Java Mobile Code Dynamic Verification by Bytecode Modification for Host Confidentiality

Dan Lu, Yoshitake Kobayashi, Ken Nakayama, and Mamoru Maekawa
(Corresponding author: Dan Lu)

Graduate School of Information Systems, University of Electro-Communications
1-5-1, Chofugaoka, Chofu-shi, Tokyo, 182-8585, Japan (Email: ludan@maekawa.is.uec.ac.jp)
(Received Aug. 2, 2006; revised Oct. 6, 2006; and accepted June 12, 2007)

Abstract

In this paper we present a novel dynamic verification approach to protect the local host confidentiality from malicious Java mobile code. In our approach we use Bytecode Modification to add the verification function to the Java mobile code's class files before the local JVM executes them. Thus the verification work is done when the host JVM executes the modified class files. By this way our approach could achieve higher verification precision because the verification is done in runtime. Furthermore our approach can deal with the information flow in exception handling, which makes our approach more practicable.

Keywords: Bytecode modification, dynamic verification, host confidentiality, mobile code

1 Introduction

Mobile code computation brings great opportunities as well as the problems of security. A malicious mobile program could try to observe, leak or alter the information it is not authorized on a host. It has been a general consensus that security is the key to the success of mobile code computation.

In this paper we try to address the Java mobile code's security problems concerning the host confidentiality, which means that sensitive information should be protected from leaking through unauthorized channels. Considering all mobile code as malicious and rejecting them can give the host the maximum security. But it is the most useless method. Our objective is to verify the Java mobile code precisely as much as possible, that is, to let the mobile code that will not cause any data leaking to pass our verification as many as possible.

Some protection techniques such like Authentication, Access Control and Secure Information Flow have been used to prevent host data from leaking to unauthorized hosts in mobile code systems. Compared to simply informal endorsement such as Authentication and Access Control, the Secure Information Flow, a kind of program-analytic mechanisms, is more precise. Though the approaches in [1, 2, 3] are based on the Secure Information Flow theory, they neglect the distinctiveness of security demand in mobile code systems and assign security-levels (denoting the clearance) to information carriers (objects, method's parameters and return value, etc.) in the mobile code. These works set unnecessary restrictions in the verification procedure and result in impairment to verification precision. In [4, 11] the authors presented security models fitted for the mobile code system and achieved better verification precision than those in [1, 2, 3]. But all those approaches are static approaches. They cannot achieve satisfying verification precision in implicit information transferring because of the inherent limitation of static verification approaches, which is that it is impossible for them to judge which branch of the implicit information transferring will be executed in runtime. Furthermore the static approaches cannot trace the information flow in the exception handling because exceptions are thrown dynamically during the execution, which makes the static approaches lose the practicality.

To improve the verification precision further and resolve the security verification problem of exception handling, we meliorate the security model in [11] and bring forward a dynamic verification approach in this paper. By bytecode modification, we add the verification function to the downloaded mobile code before the code is submitted to the local JVM. Then during the execution of the modified mobile code, the JVM will execute the verification function as well as the original functions of the mobile code. Since the verification is done in runtime, we can get better verification precision than [4, 11] and resolve the verification problem of exception handling.

The rest of this paper is organized as follows. At first we define the security model in Section 2. Then we introduce the bytecode modification method in Section 3 and analyze the information transferring in exception handling of Java bytecode in Section 4. At last we draw a conclusion in Section 5.

2 Security Model

In [6, 7], Dorothy Denning laid the foundation of the Secure Information Flow theory. In the theory, 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 its 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. A subject may only read objects with classification level no higher than its clearance, but may only write to objects with classification level not lower than its clearance.

In mobile code systems, the objects are the files containing sensitive information on the host we need to protect (we refer to it as the local-host) and the subjects are all the other hosts trying to get information from the local host (we refer to them as observer-hosts). Therefore security classes denoting classification levels should be assigned to the files on the local-host, and security classes denoting the clearance should be assigned to observer-hosts. As for the mobile code, it is just the intermediate between objects and subjects. It is not necessary to assign any security class to the mobile code or its information carriers.

Before the mobile code leave the local-host, the information being transferred in the mobile code is not leaked yet. It is when the mobile code tries to send information to any observer-hosts that data-leaking may be caused. Therefore it is not necessary to set any restriction or do any checking when information is being transferred in mobile code. Instead, what we need to do is only tracing and recording the information flow in the mobile code. By this way when the mobile code tries to send information to an observer-host, we can understand the information's classification level and check whether the observer-host has the right to get the information.

The works in [2, 3, 11] assign security levels to the information carriers in the mobile code, which denote both the clearance of carriers and the classification levels of the information in carriers. This mistake leads to a second mistake that they detect data-leaking when the information is still being transferred in the mobile code. These two mistakes result in unnecessary restrictions in verification procedure and reduce the verification precision.

Based on the analysis above, we give the definition of basic conceptions and the security model as follows.

1) **Security-level.** In our approach, we refer to the security class denoting classification level as security-level. The security-level indicates the host files' sensitivity. The higher the security-level is, the more sensitive the file is. All the information stored in a file get the file's security-level.

2) **Clearance-level.** In our approach, we refer to the security class denoting clearance as clearance-level. The clearance-level indicates the trust level of an observer-host to receive the information on the local-host. The higher the clearance-level is, the more trustful the observer-host is.

3) **Data-leaking.** In our approach the data-leaking is defined as that the mobile code sends sensitive information to an unauthorized observer-host, that is, the security-level of the information is higher than the clearance-level of the observer-host.

4) **Data-leaking channel.** We define the way by which the mobile code may cause data-leaking directly or indirectly as a data-leaking channel. The action of detecting data-leaking should be done at every data leaking channel in the mobile code. In mobile code systems, data-leaking channels have three types: 1) the mobile code requests an Internet link, 2) the mobile code moves to next destination and 3) the mobile code writes information to a host file. The former two are direct channels for the malicious mobile code to leak sensitive information. We should compare the security level of the information to be sent with the clearance-level of the observer-host to receive the information. The latter one is an indirect channel. If the information's security-level is higher than the file's security-level, such operation will reduce the information's security-level and may cause a data-leaking later. Therefore we should compare the security-levels of the information and the file.

