An Efficient Security Verification Method for Programs with Stack Inspection

Naoya Nitta
Graduate School of Information Science
Nara Institute of Science and Technology
8916-5, Takayama, Ikoma, Nara, 630-0101, Japan
naoya-n@is.aist-nara.ac.jp

Yoshiaki Takata
Graduate School of Information Science
Nara Institute of Science and Technology
8916-5, Takayama, Ikoma, Nara, 630-0101, Japan
y-takata@is.aist-nara.ac.jp

Hiroyuki Seki
Graduate School of Information Science
Nara Institute of Science and Technology
8916-5, Takayama, Ikoma, Nara, 630-0101, Japan
seki@is.aist-nara.ac.jp

ABSTRACT

Stack inspection is a key technology for runtime access control of programs in a network environment. In this paper, a verification problem to decide whether a given program with stack inspection satisfies a given security property is discussed. First, the computational complexity of the problem is investigated. Since the result implies the problem is computationally intractable in general, we introduce a practically important subclass of programs which exactly model programs containing checkPermission of Java development kit 1.2. We show that the problem for this subclass is solvable in linear time in the size of a program.

Categories and Subject Descriptors

D.4.6 [**Software**]: Operating System—*Security and Protection*; K.6.5 [**Computing Milieux**]: Management of Computing and Information Systems—*Security and Protection*

General Terms

Security, Verification

Keywords

access control, security verification, stack inspection, Java

1. INTRODUCTION

Stack inspection is a simple but sufficiently practical access control technology, which is provided by Java development kit 1.2 (JDK1.2) [9]. In the JDK1.2 environment, every method belongs to one of the protection domains, and each protection domain is granted several *permissions*. If a method m belongs to a protection domain d and d is granted a permission p, then we simply say m has permission p. If the method *checkPermission*(p) with a permission p as an actual argument is invoked from a method m, then JDK1.2 examines not only whether the method m has p but also whether every ancestor method which directly or indirectly invokes m have p. If all those methods have p, then the execution continues. Otherwise, the execution is aborted. The runtime control stack (or simply, stack) consists of frames for the active method and its ancestor methods. A frame for a method m contains the protection domain which m belongs to as well as actual arguments, local variables and the return address of m. The stack is inspected by *checkPermission*(p) from the top (the active method) to the bottom to examine whether the above mentioned condition is met; if a method which does not have p is encountered, then the execution is aborted. If the stack bottom or a method with a particular mode (called *privileged*) is encountered, then *checkPermission*(p) terminates successfully and returns to the active method.

The owner of local methods which directly or indirectly access secret local resources is responsible for placing appropriate check statements, which invokes *checkPermission*, in those local methods. The system is expected to satisfy a certain *global security property* such as 'if the control reaches *write* method, then the control has passed through only methods which have permission $p_{customer}$ or p_{write}.' However, ensuring that a program satisfies such a global security property by hand becomes difficult when the whole program is large and complicated.

In [14], we introduced a security verification problem for programs with stack inspection based on [12]. The problem was defined as whether every reachable state of P satisfies ψ for a given program P and a global security property ψ. We showed that the problem is decidable. However, neither the computational complexity of solving the problem nor a practical verification method has been investigated.

In this paper, we first analyze how the complexity of the verification problem alters when the representation of regular languages to specify a check statement and a global security property is changed (deterministic finite automaton, nondeterministic finite automaton and regular expression). Since the complexity results imply that the problem is generally intractable (unless P = PSPACE), we introduce

CCS'01, November 5-8, 2001, Philadelphia, Pennsylvania, USA.

a subclass of programs which model programs containing *checkPermission* in JDK1.2. This subclass is called $\Pi_{\text{JDK1.2}}$. Next, we show that the verification problem for $\Pi_{\text{JDK1.2}}$ is efficiently solvable for the size of a given program.

The rest of the paper is organized as follows. In section 2, we review the verification problem with a brief example. The computational complexity of the problem is investigated in section 3. In section 4, we show that time complexity of the problem is linear in the program size for the subclass $\Pi_{\text{check-free}}$ such that the property in each check statement in a program in this subclass is trivially true. In section 5, time complexity of the verification problem for the subclass $\Pi_{\text{JDK1.2}}$ is shown to be linear in the program size by reducing the problem for $\Pi_{\text{JDK1.2}}$ to the problem for $\Pi_{\text{check-free}}$. Finally, we draw some concluding remarks in section 6.

RELATED WORKS. Our program model and the definition of the verification problem are based on the model introduced in [12], the pioneering paper which first discussed the security verification of programs with stack inspection. The difference between the model in [12] and ours is that [12] uses linear temporal logic (LTL) formulas [4] to describe both the property in a check statement and a global security property while we use regular languages, whose expressive power to represent a set of finite sequences is properly stronger than that of LTL formulas [7]. Also, the verification algorithm in [12] is based on model checking [4] and works only for mutual recursion-free programs, while we showed that the problem is generally decidable even for programs which contain mutual recursion. In [17], a more general notion of stack inspection is proposed by using ABLP logic [1], which is a kind of belief logic, and a sufficient condition for a check statement to succeed is shown. However, [17] does not discuss the verification of a global security property which is discussed in [12] and our papers.

[5] and [6] are the pioneering works which proposed a static analysis of information flow based on a lattice model of security classes. Denning's analysis method has been formalized and extended in various ways by abstract interpretation [15], type theory [16, 10, 13] and process algebra [2].

2. PRELIMINARIES

2.1 Program Model

Following [12], we model a program as a directed graph called a *flow graph*. Each node of a flow graph corresponds to a location in the program (*program point*). A statement which performs runtime check of access permission such as *checkPermission* in JDK1.2 is called a check statement and is incorporated into the model. A check statement examines whether the current state of the executed program satisfies the property specified in the statement. If the property is satisfied, the program continues its execution. Otherwise, the execution is aborted.

A flow graph has two kinds of edges. The first one is a *transfer edge* (tg), which represents a control flow within a method. For example, if there is a tg from n_1 to n_2 (denoted as $n_1 \stackrel{TG}{\rightarrow} n_2$), then the control can move to n_2 just after the execution of n_1. The second edge is a *call edge* (cg), which represents a method invocation. For example, suppose that there is a cg from n_1 to n_2 (denoted as $n_1 \stackrel{CG}{\rightarrow} n_2$). If the control reaches n_1, then the control can further be passed to n_2.

Formally, a program P is a directed graph represented as a 5-tuple $P = (NO, IS, IT, TG, CG)$ such that:

$$\begin{array}{lcl} IS & : & NO \rightarrow \{call, return, check(L_\phi)\} \\ IT & : & NO \\ TG & : & NO \rightarrow 2^{NO} \\ CG & : & NO \rightarrow 2^{NO}. \end{array}$$

NO is a set of nodes representing program points and 2^{NO} is the set of all subsets of NO. IT is the entry point of the entire program called the *initial node*. TG and CG are sets of transfer edges and call edges, respectively.

The set of nodes is divided into the following three subsets by IS. Let $n \in NO$.

- $IS(n) = call$. n is a *call node* which represents a method call.
- $IS(n) = return$. n is a *return node* which represents the return from a callee method.
- $IS(n) = check(L_\phi)$. n is a *check node* which represents a check statement. If the current state of the program satisfies the property represented by the language L_ϕ, then the execution is continued. Otherwise, the execution is aborted. The syntax and semantics of the language L_ϕ are defined in section 2.2.

2.2 Operational Semantics

A *state* of a program $P = (NO, IS, IT, TG, CG)$ is a finite sequence of nodes, which is also called a *stack*. The initial state of P is the stack which contains only the initial node IT. The state immediately after a method call is the state (stack) obtained by pushing the node of the callee onto the current state. The concatenation of sequences s_1 and s_2 of nodes is represented as $s_1 : s_2$. The sequence which consists of only one node n is denoted by $\langle n \rangle$. We may write n instead of $\langle n \rangle$ if no ambiguity occurs.

Let ϵ denote the empty sequence. For a finite set Σ of symbols, let Σ^* denote the set of all finite sequences on Σ including ϵ. Also let $\Sigma^+ = \Sigma^* - \{\epsilon\}$.

The semantics of a program is defined by a *transition relation* $\triangleright$ on the set of states. For states s_1 and s_2, $s_1 \triangleright s_2$ means that the transition from s_1 to s_2 is possible by a unit execution step of the program.

