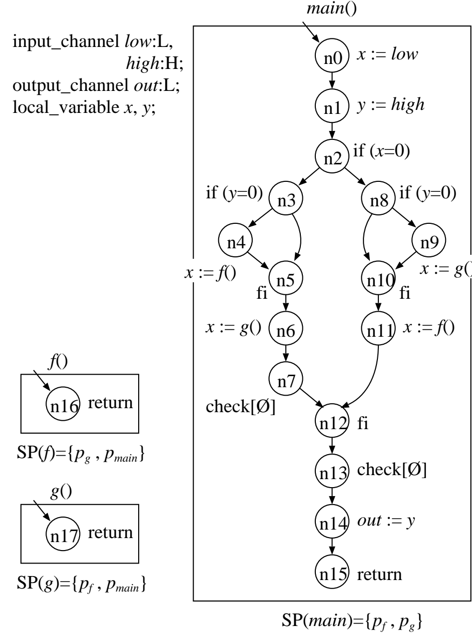

Fig. 4. Program π_2.

$$
\begin{aligned}
&\Rightarrow \langle n_{15}, (\perp, \perp, L, \perp, \perp, \perp, L), \{p_f\}\rangle : \langle n_3, (\ldots), \{\ldots\}\rangle : \langle n_1, (\ldots), \{\ldots\}\rangle \\
&\Rightarrow \langle n_{16}, (\perp, \perp, L, \perp, H, \perp, L), \{p_f\}\rangle : \langle n_3, (\ldots), \{\ldots\}\rangle : \langle n_1, (\ldots), \{\ldots\}\rangle \\
&\Rightarrow \langle n_4, (L, H, L, \perp, \perp, \perp, L), \{p_f\}\rangle : \langle n_1, (\ldots), \{\ldots\}\rangle \\
&\Rightarrow \langle n_5, (L, H, L, \perp, \perp, \perp, L), \{p_f\}\rangle \\
&\Rightarrow \langle n_6, (L, H, L, \perp, \perp, \perp, L), \{p_f\}\rangle \\
&\Rightarrow \langle n_7, (L, H, L, \perp, \perp, \perp, L), \{p_f\}\rangle : \langle n_6, (\ldots), \{\ldots\}\rangle \\
&\Rightarrow \langle n_8, (L, H, L, \perp, \perp, \perp, L), \{p_f\}\rangle : \langle n_6, (\ldots), \{\ldots\}\rangle \\
&\Rightarrow \langle n_{11}, (L, H, L, \perp, \perp, \perp, L), \{p_f\}\rangle : \langle n_6, (\ldots), \{\ldots\}\rangle \\
&\Rightarrow \langle n_{12}, (L, H, L, \perp, \perp, \perp, L), \{p_f\}\rangle
\end{aligned}
$$

In the above trace, a type error of E1 occurs in the statement $out_1 := y$ at n_8. To remove this error, we can add permission p_g to the argument of the check statement at n_7. After this addition, the execution along the above trace is aborted at n_7 since the current permission set $\{p_f\}$ at n_7 in that trace does not contain p_g, and the above error is removed. Moreover, this addition does not bring any other type errors of E3 or E4 because the security classes of p_g and ν_{if} at n_7 are L in every trace from n_0 to n_7.

Example 3. Consider the program π_2 and the information flow specification in Figure 4. When the control reaches n_{14}, a type error of E1 occurs since the

security class of variable y is H. We thus have to modify the arguments of the check statements at n_7 and n_{13} to make n_{14} unreachable. When the control reaches n_7, the stack top is fr_1 or fr_2 shown below according to the execution trace from n_0 to n_7. When the control reaches n_{13}, the stack top is one of $fr_3, \ldots, fr_6$. In the following stack frames, the second component is the same as Example 2; that represents the security classes of x, y, ν_{if}, ret_f, ret_g, p_f, and p_g.

