Java Bytecode Dependence Analysis for Secure Information Flow

Gaowei Bian, Ken Nakayama, Yoshitake Kobayashi, and Mamoru Maekawa
(Corresponding author: Gaowei Bian)

Department of Information Systems, Graduate School of Information Systems
University of Electro-Communications, 1-5-1, Chofugaoka, Chofu-shi, Tokyo, 182-8585, Japan.
(Email: {bian, ken, yoshi, maekawa}@maekawa.is.uec.ac.jp)

(Received Aug. 19, 2005; revised and accepted Sept. 29, 2005)

Abstract

Java programs can be transmitted and executed on another host in bytecode format, thus the sensitive information of the host may be leaked via these assembly-like programs. Information flow policy can ensure data confidentiality, however, conventional information flow analysis mainly focused on the programs written in high-level programming languages and is generally performed by type checking approach, which assigns security classes to the variables then verifies information flow policy in program executing order. These approaches are inadequate to address the information flow in bytecode and the type systems verification method is imprecise. This paper presents a method to disclose java bytecode information flow by dependence analysis, in which the information flow analysis is separated to two phases to improve precision. First is determining information dependence relationship among the variables in the bytecode then is verifying the security based on security class. A prototype tool has been developed, by which the bytecode information flow of object or class files can be analyzed.

Keywords: Dependence analysis, information flow, Java Bytecode

1 Introduction

Bytecode, the program in Java virtual machine language (JVML), may be loaded over the network to a remote host, such as applets and mobile agents and may interact with the resources and facilities of the host. If the programs access the user's sensitive data and communicate over network, the private information could be released. The hosts have the option that protects sensitive information by using access control mechanism. However, this impairs the function of the programs, since useful programs generally need access host's data to perform their tasks.

To address data confidentiality problem, information flow [11] is proposed to enforce this property. By analyzing how information flows through the program, the data could be protected from leaking to public channel.

Denning first proposed a static certification method to verify the information flow security of a program [12]. Each program object is assigned a certain security class and the security classes are assumed to form a lattice structure, ordered by $\leq$. Their method is a type-based approach to enforce secure information flow. They only provided an informal argument that their approach is sound, that is, it certifies only secure programs. Vopano developed an elegant syntax-directed type system for annotating program variables, commands, and procedure parameters with security classes [16]. They also proved that their type system ensures noninterference. Banerjee and Naumann extended the type system of Volpano to support a more realistic object oriented sequential language [3]. Their extension encompasses data flow via mutable object fields and control flow via dynamically dispatched method calls. They prove noninterference for a language with pointers, mutable state, private fields, class-based visibility, dynamic binding and inheritance, type casts, type tests, and mutually recursive classes and methods. A weak form of noninterference is used which does not consider termination behavior to be observable because, unfortunately, strong noninterference for a sequential language requires loop guards to have low security which would lead to complications for conditionals that involve recursive calls. Type system approaches to secure information flow are simple to implement, but they are often too imprecise. Consider the example:

$$L := H; L := 0;$$

where L and H correspond to low and high security variables respectively. A security-type system would reject this program based on the first assignment, yet the program clearly satisfies noninterference. Most type-based approaches reject any program with insecure subprograms because they check the program line-by-line and the context of the program are ignored.

Avvenuti [2] developed a tool for Java bytecode verification for secure information flow. Cinzia [8] proposed an approach using standard bytecode verifier to ensure information flow. These approaches are also essentially an implementation of type system.

Most of these existing approaches use type checking method to enforce information flow policy, and the research about assembly-like stack-based bytecode is inadequate. Because Java programs are transmitted and executed in bytecode format and the remote host can not get the source code of program to verify the security, the information flow analysis for bytecode is necessary.

In the previous study of bytecode security [5, 6], the security model is proposed and the information flow security based on dependence analysis is introduced. In this paper the analyzing algorithm is improved and detail implementation of the approach is presented. A method that analyzes Java bytecode information flow to ensure data confidentiality is proposed. This approach analyzes bytecode information flow by disclosing information dependence relations among variables then checking the security policy by computing secure classes of the variables. Unlike type systems, which analyze and verify program in type-level command-by-command, our approach certifies the program after analyzing the whole program thus can provide more precision. When analyzing information dependence the security class, which depends on various host security policies, is not involved, so the dependence analysis can be performed and prepared beforehand. The fixed program such as API can be analyzed and archived to library.

The rest of the paper is organized as follows. Section 2 gives related research background. Section 3 is an introduction of our approach, including verification mechanism and an example. Section 4 introduces the prototype implementation and Section 5 is the conclusion.

2 Background

2.1 Java Bytecode and JVM

In Java, programs are being compiled into a binary format called bytecode which is a sequence of instructions of the machine language for Java virtual machine. The bytecode in a method are executed when that method is invoked during the course of running a program. Each instruction consists of a one-byte *opcode* specifying the operation to be performed, followed by zero or more *operands* supplying arguments or data used by the operation [14].

