Java Mobile Code Security by Bytecode Analysis

Gaowei Bian, **Ken Nakayama**,
Yoshitake Kobayashi, and **Mamoru Maekawa**, Non-members

ABSTRACT

Since mobile code can migrate from a remote site to a host and can interact with the resources and facilities of the host, security becomes the key to the success of mobile code computation. Existing mobile code security mechanisms such as access control are not able to fully address the import security properties of the host including confidentiality and integrity. And these practices tend to protect the host from potential attacks by confining the mobile code, thus will impair the function of mobile code. Information-flow policy is a technique that can ensure confidentiality, however the analysis of the information flow is practically difficult. This paper describes an innovative approach to provide Java mobile code system security by bytecode analysis. The key technique of the approach is the dependence analysis adapted to information flow analysis. A security model for mobile code system is also proposed in this paper. By this approach, two major properties of the host security – integrity and confidentiality can be protected while the additional restriction on mobile code can be greatly avoided. A prototype has been implemented, which can be applied to analyze Java class file, applet and mobile agent.

Keywords: Mobile code security, Bytecode, Dependence analysis, Information flow

1. INTRODUCTION

With the growth of distributed computers and network systems and the Internet Technologies, the utilization of mobile code system such as applets, mobile agents systems is increasing. Mobile code utilization may raise security problems because it is generated by an untrusted producer and can run at a remote host without user's approval or intervention. Malicious mobile code can attack the local host by destroying data, releasing sensitive information. To avoid these risks, existing security approaches focused on confining the mobile code so as to ensure that it can do no harm to the host but the cost is a large amount of useful applications of mobile code are sacrificed to comply with this security policy. The sand-boxing technique can ensure the security of the host, but it also unduly restricted the function of the mobile code. For example, certain application needs information from a file however it is considered sensitive information of the host thus the privilege to read the file cannot be granted to this application; therefore this application can not get the information because of the restriction of security policy. Although the application does no harm to the host, it still cannot fully perform its task without necessary information. Furthermore, existing security approaches are inadequate and unsatisfactory to guarantee the data confidentiality and integrity. These approaches can only restrict the information release but not propagation. Once information is released from its container the accessing program may improperly transmit the information out in another form.

To address the inadequacies, several new approaches have been developed. Cryptographic codesigning is a token that denotes a security property of the signed code unit. This approach does no restriction on the mobile code. It works by certifying the origin of the mobile code and its integrity, but it cannot address the fundamental security risk inhered in mobile code, which is related to the behavior of mobile code. In another word, this approach can certify the producer but not the behavior of the mobile code, thus the security of the host cannot be fully ensured. The Proof-carrying code (PCC) [17] approach enables safe execution of a code from untrusted source by requiring a producer to furnish a proof regarding the safety of the mobile code. It is applicable to generate proofs for simple properties such as memory safety, but for complex security properties, it is a daunting problem to automatically generate proofs. On the other hand, it is also very difficult for a producer to know all the security policies which the consumers are interested in when generating the proofs. Other approaches, such as code verification [10,12], code transformation [18], cannot fully address these problems either. Information flow verifier focuses on protecting confidentiality by information flow analysis. The current approach is using type systems, by which security properties are verified as ordinary data type checking [21,22,23]. In this approach, the lack of

04PSI20: Manuscript received on December 28, 2004 ; revised on May 11, 2005.

The authors are with the Department of Information Systems Graduate School of Information Systems University of Electro-Communications 1-5-1, Chofugaoka, Chofu-shi, Tokyo, 182-8585, Japan. E-mail:{bian, ken, yoshi, maekawa}@maekawa.is.uec.ac.jp

principal type may result in the loss of precision [24]. The above approaches can protect the host against attacks however they will restrict the mobile code to a certain degree and can hardly enforce confidentiality and integrity of security policies.

Both [12] and [13] focus on handling information flow policies. [12] transforms bytecode and uses JVM verification, while [13] use abstract interpretation to check the information flow inside the bytecode. These approaches abstractly execute the bytecode in secrecy level to verify information flow policy. The performance can be a problem for large systems.

Java, a safe intermediate language, is widely used in mobile code system transmitted in bytecode format. This paper describes a technique that protects host security by verifying information flow via Java bytecode analysis. There are three ingredients in host security: integrity, confidentiality and availability. The protection of availability is achieved by JVM because of the safe characteristic of java language. So the major properties need to be protected are integrity and confidentiality. If a program intends to destroy a host's integrity or to let out sensitive information from a host, it must access the system resources first, like file system, network, or other I/O devices. By this discretion, once the behaviors that export data to system resource are controlled, the violation towards system integrity or confidentiality will be prevented. In our approach, information-flow policy is applied to ensure confidentiality properties. Analyzing information flow inside a program is difficult and has not been fully addressed yet. To address the issue, information dependence analysis is introduced in. Dependence analysis was originally used in programs written in high-level programming languages used for software engineering such as program understanding, compiling optimization. In this paper, this technique is introduced to analyze Java bytecode for enforcing host information-flow policy.