$$\begin{aligned}
fr_1 &= \langle n_7, (\bot, H, L, \bot, \bot, H, L), \emptyset\rangle \\
fr_2 &= \langle n_7, (\bot, H, L, \bot, \bot, \bot, L), \{p_f\}\rangle \\
fr_3 &= \langle n_{13}, (\bot, H, \bot, \bot, \bot, H, L), \emptyset\rangle \\
fr_4 &= \langle n_{13}, (\bot, H, \bot, \bot, \bot, \bot, L), \{p_f\}\rangle \\
fr_5 &= \langle n_{13}, (\bot, H, \bot, \bot, \bot, L, H), \emptyset\rangle \\
fr_6 &= \langle n_{13}, (\bot, H, \bot, \bot, \bot, L, \bot), \{p_g\}\rangle
\end{aligned}$$

Stack frames fr_1 and fr_3 correspond to traces going through both n_4 and n_6, while fr_2 and fr_4 correspond to traces going through n_6 and not through n_4. Similarly, fr_5 corresponds to a trace going through both n_9 and n_{11}, while fr_6 corresponds to a trace going through n_{11} and not through n_9.

If we add p_f to the argument of n_{13}, then a type error of E3 occurs at fr_3. On the other hand, if we add p_g to the argument of n_{13}, then a type error of E3 occurs at fr_5. However, we can add p_g to the argument of n_7 without type errors of E3 or E4, and this addition makes fr_3 and fr_4 unreachable. Therefore we can make n_{14} unreachable without type errors of E3 or E4 by letting the argument of n_7 be $\{p_g\}$ and the argument of n_{13} be $\{p_f\}$.

4.2 Complexity

Theorem 2. *The permission-check statement insertion problem is co-NP-hard.*

Proof. Let ChkIns be the decision problem corresponding to the permission-check statement insertion problem; i.e., ChkIns is the problem that answers whether or not a solution of the permission-check statement insertion problem exists for a given program π and a given information flow specification Γ. We show a reduction from the complement of 3SAT to ChkIns.

Let $\langle U, C\rangle$ be an instance of 3SAT where $U = \{x_1, \ldots, x_n\}$ is the set of variables and $C = \{c_1, \ldots, c_m\}$ is the set of clauses. We write each positive literal x_i as x_{i1} and each negative literal $\overline{x_i}$ as x_{i0}. We also write the three literals of each clause c_j as $x_{c_{j1}}$, $x_{c_{j2}}$, and $x_{c_{j3}}$, respectively.

By the reduction, an instance $\langle U, C\rangle$ of 3SAT is transformed into an instance $\langle \pi_3, \Gamma\rangle$ of ChkIns shown in Figure 5. Program π_3 consists of three parts A, B, and C, and a type error occurs whenever the control reaches part C because a value read out from h is written to *out*.

Just after the control pass through part A, the following properties hold.

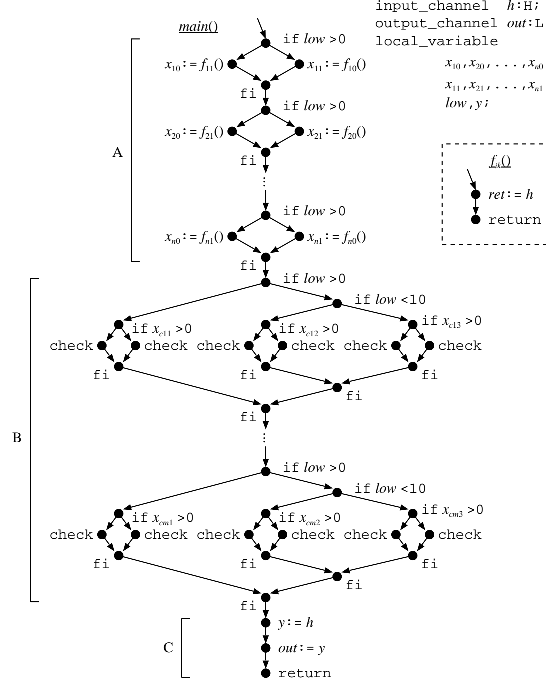

Fig. 5. Program π_3.

(1) For every possible truth assignment S of U, there exists an execution trace of π_3 such that if $S(x_i)$ is true then the security classes of x_{i1} and x_{i0} are H and L, respectively, and if $S(x_i)$ is not true then the security classes of x_{i1} and x_{i0} are L and H, respectively.

(2) The security class of x_{ik} is H if and only if function f_{ik} has never been called.