Figure 1 summarizes the instruction set of the Java virtual machine. v is an integer, real number or the special value null; x is a local variable; L is an instruction address; σ and τ are a class name and valid array component type respectively. A specific instruction, with type information, is built by replacing the T in the instruction template in the opcode column by the letter in the type column. For instance, *iload* reprensents loading an integer value, *aload* represents loading an object [14].

$$
\begin{aligned}
\text{Instruction} ::= {} & T\text{push } v \mid \text{pop} \mid T\text{store } x \mid T \text{ load } x \\
& \mid \textit{prim} \text{ op} \mid \text{ifeq } L \mid \text{goto } L \\
& \mid \text{new } \sigma \\
& \mid \text{invokevirtual } \textit{Method-Ref} \\
& \mid \text{invokeinterface } \textit{Interface-Method-Ref} \\
& \mid \text{invokespecial } \textit{Method-Ref} \\
& \mid \text{getfield } \textit{Field-Ref} \\
& \mid \text{putfield } \textit{Field-Ref} \\
& \mid \text{newarray } \tau \mid \text{arraylength} \\
& \mid \text{arrayload } \tau \mid \text{arraystore } \tau \\
& \mid \text{throw} \mid \text{jsr } L \mid \text{ret } x \\
& \mid \text{return} \mid T\text{return}
\end{aligned}
$$

Figure 1: JVM instruction set

Bytecode programs use method references, interface method references, and field references to identify methods, interfaces methods, and fields from Java language programs. These references contain three pieces of information about the method of field that they describe: the class of interface in which it was declared, the field or method name, and its type.

$$
\begin{aligned}
& Method-Ref \\
& \qquad ::= \{|Class-Name, Label, Method-Type|\} \\
& Interface-Method-Ref \\
& \qquad ::= \{|Interface-Name, Label, Method-Type|\} \\
& Field-Ref \\
& \qquad ::= \{|Class-Name, Label, Field-Type|\}
\end{aligned}
$$

JVM's execution state is a pair $< A, O >$ where A is the activation stack and O is the current state of the object. $O(r.f)$ denotes the contents of field f of object r. The semantics of the language is presented in Figure 2. An execution state of a method $C.mt$ is a tuple (B, i, M, S), where B is the bytecode corresponding to C.mt, i is the address held by the program counter, M: Registers $\rightarrow$ Values is the local memory, representing the current state of the local registers of B, and $S \in$ Values is the current state of the operand stack. Given x, the content of x in the memory M is denoted by $M(x)$. The initial state of the execution of a method C.mt is $(B, 0, M_0, \lambda)$, where 0 is the address of the first instruction, $M_0(x_0)$ and $M_0(x_1)$ are set to the reference to the object and to the actual parameter respectively. The set C is the class definitions and the set Objects $= \{r_1, \ldots r_n\}$ is the references to instances of the classes in C. $O = Objects \rightarrow ObjectValues$ denotes the domain of object valuations. $M[k/x]$ is used to indicate the memory M' which agrees with M for all registers, except for x, which is $M'(x) = k$. Similarly, $O[k/r.f]$ indicates the object valuation O', which differs from O only on field f of object r, which is assigned k.

2.2 Information Flow in Bytecode

The concept of secure information flow is typically formalized in terms of what is known as "noninterference". Noninterference states that confidential data may not interfere with, meaning affect, public data.

The information flow model can be defined by

$$FM =< N, P, SC, \oplus, \rightarrow>$$

N is a set of logical storage objects or information receptacles. Elements of N may be files, or program variables. P is a set of process. SC is a set of security classes corresponding to disjoint classes of information. The class-combining operator "*oplus*" is an associative and commutative binary operator. A flow "$\rightarrow$" relation is defined on pairs of security classes. For classes A and B, $A \rightarrow B$ means if and only if information in class A is permitted to flow into class B [18].

The security requirement of the model is that a flow model FM is secure if and only if execution of a sequence of operations cannot violate the relation "$\rightarrow$". To comply with this policy, information at a given security level is not allowed to flow to lower levels. A security system is composed of a set S of subjects and a disjoint set O of objects. Each subject $s \in S$ is associated with a fixed security class $C(s)$, denoting it clearance. Likewise, each object $o \in O$ is associated with a fixed security class $C(o)$, denoting its classification level. The security classes are partially ordered by a relation $\leq$, which forms a lattice.

Every program variable x has a security class denoted by $\underline{x}$. It is assumed that $\underline{x}$ can be determined statically and that it does not vary at run time. If x and y are variables and there is flow of information from x to y then it is a permissible flow if $\underline{x} \leq \underline{y}$.

Every programming construct has a certification condition. It is a purely syntactic condition relating security classes. Some of these conditions control explicit flows while others control implicit flows. For example, the statement $y := x$ has the condition $\underline{x} \leq \underline{y}$, which means the flow of information from the security class of x to that of y must be permitted by the flow policy. The conditions for other constructs, such as if statements, control implicit flows. For example, there is always an implicit flow form the guard of a conditional to its branches. In the statement

$$\text{if } x > y \text{ then } z := w \text{ else } i := i + 1$$

There is an implicit flow from x and y to z and i, So the statement has the certification condition $\underline{x} \vee \underline{y} \leq \underline{z} \wedge \underline{i}$ where $\vee$ and $\wedge$ denote least upper bound and greatest bound operators respectively. The lattice property makes it possible to enforce these conditions.

