Type-Based Information Flow Analysis for Low-Level Languages

Naoki Kobayashi and Keita Shirane

Department of Computer Science, Tokyo Institute of Technology
kobayasi@cs.titech.ac.jp, shirane@fuji.cs.titech.ac.jp

Abstract. A static program analysis called information flow analysis has been studied for high-level programming languages, to check that programs do not leak information about secret data such as passwords. The goal of this research is to establish a type-based method for information flow analysis for low-level languages such as assembly languages and virtual machine languages, so that information flow analysis can be performed even if source programs do not exist. Taking a subset of the Java virtual machine language as a target language, we formalize a type system for information flow analysis and implement an information flow analyzer based on the type system.

1 Introduction

Programs may manipulate secret data like passwords. In such cases, we usually prevent secret information from being leaked (or altered) by restricting the execution authority of those programs. But, if the programs are wrong, this measure is insufficient. For example, the program may wrongly writes passwords on a file which anyone can read.

To solve the problem above, a static program analysis called information flow analysis has been studied to statically check whether programs leak secret information or not [2, 3, 11, 14, 17]. They analyze how information about high-security data is propagated. For example, consider the following assignment statement.

$$non\_secret := \textbf{if } passwd < s \textbf{ then } 1 \textbf{ else } 2$$

Here, suppose that the variable *passwd* holds a string of passwords, s holds a string, the binary operator $<$ compares two strings. Since one can obtain a part of information about *passwd* by reading the value assigned to *non_secret*, we consider that information about *passwd* propagates (flows) into *non_secret*. If anyone can read the variable *non_secret*, the statement above is rejected (or warned) by information flow analysis because there is a possibility of information leakage. In fact, if the statement above is part of a procedure that can be called from the outside and s is an argument to the procedure, one can easily obtain the value of *passwd* by calling the procedure many times.

Information flow analysis has so far been mainly studied for high-level languages such as procedural languages and functional languages. For example, Denning [2] and Volpano et al. [17] proposed methods for information flow analysis for a procedural language consisting of if-statement, while-statement and assignment statements, etc. Heintze and Riecke [3] proposed a type system for information flow analysis for the λ-calculus. Smith and Volpano [14] and Honda et al. [4, 5] studied information flow analysis for concurrent languages.

In this paper, we propose type-based information analysis for a low-level language and prove its correctness An advantage of information flow analysis for a low-level language is that the analysis can be performed even if source programs are not available (as is often the case for libraries). We use a subset of the Java virtual machine language as the target language of our analysis. We briefly explain the idea of our information flow analysis below.

The basic idea of our type-based information flow analysis is the same as that of Volpano et al.[17] and Heintze and Riecke[3]: We extend the usual types with information about the security level of data. For example, we split the integer type **Int** into $\mathbf{Int_H}$, describing integers whose security level is high, and $\mathbf{Int_L}$, describing integers whose security level is low. Based on the extended types, we modify typing rules. For example, the usual typing rule for if-expressions is:

$$\frac{\Gamma \vdash b : \mathbf{Bool} \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathbf{if}\ b\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : \tau}$$

This rule is refined as follows.

$$\frac{\begin{array}{c}\Gamma \vdash b : \mathbf{Bool}_{\kappa_1} \qquad \Gamma \vdash e_1 : \tau_{\kappa_2} \qquad \Gamma \vdash e_2 : \tau_{\kappa_3} \\ \kappa_1, \kappa_2, \kappa_3 \sqsubseteq \kappa_4\end{array}}{\Gamma \vdash \mathbf{if}\ b\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : \tau_{\kappa_4}}$$

Here, τ_κ denotes the type of values whose security level is κ. The condition $\kappa_1, \kappa_2, \kappa_3 \sqsubseteq \kappa_4$ says that the security level of the return value of if-expression must be greater than or equal to not only the security levels of the values of e_1 and e_2 but also the security level of the value of the boolean expression b used in the branch. In this manner, we can construct a type system satisfying the property that well-typed programs do not leak secret information. Therefore, the problem of information flow analysis is reduced to the problem of type-checking or type inference.

The following new difficulties arise, however, in the case of information flow analysis for low-level languages: (1) the types of the values stored at the same variable (or register) may change during program execution, and (2) there are non-structured branch instructions (such as the goto instructions). We solve the first problem by assigning a different type to each variable for each program point (each instruction address), following Stata and Abadi's type system [15] for JVML. To understand the second problem, let us consider the if-expression. **if** b **then** e_1 **else** e_2. If there are no jump instructions, information about the value of the expression b propagates only to the result of the if-expression and the variables assigned in e_1, e_2. But, if there are jump instructions, it is not obvious where the value of b propagates. For example, consider the following program.

$$\begin{aligned}&1: x := 0;\\&2: x := x+1;\\&3: \textbf{if } b \textbf{ goto } 2;\\&4: \cdots\end{aligned}$$

Here, the left column denotes line numbers. At the third line, the control jumps to the second line if b is true, and otherwise the control jumps to the fourth line. In this case, we must consider that information about b propagates to the variable x assigned at the second line (since one can obtain information about b by reading the value of x). To deal with this problem, we statically estimate the range where information about the result of each branch is propagated and use the estimated range in typing rules.