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

2. BACKGROUND

2.1 Mobile Code Security

Mobile code from any arbitrary sources may be executed in an environment that exposes access to system resources, so the host may face the following risks when the code is executed.

1. Denial of services. Execution code may monopolize certain resource like screen, CPU time or threading service, etc. System may be rendered.
2. Corruption. Execution code may modify or erase important data of the system.
3. Leakage. Mobile code may release sensitive information to an outside party.
4. Spoofing. Mobile code may masquerade as another one by faking the UI, thus fooling the users to entrust them to obtain critical resources and data.

Viewing from the perspective of host security, the following properties should be protected.

- Integrity. System resources should be protected from unauthorized modification, erasing, or other means of tampering.
- Confidentiality. Sensitive information should be protected from being released by unauthorized channels.
- Availability. System should be protected from interferences that affect its normal operation and availability of service.

Java is a strong typed intermediate language, and many protections can be achieved by its safe characteristic. The Java Virtual Machine (JVM) bytecode [1,2] representation is specially designed to protect the execution units from interfering with each other and from accessing the JVM's internal state. Firstly, JVM bytecode is strictly typed. Secondly, it does not allow pointer arithmetic. Before dynamically linked into the JVM, bytecode should pass the verification to guarantee that it is type-safe. Therefore, the system's safety (in another word, low level security) can be achieved by JVM bytecode verification mechanism. Thus, in Java mobile code system, the system integrity and confidentiality are the key properties to ensure host security.

2.2 Java Bytecode

Because mobile code is transmitted in bytecode format, understanding the characteristic of Java bytecode is necessary.

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 bytecodes 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 [2].

Figure 1 summarizes the instruction set of the Java virtual machine. 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 [10].

2.3 Information Flow

Data confidentiality can be ensured by enforcing information-flow policy. Secret input data cannot be inferred by an attacker through the attacker's observations of system output, this policy regulates information flow.

*T*load x:	Push the value with type T of the register x onto the operand stack.
*T*store x:	Pop a value with a type T off the stack and store it to local register x.
*T*ipush d:	Push a constant d onto the operand stack
*T*const d:	Push constant d with type T onto the operand stack.
pop:	Pop top operand stack element.
dup:	Duplicate top operand stack element.
Top:	Pop two operands with type T off the operand stack, perform the operation op 2 { add, mult, compare .. }, and push the result onto the stack.
ifcond j:	op a value off the operand stack, and evaluate it against the condition cond = { eq, ge, null, ... }; branch to j if the value satisfies cond.
goto j:	Jump to j.
if*c* j:	Pop a value off the stack, Branch to j if the condition c is true.
getfield Cf:	Pop a reference to an object of class C off the operand stack; fetch the object's field f and put it onto the operand stack.
putfield Cf:	Pop a value v and a reference to an object of class C from the operand stack; set field f of the object to v.
invoke C.mt:	Pop value v and a reference r to an object of class C from the operand stack; invoke method C.mt of the referenced object with actual parameter v.
*T*return:	Pop the T value off the operand stack and return it from the method.
new C:	Create an instance of class C and push a reference to this instance on stack

Fig.1: *JVM Instruction Set*

The information flow model can be defined by

$$FM =< N, P, SC, \oplus, \rightarrow > \quad (1)$$

$N = \{a, b, \ldots\}$ is a set of logical storage objects or information receptacles. Elements of N may be files, or program variables. $P = \{p, q, \ldots\}$ is a set of process. $SC = \{A, B, \ldots\}$ 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 [9].

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 $\leq$, and $\leq$ forms a lattice.

To prevent subjects with low clearance accessing sensitive data, we need the following property:

A subject s may have read access to an object o only if C(o) $\leq$ C(s).

To prevent subjects with high clearance to release sensitive data to low clearance subjects, we need the following property:

A subject s who has read access to an object o may have write access to an object p only if C(o) $\leq$ C(p).

The information-flow theory is highly theoretical, the practical application to a program still needs further study. In our approach, information dependence analysis is applied to analyze information flow in bytecode.

2.4 Dependence analysis

To analyze bytecode information flow, a way that applies the dependence analysis technique traditionally used in software engineering for program analysis is proposed. Program dependencies are relationships among elements in a program, they are determined by the control flows and data flows in the program. Dependence analysis is the process to disclose the program's dependencies by analyzing control flows and data flows in the program. Previous works on dependence analysis mainly focused on programs that written in high-level programming languages [26,27], rather than programs written in low-level programming languages. Several dependence analysis techniques for bytecode have been proposed [5,6,7,8]. Now this technique is introduced to analyze Java bytecode information dependence for protecting host security.