DEFINITION 2.1. **(transition relation)** *For a given program $P = (NO, IS, IT, TG, CG)$, the relation $\triangleright$ is the least relation which satisfies the following three rules, where s is a state ($\in NO^*$) and n, m, n_i, n_j are nodes ($\in NO$).*

$$\frac{IS(n) = call,\ n \stackrel{CG}{\rightarrow} m}{s : n \ \triangleright\ s : n : m}$$

$$\frac{IS(m) = return,\ n_i \stackrel{TG}{\rightarrow} n_j}{s : n_i : m \ \triangleright\ s : n_j}$$

$$\frac{IS(n) = check(L_\phi),\ s : n \in L_\phi,\ n \stackrel{TG}{\rightarrow} n_j}{s : n \ \triangleright\ s : n_j}$$

□

For a program P, a *trace* of P is a finite sequence of states in which the first state is the initial state and every pair of adjacent states satisfies the transition relation $\triangleright$. The concatenation of states is denoted as $\triangleright$ by slightly abusing the notation. The *set of traces* is defined as follows.

DEFINITION 2.2. **(set of traces)** *For a given program $P = (NO, IS, IT, TG, CG)$, the set $[\![P]\!]$ of traces of P is:*

$$[\![P]\!] = \{s_1 \triangleright \cdots \triangleright s_k \mid s_1 = \langle IT \rangle, s_1, \ldots, s_k \in NO^*, \forall i < k, s_i \triangleright s_{i+1}\}.$$

□

A language L_ϕ in a node $check(L_\phi)$ is a regular language over NO (thus, $L_\phi \subseteq NO^*$). Recall that every state s is a sequence of nodes, that is, $s \in NO^*$. As defined in the third rule of Definition 2.1, if the control reaches a node $check(L_\phi)$, then the execution is continued if and only if the current state belongs to L_ϕ. The model itself does not assume any particular representation (*e.g.*, regular expression, finite automaton) to denote a regular language L_ϕ although we will use regular expression in this section.

EXAMPLE 2.1. **(Java stack inspection in JDK1.2)** *A method invocation checkPermission(p) succeeds if*

- *every frame of the stack has permission p, or*
- *a frame (say f) is privileged and every later frame (including f) in the stack has p.*

Let $\mathcal{N}(p)$ be the set of nodes which have permission p and let PRV be the set of privileged nodes. We can represent checkPermission(p) as the check node check(JDK(p)) where:

$$JDK(p) = (NO^*(PRV \cap \mathcal{N}(p)) \cup \epsilon)(\mathcal{N}(p))^*. \qquad (1)$$

□

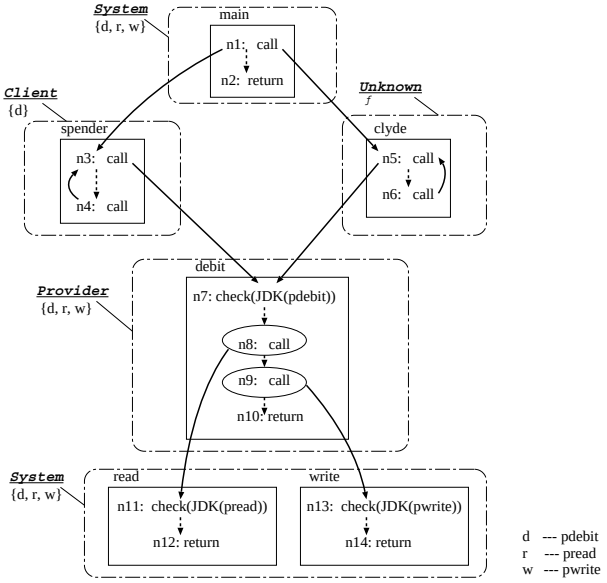

Figure 1: A Sample Program

EXAMPLE 2.2. *Figure 1 shows a program $P = (NO, IS, IT, TG, CG)$ which models a part of an on-line banking system, which serves its clients with a method for withdrawing money. There are four protection domains called System, Provider, Client, and Unknown. A reliable provider is supplied with read and write methods and is privileged by the system. All users including clients and unknowns can invoke a debit method, which invokes read and write methods. In the figure, a solid arrow represents a call edge and a dotted arrow represents a transfer edge. Let $NO = \{n_i \mid 1 \leq i \leq 14\}$. Permissions granted to each protection domain as well as the protection domain which each node belongs to are also shown in Figure 1. For example, the set of permissions granted to System is $\{p_{debit}, p_{read}, p_{write}\}$. Also, the set $\mathcal{N}(p)$ of nodes which have permission p is: $\mathcal{N}(p_{debit}) = \{n_1, n_2, n_3, n_4, n_7, n_8, \cdots, n_{14}\}$ and $\mathcal{N}(p_{read}) = \mathcal{N}(p_{write}) = \{n_1, n_2, n_7, n_8, \ldots, n_{14}\}$. Let $PRV = \{n_8, n_9\}$ be the set of nodes which are privileged. The property $JDK(p_{debit})$ specified in node n_7 is represented by the following regular expression:*

$$JDK(p_{debit}) = (NO^* \ (PRV \cap \mathcal{N}(p_{debit})) \cup \epsilon)(\mathcal{N}(p_{debit}))^*$$

by (1) in Example 2.1. The properties $JDK(p_{read})$ in n_{11} and $JDK(p_{write})$ in n_{13} are represented in the same way. Consider the following two sequences:

$$\begin{aligned}\alpha_1 &= n_1 \triangleright n_1 n_3 \triangleright \underline{n_1 n_3 n_7} \triangleright n_1 n_3 n_8 \triangleright \underline{n_1 n_3 n_8 n_{11}} \triangleright n_1 n_3 n_8 n_{12} \\ &\quad \triangleright n_1 n_3 n_9 \triangleright \underline{n_1 n_3 n_9 n_{13}} \triangleright n_1 n_3 n_9 n_{14} \triangleright n_1 n_3 n_{10} \triangleright n_1 n_4, \\ \alpha_2 &= n_1 \triangleright n_1 n_5 \triangleright \underline{n_1 n_5 n_7} \triangleright n_1 n_5 n_8.\end{aligned}$$

For the state sequence α_1, check nodes are executed three times, at the underlined states. In each case, the state satisfies the property (belongs to the language) specified in the check node. Therefore, $\alpha_1 \in [\![P]\!]$. For the sequence α_2, $n_1 n_5 n_7 \notin JDK(p_{debit})$ since $n_5 \notin \mathcal{N}(p_{debit})$, and hence $\alpha_2 \notin [\![P]\!]$. □

2.3 The Verification Problem

Intuitively, the verification problem is to verify whether every state in every trace in $[\![P]\!]$ of a given program P satisfies a given global security property. A global security property is expressed as a regular language L_ψ, which is called a *verification property*.

Let $L_{\text{safe}}[\psi] = \{\alpha \mid \alpha$ is a state sequence such that every state in α belongs to $L_\psi\}$. $L_{\text{safe}}[\psi]$ can be represented as $L_{\text{safe}}[\psi] = (L_\psi \triangleright)^* L_\psi$. A program P *satisfies* a verification property L_ψ if and only if every state in every trace in $[\![P]\!]$ belongs to L_ψ. Formally, we define the **verification problem** as follows:

Instance: A program P and a verification property L_ψ.

Question: $[\![P]\!] \subseteq L_{\text{safe}}[\psi]$?

EXAMPLE 2.3. *Consider Example 2.2 again. Let $L_\psi = (\overline{E_{RW}})^* \cup (\mathcal{N}(p_{debit}))^* E_{RW} (\overline{E_{RW}})^*$ be the verification property where $E_{RW} = \{n_{11}, n_{12}, n_{13}, n_{14}\}$. $L_{\text{safe}}[\psi] = (L_\psi \triangleright)^* L_\psi$ means that if the control reaches either the read or write method successfully, then the control has passed through only nodes which have p_{debit}. In this particular example, $[\![P]\!] \subseteq L_{\text{safe}}[\psi]$ holds, that is, program P satisfies L_ψ.* □

In [14], we showed the following fundamental property by using indexed grammar [3] to denote the set of traces.

