Complexity of the Security Verification Problem for Programs with Stack Inspection

Naoya Nitta, Yoshiaki Takata and Hiroyuki Seki

Graduate School of Information Science
Nara Institute of Science and Technology
{naoya-n,y-takata,seki}@is.aist-nara.ac.jp

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 permission. For such a programming language, it is desirable to guarantee that each execution of a program satisfies required security properties. In the previous paper, we introduced a verification problem of deciding for a given program P and a security property ψ specified by a regular language whether every reachable state of P satisfies ψ, and showed that the problem is decidable. In this paper, we analyze the computational complexity of the verification problem. For example, the problem is shown to be deterministic EXP-POLY time complete if each check statement in P is specified by a NFA and a security property is specified by a DFA.

Keywords: access control, security verification, stack inspection, Java

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. 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 invoked 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 vaild 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 privileged, *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 one-step execution.
- A local security check statement has the form of *check*(ϕ) where ϕ is an LTL (linear temporal logic) formula [CGP99]. The execution of *check*(ϕ) 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 the previous paper [INTS00, NTS01], 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].

In this paper, we extend the result of the previous paper and 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 4. For example, the problem is shown to be deterministic EXP-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. We will omit some of the detailed proofs due to space limitation and refer to [NITS01] for the details.

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. Section 4 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, M99]. 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]). 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.

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 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 be further 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 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 2.2.

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]).

2.2 State and Trace

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 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. 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:

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

□

A language L_ϕ in a node *check*(L_ϕ) 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_ϕ), 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 often use a regular expression until Section 4.

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*(ϕ)) where:

$$JDK(\phi) = (NO^*(Priv \cap \phi) \cup \epsilon)\phi^* \qquad (1)$$

and ϵ is the empty sequence. □

3 The Verification 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. □

For a verification property L_ψ, Let $L_{\text{safe}}[\psi]$ be the set of state sequences α such that every state in α belongs to L_ψ. $L_{\text{safe}}[\psi]$ 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: $[\![P]\!] \subseteq L_{\text{safe}}[\psi]$?

We show an example of the verification problem for electronic commerce. As shown in Theorem 3.1, 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.

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 2.2. In a similar way, the property $JDK(P_{Canpay})$ specified in node n_{10}, $JDK(P_{Credit})$ in n_{18},

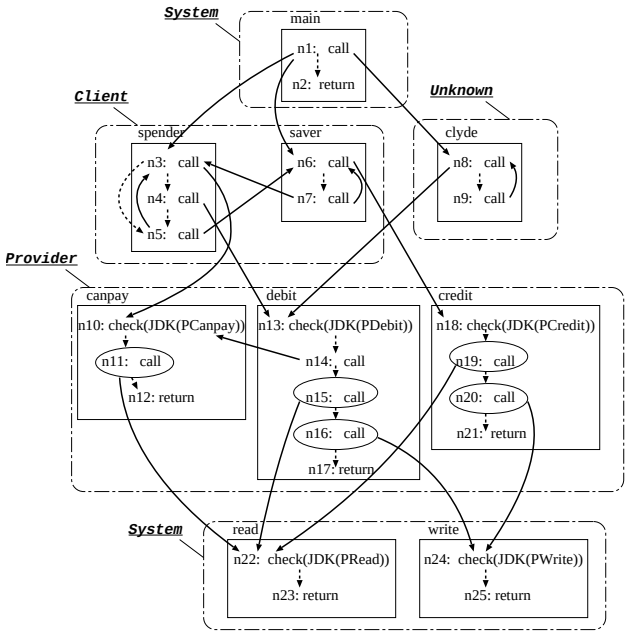

Figure 1: The flow graph of the example

$JDK(P_{Read})$ in n_{22}, and $JDK(P_{Write})$ in n_{24} are represented by regular expressions.

Consider the following two sequences:

$$\begin{aligned}\alpha_1 \;=\;& n_1 \triangleright n_1n_3 \triangleright \underline{n_1n_3n_{10}} \triangleright n_1n_3n_{11} \triangleright \underline{n_1n_3n_{11}n_{22}}\\ & \triangleright n_1n_3n_{11}n_{23} \triangleright n_1n_3n_{12} \triangleright n_1n_4 \triangleright \underline{n_1n_4n_{13}}\\ & \triangleright n_1n_4n_{14} \triangleright \underline{n_1n_4n_{14}n_{10}} \triangleright n_1n_4n_{14}n_{11}\\ & \triangleright \underline{n_1n_4n_{14}n_{11}n_{22}} \triangleright n_1n_4n_{14}n_{11}n_{23} \triangleright n_1n_4n_{14}n_{12}\\ & \triangleright n_1n_4n_{15} \triangleright \underline{n_1n_4n_{15}n_{22}} \triangleright n_1n_4n_{15}n_{23} \triangleright n_1n_4n_{16}\\ & \triangleright \underline{n_1n_4n_{16}n_{24}} \triangleright n_1n_4n_{16}n_{25} \triangleright n_1n_4n_{17} \triangleright n_1n_5,\\ \alpha_2 \;=\;& n_1 \triangleright n_1n_8 \triangleright \underline{n_1n_8n_{13}} \triangleright n_1n_8n_{14}.\end{aligned}$$

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 belongs to its security property. Therefore, $\alpha_1 \in [\![P]\!]$. For the sequence α_2, $n_1n_8n_{13} \notin JDK(P_{Debit})$ 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 $E_{Read} = \{n_{22}, n_{23}\}$ and $E_{Write} = \{n_{24}, n_{25}\}$. 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.

In the previous paper [INTS00, NTS01], we showed the verification problem is decidable. The outline of the proof is as follows.

(a) 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 can construct an indexed grammar[A68] $G_{P,\mathrm{T}}$ which generates the set of unchecked traces in P (see Appendix).

(b) An unchecked trace α is also a trace if α satisfies the properties specified in check nodes in α. We can 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 specified in check nodes (see Appendix). That is, $[\![P]\!] = L(G_{P,\mathrm{T}}) \cap L_{P,\mathrm{C}}$ where $L(G)$ is the language generated by a grammer G.

(c) By (b), the verification problem is equivalent to deciding whether $L(G_{P,\mathrm{T}}) \cap L_{P,\mathrm{C}} \cap \overline{L_{\text{safe}}[\psi]} = \emptyset$. 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.

Theorem 3.1 *For a given program P and a given verification property L_ψ, the verification problem $[\![P]\!] \subseteq L_{\text{safe}}[\psi]$ is decidable.* □

4 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. 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 $check(L_{\phi_i})$ ($1 \le i \le k$) 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 a grammar G, let $\|G\|$ denote the description length of G. 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 4.1[A68] *(a) For an indexed grammar G and a NFA M, an indexed grammar 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 indexed grammar G is solvable in deterministic $O(2^{p(\|G\|)})$ time for a polynomial p.* □

Lemma 4.2 *Let $P = (NO, IS, IT, TG, CG)$ be a program. Assume that P contains $check(L_{\phi_i})$ ($1 \le i \le k$) 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 sketch. As mentioned at the end of Section 3, the problem is equivalent to deciding whether

$$L(G_{P,\mathrm{T}}) \cap L_{P,\mathrm{C}} \cap \overline{L_{\mathrm{safe}}[\psi]} = \emptyset \qquad (2)$$

where $G_{P,\mathrm{T}}$ is the indexed grammar which generates unchecked traces, $L_{P,\mathrm{C}}$ is the regular language defined from L_{ϕ_i} ($1 \leq i \leq k$) and $L_{\mathrm{safe}}[\psi] = (L_\psi \triangleright)^* L_\psi$ (see Appendix for the formal definition of $G_{P,\mathrm{T}}$ and $L_{P,\mathrm{C}}$). We can show the following properties.

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

(ii) Let $n_1 = \max\{\#M_{\phi_1}, \ldots, \#M_{\phi_k}\}$. A NFA $M_{P,\mathrm{C}}$ can be constructed such that $L_{P,\mathrm{C}} = L(M_{P,\mathrm{C}})$ and $\#M_{P,\mathrm{C}}$ 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 $\#M_{\overline{\mathrm{safe}}} = n_2$.

Hence, by Lemma 4.1 (a), an indexed grammar G_P can be constructed such that $L(G_P) = L(G_{P,\mathrm{T}}) \cap L(M_{P,\mathrm{C}}) \cap L(M_{\overline{\mathrm{safe}}}) = L(G_{P,\mathrm{T}}) \cap L_{P,\mathrm{C}} \cap \overline{L_{\mathrm{safe}}[\psi]}$ and $\|G_P\|$ is $O((|NO|^2 + |TG| + |CG|)(|NO| \cdot n_1 \cdot n_2)^3)$, which is a polynomial order of $\|P\|$ and n_2. By (2), the verification problem is equivalent to deciding whether $L(G_P) = \emptyset$. By Lemma 4.1 (b), this 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 4.2, 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 4.3 *Let $P = (NO, IS, IT, TG, CG)$ be a program and L_ψ a verification property which satisfies the assumption stated in Lemma 4.2. The verification problem for P and L_ψ is DEXP-POLY time-complete.*