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 $i, j \in V, j$ post dominates i, denoted by j pd i, if j $\neq$ i and j is on every path from i to the final node.

By analyzing data dependence in bytecode, the dependence relationships among variables can be disclosed. Different from high-level programming language, in bytecode instruction set there is no assignment directly from a variable to another. Instead, a value is assigned by loading to the stack and storing to a local variable. According to JVM, once a method is invoked, a fixed sized frame is allocated, which consists of a fixed sized operand stack and a set of local variables. A bytecode instruction that loaded the value of a variable forms a use of the vari-

able while an instruction that stored a value from the stack to a variable forms a definition of the variable.

The explicit information flow can be determined by data dependence analysis. However, it is inadequate to address information flow because information may flow implicitly. Given two variables m and n, m contains some sensitive data and n is globally accessible, in another word, variable m has higher secrecy level than variable n. In the following expressions:

$$\begin{aligned} x &= m; \\ n &= x + 1; \end{aligned}$$

The value of m flows to variable n explicitly. By data dependence analysis, n depends on m, shows the value of defined variable flows to used variable.

However, information may flow implicitly by the following expressions:

$$\begin{aligned} &\text{if } (m > 0) \quad n = 1; \\ &\text{else} \qquad\quad n = 0; \end{aligned}$$

The information in variable n depends on the value of variable m. The value of n can be used to conjecture the value of m.

Being combined with control flow analysis, data dependence analysis is extended and used for information flow analysis. It is called information dependency analysis in this paper.

3. OUR APPROACH

3.1 Security Model and Verification Mechanism

In the security model, the mobile code system is considered being composed of two parts, the mobile code process and the system resources. The mobile code process is considered as a subject, the system resources including file systems, network resource, IO devices and others are a set of objects. Regarding the operations that the mobile code process interacts with the system resources, there are two types of actions: input and output. Input action means the process obtains data from system resources, such as reading data from file or network. Output action means the subject's activities change the status of the object, such as writing data to a file or doing some operations on a device. The output action causes the status change of an object. Since only the output action can finally modify the host's data or release sensitive information to a third party, the proposal in our approach is to monitor these output actions.

Figure 2 presents the model of our approach. Mobile code execution in a host run time environment is a process, which may interact with the system resources. When it gets data from the system, the input action will occur and the information will flow from system resource to the process; when it puts data into the system, the output action will occur and the information will flow from the process to system resource. It is the output action that causes security violation. If these actions are under control, the system security properties can be protected. Because the input actions are not restricted, the mobile code impairment caused by conventional access control can be avoided.

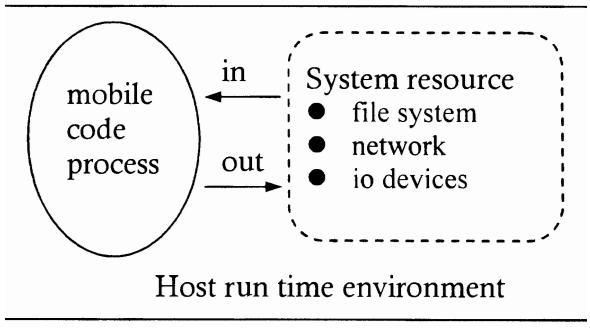

Fig.2: *Security Model.*

The types of system resources (system storage objects) to which access is required include:

- file system
- network
- output devices (entire display, various windows, speaker ...)
- input devices (keyboard, microphone, ...)

To protect host integrity, a privilege level is granted to the process and integrity levels are assigned to the objects. When an output action occurs, the integrity level will be checked to ensure that this behavior does not violate the host's integrity policy. The rule of this policy is that the privilege level of the process cannot be lower than the integrity level of the object.

To protect host confidentiality, the secrecy levels are assigned to the objects. When an output action occurs, the secrecy level of the data that the action depends on will be checked. To perform this verification, the secrecy level of the data is needed. After the information dependence analysis the secrecy level of every variable in the bytecode can be computed. If the data's secrecy level is higher than the object that the action affects, this operation is considered illegal according to the information flow policy.