Another novel point of the present paper is that our type system guarantees the correctness of information flow analysis in a stronger sense than the previous information flow analyses. Most of the previous information flow analyses uses, as a criterion of the correctness of information flow analysis, non-interference [3, 17], which says that low-security output data does not depend on high-security input data (therefore, we can not guess information about high-security data from low-security output data). Even if this property holds, however, secret information may be leaked from the program execution time. For example, consider the following expression:

$$\textbf{if } b \textbf{ then } \mathit{long\_computation} \textbf{ else } 1$$

Suppose that *long_computation* returns 1 after a long time. One can guess the value of b if one can observe the execution time. Our type system can prevent such leakage of information from the execution time. Agat[1] also presented a method for preventing such information leakage for high-level languages, but it uses program transformation, while we do not need program transformation.

The rest of this paper Section 2 introduces our target language. Section 3 presents our type system for information flow analysis, and Section 4 discusses type inference. Section 5 discusses the related work and Section 6 concludes this paper.

2 Target language *BL*

We introduce our target language *BL* for information flow analysis. *BL* is a subset of the Java virtual machine language JVML, obtained from removing objects, subroutines, etc. We believe that it is not difficult to extend the method in this paper to deal with objects and subroutines.

In *BL*, a program is a sequence of instructions, which operate a virtual machine similar to that of Java. The virtual machine consists of an operand stack and local variables. Each instruction applies an arithmetic operation to data on the operand stack and moves data between the stack and local variables. Here, we consider only integers as basic data types.

2.1 The syntax of *BL*

We write **Adr**, **LVar** and **SVar** for the set of instruction addresses, the set of names of local variables and the set of stack addresses respectively. These are subsets of the set of natural numbers.

Definition 1. The set $\mathcal{I}$ of *instructions*, ranged over by I, is given by:

$$\begin{aligned} I ::= & \ \texttt{inc} \mid \texttt{pop} \mid \texttt{push0} \mid \texttt{load}\ x \\ & \mid \texttt{store}\ x \mid \texttt{if}\ l \mid \texttt{goto}\ l \mid \texttt{return} \end{aligned}$$

We use a meta-variable l to denote an instruction address, a meta-variable x to denote a local variable.

Intuitive meaning of each instruction is as follows. Instruction `inc` increments the integer stored at the top of the operand stack. Instruction `pop` pops a value from the operand stack and `push0` pushes the integer 0 onto the operand stack. Instruction `load` x pushes the value stored in local variable x onto the operand stack, and `store` x removes the top value form the operand stack and stores the value into local variable x. Instruction `if` l pops the top value from the operand stack and jumps to the address l if the value is 0, and otherwise proceeds to the next address. Instruction `goto` l jumps to address l. Instruction `return` returns from the current method.

Definition 2 (security levels). The set of *security levels* is $\{\mathbf{L}, \mathbf{H}\}$. A binary relation $\sqsubseteq$ on security levels is the total order satisfying $\mathbf{L} \sqsubseteq \mathbf{H}$.

We use a meta-variable κ to denote a security level. For simplicity, we have only two security levels: a high security level **H** and a low security level **L**. Data of security level **H** are classified, and should not be revealed to non-privileged principals.

We define a binary operation $\sqcup$ on security levels as follows.

$$\kappa_1 \sqcup \kappa_2 = \begin{cases} \kappa_1 & \kappa_2 \sqsubseteq \kappa_1 \\ \kappa_2 & \text{otherwise} \end{cases}$$

Definition 3. A *method body*, denoted by B, is a mapping from a finite subset of **Adr** to $\mathcal{I}$. A *method*, denoted by the meta-variable M, is a pair consisting of security levels and a method body $((\kappa_1, \cdots, \kappa_n,\ \kappa_r),\ B)$.

A tuple of security levels $(\kappa_1, \ldots, \kappa_n,\ \kappa_r)$ in a method means that the number of argument is n, that the security level of the i-th argument (stored into local variable i) is κ_i and that the security level of the return value is κ_r. If $\kappa_r = \mathbf{H}$, we assume that an observer of low-level security cannot know whether the execution of a method terminates or not. [1]

[1] If we disclose information about the termination, we need to modify our type system in Section 3 so that only values of security level **L** can be checked in `if` instructions.

For example, a method $((\mathbf{H}, \mathbf{L}, \mathbf{L}), B)$ denotes that it takes two arguments, the first of which has security level $\mathbf{H}$ and the second of which has $\mathbf{L}$, and returns a value of low security level. (Therefore, this method must not return a value that includes information about the first argument.) We consider only a program consisting of a single method in this paper. Therefore, we identify a method with a program.

Example 1. Consider the method $((\mathbf{H}, \mathbf{L}), B)$ whose body B is shown in Figure 1. At address 2, it checks whether the first argument is 0 or not. If the value is

l	$B(l)$	Meaning
1	`load` 1	load the first argument
2	`if` 6	if the loaded value is not 0, then jump to address 6
3	`push0`	push 0 onto the stack
4	`store` 2	store 0 in variable 2
5	`goto` 10	jump to address 10
6	`push0`	push 0 onto the stack
7	`inc`	increase the stack top
8	`store` 2	store 1 into variable 2
9	`goto` 10	jump to address 10
10	`load` 2	load the value of variable 2
11	`return`	terminate this method, and return the stack top value

Fig. 1. A example of method body

0, it stores the integer 0 into variable 2 at addresses 3 and 4. If not, it stores the integer 1 at addresses 6–8. An observer can find information about the first argument by reading the return value, since the return value depends on the value of the first argument. This method is incorrect (in the sense that it leaks secret information) because the security level of the first argument is $\mathbf{H}$ while the security level of the return value is $\mathbf{L}$. On the other side, for the method body B above, the methods $((\mathbf{H}, \mathbf{H}), B)$ and $((\mathbf{L}, \mathbf{L}), B)$ are correct. In Section 3, we introduce a type system to make this kind of judgement automatically.

