Technical Report

Decidability and Complexity of the Security Verification Problem for Programs with Stack Inspection

Naoya Nitta, Satoshi Ikada, Yoshiaki Takata and Hiroyuki Seki

{naoya-n,y-takata,seki}@is.aist-nara.ac.jp

March 9, 2001

Graduate School of Information Science
Nara Institute of Science and Technology

Abstract

Java development kit 1.2 provides a runtime access control mechanism which inspects a control stack to examine whether the program has appropriate access permissions. For such a programming language, it is desirable to guarantee that each execution of a program satisfies required security properties. Jensen et al. introduced a verification problem of deciding for a given program P and a given security property ψ written in a temporal logic formula, whether every reachable state of P satisfies ψ. They showed that the problem is decidable for the class of programs which do not contain mutual recursion. In this paper, we show that the set of state sequences of a program is always an indexed language and consequently the verification problem is decidable. Our result is stronger than Jensen's in that a security property can be specified by a regular language, whose expressive power is stronger than temporal logic, and in that a program can contain mutual recursion. Furthermore, we discuss the computational complexity of the problem. For example, the problem is shown to be deterministic DEXP-POLY time complete if each check statement in P is specified by a NFA and a security property is specified by a DFA.

Contents

1 Introduction

As a world wide computer network grows rapidly, providing a well-defined access control mechanism for network application systems becomes more important. For example, consider an electronic commerce application. Since a number of anonymous external mobile processes as well as local ones can be executed in a user's site, an appropriate access control is needed to prevent a malicious external process from accessing secret local resources. Java™ sandbox model provides a security protection mechanism for such a distributed computational environment. However, the sandbox model lacks flexibility since it imposes too strong restriction on the behavior of an external process which may access local resources.

For this purpose, Java development kit 1.2 (JDK 1.2) provides a security check method called *checkPermission* [GMPS97]. As is the case with other programming languages, the Java execution environment has a runtime control stack, which contains a series of frames for invoked methods. A frame for a method m involves information on access permissions assigned to m as well as actual input parameters, values of local variables and the return address. When the control reaches an invocation of *checkPermission*($Perm$) method where the actual parameter $Perm$ is a particular access permission, the current contents of the stack are inspected to examine whether the currently activated method m and all the ancestor methods of m have permission $Perm$ until a privileged method or the stack bottom is encountered. If all the inspected methods have $Perm$, then the execution continues. Otherwise, the execution is aborted.

Appropriate check statements should be carefully placed in the local methods which directly or indirectly access secret local resources. Consider the following simple example. Let *write* be a local method which directly updates a customer's bank account and *credit* be a method which is called by a customer's method and updates the customer's bank account by calling *write* method. Suppose the following situation. Both *write* and *credit* methods have write permission P_{Write} and every method of a valid customer has permission $P_{Customer}$. The system manager places *checkPermission*(P_{Write}) at the beginning of *write* method. The system manager makes *credit* method privileged, but there is no check statement in *credit* method. If a malicious user's method which does not have $P_{Customer}$ calls *credit*, then *credit* successively calls *write*. Since *credit* is priv-

ileged, $checkPermission(P_{Write})$ in *write* method succeeds and an illegal update may occur. Let ψ be the property that 'if the control reaches *write* method, then the control has passed through only methods which have $P_{Customer}$ or P_{Write},' which the system is expected to satisfy. Let us call such a property ψ as a global security property. The above mentioned execution does not satisfy ψ. In this particular example, if $checkPermission(P_{Customer})$ is placed in *credit* method, then every execution satisfies ψ. However, ensuring that a program satisfies such a global security property by hand becomes difficult when the whole program is large and complicated. Therefore, an automatic verification method is needed which verifies that every execution of P satisfies ψ for a given program P and a global security property ψ.

In [JMT99], a pioneering paper in the formal verification of a program with stack inspection, the verification problem is defined as follows:

- A program is modeled as a directed graph called a flow graph. Since a flow graph does not have a data part, the contents of a control stack can be represented as a sequence of nodes, each of which is a program point where a method invocation has occurred. A trace is a finite sequence $s_0, s_1, \ldots, s_k$ of stacks (also called states), where s_0 is the initial state and $s_{i+1}(0 \leq i < k)$ is a state reachable from s_i by a unit step execution.

- A local security check statement has the form of $check(\phi)$ where ϕ is an LTL (linear temporal logic) formula [CGP99]. The execution of $check(\phi)$ at a state s succeeds if and only if s, interpreted as a Kripke structure, satisfies ϕ. A global security property ψ to be verified is also represented by an LTL formula.

- The verification problem for a given program (a flow graph) P and a global security property ψ is to decide whether every state in every trace in P satisfies ψ.

In this formulation, [JMT99] presents a verification method using model checking [CGP99] of LTL formulas. In [JMT99], it is also shown that if a given program does not contain mutual recursion, then the verification problem is decidable.

In this paper, we define a program model and a verification problem using regular languages instead of LTL formulas for specifying both a local check statement

and a global security property. Since the class of regular languages is known to properly include the class of languages represented by LTL formulas [E90], this formulation is an extension of the one in [JMT99]. Furthermore, we show that the verification problem is decidable in general even if a program *does* contain mutual recursion, which is a proper improvement of the result of [JMT99]. The outline of the proof of the decidability is as follows:

(i) For every program P, the set of traces in P is shown to be an indexed language. An indexed language is a language which can be generated by an indexed grammar [A68], which is an extension of a context-free grammar.

(ii) The decidability of the verification problem follows from the fact that the class of indexed languages is closed under intersection with regular languages and the emptiness problem for the class of indexed languages is decidable.

In this paper, we also analyze the computational complexity of the verification problem. The complexity of the problem generally depends on the representations of regular languages for specifying a local check statement (called a check node in this paper) and a global security property (called a verification property). The results of this paper are summarized in table 1 in section 5. For example, the problem is shown to be deterministic DEXP-POLY time complete if each check node is specified by a NFA and a verification property is specified by a DFA. As a special case, we show that the problem is solvable in polynomial time if a program contains no check node and a verification property is specified by a DFA.

The rest of the paper is organized as follows. In section 2, the program model is presented based on [JMT99]. The verification problem is defined and a simple example is presented in section 3. In section 4, we show that the verification problem is decidable. Section 5 discusses the computational complexity of the problem.

Related Works In [WF98], a general notion of stack inspection which is independent of a particular programming language is introduced by using ABLP logic [ABLP93]. The authors also provide a sufficient condition for a local security check to succeed. However, [WF98] does not mention the verification of a global security property.

[D76] and [DD77] are the pioneering works which proposed a systematic method of analyzing security information flow based on a lattice model of security classes. Subsequently, there have been a number of works which generalize Denning's method [BBM94, O95, VS97, HR98, LR98]. For example, [O95] proposes an analysis algorithm and proves its correctness using abstract interpretation. Various type systems have been also proposed in which if a given program is well-typed then the program has the *noninterference* property such that it does not cause undesirable security information flow (*e.g.*, [VS97, HR98, LR98]). A structure of security classes such as {topsecret, trusted, untrusted} is usually modeled as a finite lattice. [ML98] proposes a fine grained model of security classes called decentralized labels for a distributed access control. All of the above mentioned studies are basically concerned with information flow analysis to ensure that high level secret information does not flow into an insecret storage. In contrast, [JMT99] and this paper discuss the problem of deciding whether a program possesses an arbitrarily given global security property provided that the program passes local security checks.

Security verification in a distributed system has been extensively studied by using a process algebra called spi calculus and its extensions in [AFG99] and its companion papers. [SV98] showed that their type system in [VS97] is no longer correct in a distributed environment and presented a new type system for a multi-threaded language.

In the field of data engineering, access control and information flow control issues have been also extensively studied for a distributed and object-oriented environment (see [BW94]). For example, [SBCJ97] presents an information flow control algorithm which blocks illegal information flows in object-oriented databases. However, their algorithm does not perform semantic analysis inside a method. Semantic analysis of security flows or security verification against inference attacks in object-oriented databases are discussed in [T96, IMI99].

2 Program Model

2.1 Flow Graph

Following [JMT99], we model a program as a directed graph called a *flow graph*. Each node of a flow graph corresponds to a location of the program (*program point*). A statement which performs runtime check of access permission such as *checkPermission* in JDK 1.2 is incorporated into the model. A check statement examines whether the current state of the executed program satisfies the property specified in the statement (*e.g.*, whether the executed method (procedure) and all of the ancestor methods have access permission of specific resources). If the property is satisfied, the program continues its execution. Otherwise, the execution is aborted. In the model, such a check statement is also represented as a node.

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. If the execution starting from n_2 has been terminated by a return node explained later, then the control is passed to a node n_3 which satisfies $n_1 \stackrel{TG}{\rightarrow} n_3$.

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. 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 a language L_ϕ are defined in section 2.3.

Conditionals such as *if* statements and *while* statements are substituted for nondeterministic statements. For example, consider the following sequence of statements: $m_1()$; if ... then $m_2()$ else $m_3()$. In a flow graph, there will be two TGs $n_1 \stackrel{TG}{\rightarrow} n_2$ and $n_1 \stackrel{TG}{\rightarrow} n_3$, where n_1, n_2, and n_3 represent $m_1()$, $m_2()$, and $m_3()$, respectively. Transformation methods from an object-oriented program into a flow graph using data flow analysis or type inference have been studied (*e.g.*, [PS94]).

Note that a flow graph does not always represent the exact behaviors of an original program. More specifically, if an ordinal imperative program P_0 is modeled as a flow graph P, then every execution sequence (trace) of P_0 is also a trace of P, but not vice versa. The reasons why we do such an "approximation" are as follows:

- Most of decision problems for imperative programs which contain either conditional statements and procedure calls or *while* statements are undecidable. Therefore, static program analyses such as type inference and abstract interpretation use an approximatation such as the one described here.

- If a flow graph P of an original program P_0 satisfies a safety property discussed in this paper, then it is guaranteed that P_0 also satisfies the property. (This is not true for liveness properties.)

2.2 State of Program

Each state of an ordinary imperative program can be represented by a tuple of a current program point (or a continuation), contents of global variables, and a

runtime control stack (shortly, stack).