THEOREM 2.1. *For a given program P and a given verification property L_ψ, the verification problem $[\![P]\!] \subseteq L_{\text{safe}}[\psi]$ is decidable.* □

3. COMPLEXITY OF THE VERIFICATION PROBLEM

Since an input of the verification problem is a program P and a verification property L_ψ, the complexity of the problem depends on the representation of regular languages specified in check nodes and L_ψ. We use a finite automaton (FA) as the representation of a regular language. A deterministic FA and a nondeterministic FA are denoted by a DFA and an NFA, respectively. (The results of this paper for NFA remain valid for regular expression.) Let DEXP-POLY time denote the class of decision problems solvable in deterministic $O(c^{p(n)})$ time for a constant c (> 1) and a polynomial p. For a set A, let $|A|$ denote the cardinality of A. The number of states of an FA M is denoted as $\#M$. Let $P = (NO, IS, IT, TG, CG)$ be a program where P contains $check(L_{\phi_i})$ ($1 \leq i \leq k$) and each L_{ϕ_i} is specified by an FA M_{ϕ_i}. The size of P is defined as $\|P\| = |NO| + |TG| + |CG| + \max\{\#M_{\phi_1}, \ldots, \#M_{\phi_k}\}$.

THEOREM 3.1.*[14] Let $P = (NO, IS, IT, TG, CG)$ be a program. Assume that P contains $check(L_{\phi_i})$ ($1 \leq i \leq k$) where each L_{ϕ_i} is specified by an NFA M_{ϕ_i}. Also let L_ψ be a verification property specified by a DFA M_ψ. The verification problem for P and L_ψ is DEXP-POLY time-complete.* □

NOTE. If a verification property L_ψ is also specified by an NFA instead of a DFA in Theorem 3.1, then the upper-bound of the complexity of the algorithm becomes double exponential time.

It is unknown whether the verification problem is still DEXP-POLY time-hard if the language in each check node is specified by a DFA. However, the verification problem in this case can be shown to be PSPACE-hard.

PROPOSITION 3.2. *Let $P = (NO, IS, IT, TG, CG)$ be a program and L_ψ a verification property which satisfies the assumption stated in Theorem 3.1 except that the language L_{ϕ_i} in each $check(L_{\phi_i})$ is specified by a DFA M_{ϕ_i}. The verification problem for P and L_ψ is PSPACE-hard.*

PROOF SKETCH. By transforming the QUANTIFIED 3-SATISFIABILITY (QUANTIFIED 3SAT) problem to the verification problem. □

The time complexity of the verification algorithm in [12] is exponential to both the size of a program and the size of an LTL formula. Also note that although we extend the [12]'s model so that check statements in a program are specified by regular languages instead of LTL formulas, the PSPACE-hardness of the verification problem shown by Proposition 3.2 holds even for the [12]'s model because the languages in check nodes of the program P_{Q3SAT} and the verification property L_ψ in the proof of Proposition 3.2 (see Appendix) can be specified by simple LTL formulas. Also note that P_{Q3SAT} does not contain mutual recursion, and thus the verification problem is intractable (unless P = PSPACE) even if we assume the absence of mutual recursion as is done in the verification algorithm in [12].

4. THE SUBCLASS $\Pi_{\text{check-free}}$ OF PROGRAMS

In this section, we consider the subclass of programs which contain only $check(NO^*)$ as a check node. Since NO^* is the set of all states (i.e., $s \in NO^*$ for every state s), the execution of $check(NO^*)$ always succeeds. We will show that time complexity of the verification problem for this subclass is linear in the size of a program while the complexity depends on the representation of a verification property. This subclass, called $\Pi_{\text{check-free}}$, might seem of no practical use since no program in this class can substantially control any access. However, in section 5 we will introduce a broader subclass $\Pi_{\text{JDK1.2}}$ of programs which exactly model programs with *checkPermission* in JDK1.2, and show that the verification problem for $\Pi_{\text{JDK1.2}}$ can be efficiently solved by transforming a program in $\Pi_{\text{JDK1.2}}$ to a program in $\Pi_{\text{check-free}}$.

The set $[P]$ of all states which are reachable from the initial state of a program P is defined as follows:

$$[P] = \{s \in NO^* \mid \exists s_1 \triangleright s_2 \triangleright \cdots \triangleright s_i \in [\![P]\!],\ s = s_i\}.$$

It is easy to see that for a program P and a verification property L_ψ, $[P] \subseteq L_\psi$ if and only if $[\![P]\!] \subseteq (L_\psi \triangleright)^* L_\psi$. Hence, we can solve the verification problem by deciding $[P] \subseteq L_\psi$ instead of deciding $[\![P]\!] \subseteq (L_\psi \triangleright)^* L_\psi$. If $[P]$ belongs to a class of languages which is closed under intersection with regular languages and for which the emptiness problem is decidable, then we can obtain a decision algorithm for the verification problem since $[P] \cap \overline{L_\psi} = \emptyset$ if and only if $[P] \subseteq L_\psi$. Let $L(G)$ denote the language generated by a grammar G and let $||G||$ denote the size of G. It is known that a context-free grammar (cfg) G' can be constructed from a cfg G such that $L(G') = L(G) \cap L_\psi$ and $||G'||$ is $O(||G||)$, and also the emptiness problem for context-free language is solvable in linear time in the size of cfg [11]. Hence, if $[P]$ is generated by a cfg G such that $||G||$ is $O(||P||)$, then the verification problem is solvable in polynomial time in $||P||$. However, it is open at the current time whether $[P]$ is a context-free language for an arbitrary program P.

Let $\Pi_{\text{check-free}}$ denote the subclass of programs which contain only $check(NO^*)$ as a check node. We will show that for a program P in $\Pi_{\text{check-free}}$, $[P]$ is a regular language and hence we can efficiently decide whether $[P] \subseteq L_\psi$. For any given program $P = (NO, IS, IT, TG, CG)$ in $\Pi_{\text{check-free}}$, we define a predicate CR (can return) : $NO \to \{True,\ False\}$ such that $\text{CR}(n) = True$ if and only if the control can return after n is invoked.

$$\frac{IS(n_1) = call,\ n_1 \overset{CG}{\to} n_2,\ n_1 \overset{TG}{\to} n_3,\ \text{CR}(n_2) = \text{CR}(n_3) = True}{\text{CR}(n_1) = True} \tag{2}$$

$$\frac{IS(n) = return}{\text{CR}(n) = True} \tag{3}$$

$$\frac{IS(n_1) = check(NO^*),\ n_1 \overset{TG}{\to} n_2,\ \text{CR}(n_2) = True}{\text{CR}(n_1) = True} \tag{4}$$

LEMMA 4.1. *Let $P = (NO, IS, IT, TG, CG)$ be a program in $\Pi_{\text{check-free}}$. For an arbitrary node $n \in NO$ of P,*

$\text{CR}(n) = True$ *if and only if* *there exist a node n' such that $IS(n') = return$ and a valid state sequence $n \triangleright \cdots \triangleright n'$,*

where a valid state sequence T is a sequence of states satisfying the following condition:

$T = s_1 \triangleright \cdots \triangleright s_k$ such that $s_1, \ldots, s_k \in NO^*$ and $\forall i < k.\ s_i \triangleright s_{i+1}$.

PROOF SKETCH. The *only if* part is shown by induction on the number of inference rules of CR used for deriving $\mathrm{CR}(n) = \mathit{True}$. The *if* part is shown by induction on l of a valid state sequence $s_1 \triangleright \cdots \triangleright s_l$. □

Using the predicate CR, we can construct a regular grammar $G_{P,\mathrm{S}} = (N, T, R, S)$ which generates the set $[P]$ as follows.

(a) N is a finite set of nonterminal symbols and $N = \{S\} \cup \{N_i \mid n_i \in NO\}$.

(b) T is a finite set of terminal symbols and $T = NO$.

(c) S is the start symbol.

(d) R is the set of productions which consists of:

$$S \to N_1 \quad \text{for } n_1 = IT \tag{5}$$

$$N_i \to n_i \quad \text{for } \forall n_i \in NO. \tag{6}$$

For each node n_i, the following productions are added to R according to $IS(n_i)$.

(1) $IS(n_i) = \mathit{call}$:

$$N_i \to n_i N_j \quad \text{for } \forall n_j.\ n_i \overset{CG}{\to} n_j \tag{7}$$

If $\exists n_j.\ n_i \overset{CG}{\to} n_j$ and $\mathrm{CR}(n_j) = \mathit{True}$, then the following production is also added to R.

$$N_i \to N_k \quad \text{for } \forall n_k.\ n_i \overset{TG}{\to} n_k \tag{8}$$

(2) $IS(n_i) = \mathit{check}(NO^*)$:

$$N_i \to N_j \quad \text{for } \forall n_j.\ n_i \overset{TG}{\to} n_j \tag{9}$$

We can show that the language $L(G_{P,\mathrm{S}})$ generated by regular grammar $G_{P,\mathrm{S}}$ coincides with the set $[P]$ of states.

THEOREM 4.2. *For a program* $P = (NO, IS, IT, TG, CG)$ *in* $\Pi_{\text{check-free}}$, $L(G_{P,\mathrm{S}}) = [P]$.