Definition 1. *Let DLC be a data-leaking channel in one mobile code. Let I be the information to be sent at theDLC and D be the information destination (the observer-host or the file on the local-host) at the DLC. Denoting by L_I the security-level of I and by L_D the clearance-level or security-level of D, a DLC is secure if and only if the following property holds:*

$$L_I \leq L_D.$$

Definition 2. *Let MC is a mobile code. MC is secure if and only if each DLC in the MC is secure.*

3 Bytecode Modification Method

This section outlines the bytecode modification method implementing our security model dynamically. We first introduce the general idea of the modification method. Then we describe the tow main parts of the modification in detail. In this paper, the bytecode with the multi-dimension array is not considered. We will extend our research to cover it in the future work.

3.1 Overview

Why Dynamic Verification. By now the works about mobile code security are almost static verification approaches. The static approach verifies the mobile code before the local JVM executes the code and it will not

slow the execution. But the static approach could not get any runtime information (such as which branch of a conditional structure will be executed, where an exception will happen, and so on). This serious limitation affects the verification precision badly and may make the static approach to loose practicality. To prevent the data-leaking, the static approach has to verify each branch of one implicit transferring respectively, and use the **Least Upper Bounder** (LUB) of all results as the final result. Such operation has a big probability to verify the mobile code as malicious because of the braches that are not executed in runtime. Therefore the verification precision of static approaches is not satisfiable. Considering our object, we implement the security model by dynamic approach to get better verification precision. And we use some skills to reduce the additional overhead in runtime caused by the dynamic verification.

Why Bytecode Modification. To achieve dynamic verification, we select bytecode modification to implement our security model. Although the customable JVM could achieve the same purpose as the bytecode modification, the customable JVM looses the portability, which is one of the most important advantages of Java. To preserve Java's promise "Write Once, Run Anywhere", we use bytecode modification rather than customable JVM to achieve dynamic verification. And Java has tow properties that assist the bytecode modification. Transportable Java code arrives from the network as class files: these class files retain a great deal of symbolic information, allowing the receiver to determine the structure of the class and to modify it on-the-fly. Methods are represented as JVM bytecode: since JVM bytecode are stack instructions, it is relatively easy to splice new code into existing methods.

Load-time Modification. In the program development life cycle, we choose the load-time to apply the bytecode modification. All the mobile code downloaded to the local host are loaded to the JVM by one class loader. Load-time modification is precisely late enough that the modification cannot burden other users, and yet early enough that the JVM is unaware that any modification has taken place, and the modified class is still verified by the JVM before it is accepted. Furthermore, Java is an ideal environment for load-time modification because the JVM uses a user-extensible class loader to locate and load new classes on demand. The class loader could be used to apply the bytecode modification to every class file brought into the local environment.

Modification Content. To apply our security model, we should trace the information transferring and detect data-leaking in mobile code. We add new data containers to store the security-levels of the information held by the information-carriers in mobile code. We insert additional instructions to calculate the change of the security-levels caused by information transferring, both the transferring among information-carriers in one method and the transferring between methods. At each data-leaking channel we insert instructions to check whether it is a secure channel. So the bytecode modification in our approach is composed of tow main parts, *class redefinition* and *instruction insertion*.

3.2 Class Redefinition

Security-Level Containers Adding. As mentioned above, we should add new data containers to store security-levels of the information held by information-carriers in mobile code at first. We refer to the added containers as security-level containers. The JVM is a stack machine manipulating an operand stack and a set of local registers for each method and a heap containing object instances. The elementary information-carrier in JVM bytecode could be the element of operand stack, the local register or the field of an object instance. Thus we need to add security-level containers for the information held by these elementary carriers respectively.

For the local register, we allocate an additional register as the security-level container of the information in the original register. In the attributes table of the $method\_info$ structure, the *Code* attribute defines the maximum size of the local registers in the item max_locals. To allocate more local registers, we reset the value of the item max_locals to the twice of the original size at first. And we insert a new register behind one original register as the security-level container for the information in original register, and recalculat the local registers' indices in instructions. By this way it is convenient to calculate the index of one local register's security-level container (the index of the local register plus one). We give an example of allocating new registers as seceurity-level containers in Figure 1.

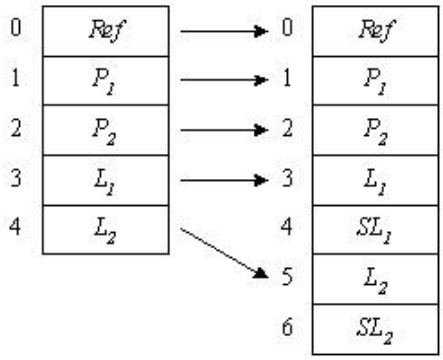

Figure 1: An example of allocating additional local registers. Ref is a reference to the method's instance. P_1 and P_2 are the parameters of the method. V_1 and V_2 are the local variables in the method. SL_1 and SL_2 are the security-levels of the information in V_1 and V_2 respectively.

For the element on the operand stack, we allocate an additional stack element as the security-level container of the information in the original element. Similar to the local registers, we reset the value of the item max_stack

defining the maximum size of the stack to the twice of original size. We store the security-level of the information in one stack element to the element just before the original one. We give an example of allocating new stack elements as security-level containers in Figure 2.

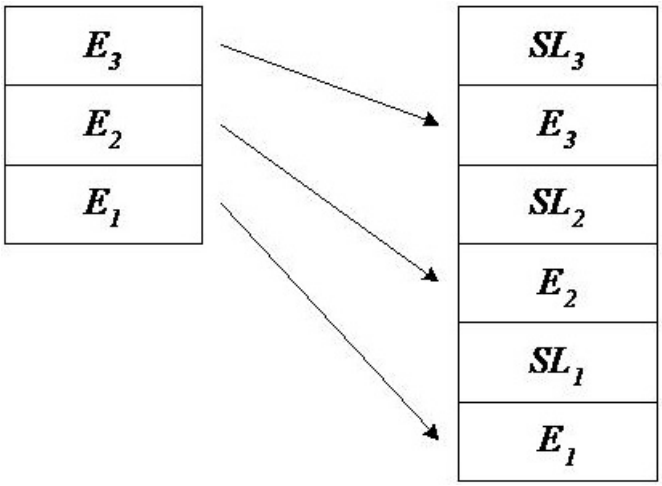

Figure 2: An example of allocating additional operand stack. E_1, E_2 and E_3 are the elements on the operand stack. SL_1, SL_2 and SL_3 are the security-levels of the information in E_1, E_2 and E_3 respectively.