Firstly, assume that the set C of clauses is satisfiable. In this case, there exists the execution trace corresponding to a truth assignment that satisfies C. When the control has passed through part A along the trace, there exists some $\ell_j \in \{1, 2, 3\}$ for each clause c_j such that the security class of $x_{c_{j\ell_j}}$ is H. Hence there exists an the execution trace in part B that passes the node "`if` $x_{c_{j\ell_j}} > 0$" for every j, and every check statement on this trace cannot abort the execution without a type error of E4. Therefore no solution exists for $\langle \pi_3, \Gamma \rangle$.

Secondly, assume that C is not satisfiable. In this case, we can obtain a type-safe program π' from π by letting the argument of every check statement be $\{p_{f_{c_{j\ell}}}\}$ if the check statement is in a branch of `if` $x_{c_{j\ell}} > 0$. The type-safeness is shown as follows. Since C is unsatisfiable, after any execution of part A, there exists some j such that the security classes of $x_{c_{j1}}$, $x_{c_{j2}}$, and $x_{c_{j3}}$ are L. Thus $f_{c_{j1}}$, $f_{c_{j2}}$, and $f_{c_{j3}}$ have been called by the above (2), and the execution is going

to be aborted at any of six check statements corresponding to c_j; i.e., the control never reaches part C. Moreover, a type error of E4 never occurs in any check statement because if the execution is aborted at a check statement in a branch of `if` $x_{c_{j\ell}} > 0$, then permission $p_{f_{c_{j\ell}}}$ is not in the current permission set and thus $f_{c_{j\ell}}$ has been called. By the above (2), the security class of $x_{c_{j\ell}}$, which is the same as $sc(\nu_{\mathrm{if}})$, is L. □

5 Automatic Generation

5.1 Algorithm

We call a program point n a *check node* if $\lambda(n) = \mathtt{check}[P]$ for some P. On the permission-check statement insertion problem, all we can do is to add permissions to the argument of check statements. When a program π' is obtained from π by adding permissions to the arguments of check statements, the transition relation of $M^{\sharp}_{\pi'}$ is a subset of that of $M^{\sharp}_{\pi}$, since the precondition of the inference rule (15) in Section 3.1 holds for $M^{\sharp}_{\pi}$ if it holds for $M^{\sharp}_{\pi'}$ and the other rules do not depend on check statements. Hence, we can design an algorithm for solving the problem as follows. For each reachable stack frame fr of $M^{\sharp}_{\pi}$ that causes a type error of E1 or E2, make fr unreachable by adding a permission to the argument of some check statement in each execution trace α from an initial configuration to fr. Let $\alpha = cnf_0 \Rightarrow \cdots \Rightarrow \langle n, sc, C\rangle : \xi \Rightarrow \cdots \Rightarrow fr : \xi'$ be such a trace where n is the check node whose argument is to be modified. If a permission $p \notin C$ is added to the argument of n, the execution will be aborted at n in α. So if we perform this modification for every α, then fr becomes unreachable. However, the above modification may introduce another type error of E3 or E4. Let us fix the node n for a while. The necessary and sufficient condition to avoid such a type error as a side effect is: $sc(p) = L$, $sc(\nu_{\mathrm{if}}) = L$, and any other stack frame $\langle n, sc', C'\rangle$ for the same n for which p brings a type error (i.e. $sc'(p) \neq L$ or ($p \notin C'$ and $sc'(\nu_{\mathrm{if}}) \neq L$)) is unreachable (by possibly modifying the argument of another check statement).

Based on the above observation, the algorithm consists of two phases:

(1) For each check node n, compute $SafeP(n)$, which is the set of all permissions that can be added to the argument of n without type error or with type error that can be removed by other check statements; ($SafeP(n)$ is formally defined in Phase (1) below.)

(2) For every type-error stack frame fr and a trace $\alpha = cnf_0 \Rightarrow \cdots \Rightarrow fr : \xi'$, find a configuration $\langle n, sc, C\rangle : \xi$ in α and a permission p such that n and p satisfy the condition mentioned in the previous paragraph, by using $SafeP(n)$. If the addition of p introduces a type error, then repeat Phase (2).