We propose a new abstract model to represent this protecting mechanism as following:

$$\frac{O_p V}{Q \rightarrow \acute{Q}} \tag{2}$$

In this formula, O_p denotes an output action, while V denotes the data related to the output action, Q and $\acute{Q}$ are the status of the object before and after the output action occurs. Output action depends on related data. For example, mobile code process writes some data to a file, the data are related data. Or the process tries to delete a file in some condition, the

```
0:   aload_0
1:   aload_1
2:   getfield    B.f
3:   ifge        6
4:   iconst_0
5:   goto        7
6:   iconst_1
7:   putfield    A.f
8:   return
```

Fig.3: *An implicit flow in bytecode.*

data used for conditional test is related data. These related data are denoted by "V" in our model. When an output action occurs on an object, the status of this object will be modified. Our approach is to check if this modification complies with the host's integrity and confidentiality policy.

The policy for the mechanism is that the privilege of O_p should not be lower than the integrity level of the Q and the secrecy level of V should not be higher than that of the object Q.

$$P(O_p) \leq P(Q) \quad (3)$$

$$S(V) \leq S(Q) \quad (4)$$

The privilege of O_p, the integrity level and the secrecy level of object Q are defined beforehand. When the O_p occurs, the value of these levels can be used for verification. The secrecy level of V [S (V)] can be computed with the secrecy levels of the input objects related to the data V. The relationship between the input objects and the data V can be disclosed by information dependence analysis.

3.2 Information Dependence Analysis

Information flow in the bytecode of a method can be either explicit or implicit. When an assignment is executed, the information flows explicitly. When a conditional branch occurs, the values defined inside the scope of the branch depend on the data tested by the conditional transfer instruction and the information flows implicitly.

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 2, which is tested by the branch instruction 3 and has not been assigned to another variable. The final value of A.f, 0 or 1, depends on this value. Even without storing instruction, the information in B.f still flows to A.f implicitly.

When analyzing information flow, the implicit flow should also be disclosed. A data flow shows the flow of information from its source to its destination, and a control flow represents the executing flow of a program. Therefore, data flow can be used to disclose explicit information flow, while control flow may be applied to determine implicit information flow.

Control dependencies represent bytecode instructions related to control conditions on which the execution of an instruction depends. There are four 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 cause certain control dependence.

Fourth, instructions that may explicitly or implicitly throw exceptions can also cause control dependencies because they can either explicitly or implicitly change the control flow from one instruction to another. These instructions lead to another source of control dependencies.

- Explicit throwing exception instruction: $athrow$
- Implicit throwing exception instructions: $aaload, aastore, anewarray, arraylength, baload, bastore, caload, castore, checkcast, daload, dastore, faload, fastore, getfield, getstatic, iaload, iastore, idiv, instanceof, invokeinterface, invokespecial, invokestatic, invokevirtual, irem, laload, lastore, ldc, ldc\_w, ldc2\_w, ldiv, lrem, monitorenter, monitorexit, multianewarray, new, newarray, putfield, putstatic, saload, sastore$.

After analyzing control dependencies, the CFG of the method can be built. Explicit information flow can be determined via data dependencies analysis.

Data dependencies represent the data flows among the instructions in the bytecode method. Informally a variable u is directly data-dependent on another variable v, if the value of v has a direct influence on the value of u. Data dependencies can be computed by determining the definition and use information, i.e., the set D and U.

First, the definition information, i.e., the set D, of each instruction in a bytecode method can be determined as following:

• A bytecode instruction that stores a value to a local variable forms a definition of that variable. Therefore, the instructions $istore\_<n>, istore, iinc, fstore\_<n>, fstore, astore\_<n>, astore, dstore\_<n>, dstore, lstore\_<n>$, and $lstore$ form the definitions of the local variable.

Second, the use information, i.e., the set U, of each instruction in a bytecode method can be determined as following:

• A bytecode instruction that loads the value of a local variable forms a use of that variable. Therefore, the instructions $iload\_<n>, iload, iinc, fload\_<n>, fload, aload\_<n>, aload, dload\_<n>, dload, lload\_<n>$, and $lload$ forms uses of a single local variable.

Besides data dependence determination, control dependence must be taken into account. The conditional branch depends on the value used for test. In the conditional branch, the variables loaded for the conditional transfer instruction form the use information and the variables stored within the scope of the conditional branch depend on these used variables.

3.3 Analysis Algorithm

To verify bytecode information flow, three steps are taken in our approach: disclosing the dependence relationship among variables in every bytecode method, computing the secrecy level of variables, and verifying security policies by checking the rules (3) and (4). The key technique in this approach is information dependence analysis which is achieved by control flow analysis and data dependence analysis.

3.3.1 Control Flow Analysis

Branches introduce forks and joins in the control flow of a method. By detecting the conditional transfer instructions introduced in 3.2, the forks of control flow can be determined. The joins of the control flow can be achieved by tracing the branch path. In our approach, the CFG of a method is presented by a set of special directed graphs (V, A), in which V is a list of nodes, one for each instruction, and A is a set of the Cartesian product V×V. For any two nodes i, j∈V and j>i, j pd i. In short, the special graph is a chain, which is a subset of CFG. Each of these special graphs can be denoted by:

$S = \{i_0, i_1, \ldots i_n\}$ i : bytecode instruction.