Notation 1 We write $A \rightharpoonup B$ for the set of partial maps from the set S_1 to the set S_2. We write $dom(f)$ for the domain of function f, $f \backslash x$ for the function obtained by excluding x from the domain of the function f, and $\emptyset$ for the function whose domain is the empty set. $f\{x \mapsto v\}$ denotes the function defined by:

$$dom(f\{x \mapsto v\}) = dom(f) \cup \{x\}$$
$$(f\{x \mapsto v\})(y) = \begin{cases} v & y = x \\ f(y) & y \neq x \end{cases}$$

2.2 The operational semantics of *BL*

We express a state of an execution as a tuple (pc, f, s), where pc ($\in$ **Adr**) is an instruction address, f ($\in \mathbf{LVar} \rightharpoonup \mathbf{Nat}$) is a function from the finite set of names of local variables to the set of natural numbers, s ($\in \mathbf{SVar} \rightharpoonup \mathbf{Nat}$) is a function from the subset of stack addresses $\{i \mid 0 \leq i < n\}$ to the set of natural numbers. pc, $f(x)$ and $s(i)$ denotes the current instruction address, the value stored in local variable x and the value stored at the i-th position of the stack, respectively. We call pc *a program counter*, f *a frame* (or an *environment*) and s *a stack state* (or simply a *stack*) respectively.

The relation $(pc, f, s) \longrightarrow_B (pc', f', s')$ means that that the state (pc, f, s) is changed to (pc', f', s') by a single step of execution. We write $\longrightarrow_B^*$ for the reflexive and transitive closure of $\longrightarrow_B$, and write $\longrightarrow_B^n$ for n steps execution. We also write $\longrightarrow_B^{\leq n}$ for $\cup_{i \leq n} \longrightarrow_B^{\leq i}$. These relations are formally defined in the full version of this paper [7].

3 Type System

This section presents a type system to check whether programs leak secret information. Well-typed programs are guaranteed not to leak secret information.

As described in Section 1, the main idea of our type system is to augment the standard type of a value with its security level. Following Stata and Abadi's type system [15], our type system assigns a type to each local variable and stack location for each instruction address. In addition, we assign a security level to each instruction address, which expresses the security level of information about whether the address is executed or not. In Example 1, **H** is assigned as the security level at address 6, since by checking whether the address 6 is executed, one can get information about the value of the first argument (whether it is 0 or not).

In Section 3.1, we introduce the types of values, a stack and local variables. Section 3.2 introduces a notion of *control information propagation*, which is important for dealing with the `if` instruction. Section 3.3 gives the typing rules and Section 3.4 shows the soundness of our type system. We need to perform type inference to check whether a given method leaks secret information. It is discussed in Section 4.

3.1 Types and type environments

Definition 4. The set of *types* is $\{\mathbf{Int_L}, \mathbf{Int_H}, \top\}$.

$\mathbf{Int}_s$ is the type of integers whose security level is s. $\top$ is the type of values that cannot be used at all.

The relation $\tau_1 \leq \tau_2$, defined below, means that a value of type τ_1 can be used as a value of type τ_2.

Definition 5. The *subtype* relation $\leq$ is the total order on types defined by:

$$\mathbf{Int_L} \leq \mathbf{Int_H} \leq \top$$

If $\tau_1 \leq \tau_2$, we call τ_1 a subtype of τ_2.

A *frame type*, denoted by a meta-variable F, is a mapping from the finite set of variables to types. $F(x)$ denotes the type of the value stored in local variable x. A *stack type*, denoted by a meta-variable S, is a mapping from the finite set of stack addresses $\{i \mid 0 \leq i < n\}$ to types. $S(k)$ is the type of the value stored at the k-th position of the stack (the 0-th position is the top). $\tau \cdot S$ is the types of the stack defined by $(\tau \cdot S)(0) = \tau, (\tau \cdot S)(n+1) = S(n)$. A *frame type environment*, denoted by a meta-variable $\mathcal{F}$, is a mapping from a finite set of addresses to types of local variables. A *stack type environment*, denoted by $\mathcal{S}$, is a mapping from the finite set of addresses to types of the stack. An *address type environment*, denoted by a meta-variable $\mathcal{A}$, is a mapping from a finite set of addresses to the set of security levels. $\mathcal{A}(l)$ denotes the security level of information about whether the instruction at address l is executed or not. For example, $\mathcal{A}(l) = \mathbf{H}$ means that information about whether the instruction at address l is executed can be revealed only to principals of high security.

We write $F_1 \leq F_2$ when F_1 and F_2 satisfy the following condition.

$$dom(F_1) = dom(F_2) \wedge \forall x \in dom(F_1).(F_1(x) \leq F_2(x)).$$

We define the subtype relation $S_1 \leq S_2$ on types of the stack in the same way.