In the following, let cnf_0 be an initial configuration of $M^{\sharp}_{\pi}$, $top(fr : \xi) = fr$ be the function that answers the stack top, and $LP(sc) = \{ p \mid sc(p) = L \}$ be the subset of permissions whose security class is L for a given sc.

Phase (1): Computation of *SafeP(n)* We define the following two inference rules (18) and (19).

$$\frac{\lambda(n) \neq \texttt{check[}P\texttt{]} \text{ or } sc(\nu_{\text{if}}) \neq L \text{ or } (\mathit{SafeP}(n) \cap \mathit{LP}(sc)) \setminus C = \emptyset}{\neg \mathit{Stoppable}(\langle n, sc, C \rangle)} \tag{18}$$

$$\frac{\begin{array}{l} \mathit{cnf}_0 \Rightarrow \ldots \Rightarrow \mathit{cnf}_\ell \Rightarrow \langle n, sc, C \rangle : \xi, \quad (sc(p) \neq L \text{ or} \\ (p \notin C,\ sc(\nu_{\text{if}}) \neq L)), \quad \neg \mathit{Stoppable}(\mathit{top}(\mathit{cnf}_i)) \text{ for } 0 \leq i \leq \ell \end{array}}{p \notin \mathit{SafeP}(n)} \tag{19}$$

Since every occurence of $\mathit{Stoppable}(\cdot)$ and $\mathit{SafeP}(\cdot)$ is negative, these two rules have the greatest fixpoints for these two predicates, which can be computed as follows. We say that a stack frame *fr* is stoppable if $\mathit{Stoppable}(\mathit{fr})$ holds during the computation.

(i) Let $\mathit{SafeP}(n) = \mathit{PRM}$ for each check node n (see Section 2.2 for PRM) and $M^{\sharp\prime}_\pi$ be $M^\sharp_\pi$ without any transitions.
(ii) Compute $\mathit{Stoppable}(\mathit{fr})$ for each stack frame *fr* of $M^\sharp_\pi$ by (18) according to the current $\mathit{SafeP}(\cdot)$, and add transitions to $M^{\sharp\prime}_\pi$ from each non-stoppable frame.
(iii) Compute $\mathit{SafeP}(n)$ for each check node n by (19) according to the current $\mathit{Stoppable}(\cdot)$. To do this, compute all reachable stack frames of $M^{\sharp\prime}_\pi$ based on a model checking method for pushdown systems. If a stack frame $\langle n, sc, C \rangle$ for a check node n is reachable, then remove every permission p satisfying the second precondition of (19) from $\mathit{SafeP}(n)$.
(iv) Repeat Steps (ii) and (iii) until no more change occurs.

Phase (2): Changing the arguments of check statements Let $M^\sharp_{\text{ex}}$ be the pushdown system obtained from $M^\sharp_\pi$ by extending the stack frames to 4-tuples and substituting the following inference rules (20) and (21) for Rule (15) for check statements. The fourth component of a stack frame is a pair $\langle n, Q \rangle$ of a program point and a subset of permissions. This pair represents that $\mathit{top}(\mathit{cnf}) = \langle n, sc, C \rangle$ for some sc and C such that *cnf* is the last stoppable configuration on the execution trace to the current configuration and the execution will be actually aborted by adding an arbitrary element of Q to the argument of n.

$$\frac{\begin{array}{l} \lambda(n) = \texttt{check}[P], \quad P \subseteq C, \quad n \to n', \quad sc(\nu_{\text{if}}) = L, \\ Q = (\mathit{SafeP}(n) \cap \mathit{LP}(sc)) \setminus C \neq \emptyset \end{array}}{\langle n, sc, C, X \rangle : \xi \Rightarrow \langle n', sc, C, \langle n, Q \rangle \rangle : \xi} \tag{20}$$

$$\frac{\begin{array}{l} \lambda(n) = \texttt{check}[P], \quad P \subseteq C, \quad n \to n', \quad (sc(\nu_{\text{if}}) \neq L \text{ or} \\ (\mathit{SafeP}(n) \cap \mathit{LP}(sc)) \setminus C = \emptyset) \end{array}}{\langle n, sc, C, X \rangle : \xi \Rightarrow \langle n', sc, C, X \rangle : \xi} \tag{21}$$

By the following algorithm, the arguments of check statements are modified to remove type errors.