For the class field, we add an additional filed as the security-level container of the information in the original filed of primitive types and type `String`. No matter what type original field is, the added field's type is `byte` because all security-levels are integers. The added field's name is the original field's name suffixed with "_SL". The added field has the same access flag as the original one. Since there are no ordering constraints on the *Constant Pool* and *Fields* structures, any new fields and entries could be appended rather than inserted in the middle in order to preserve the indices of existing entries. By this way we encapsulate the original class's data and the data's security-levels to the modified class. That is why it is not necessary to add an additional field for the original field of other class types. As for the field of an array `T[n]`, we add one additional field of type `byte` to store the security-levels of the array itself. (The security-level of an array itself is the security-level of the variable used to define the array's length.) If the element's type in the array is primitive type or type `String`, we add a second field of an array `byte[n]` to store the security-levels of the elements in the array. We give an example of adding new fields as security-level containers Figure 3.

Method Redefinition. The information could be transferred not only among the information carriers in one method, but also between methods by arguments and return values. Thus it is necessary to add new arguments and return values to transfer the security-levels of the information being transferred between methods at the same time.

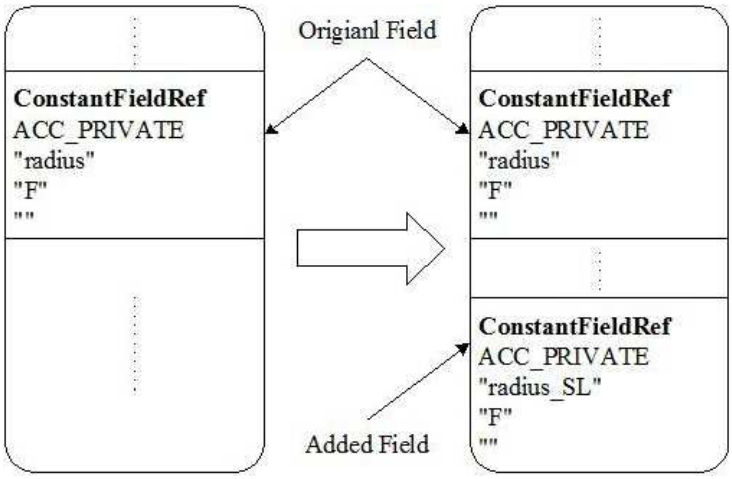

Figure 3: An example of adding field

Given one method having one or more arguments, we add one new argument of type `byte` for each original argument of primitive type or type `String` to transfer the security-level of the information in the original argument from the caller method to the callee method. As for the argument of an array, we add new argument(s) for it just like we add additional field(s) for the original field of an array. Because the arguments' names are not saved in the class file, we need not name the new arguments. We insert the new argument as the security-level container of the information in one original argument just after the original argument. By this way, each local register storing sensitive information is flowed by the register storing the security-level of the information when the arguments are stored to local registers. For example, considering a method $voidCircle(floatx, floaty, floatr)$ with the descriptor $(FFF)V$, we add one argument of type `byte` after each original argument. Thus the descriptor of the modified method Circle is $(FBFBFB)V$.

Given one method with a return value of primitive type or type `String`, we alter the return type to an array of the original return type, which has two elements: the first one is the return value and the second one is the security-level of the return value. (In the caller method, we convert the security-level to the type `byte`.) If the return type is an array `T[n]`, we alter the return type to an newly defined class. If the type of the element in array is primitive type or type `String`, the new class has three fields, whose types are `T[n]`, `byte [n]` and `byte`. Or else the new class has two fields, whose types are `T[n]` and `byte`. By this way we could transfer the security-level of the information in the original return value from the callee method to the caller method. For example, considering a method $floatarea()$ with the descriptor $()F$, we alter the return `type` to the array of type `float`. Thus the descriptor of the modified method area is $()[F$.

As mentioned above, we encapsulate the data and the data's security levels of one class to the modified class.

Therefore the argument or the return value of class types can transfer the data and the security-levels together, and it is not necessary to do any modification for them.

3.3 Instruction Insertion

To achieve dynamic verification, we need to insert proper instructions to calculate the security-levels of the information in the mobile code's data carrier and check whether every data-leaking channel in the mobile code is secure. To reduce the additional overhead in runtime caused by the bytecode modification we make the JVM execute the inserted instructions and the original instructions in one frame, that is, the inserted instructions and the original instructions share one set of local registers and one operand stack. (The adding of security-level containers mentioned above also follows this principle.) Therefore we should make sure that the inserted instructions would not do any harm to the original functions of the class. Another important thing is that the instruction addresses should be recalculated so that the conditional instructions could branch to the correct instruction.

Instruction Insertion for Intra-procedural Information Transferring. The information transferring can be divided into explicit transferring and implicit transferring. By the algorithms given in [10], we can partition the bytecode of one method into explicit blocks and implicit blocks. In explicit blocks, the information is transferred from the used variable(s) to the defined variable. Thus we should insert proper instruction(s) to assign the security-level of the used variable or the LUB of the security-levels of the used variables to the security-level container of the defined variable. In implicit blocks, the information is from the conditional variable of the implicit block to the defined variables in the implicit block additionally. Thus besides the instructions inserted in explicit blocks, we should insert proper instructions to assign the LUB of the conditional variable' security-level and the defined variable's original security-level to the security-level container of the defined variable. And we also need to insert instructions to calculate and store the conditional variable's security-level at the beginning of one implicit block.

The execution of a method's bytecode is a procedure of pushing data to the stack and popping data from the stack. According to the operation on the stack, the JVM bytecode instructions could be divided into three kinds: loading instructions (those pushing data to the stack, such as *iload*, *faload*, *bipush*), storing instructions (those popping data from the stack, such as *lstore*, *putfield*, *pop*) and operating instructions (those popping and operating two element on the stack top and pushing back the result to the stack, such as *dadd*, *lrem*, *ior*). In particularly we consider faload as a loading instruction but not an operating instruction because the semantics of faload is loading data to the stack and such classification could reduce the number of inserted instructions for faload. The similar cases are *putfield*, *getfield*, *iastore*, etc.