The bytecode shown in Figure 3 corresponds to a method mt of a class A. The value of the field f of a class B is loaded in instruction 3, which is tested by the branch instruction 4 and has not been assigned to a variable. The final value of $A.f$, 0 or 1, depends on the value used in instruction 3. Even without storing instruction, the information in $B.f$ still flows to $A.f$ implicitly.

$$\frac{B[i] = \text{push } n \ < (B, i, M, S) \cdot A, O >}{< (B, i+1, M, n \cdot S) \cdot A, O >}$$

$$\frac{B[i] = \textit{prim op} \ < (B, i, M, n1 \cdot n2 \cdot S) \cdot A, O >}{< (B, i+1, M, n \cdot S) \cdot A, O >}$$

$$\frac{B[i] = \text{pop} \ < (B, i, M, v \cdot S) \cdot A, O >}{< (B, i+1, M, S) \cdot A, O >}$$

$$\frac{B[i] = \text{load } x \ < (B, i, M, S) \cdot A, O >}{< (B, i+1, M, M(x) \cdot S) \cdot A, O >}$$

$$\frac{B[i] = \text{store } x \ < (B, i, M, v \cdot S) \cdot A, O >}{< (B, i+1, M[k/x], S) \cdot A, O >}$$

$$\frac{B[i] = \text{if}\textit{cond } j \ \textit{true} \ < (B, i, M, n \cdot S) \cdot A, O >}{< (B, i+j, M, S) \cdot A, O >}$$

$$\frac{B[i] = \text{if}\textit{cond } j \ \textit{false} \ < (B, i, M, v \cdot S) \cdot A, O >}{< (B, i+1, M, S) \cdot A, O >}$$

$$\frac{B[i] = \text{goto } j \ < (B, i, M, S) \cdot A, O >}{< (B, i+j, M, S) \cdot A, O >}$$

$$\frac{B[i] = \text{getfield } C.f \ < (B, i, M, r \cdot S) \cdot A, O >}{< (B, i+1, M, O(r.f) \cdot S) \cdot A, O >}$$

$$\frac{B[i] = \text{putfield } C.f \ < (B, i, M, k \cdot r \cdot S) \cdot A, O >}{< (B, i+1, M, S) \cdot A, O[k/r.f] >}$$

$$\frac{B[i] = \text{invoke } C.mt \ < (B, i, M, k \cdot r \cdot S) \cdot A, O >}{< (B', 0, M[r/x_0][k/x_1], \lambda) \cdot (B, i+1, M, S) \cdot A, O >}$$

$$\frac{B[i] = \text{return} \ < (B, i, M, k \cdot \lambda) \cdot (B', j, M', S) \cdot A, O >}{< (B', j, M', k \cdot S) \cdot A, O >}$$

$$\frac{B[i] = \text{load } x \ < (B, i, M, S) \cdot A, O >}{< (B, i+1, M, M(x) \cdot S) \cdot A, O >}$$

Figure 2: JVM instruction semantics

```
0:   iconst_0
1:   istore_2
2:   aload_1
3:   getfield   B.f
4:   ifne       8
5:   iconst_0
6:   istore_2
7:   goto       10
8:   iconst_1
9:   istore_2
10:  aload_0
11:  iload_2
12:  putfield   A.f
13:  iload_2
14:  return
```

Figure 3: An implicit flow in bytecode

3 Our Approach

3.1 Overview

In our approach, the information flow is separated into two steps: disclosing dependence relationships among variables by transforming the bytecode instructions to a set of definitions; verifying information flow by computing and checking security classes.

In existing type checking method, each variable x occurring in a program is declared with a particular security class. These security classes are assumed to form a lattice, with meet (greatest lower bound) and join (least upper bound). The type checker computes the class of an expression and certifies a program with information flow rule. The advantage of using a type system as the basis of a certification mechanism is that it is simple to implement. However, most certification mechanisms based on types reject any program that contains an insecure subprogram.

Consider these statements:

$$k := h; k := 6; \tag{1}$$

$$h := k; k := h; \tag{2}$$

h denotes a high-security variable and k denotes a low-security variable.

The type system approaches are less precise because it rejects the Secure Programs (1) and (2).

This paper describes a method that analyzes bytecode secure information flow. The analysis process is divided into two phases: variables information dependence disclosing and secure class computing. In the first phase, the bytecode instructions are transformed to a set of definitions of variables which represents the information dependence relation among variables in the program. In the next phase, the security classes are assumed to variables in the definition set and verify the information flow as the traditional type checking method.

For instance, the following two instructions sequence can be transformed to a definition-use: the local variable $r1$ is used and $r2$ is defined.

$$\text{iload\_1; istore\_2;} \Rightarrow r2 \leftarrow r1.$$

Different from the method of type checking, the whole program is analyzed and the information dependence among variables is disclosed before the security verification, thus can improve the analysis precision. The next phase is secure class computing and verification. Because the program is transformed to a sequence of definition-use pair with no branch, the computing and verification is simply performed by the least upper bound operation of security class and certifying secure information flow.

The other significance of our approach is that, the first phase can be prepared beforehand for the fixed program such as API because this phase does not handle the secure classes that corresponding to various host security policies. So the analysis result of the first phase can be prepared and archived to a library for the fixed programs, thus the analysis efficiency can be improved.

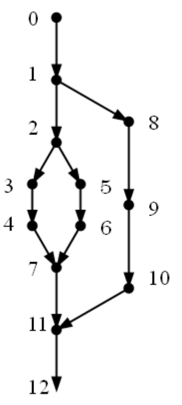


Figure 4: An example bytecode CFG

3.2 Bytecode Control Flow Analysis

Given a bytecode B, the control flow graph (CFG) of the bytecode is an ordered pair (V, A), where V is a finite set vertices; and A is a finite set of the Cartesian product $V \times V$, called arcs, i.e., $A \subseteq V \times V$ is a binary relation on V. For any arc $(v_1, v_2) \in A$, v_1 is called the initial vertex of the arc, and v_2 is called terminal vertex of the arc. Assuming that the control flow graph has one and only one final node, if $i, j \in A$, j post dominates i, denoted by $j\ pd\ i$, if $j \neq i$ and j is on every path from i to the final node.