Proof sketch. By Lemma 4.2, 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. □

In the proof of Theorem 4.3, the language in each check node of P_M is specified by a NFA. 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 4.4 *Let $P = (NO, IS, IT, TG, CG)$ be a program and L_ψ a verification property which satisfies the assumption stated in Lemma 4.2 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.*

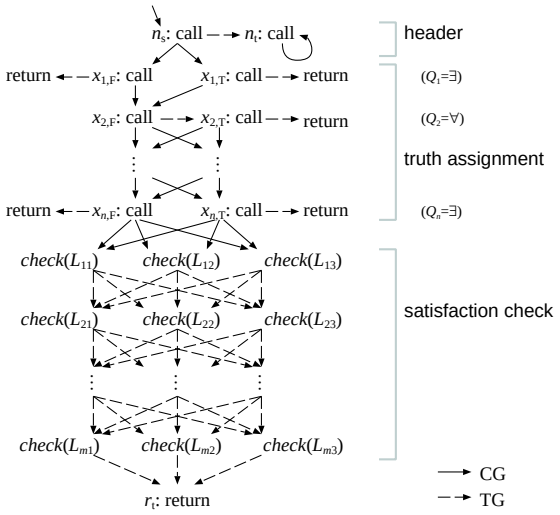

Figure 2: The program P_{Q3SAT} to solve QUANTIFIED 3SAT

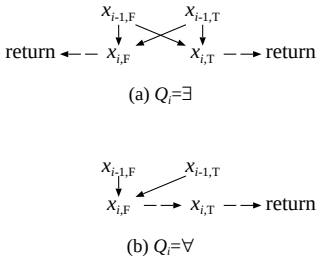

Figure 3: Edges in the truth assignment-part of P_{Q3SAT}

Proof. We transform QUANTIFIED 3-SATISFIABILITY (QUANTIFIED 3SAT) problem to the verification problem. An instance of QUANTIFIED 3SAT is a 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 "∃" or "∀."

The program P_{Q3SAT} constructed by the transformation consists of three parts: *header*, *truth assignment*, and *satisfaction check* (Figure 2). 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 every $V, V' \in \{\mathrm{F}, \mathrm{T}\}$ if Q_i is "∃" (Figure 3). 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 each $V \in \{\mathrm{F}, \mathrm{T}\}$ 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, $[\![P_{\text{Q3SAT}}]\!] \not\subseteq L_{\text{safe}}[\psi]$ for $L_\psi = (NO - \{n_t\})^*$ 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 4.5 *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. Define the set $[P]$ of all states reachable from the initial state of P as:

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

It is easy to see that $[P] \subseteq L(M_\psi)$ if and only if $[\![P]\!] \subseteq L_{\text{safe}}[\psi]$.

We can construct a regular grammar G_P such that $L(G_P) = [P] \cap \overline{L(M_\psi)}$ and $\|G_P\|$ 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 $L(G_P) = \emptyset$ in polynomial time in $\|P\|$ and $\#M_\psi$.

□

As is the case with Lemma 4.2, 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. For a regular expression E, let $L(E)$ be the language denoted by E.

Proposition 4.6 *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 4.5, we can construct a NFA $M_{P,S}$ such that $[P] = L(M_{P,S})$ and $\#M_{P,S} = O(|NO|)$. We can also construct a NFA $M_{P,S,\psi}$ such that $L(M_{P,S,\psi}) = L(M_{P,S}) \cap L(M_\psi)$ and $\#M_{P,S,\psi} = \#M_{P,S} \cdot \#M_\psi$. This construction of $M_{P,S}$ and $M_{P,S,\psi}$ completes the transformation since $[P] \not\subseteq L(M_\psi)$ iff $[P] \neq [P] \cap L(M_\psi)$ iff $L(M_{P,S}) \neq L(M_{P,S,\psi})$.

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 4, 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 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 $\Sigma^* \neq L(E)$. □

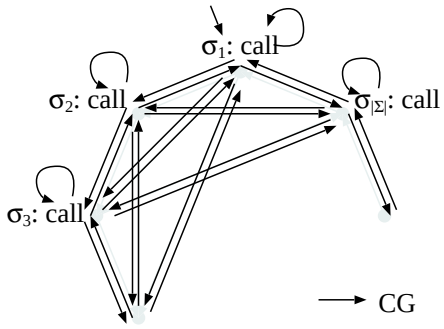

Figure 4: The program P_{Σ^*}

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. Below we briefly analyze their 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 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 4.4 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 4.4 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].