3.2 The range of propagation of control information: *depend*(l)

The security level of each instruction address (i.e., the security level of information about whether each instruction is executed) depends on the security levels of the values inspected at the past branches For instance, in the program of Example 1, since one can get information about the value inspected at address 2 by checking which of addresses 3–9 are executed, the security levels at address 3–9 must be greater than or equal to the security level of the level **H** of the value used in the branch at 2. On the other hand, the security levels at address 1 and 10 do not depend on the security level of the value inspected at address 2 because the instructions at address 1 and 10 are executed irrespectively of the result of the branch at address 2. We write *depend*(l) for the set of addresses whose security levels depend on the security level of the result of a branch at address l (i.e., the set of addresses such that leakage of information about they are executed or not may lead to leakage of the result of a branch at address l). We call *depend*(l) the *range of control information propagation.* In the example above, $depend(2) = \{3, 4, 5, 6, 7, 8, 9\}$. We define *depend*($l$) only informally. The formal definition is given in the full paper [7].

An *execution path* of a program is a sequence of addresses such that the program may be executed in that order. A *complete execution path* from l is either a finite execution path which begins with the next instruction of address l and ends with an address l' such that $B(l') = \texttt{return}$, or an infinite execution path that begins with the next instruction of address l. A *merge point* from instruction address l is an address included in all the complete execution paths from instruction address l. The *first merge point* from instruction address l is a merge point appearing first in some complete execution path from address l. If there exist more than one address satisfying the condition above, the first merge point is chosen to be the smallest one among those satisfying the condition above. If there exists a merge point, $depend(l)$ is defined to be the set of addresses which appear before the first merge point in some complete execution path from l. If there does not exist a merge point, $depend(l)$ is the set of addresses occurring in some complete execution path.

For instance, for the method of Example 1, complete execution paths from address 2 are 3.4.5.10.11 and 6.7.8.9.10.11. The set of merge points is $\{10, 11\}$ and the first merge point is 10. Therefore, $depend(2) = \{3, 4, 5, 6, 7, 8, 9\}$. For an address $l(\neq 2)$, $depend(l) = \emptyset$.

If we exclude infinite sequences from complete execution paths in the definition above, it corresponds to "immediate forward dominator"[10] used by optimizing compilers. Since secret information may be leaked from information about whether a program terminates or not, it is necessary to consider infinite execution paths.

3.3 Typing rules

In this section, we define the relation $\mathcal{F}, \mathcal{S}, \mathcal{A} \vdash M$, which denotes that M can be safely executed if the local variables, the stack, and the address have types $\mathcal{F}(l)$, $\mathcal{S}(l)$ and $\mathcal{A}(l)$ respectively at each address l of the method M. First, we define a relation $\mathcal{F}, \mathcal{S}, \mathcal{A}, l \vdash M$, which denotes that M is well-typed at each address.

Definition 6. $\mathcal{F}, \mathcal{S}, \mathcal{A}, l \vdash M$ is the least relation satisfying the rules in Figure 2.

In Figure 2, $\mathcal{F}_l$ and $\mathcal{S}_l$ are shorthand notations for $\mathcal{F}(l)$ and $\mathcal{S}(l)$ respectively. We explain the main rules below.

Rule(T-Inc): The first line states that the instruction at address l is `inc`. Since instruction `inc` adds 1 to the integer at the top of the stack, the type of the stack top $\mathcal{S}(l)(0)$ must be an integer type. The type of the stack top at $l+1$ must also be an integer type or $\top$. The security level of the stack top must be greater than or equal to the security level of the former value. These conditions are stated in the second line. Since instruction `inc` does not access values of local variables and the other values of the stack, $\mathcal{F}(l)$ is a subtype of $\mathcal{F}(l+1)$, as stated in the third line. The third line states that the security level of the top value of the stack must be greater than or equal to the security level at address l. The fourth line states that there exists an instruction at the next address.

Rule(T-Load): Since instruction `load` x does not change local variables, $\mathcal{F}(l)$ must be a subtype of $\mathcal{F}(l+1)$ as stated by the second line. Since the value of

$$\frac{\begin{array}{c} B(l) = \texttt{inc} \\ \mathcal{S}_l(0) = \mathbf{Int}_\kappa \qquad \mathcal{S}_l \leq \mathcal{S}_{l+1} \\ \mathcal{F}_l \leq \mathcal{F}_{l+1} \qquad \mathbf{Int}_{\mathcal{A}(l)} \leq \mathcal{S}_{l+1}(0) \\ l+1 \in \mathit{dom}(B) \end{array}}{\mathcal{F}, \mathcal{S}, \mathcal{A}, l \vdash ((\kappa_1, \ldots, \kappa_n, \kappa_r), B)} \text{ (T-INC)}$$

$$\frac{\begin{array}{c} B(l) = \texttt{store } x \\ S_l \leq \mathcal{F}_{l+1}(x) \cdot S_{l+1} \\ \mathbf{Int}_{\mathcal{A}(l)} \leq \mathcal{F}_{l+1}(x) \\ \mathcal{F}_l \backslash x \leq \mathcal{F}_{l+1} \backslash x \\ l+1 \in \mathit{dom}(B) \end{array}}{\mathcal{F}, \mathcal{S}, \mathcal{A}, l \vdash ((\kappa_1, \ldots, \kappa_n, \kappa_r), B)} \text{ (T-STR)}$$

$$\frac{\begin{array}{c} B(l) = \texttt{pop} \\ \mathcal{F}_l \leq \mathcal{F}_{l+1} \qquad \mathcal{S}_l \leq \tau \cdot \mathcal{S}_{l+1} \\ l+1 \in \mathit{dom}(B) \end{array}}{\mathcal{F}, \mathcal{S}, \mathcal{A}, l \vdash ((\kappa_1, \ldots, \kappa_n, \kappa_r), B)} \text{ (T-POP)}$$