In explicit blocks considering the operand stack in JVM is LIFO (last-in-first-out), we insert instruction(s) loading the security-level from proper container to the stack for each loading instruction after it, and insert instruction(s) storing the security-level from the stack to proper container for each storing instruction before it. The operating instruction is a little complicated. One operating instruction will first pop elements from the stack and then push back the result to the stack. Therefore we insert the instructions popping the security-levels of the operands and calculating the LUB of them before the operating instruction, and insert instructions loading the result of LUB calculation to the stack after the operating instruction. To reduce the number of the inserted instructions, we make instructions calculating LUB as one subroutine. We give an example of inserting instructions in Figure 4. We list the Java source code and the original bytecode compiled from it at the left. The bytecode at the right is the modified code. In modified bytecode the green instructions (1, 3, 10 and 12), the blue instructions (7 and 16) and red instructions (4, 6, 13, 15 and 30 to 42) are inserted for loading instructions, storing instructions and operating instructions in original bytecode respectively.

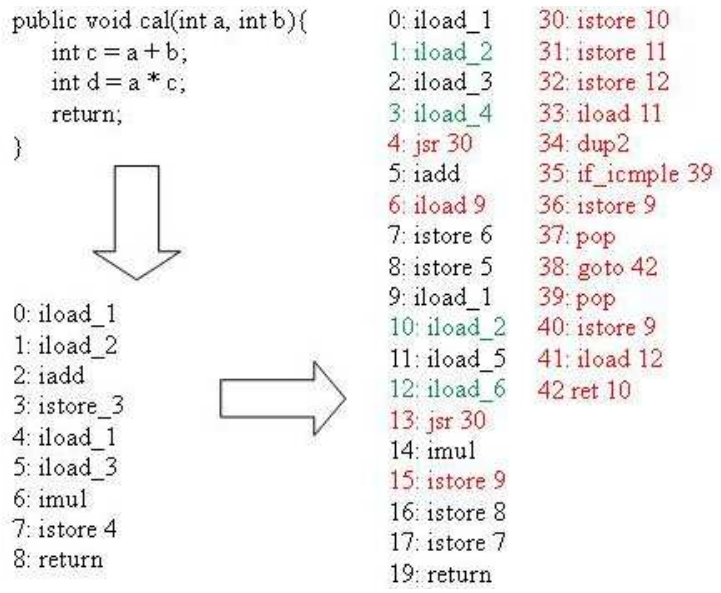

Figure 4: An example of inserting instructions

In implicit blocks besides the instructions inserted in explicit blocks, we should insert additional instructions to calculate the LUB of the security-level of defined variable and the conditional security-level of the implicit block, and assign the result to the defined variable's security-level container. For an implicit block the conditional security-level of is the LUB of all security-levels of its conditional variables. And the current environment security-level SL_c of one implicit block is the LUB of the old SL_c and the conditional security-level of the implicit block. Thus we allocate an array of type `byte` to store the environment security-levels of each layer for nested implicit blocks. At the beginning of an implicit block in nested

implicit blocks, we store the old SL_c to the array and calculate the new one. Then at the end of that implicit block we load back the old SL_c from the array. By this way in each block of nested implicit blocks we could use the correct current environment security-level to calculate the defined variable's security-level. In JVM the operand stack is just a kind of intermedial information carrier and all data pushed to the stack could not be transferred to other carriers until they are popped from the stack. Considering this characteristic, we calculate the LUB of the conditional security-level and the defined variable's security-level only when the variable is popped from the stack, that is, we insert the instructions to calculate the LUB only for storing instructions rather than for all the loading instructions, operating instructions and storing instructions. By this way the additional overhead cause by bytecode modification could be reduced.

Instruction Insertion for Inter-procedural Information Transferring. If the type of a method's return value is not `void`, we alter the type of the return value to an array of original type. Therefore we should insert instructions into the callee method to encapsulate the return value and the security-level to an array of proper type so that a value of the correct type could be returned. The encapsulation procedure is 1) allocating a new array of the proper type with two elements, 2) storing the security-level to the second element and the return value to the first element, and 3) returning the reference of the array to the caller method. Furthermore, we should insert instructions into the caller method to push the elements of the returned array to the stack. To preserve the consistency of the arrangement of security-levels and information on the operand stack, we push the original return value (the first element) at first and then the security-level (the second element) to the stack. We also insert instructions to convert the security-level to type `byte` if it is not.

Instruction Insertion for Data-leaking Channel Checking. As mentioned in Section 2, at each data-leaking channel we should compare the security-level of the information to be sent with the clearance-level or security-level of the destination in order to check whether the data-leaking channel is secure. We insert the checking instructions after the instructions loading the information to be sent to the operand stack, but before the instructions sending the information. The checking procedure of is 1) at first reading the clearance-level or security-level of the destination from the certain local host file and pushing it to the stack, 2) then comparing the two security-levels or the security-level and the clearance-level on the stack, 3) if the security-level of the information to be sent is higher, the data-leaking channel is not secure and a user-defined exception is thrown out to inform the host user the mobile code is not secure. Or else the execution of the mobile code continues.

4 Information Transferring in Exception Handling

The Java programming language supports exception-handling mechanisms to ease the difficulty of developing robust software system. An exception will cause a non-local transfer of control and affect the information flow. In this section we will analyze the information transferring in the exception handling of Java bytecode and give the mechanisms to deal with the information flow.

4.1 Exception Handling in Java Bytecode

When a program violates the semantic constraints of the Java programming language, the Java virtual machine signals the error to the program as an *exception*. The Java programming language specifies that an exception will cause a non-local transfer of control from the point where the exception occurred to a point that can be specified by the programmer. An exception is said to be thrown from the point where it occurred and is said to be caught at the point to which control is transferred.

In Java all thrown exceptions are instances of classes derived from the class `java.lang.Throwable` and are raised as results of expression evaluations, statement executions or *throw* statements. A *try* statement is the exception-handling construct and consists of `try` block and, optionally, a `catch` block and a `finally` block. The legal constructs for a *try* statement are `try-catch`, `try-catch-finally` and `try-finally`. When an exception is raised in a statement within a `try` block or in some method called within a `try` block, control transfers to the `catch` block associated with the last `try` block in which control entered, but has not yet exited. This `catch` block is the nearest dynamically-enclosing `catch` block, and can be in the same *try* statement, in an enclosing *try* statement, or in a calling method. If a matching catch handler is found, the handler code is executed and normal execution resumes at the first statement following the *try* statement where the exception was handled. If no matching catch handler is found in the nearest dynamically-enclosing `catch` block, the search continues in the `catch` block of the enclosing try statement and subsequently in some calling method. Before the control exits a *try* statement, the `finally` block of the *try* statement is executed, if it exists, regardless of whether control exits the *try* statement with an unhandled exception. Thus the exception handling in Java can cause intra-procedural transfer of control and inter-procedural transfer of control. We summarize the exception handling process in Figure 5.