For each invocation to a method m, a frame f_m is allocated and pushed onto the stack. A frame f_m contains actual values of the input arguments of m, values of local variables, the return address, and other information on access permissions which m has. In the flow graph model, however, values of variables and arguments are abstracted away. Hence, a frame degenerates into a return address, i.e, a node. If, in addition, the current program point is also kept on top of the stack, then each state of a program can be represented as a stack, that is, a sequence of nodes as follows:

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 entry point IT of P. The state immediately after a method call is the state (stack) obtained by pushing the node of the callee onto the current state. If the top element (current program point) of the stack is a node n_1 and $n_1 \stackrel{TG}{\rightarrow} n_2$ holds, then the next state can be the state obtained by replacing the top element of the stack n_1 with n_2. 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. For example, a stack $n_1 : n_2 : n_3$ indicates that first the method including the program point n_2 has been called from n_1, the control has reached n_2, the method including program point n_3 has been called from n_2, and the current program point is n_3. If the top element of the stack is n_1 and n_1 is a return statement ($IS(n_1) = return$), then the next state is the state obtained by popping n_1 from the stack and replacing the top node n_2 of the resultant stack with a node n_3 such that $n_1 \stackrel{TG}{\rightarrow} n_3$.

2.3 Trace

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 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 ($\in NO$).

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

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

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

□

For a program P, a *trace* in 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 in 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 (i.e., $L_\phi \subseteq NO^*$). 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 $check(L_\phi)$, the execution is continued if and only if the current state belongs to L_ϕ.

2.4 Security Property in Check Node

A language L_ϕ in a node $check(L_\phi)$ is a regular language over NO. 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 often use a regular expression until section 5.

Example 2.1 (Java stack inspection in JDK 1.2)
A method invocation *checkPermission*($Perm$) succeeds if

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

Let $Priv$ be the set of privileged nodes and ϕ be the set of nodes which have permission $Perm$. $checkPermission(Perm)$ can be represented as the check node $check(JDK(\phi))$ where:

$$JDK(\phi) = (NO^*(Priv \cap \phi) \cup \epsilon)\phi^* \tag{1}$$

and ϵ is the empty sequence. □

3 The Verification Problem

3.1 Definition of the Problem

In this section, we define a verification problem, which is a generalization of the one in [JMT99]. Each program is required to satisfy a certain global security property such as 'local resource r never be read out by any (malicious) method which does not have permission *Perm*.' Intuitively, the 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, which is called a *verification property*.

Example 3.1 (A critical section) [JMT99] Let *Critical* be the set of nodes which belong to a critical section and *Manager* be the set of nodes which have permission for managers.

$$CS = (\overline{Critical})^* Manager\ NO^*$$

means that the control of the program must not reach the critical section before it reaches the program point which has permission for managers. □

Let $L_{\mathrm{safe}}[\psi]$ be the set of state sequences α such that every state in α belongs to L_ψ. $L_{\mathrm{safe}}[\psi]$ can be represented as $L_{\mathrm{safe}}[\psi] = (L_\psi \triangleright)^* L_\psi$. 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: $[\![P]\!] \subseteq L_{\mathrm{safe}}[\psi]$?

3.2 An example

In this subsection, we show an example of the verification problem for electronic commerce. As shown in Theorem 4.7, the verification problem is decidable in our framework while this instance cannot be directly solved by the verification algorithm in [JMT99] since the program of this instance contains mutual recursion.

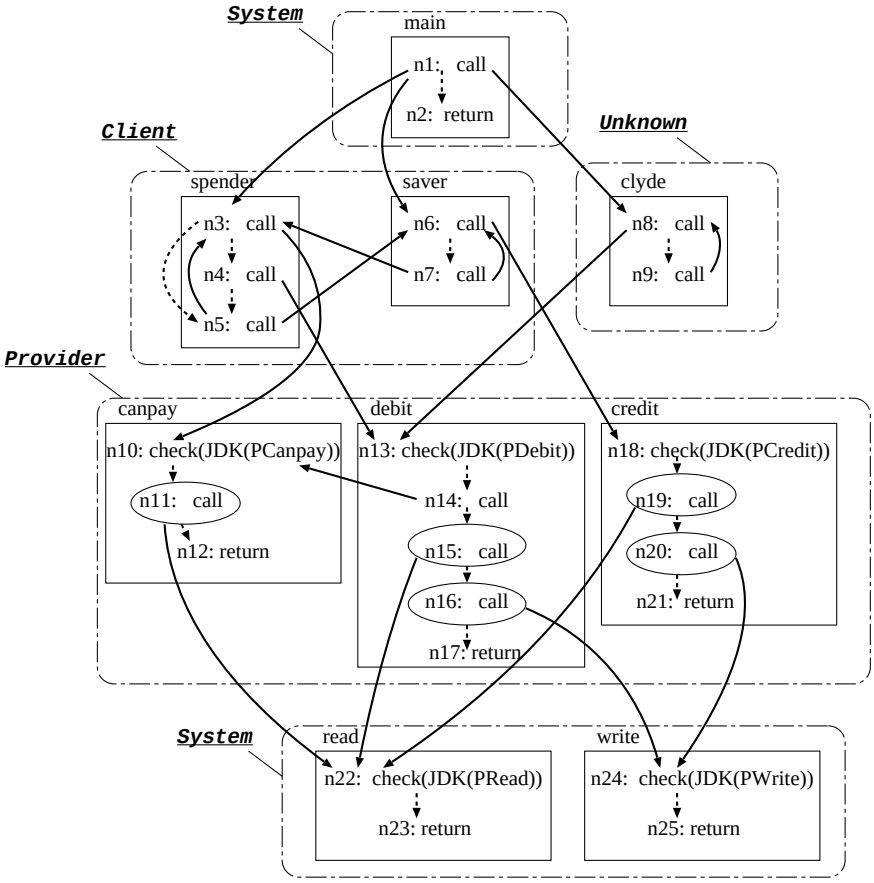


Figure 1: The flow graph of the example

Consider an on-line banking system which serves its clients with methods for spending and saving some 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 *canpay*, *debit*, and *credit* methods, which invoke *read* and/or *write* methods. The flow graph P of this banking system is shown in Figure 1. In the figure, a solid arrow represents a call edge and a dotted arrow represents a transfer edge. Let $NO = \{n_i \mid 1 \leq i \leq 25\}$. Protection domains are represented by the following sets of nodes $D_{Client} = \{n_3, n_4, n_5, n_6, n_7\}$, $D_{Provider} = \{n_{10}, n_{11}, \ldots, n_{21}\}$, $D_{System} = \{n_1, n_2, n_{22}, n_{23}, n_{24}, n_{25}\}$, and $D_{Unknown} = \{n_8, n_9\}$.

Let $Priv = \{n_{11}, n_{15}, n_{16}, n_{19}, n_{20}\}$ be the set of nodes which are given a privilege and let $P_{Debit} = D_{Client} \cup D_{Provider} \cup D_{System}$ be the set of nodes which are given debit permission. Also let $P_{Canpay} = P_{Credit} = D_{Client} \cup D_{Provider} \cup D_{System}$ and $P_{Read} = P_{Write} = D_{Provider} \cup D_{System}$. Then, the property $JDK(P_{Debit})$ specified in node n_{13} is represented by the following regular expression:

$$JDK(P_{Debit}) = (NO^* \ (Priv \cap P_{Debit}) \cup \epsilon)(P_{Debit})^*$$

by (1) in section 2.3. The property $JDK(P_{Canpay})$ specified in node n_{10}, $JDK(P_{Credit})$ in n_{18}, $JDK(P_{Read})$ in n_{22}, and $JDK(P_{Write})$ in n_{24} are represented by the following regular expressions, respectively:

$$\begin{aligned}
JDK(P_{Canpay}) &= (NO^* \ (Priv \cap P_{Canpay}) \cup \epsilon)(P_{Canpay})^* \\
JDK(P_{Credit}) &= (NO^* \ (Priv \cap P_{Credit}) \cup \epsilon)(P_{Credit})^* \\
JDK(P_{Read}) &= (NO^* \ (Priv \cap P_{Read}) \cup \epsilon)(P_{Read})^* \\
JDK(P_{Write}) &= (NO^* \ (Priv \cap P_{Write}) \cup \epsilon)(P_{Write})^*
\end{aligned}$$

where:

$$\begin{aligned}
P_{Canpay} &= D_{Client} \cup D_{Provider} \cup D_{System} \\
P_{Credit} &= D_{Client} \cup D_{Provider} \cup D_{System} \\
P_{Read} &= D_{Provider} \cup D_{System} \\
P_{Write} &= D_{Provider} \cup D_{System}.
\end{aligned}$$

Consider the following two sequences:

$$\begin{aligned}
\alpha_1 = \ & n_1 \triangleright n_1 n_3 \triangleright \underline{n_1 n_3 n_{10}} \triangleright n_1 n_3 n_{11} \triangleright \underline{n_1 n_3 n_{11} n_{22}} \triangleright n_1 n_3 n_{11} n_{23} \triangleright n_1 n_3 n_{12} \\
& \triangleright n_1 n_4 \triangleright \underline{n_1 n_4 n_{13}} \triangleright n_1 n_4 n_{14} \triangleright \underline{n_1 n_4 n_{14} n_{10}} \triangleright n_1 n_4 n_{14} n_{11} \triangleright \underline{n_1 n_4 n_{14} n_{11} n_{22}} \\
& \triangleright n_1 n_4 n_{14} n_{11} n_{23} \triangleright n_1 n_4 n_{14} n_{12} \triangleright n_1 n_4 n_{15} \triangleright \underline{n_1 n_4 n_{15} n_{22}} \triangleright n_1 n_4 n_{15} n_{23} \\
& \triangleright n_1 n_4 n_{16} \triangleright \underline{n_1 n_4 n_{16} n_{24}} \triangleright n_1 n_4 n_{16} n_{25} \triangleright n_1 n_4 n_{17} \triangleright n_1 n_5, \\
\alpha_2 = \ & n_1 \triangleright n_1 n_8 \triangleright \underline{n_1 n_8 n_{13}} \triangleright n_1 n_8 n_{14}.
\end{aligned}$$

First, we consider α_1. Through the entire execution of α_1, check nodes have been executed seven 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, $n_1 n_8 n_{13} \notin JDK(P_{Debit})$ since $n_8 \notin P_{Debit}$, and hence, $\alpha_2 \notin [\![P]\!]$.