$$\frac{\begin{array}{c} B(l) = \texttt{if } l' \\ \mathcal{F}_l \leq \mathcal{F}_{l+1} \qquad \mathcal{F}_l \leq \mathcal{F}_{l'} \\ \mathcal{S}_l \leq \mathbf{Int}_\kappa \cdot \mathcal{S}_{l+1} \qquad \mathcal{S}_l \leq \mathbf{Int}_\kappa \cdot \mathcal{S}_{l'} \\ \forall k \in \mathit{depend}(l). \kappa \sqsubseteq \mathcal{A}(k) \\ l+1, l' \in \mathit{dom}(B) \end{array}}{\mathcal{F}, \mathcal{S}, \mathcal{A}, l \vdash ((\kappa_1, \ldots, \kappa_n, \kappa_r), B)} \text{ (T-IF)}$$

$$\frac{\begin{array}{c} B(l) = \texttt{push0} \\ \mathcal{F}_l \leq \mathcal{F}_{l+1} \qquad \mathbf{Int}_{\mathcal{A}(l)} \cdot \mathcal{S}_l \leq \mathcal{S}_{l+1} \\ l+1 \in \mathit{dom}(B) \end{array}}{\mathcal{F}, \mathcal{S}, \mathcal{A}, l \vdash ((\kappa_1, \ldots, \kappa_n, \kappa_r), B)} \text{ (T-PUSH)}$$

$$\frac{\begin{array}{c} B(l) = \texttt{goto } l' \\ \mathcal{F}_l \leq \mathcal{F}_{l'} \qquad \mathcal{S}_l \leq \mathcal{S}_{l'} \\ l' \in \mathit{dom}(B) \end{array}}{\mathcal{F}, \mathcal{S}, \mathcal{A}, l \vdash ((\kappa_1, \ldots, \kappa_n, \kappa_r), B)} \text{ (T-GOTO)}$$

$$\frac{\begin{array}{c} B(l) = \texttt{load } x \\ \mathcal{F}_l \leq \mathcal{F}_{l+1} \\ \mathcal{F}_l(x) \cdot \mathcal{S}_l \leq \mathcal{S}_{l+1} \\ \mathbf{Int}_{\mathcal{A}(l)} \leq \mathcal{S}_{l+1}(0) \\ l+1 \in \mathit{dom}(B) \end{array}}{\mathcal{F}, \mathcal{S}, \mathcal{A}, l \vdash ((\kappa_1, \ldots, \kappa_n, \kappa_r), B)} \text{ (T-LOAD)}$$

$$\frac{\begin{array}{c} B(l) = \texttt{return} \\ \mathcal{S}_l(0) \leq \mathbf{Int}_{\kappa_r} \qquad \mathcal{A}(l) \sqsubseteq \kappa_r \end{array}}{\mathcal{F}, \mathcal{S}, \mathcal{A}, l \vdash ((\kappa_1, \ldots, \kappa_n, \kappa_r), B)} \text{ (T-RET)}$$

Fig. 2. Typing rules

local variable x is pushed, $\mathcal{F}(l)(x) \cdot \mathcal{S}(l) \leq \mathcal{S}(l+1)$ must hold. The fourth line states that the security level of the pushed value must be greater than or equal to the security level of address $l+1$.

Rule(T-If): Since the control jumps to $l+1$ or l', $\mathcal{F}(l)$ must be a subtype of $\mathcal{F}(l+1)$ and $\mathcal{F}(l')$ as stated by the second and third lines. $\mathcal{S}(l)$ must be a subtype of $\mathbf{Int}_\kappa \cdot \mathcal{S}(l+1)$ and $\mathbf{Int}_\kappa \cdot \mathcal{S}(l')$. Moreover, the security level of $depend(l)$ must be greater than or equal to the security level κ of the value inspected at l.

Rule(T-Ret): Since instruction `return` returns the integer of the stack top, the type of the value of the stack top must be a subtype of $\mathbf{Int}_{\kappa_r}$ and the security level κ_r of the return value must be greater than or equal to the security level of the address. The second and third lines express these conditions.

By using the relation above, $\mathcal{F}, \mathcal{S}, \mathcal{A} \vdash M$ is defined as follows.

Definition 7. The type judgment relation $\mathcal{F}, \mathcal{S}, \mathcal{A} \vdash M$ for a method is the least relation satisfying the following rules.

$$\frac{\begin{array}{c} \{1, \ldots, n\} \subseteq dom(\mathcal{F}(1)) \\ \forall j \in \{1, \ldots, n\}. \mathcal{F}(1)(j) = \mathbf{Int}_{\kappa_j} \\ \forall j \in dom(\mathcal{F}(1)) \backslash \{1, \ldots, n\}. \mathcal{F}(1)(j) = \top \\ \mathcal{S}(1) = \emptyset \\ \forall l \in dom(B). \mathcal{F}, \mathcal{S}, \mathcal{A}, l \vdash ((\kappa_1, \ldots, \kappa_n, \kappa_r), B) \end{array}}{\mathcal{F}, \mathcal{S}, \mathcal{A} \vdash ((\kappa_1, \ldots, \kappa_n, \kappa_r), B)}$$

If there exist $\mathcal{F}, \mathcal{S}, \mathcal{A}$ such that $\mathcal{F}, \mathcal{S}, \mathcal{A} \vdash M$ holds, we say that a method M is *well-typed*.

