Java Mobile Code Dynamic Verification by Bytecode Modification

DAN LU

Graduate School of Information Systems

THE UNIVERSITY OF ELECTRO-COMMUNICATIONS

June 2008

Java Mobile Code Dynamic Verification by Bytecode Modification

バイトコードモディフィケーションによる JAVAモバイルコードの動的な検証方法の研究

概　要

モバイルコードはネットワーク内で移動、リモートマシンで実行されるプログラムである。こうしたプログラムは、セキュリティが重要な課題となる。Java言語は移動性、安全性、プラットフォーム独立などの特徴を有し、現在モバイルコードの実現に広く用いられている。本研究は、Javaモバイルコードシステムのホストセキュリティを保守するため、検証精度がより高い検証方法を確立することを目指している。

アクセル制御とオーセンティケーションなどのセキュリティ保護方式と比べて、情報流分析技術はセキュリティ保護、特に秘密保護に有用な考え方である。ソースコードを分析対象としている従来の情報流分析に対して、Javaモバイルコードシステムにおいてはモバイルコードを実行するホストはソースコードを知ることはなし、バイトコードという形のコードを実行する。ですから、本研究ではバイトコードの情報流分析を行った。

モバイルコードに情報流分析を適用したアプローチはいくつかあるが、これらのアプローチはモバイルコードシステムにおけるセキュリティの特徴を見落とし、検証精度が満足できるレベルに達成していない。本研究は、モバイルコードシステムにおけるオブジェクトとサブジェクトを明確にし、モバイルコードシステムに相応しいセキュリティモデルを提案した。オブジェクトであるローカルホストのデータにセキュリティレベルを、サブジェクトである第三者ホストに許可レベルを割り当てる。そして、モバイルコードの中の情報流を監視し、情報のセキュリティレベルを計算する。モバイルコードが情報を第三者ホストに送信する時、情報のセキュリティレベルと第三者ホストの許可レベルによって情報漏洩が発生するかを判断する。

また、実行前にモバイルコードを検証する静的なアプローチに対して、本研究は実行中に検証を行う動的なアプローチである。検証対象のモバイルコードがJavaVMへ送られる前に、モバイルコードの情報流に対してもとのモバイルコードを変更し、検証機能を実現するコードを追加する。そして、変更されたモバイルコードを実行するとき、もとのモバイルコードの機能と共に検証機能も行う。実行中の状況を把握できるため、本研究はより高い検証精度を達成した。

その上、動的な検証方法しか対応できない異常処理の情報流も本研究で検討された。本研究はモバイルコードにおける異常処理が如何な情報流を生成するかを分析して、対応する検証コードの生成方法を提案した。異常処理の情報流の検証が対応されたため、本研究では検証精度と実用性が満足できるレベルに到達した。

Java Mobile Code Dynamic Verification by Bytecode Modification

Dan Lu

ABSTRACT

Mobile code program can be transmitted via network from a remote source to a local system and be executed on that local host. And such programs may raise the security problems of the host because they could interact with the resources of the local host and malicious or defective programs will tamper data or release secure information of the local host. The Java language has been used widely in the implementation of mobile code systems because of its portability, security and platform-independency. In our research, we are heading for building a verification approach with high precision in order to protect the host security in mobile code systems.

Compared with the security mechanisms such as access control and authentication, the Secure Information Flow theory is a useful mechanism for the security protection, especially for the confidentiality protection. In the traditional information flow analysis, the source code is taken as the analysis object. While in mobile code systems, the host executing the mobile code can only get the bytecode of the programs. Therefore, we analyzed the bytecode's information flow in our research.

Though several approaches have used the information flow analysis, those approaches neglected the characteristics of security demand in mobile code systems and did not achieve satisfying verification precision. In our approach, we make it clear that what are the subjects and objects in mobile code systems, and put forward the appropriate security model. We assign security-levels to the data of the local-host and clearance-level to the third-party hosts. Then we trace the information flow during the mobile code and check whether a data-leaking is raised when the mobile code tries to send data to a third-party host.