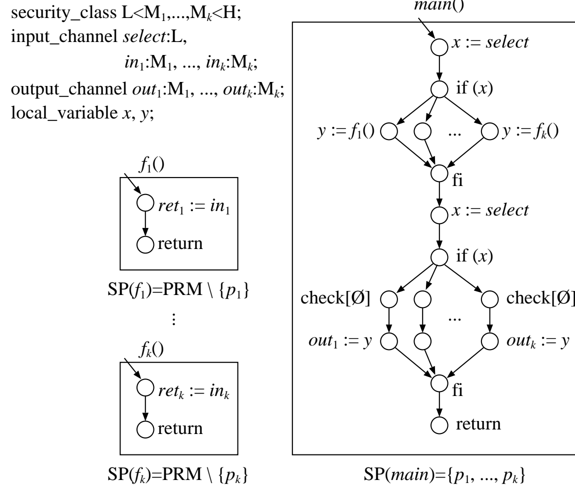

Fig. 6. Program $\pi_a(k)$.

(i) Let the fourth component of the initial configuration of $M^\sharp_{\text{ex}}$ be $\bot$, and compute all reachable stack frames of $M^\sharp_{\text{ex}}$.
(ii) If a type-error stack frame $\langle n, sc, C, X\rangle$ is reachable and $X = \langle n', Q\rangle$, then add an arbitrary element of Q to the argument of the check statement at n'. If $X = \bot$, then notify a user that the given problem instance has no solution, and halt.
(iii) Repeat Steps (i) and (ii) until no more change occurs.

5.2 Experiment

We describe the results of an experiment in which a prototype implementation of the above algorithm is applied to the following two kinds of problem instances. Each input program is an extension of program π_1 in Figure 3.

(1) The input program $\pi_a(k)$ in Figure 6 has k functions f_i for $1 \leq i \leq k$, and the security class of the return value of function f_i is M_i. Security class M_i ($1 \leq i \leq k$) satisfies $L \sqsubseteq M_i \sqsubseteq H$ and $M_i \not\sqsubseteq M_j$ and $M_j \not\sqsubseteq M_i$ for every $j \neq i$. Program $\pi_a(k)$ also has k check statements and k output channels, and the security class of each output channel out_i is M_i. Thus we have to modify the argument of each check statement so that the return value of f_i is written only to out_i.

(2) Program $\pi_b(k)$ in Figure 7 is a program obtained from $\pi_a(k)$ by splitting the lower part of the main function into a separate function, which can be arbitrarily repeated by a tail call to itself.

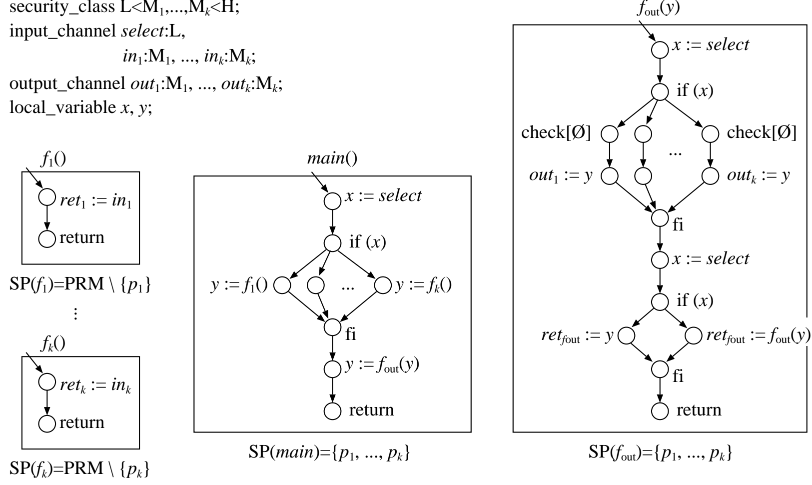

Fig. 7. Program $\pi_b(k)$.