In the bytecode method, each instruction is a vertex in the CFG. There are three types of JVM instructions that may cause control dependencies.

First, the control transfer instructions conditionally or unconditionally cause the Java virtual machine to continue execution with an instruction other than the one following the control transfer instruction. These instructions can cause control dependencies.

- Unconditional branch instructions: *goto, goto_w, jsr, jsr_w, ret.*
- Conditional branch instructions: *ifeq, iflt, ifle, ifne, ifgt, ifge, ifnull, ifnonnull, if_icmpeq, if_icmpne, if_icmplt, if_icmpgt, if_icmple, if_icmpge, if_acmpeq, if_acmpne.*
- Compound conditional branch instructions: *tableswitch, lookupswitch.*

Second, in JVM, a method must return the control to its caller after execution. The caller is often expecting a value from the method called. JVM provides six return instructions. These return instructions cause another kind of control dependencies.

- Return instructions: *ireturn, lreturn, freturn, dreturn, areturn, areturn.*

Third, another kind of special branch is the jsr. It remembers where it came from. Instruction jsr branches to the location specified by the label, it leaves a special kind of value on the stack called returnAddress to represent the return address. This causes certain control dependence.

By detecting the conditional transfer instructions, the forks of the control flow can be determined. The joins of the control flow can be achieved by tracing the branch path. In our approach, the CFG is represented by:

$$\begin{aligned} \text{CFG} \quad : \quad & (V, A) = \{S_1, S_2, S_3, \ldots S_m\} \\ & S_i \subseteq V, 1 \leq i \leq m \\ & S_i = \{I_0, I_1, \ldots I_n\} \\ & I_k = (i_k, cd_k) \end{aligned}$$

In S_i, for any two nodes i, $j \in V$ and $j > i$, j pd i. In short, the special graph is a chain, which is a subset of CFG. Each element of S_i is a pair (i_k, cd_k), in which the i denotes instruction and cd denotes control dependence information. Initial node I_0 is the start node or a fork while terminal node I_n is the end node or a join of the method's CFG. The scope of conditional branch can be determined by these fork and join nodes.

To deal with implicit flows, cd_k is introduced in to represent the guard (the conditional transfer instruction) that lead to execution of the instruction k. With this information, the implicit information dependence can be determined. In the programs with multiple branch nesting, the *cd* of the node of conditional transfer instruction can be used for the sub branch instructions to disclose the parent control dependence. To distinguish main control flow from the branch path, *cd* of the nodes in main path is set to -1.

Given a CFG in Figure 4, the CFG is presented by:

$$\begin{aligned} \text{CFG} &= \{S1, S2, S3\} \\ S_1 &= \{(0, -1), (1, -1), (2, 1), (3, 2), (4, 2), (7, 1), \\ & \quad (11, -1), (12, -1)\} \\ S_2 &= \{(2, 1), (5, 2), (6, 2), (7, 1)\} \\ S_3 &= \{(1, -1), (8, 1), (9, 1), (10, 1), (11, -1)\} \end{aligned}$$

For example, node 5 depends on node 2 and node 2 depends on node 1, so node 5 depends on both node 2 and 1. These dependence relations will be used to disclose implicit information flow.

3.3 Definition Determination

JVM is a stack-based abstract machine, in JVM most operations occur via stack. Different from memory variable, stack does not keep the value after stored to a variable. A storing operation causes the data flow from source variables to a destination variable, which can be presented by a definition-use pair (DUP), in which the variable loading a value to the stack is used and the variable the value stored from the stack is defined. Data flow in a method is caused by a series of data loading and storing operations. It can be presented by an ordered set of DUP.

$$\frac{B[i] = T\text{push } n \ |T\text{const null} \ \ < U, D, S >}{< U, D, \lambda \cdot S >}$$

$$\frac{B[i] = \mathit{prim\ op} \ \ < U, D, v1 \cdot v2 \cdot S >}{< U, D, (v1 \cup v2) \cdot S >}$$

$$\frac{B[i] = \text{pop} \ \ < U, D, v \cdot S >}{< U, D, S >}$$

$$\frac{B[i] = \text{load } x \ \ < U, D, v1 \cdot v2 \cdot S >}{< U, D, (v1 \cup v2) \cdot S >}$$

$$\frac{B[i] = T\text{store } x \ \ < U, D, v \cdot S >}{< \phi, (x, v \cup U) \cup D, S >}$$

$$\frac{B[i] = T\text{astore } x \ \ < U, D, v1 \cdot v2 \cdot v3 \cdot S >}{< \phi, (v1, v2 \cup v3 \cup U) \cup D, S >}$$

$$\frac{B[i] = \text{if}\mathit{cond}\ j|\mathit{tableswitch}|\mathit{lookupswitch} \ \ < U, D, v \cdot S >}{< \phi, (\mathit{null}, v \cup U) \cup D, S >}$$

$$\frac{B[i] = \text{iinc } x \ \ < U, D, S >}{< U, (x, \lambda) \cup D, S >}$$

$$\frac{B[i] = \text{ldc } x \ \ < U, D, S >}{< U, D, x \cdot S >}$$

$$\frac{B[i] = \text{newarray} \ \ < U, D, v \cdot S >}{< v \cup U, D, \lambda \cdot S >}$$

$$\frac{B[i] = \text{anewarray } x \ \ < U, D, v \cdot S >}{< v \cup U, D, \lambda \cdot S >}$$

$$\frac{B[i] = \text{getfield } C.f \ \ < U, D, v \cdot S >}{< U, D, C.f \cdot S >}$$

$$\frac{B[i] = \text{putfield } C.f \ \ < U, D, v1 \cdot v2 \cdot S >}{< \Phi, (d = C.f, U \cup v2) \cup D, S >}$$

$$\frac{B[i] = \text{invoke } C.mt < U, D, S >}{< U, D, S >< U', D', S' >}$$

$$\frac{B[i] = \text{return} \ \ < U, D, S >< U', D', S' >}{< \Phi, (\mathit{ret}, U \cup U') \cup D \cup D', S >< \Phi, \Phi, \Phi >}$$