Furthermore in Java bytecode, exceptions can only be thrown explicitly by the instruction *athrow* or implicitly by some specific instructions such as those shown in Table 1. We define the security-level of one exception as the LUB of security-levels of the variables determining whether the exception is raised. For example consider the exception of type `NullPointerException` raised by

the instruction `iaload`. Since whether the exception is raised or not depends on the value of the variable *arrayref*, the security-level of the exception is the security-level of the variable *arrayref*. Considering that exception handling will cause not only intra-procedural control transfer but also inter-procedural control transfer, we add one new field *SL* of type `byte` to every exception class including both pre-defined exception classes and user-defined exception classes, and use it to store the security-level of the exception instance. By this way when an exception is thrown from the callee method to the caller method, we can trace the information flow correctly and understand the security-level of the exception in the caller method. As for the presentation format of *try* statement in Java bytecode, the exception table is used to indicate the scope of `try` blocks, the beginning address of `catch` blocks and the exception types that `catch` blocks can handle. The compiler generates an exception table entry for each `catch` block and one or more entries with type `any` for the `finally` block. In the following we will analysis the information flow in Java bytecode exception handling in detail.

4.2 Implicit Information Transferring in Exception Handling

As mentioned above, in Java bytecode exceptions can be thrown implicitly by some specific instructions, which are called as ***PEIs*** (Potential Exception-throwing Instructions). When the execution of Java bytecode encounters a PEI, where the control flow is transferred depends on that whether the PEI raises an exception and what exception the PEI will raise. In other words, the PEI acts as a branch node and it may have some of the branches shown in Figure 5. It means that one PEI can cause implicit information transferring just like the *if*-instructions and initiates an implicit transferring block. Obviously the conditional security-level of such one implicit transferring block is the security-level of the exception that can be raised by the PEI (or the LUB of the security-levels of all exceptions that can be raised by the PEI). To distinguish it from the conditional security-level of *if*-instructions, we call those conditional security-level as the *exceptional security-level* of the PEI. In Java Virtual Machine specification, which instructions can raise exceptions and what type of exceptions they can raise have been defined clearly. We can calculate the exceptional security-level of one PEI just before the PEI is executed. As for the scope of the implicit transferring block initiated by one PEI, it varies with the location where the exception(s) raised by the PEI can be handled, that is, in the same method where the PEI raises exception(s) or in the caller method. According to the Java Virtual Machine specification, 40 instructions could throw exceptions implicitly in the total 204 instructions in Java bytecode. And in those 40 PEIs, 7 instructions can only throw one type of exception and the others can throw two or more types. Thus for the exceptions that can be raised by one PEI in those 33 kinds, all of them may be handled in the same method where they are raised (causing only intra-procedural transferring), none of them may be handled in the same method where they are raised (causing only inter-procedural transferring), or some of them may be handled and the others can not be handled in the same method where they are raised (causing intra-procedural or inter-procedural information transferring). Since what type of exceptions that one PEI can raise has been defined by JVM specification and what type of exceptions one method can handle has been defined by the exception table, we can judge that one PEI may cause intra-procedural information transferring, inter-procedural transferring or both of them. We discuss these cases respectively.

Intra-procedural Information Transferring May be Caused. In this case, all the exception(s) raised by one PEI can be handled by the proper `catch` block in the same method. The branches of the implicit transferring block caused by the PEI are 1 and 5-7 (no `finally` block) or 3 and 5-6 (`finally` block specified) in Figure 5. The scope of the normal branch (1 or 3 in Figure 5) is from the immediate post-dominator of the PEI to the end of the `try` block. (This branch will be blank if the PEI is the last instruction in the `try` block.) And the scope of the exceptional branch(es) (5-7 or 5-6 in Figure 5) is the whole `catch` block(s) handling the exception(s). Thus when the execution encounters one PEI that can only raise intra-procedural implicit information transferring, we should backup the current environment security-level SL_c, and set SL_c to the LUB of original SL_c and the exceptional security-level of the PEI. Then at the end of each branch we should set the SL_c back to the original one.

Inter-procedural Information Transferring May be Caused. In this case, no proper `catch` block can be found in the same method for the exception(s) raised by the PEI, and JVM throws the exception(s) to the caller method. The branches of the implicit transferring block caused by the PEI is 1 and 2 (no `finally` block) or 3 and 4-9 (`finally` block specified) in Figure 5. The scope of the normal branch (1 or 3 in Figure 5) is from the immediate post-dominator of the PEI to the end of the method exclusive the `finally` block if it is specified. And the scope of the exceptional branch(es) (2 or 4-9 in Figure 5) is the whole `catch` block(s) that can handle the exception(s). (Such `catch` block may be in the caller method or in the further outer caller method, or does not exist in which case the exceptional branch is blank.) Since the control may be transferred to the caller method in this case, we should insert proper instructions to transfer the exceptional security-level of the PEI raising the exception(s) to the caller method. As mentioned above, we add one new field SL to every exception class to transfer the security-level between methods in the case of one method's exceptional exiting. Thus we should set the field SL of the current exception instance to the current environment security-level before the execution exits the method. If there is one `finally` block specified in the *try* statement where the exception is raised, we can in-