Let $L_\psi = ((\overline{E_{Write}})^* \cup (P_{Debit})^* E_{Write} (\overline{E_{Write}})^*) \cap ((\overline{E_{Read}})^* \cup (P_{Canpay})^* E_{Read} (\overline{E_{Read}})^*)$ be the verification property, where:

$$\begin{aligned} E_{Read} &= \{n_{22}, n_{23}\} \\ E_{Write} &= \{n_{24}, n_{25}\}. \end{aligned}$$

Then, $L_{\text{safe}}[\psi] = (L_\psi \triangleright)^* L_\psi$ means that if the control reaches E_{Read}/E_{Write} successfully, then the control has passed through only nodes in P_{Debit}/P_{Canpay}. In this particular example, $[\![P]\!] \subseteq L_{\text{safe}}[\psi]$ holds, that is, program P satisfies L_ψ.

4 Decidability of the Verification Problem

In this section, we present our approach to the verification problem. We will show that for each program P, the set $[\![P]\!]$ of traces in P can be generated by an indexed grammar. The generative capacity of indexed grammars (IGs)[A68] is stronger than that of context-free grammars (CFGs) while IGs inherit good mathematical properties from CFGs. Using these properties, we prove the decidability of the verification problem.

As stated in the following lemma, the set of traces can not always be generated by a CFG.

Lemma 4.1 *There exists a program P such that the set $[\![P]\!]$ of traces is not a context-free language (CFL).*

Proof. Let $P = (NO, IS, IT, TG, CG)$ be the program where $NO = \{n\}$, $IT = n$, there is no transfer edge in P, and there is only one call edge $n \stackrel{CG}{\rightarrow} n$. The set of traces in P is as follows.

$$[\![P]\!] = \{n,\ n \triangleright nn,\ n \triangleright nn \triangleright nnn, \cdots\}.$$

Let h be the homomorphism defined by $h(n) = n$ and $h(\triangleright) = \epsilon$. Then we obtain:

$$h([\![P]\!]) = \{n,\ nnn,\ nnnnnn,\ nnnnnnnnnn, \cdots\} = \{n^{i(i+1)/2} \mid i \geq 1\}.$$

It is easy to prove that $h([\![P]\!])$ is not a CFL using the pumping lemma for CFLs. Since the class of CFLs is closed under homomorphism, $[\![P]\!]$ is not a CFL. □

The above lemma implies that a class of grammars of which generative capacity is stronger than CFGs is needed to generate the set of traces. The outline of this section is as follows.

(a) A trace in a program is a sequence of stacks, which are sequences of nodes. A state transition does not alter the nodes other than the top and the next to the top node in a stack (see section 2.3) at a time. That is, if $s_i \triangleright s_{i+1}$, then there are $\alpha, \beta, \gamma \in NO^*$ with $\mid \beta \mid \leq 2$, $\mid \gamma \mid \leq 2$ such that s and s' can be written as $s = \alpha\ \beta$ and $s' = \alpha\ \gamma$, respectively. As explained in section

4.1, an indexed grammar can generate a sequence which is controlled by such a stack-like data structure.

A sequence of states is called an unchecked trace if the sequence becomes a trace by assuming that every possible local check node succeeds. More precisely, a sequence α of states is an *unchecked trace* in a program P if α is a trace in P when the third inference rule in Definition 2.1 is replaced with

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

For a given program P, we will define an indexed grammar $G_{P,\mathrm{T}}$ which generates the set of unchecked traces in P in section 4.2.

(b) An unchecked trace α is also a trace if α satisfies the properties specified in check nodes in α. That is, if the property specified in a check statement holds at the current state, then the execution continues; otherwise, the execution should be aborted at the state. For this reason, in section 4.3, we will also define the regular language $L_{P,\mathrm{C}}$ of state sequences in which any pair of states can be adjacent to each other as long as those states satisfy the property (belong to the language) specified in check nodes. Finally, we will show $[\![P]\!] = L(G_{P,\mathrm{T}}) \cap L_{P,\mathrm{C}}$ in section 4.4.

4.1 Indexed Grammar

Indexed grammar[A68] is an extension of CFG. An index grammar (IG) is a 5-tuple $G = (N, T, I, R, S)$ where:

(a) N is a finite set of *nonterminal symbols*,

(b) T is a finite set of *terminal symbols*,

(c) I is a finite set of *index symbols*,

(d) $S \in N$ is the *start symbol*, and

(e) R is a finite set of *productions* of one of the following forms:

(Type 1) $A \rightarrow \alpha$

(Type 2) $A \rightarrow Bf$

(Type 3) $Af \rightarrow \alpha$,

where A, $B \in N$, $f \in I$, and $\alpha \in (N \cup T)^*$.

A nonterminal symbol, a terminal symbol and an index symbol are abbreviated as a nonterminal, a terminal, and an index, respectively. A derivation in IG is similar to a derivation in CFG except that IG has operations on index sequences. The derivation relation $\underset{G}{\rightarrow}$ on $(NI^* \cup T)^*$ is defined as the least relation which satisfies the following conditions (1), (2) and (3). In the following, let β, $\gamma \in (NI^* \cup T)^*$, $\xi \in I^*$ and $X_i \in N \cup T$.

(1) Let $A \rightarrow X_1 X_2 \cdots X_k \in R$ be a Type 1 production. Then,

$$\beta A \xi \gamma \underset{G}{\rightarrow} \beta X_1 \xi_1 X_2 \xi_2 \cdots X_k \xi_k \gamma$$

where if $X_i \in N$ then $\xi_i = \xi$, and if $X_i \in T$ then $\xi_i = \epsilon$.

(2) Let $A \rightarrow Bf \in R$ be a Type 2 production. Then,

$$\beta A \xi \gamma \underset{G}{\rightarrow} \beta B f \xi \gamma.$$

(3) Let $Af \rightarrow X_1 X_2 \cdots X_k \in R$ be a Type 3 production. Then,

$$\beta A f \xi \gamma \underset{G}{\rightarrow} \beta X_1 \xi_1 X_2 \xi_2 \cdots X_k \xi_k \gamma$$

where if $X_i \in N$ then $\xi_i = \xi$, and if $X_i \in T$ then $\xi_i = \epsilon$.

A Type 1 production distributes index sequences associated with the nonterminal to which the production is applied to all nonterminals on the right-hand side. A Type 2 production adds an index f to the left-end of the index sequences associated with the left-hand side (and provides the nonterminal on the right-hand side with the resultant index sequence). A Type 3 production deletes the leftmost index of the index sequence and distributes the remaining sequence to all nonterminals on the right-hand side. The reflexive and transitive closure of $\underset{G}{\rightarrow}$ is denoted by $\overset{*}{\underset{G}{\rightarrow}}$. We will simply write $\rightarrow$ for the relation $\underset{G}{\rightarrow}$ if G is clear from the context. The language generated by an IG $G = (N, T, I, R, S)$ is defined as $L(G) = \{w \in T^* \mid S \overset{*}{\underset{G}{\rightarrow}} w\}$.

4.2 Set of Unchecked Traces

We will omit the concatenation symbol : of node sequences in the following. For a given program $P = (NO, IS, IT, TG, CG)$, the index grammar $G_{P,\mathrm{T}} = (N, T, I, R, S)$ is constructed as follows.

(a) $N = \{S, W, A, B, C\} \cup \{N_i, N_i', N_i'' \mid n_i \in NO\}$.

(b) $T = NO \ \cup \{\triangleright\}$.

(c) $I = \{\dot{n}_i \mid n_i \in NO\} \cup \{\$\}$.

(d) Let R be the set of productions which consists of:

$$S \ \rightarrow \ W\$ \tag{2}$$
$$W \ \rightarrow \ A\dot{n}_1 \ \text{ for } n_1 = IT \tag{3}$$
$$A \ \rightarrow \ B \triangleright C \tag{4}$$
$$A \ \rightarrow \ B \tag{5}$$
$$B\dot{n}_i \ \rightarrow \ Bn_i \ \text{ for } \forall n_i \in NO \tag{6}$$
$$B\$ \ \rightarrow \ \epsilon \tag{7}$$
$$C\dot{n}_i \ \rightarrow \ N_i \ \text{ for } \forall n_i \in NO \tag{8}$$
$$N_j'' \ \rightarrow \ A\dot{n}_k \ \text{ for } \forall n_j, n_k \text{ such that } n_j \overset{TG}{\rightarrow} n_k \tag{9}$$

For each node n_i, one of the following sets (i), (ii) and (iii) of productions is added to R according to $IS(n_i)$.

(i) $IS(n_i) = call$:

$$N_i \ \rightarrow \ N_i'\dot{n}_i \tag{10}$$
$$N_i' \ \rightarrow \ A\dot{n}_j \ \text{ for } \forall n_j \text{ such that } n_i \overset{CG}{\rightarrow} n_j \tag{11}$$

(ii) $IS(n_i) = return$:

$$N_i\dot{n}_j \ \rightarrow \ N_j'' \ \text{ for } \forall n_j \in NO \tag{12}$$

(iii) $IS(n_i) = check(L_\phi)$:

$$N_i \ \rightarrow \ A\dot{n}_j \ \text{ for } \forall n_j \text{ such that } n_i \overset{TG}{\rightarrow} n_j \tag{13}$$

It is clear from production (13) that $L(G_{P,\mathrm{T}})$ is the set of unchecked traces in P.

4.3 Set of Sequences Satisfying Local Checks

The execution which has resulted in a state s continues if and only if s satisfies the following condition.

> If the top element of the state s is n_i and $IS(n_i) = check(L_{\phi_i})$, then $s \in L_{\phi_i}$ holds.