PROOF SKETCH. It suffices to show that for every $s \in NO^*$, $S \to^*_{G_{P,\mathrm{S}}} s$ *if and only if* $IT \triangleright \cdots \triangleright s \in [\![P]\!]$, which can be proved by using Lemma 4.1. □

THEOREM 4.3. *Let* $P = (NO, IS, IT, TG, CG)$ *be a program in* $\Pi_{\text{check-free}}$. *The verification problem for* P *and a verification property specified by a DFA* M_ψ *is solvable in* $O((|TG| + |CG|) \cdot \#M_\psi)$ *time.*

PROOF. For an FA M, let $L(M)$ denote the language accepted by M. From Theorem 4.2, we can construct a regular grammar G_P such that $L(G_P) = [P] \cap \overline{L(M_\psi)}$ and $\|G_P\|$ is $O((|TG| + |CG|) \cdot \#M_\psi)$. The emptiness problem for the language generated by a regular grammar G is solvable in $O(\|G\|)$ time. Hence, we can decide whether $L(G_P) = \emptyset$ in $O((|TG| + |CG|) \cdot \#M_\psi)$ time. Furthermore, all the values of predicate CR can be determined in $O(|TG| + |CG|)$ time by applying the inference rules (in a non-redundant way) until no value of CR for each node changes. Therefore, the verification problem is solvable in $O((|TG| + |CG|) \cdot \#M_\psi)$ time. □

As is the case with Theorem 3.1, the proof of the above theorem is no longer valid if a verification property L_ψ is specified by an NFA instead of a DFA. Especially, the verification problem in this case can be shown to be PSPACE-complete.

PROPOSITION 4.4. *Let* $P = (NO, IS, IT, TG, CG)$ *be a program in* $\Pi_{\text{check-free}}$. *The verification problem for* P *and a verification property specified by an NFA* M_ψ *is PSPACE-complete.*

PROOF SKETCH. Both problems, FINITE AUTOMATON INEQUIVALENCE and REGULAR EXPRESSION NON-UNIVERSALITY are known to be PSPACE-complete[8]. PSPACE-solvability of the verification problem can be shown by transforming the problem to the FINITE AUTOMATON INEQUIVALENCE problem. PSPACE-hardness of the verification problem can be shown by transforming REGULAR EXPRESSION NON-UNIVERSALITY to the problem. □

5. THE SUBCLASS $\Pi_{\text{JDK1.2}}$ OF PROGRAMS

5.1 Permission Based Model

As shown in sections 3 and 4, the verification problem is computationally intractable unless P = PSPACE while the problem for the subclass $\Pi_{\text{check-free}}$ is solvable in polynomial time in the program size. In this section, we introduce another subclass $\Pi_{\text{JDK1.2}}$ of programs which contain only check nodes equivalent to *checkPermission* in JDK1.2. $\Pi_{\text{JDK1.2}}$ properly includes $\Pi_{\text{check-free}}$. Also we show that the verification problem for $\Pi_{\text{JDK1.2}}$ is solvable in linear time in the size of a program by reducing the problem for $\Pi_{\text{JDK1.2}}$ to the problem for $\Pi_{\text{check-free}}$.

In JDK1.2, an access to a resource is controlled by inspecting the current contents of the stack. This mechanism can be implemented as follows (called *eager evaluation* in section 2.4 of [9]).

- Assume that a set of permissions is granted to each method and every node in a method has the permissions granted to that method. Also assume that at each state of a program, the control keeps the set of effective permissions. The set of effective permissions is updated as follows when a method invocation or a return occurs.
- When a method m_2 is invoked from a method m_1 and the invocation is not privileged, then the set of effective permissions becomes the intersection of the current set and the set of permissions granted to m_2.
- When a method m_2 is invoked from a method m_1 and the invocation is privileged, then the set of effective permissions becomes the intersection of the sets of permissions granted to m_1 and granted to m_2.
- An access is controlled by inspecting the current set of effective permissions instead of by inspecting the contents of the stack.

From a program $P_{\text{JDK1.2}}$ in $\Pi_{\text{JDK1.2}}$, we can construct an equivalent program $\widehat{P}$ where each node is a pair of a node of $P_{\text{JDK1.2}}$ and the set of effective permissions at that node. For every check node of $P_{\text{JDK1.2}}$, using the set of effective

permissions in the node, we can statically know the result of the execution of the check node. Therefore, by removing all transfer edges emitted from the check nodes at which the execution is aborted and by replacing all check nodes with $check(NO^*)$, we can obtain an equivalent program $\widehat{P}$ which belongs to $\Pi_{\text{check-free}}$. By Theorem 4.3, the verification problem can be solved in linear time in the size of $\widehat{P}$.

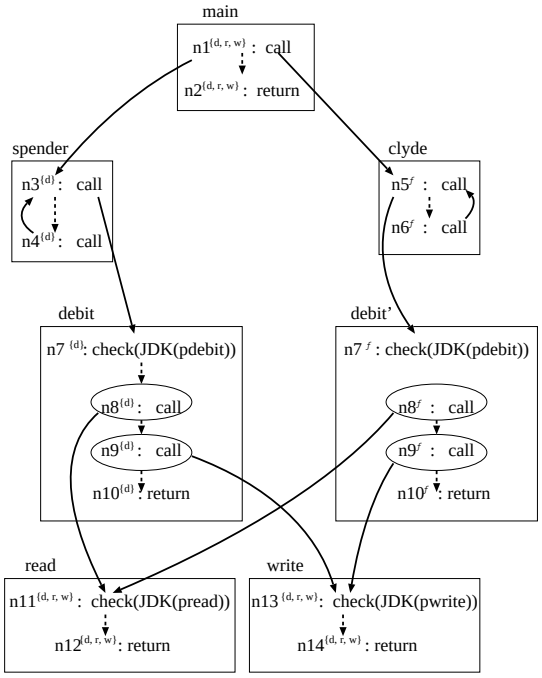

Figure 2: An equivalent program in $\Pi_{\text{check-free}}$

For example, from the program shown in Figure 1, we can construct an equivalent program shown in Figure 2, where each node is labeled with the set of effective permissions. Since there are two paths in Figure 1 from the initial node to the *debit* method and the sets of effective permissions at the entry point of the *debit* method are different according to the selected path, the *debit* method is duplicated in Figure 2. Since the execution of node $n_7^{\emptyset}$ does not succeed ($p_{debit} \notin \emptyset$), transfer edge from $n_7^{\emptyset}$ to $n_8^{\emptyset}$ is removed. The details of this construction are described as follows.

Let PRM be a finite set of access permissions. A program $P_{\text{JDK1.2}}$ in $\Pi_{\text{JDK1.2}}$ is a 7-tuple $P_{\text{JDK1.2}} = (NO, IS, IT, TG, CG, P\_BY, PRV)$. The first five components (NO, IS, IT, TG, CG) are the same as those defined in section 2.1. The last two components are:

$$P\_BY : NO \to 2^{PRM}$$
$$PRV \subseteq \{n \mid n \in NO,\ IS(n) = call\}.$$

$P\_BY(n)$ (possessed by n) is the set of permissions which a node n has. In this model, we assume that all nodes in a method have the same set of permissions, that is,

$$n \overset{TG}{\to} n' \Rightarrow P\_BY(n) = P\_BY(n'). \tag{10}$$

The set $\mathcal{N}(p)$ of nodes which have a permission p is

$$\mathcal{N}(p) = \{n \mid p \in P\_BY(n)\}.$$

PRV is the set of privileged nodes. As a check node, only $check(JDK(p))$ is allowed, where $p \in PRM$. Recall that $JDK(p)$ is represented as:

$$JDK(p) = (NO^*(PRV \cap \mathcal{N}(p)) \cup \epsilon)(\mathcal{N}(p))^*. \tag{11}$$

We present a transformation from a given program in $\Pi_{\text{JDK1.2}}$ to a program in $\Pi_{\text{check-free}}$.