The first three lines state that the security levels of the types of local variables match the security levels $\kappa_1, \ldots, \kappa_n$ of arguments for the method. The fourth line states that the stack is empty at the first address. The fifth line states that the program is well-typed at each address.

3.4 Correctness of the type system

In this section, we show that a well-typed program satisfies the following correctness criteria:

- The high security part of an input data does not affect the low security part of the result of the program.
- If the execution result contains a value of low security, even if the high security part of the input data results is changed, the execution time changes only by a constant factor.

The first property, the non-interference property [3], is used as a standard criterion for the correctness of information flow analysis. Even if this property holds, secret information may be leaked from the execution time of a program. For example, we consider the following program.

$$\textbf{if } b \textbf{ then } \mathit{long\_computation} \textbf{ else } 1$$

Suppose that *long_computation* returns 1 after a long time. Even if one observes that the execution result of this program is 1, one cannot know the value of b. But, one can still guess the value of b by observing the execution time. As mentioned below, if the second property holds, we can prevent leakage of secret information from the execution time.

Below, we give a formal definition of the above correctness criteria. We first define equivalence relations on values and states.

Definition 8. A ternary relation $v_1 \sim_\kappa v_2$ between integers v_1, v_2 and a security level κ is defined by:.

$$v_1 \sim_\kappa v_2 \Leftrightarrow v_1 = v_2 \vee \kappa = \mathbf{H}$$

A ternary relation $v_1 \sim_\tau v_2$ between integers v_1, v_2 and a type τ is defined by:

$$v_1 \sim_\tau v_2 \Leftrightarrow (\tau = \top) \vee \exists \kappa.(\tau = \mathbf{Int}_\kappa \wedge v_1 \sim_\kappa v_2)$$

The above relation $v_1 \sim_\kappa v_2$ states that observers of security level $\mathbf{L}$ cannot distinguish v_1 from v_2 when the security levels of v_1, v_2 are κ (in other words, v_1 and v_2 must be the same value if the security levels of v_1, v_2 are $\mathbf{L}$)

We can state the correctness of our type system as follows by using the relations above.

Theorem 1 (correctness).
Suppose that a method $((\kappa_1, \ldots, \kappa_n, \kappa_r), B)$ is well-typed. Then there exists a constant c such that if (i) $(1, f_1, \emptyset) \longrightarrow_B^m (pc_1, f_1', s_1')$, (ii) $\kappa_r = \mathbf{L}$, (iii)$B(pc_1) =$ `return`, and (iv) $\forall i \in \{1, \ldots, n\}. f_1(i) \sim_{\kappa_i} f_2(i)$, then there exist pc_2, f_2', s_2' that satisfy the following conditions.

(i) $(1, f_2, \emptyset) \longrightarrow_B^{\leq cm} (pc_2, f_2', s_2')$
(ii) $B(pc_2) =$ `return`
(iii) $s_1'(0) \sim_{\kappa_r} s_2'(0)$

A proof of Theorem 1 is found in the full version of this paper [7].

By using the fact above, we can prevent leakage of information from the execution time by executing a program in the following steps. (We suppose that the time taken to each step of an execution is constant.)

(1) First, change all the high security input data to the constant 0 and execute the method. Count the number of execution steps and let it be n.
(2) Second, restore the input data and execute the program. Let m be the number of execution steps.
(3) After waiting for $cn - m$ more steps, return the execution result. (The above theorem implies that $cn - m$ is a nonnegative integer)

Note that the result is always returned after $(c + 1)n$ steps, irrespectively of high-security data. Therefore, one cannot guess secret information by observing the execution time of a program.