Figure 5: Rules of the definition analysis

A DUP is denoted by

$$\begin{aligned} & d \leftarrow U \text{ or } d \leftarrow \{u_0, u_1, \ldots\} \\ & d : \text{defined variable} \\ & U : \text{set of used variables} \end{aligned}$$

In JVM, the data types are divided into two categories: primitive types and reference types. The primitive data flow causes the definition and use information, however, loading or storing a reference value will not necessarily form definition of the data. For example, an operation that loading an object reference data to the stack may not lead the data of the object flow to the stack because this reference may be used to store data to the field of the object or invoke a method of the object. To determine the data flow, the set S and set D is introduced in. Set S contains the variables loaded to the stack while set D contains the DUP of the method. A temporary set U records the variables used.

Definition determination is achieved by transforming

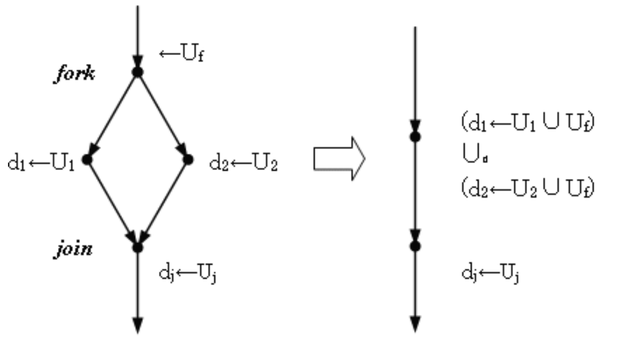


Figure 6: Handling branch in control flow

bytecode instructions to an ordered set of DUPs.

$$B \leftarrow D$$

Where B is the set of bytecode instructions, D is the definition-use set.

Figure 5 defines the rules for data dependence analysis, in which the abstract status of an instruction information flow is presented as a transition relation $i : (S, U, D) \rightarrow (S', U', D')$. i is the instruction, S is a set of variables loaded to the stack, U is a set of used variables and D is a set of DUP. D_i is a DUP corresponding to the instruction i. Constant is denoted by λ. According to these rules, the DUPs of the method can be achieved by analyzing these status transformation in bytecode executing order.

The bytecode is analyzed method-by-method. In a bytecode method, the formal arguments are treated as local variables from 1 to n, where n is the number of argument. When the method is invoked, the variables loaded to the stack are passed to arguments. So the dependence relationship among these arguments is that among the local variables.

3.4 Dependence Analysis

For determining information flow, forks and joins in the control flow of the method need to be cleared. The definition of variables in the conditional branch also depends on used variables in the forks. For an instance, there is a conditional branch with two arms. The set of used variable in the fork is U_f, the DUP in one arm is $d_1 \leftarrow U_1$, in the other arm is $d_2 \leftarrow U_2$. d_1 and d_2 also depend on used variables U_f, so $d_1 \leftarrow U_1 \cup U_f$ and $d_2 \leftarrow U_2 \cup U_f$. Then the graph of this branch is converted to a chain. (see Figure 6).

Branch merging rule:

$$\begin{aligned}
&\text{For } D_i = (d_i \leftarrow U_i), D_j = (d_j \leftarrow U_j) \\
&\quad \{D_i\} \cup_m \{D_j\} := \\
&\qquad \begin{cases} \{d_i \leftarrow U_i \cup U_j\} & \text{if } d_i = d_j \\ \{D_i \cup D_j\} & \text{otherwise} \end{cases}
\end{aligned}$$

After the transformation, the method of bytecode is represented by an ordered set of definition-use pairs. Information flow analysis is simple based on this ordered set.

Information transfer rule:

$$\begin{aligned}
&D_i = (d_i \leftarrow U_i)\ D_j = (d_j \leftarrow U_j)\ i < j \\
&\text{if } d_i \in U_j \text{ and } i < j \\
&\qquad \text{then } U_i \cup D_j := (d_j \leftarrow U_j \cup U_j) \qquad (3)
\end{aligned}$$

The operation with respect to constant

$$U_i \cup \lambda = U_i$$

Definition reaching rule:

$$\begin{aligned}
&\text{Reach}(i) := (\text{ReachIn}(i) \setminus \mathit{kill}(i)) \cup \mathit{gen}(i) \qquad (4) \\
&\mathit{gen}(i) := D(i) \\
&\mathit{kill}(i) := \text{ReachIn}(i) \wedge_d D(i)
\end{aligned}$$

Where Reach(n) denotes the variables definition at node n. "$\wedge_d$" denotes a sub intersection operator.

For two DUPs $D_i = (d_i \leftarrow U_i)$ and $D_j = (d_j \leftarrow U_j)$,

$$\{D_i\} \wedge_d \{D_j\} := \begin{cases} \{D_j\} & \text{if } d_i = d_j \\ \{\} & \text{otherwise} \end{cases}$$

When a method is invoked, if there is no output action occurred inside, then only the DUPs among the arguments, global variables and return of the method are concerned. These definitions of local variables can be removed from the DUP set.