Construction 5.1.
Input: *a program* $P_{\text{JDK1.2}} = (NO, IS, IT, TG, CG, P\_BY, PRV)$.
Output: *the program* $\widehat{P} = (\widehat{NO}, \widehat{IS}, \widehat{IT}, \widehat{TG}, \widehat{CG})$ *in* $\Pi_{\text{check-free}}$ *where:*

1. $\widehat{NO} = NO \times 2^{PRM}$. *An element of* $\widehat{NO}$ *is represented as* $n^{\mathcal{P}_i}$ $(n \in NO,\ \mathcal{P}_i \subseteq PRM)$.
2. *For arbitrary* $n \in NO$ *and* $\mathcal{P}_i \subseteq PRM$,
   - (2.1) $IS(n) = call \Rightarrow \widehat{IS}(n^{\mathcal{P}_i}) = call$,
   - (2.2) $IS(n) = return \Rightarrow \widehat{IS}(n^{\mathcal{P}_i}) = return$,
   - (2.3) $IS(n) = check(JDK(p)) \Rightarrow \widehat{IS}(n^{\mathcal{P}_i}) = check(\widehat{NO}^*)$.
3. $\widehat{IT} = IT^{P\_BY(IT)}$.
4. $\widehat{TG}$ *and* $\widehat{CG}$ *are defined as follows. For an arbitrary* $\mathcal{P}_i \subseteq PRM$,
   - (4.1) $n_i \overset{CG}{\to} n_j \Rightarrow n_i^{\mathcal{P}_i} \overset{\widehat{CG}}{\longrightarrow} n_j^{\mathcal{P}_j}$ *where*
     $$\mathcal{P}_j = \begin{cases} \mathcal{P}_i \cap P\_BY(n_j) & n_i \notin PRV, \\ P\_BY(n_i) \cap P\_BY(n_j) & n_i \in PRV, \end{cases}$$
   - (4.2) $n_i \overset{TG}{\to} n_j$ *and* $IS(n_i) = call \Rightarrow n_i^{\mathcal{P}_i} \overset{\widehat{TG}}{\longrightarrow} n_j^{\mathcal{P}_i}$,
   - (4.3) $n_i \overset{TG}{\to} n_j$, $IS(n_i) = check(JDK(p))$ *and* $p \in \mathcal{P}_i \Rightarrow n_i^{\mathcal{P}_i} \overset{\widehat{TG}}{\longrightarrow} n_j^{\mathcal{P}_i}$.

□

Note that program $\widehat{P}$ contains only $check(\widehat{NO}^*)$ as a check node and hence $\widehat{P} \in \Pi_{\text{check-free}}$. In practice, it suffices to construct the nodes reachable from the initial node $\widehat{IT}$ and the edges connecting them.

We show that the set $[\![P_{\text{JDK1.2}}]\!]$ of traces of a program $P_{\text{JDK1.2}}$ in $\Pi_{\text{JDK1.2}}$ coincides with the set $[\![\widehat{P}]\!]$ of traces of program $\widehat{P}$ (modulo the homomorphism which erases the effective permissions).

Lemma 5.1. *For a program* $P_{\text{JDK1.2}} = (NO, IS, IT, TG, CG, P\_BY, PRV)$, *let* $\widehat{P} = (\widehat{NO}, \widehat{IS}, \widehat{IT}, \widehat{TG}, \widehat{CG})$ *be the program obtained from* $P_{\text{JDK1.2}}$ *by Construction 5.1. Let us define the homomorphism* $h : (\widehat{NO} \cup \{\triangleright\})^* \to (NO \cup \{\triangleright\})^*$ *as* $h(n^{\mathcal{P}}) = n$ *for* $n^{\mathcal{P}} \in \widehat{NO}$ *and* $h(\triangleright) = \triangleright$. *Then,* $[\![P_{\text{JDK1.2}}]\!] = h([\![\widehat{P}]\!])$.

Proof Sketch. It suffices to show that $IT \triangleright \cdots \triangleright n_1 n_2 \cdots n_k \in [\![P_{\text{JDK1.2}}]\!]$ *if and only if* $\widehat{IT} \triangleright \cdots \triangleright n_1^{\mathcal{P}_1} n_2^{\mathcal{P}_2} \cdots n_k^{\mathcal{P}_k} \in [\![\widehat{P}]\!]$ where $\mathcal{P}_1 = P\_BY(n_1)$ and

$$\mathcal{P}_i = \begin{cases} \mathcal{P}_{i-1} \cap P\_BY(n_i) & n_{i-1} \notin PRV \\ P\_BY(n_{i-1}) \cap P\_BY(n_i) & n_{i-1} \in PRV \end{cases}$$

for each $1 < i \leq k$. The *only if* part is shown by induction on the length of a trace of $P_{\text{JDK1.2}}$. The *if* part is shown by induction on the length of a trace of $\widehat{P}$. □

THEOREM 5.2. *The verification problem for a program $P_{\text{JDK1.2}}$ in $\Pi_{\text{JDK1.2}}$ and a verification property specified by a DFA M_ψ is solvable in linear time in the program size $||P_{\text{JDK1.2}}||$ and the number $\sharp M_\psi$ of states of M_ψ.*

PROOF. Let $P_{\text{JDK1.2}} = (NO, IS, IT, TG, CG, P\_BY, PRV)$ be a program in $\Pi_{\text{JDK1.2}}$ and let $\widehat{P}$ be the program in $\Pi_{\text{check-free}}$ obtained from $P_{\text{JDK1.2}}$ by Construction 5.1. Recall that solving the verification problem for $P_{\text{JDK1.2}}$ and $L(M_\psi)$ is equivalent to deciding whether $[P_{\text{JDK1.2}}] \subseteq L(M_\psi)$ or not. By Lemma 5.1, this decision is further equivalent to deciding whether $h([\widehat{P}]) \subseteq L(M_\psi)$. $\widehat{P}$ belongs to $\Pi_{\text{check-free}}$ and the class of regular languages is closed under homomorphism (for any regular grammar G, a regular grammar G' can be constructed such that $L(G') = h(L(G))$ and $||G|| = O(||G||)$). Hence, by Theorem 4.3, the verification problem for $P_{\text{JDK1.2}}$ and $L(M_\psi)$ is solvable in polynomial time in $||\widehat{P}||$ and $\sharp M_\psi$. More specifically, the problem is solvable in $O((|\widehat{TG}| + |\widehat{CG}|) \cdot \sharp M_\psi)$ time. Since $|\widehat{TG}|$ and $|\widehat{CG}|$ are $O(|TG| \cdot 2^{|PRM|})$ and $O(|CG| \cdot 2^{|PRM|})$ respectively, the time complexity is $O((|TG| + |CG|) \cdot \sharp M_\psi \cdot 2^{|PRM|})$. □

NOTE. By Proposition 4.4, we can see that if a verification property L_ψ is specified by an NFA M'_ψ instead of a DFA M_ψ in Theorem 5.2, then the time complexity of the verification problem becomes PSPACE-complete.

5.2 Domain Based Model

In Construction 5.1, we let each node of $\widehat{P}$ be a pair of a node of $P_{\text{JDK1.2}}$ and the set of effective permissions, and thus the size of $\widehat{P}$ is $O(\|P_{\text{JDK1.2}}\| \cdot 2^{|PRM|})$. In this section we show an alternative way of constructing $\widehat{P}$ from $P_{\text{JDK1.2}}$ where each node of $\widehat{P}$ is a pair of a node of $P_{\text{JDK1.2}}$ and a subset of protection domains; then the size of $\widehat{P}$ is $O(\|P_{\text{JDK1.2}}\| \cdot 2^{|DOM|})$ where DOM is the set of protection domains. We refer to this construction algorithm as the *domain based construction* of $\widehat{P}$. Practically, the number of protection domains is often smaller than the number of permissions, in which case the domain based construction is more efficient than Construction 5.1.