For a check node n_i, let the language $L^{(i)}_{P,C}$ consisting of state sequences which satisfy the above condition for this particular node n_i is represented by the following regular expression.

$$L^{(i)}_{P,\mathrm{C}} = NO^*(NO - \{n_i\}) \cup \epsilon \cup L_{\phi_i}.$$

We can define the language $L_{P,\mathrm{C}}$ consisting of state sequences which satisfy the above condition for all check nodes as follows:

$$L_{P,\mathrm{C}} = (X \triangleright)^* NO^* \tag{14}$$

where $\{n_1, \ldots, n_l\}$ is the set of check nodes in P and $X := L^{(1)}_{P,C} \cap L^{(2)}_{P,C} \cap \cdots \cap L^{(l)}_{P,C}$.

4.4 Main Results

First, we will show that the language $L(G_{P,\mathrm{T}}) \cap L_{P,\mathrm{C}}$ coincides with the set $[\![P]\!]$ of traces in P.

Lemma 4.2 *Every $\alpha \in L(G_{P,\mathrm{T}})$ can be written as $\alpha = s_1 \triangleright \cdots \triangleright s_n$ where $s_i \in NO^*$ $(1 \leq i \leq n)$ and $n \geq 1$, and there is a derivation of the following form resulting in α. The number at the right-end of each line is the number of the applied production in section 4.2, and $\delta_n \in (I - \{\$\})^+$ $(1 \leq i \leq n)$.*

$$\begin{array}{lll}
S & \stackrel{*}{\rightarrow} & A\delta_1\$ & (1),\ (2) \\
 & \rightarrow & B\delta_1\$ \triangleright C\delta_1\$ & (3) \\
 & \stackrel{*}{\rightarrow} & B\delta_1\$ \triangleright A\delta_2\$ & (7) - (10) \\
 & \rightarrow & B\delta_1\$ \triangleright B\delta_2 \triangleright C\delta_2\$ & (3) \\
 & \stackrel{*}{\rightarrow} & B\delta_1\$ \triangleright \cdots \triangleright B\delta_{n-1}\$ \triangleright A\delta_n\$ & \\
 & \rightarrow & B\delta_1\$ \triangleright \cdots \triangleright B\delta_{n-1}\$ \triangleright B\delta_n\$ & (4) \\
 & \stackrel{*}{\rightarrow} & s_1 \triangleright \cdots \triangleright s_n. & (5),\ (6)
\end{array}$$

Proof. For an arbitrary derivation $S \stackrel{*}{\rightarrow} s_1 \triangleright \cdots \triangleright s_n$, the only production which can directly generate a terminal symbol is (6). The derivation steps using productions (7) and (8) can be moved to the end of the derivation. It is easy to see that the other steps in this derivation is exactly those stated in the lemma by the definition of $G_{P,\mathrm{T}}$. □

Let $\sigma : NO^* \rightarrow (I - \$)^*$ be the mapping defined by $\sigma(n_{i_1} \cdots n_{i_n}) := \dot{n_{i_n}} \cdots \dot{n_{i_1}}$.

Lemma 4.3 *For each program P, $\delta \in (I - \$)^*$ and $s \in T^*$,*

$$B\delta\$ \underset{G_{P,\mathrm{T}}}{\stackrel{*}{\rightarrow}} s \quad \textit{if and only if} \quad s \in NO^* \textit{ and } \delta = \sigma(s).$$

□

Lemma 4.4 *For each program P, $s_1 \triangleright \cdots \triangleright s_n \in [\![P]\!]$ if and only if $S \underset{G_{P,\mathrm{T}}}{\stackrel{*}{\rightarrow}} B\delta_1\$ \triangleright \cdots B\delta_{n-1}\$ \triangleright A\delta_n\$$ and $s_1 \triangleright \cdots \triangleright s_n \in L_{P,\mathrm{C}}$, where $\delta_i = \sigma(s_i)$ $(1 \leq i \leq n)$.*

Proof. The *only if* part is shown by induction on n. (The proof of the *if* part is similar.)

(basis) Assume $s_1 \in [\![P]\!]$. By the definition of a trace, $s_1 = \langle IT \rangle = \langle n_1 \rangle$. By productions (2) and (3),

$$S \rightarrow W\$ \rightarrow A\dot{n_1}\$$$

holds. Also, $\dot{n_1} = \sigma(n_1)$ and $s_1 \in NO \subseteq (X \triangleright)^* NO^* = L_{P,\mathrm{C}}$.

(inductive step) For an integer $n\,(\geq 1)$, assume an inductive hypothesis if $s_1 \triangleright \cdots \triangleright s_n \in [\![P]\!]$, then $S \stackrel{*}{\rightarrow} B\delta_1\$ \triangleright \cdots \triangleright B\delta_{n-1}\$ \triangleright A\delta_n\$$, $\delta_i = \sigma(s_i)$ $(1 \leq i \leq n)$, and $s_1 \triangleright \cdots \triangleright s_n \in L_{P,\mathrm{C}}$. Further assume that $s_1 \triangleright \cdots \triangleright s_n \triangleright s_{n+1} \in [\![P]\!]$. Since $s_n \neq \epsilon$ we can write $s_n = s'n_v$ for some $s' \in NO^*$ and $n_v \in NO$. By production (4),

$$A\delta_n\$ \rightarrow B\delta_n\$ \triangleright C\delta_n\$. \tag{15}$$

One of the following three cases holds according to $IS(n_v)$.

(1) $IS(n_v) = \mathit{call}$. Since $s_n \triangleright s_{n+1}$ and $s_n = s'n_v$, there is a node n_j such that $n_v \stackrel{CG}{\rightarrow} n_j$ and $s_{n+1} = s'n_v n_j$. Productions (8), (10) and (11) can be applied in this order to $C\delta_n\$$ in derivation (15), and it follows from $\delta_n = \sigma(s_n)$, $s_n = s'n_v$ and $s_{n+1} = s'n_v n_j$ that

$$C\delta_n\$ = C\dot{n_v}\sigma(s')\$ \rightarrow N_v\sigma(s')\$ \rightarrow N'_v\dot{n_v}\sigma(s')\$ \rightarrow A\dot{n_j}\dot{n_v}\sigma(s')\$ = A\sigma(s_{n+1})\$.$$

Hence, $S \stackrel{*}{\rightarrow} B\delta_1\$ \triangleright B\delta_2\$ \triangleright \cdots \triangleright B\delta_n\$ \triangleright A\delta_{n+1}\$$ and $\delta_i = \sigma(s_i)$ $(1 \leq i \leq n+1)$.

Next, we will show $s_1 \triangleright s_2 \triangleright \cdots \triangleright s_n \triangleright s_{n+1} \in L_{P,\mathrm{C}}$. From the inductive hypothesis $s_1 \triangleright s_2 \triangleright \cdots \triangleright s_n \in L_{P,\mathrm{C}}$, we can see $s_i \in X$ $(1 \leq i < n)$ by (14). Hence, it suffices to show that $s_n \in X$ and $s_{n+1} \in NO^*$. The latter is trivial. It follows from $s_n = s'n_v$ and $IS(n_v) = \mathit{call}$ that $s_n \in X$ holds by the definition of $L_{P,\mathrm{C}}$.

(2) $IS(n_v) = \mathit{return}$.
Since $s_n \triangleright s_{n+1}$, we can write s_n as $s_n = sn_un_v$ for some $n_j \in NO$ and $s \in NO^*$ such that $n_u \stackrel{TG}{\rightarrow} n_j$ and $s_{n+1} = sn_j$. Productions (8), (12) and (9) can be applied in this order to $C\delta_n\$$ in derivation (15), and

$$\begin{aligned} C\delta_n\$ &= C\dot{n}_v\dot{n}_u\sigma(s)\$ \rightarrow N_v\dot{n}_u\sigma(s)\$ \\ &\rightarrow \quad N''_u\sigma(s)\$ \rightarrow A\dot{n}_j\sigma(s)\$ = A\sigma(s_{n+1})\$. \end{aligned}$$

Hence, $S \stackrel{*}{\rightarrow} B\delta_1\$ \triangleright B\delta_2\$ \triangleright \cdots \triangleright B\delta_n\$ \triangleright A\delta_{n+1}\$$ and $\delta_i = \sigma(s_i)$ $(1 \leq i \leq n+1)$.

It can be shown that $s_1 \triangleright s_2 \triangleright \cdots \triangleright s_n \triangleright s_{n+1} \in L_{P,\mathrm{C}}$ in a similar way to the case of $IS(n_v) = \mathit{call}$.

(3) $IS(n_v) = \mathit{check}(L_{\phi_v})$. Since $s_n \triangleright s_{n+1}$, there is a node n_j such that $n_v \stackrel{TG}{\rightarrow} n_j$ and $s_n \in L_{\phi_v}$. Also, $s_{n+1} = s'n_j$ follows.

Productions (8) and (13) can be applied in this order to $C\delta_n\$$ in derivation (15), and

$$\begin{aligned} C\delta_n\$ &= C\dot{n}_v\sigma(s')\$ \\ &\rightarrow \quad N_v\sigma(s')\$ \rightarrow A\dot{n}_j\sigma(s')\$ = A\sigma(s_{n+1})\$. \end{aligned}$$

Hence, $S \stackrel{*}{\rightarrow} B\delta_1\$ \triangleright B\delta_2\$ \triangleright \cdots \triangleright B\delta_n\$ \triangleright A\delta_{n+1}\$$ and $\delta_i = \sigma(s_i)$ $(1 \leq i \leq n+1)$.

In a similar way to the other cases, it suffices to show $s_n \in X$ in order to prove that $s_1 \triangleright s_2 \triangleright \cdots \triangleright s_n \triangleright s_{n+1} \in L_{P,\mathrm{C}}$. It follows from $s_n = s'n_v$ and $s_n \in L_{\phi_v}$ that $s_n \in L^{(v)}_{P,C}$. Therefore, $s_n \in X$.