Furthermore, different from static approaches that verify the mobile code before the JVM executes the bytecode, our approach is a dynamic approach that verifies the mobile code when the JVM is executing the bytecode. Before the mobile code is sent to the JVM, we analyze the information flow in the bytecode and insert proper instructions implementing the verification function into the original bytecode. Thus when the JVM executes the modified bytecode, the verification function is done as while as the original function of the mobile code. The dynamic approach can get the runtime information during the mobile code execution (such as which branch of the implicit information transferring will be executed, whether an instruction will throw an exception or not, and so on), and it can achieve better verification precision than static ones.

In addition, we also discuss the information flow during the exception handling in our research, which is almost impossible for static verification approaches. We analyze what kind of information flow can be caused during the exception handling of Java mobile code, and put forward the corresponding verification bytecode for the information flow caused by exception handling. Since our approach can deal with the verification of the exception handling in bytecode, the verification precision and practicality of our approached are improved further.

Contents

1 Introduction

With the significant development of distributed computing and the internet technology, the utilization of mobile code systems (such as applets, mobile-agent systems) is increasing. And the Java language is widely used to build mobile code systems because of its mobile and safe characteristics.

This new mode of distributed computation promises great opportunities for electronic commerce, mobile computing, and information harvesting as well as the problems of security. As a distributed system architecture, a mobile code system usually involves two processes, that is, a code producer process (e.g., a web server process) and a code consumer process (e.g., a web browser). Mobile programs are able to migrate from remote sites to a host when the producer process sends the consumer process a program (e.g., a mobile agent), and interact with the resources and facilities of the host, which causes side effects to be produced on the consumer side. Such an arrangement gives rise to serious security threats. If there is no control on this kind of mobile programs that can be executed in the consumer process, a malicious mobile program could try to observe, leak or alter the information it is not authorized on the host and then compromise data confidentiality, system integrity and resource availability. The situation in mobile code systems requires more stable security mechanisms to provide protection of the host against the potential attack caused by executing such malicious code than in the stand-alone systems. It has been a general consensus that security is the key to the success of mobile code computation.

The host security involves three aspects: confidentiality, integrity and availability. Decades of research in operating systems has provided significant experience and insight into the nature of system security. Some protection techniques used in operating systems, 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. However, the existing protection mechanisms for operating systems do not fully address the security needs of mobile code systems. In Java mobile code systems, the availability property can be protected by JVM secure mechanism. Existing approaches for enforcing confidentiality and integrity properties tend to confine mobile code so as to ensure that it can do no harm to the host. This goal is achieved by enforcing stringent access control policies that prevent mobile code from executing any action that can potentially compromise the security of the host system running the code. For example, Java applets are denied to read or write any files resident on their host system since malicious applets may be able to use such an access to alter user data or leak it to unauthorized parties. Such access control policies are useful to keep malicious applets in check. However they preclude a large number of useful applications of mobile code.

An ideal mobile code security framework should put less restriction as much as possible on the mobile code on the precondition that the host is protected from the attack of malicious or faulty mobile code. Compared to simply informal endorsement such as Authentication and Access Control, the Secure Information Flow is a kind of program-analytic mechanisms and more precise. Information-flow control is a technique that provides ensuring confidentiality and integrity. However, the studies in this field focus on high-level languages and the source code of the program is analyzed in compile time. These approaches cannot be applied to mobile code system because the consumer of the mobile code system cannot obtain the source code of the mobile programs that migrate from the producer.

The approaches for Java mobile code security by now are almost static ones. They verify Java mobile code and decide the code is secure or not before the local JVM executes the code. Thus all static approaches could not get any runtime information of the mobile code execution. That inherent limitation of static approaches causes that those approaches cannot achieve satisfying verification precision in implicit information transferring since it is impossible for static approaches 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. And this limitation makes the static approaches lose the practicality.