Below we describe the domain based construction of $\widehat{P}$. In JDK1.2, each method belongs to exactly one protection domain (or for short, domain) and each domain is granted a set of permissions. Let $P_{\text{JDK1.2}} = (NO, IS, IT, TG, CG, P\_BY, PRV)$ be a program in $\Pi_{\text{JDK1.2}}$ and let DOM a finite set of domains. Assume that the domain which a node $n \in NO$ belongs to is given and is denoted by $D\_OF(n)$ $(\in DOM)$. The set of domains which is granted a permission p is also given and is denoted by $GRNT(p)$ $(\subseteq DOM)$. We assume that every node in a method belongs to the same domain, that is,

$$n \overset{TG}{\to} n' \Rightarrow D\_OF(n) = D\_OF(n').$$

(Note that $P\_BY(n) = \{p \mid D\_OF(n) \in GRNT(p)\}$ should hold and thus this model does not conflict with the assumptions in section 5.1.) The access control of JDK1.2 can be achieved by inspecting whether the domain of every node in the stack (precisely, every node which is not a proper ancestor of any privileged node) is granted a specified permission. The domain based construction of a program $\widehat{P} = (\widehat{NO}, \widehat{IS}, \widehat{IT}, \widehat{TG}, \widehat{CG})$ in $\Pi_{\text{check-free}}$ from $P_{\text{JDK1.2}}$ is the same as Construction 5.1, but we let each node of $\widehat{P}$ be augmented by the set of domains of nodes in the stack instead of the set of effective permissions. The differences between this construction and Construction 5.1 are:

(1) $\widehat{NO} = NO \times 2^{DOM}$,

(3) $\widehat{IT} = IT^{\{D\_OF(IT)\}}$,

(4.1) $n_i \overset{CG}{\to} n_j \Rightarrow n_i^{\mathcal{D}_i} \overset{\widehat{CG}}{\longrightarrow} n_j^{\mathcal{D}_j}$ where

$$\mathcal{D}_j = \begin{cases} \mathcal{D}_i \cup \{D\_OF(n_j)\} & \text{if } n_i \notin PRV, \\ \{D\_OF(n_i), D\_OF(n_j)\} & \text{if } n_i \in PRV, \end{cases}$$

(4.3) $n_i \overset{TG}{\to} n_j$, $IS(n_i) = check(JDK(p))$ and $\mathcal{D}_i \subseteq GRNT(p) \Rightarrow n_i^{\mathcal{D}_i} \overset{\widehat{TG}}{\longrightarrow} n_j^{\mathcal{D}_i}$.

The equality between $\widehat{P}$ and $P_{\text{JDK1.2}}$ can be shown in the same way as shown in Lemma 5.1.

Although so far we assumed that DOM, D_OF and $GRNT$ are given, we can derive them from P_BY as follows: Define DOM as the set of the equivalence classes on NO defined by the relation $\sim$ such that $n \sim n' \Leftrightarrow P\_BY(n) = P\_BY(n')$, $D\_OF(n) = \{n' \mid P\_BY(n) = P\_BY(n')\}$ and $GRNT(p) = \{D\_OF(n) \mid p \in P\_BY(n)\}$. Note that the cardinality of DOM in this definition is not larger than that of any given set of protection domains (there exists at most one protection domain which is granted a given subset of permissions in this definition).

By Theorem 5.2, the verification problem for a program $P_{\text{JDK1.2}}$ in $\Pi_{\text{JDK1.2}}$ and a verification property specified by a DFA M_ψ is solvable in $O(\|P_{\text{JDK1.2}}\| \cdot \sharp M_\psi \cdot 2^{|PRM|})$ by Construction 5.1, and in $O(\|P_{\text{JDK1.2}}\| \cdot \sharp M_\psi \cdot 2^{|DOM|})$ by the domain based construction. We can choose one of these algorithms based on whether $|PRM| < |DOM|$.

The results on the complexity of the verification problem shown in sections 3, 4 and 5 are summarized in Table 1.

6. CONCLUSION

In this paper, we analyzed the computational complexity of the verification problem to decide whether a given program which may contain stack inspection satisfies a given security property. Narrowing the gap between the upperbound and the lowerbound of the problem's complexity in three cases in Table 1 is a future study. Applying our model to other types of the verification problems (*e.g.,* liveness verification) is another interesting question.

7. REFERENCES

[1] M. Abadi, M. Burrows, B. Lampson, and G. Plotkin. A calculus for access control in distributed systems. *ACM Trans. on Prog. Lang. and Systems*, 15(4):706–734, 1993.

[2] M. Abadi, C. Fournet, and G. Gonthier. Secure communications processing for distributed languages. In *IEEE Symp. on Security and Privacy*, pages 74–88, 1999.

[3] A. V. Aho. Indexed grammars — an extension of context-free grammars. *Journal of the ACM*, 15(4):647–671, 1968.

[4] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.

Table 1: Complexity of the verification problem

		verification property	
		NFA (RE)	DFA
language in a check node	NFA (RE†)	in double exponential time DEXP-POLY time-hard	DEXP-POLY time-complete
	DFA	in double exponential time PSPACE-hard	in DEXP-POLY time PSPACE-hard
$\Pi_{\text{JDK1.2}}$‡		PSPACE-complete	PTIME
$\Pi_{\text{check-free}}$		PSPACE-complete	PTIME

†RE denotes regular expression.
‡Either the number of permissions or the number of protection domains is assumed to be constant.

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

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

[7] E. A. Emerson. *Temporal and Modal Logic, in Handbook of Theoretical Computer Science, 1023-1024*. Elsevier, 1990.

[8] M. R. Garey and D. S. Johnson. *Computers and Intractability*. W. H. Freeman and Company, 1979.

[9] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going beyond the sandbox: An overview of the new security architecture in the Java™ development kit 1.2. In *USENIX Symp. on Internet Technologies and Systems*, pages 103–112, 1997.

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

[11] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

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

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

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

[15] P. Ørbæk. Can you trust your data? In *TAPSOFT '95, LNCS 915*, pages 575–589, 1995.

[16] D. Volpano and G. Smith. A type-based approach to program security. In *TAPSOFT '97, LNCS 1214*, pages 607–621, 1997.

[17] D. S. Wallach and E. W. Felten. Understanding Java stack inspection. In *IEEE Symp. on Security and Privacy*, pages 52–63, 1998.

Appendix: Detailed Proofs

PROPOSITION 3.2 Let $P = (NO, IS, IT, TG, CG)$ be a program and L_ψ a verification property which satisfies the assumption stated in Theorem 3.1 except that the language

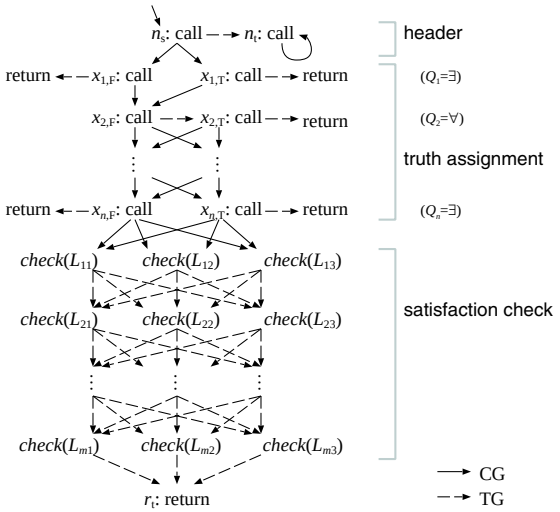

Figure 3: The program P_{Q3SAT} to solve QUANTIFIED 3SAT

L_{ϕ_i} in each $check(L_{\phi_i})$ is specified by a DFA M_{ϕ_i}. The verification problem for P and L_ψ is PSPACE-hard.

PROOF. We transform QUANTIFIED 3-SATISFIABILITY (QUANTIFIED 3SAT) problem to the verification problem. An instance of QUANTIFIED 3SAT is a Boolean formula $F = (Q_1x_1)(Q_2x_2)\dots(Q_nx_n)E$ where E is a conjunction of 3-literal disjunctive clauses involving the variables $x_1, x_2, \dots, x_n$ and each Q_i is either "$\exists$" or "$\forall$."

The program P_{Q3SAT} constructed by the transformation consists of three parts: *header*, *truth assignment*, and *satisfaction check* (Figure 3). A header consists of two nodes n_{s} and n_{t}. A truth assignment consists of $x_{i,\text{F}}$ and $x_{i,\text{T}}$ for $1 \le i \le n$. We consider that "false" is assigned to x_i when the control passes through $x_{i,\text{F}}$ and "true" is assigned to x_i when the control passes through $x_{i,\text{T}}$. We put a call edge between $x_{i-1,V}$ and $x_{i,V'}$ for every $V, V' \in \{\text{F}, \text{T}\}$ if Q_i is "$\exists$" (Figure 4). That is, in this case the program tries assigning one of the truth values to x_i. If Q_i is "$\forall$," we put a call edge from $x_{i-1,V}$ to $x_{i,\text{F}}$ for each $V \in \{\text{F}, \text{T}\}$ and put a transfer edge between $x_{i,\text{F}}$ and $x_{i,\text{T}}$. The program tries assigning each truth value to x_i in this case.