5 Conclusion

In this paper, we analyze the computational complexity of the verification problem of deciding whether a given program which may contain stack inspection satisfies a given security property. 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.

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

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.

[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.

[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.

[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.

[NITS01] N. Nitta, S. Ikada, Y. Takata and H. Seki: Decidability and complexity of the security verification problem for programs with stack inspection, Technical Report, Nara Institute of Science and Technology, 2001, http://www.aist-nara.ac.jp/~naoya-n/papers/nits01.ps.

[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.

[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.

Appendix

We present the definition of indexed grammar and the proof outline of the decidability of the verification problem, which is referred to in the proof of Lemma 4.2.

A.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 N is a finite set of *nonterminal symbols*, T is a finite set of *terminal symbols*, I is a finite set of *index symbols*, $S \in N$ is the *start symbol*, and R is a finite set of *productions* of one of the following forms: (1) $A \to \alpha$ (2) $A \to Bf$ (3) $Af \to \alpha$, 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 $\underset{G}{\to}$ on $(NI^* \cup T)^*$ is defined as the least relation which satisfies the following conditions (1), (2) and (3). In the following, let $\beta, \gamma \in (NI^* \cup T)^*$, $\xi \in I^*$ and $X_i \in N \cup T$. The empty sequence is denoted as ϵ.

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

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

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,

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

(3) Let $Af \rightarrow X_1X_2\cdots X_k \in R$ be a Type 3 production. Then,

$$\beta Af\xi\gamma \underset{G}{\rightarrow} \beta X_1\xi_1X_2\xi_2\cdots X_k\xi_k\gamma$$

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 $\overset{*}{\underset{G}{\rightarrow}}$. The language generated by an IG $G = (N, T, I, R, S)$ is defined as $L(G) = \{w \in T^* \mid S \overset{*}{\underset{G}{\rightarrow}} w\}$.

A.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:

$$S \rightarrow W\$ \quad (3)$$
$$W \rightarrow A\dot{n_1} \text{ for } n_1 = IT \quad (4)$$
$$A \rightarrow B \triangleright C \quad (5)$$
$$A \rightarrow B \quad (6)$$
$$B\dot{n_i} \rightarrow Bn_i \text{ for } \forall n_i \in NO \quad (7)$$
$$B\$ \rightarrow \epsilon \quad (8)$$
$$C\dot{n_i} \rightarrow N_i \text{ for } \forall n_i \in NO \quad (9)$$
$$N_j'' \rightarrow A\dot{n_k} \text{ for } \forall n_j, n_k \text{ such that } n_j \overset{TG}{\rightarrow} n_k \quad (10)$$

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*:

$$N_i \rightarrow N_i'\dot{n_i} \quad (11)$$
$$N_i' \rightarrow A\dot{n_j} \text{ for } \forall n_j \text{ such that } n_i \overset{CG}{\rightarrow} n_j \quad (12)$$

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

$$N_i\dot{n_j} \rightarrow N_j'' \text{ for } \forall n_j \in NO \quad (13)$$

(iii) $IS(n_i) =$ *check*(L_ϕ):

$$N_i \rightarrow A\dot{n_j} \text{ for } \forall n_j \text{ such that } n_i \overset{TG}{\rightarrow} n_j \quad (14)$$

It is clear from production (14) that $L(G_{P,\mathrm{T}})$ is the set of unchecked traces in P.

A.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_{ϕ_i}), then $s \in L_{\phi_i}$ holds.

For a check node n_i, let $L_{P,\mathrm{C}}^{(i)} = NO^*(NO - \{n_i\}) \cup \epsilon \cup L_{\phi_i}$. We can define the language $L_{P,\mathrm{C}}$ consisting of state sequences which satisfy the above condition for all check nodes as follows:

$$L_{P,\mathrm{C}} = (X\triangleright)^* NO^* \quad (15)$$

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

A.4 Results

Theorem A.1 *For each program P, $[\![P]\!] = L(G_{P,\mathrm{T}}) \cap L_{P,\mathrm{C}}$.* □

Lemma A.2 [A68] *The class of indexed languages is closed under intersection with regular languages. The emptiness problem for indexed languages is decidable.* □

Theorem 3.1 *For a given program P and a given verification property L_ψ, the verification problem $[\![P]\!] \subseteq L_{\mathrm{safe}}[\psi]$ is decidable.* □