This dissertation provides a dynamic verification approach to protect the host security in Java mobile code systems by the bytecode modification. We analyze the information flow of Java bytecode, and put forward a suitable security model for mobile code systems based on the secure information flow theory. In our security model, we do not restrict the mobile code to read sensitive information from the host and transfer the information in the mobile code. Instead, we record and calculate the information flow to master where the sensitive information is in the mobile code. Only when the mobile code sends information out, we check and restrict the possible information-leak.

Furthermore, we make use of the bytecode modification technique to achieve dynamic implementation of our security model. Before the JVM executes the mobile code, we modify it in order to add the verification function into the original bytecode. Then the modified bytecode is submitted to the local JVM, and its original functionalities and the added verification function are executed at the same time.

We analyze the structure of Java bytecode and class files, and put forward the modification mechanism that can insert appropriate instructions into the original bytecode to trace and check the information flow during the bytecode execution. Especially, our research covers the information flow in Java bytecode exception handling, which makes our approach more practical and achieve better verification precision.

2 Mobile Code Security

2.1 Mobile Code System

Mobile code is an architectural paradigm for structuring distributed software systems. Different from the other paradigms used to construct the distributed system such as Client-Server Paradigm, the most impressive character of mobile code systems is the notion of code mobility: communicating processes in mobile code systems exchange program code instead of simply passing data messages.

In one program, there are three elements:

- Data (stored result sets)
- Code (commands)
- State (current execution status of the program)

A distributed computing system can be called as a mobile code system if there are the codes that can migrate from one host to another. Mobile code systems can revolutionize the design and development of distributed systems. In the following, we will provide a brief overview and comparison of four programming paradigms for distributed computing: *client-server*, *code-on-demand*, *remote revaluation* and *mobile agents*.

2.1.1 Client-Server Paradigm

In the client-server paradigm, there is a set of services provided by the server, by which the client is able to access to some resources (e.g., databases, files). Although the service is used by the client, the code that implements these services still belongs to the server. In a word, it is the server itself that executes the service, and thus has the processor capability. If the client wants to get information from certain resource hosted by the server, it is able reached the data resource not by itself but by seeking help from the server to provide an appropriate service instead. The server owns all, including the resources, the tool to get the resource and processor. Currently, most distributed systems were constructed on this paradigm in which a wide range of technologies have been involved, such as Remote Procedure Calling, Object Request Brokers (CORBA) and Java Remote Method Invocation (RMI). In the client-server systems, the ownership of the code used in the service and the host is not changed during the process of information transition, only the data of the program is transferred, thus these systems are not considered as mobile code systems.

- Data → mobile
- Code→ static
- State→ static

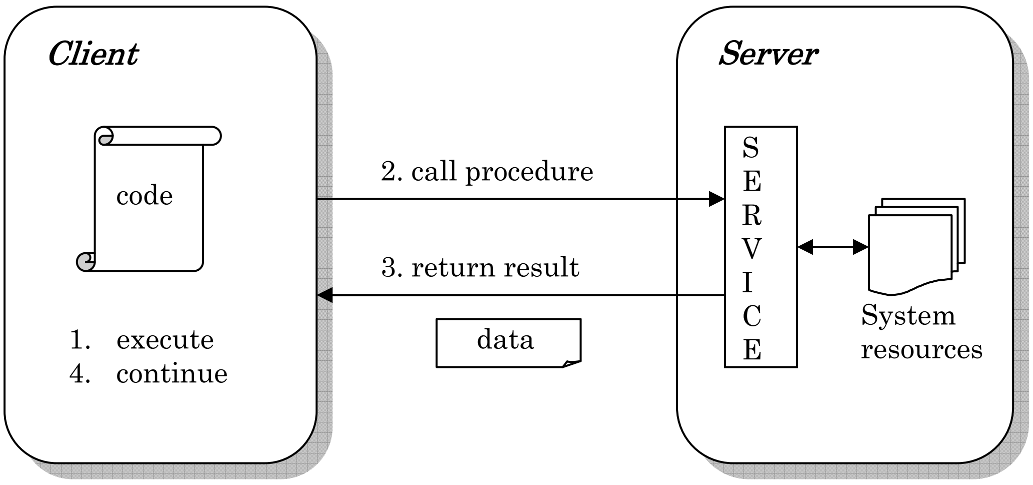


Figure 2-1. Client-Server Paradigm.