l	$B(l)$	$\mathcal{F}_l(1)$	$\mathcal{F}_l(2)$	$\mathcal{S}_l$	$\mathcal{A}(l)$	Constraints
1	`load 1`	$\mathbf{Int_H}$	$\top$	$\emptyset$	δ_1	$\mathbf{H} \sqsubseteq \alpha_2, \mathbf{H} \sqsubseteq \gamma_2, \delta_1 \sqsubseteq \gamma_2$
2	`if 6`	$\mathbf{Int}_{\alpha_2}$	$\top$	$\mathbf{Int}_{\gamma_2} \cdot \emptyset$	δ_2	$\alpha_2 \sqsubseteq \alpha_3, \alpha_2 \sqsubseteq \alpha_6, \forall k \in \{3,4,5,6,7,8,9\}.\gamma_2 \sqsubseteq \delta_k$
3	`push0`	$\mathbf{Int}_{\alpha_3}$	$\top$	$\emptyset$	δ_3	$\alpha_3 \sqsubseteq \alpha_4, \delta_3 \sqsubseteq \gamma_4$
4	`store 2`	$\mathbf{Int}_{\alpha_4}$	$\top$	$\mathbf{Int}_{\gamma_4} \cdot \emptyset$	δ_4	$\gamma_4 \sqsubseteq \beta_5, \delta_4 \sqsubseteq \beta_5, \alpha_4 \sqsubseteq \alpha_5$
5	`goto 10`	$\mathbf{Int}_{\alpha_5}$	$\mathbf{Int}_{\beta_5}$	$\emptyset$	δ_5	$\alpha_5 \sqsubseteq \alpha_{10}, \beta_5 \sqsubseteq \beta_{10}$
6	`push0`	$\mathbf{Int}_{\alpha_6}$	$\top$	$\emptyset$	δ_6	$\alpha_6 \sqsubseteq \alpha_7, \delta_6 \sqsubseteq \gamma_7$
7	`inc`	$\mathbf{Int}_{\alpha_7}$	$\top$	$\mathbf{Int}_{\gamma_7} \cdot \emptyset$	δ_7	$\gamma_7 \sqsubseteq \gamma_8, \delta_7 \sqsubseteq \gamma_8, \alpha_7 \sqsubseteq \alpha_8$
8	`store 2`	$\mathbf{Int}_{\alpha_8}$	$\top$	$\mathbf{Int}_{\gamma_8} \cdot \emptyset$	δ_8	$\gamma_8 \sqsubseteq \beta_9, \delta_8 \sqsubseteq \beta_9, \alpha_8 \sqsubseteq \alpha_9$
9	`goto 10`	$\mathbf{Int}_{\alpha_9}$	$\mathbf{Int}_{\beta_9}$	$\emptyset$	δ_9	$\alpha_9 \sqsubseteq \alpha_{10}, \beta_9 \sqsubseteq \beta_{10}$
10	`load 2`	$\mathbf{Int}_{\alpha_{10}}$	$\mathbf{Int}_{\beta_{10}}$	$\emptyset$	δ_{10}	$\alpha_{10} \sqsubseteq \alpha_{11}, \beta_{10} \sqsubseteq \beta_{11}, \beta_{10} \sqsubseteq \gamma_{11}, \delta_{10} \sqsubseteq \gamma_{11}$
11	`return`	$\mathbf{Int}_{\alpha_{11}}$	$\mathbf{Int}_{\beta_{11}}$	$\mathbf{Int}_{\gamma_{11}} \cdot \emptyset$	δ_{11}	$\gamma_{11} \sqsubseteq \mathbf{L}, \delta_{11} \sqsubseteq \mathbf{L}$

Table 1. Types and constraints for the method of Example 1

4 Type inference algorithm

By Theorem 1, well-typed programs do not leak secret information. Therefore, to check that a program M does not leak secret information, it is sufficient to check whether there exist $\mathcal{F}, \mathcal{S}, \mathcal{A}$ such that $\mathcal{F}, \mathcal{S}, \mathcal{A} \vdash M$ by performing type inference. As in the standard constraint-based type inference algorithms [6, 8], our type inference proceeds as follows.

(1) Based on the typing rules, generate constraints on types and security levels from each address of the method body.
(2) Reduce the constraints and check whether they are satisfiable.

We show an outline of the type inference by using the program of Example 1. For simplicity, we assume that type information except for security levels has been already obtained (for example, by using Stata and Abadi's type system[15]). In Table 1, we show type information except for security levels for the program $((\mathbf{H}, \mathbf{L}), B)$ in Example 1. Here, $\alpha_l, \beta_l, \gamma_l, \delta_l$ are variables denoting unknown security levels. By the rules in Figure 2, we can obtain constraints as shown in the rightmost column. For example, by the rule (T-Str) in Figure 2, we can generate the following constraints on address 4:

$$\begin{array}{l}\mathbf{Int}_{\alpha_4} \cdot \emptyset \leq \mathbf{Int}_{\beta_5} \cdot \emptyset \\ \mathbf{Int}_{\delta_4} \leq \mathbf{Int}_{\beta_5} \\ \{1 \mapsto \mathbf{Int}_{\alpha_4}\} \leq \{1 \mapsto \mathbf{Int}_{\alpha_5}\}\end{array}$$

By reducing these constraints, the constraints shown in the rightmost column of Table 1 are obtained.

Let $\boldsymbol{\rho}$ be $(\alpha_2, \alpha_3, \ldots, \delta_{11})$. Then, the constraints of Table 1 are expressed in the form

$$\begin{array}{l}\boldsymbol{g}(\boldsymbol{\rho}) \sqsubseteq \boldsymbol{\rho} \\ (\gamma_{11}, \delta_{11}) \sqsubseteq (\mathbf{L}, \mathbf{L})\end{array}$$

Here, $\sqsubseteq$ has been extended pointwise to the relation on tuples of security levels, and $\boldsymbol{g}$ is a monotonic function with respect to that relation. Therefore, there exists a natural number n such that $\boldsymbol{\rho} = \boldsymbol{g}^n(\mathbf{L}, \ldots, \mathbf{L})$ is the least solution of $\boldsymbol{g}(\boldsymbol{\rho}) \sqsubseteq \boldsymbol{\rho}$. (We can obtain the least solution by calculating the $\boldsymbol{g}^n(\mathbf{L}, \ldots, \mathbf{L})$ from $n = 0$ step by step and looking for n satisfying $\boldsymbol{g}^{n+1}(\mathbf{L}, \ldots, \mathbf{L}) = \boldsymbol{g}^n(\mathbf{L}, \ldots, \mathbf{L})$.) The solution for the above example satisfies $\gamma_{11} = \mathbf{H}$, which contradicts with the condition $\gamma_{11} \sqsubseteq \mathbf{L}$. Therefore, we can judge that the method $((\mathbf{H}, \mathbf{L}), B)$ of Example 1 is not well-typed. On the other hand, we can judge that $((\mathbf{H}, \mathbf{H}), B)$ is well-typed.