By these rules, all local variables in the DUPs of a method can be cleared. The dependence relationship among the arguments, global variables and return can be disclosed. The Programs (1) and (2) can be certified by our approach. In (1), definition $k := h$ is killed by the later definition $k := 6$ by definition reaching Rule (4). In (2), the program is transformed to $k := k$ in terms of the information transfer Rule (3).

3.5 Security Class Computation and Verification

By the information dependence analysis, the information flow of a method is represented by a sequence of DUPs. In order to verify the security policy, the security class of definition should be computed. The rule of security class computation based on DUP is as following:

- Security class of defined variable is the least upper bound of that of used variable.
  for $d \leftarrow \{u_0, u_1, \ldots, u_n\}$
  the secure class of d will be updated to:

$$S'(d) := S(u_0) \vee S(u1) \vee \ldots \vee S(u_n) \qquad (5)$$

- Constant has lowest security class.

$$S(\lambda) = \perp \qquad (6)$$

Once the security classes of all the variables are achieved, the security verification can be performed by this rule:

$$\begin{array}{l}\text{If } S'(d) \leq S(d) \\ \text{Then } d \leftarrow u_0, u_1, \ldots, u_n \text{ is secure}\end{array} \quad (7)$$

Regarding to the whole bytecode security verification process, the first step is to detect and to mark all input and output operations in the program. Next step is to analyze DUPs of all the methods, then to compute the security classes and verify the security policies. If there is no monitored input/output operation inside a method, only the DUPs among the arguments, global variables and return of the methods are used to compute the security class.

3.6 An Example

Here is a simple example illustrating our approach. Given the bytecode and it's CFG of a method in Figure 7.

Step 1, the CFG is separated to two chains:
S_1= {0,1,2,3,4,5,6,7,8,9,10,11,14,15,16,17,18,19,20,21,22}
S_2= {8,12,13,14}
There is a conditional branch with two arms:
$b1$= {12,13} and $b2$= {9,10,11}
Both depend on the conditional transfer instruction at 8.

Step 2, by the rule of data dependence analysis in Figure 3, the analysis process is shown in Figure 8. $r1$, $r2$ and $r3$ are formal arguments of the method while $r4$ and $r5$ denote local variable in the method.

The following is the DUPs transformed from the method foo.

$$\text{DUPs} = \{D_6, D_8, D_{10}, D_{13}, D_{17}, D_{20}, D_{22}\}.$$

The branch arms $b1$ and $b2$ depend on the variable used at the fork 8, Instruction 10 is in $b2$ and Instruction 13 is in $b1$. Used set in the fork is $r4$. The DUP at 10 and 13 should be updated.

$$D_{10} := D_8 \cup D_{10} = (r5 \leftarrow \lambda \cup r4) \Rightarrow (r5 \leftarrow 4)$$
$$D_{13} := D_8 \cup D13 = (r5 \leftarrow \lambda \cup r4) \Rightarrow (r5 \leftarrow 4)$$
$$\{D_{10}\} \cup_m \{D_{13}\} = r5 \leftarrow r4.$$

Thus, the branch can be merged, and the information dependence pairs of this method can be represented by the following DUP list.

$$\begin{array}{l}\{(r4 \leftarrow r2.f \cup \lambda), \\ (r5 \leftarrow r4), \\ (r5 \leftarrow r5 \cup r1), \\ (r3.f \leftarrow r5), \\ (ret \leftarrow r5)\}.\end{array}$$

According to the Dependence analysis rules, the information dependence relation among arguments and return can be achieved:

$$\{(r3.f \leftarrow r2.f \cup r1), (ret \leftarrow r2.f \cup r1)\}$$

These DUPs show that in this method, information from $p1$ and $p2$ may flow to the field of $p3$ and to the return value.

Step 3, the security class computing.

To verify the information-flow policy, suppose $r1$, $r2.f$ and $r3.f$ have security class 3,2 and 1 respectively.

$$\begin{array}{rcl} S(r1) & = & 3 \\ S(r2.f) & = & 2 \\ S(r3.f) & = & 1 \end{array}$$

So the security class of return can be computed by: $S(ret) = S(r1) \vee S(r2.f) = 3$.