Initial node i_0 is the start node or a fork of the method's CFG. Terminal node in is the end node or a join of the method's CFG. In the CFG, the scope of conditional branch scope can be determined by these forks and joins. The instructions inside any conditional branch are marked with the fork instruction to disclose the control dependence relationship.

For the method in figure 3, CFG is presented by

$$\begin{aligned} CFG &= \{S1, S2\} \\ S1 &= \{0, 1, 2, 3, 4, 5, 7, 8\} \\ S2 &= \{3, 6, 7\} \end{aligned}$$

Instruction 3 is a fork and 7 is a join. There is a two arms branch in this CFG.

$$\begin{aligned} b1 &= \{3, 4, 5, 7\} \\ b2 &= \{3, 6, 7\} \end{aligned}$$

The control dependence of these arms is presented by:

$$\begin{aligned} cd1 &= \{, 3, 3, \} \\ cd2 &= \{, 3, \} \end{aligned}$$

3.3.2 Data Dependencies Analysis

JVM is a stack-based abstract machine, in JVM most operations occur via stack. Different from variable, stack does not keep the value after stored it to the variable. Once a storing operation occurs, the stack is cleared. A storing operation causes the data flow from source variables to a destination variable, which can be presented by a definition-use pair (DUP). As it was discussed in 3.2, an instruction that stores a value to a variable forms a definition of that variable and a bytecode instruction that loads the value of a variable forms a use of the variable. Data flow in a method is caused by a series of data loading and storing operations. So it can be presented by a sequence of DUPs.

A DUP is denoted by

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

To disclose the data flow in a method, DUP caused by instructions should be determined. Figure 4 defines the rules for data dependence analysis, in which the abstract status of an instruction information flow is presented as a transition relation $i : (S, U, D_i) \rightarrow (S', U', Di')$. i is the instruction, S is a set of variables loaded to the stack, U is a set of used variables and D_i is a DUP corresponding to the instruction i. Constant is denoted by τ, and $\Phi = \{\}$. var(n) denotes a variable assigned by the bytecode instruction index n. $\downarrow$ denotes down-set of set. Umt represents the set of used variables formed in the method mt. According to these rules, the DUPs of the method can be achieved by analyzing these status transformations in bytecode executing order. The DUPs form an ordered set in a method, which is the base for computing secrecy level of the variables in the bytecode.

aastore : (S, U, Φ)→(↓ S, Φ, {(v∈S)←U})
aconst_null : (S, U, Φ) → (S, U∪τ, Φ)
aload n: (S, U, Φ)→(S∪var(n), U, Φ)
arraylength: (S, U, Φ)→(↓ S, U∪τ, Φ)
astore n: (S, U, Φ)→(↓ S, Φ, {var(n) ←U ∪ S })
*T*aload: (S, U, Φ)→(↓ S, U∪ ↓ S, Φ)
*T*astore: (S, U, Φ)→(↓ S, Φ, { (v∈S) ←U })
*T*ipush d: (S, U, Φ) → (S, U∪τ, Φ)
*T*const n: (S, U, Φ) → (S, U∪τ, Φ)
*T*load n: (S, U, Φ)→(S, U∪var(n), Φ)
*T*store n: (S, U, Φ)→(↓ S, Φ, {var(n) ←U})
if*cond*: (S, U, Φ)→(Φ, Φ, {var(n) ←U})
iinc: (S, U, Φ)→(S, U∪τ, {var(n) ←τ})
invoke *m*: (S, U, Φ)→(↓ S, U∪ ↓ S ∪U*mt*, Φ)
new n: (S, U, Φ)→(S∪var(n), U, Φ)
getfield C.f: (S, U, Φ)→(↓ S, ↓ S∪var(C.f), Φ)
getstatic C.f: (S, U, Φ)→(↓ S, ↓ S∪var(C.f), Φ)
putfield C.f: (S, U, Φ)→(↓ S, Φ, {var(C.f) ←U})
*T*return: (S, U, Φ)→(Φ, Φ, {var(ret) ←U})