Suppose $E = C_1 \wedge C_2 \wedge \dots \wedge C_m$ and $C_i = u_{i1} \vee u_{i2} \vee u_{i3}$ for $1 \le i \le m$, where u_{ij} is a literal over $\{x_1, \dots, x_n\}$. A satisfaction check consists of $check(L_{ij})$ for $1 \le i \le m$ and

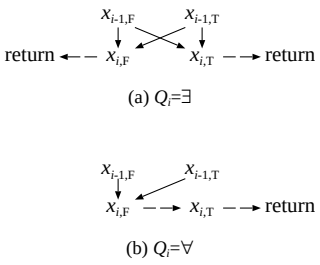

Figure 4: Edges in the truth assignment-part of P_{Q3SAT}

$1 \leq j \leq 3$, where

$$L_{ij} = \begin{cases} n_{\text{s}} NO^{k-1} x_{k,\text{F}} NO^* & \text{if } u_{ij} = \overline{x_k}, \\ n_{\text{s}} NO^{k-1} x_{k,\text{T}} NO^* & \text{if } u_{ij} = x_k. \end{cases}$$

The control can reach the return node r_{t} if the current truth assignment satisfies E.

Therefore, an execution of P_{Q3SAT} reaches the node n_{t} if and only if F is true. That is, $[\![P_{\text{Q3SAT}}]\!] \not\subseteq L_{\text{safe}}[\psi]$ for $L_\psi = (NO - \{n_{\text{t}}\})^*$ if and only if F is true. Since the language L_{ij} can be specified by a DFA M_{ij} such that $\#M_{ij} \leq n+2$, this transformation can be performed in polynomial time. □

LEMMA 4.1 Let $P = (NO, IS, IT, TG, CG)$ be a program in $\Pi_{\text{check-free}}$. For an arbitrary node $n \in NO$ of P,

$\text{CR}(n) = \mathit{True}$ if and only if there exists a node n' such that $IS(n') = \mathit{return}$ and a valid state sequence $n \triangleright \cdots \triangleright n'$,

where a valid state sequence T is a sequence of states satisfying the following condition:

$T = s_1 \triangleright \cdots \triangleright s_k$ such that $s_1, \ldots, s_k \in NO^*$ and $\forall i < k.\ s_i \triangleright s_{i+1}$.

PROOF. The *only if* part can be shown by induction on the application number of inference rules of CR used for deriving $\text{CR}(n) = \mathit{True}$.

The proof of the *if* part is as follows. Suppose that there is a valid state sequence $T_1 = s_1 \triangleright \cdots \triangleright s_l$ where $s_i \in NO^*$ $(1 \leq i \leq l)$, $s_1 = n$, $s_l = n'$, and $IS(n') = \mathit{return}$. This part is proved by induction on l. Note that in the case of $l = 1$, $IS(n) = \mathit{return}$.

- $IS(n) = \mathit{return}$. By inference rule (3), $\text{CR}(n) = \mathit{True}$.
- $IS(n) = \mathit{check}(NO^*)$. There is a node n_2 such that $n \overset{TG}{\rightarrow} n_2$ and $s_2 = n_2$. The state sequence $s_2 \triangleright \cdots \triangleright s_l$ satisfies the condition of this lemma and is shorter than T_1. By the induction hypothesis, $\text{CR}(n_2) = \mathit{True}$. Hence, $\text{CR}(n) = \mathit{True}$ holds by inference rule (4).
- $IS(n) = \mathit{call}$. The valid state sequence T_1 can be written as $T_1 = n \triangleright nm \triangleright ns'_1 \triangleright \cdots \triangleright ns'_k \triangleright nm' \triangleright n_2 \triangleright \cdots \triangleright n'$, where $s'_i \in NO^+$ $(1 \leq i \leq k)$, $IS(m') = \mathit{return}$, $n \overset{CG}{\rightarrow} m$ and $n \overset{TG}{\rightarrow} n_2$. Since valid state sequence $m \triangleright s'_1 \triangleright \cdots \triangleright s'_k \triangleright m'$ obtained by removing the leftmost node n from every state in subsequence $nm \triangleright ns'_1 \triangleright \cdots \triangleright ns'_k \triangleright nm'$ of T_1 satisfies the condition of this lemma and is shorter than T_1, $\text{CR}(m) = \mathit{True}$ holds by the induction hypothesis. Similarly, for subsequence $n_2 \triangleright \cdots \triangleright n'$ of T_1, $\text{CR}(n_2) = \mathit{True}$ holds. Hence, we obtain $\text{CR}(n) = \mathit{True}$ by inference rule (2).

□

THEOREM 4.2 For a program $P = (NO, IS, IT, TG, CG)$ in $\Pi_{\text{check-free}}$, $L(G_{P,\text{S}}) = [P]$.

PROOF. It suffices to show that for every $s \in NO^*$, $S \overset{*}{\rightarrow}_{G_{P,\text{S}}} s$ *if and only if* $IT \triangleright \cdots \triangleright s \in [\![P]\!]$ holds.

The *only if* part is shown by induction on the sum of application numbers of production rules (7)–(9).

(basis) If the derivation $S \overset{*}{\rightarrow}_{G_{P,\text{S}}} s$ is obtained without applying the production rules (7)–(9), then

$$\begin{array}{lll} S & \rightarrow_{G_{P,\text{S}}} N_1 & \text{by (5)} \\ & \rightarrow_{G_{P,\text{S}}} n_1 = s. & \text{by (6)} \end{array}$$

Clearly, $n_1 = IT \in [\![P]\!]$ holds.

(inductive step) We consider the case that the last production among (7)–(9) applied in the derivation is (8). The proof of the other cases is similar. Suppose there is a derivation of the following form.

$$\begin{array}{lll} S & \overset{*}{\rightarrow}_{G_{P,\text{S}}} n_1 \cdots n_l N_i & \\ & \rightarrow_{G_{P,\text{S}}} n_1 \cdots n_l N_k & \text{by (8)} \\ & \rightarrow_{G_{P,\text{S}}} n_1 \cdots n_l n_k. & \text{by (6)} \end{array} \tag{12}$$

Then, the following derivation also exists.

$$\begin{array}{lll} S & \overset{*}{\rightarrow}_{G_{P,\text{S}}} n_1 \cdots n_l N_i & \\ & \rightarrow_{G_{P,\text{S}}} n_1 \cdots n_l n_i. & \text{by (6)} \end{array}$$

Since the application number of production rules (7)–(9) in the above derivation is less than that of the derivation (12), we can use the inductive hypothesis and obtain

$$IT \triangleright \cdots \triangleright n_1 \cdots n_l n_i \in [\![P]\!]. \tag{13}$$

On the other hand, by the existence of the production (8) $(N_i \rightarrow N_k)$, we can see that $IS(n_i) = \mathit{call}$, $\exists n_j.\ n_i \overset{CG}{\rightarrow} n_j$, $\text{CR}(n_j) = \mathit{True}$ and $n_i \overset{TG}{\rightarrow} n_k$. Therefore, by Lemma 4.1, there is a node n_r such that $IS(n_r) = \mathit{return}$ and a valid state sequence:

$$n_j \triangleright \cdots \triangleright n_r. \tag{14}$$

From (13), (14) and the fact that $n_i \overset{CG}{\rightarrow} n_j$, $IT \triangleright \cdots \triangleright n_1 \cdots n_l n_i \triangleright n_1 \cdots n_l n_i n_j \triangleright \cdots \triangleright n_1 \cdots n_l n_i n_r \in [\![P]\!]$. Since $n_i \overset{TG}{\rightarrow} n_k$, $IT \triangleright \cdots \triangleright n_1 \cdots n_l n_k \in [\![P]\!]$.

The proof of *if* part is similar and can be shown by induction on the length of the trace.

□

PROPOSITION 4.4 Let $P = (NO, IS, IT, TG, CG)$ be a program in $\Pi_{\text{check-free}}$. The verification problem for P and a verification property specified by an NFA M_ψ is PSPACE-complete.

PROOF. Both of the following two problems are known to be PSPACE-complete[8].

FINITE AUTOMATON INEQUIVALENCE

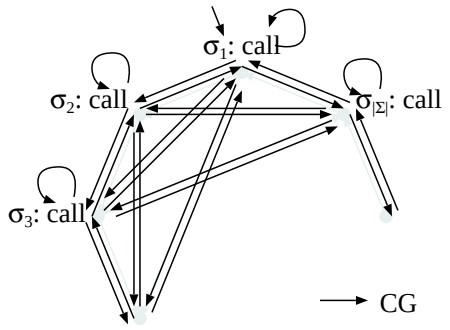

Figure 5: The program P_{Σ^*}

Instance: Two NFA M_1 and M_2 having the same input alphabet Σ.
Question: $L(M_1) \neq L(M_2)$?

Regular Expression Non-Universality
Instance: A regular expression E over a finite alphabet Σ.
Question: For a regular expression E, let $L(E)$ denote the language expressed by E. Then, $L(E) \neq \Sigma^*$?