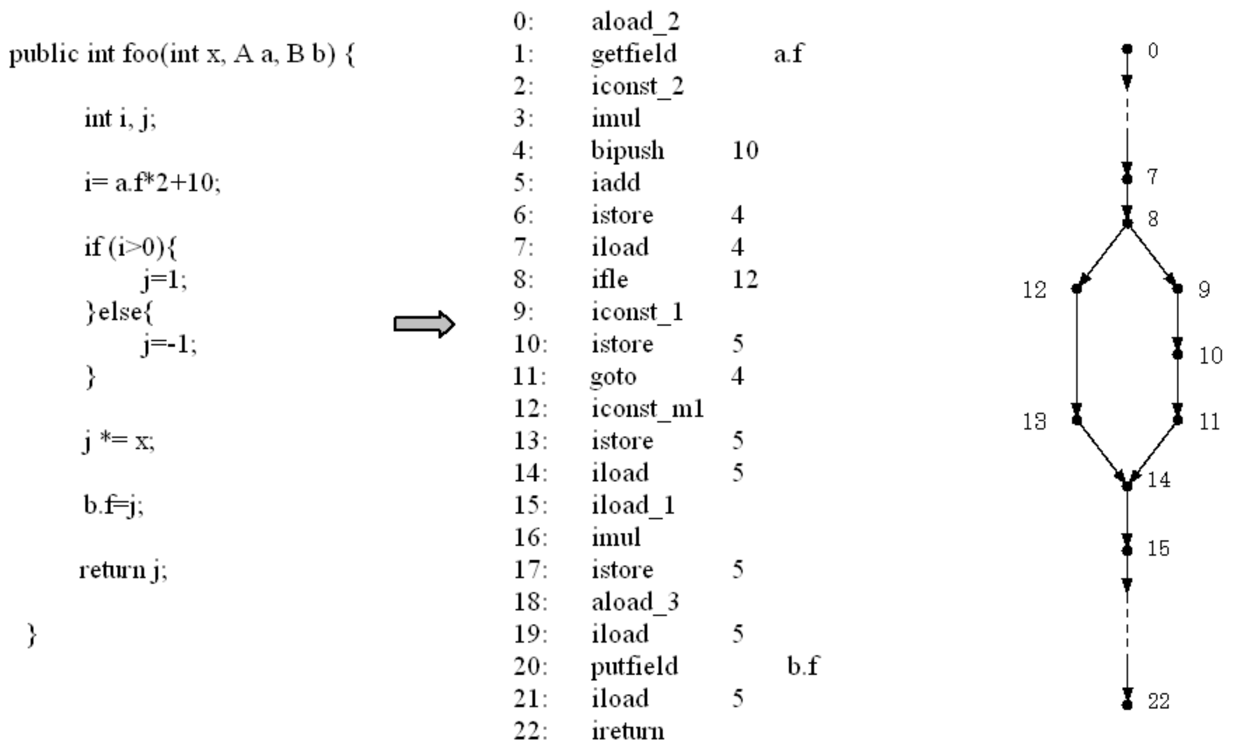
On the other hand, $S'(r3.f) = S(r1) \vee S(r2.f) = 3$ so $S(r1)S'(r3.f) > S(r3.f)$, an information flow policy violation occurs when the data is stored to $r3.f$.

By this approach, a potential information-flow policy violation in this method can be detected automatically. To be more specific, it is at instruction 20.

4 The Tool

A prototype tool is developed based on the method described above, which analyzes the object in method-by-method manner. The tool is written in Java, and is composed of the following components:

- Method Parser: the bytecode engineering library (BCEL) is used to read bytecode information from object or class file. The instructions of the methods are transformed to an ordered set of nodes.
- Control flow Analyzer: the control flow is analyzed by this analyzer. The control dependence is also disclosed in this process. The output of this component is the CFG of the method, which described in Section 3.2.
- Dependence analyzer: the instructions of bytecode are transformed to an ordered set of definition based on the rules in Section 3.3. Then the information flow is analyzed based on the data flow rules in Section 3.4.
- Security Verifier: secure class of input and output objects are assigned to the variables. The secure classes of variables are computed according to the Rules (5) and (6), and the secure verification is performed based on Rule (7).



```
public int foo(int x, A a, B b) {

    int i, j;

    i= a.f*2+10;

    if (i>0){
        j=1;
    }else{
        j=-1;
    }

    j *= x;

    b.f=j;

    return j;

}
```

```
0:   aload_2
1:   getfield        a.f
2:   iconst_2
3:   imul
4:   bipush     10
5:   iadd
6:   istore     4
7:   iload      4
8:   ifle       12
9:   iconst_1
10:  istore     5
11:  goto       4
12:  iconst_m1
13:  istore     5
14:  iload      5
15:  iload_1
16:  imul
17:  istore     5
18:  aload_3
19:  iload      5
20:  putfield        b.f
21:  iload      5
22:  ireturn
```

Figure 7: A method in bytecode and its CFG

The tool is under development, and the current version can be used to disclosing variables dependence of the programs without handling exception and the arguments-return dependences of invoked API methods are prepared in a library. The executing time for analyzing a common bytecode program with 4253 instructions is about 2783ms.

5 Conclusions

This paper described an innovative approach to provide host data confidentiality by analyzing the Java bytecode. This approach monitors the system access actions, analyses information flow inside the Java bytecode and checks if there is any violation that will destroy the host confidentiality. The information dependence analysis and its algorithm are described, by which the Java bytecode security verification can be more flexible, extensible and effective and the impairment on bytecode function can be avoided. Dependence analysis was studied in various programming languages, and was mainly applied to compiler optimizing, program slicing and various software engineering tasks such as program debugging, testing and maintenance in high-level languages. In this paper, it is introduced into Java bytecode information flow analysis.

Different from type systems method which abstractly executes bytecode in security classes, our approach determines the information dependence of variables then computes the security class to certify the program. The secure classes which related to various host security polices are

Inst	Stack	Definition
0	$r2$	
1	$r2.f$	
2	$r2.f, \lambda$	
3	$r2.f \cup \lambda$	
4	$r2.f \cup \lambda$, λ	
5	$r2.f \cup \lambda$	
6		$D_6 = (r4 \leftarrow r2.f \cup \lambda)$
7	$r4$	
8		$D_8 = (\leftarrow r4)$
9	λ	
10		$D_{10} = (r5 \leftarrow \lambda)$
11		
12	λ	
13		$D_{13} = (r5 \leftarrow \lambda)$
14	$r5$	
15	$r5, r1$	
16	$r5 \cup r1$	
17		$D_{17} = (r5 \leftarrow r5 \cup r1)$
18	$r3$	
19	$r3, r5$	
20		$D_{20} = (r3.f \leftarrow r5)$
21	$r5$	
22		$D_{22} = (ret \leftarrow r5)$

Figure 8: Information dependence of method foo

not involved when disclosing information dependence of variables in the bytecode, so the DUPs of the bytecode can be achieved in advance. For the fixed methods such as API, the DUPs can be archived beforehand so as to avoiding reiterative computation while being invoked, thus can save a lot of time on the fly.