Fig.4: *Rules of the data dependence analysis.*

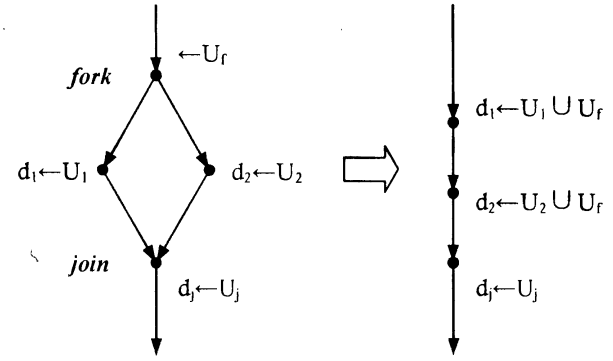

Fig.5: *Handling branch in control flow.*

3.3.3 Dealing with Branch and Predigesting DUPs

For determining information flow, forks and joins in the control flow of the method should be cleared. The variable defined inside the scope of the conditional branch depends on used variables in the fork. For an instance, there is a conditional branch with two arms. The set of used variable in fork is U_f, the DUP in one arm is $d_1 \leftarrow U_1$, and in the other arm is $d_2 \leftarrow U_2$. d1 and d2 also depend on used variables U_f, so $d_1 \leftarrow U_1 \cup U_f$ and $d_2 \leftarrow U_2 \cup U_f$. And then the graph of this branch is converted to a chain. (see figure 5).

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 is concerned. These dependence relationships can be achieved by predigesting the DUPs inside the method.

Suppose there are two DUPs as following:

$$\begin{aligned} d_i &\leftarrow U_i \\ d_j &\leftarrow U_j \\ i &< j \end{aligned}$$

The rules for DUP predigesting are:

- if $d_i \leftarrow U_j$, then $d_j \leftarrow U_j \cup U_j$
- if $d_i = d_j$ then $d_j \leftarrow U_j \cup U_j$
- $U_i \cup \Box = U_i$

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.

3.4 Secrecy Level 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 secrecy level of output data should be computed. The rule of secrecy level computation based on DUP is as following:

- Secrecy level of defined variable is the least upper bound of that of used variable. If $d \leftarrow \{u_0, u_1, \ldots, u_n\}$, Then $S(d)=S(u_0) \vee S(u_1) \vee \ldots \vee S(u_n)$ '$\vee$'denotes the least upper bound operator
- Constant has lowest secrecy level. $S(\tau)=\perp$

Once the secrecy levels of all the variables are achieved, the security verification can be performed by rule (3) and (4).

Regarding the whole mobile code security verification process, the first step is to detect and to mark all input and output operations in the program. And next step is to analyze DUPs of all the methods, then to compute the secrecy levels 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 secrecy level.

In our approach, the secrecy level of variables is not involved when analyzing information dependence. This is valuable for the fixed methods such as java API, because the DUPs of these methods can be analyzed and archived to library beforehand. When the method is invoked, the analysis result can be used to compute secrecy level, thus to save a lot of analyzing time on the fly.

3.5 An Example

Here is a simple example illustrating our approach. Given the bytecode and it's CFG of a method in Figure 6. Step 1, the CFG is separated to two chains:

$$\begin{aligned} S_1 &= \{0,1,2,3,4,5,6,7,8,9,10,11,14,15,16,17, \\ &\quad 18,19,20,21,22\} \\ S_2 &= \{8,12,13,14\} \end{aligned}$$