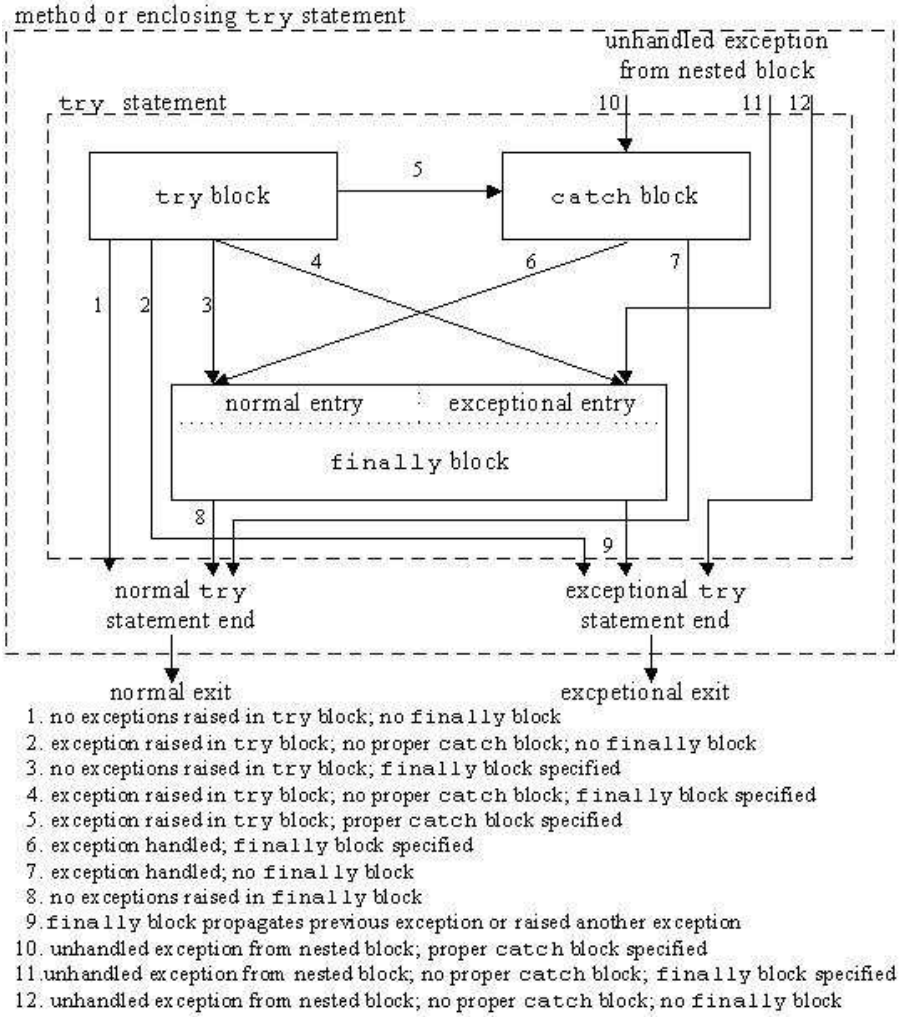

Figure 5: Control flow in Java exception-handling constructs

sert the instructions setting the field SL in the `finally` block. Or else we should add one `finally` block that does nothing but the setting of the filed SL. When the execution encounters one PEI that may raise inter-procedural implicit information transferring, we should set the SL_c to the LUB of the SL_c and the exceptional security-level of the PEI. If a `finally` block is specified, we should backup the current environment security-level SL_c before we change it, then restore the backup SL_c at the beginning of the `finally` block in order to exclude it from the normal branch, and at the end of the `finally` block we set SL_c to the one that is calculated just before the PEI. As for the exceptional branch, we should backup the current environment security-level SL_c (which is the SL_c of the method being executed, not the method where the exception is raise since at that point the execution has exited that method), and then set the SL_c to the LUB of the original SL_c and the security-level stored in the filed SL of the exception. At the end of the exceptional branch we should restore the SL_c to the original one.

Both Intra-procedural and Inter-procedural Information Transferring May be Caused. In this case, some of the exceptions that may be raised by the PEI can be handled in the same method and the others can not be. The branches of the implicit transferring block caused by the PEI are 1, 2 and 5-7 (no `finally` block) or 3, 4-9 and 5-6 (`finally` block specified) in Figure 5. The scope of the normal branch (1 or 3 in Figure 5) is from the immediate post-dominator of the PEI to the end of the method exclusive the `finally` block if it is specified. The scope of the intra-procedural exceptional branch(es) (5-7 or 5-6 in Figure 5) is from the beginning of the `catch` block that can handle the exception in the same method to the end of the method exclusive the `finally` block if it is specified. The scope of the inter-procedural exceptional branch is the whole `catch` block that can handle the exception, which may be in the caller method or in the further outer caller method, or does not exist in which case the inter-procedural exceptional branch is blank. What we should do in the normal branch and inter-procedural exceptional branch is same to what we should do in case of inter-procedural implicit information transferring. As for

Table 1: Some instructions that could throw exceptions in Java bytecode

Instruction	Stack	Exceptions Thrown
aaload	$arrayref, index \Rightarrow v$	ArrayIndexOutOfBoundsException, NullPointerException
bastore	$arrayref, index, v \Rightarrow$	ArrayIndexOutOfBoundsException, NullPointerException
iaload	$arrayref, index \Rightarrow v$	ArrayIndexOutOfBoundsException, NullPointerException
idiv	$value1, value2 \Rightarrow result$	ArithmeticException
instanceof	$objectref \Rightarrow restult$	Resolution Exceptions
invokestatic	$[arg_1, [arg_2]] \cdots$	Resolution Exceptions
ldc	$\cdots \Rightarrow item$	Resolution Exceptions
newarray	$count \Rightarrow arrayref$	NegetiveArraySizeException
putfield	$objectref, value \Rightarrow$	Resolution Exceptions, NullPointerException

the intra-procedural exceptional branch, we should only exclude the `finally` block from the intra-procedural exceptional branch with the same way used in the normal branch if the `finally` block is specified.

Locating the Scope of try block, catch Block and finally block. The scope of the `try` block is indicated by the index pairs $[from, to)$ in the exception table entries. Each different pair determines one `try` block.

While the locating of the `finally` block's scope is complicate. The exception table entry with handler type `any` is generated and could be only generated for the `finally` block in the *try* statement by the complier. Thus if there are any entries with handler type `any` in the exception table, the `finally` block is specified. In general the code of the `finally` block could be complied in two ways: complied into one subroutine or appended to the code of the `try` block and the `catch` block as shown in *b* and *c* of Figure 6. In the case of subroutine, the instruction immediately following the `try` block (whose index is indicated by the value of the *to* in the exception table entry) should be the instruction *jsr i*, which transfer the control to the subroutine compiled from the `finally` block. Assuming the index of the instruction *ret* in the subroutine is i', the scope of the `finally` block is (i, i').