2.1.2 Code-on-Demand Paradigm

In the code on demand paradigm, one client has to first get the know-how when the client needs it because the client initially is not able to perform its task properly due to a lack of code (know-how). In the network, there is another host can provide the code needed. Once the code is received, the client performs the computation by itself. The client holds the processor capability as well as the local resources. Different from the client-server paradigm, the client does not need the detail information of the remote since all the necessary code will be transmitted to the local system. The client has both the resources and processor, while the server has the know-how. A good example is the Java applet. In the paradigm, applets are downloaded from remote sites in the web and executed locally.

- Data → static
- Code→ mobile
- State→ static

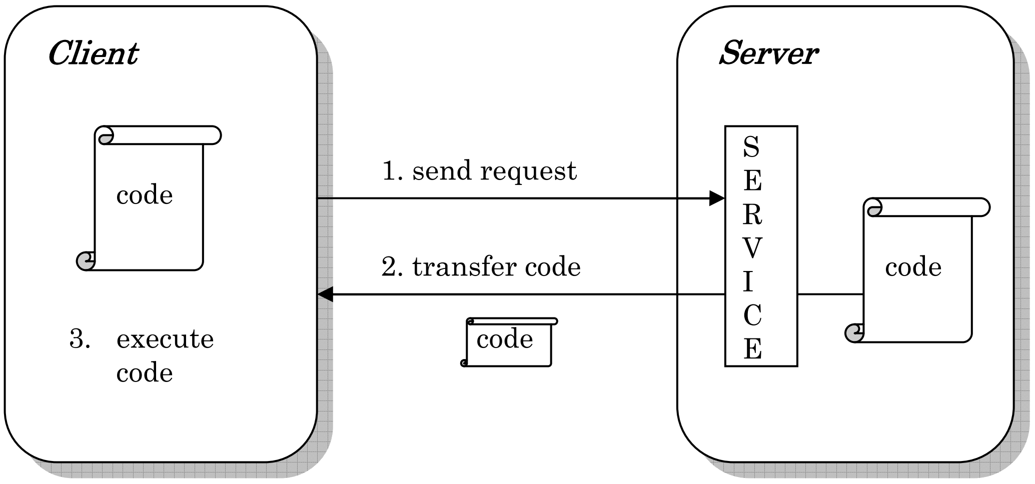


Figure 2-2. Code-on-Demand Paradigm.

2.1.3 Remote Evaluation Paradigm

In the remote evaluation paradigm, see Figure 2-3, a client has the know-how (code) necessary to perform the service and a remote server owns the resource. To perform the task the client sends the service know-how to the remote site. When the server receives the service know-how, it will execute the code using the resources available there. After the execution, the server will return the result back to the client. A typical example is SQL. The client sends SQL query to the DB server, and then the server executes the query and returns data to the client.