□

Theorem 4.5 *For each program P, $[\![P]\!] = L(G_{P,\mathrm{T}}) \cap L_{P,\mathrm{C}}$.*

Proof. We will show $[\![P]\!] \subseteq L(G_{P,\mathrm{T}}) \cap L_{P,\mathrm{C}}$. Assume that $s_1 \triangleright s_2 \triangleright \cdots \triangleright s_n \in [\![P]\!]$. By Lemma 4.4, $G_{P,\mathrm{T}}$ satisfies $S \stackrel{*}{\rightarrow} B\delta_1\$ \triangleright \cdots \triangleright B\delta_{n-1}\$ \triangleright A\delta_n\$$, where $\delta_i = \sigma(s_i)$ $(1 \leq i \leq n)$. By applying Production (5) to $A\delta_n$ in the above derivation, we obtain

$$B\delta_1\$ \triangleright \cdots \triangleright B\delta_{n-1}\$ \triangleright A\delta_n\$ \rightarrow B\delta_1\$ \triangleright \cdots \triangleright B\delta_{n-1}\$ \triangleright B\delta_n\$.$$

Since $\delta_i = \sigma(s_i)$ $(1 \leq i \leq n)$, by applying Lemma 4.3 to each $B\delta_i\$$, $B\delta_1\$ \triangleright \cdots \triangleright B\delta_n\$ \stackrel{*}{\rightarrow} s_1 \triangleright \cdots \triangleright s_n$ holds, from which $s_1 \triangleright \cdots \triangleright s_n \in L(G_{P,\mathrm{T}})$ follows. By Lemma 4.4, $s_1 \triangleright \cdots \triangleright s_n \in L_{P,\mathrm{C}}$ holds, and hence $s_1 \triangleright s_2 \triangleright \cdots \triangleright s_n \in L(G_{P,\mathrm{T}}) \cap L_{P,\mathrm{C}}$. Thus, $[\![P]\!] \subseteq L_P$ holds.

$L(G_{P,\mathrm{T}}) \cap L_{P,\mathrm{C}} \subseteq [\![P]\!]$ can be shown in a similar way. □

Lemma 4.6 [A68] *The class of indexed languages is closed under intersection with regular languages. The emptiness problem for indexed languages is decidable.* □

Theorem 4.7 *For a given program P and a given verification property L_ψ, the verification problem $[\![P]\!] \subseteq L_{\mathrm{safe}}[\psi]$ is decidable.*

Proof. The theorem follows from the fact $[\![P]\!] \subseteq L_{\mathrm{safe}}[\psi] \Leftrightarrow [\![P]\!] \cap \overline{L_{\mathrm{safe}}[\psi]} = \emptyset$ (the empty set), Theorem 4.5, and Lemma 4.6. □

4.5 An Alternative Method

The set of all states $[P]$ which are reachable from the initial state of 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 decide 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. If P contains no check node, then it is easy to see that $[P]$ is a CFL and hence we can decide whether $=[P] \subseteq L_\psi$ (see Theorem 5.5 in Section 5).

5 Complexity of the Verification Problem

Since an input for 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 a NFA, respectively. 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.

In the following, we will show that the verification problem is DEXP-POLY time-complete if the regular language in each check node is specified by a NFA and a verification property is specified by a DFA. For a set A, let $|A|$ denote the cardinality of A. The number of states of a 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 a FA M_{ϕ_i}. The size of P is defined as $\|P\| = |NO| + |TG| + |CG| + \max\{\#M_{\phi_1}, \ldots, \#M_{\phi_k}\}$.

For an IG $G = (N, T, I, R, S)$, the size of G is defined as $\|G\| = |N| + |T| + |I| + \|R\|$ where $\|R\|$ is the description length of R. We can obtain the following lemma by analyzing the proofs of Lemma 3.2 and Theorem 4.1 of [A68].

Let $L(M)$ denote the language accepted by a FA M.

Lemma 5.1[A68] *(a) For an IG G and a NFA M, an IG G' can be constructed such that $L(G') = L(G) \cap L(M)$ and $\|G'\|$ is $O(\|G\|(\#M)^3)$. (b) The emptiness problem for the language generated by an IG G is solvable in deterministic $O(2^{p(\|G\|)})$ time for a polynomial p.* □

Lemma 5.2 *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 a 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 solvable in DEXP-POLY time.*

Proof. By Theorems 4.5 and 4.7, the problem is equivalent to deciding whether

$$L(G_{P,\mathrm{T}}) \cap L_{P,\mathrm{C}} \cap \overline{L_{\mathrm{safe}}[\psi]} = \emptyset \tag{16}$$

where $G_{P,\mathrm{T}}$ is the indexed grammar constructed in section 4.2, $L_{P,\mathrm{C}}$ is the regular language defined from L_{ϕ_i} $(1 \leq i \leq k)$ in section 4.3 and $L_{\mathrm{safe}}[\psi] = (L_\psi \rhd)^* L_\psi$. We can show the following properties.

(i) $\|G_{P,\mathrm{T}}\|$ is $O(|NO|^2)$.

(ii) Let $n_1 = \max\{\#M_{\phi_1}, \ldots, \#M_{\phi_k}\}$. A NFA $M_{P,\mathrm{C}}$ can be constructed as follows such that $L_{P,\mathrm{C}} = L(M_{P,\mathrm{C}})$ and $\#M_{P,\mathrm{C}}$ is $O(|NO| \cdot n_1)$.

Let n_{ϕ_i} be the node $check(L_{\phi_i})$. In section 4.3, $L_{P,\mathrm{C}}$ is defined as $(X \rhd)^* NO^*$ where

$$\begin{aligned} X &= \bigcap_{i=1}^{k} (NO^*(NO - \{n_{\phi_i}\}) \cup \epsilon \cup L_{\phi_i}) \\ &= \epsilon \cup NO^*(NO - \{n_{\phi_1}, \ldots, n_{\phi_k}\}) \cup \bigcup_{i=1}^{k} (NO^* n_{\phi_i} \cap L_{\phi_i}). \end{aligned}$$

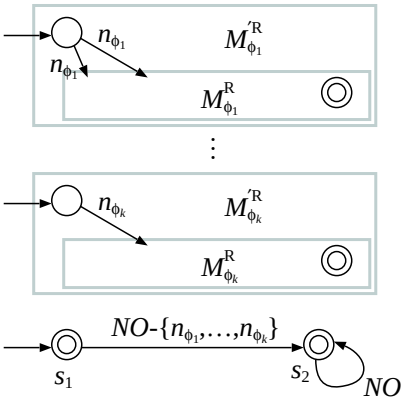
(These equations follow the fact that for any sets $A_1, \ldots, A_k$, $B_1, \ldots, B_k$,

$$\begin{aligned} (A_1 \cup B_1) \cap \cdots \cap (A_k \cup B_k) = \quad & (A_1 \cap \cdots \cap A_{k-2} \cap A_{k-1} \cap A_k) \\ & \cup (A_1 \cap \cdots \cap A_{k-2} \cap A_{k-1} \cap B_k) \\ & \cup (A_1 \cap \cdots \cap A_{k-2} \cap B_{k-1} \cap A_k) \\ & \cup (A_1 \cap \cdots \cap A_{k-2} \cap B_{k-1} \cap B_k) \\ & \cup \cdots \\ & \cup (B_1 \cap \cdots \cap B_{k-2} \cap B_{k-1} \cap B_k). \end{aligned}$$

Note that $NO^*(NO - \{n_{\phi_i}\}) \cup \epsilon \cup L_{\phi_i} = \epsilon \cup NO^*(NO - \{n_{\phi_i}\}) \cup (NO^* n_{\phi_i} \cap L_{\phi_i})$. When we let $A_1 = \cdots = A_k = \{\epsilon\}$ and $B_i = NO^*(NO - \{n_{\phi_i}\}) \cup (NO^* n_{\phi_i} \cap L_{\phi_i})$, the right-hand side equals $\{\epsilon\} \cup \bigcap_{i=1}^{k} B_i$. When we let $A_i = NO^*(NO - \{n_{\phi_i}\})$ and $B_i = NO^* n_{\phi_i} \cap L_{\phi_i}$ for $1 \leq i \leq k$, the right-hand side equals $\bigcap_{i=1}^{k} A_i \cup \bigcup_{i=1}^{k} B_i$ since $A_i \cap B_j = B_j$ and $B_i \cap B_j = \emptyset$ for any distinct i and j.)

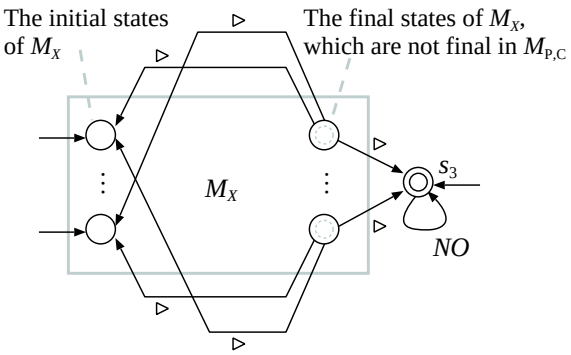
Without loss of generality, we assume that each $M_{\phi_i} (1 \leq i \leq k)$ has no ϵ-move. (If M_{ϕ_i} has an ϵ-move, then we can construct a NFA equivalent to M_{ϕ_i} which has the same number of states as M_{ϕ_i} in $O((\#M_{\phi_i})^2)$ time.)

For a language L, let L^{R} denote the language obtained by reversing words in L. For a FA M, we can construct a NFA M^{R} such that $L(M^{\mathrm{R}}) = (L(M))^{\mathrm{R}}$



The initial states of M^{R}_X are the ones of any $M'^{\mathrm{R}}_{\phi_i}$ and s_1. The final states of M^{R}_X are the ones of any $M'^{\mathrm{R}}_{\phi_i}$, s_1 and s_2.

Figure 2: NFA M^{R}_X



The initial states of $M_{P,\mathrm{C}}$ are the ones of M_X and s_3. The final state of $M_{P,\mathrm{C}}$ is s_3.