Figure 8 shows the computation time for $\pi_a(k)$ and $\pi_b(k)$[4]. The computation time for $\pi_a(k)$ is approximately $O(k^2)$, and the time for $\pi_b(k)$ is approximately $O(k^3)$. Computation of reachable stack frames of $M^\sharp_{\mathrm{ex}}$ is dominant in the proposed algorithm. We adopt an efficient method for computing the set of reachable stack frames described in [24, Section 4.4], whose computation time is proportional to the number of reachable stack frames. The number of reachable stack frames for $\pi_a(k)$ and $\pi_b(k)$ is shown in Figure 9, which plots the same curves as Figure 8. This result suggests that the time complexity of the proposed algorithm is the order of the number of reachable stack frames.

Comparison between the proposed implementation and Moped The main part of the proposed algorithm is the computation of the set of reachable stack frames, and it can be performed using existing model checking tools for PDS. However, those are not optimized for analysis of HBAC programs and their suitability for the permission-check statement insertion problem is unknown. Hence we measured the computation time required by PDS model checking tool *Moped* (version 1.0.14)[5] for computing the set of reachable stack frames of $\pi_a(k)$ (Figure 10).

While our implementation required at most two seconds when $k \leq 100$, the computation time of Moped rapidly increases, and it becomes more than several hours when $k \geq 17$. From this results, our implementation is more suitable for the permission-check statement insertion problem than Moped.

---

[4] The prototype implementation is written in C (GCC 4.1.2). We use a computer with Intel Core 2 Duo 1.06 GHz, 2 GB RAM, and CentOS 5.3.

[5] http://www.fmi.uni-stuttgart.de/szs/tools/moped/

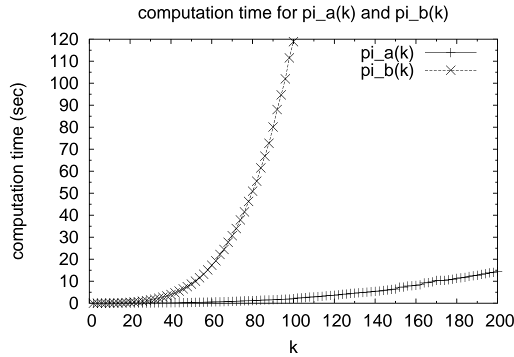

Fig. 8. Computation time for $\pi_a(k)$ and $\pi_b(k)$.

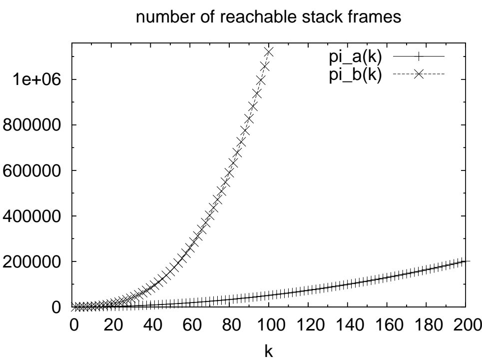

Fig. 9. The number of reachable stack frames for $\pi_a(k)$ and $\pi_b(k)$.

6 Conclusion

In this paper we studied on a problem to automatically insert permission-check statements for making a given program satisfy a given information flow specification. We showed that the problem is co-NP-hard. We also proposed an algorithm based on a model checking method of pushdown systems. Applying a prototype implementation to problem instances, we found that the complexity of the proposed algorithm is proposional to the number of reachable stack frames.

Future work includes the followings.

(1) A method for finding the optimal solution: Some problem instances have more than one solution, and the proposed algorithm does not necessarily answer an optimal one. We would like to investigate an algorithm to find the solution that minimize the total size of the argument of check statements.

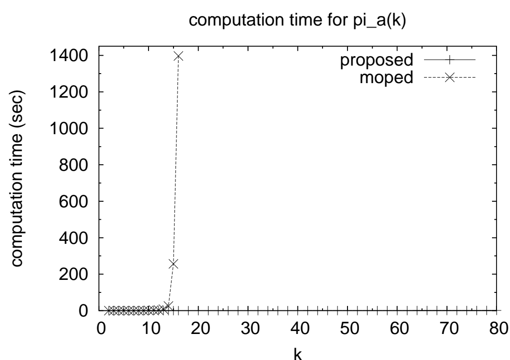

Fig. 10. Comparison between the proposed implementation and Moped.

(2) An algorithm for the original definition of HBAC programs: We would like to extend the proposed algorithm to the original definition of HBAC programs, where freedom of the static permission set of each function and the grant and accept set of each function call statement is imposed.