- Data → static
- Code→ mobile
- State→ static

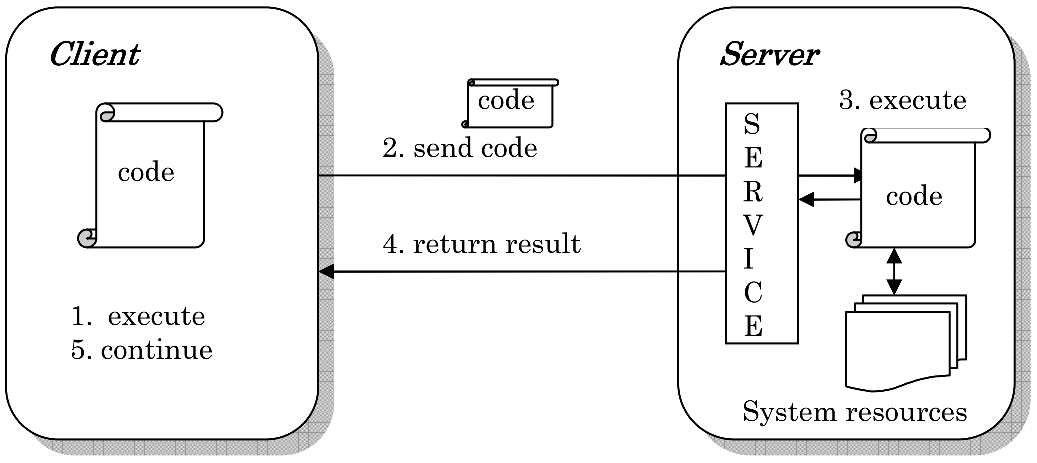


Figure 2-3. Remote evaluation Paradigm.

2.1.4 Mobile Agent Paradigm.

A key characteristic of the mobile agent paradigm, see Figure 2-4, is that any host in the network is allowed a high degree of flexibility to possess any mixture of know-how, resources, and processors. Its processing capabilities can be combined with local resources. Know-how expressed in the form of mobile agents is not limited in a single host but can execute freely at any host in the net work.