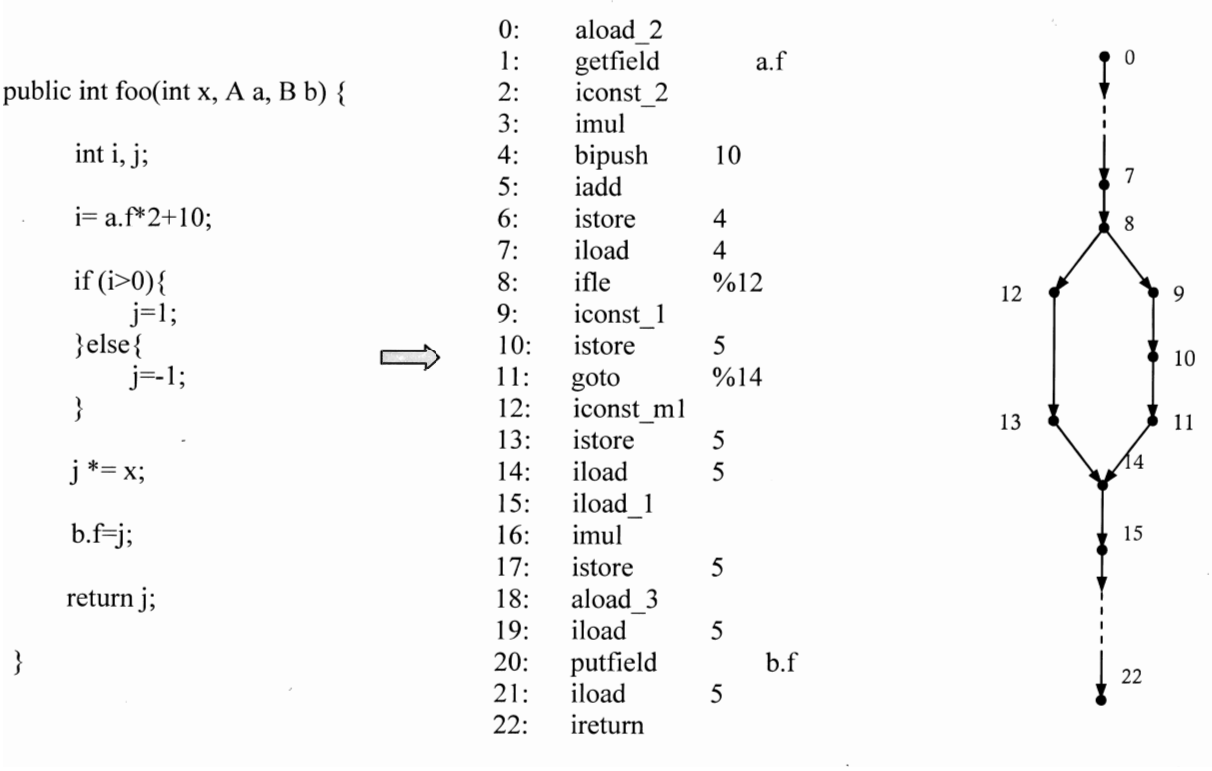

Fig.6: *A Method in Bytecode and it's CFG.*

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 result is shown in figure 7. p denotes argument of the method, while r denotes local variable in the method.

The following is the DUPs formed in the method.

$$\begin{aligned} 6 &: r4 \leftarrow p2 \\ 8 &: \leftarrow r4 \\ 10 &: r5 \leftarrow \tau \\ 13 &: r5 \leftarrow \tau \\ 17 &: r5 \leftarrow r5 \cup p1 \\ 20 &: p3 \leftarrow r5 \\ 22 &: ret \leftarrow r5 \end{aligned}$$

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 must be updated.

$$\begin{aligned} 10 : r5 \leftarrow \tau \cup r4 \quad &\Rightarrow \quad r5 \leftarrow r4 \\ 13 : r5 \leftarrow \tau \cup r4 \quad &\Rightarrow \quad r5 \leftarrow r4 \end{aligned}$$

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

$$\begin{aligned} 6 &: r4 \leftarrow p2 \\ 10 &: r5 \leftarrow r4 \\ 13 &: r5 \leftarrow r4 \\ 17 &: r5 \leftarrow r5 \cup p1 \\ 20 &: p3 \leftarrow r5 \\ 22 &: ret \leftarrow r5 \end{aligned}$$

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

$$\begin{aligned} p3 &\leftarrow p1 \cup p2 \\ ret &\leftarrow p1 \cup p2 \end{aligned}$$

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

Step 3, the secrecy level computing.

To verify the information-flow policy, suppose p1, p2 and p3 have secrecy level 3,2 and 1 respectively.

$$\begin{aligned} S(p1) &= 3 \\ S(p2) &= 2 \\ S(p3) &= 1 \end{aligned}$$

So the secrecy level of return can be computed by: S(ret)= S(p1) ∨S(p2)=2

On the other hand, because p3←p1∪p2 and S(p1) ∨S(p2)< S(p3), an information flow policy violation occurs when the data is stored to p3.

	Stack	Used Set	DUP
0	p2		
1		p2	
2		τ ∪ p2	
4		τ ∪ p2	
6			r4←τ ∪ p2=p2
7		r4	
8			←r4
9		τ	
10			r5←τ
12		τ	
13			r5←τ
14		r5	
15		r5 ∪ p1	
17			r5←r5 ∪ p1
18	p3		
19		r5	
20			p3←r5
21		r5	
22			ret←r5

Fig.7: *Information Dependence of Method foo.*

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. CONCLUSION

This paper described an innovative approach to provide protect host security against attack of malicious mobile code by analyzing the Java bytecode. This approach monitors the system access actions, analyses information flow inside the mobile code and checks if there is any violation that will destroy the host integrity and confidentiality. A security model for mobile code system is also proposed and a technique named information dependence analysis and its algorithm is described, by which the mobile code security verification can be more flexible, extensible and effective and the impairment on mobile code 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 creatively introduced into Java bytecode information flow study.

Different from Abstract Interpreter which abstractly executes bytecode in secrecy level, our approach determine the DUPs and then compute the secrecy level. A method in bytecode the dependence relationship among the variables does not change, so the DUPs can be determined in advance. For the fixed program such as API, the DUPs can be achieved beforehand so as to avoiding repeating computing while being invoked, Thus can save a lot of time on the fly.