A prototype based on these techniques has been developed, however, the exception handling has not been included in current version, and the DUP library for Java API has not been built. These will be the focus area in future research.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[2] M. Avvenuti, C. Bernardeschi, and N. D. Francesco, "Java bytecode verification for secure information flow," *ACM SIGPLAN NOTICES*, vol. 38, no. 12, pp. 20-27, 2003.

[3] A. Banerjee and D. Naumann, "Secure information flow and pointer confinement in a java-like language," in *Proceedings of IEEE Computer security Foundations Workshop*, pp. 253-267, June 2002.

[4] R. Barbuti, C. Bernardeschi, and N. D. Francesco, "Checking security of java bytecode by abstract interpretation," in *The 17th ACM Symposium on Applied Computing: Special Track on Computer Security Proceedings*, pp. 229–236, Madrid, Mar. 2002.

[5] G. Bian, K. Nakayama, Y. Kobayashi, and M. Maekawa, "Mobile code security by java bytecode dependence analysis," in *Proceedings of the International Symposium on Communications and Information Technologies 2004 (ISCIT 2004)*, pp. 923-926, Sapporo, Japan, Oct. 26- 29, 2004.

[6] G. Bian, K. Nakayama, Y. Kobayashi, and M. Maekawa, "Java Mobile Code Security by Bytecode Analysis," *ECTI Transactions on Computer and Information Technology*, vol. 1, no. 1, pp. 30-39, 2005.

[7] C. Bernardeschi, N. D. Francesco, and G. Lettieri, "An abstract semantics tool for secure information flow of stack-based assembly programs," *Microprocessors and Microsystems*, vol. 26, no. 8, pp. 391-398, 2002.

[8] C. Bernardeschi, N. D. Francesco, and G. Lettieri, "Using standard verifier to check secure information flow in java bytecode," in *COMPSAC 2002*, pp. 850-855, 2002.

[9] C. Bernardeschi and N. D. Francesco, "Combining abstract interpretation and model checking for analysing security properties of java bytecode," in *Third International Workshop on Verification, Model Checking and Abstract Interpretation Proceedings*, LNCS 2294, pp. 1-15, Springer-Verlag, Venice, Jan. 2002.

[10] C. Chambers, I. Pechtchanski, V. Sarkar, M. J. Serrano and H. Srinivasan, "Dependence analysis for java," *LCPC 1999*, pp. 35-52, 1999.

[11] D. E. Denning, "A lattice model of secure information flow," *Communications of the ACM*, vol. 19, no. 5, pp. 236-243, 1976.

[12] D. E. Denning and P. J. Denning, "Certification of programs for secure information flow," *Communications of the ACM*, vol. 20, no. 7, pp. 504-513, 1977.

[13] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5-19, Jan. 2003.

[14] L. Tim and F. Yellin, *The Java Virtual Machine Specification*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1996.

[15] B. Venners, *Inside the Java Virtual Machine*, 1998.

[16] D. Volpano, G. Smith, and C. Irvine, "A sound type system for secure flow analysis," *Jornal of Computer Security*, vol. 4, no. 3, pp. 167-187, 1996.

[17] J. Zhao, "Dependence analysis of java byte-code," in *Proceedings of 24th IEEE Annual International Computer Software and Applications Conference (COMP-SAC'2000)*, pp.486-491, IEEE Computer Society Press, Taipei, Taiwan, Oct. 2000.

[18] J. Zhao, "Static analysis of java bytecode," *Proceedings 2001 International Software Engineering Symposium*, pp. 383-390, Wuhan, China, Mar. 2001.

Gaowei Bian received the M.S. degree in techniques and instruments of electromagnetic measurement from Shanghai University in 1997. He worked as a supervisor and system engineer for NEC system integration (China) Co., Ltd. Currently he is a Ph.D student of the Graduate School of Information Systems, University of Electro-Communications, Tokyo, Japan. His research interests include distributed systems and JVM. He is a student member of IEICE.

Ken Nakayama received his B.S. and M.S. degrees from the University of Tokyo in 1987 and in 1990, respectively. He is currently a Research Associate at the Graduate School of Information Systems, University of Electro-Communications, Tokyo, Japan. His research interests include software engineering, multimedia systems, user interface systems, and data analysis systems.

Yoshitake Kobayashi received his B.E. degree from the Shonan Institute of Technology in 1996 and his M.E. and Ph.D. degrees from the University of Electro-Communications in 1999 and 2002, respectively. He is currently a research associate of the Graduate School of Information Systems, University of Electro-Communications, Tokyo, Japan. His research interests include operating systems, distributed systems and dynamically reconfigurable systems.

Mamoru Maekawa received his B.S. and Ph.D degrees from Kyoto University and the University of Minnesota, respectively. He is currently Professor of the Graduate School of Information Systems, University of Electro-Communications, Tokyo, Japan. His research interests include distributed systems, operation systems, software engineering, and GIS. He is listed in many major Who's Who's.