In the other case, the last instruction in the `try` block could not be the instruction *jsr*. To divide the `try` block from the following `catch` or `finally` block, the complier generates instruction *return* ($\alpha return$) or *goto* between them. Assuming the index of that dividing instruction is j' and the index of the last instruction in the `try` block is j, the scope of the `finally` block appended to the `try` block is $[j, j']$.

The index of the first instruction in the `catch` block is indicated by the $Target'$s value in the exception table entry. From the first instruction, we search the instruction-block matching the instructions in $[j, j']$ and the last one of the found instruction-blocks is the `finally` block appended to the `catch` block. If there are any entries whose handler types are not type *any* in the exception table, the `catch` block is specified. The locating of the `catch` block's scope is based on the scope of the `finally` block. In the case of subroutine, assume the value of the *Target* in the exception table entry indicating the `catch` block is k and the index of the instruction *jsr* transferring the control to the `finally` block is k'. The scope of the catch block is $[k, k']$. In the other case, assume the value of the *Target* is k and the scope of the `finally` block appended to the `catch` block is $[l, l']$. The scope of the `catch` block is $[k, l]$.

For example, in the *b* of Figure6, the scope of the `try` block is [0, 10), the scope of the `finally` block is (36, 42) and the scope of the `catch` block is [16, 22). In the *c* of Figure6, the scope of the `try` block is [0, 12), the scopes of the `finally` blocks are [12, 13], [25, 26] and [34, 35] respectively, and the scope of the `catch` block is [19, 25). [1]

Procedure of Dealing with Implicit Transferring Block Caused by PEIs. Based on the analysis above, we could find that the PEIs in Java bytecode act as the *if*-instructions in the information transferring. Here we define the procedure of dealing with the implicit block caused by PEIs and give an example in the following.

Given a method m, the procedure could be defined as following.

1) Locate the scopes of all the `try` blocks, `finally` blocks `and` catch blocks in m.

2) Search for all the PEIs in m and calculate the exceptional security-level of each PEI just before it.

3) If all the PEIs in one `try` block are intra-procedural PEIs, at the end of the `try` block and the corresponding `catch` blocks (if they are specified) restore the SL_c that is backupped before the first PEI in the `try` block.

4) If any PEIs in one `try` block is inter-procedural PEIs, at beginning of the corresponding `finally` block backup the SL_c and restore the SL_c that is backupped before the first PEI in the `try` block. Then at the end of the `finally` block, restore the SL_c that is backupped at the beginning of the `finally` block.

[1] The bytecode in *b* of Figure6 is generated by JDK 1.4.1 and the one in *c* of Figure6 by JDK 1.4.2.

```
void testOfCatch(int[] a, int b){
    try{
        int c = a.length;
        int d = a[b];
        rauseException();
    }catch(NullPointerException e){
        handleException(e);
    }finally{
        wrapItUp();
    }
}
```

a

```
void testOfCatch(int[], int);
0: aload_1
1:arraylength
2: istore_3
3: aload_1
4: iload_2
5: iaload
6: istore 4
8: aload_0
9: invokevirtual #5
10: jsr 36
13: goto 44
16: astore_3
17: aload_0
18: aload_3
19: invokevirtual #8
22: jsr 36
25: goto 44
28: astore 5
30: jsr 36
33: aload 5
35: athrow
36: astore 6
38: aload_0
39: invokevirtual #6
42: ret 6
44: return
Exception table:
from   to   target   type
  0    10     16     Class java/lang/
                     NullPointerException
  0    13     28     any
 16    25     28     any
 28    33     28     any
```

b

```
void testOfCatch(int[], int);
0: aload_1
1:arraylength
2: istore_3
3: aload_1
4: iload_2
5: iaload
6: istore 4
8: aload_0
9: invokevirtual #5
12:aload_0
13: invokevirtual #6
16: goto 41
19: astore_3
20: aload_0
21: aload_3
22: invokevirtual #8
25: aload_0
26: invokevirtual #6
29: goto 41
32: astore 5
34: aload_0
35: invokevirtual #6
38:aload 5
40: athrow
41: return
Exception table:
from   to   target   type
  0    12     19     Class java/lang/
                     NullPointerException
  0    13     32     any
 19    25     32     any
 32    34     32     any
```