Figure 3: NFA $M_{P,\mathrm{C}}$

and $\#M^{\mathrm{R}} = \#M$ by reversing the transitions and exchanging the set of initial states and the set of final states. The reverse of X can be denoted as $X^{\mathrm{R}} = \epsilon \cup (NO - \{n_{\phi_1}, \ldots, n_{\phi_k}\})NO^* \cup \bigcup_{i=1}^{k} (n_{\phi_i} NO^* \cap L^{\mathrm{R}}_{\phi_i})$. We can construct a NFA M^{R}_X such that $X^{\mathrm{R}} = L(M^{\mathrm{R}}_X)$ and $\#M^{\mathrm{R}}_X = \sum_{i=1}^{k}(\#M_{\phi_i} + 1) + 2$, as shown in figure 2 where $M'^{\mathrm{R}}_{\phi_i}$ is a NFA such that $L(M'^{\mathrm{R}}_{\phi_i}) = n_{\phi_i} NO^* \cap L^{\mathrm{R}}_{\phi_i}$. This NFA $M'^{\mathrm{R}}_{\phi_i}$ can be constructed from $M^{\mathrm{R}}_{\phi_i}$ by adding a single new state s' and a transition $s' \stackrel{n_{\phi_i}}{\rightarrow} t$ for each state t such that there is a transition $s \stackrel{n_{\phi_i}}{\rightarrow} t$ for an initial state s of $M^{\mathrm{R}}_{\phi_i}$. The initial state of $M'^{\mathrm{R}}_{\phi_i}$ is s' and the final states are the ones of $M^{\mathrm{R}}_{\phi_i}$. Thus, a NFA M_X can be obtained such that $X = L(M_X)$ and $\#M_X = \#M^{\mathrm{R}}_X$.

Using M_X, we can obtain $M_{P,\mathrm{C}}$ as figure 3 and $\#M_{P,\mathrm{C}} = \#M_X + 1$, that is, $O(|NO| \cdot n_1)$.

(iii) Let $n_2 = \#M_\psi$. A FA $M_{\overline{\mathrm{safe}}}$ can be constructed such that $\overline{L_{\mathrm{safe}}[\psi]} = L(M_{\overline{\mathrm{safe}}})$ and $\#M_{\overline{\mathrm{safe}}} = n_2$ by defining the set of final states of $M_{\overline{\mathrm{safe}}}$ as the set of non-final states of M_ψ.

Hence, by Lemma 5.1 (a), an indexed grammar G_{P} can be constructed such that $L(G_{\mathrm{P}}) = L(G_{P,\mathrm{T}}) \cap L(M_{P,\mathrm{C}}) \cap L(M_{\overline{\mathrm{safe}}}) = L(G_{P,\mathrm{T}}) \cap L_{P,\mathrm{C}} \cap \overline{L_{\mathrm{safe}}[\psi]}$ and $\|G_{\mathrm{P}}\|$ is $O(|NO|^2(|NO| \cdot n_1 \cdot n_2)^3)$, which is a polynomial order of $\|P\|$ and n_2. By (16), the

verification problem is equivalent to deciding whether $L(G_{\mathrm{P}}) = \emptyset$. By Lemma 5.1 (b) and the above facts, (16) can be done in deterministic $O(2^{p(\|P\|+n_2)})$ time for some polynomial p. □

Note. If a verification property L_ψ would also be specified by a NFA instead of a DFA in Lemma 5.2, then $\#M_{\overline{\mathrm{safe}}}$ in the proof would no longer be a polynomial of n_2 $(= \#M_\psi)$. The complexity of the problem in this case becomes a double exponential time.

Theorem 5.3 *Let $P = (NO, IS, IT, TG, CG)$ be a program and L_ψ a verification property which satisfies the assumption stated in Lemma 5.2. The verification problem for P and L_ψ is DEXP-POLY time-complete.*

Proof. By Lemma 5.2, it suffices to show that the problem is DEXP-POLY time-hard. It is known that a language L belongs to DEXP-POLY time if and only if L is accepted by a polynomial space-bounded alternating Turing machine (ATM)[CKS81]. For any given polynomial space-bounded ATM M and any input x of M, we can transform M and x into a program P_M and a verification property L_ψ within polynomial time such that

$$[\![P_M]\!] \not\subseteq L_{\mathrm{safe}}[\psi] \;\Leftrightarrow\; M \text{ accepts } x.$$

Therefore the verification problem is DEXP-POLY time-hard.

Below we sketch the transformation. Assume that for any input x whose length equals n, M uses not more than $p(n)$ space for a polynomial p. Let $\Gamma = \{\gamma_1, \ldots, \gamma_{|\Gamma|}\}$ be the set of tape symbols of M and γ_1 be the blank symbol. Let δ be the transition function of M.

Consider a program P_1 shown in figure 4. A node with $check(NO^*)$ is the one with no operation since every state satisfies NO^*. When the control reaches the node n_1 for the first time, the state (stack) of the program P_1 is a sequence of nodes whose length equals $p(n) + 1$. Considering that each $\gamma_{i,j}$ corresponds to the tape symbol γ_i, we can regard this state as one of the $|\Gamma|^{p(n)}$ possible strings contained by the $p(n)$ tape squares of M, which we will refer to as an instantaneous description (ID) as usual. In general, the node n_1 has been visited

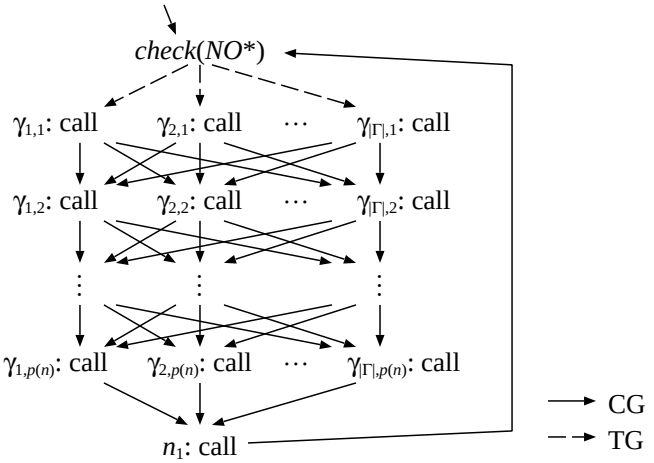
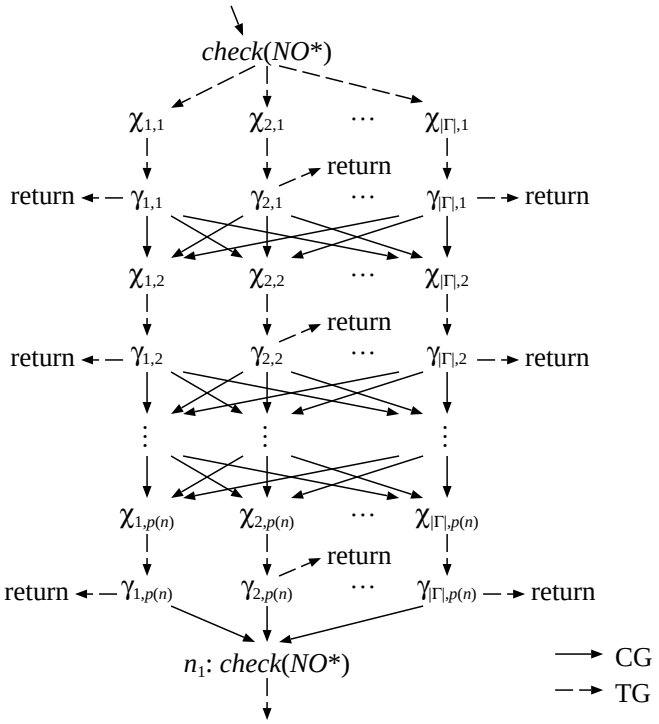


Figure 4: Program P_1

more than once (by recursive calls), and thus the state of P_1 can be regarded as a sequence of IDs separated by n_1. However, this sequence of IDs may not represent a valid computation of M because any two IDs may be adjacent. To simulate a computation of M, the program should guarantee that the tape square scanned at the last computation step is rewritten to a specified symbol and the other tape squares are preserved.

For an ID $\alpha = \sigma_1\sigma_2\cdots\sigma_{p(n)}$ (where $\sigma_i = \gamma_{u,i}$ with $1 \leq u \leq |\Gamma|$ and $1 \leq i \leq p(n)$), let $\alpha[\sigma/k] = \sigma_1\cdots\sigma_{k-1}\sigma\sigma_{k+1}\cdots\sigma_{p(n)}$, where k corresponds to the position of the tape head of M at the last computation step and σ is the tape symbol which the tape square k is rewritten to. By modifying P_1, we can construct a program $P_2[k,\sigma]$ (figure 5) which pushes $\alpha[\sigma/k]$ onto the stack where α is the topmost ID of the stack. More precisely, let $s = s_0\alpha n_0$ (for $s_0 \in NO^*$, α an ID, and $n_0 \in NO$) be a stack. When $P_2[k,\sigma]$ is called with stack s and the control reaches the node n_1 at the bottom of figure 5, the stack becomes $s\alpha[\sigma/k]n_1$ (cf. figure 8). $P_2[k,\sigma]$ is obtained by attaching a check node $\chi_{i,j}$ and a return node to each $\gamma_{i,j}$ in P_1. $\chi_{i,j}$ is $check(NO^*\gamma_{i,j}NO^{p(n)+1})$ for all $j \neq k$, $check(NO^*)$ for $j = k$ and i such that $\gamma_i = \sigma$, and $check(\emptyset)$ otherwise. Figure 6 shows the state of $P_2[k,\sigma]$ when the control is at $\chi_{i,j}$. These check nodes obstruct a sequence of IDs which does not represent a valid computation of M.