References

1. Abadi, M., Fournet, C.: Access control based on execution history. In: Network & Distributed System Security Symp. pp. 107–121 (2003)
2. Banâtre, J., Bryce, C., Le Métayer, D.: Compile-time detection of information flow in sequential programs. In: 3rd ESORICS, LNCS 875. pp. 55–73 (1994)
3. Banerjee, A., Naumann, D.A.: Using access control for secure information flow in a Java-like language. In: IEEE 16th CSFW. pp. 155–169 (2003)
4. Banerjee, A., Naumann, D.A.: History-based access control and secure information flow. In: CASSIS 04, LNCS 3362. pp. 27–48 (2004)
5. Banerjee, A., Naumann, D.A.: Stack-based access control and secure information flow. Journal of Functional Programming 5(2), 131–177 (2005)
6. Bartoletti, M., Degano, P., Ferrari, G.L.: Static analysis for stack inspection. ConCoord, Electric Notes in Theoretical Computer Science 54 (2001)
7. Besson, F., Blanc, T., Fournet, C., Gordon, A.D.: From stack inspection to access control: A security analysis for libraries. In: IEEE 17th CSFW. pp. 61–75 (2004)
8. Denning, D.E.: A lattice model of secure information flow. Commun. ACM 19(5), 236–243 (1976)
9. Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. Commun. ACM 20(7), 504–513 (1977)
10. Esparza, J., Hansel, D., Rossmanith, P., Schwoon, S.: Efficient algorithms for model-checking pushdown systems. In: CAV 2000, LNCS 1855. pp. 232–247 (2000)
11. Esparza, J., Kučera, A., Schwoon, S.: Model-checking LTL with regular variations for pushdown systems. In: TACS 01, LNCS 2215. pp. 316–339 (2001)
12. Fong, P.W.: Access control by tracking shallow execution history. In: IEEE Security & Privacy. pp. 43–55 (2004)

13. Guernic, G.L., Banerjee, A., Jensen, T., Schimidt, D.A.: Automata-based confidentiality monitoring. In: ASIAN 2006 (2006)
14. Heintze, N., Riecke, J.G.: The Slam calculus: programming with secrecy and integrity. In: 25th ACM POPL. pp. 365–377 (1998)
15. Jensen, T., le Métayer, D., Thorn, T.: Verification of control flow based security properties. In: IEEE Security & Privacy. pp. 89–103 (1999)
16. Koved, L., Pistoia, M., Kershenbaum, A.: Access rights analysis for Java. In: ACM 17th OOPSLA. pp. 359–372 (2002)
17. Leroy, X., Rouaix, F.: Security properties of typed applets. In: 25th ACM POPL. pp. 391–403 (1998)
18. Myers, A.C.: JFLOW: Practical mostly-static information flow control. In: 26th ACM POPL. pp. 228–241 (1999)
19. Myers, A.C., Liskov, B.: Complete, safe information flow with decentralized labels. In: IEEE Security & Privacy. pp. 186–197 (1998)
20. Nitta, N., Takata, Y., Seki, H.: An efficient security verification method for programs with stack inspection. In: 8th ACM CCS. pp. 68–77 (2001)
21. Ørbæk, P.: Can you trust your data? In: TAPSOFT '95, LNCS 915. pp. 575–589 (1995)
22. Pistoia, M., Banerjee, A., Naumann, D.A.: Beyond stack insepction: A unified acess-control and information-flow security model. In: IEEE Security & Privacy. pp. 149–163 (2007)
23. Pottier, F., Skalka, C., Smith, S.: A systematic approach to access control. In: ESOP 2001, LNCS 2028. pp. 30–45 (2001)
24. Takata, Y., Wang, J., Seki, H.: A formal model and its verification of history-based access control. IEICE Trans. on Information and Systems (Japanese Edition) J91-D(4), 847–858 (2008)
25. Volpano, D., Smith, G.: A type-based approach to program security. In: TAPSOFT '97, LNCS 1214. pp. 607–621 (1997)
26. Wang, J., Takata, Y., Seki, H.: HBAC: A model for history-based access control and its model checking. In: 11th ESORICS, LNCS 4189. pp. 263–278 (2006)