c

Figure 6: The `finally` block in Java bytecode

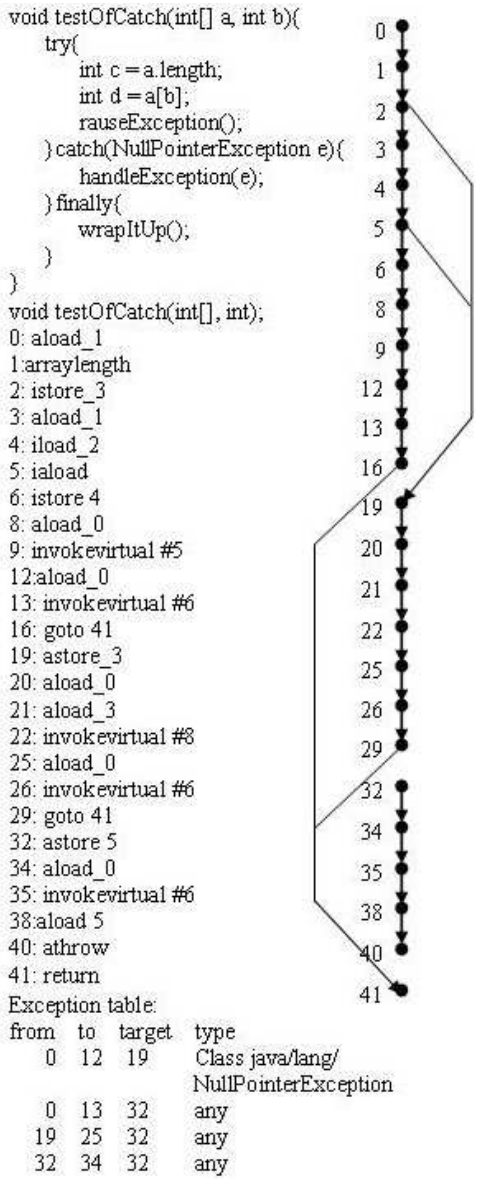

Figure 7: An example of implicit transferring caused by PEI

```
void testOfCatch(int[], byte[], byte, int, byte);
0: aload_1          23: aload_0          47: getfield #9
1: iload_3          24: invokevirtual #5 48: ifne 52
2: dup              25: aload_0          49: aload 10
3: istore 13        26: invokevirtual #6 50: iload 12
4: istore 7         27: goto 56          51: putfield #9
5: arraylength      30: astore 6         52: aload_0
6: istore 6         31: aload 6          53: invokevirtual #6
8: aload_1          32: getfield #9      54: aload 10
9: aload_2          33: dup              55: athrow
10: iload 4         34: ifne 36          56: return
11: iload 5         35: iload 13
12: iload_3         36: istore 12        80: istore 14
14: jsr 80          37: aload_0          81: dup2
15: istore 13       38: aload 6          82: if_icmple 86
16: swap            39: invokevirtual #8 83: istore 15
17: iload 4         40: aload_0          84: pop
18: iaload          41: invokevirtual #6 85: goto 68
19: istore 9        42: goto 56          86: pop
20: iaload          45: astore 10        87: istore 15
21: istore 8        46: aload 10         88: ret 14
Exception table:
from   to   target   type
    0   25   30     Class java/lang/NullPointerException
    0   25   45     any
   30   49   45     any
   45   52   45     any
```

Figure 8: The modified bytecode

5) At the beginning of each `catch` block, check the value of the field SL in the exception instance caught. If it is not 0, set the SL_c to the LUB of the SL and SL_c.

6) If there are any inter-procedural PEIs and the `finally` block is specified, set the field SL of the exception instance caught in the `finally` block to the SL_c.

7) If there are any inter-procedural PEIs and the `finally` block is not specified, add one `finally` block to m and set the field SL of the exception instance caught in that `finally` block to the SL_c.

Consider the section of Java program whose bytecode and CFG are shown in Figure7. Using the procedure above we could deal with implicit transferring caused by PEIs in that Java bytecode. We give the modified bytecode in Figure8. Referring to the exception table, we can find the scope of the `try` block is [0, 12), the scopes of the `finally` blocks are [12, 16], [25, 26] and [34, 35] respectively, and the scope of the `catch` block is [19, 25) in Figure7. The PEIs are *arraylength* at index 1 and *iaload* at the index 5 in Figure7. (Here to simplify the example we assume that the instruction *invokevirtual* itself will not raise any exception.) We insert instructions of [2, 3] and [11, 15] in Figure8 to calculate the exceptional security-level of the two PEIs. By checking the handler type of the `catch` block, we can find that the *arraylength* is intra-procedural PEI and the *iaload* is inter-procedural PEI. Thus we insert instructions of [46, 51] in Figure8 to the `finally` blocks to set the correct current environment security-level. At the beginning of the `catch` block we insert instructions of [31, 36] in Figure8 to check whether the field SL of the caught exception instance is 0 and set the current environment security-level to the correct value.

4.3 Explicit Information Transferring in Exception Handling

In Java bytecode the exception can be thrown by the instruction *athrow* explicitly. Thus the instruction *athrow* can cause explicit information transferring. Similar to the implicit information transferring caused by PEIs, the explicit information transferring caused by athrow can also be divided into the intra-procedural and inter-procedural transferring. But different from the implicit one, the explicit information transferring caused by *athrow* acts as unconditional branch instruction *goto* in the information flow. To deal with the explicit information transferring caused by *athrow* is quite simple. What we should do is to set the field SL of the exception instance that will be thrown by *athrow* to the current environment security-level SL_c.

5 Conclusion

In this paper, we put forward a dynamic verification approach based on the Secure Information Flow to protect the host confidentiality in mobile code systems. The secure information flow property of programs was first formulated in [6, 7].

In [1, 2, 3, 4, 11], the authors adopted the Secure Information Flow to address the security problems in mobile programs. But all the verification approaches of mobile code security in these works are static approaches and could not achieve satisfying verification precision in implicit transferring. Our approach presented in this paper is a dynamic approach and improve the verification precision greatly especially in implicit information transferring. We adopted bytecode modification to implement our security model dynamically in order to preserve the portability of JAVA mobile code. To reduce the additional overhead caused by inserted instruction, we make the JVM execute the inserted instructions and original instructions in the same frame.

And in this paper we also analyze the information flow in the Java bytecode exception handling, which has not been done in [1, 2, 3, 4, 11]. We add a new field to all exception classes to transfer the security-level of the exceptions raised. We give the mechanism to locate the scope of try, catch and finally blocks in Java bytecode and the procedure to deal with information flow in exception handling, which is almost impossible for static approaches.

As future work, we will extend our research to the Java bytecode concerned with the multi-dimension array and local methods. We will also improve our mechanism of

inserting instructions to reduce the additional execution overhead caused by the inserted instruction. And we will complete the prototype of the verification tool based on the approach in this paper.

References

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

[2] 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.

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

[4] G. Bian, K.Nakayama, Y. Kobayashi, and M. Maekawa, "Mobile code security by Java bytecode dependence analysis," *Proceedings of ISCIT 2004*, pp. 923-926, Sapporo, Japan, Sep. 2004.

[5] G. Cohen, J. Chase, and D. Kaminsky, "Automatic program transformation with JOIE," *Proceedings of the 1998 USENIX Annual Technical Conference*, pp. 167-178, 1998.

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

[7] 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.

[8] X. Leroy, "Java bytecode verification: An overview," *Lecture Notes in Computer*, vol. 2102, pp. 265-285, July 2001.

[9] L. Lindholm, and F.Yellin, *The Java Virtual Machine Specification,* 2_nd *edition*, Addison Wesley, 1999.

[10] D. Lu, K. Nakayama, Y.Kobayashi, M. Maekawa, "Abstract Interpretation for Mobile Code Security," *Proceedings of ISCIT 2005*, pp. 1068-1071, Beijing, China, Oct. 2005.

[11] D. Lu, K. Nakayama, Y.Kobayashi, and M. Maekawa, "Verification for host confidentiality by abstract interpretation in mobile code systems," *Mobility Conference 2005*, Guangzhou, China, Nov. 2005.

[12] The Apache Software Foundation, *BCEL: Byte Code Engineering Library*, 2003, http://jakarta.apache.org/bcel.

Dan Lu is a Ph.D. candidate at of the Graduate School of Information Systems, University of Electro-Communications, Tokyo, Japan. He received his B.E. and M.E. degree in Harbin Engineering University, Harbin, China in 2001 and 2003. His research interests include: Java Virtual Machine, Java bytecode, and Mobile code Security.

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.

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.

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.