Let $\alpha = \sigma_1\sigma_2\cdots\sigma_{p(n)}$ be an ID with state q and head position k. Assume that $(q',\sigma,\Delta) \in \delta(q,\sigma_k)$, that is, a possible move at α is to rewrite k-th tape square



$\chi_{i,j}$ is $check(L_{i,j})$ where

$$L_{i,j} = \begin{cases} NO^*\gamma_{i,j}NO^{p(n)+1} & \text{if } j \neq k, \\ NO^* & \text{if } j = k \text{ and } \gamma_i = \sigma, \\ \emptyset & \text{otherwise.} \end{cases}$$

Figure 5: Program $P_2[k, \sigma]$

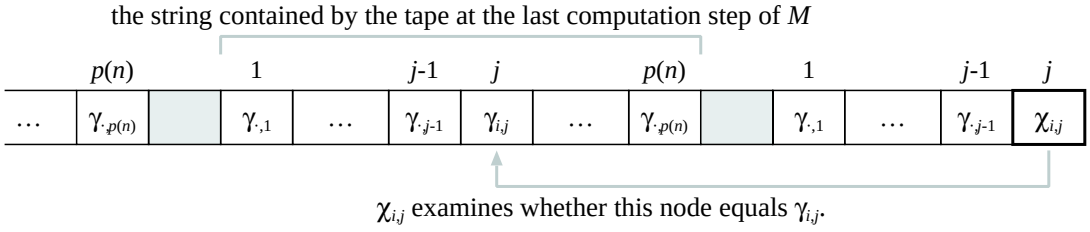


Figure 6: The stack of $P_2[k, \sigma]$ when the control is at $\chi_{i,j}$

from σ_k to σ, change the state from q to q', and change the head position from k to $k' = k + \Delta$. A program $P_3[k, \sigma, q', k']$ in figure 7 with topmost ID α on the stack first pushes the ID α' obtained from α by the above move $(q', \sigma, \Delta) \in \delta(q, \sigma_k)$, and simulates further moves from α' recursively (figure 8). $P_3[k, \sigma, q', k']$ first executes $P_2[k, \sigma]$ to write the new ID α' onto the stack. After executing $P_2[k, \sigma]$, $P_3[k, \sigma, q', k']$ examines whether the current contents of the tape square k' equals $\gamma_i \in \Gamma$ by $check(NO^*\gamma_{i,k'}NO^{p(n)-k'+1})$, and calls $P_3[k', \sigma', q'', k'']$ for each $(q'', \sigma', \Delta') \in \delta(q', \gamma_i)$ and $k'' = k' + \Delta'$. $P_3[k, \sigma, q', k']$ calls all these $P_3[k', \sigma', q'', k'']$ sequentially if q' is a universal state of M. If q' is an existential state of M, $P_3[k, \sigma, q', k']$ calls any one of these $P_3[k', \sigma', q'', k'']$ and returns. Otherwise, that is, if q' is a final state of M, $P_3[k, \sigma, q', k']$ calls no $P_3[k', \sigma', q'', k'']$ and simply returns. Thus $P_3[k, \sigma, q', k']$ returns if and only if the configuration of M which consists of q', k' and the contents of the tape (ID) written on the stack is a yes-configuration.

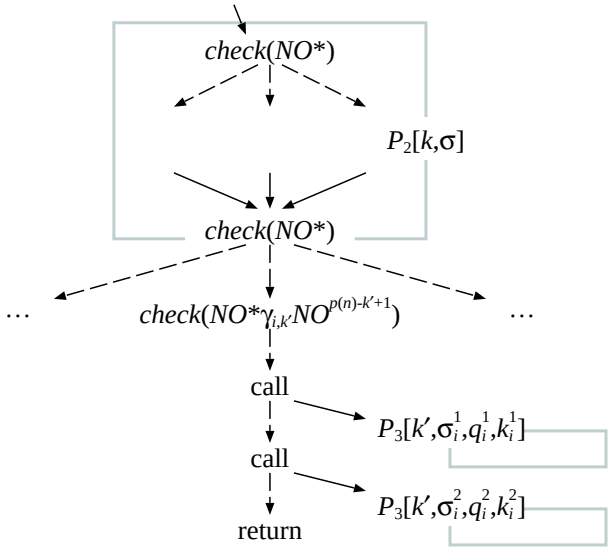
A program P_x which simulates the initial configuration of M is similar to $P_3[k, \sigma, q_0, 1]$ where q_0 is the initial state of M; P_x does not execute $P_2[k, \sigma]$ and instead it writes the input string x onto the stack (figure 9).

The overall program P_M constructed by the transformation consists of P_x, $P_3[k, \sigma, q', k']$ for all k, σ, q' and k', and two call nodes $n_{\rm s}$ and $n_{\rm t}$ (figure 10). An execution of P_M reaches the node $n_{\rm t}$ if and only if M accepts x. We can simply let $L_\psi = (NO - \{n_{\rm t}\})^*$. □

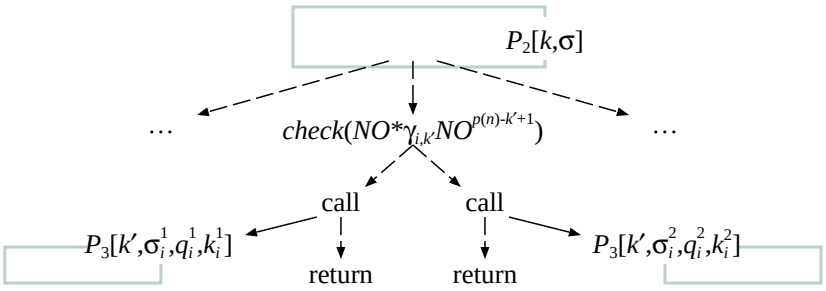
Note that the language $NO^*\gamma_{i,j}NO^{p(n)+1}$ in the check node $\chi_{i.j}$ in the proof of Theorem 5.3 does not seem to be specified by a DFA whose size is a polynomial of n. Thus it is unknown whether the verification problem is still DEXP-POLY time-hard if the language in each check node is specified by a DFA. However, the verification problem in this case can be shown to be PSPACE-hard.

Proposition 5.4 *Let $P = (NO, IS, IT, TG, CG)$ be a program and L_ψ a verification property which satisfies the assumption stated in Lemma 5.2 except that the language L_{ϕ_i} in each check(L_{ϕ_i}) is specified by a DFA M_{ϕ_i}. The verification problem for P and L_ψ is PSPACE-hard.*

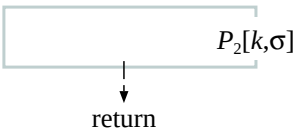
Proof. We transform Quantified 3-Satisfiability (Quantified 3SAT) problem to the verification problem. An instance of Quantified 3SAT is a



(a) q' is a universal state



(b) q' is an existential state



(c) q' is a final state

In this figure, we assume that $\delta(q', \gamma_i) = \{(q_i^1, \sigma_i^1, \Delta_i^1), (q_i^2, \sigma_i^2, \Delta_i^2)\}$, $k_i^1 = k' + \Delta_i^1$, and $k_i^2 = k' + \Delta_i^2$.

Figure 7: Program $P_3[k, \sigma, q', k']$

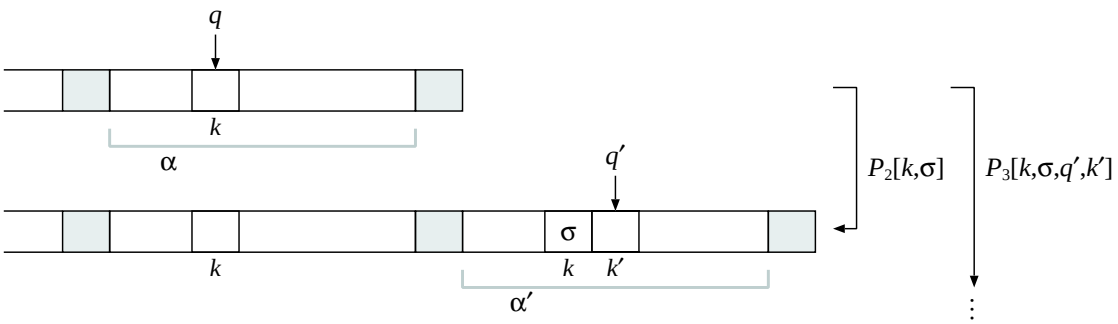
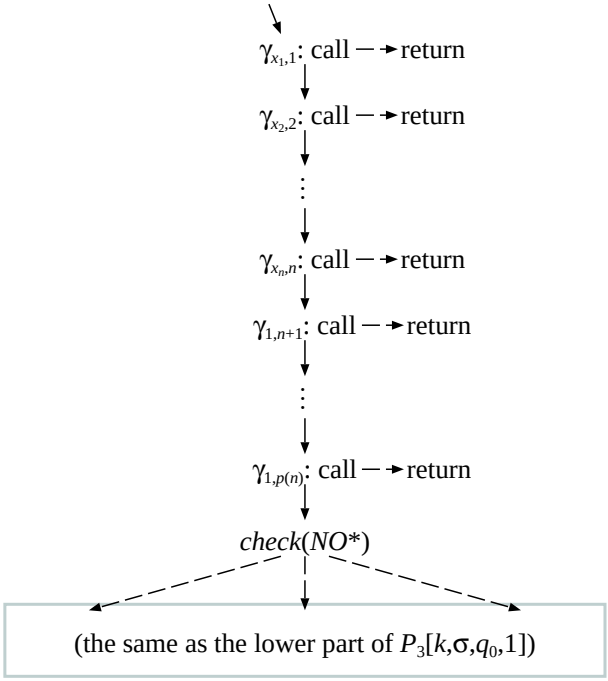


Figure 8: The extension of the stack made by $P_2[k, \sigma]$ and $P_3[k, \sigma, q, k']$



Let $x = \gamma_{x_1}\gamma_{x_2}\ldots\gamma_{x_n}$. ($\gamma_1$ is the blank symbol.)

Figure 9: Program P_x

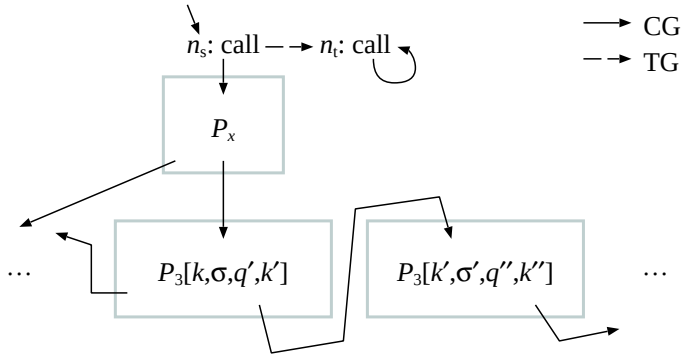


Figure 10: Program P_M

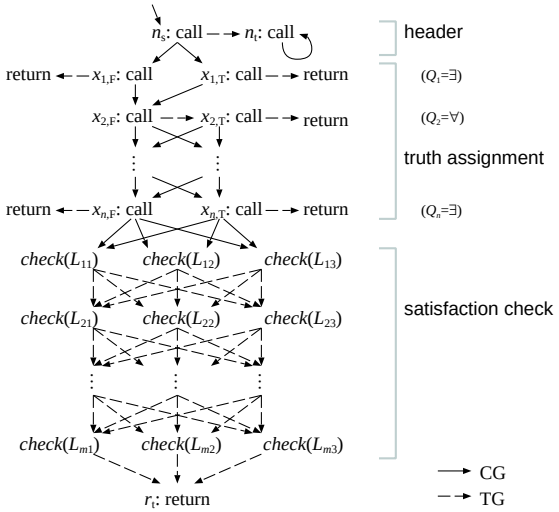


Figure 11: The program P_{Q3SAT} to solve QUANTIFIED 3SAT

Boolean formula $F = (Q_1 x_1)(Q_2 x_2) \ldots (Q_n x_n) E$ where E is a conjunction of 3-literal disjunctive clauses involving the variables $x_1, x_2, \ldots, x_n$ 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 11). Header consists of two nodes n_{s} and n_{t}. Truth assignment consists of $x_{i,\mathrm{F}}$ and $x_{i,\mathrm{T}}$ for $1 \leq i \leq n$. We consider that "false" is assigned to x_i when the control passes through $x_{i,\mathrm{F}}$ and "true" is assigned to x_i when the control passes through $x_{i,\mathrm{T}}$. We put a call edge between $x_{i-1,V}$ and $x_{i,V'}$ for each $V, V' \in \{\mathrm{F}, \mathrm{T}\}$ if Q_i is "$\exists$" (Figure 12).