In general, we obtain constraints $\boldsymbol{g}(\boldsymbol{\rho}) \sqsubseteq \boldsymbol{\rho}$ and $\boldsymbol{g'}(\boldsymbol{\rho}) \sqsubseteq \boldsymbol{\kappa}$. So, we can check the satisfiability of the constraints by finding the least solution of the former and checking whether the solution satisfies the latter constraint.

Let the size of a method (the number of instructions) be n. Then the size of the constraints generated from each instruction is $O(n)$, and hence the size of all the constraints generated is $O(n^2)$. The constraints can be solved in time $O(n^2)$ by using the linear-time algorithm of Rehof et al. [13]. Therefore, the time complexity of type inference is $O(n^2)$.

5 Related work

As mentioned in Section 1, information flow analysis has been studied for procedural languages [2, 17], functional languages [3, 12, 18], object-oriented languages [11], concurrent languages [4, 14], etc. To the authors' knowledge, this is the first work to formally present information flow analysis for a low-level language like JVML.

Denning [2] briefly mentioned how to treat goto-statement as an extension of information flow analysis for procedural languages. But, since immediate forward dominator [10], which ignores infinite execution paths is used instead of the first merge point used in our type system, their method is not correct in our criterion for information flow analysis. For example, consider the method $((\mathbf{H}, \mathbf{L}), B)$ where body B is given by:

l	$B(l)$	Meaning
1	`load 1`	load the value of argument 1
2	`if 1`	if the loaded value is not 0, jump at address 1
3	`push0`	
4	`return`	

Since the immediate forward dominator is 3, this program is judged to be correct by their analysis. One can, however, obtain information about the argument 1 by observing whether the execution of the program above terminates (there exists the return value) or not.

The type system for information flow analysis of Myers [18] deals with global jump by introducing continuation for functional languages. Since a strong restriction is imposed on the usage of continuation in their type system, it is not obvious whether jump instructions as treated in this paper can be dealt with.

Methods to prevent leakage of information from execution time have been studied by Volpano and Smith [16] and Agat [1]. Volpano and Smith [16] showed a type system that ensures execution time is not influenced by high-security data at all. The restriction is so strong that high-security data cannot be inspected at any branches. Agat [1] proposed a method for hiding information about execution time by performing program transformation. This method inserts a dummy of the else-part in the then-part and that of the then-part in the else-part, so that execution times of the then-part and the else-part are the same for any if-statement inspecting high-security data. Unlike the method of inserting delays in this paper, therefore, their method suffers from wasteful computation. Moreover, it seems difficult to apply their method to low-level languages.

Our type system in this paper was obtained by extending Stata and Abadi's type system [15] for the Java virtual machine language with security levels. Their type system deals with subroutines. We think that it is not difficult to extend our type system to deal with subroutines.

6 Conclusion

We have proposed a new type system for information flow analysis for a subset of the Java virtual machine language. Our type system enables us to perform information flow analysis even if source programs are not available.

We plan to extend our type system to deal with object fields, method invocations, subroutines and thread primitives, and implement an information flow analyzer for JVML. Future work also includes development of type systems for information flow analysis for other low-level languages, such as the typed assembly language [9].

Acknowledgements

We thank members of "the priciple of programming languages group" at University of Tokyo and Tokyo Institute of Technology for useful discussions.

References

1. Johan Agat. Transforming out timing leaks. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 40–53, 2000.
2. Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.
3. Nevin Heintze and Jon Riecke. The slam calculus: programming with secrecy and integrity. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 365–377, 1998.
4. Kohei Honda, Vasco Vasconcelos, and Nobuko Yoshida. Secure information flow as typed process behaviour. In *Proc. of European Symposium on Programming (ESOP) 2000*, volume 1782 of *Lecture Notes in Computer Science*, pages 180–199. Springer-Verlag, 2000.

5. Kohei Honda and Nobuko Yoshida. A uniform type structure for secure information flow. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 81–92, 2002.
6. Atsushi Igarashi and Naoki Kobayashi. Type reconstruction for linear pi-calculus with I/O subtyping. *Information and Computation*, 161:1–44, 2000.
7. Naoki Kobayashi and Keita Shirane. Type-based information flow analysis for a low-level language. 2002.
8. Torben Mogensen. Types for 0, 1 or many uses. In *Implementation of Functional Languages*, volume 1467 of *Lecture Notes in Computer Science*, pages 112–122, 1998.
9. J. Gregory Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, 1999.
10. Steven S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufmann Publishers, 1997.
11. Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 228–241, 1999.
12. François Pottier and Vincent Simonet. Information flow inference for ML. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 319–330, 2002.
13. Jakob Rehof and Torben Mogensen. Tractable constraints in finite semilattices. *Science of Computer Programming*, 35(2):191–221, 1999.
14. Geoffrey Smith and Dennis Volpano. Secure information flow in a multi-threaded imperative language. In *Proceedings of ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages*, pages 355–364, 1998.
15. Raymie Stata and Martín Abadi. A type system for Java bytecode subroutines. *ACM Transactions on Programming Languages and Systems*, 21(1):90–137, 1999.
16. Dennis Volpano and Geoffrey Smith. Eliminating covert flows with minimum typings. In *Proc. 10th IEEE Computer Security Foundations Workshop*, pages 156–168, 1997.
17. Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.
18. Steve Zdancewic and Andrew C. Myers. Secure information flow via linear continuations. *Higher-Order and Symbolic Computation*, 2002. To appear. A preliminary version appeared in Proceedings of ESOP 2001, Springer LNCS 2028, pp.46–61.