PSPACE-solvability of the verification problem can be shown by transforming the problem to the Finite Automaton Inequivalence problem. We can construct an NFA $M_{P,\mathrm{S}}$ such that $[P] = L(M_{P,\mathrm{S}})$ and $\#M_{P,\mathrm{S}} = O(|NO|)$. We can also construct an NFA $M_{P,\mathrm{S},\psi}$ such that $L(M_{P,\mathrm{S},\psi}) = L(M_{P,\mathrm{S}}) \cap L(M_\psi)$ and $\#M_{P,\mathrm{S},\psi} = \#M_{P,\mathrm{S}} \cdot \#M_\psi$. This construction of $M_{P,\mathrm{S}}$ and $M_{P,\mathrm{S},\psi}$ completes the transformation since $[P] \not\subseteq L(M_\psi)$ iff $[P] \neq [P] \cap L(M_\psi)$ iff $L(M_{P,\mathrm{S}}) \neq L(M_{P,\mathrm{S},\psi})$.

PSPACE-hardness of the verification problem can be shown by transforming Regular Expression Non-Universality to the problem. First we construct the program $P_{\Sigma^*} = (NO, IS, IT, TG, CG)$ in Figure 5, where the set NO of nodes is the alphabet Σ and the entry point IT is an arbitrary node $\sigma_1 \in \Sigma$. P_{Σ^*} is similar to the complete graph with $|\Sigma|$ nodes and obviously $[P_{\Sigma^*}] = \sigma_1\Sigma^*$. Second we construct an NFA M_ψ such that $L(M_\psi) = L(\sigma_1 \cdot E)$ from the regular expression E. This construction of M_ψ can be performed in polynomial time[11]. The construction of P_{Σ^*} and M_ψ completes the transformation since $[P_{\Sigma^*}] \not\subseteq L(M_\psi)$ iff $\sigma_1\Sigma^* \not\subseteq L(\sigma_1 \cdot E)$ iff $\Sigma^* \neq L(E)$. □

Lemma 5.1 For a program $P_{\mathrm{JDK1.2}} = (NO, IS, IT, TG, CG, P\_BY, PRV)$, let $\widehat{P} = (\widehat{NO}, \widehat{IS}, \widehat{IT}, \widehat{TG}, \widehat{CG})$ be the program obtained from $P_{\mathrm{JDK1.2}}$ by Construction 5.1. Let us define the homomorphism $h : (\widehat{NO} \cup \{\triangleright\})^* \to (NO \cup \{\triangleright\})^*$ as $h(n^{\mathcal{P}}) = n$ for $n^{\mathcal{P}} \in \widehat{NO}$ and $h(\triangleright) = \triangleright$. Then, $[\![P_{\mathrm{JDK1.2}}]\!] = h([\![\widehat{P}]\!])$.

Proof. It suffices to show that $IT \triangleright \cdots \triangleright n_1 n_2 \cdots n_k \in [\![P_{\mathrm{JDK1.2}}]\!]$ *if and only if* $\widehat{IT} \triangleright \cdots \triangleright n_1^{\mathcal{P}_1} n_2^{\mathcal{P}_2} \cdots n_k^{\mathcal{P}_k} \in [\![\widehat{P}]\!]$ where $\mathcal{P}_1 = P\_BY(n_1)$ and

$$\mathcal{P}_i = \begin{cases} \mathcal{P}_{i-1} \cap \mathrm{P\_BY}(n_i) & n_{i-1} \notin PRV \\ P\_BY(n_{i-1}) \cap P\_BY(n_i) & n_{i-1} \in PRV \end{cases}$$

for each $1 < i \leq k$.
The *only if* part is shown by induction on the length of a trace of $P_{\mathrm{JDK1.2}}$.

(**basis**) Clearly, $\langle IT \rangle \in [\![P_{\mathrm{JDK1.2}}]\!]$ and $\langle IT^{P\text{-}BY(IT)} \rangle \in [\![\widehat{P}]\!]$.
(**inductive step**) There are three cases to consider.

Case 1: $IT \triangleright \cdots \triangleright n_1 \cdots n_{k-1} \triangleright n_1 \cdots n_{k-1} n_k \in [\![P_{\mathrm{JDK1.2}}]\!]$, $IS(n_{k-1}) = call$ and $n_{k-1} \overset{CG}{\to} n_k$.

Case 2: $IT \triangleright \cdots \triangleright n_1 \cdots n_{k-1} n_k n_{k+1} \triangleright n_1 \cdots n_{k-1} n'_k \in [\![P_{\mathrm{JDK1.2}}]\!]$, $IS(n_{k+1}) = return$ and $n_k \overset{TG}{\to} n'_k$.

Case 3: $IT \triangleright \cdots \triangleright n_1 \cdots n_{k-1} n_k \triangleright n_1 \cdots n_{k-1} n'_k \in [\![P_{\mathrm{JDK1.2}}]\!]$, $IS(n_k) = check(JDK(p))$ and $n_k \overset{TG}{\to} n'_k$.

We will give a proof for case 3, the most difficult case. By the induction hypothesis on $IT \triangleright \cdots \triangleright n_1 \cdots n_{k-1} n_k$, we can see that $\widehat{IT} \triangleright \cdots \triangleright n_1^{\mathcal{P}_1} \cdots n_k^{\mathcal{P}_k} \in [\![\widehat{P}]\!]$ where $\mathcal{P}_1 = P\_BY(n_1)$ and

$$\mathcal{P}_i = \begin{cases} \mathcal{P}_{i-1} \cap P\_BY(n_i) & n_{i-1} \notin PRV, \\ P\_BY(n_{i-1}) \cap P\_BY(n_i) & n_{i-1} \in PRV \end{cases} \tag{15}$$

for $1 < i \leq k$. Hence, it suffices to show that

$$n_1^{\mathcal{P}_1} \cdots n_{k-1}^{\mathcal{P}_{k-1}} n_k^{\mathcal{P}_k} \triangleright n_1^{\mathcal{P}_1} \cdots n_{k-1}^{\mathcal{P}_{k-1}} {n'_k}^{\mathcal{P}_k}. \tag{16}$$

and

$$\mathcal{P}_k = \begin{cases} \mathcal{P}_{k-1} \cap P\_BY(n_{k'}) & n_{k-1} \notin PRV, \\ P\_BY(n_{k-1}) \cap P\_BY(n_{k'}) & n_{k-1} \in PRV \end{cases} \tag{17}$$

First, we prove (17). Since $n_k \overset{TG}{\to} n'_k$, we can see $P\_BY(n_k) = P\_BY(n_{k'})$ by (10). Letting $i = k$ in (15), we obtain (17). Next, we show (16) holds. It follows from $n_1 \cdots n_{k-1} n_k \triangleright n_1 \cdots n_{k-1} n'_k$ and $IS(n_k) = check(JDK(p))$ that

$$n_1 \cdots n_{k-1} n_k \in check(JDK(p)) = (NO^*(PRV \cap \mathcal{N}(p)) \cup \epsilon)(\mathcal{N}(p))^*. \tag{18}$$

There are two cases.

- Assume that $n_i \notin PRV$ for $1 \leq i < k$. By (15),

$$\mathcal{P}_k = \bigcap_{1 \leq i \leq k} P\_BY(n_i). \tag{19}$$

  By (18), $n_i \in \mathcal{N}(p)$ for $1 \leq i \leq k$, which implies $p \in \bigcap_{1 \leq i \leq k} P\_BY(n_i)$ by the definition of $\mathcal{N}(p)$. Hence, $p \in \mathcal{P}_k$ by (19).

- Assume that there exists a node $n_j \in PRV$ $(1 \leq j < k)$ and $n_i \notin PRV$ for $j < i < k$. By (15),

$$\mathcal{P}_k = \bigcap_{j \leq i \leq k} P\_BY(n_i). \tag{20}$$

  By (18), $n_i \in \mathcal{N}(p)$ $(j \leq i \leq k)$ and thus $p \in \bigcap_{j \leq i \leq k} P\_BY(n_i) = \mathcal{P}_k$ by the definition of $\mathcal{N}(p)$ and (20).

In either case, $p \in \mathcal{P}_k$ holds and $n_k^{\mathcal{P}_k} \overset{\widehat{TG}}{\longrightarrow} {n'_k}^{\mathcal{P}_k}$ is constructed by (4.3) of Construction 5.1. Therefore, (16) holds.

The *if* part can be shown by induction on the length of a trace of $\widehat{P}$.
Therefore, $[\![P_{\mathrm{JDK1.2}}]\!] = h([\![\widehat{P}]\!])$ holds by the definition of h. □