A prototype based on these techniques has been implemented, 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] K. Arnold and J. Gosling. *The Java Programming Language*, Addison-Wesley, 1996.

[2] L. T. and F. Yellin. *The Java virtual machine specification.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1996.

[3] Bill Venners. *Inside the Java Virtual Machine.*1998.

[4] Philip W. L. Fong. *Proof Linking: Modular Verification Architecture for Mobile Code Systems.* PhD Dissertation, School of Computing Science, Simon Fraser University, Burnaby, BC, Canada V5A 1S6, January 2004.

[5] Craig Chambers, Igor Pechtchanski, Vivek Sarkar, Mauricio J. Serrano, Harini Srinivasan: *Dependence Analysis for Java.* LCPC 1999: 35-52

[6] J. Zhao, "Analyzing Control Flow in Java Bytecode," *Proc. 16th Conference of Japan Society for Software Science and Technology*, pp.313-316, Japan, September 1999.

[7] J. Zhao, "Dependence Analysis of Java Bytecode," *Proc. 24th IEEE Annual International Computer Software and Applications Conference (COMP-SAC'2000)*, pp.486-491, IEEE Computer Society Press, Taipei, Taiwan, October 2000.

[8] J. Zhao, "Static Analysis of Java Bytecode," *Proc.2001 International Software Engineering Symposium*,pp.383-390, Wuhan, China, March 2001.

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

[10] 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.* Madrid, March 2002.

[11] C. Bernardeschi, N. D. Francesco, and G. Lettieri. "An abstract semantics tool for secure information flow of stack-based assembly programs". *Microprocessors and Microsystems*, 26(8):391-398, 2002.

[12] Cinzia Bernardeschi, Nicoletta De Francesco, Giuseppe Lettieri: "Using Standard Verifier to Check Secure Information Flow in Java Bytecode". *COMPSAC 2002*: 850-855

[13] 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, pages 1-15. LNCS 2294, Venice, January 2002.

[14] Xavier Leroy: *Java Bytecode Verification: Algorithms and Formalizations*. J. Autom. Reasoning 30(3-4): 235-269 (2003)

[15] R. Sekar, V. N. Venkatakrishnan, Samik Basu, Sandeep Bhatkar, Daniel C. DuVarney: "Model-carrying code: a practical approach for safe execution of untrusted applications". *SOSP 2003*: 15-28

[16] V. N. Venkatakrishnan and R. Sekar. "Empowering Mobile Code Using Expressive Security Policies". *10th New Security Paradigms Workshop (NSPW)* 2002.

[17] G Necula. *Proof-carring code*. In ACM Symposium on Principles of Programming Languagues(POPL), 1997.

[18] A. Chander, J. C. Mitchell, and I. Shin. "Mobile code security by Java bytecode instrumentation". *In 2001. DARPA Information Survivability Conference and Exposition* (DISCEX II'01)Volume II-Volume 2.

[19] J. Palsberg and P. Ørbæk, "Trust in the λ-calculus," *in Proc. Symposiumon Static Analysis*. Sept. 1995, number 983 in LNCS, pp. 314-329, Springer-Verlag.

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

[21] A. C. Myers and B. Liskov, "A decentralized model for information flow control," *in Proc. ACM Symp. on Operating System Principles*, Oct. 1997, pp. 129-142.

[22] N. Heintze and J. G. Riecke, "The SLam calculus: programming with secrecy and integrity," *in Proc. ACM Symp. on Principles of Programming Languages*, Jan. 1998, pp. 365-377.

[23] G. Smith and D. Volpano, "Secure information flow in a multithreaded imperative language," *in Proc. ACM Symp. on Principles of Programming Languages*, Jan. 1998, pp. 355-364.

[24] C. Bodei, P. Degano, H. Riis Nielson, and F. Nielson, "Security analysis using flow logics," in *Current Trends in Theoretical Computer Science*, G. Paun, G. Rozenberg, and A. Salomaa, Eds., pp. 525-542. World Scientific, 2000.

[25] A. Sabelfeld and A. C. Myers. "Language-based information-flow security". *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[26] Shigeta Kuninobu, Yoshiaki Takata, Hiroyuki Seki and Katuro Inoue: "An Efficient Information Flow Analysis of Recursive Programs Based on a Lattice Model of Security Classes", *Proceedings of Third International Conference on Information and Communications Security* (ICICS 2001), Xian, China, Nov. 2001, Lecture Notes in Computer Science 2229, pp.292-303.

[27] Shigeta Kuninobu, Yoshiaki Takata, Hiroyuki Seki and Katuro Inoue: "An Information Flow Analysis of Recursive Programs Based on Lattice Model of Security Classes", *IEICE Transactions (D-I)*, J85-D-I(10), 961-973, Oct. 2002.

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.