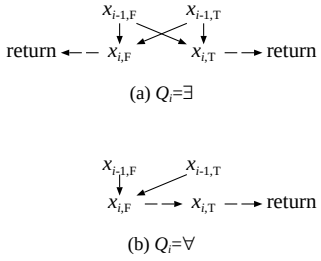


Figure 12: Edges in the truth assignment-part of P_{Q3SAT}

That is, the program tries assigning one of the truth values to x_i in this case. If Q_i is "∀," we put a call edge between $x_{i-1,V}$ and $x_{i,\mathrm{F}}$ for every $V \in \{\mathrm{F}, \mathrm{T}\}$ and put a transfer edge between $x_{i,\mathrm{F}}$ and $x_{i,\mathrm{T}}$. The program tries assigning each truth value to x_i in this case.

Let $E = C_1 \wedge C_2 \wedge \cdots \wedge C_m$ and $C_i = u_{i1} \vee u_{i2} \vee u_{i3}$ for $1 \leq i \leq m$, where u_{ij} is a literal over $\{x_1, \ldots, x_n\}$. Satisfaction check consists of $check(L_{ij})$ for $1 \leq i \leq m$ and $1 \leq j \leq 3$, where

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

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, $[\![P_{\mathrm{Q3SAT}}]\!] \not\subseteq L_{\mathrm{safe}}[\psi]$ for $L_\psi = (NO - \{n_{\mathrm{t}}\})^*$ 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. □

As a special case, the problem is solvable in polynomial time if a program contains no check nodes.

Theorem 5.5 *Let $P = (NO, IS, IT, TG, CG)$ be a program without check nodes. The verification problem for P and a verification property specified by a DFA M_ψ is solvable in polynomial time.*

Proof. Since P contains no check node, we can construct a regular grammar G_{P} such that $L(G_{\mathrm{P}}) = [P] \cap \overline{L(M_\psi)}$ and $\|G_{\mathrm{P}}\|$ is $O((|NO| + |TG| + |CG|)(\#M_\psi)^2)$.

The emptiness problem for the language generated by a regular grammar G is solvable in $O(\|G\|)$ time. Hence, we can decide whether $L(G_{\mathrm{P}}) = \emptyset$ in polynomial time in $\|P\|$ and $\#M_\psi$. □

As is the case with Lemma 5.2, the proof of the above theorem is not valid if a verification property L_ψ is specified by a NFA instead of a DFA. Especially, the verification problem in this case can be shown to be PSPACE-complete.

Proposition 5.6 *Let $P = (NO, IS, IT, TG, CG)$ be a program without check nodes. The verification problem for P and a verification property specified by a NFA M_ψ is PSPACE-complete.*

Proof. Both of the following two problems are known to be PSPACE-complete[GJ79].

FINITE AUTOMATON INEQUIVALENCE
INSTANCE: Two NFA A_1 and A_2 having the same input alphabet Σ.
QUESTION: $L(A_1) \neq L(A_2)$?

REGULAR EXPRESSION NON-UNIVERSALITY
INSTANCE: A regular expression E over a finite alphabet Σ.
QUESTION: $L(E) \neq \Sigma^*$?

PSPACE-solvability of the verification problem can be shown by transforming the problem to FINITE AUTOMATON INEQUIVALENCE problem. In the same way as the proof of Theorem 5.5, we can construct a NFA $M_{P,\mathrm{S}}$ such that $[P] = L(M_{P,\mathrm{S}})$ and $\#M_{P,\mathrm{S}} = O(|NO|)$. We can also construct a 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 13, 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 a NFA M_ψ such that $L(M_\psi) = L(\sigma_1 \cdot E)$ from the regular expression

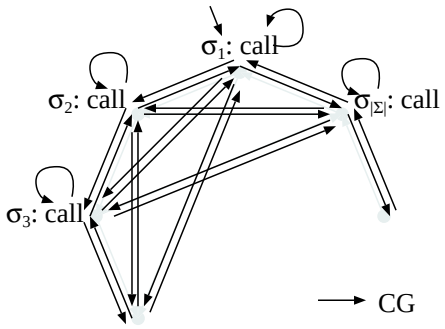


Figure 13: The program P_{Σ^*}

Table 1: Complexity of the verification problem

		the verification property	
		NFA	DFA
language in a check node	NFA	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
without check nodes		PSPACE-complete	PTIME

E. This construction of M_ψ can be performed in polynomial time[HU79]. 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 $\Sigma^* \neq L(E)$. □

The results on complexity of the verification problem shown in this section are summarized in table 1.

[JMT99] does not discuss the complexity of their verification algorithm. The best known upperbound of the time complexity of the verification problem for a Kripke structure S and an LTL formula f is linear in the size of S and exponential in the size of f [CGP99]. Let us assume that a program P in the flow graph model of [JMT99] and a verification property specified by an LTL formula ψ is given. The size of the Kripke structure induced by P and ψ in [JMT99] is exponential

in both the size $\|P\|$ of P and the size $\|\psi\|$ of ψ. Hence, the time complexity of the verification algorithm in [JMT99] is shown to be exponential in both $\|P\|$ and $\|\psi\|$.

Note that although we extend the [JMT99]'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 5.4 holds even for the [JMT99]'s model because the languages in check nodes of the program P_{Q3SAT} and the verification property L_ψ in the proof of Proposition 5.4 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 [JMT99].

6 Conclusion

In this paper, we defined the verification problem of deciding whether a given program which may contain stack inspection satisfies a given security property. Also we showed that the problem is decidable and analyze the computational complexity of the problem. Narrowing the gap between the upperbound and the lowerbound in three cases in table 1 is a future study. Improving the efficiency of the decision algorithm for practically important subclasses such as JDK1.2 stack inspection is another interesting problem.

Acknowledgments

The authors sincerely thank Associate Professor Seiji Hamaguchi of Osaka university for the explanation of the relation between regular languages and temporal logic. They also thank Assistant Professor Yasunori Ishihara of Osaka university for his valuable comments and Professor Dee A. Worman of Nara Institute of Science and Technology for carefully reading and editing the manuscript.

References

[ABLP93] 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.

[AFG99] M. Abadi, C. Fournet and G. Gonthier: Secure communications processing for distributed languages, 1999 IEEE Symp. on Security and Privacy, 74–88.

[A68] A. V. Aho: Indexed grammars — An extension of context-free grammars, Journal of the ACM, 15(4), 647–671, 1968.

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

[BW94] E. Bertino and H. Weigand: An approach to authorization modeling in object-oriented database systems, Data & Knowledge Engineering, 12, 1–29, 1994.

[CKS81] A. K. Chandra, D. C. Kozen and L. J. Stockmeyer: Alternation, Journal of the ACM, 28, 114–133, 1981.

[CGP99] E. M. Clarke, Jr., O. Grumberg and D. A. Peled: *Model Checking*, The MIT Press, 1999.

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

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

[E90] E. A. Emerson: *Temporal and Modal Logic*, in *Handbook of Theoretical Computer Science*, 1023–1024, Elsevier, 1990.

[GJ79] M. R. Garey and D. S. Johnson: *Computers and Intractability*, W. H. Freeman and Company, 1979.

[GMPS97] 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, USENIX Symp. on Internet Technologies and Systems, 103–112, 1997.

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

[HU79] J. E. Hopcroft and J. D. Ullman: *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 1979.

[IMI99] Y. Ishihara, T. Morita and M. Ito: The security problem against inference attacks on object-oriented databases, IFIP TC11 WG11.3 13th Working Conf. on Database Security, 1999, published as *Research Advances in Database and Information Systems Security*, 303–316, Kluwer Academic Publ.

[INTS00] S. Ikada, N. Nitta, Y. Takata and H. Seki: A security verification method for programs with stack inspection, Technical Report of IEICE, ISEC2000-78, 2000 (in Japanese).

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

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

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

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

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

[O95] P. Ørbæk: Can you trust your data?, TAPSOFT '95, LNCS 915, 575–589, 1995.

[PS94] J. Palsberg and M. I. Schwartzbach: *Object-Oriented Type Systems*, John Wiley & Sons, 1994.

[SA98] R. Stata and M. Abadi: A type system for Java bytecode subroutines, 25th ACM Symp. on Principles of Programming Languages, 149–160, 1998.

[SBCJ97] P. Samarati, E. Bertino, A. Ciampichetti and S. Jajodia: Information flow control in object-oriented systems, IEEE Trans. on Knowledge and Data Engneering, 9(4), 524–538, 1997.

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

[T96] K. Tajima: Static detection of security flaws in object-oriented databases, 1996 ACM SIGMOD Intl. Conf. on Management of Data, 341–352.

[VS97] D. Volpano and G. Smith: A type-based approach to program security, TAPSOFT '97, LNCS 1214, 607–621, 1997.

[WF98] D. S. Wallach and E. W. Felten: Understanding Java stack inspection, 1998 IEEE Symp. on Security and Privacy, 52–63, 1998.