- Data → static/mobile
- Code→ mobile
- State→ static/mobile

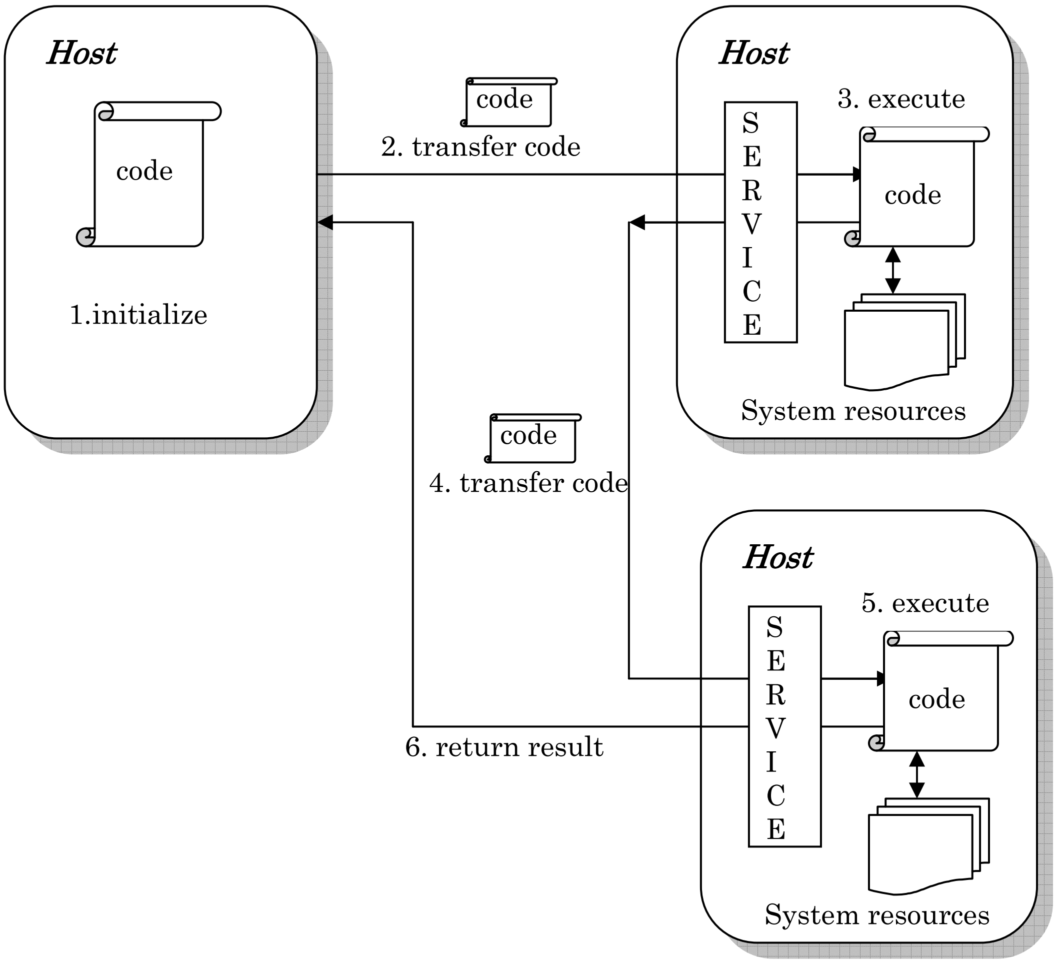


Figure 2-4. Mobile Agent Paradigm.

In these four distributed computing paradigms, *code-on-demand*, *remote revaluation* and *mobile agents* are considered as mobile code systems because the code on one host is transferred to another. Beside the code, the execution state of the program may also be moved to the other hosts, which divides the mobile code system into the strong mobility and the weak mobility.

Strong mobility enables an executing unit to move as a whole by retaining its execution state (e.g., the instruction pointer) across migration. Migration is transparent, in that the executing unit resumes execution on the new host right after the instruction that triggered the migration.

Weak mobility enables the transfer of application code towards or from a different host. At the destination, the code may be run into a newly created executing unit or it may be linked into an already running one.

2.2 Advantages of Mobile Code System

Mobile code represents a new way of building distributed software systems. Motivation for adopting the mobile code paradigm has been surveyed in great detail in [23, 29,58and 65]. Here we list several representative examples.

Real-time interaction with remote resources: Most computing resources in the host, such as databases, file systems or even physical displays, are not allowed to be transported. For a computation that requires real time interaction with these resources, it must be conducted in the exact site which the resources reside. Code mobility provides the possibility to prescribe the location of computation, so as to bring real-time interaction into reality. For example, active contents like Java applets prescribe interactive presentation that is to be rendered on the browser side.

Reduction of communication traffic: Mobile computers usually interact with servers through unreliable, low-bandwidth, high-latency, high-cost networks. Mobile programs become an attractive alternative because network traffic can be reduced by migrating the client program to the server side, thus avoiding the potential cross-network communication bottlenecks.

Customization and extension of server capabilities: In traditional client-server applications, valuable hardware resources are usually managed by server software (e.g., an operating system). The server offers a predefined set of services which are defined based on extremely general access policies and tends to ignore the specific needs of individual clients. It is very difficult to extend the capability of the serve without redefining its interface. Remote evaluation offers a flexible infrastructure for extensible server. Recently, various proposals have been made to allow application-specific extension code to be downloaded dynamically into server software, so as to customize the access policies to meet the specific needs of clients.

Asynchronous distribution computing: In traditional client-server applications, the state of computation is distributed among servers and clients. As a consequence, it is difficult to maintain the consistency of the distributed states and articulate the correctness of the computation. Mobile code systems localize computation states in a single process. They offer a better abstraction that makes the crafting of distributed software a more manageable task.

2.3 Applications of Mobile Code System

Any application that can be crafted under the mobile code paradigm can also be structured as a client-server application [58]. However mobile code systems offer many software engineering advantages that their client-server counterpart lacks such as those mentioned above. Thus mobile code systems can be applied to the following application domains.

2.3.1 Distributed Information Retrieval

Distributed information retrieval applications collect information from the certain resources scattered in the network. The information matches some specified criteria. During the information retrieval process, the information sources visited by the applicants can be defined statically or determined dynamically. This is a domain encompassing a big diversity of applications. For example, the wide range of information to be retrieved can be the list of all the publications of a given author to the software configuration of hosts in a network. The efficiency could be improved by code mobility because the code can migrate from remote to close to the information when performing the search process.

2.3.2 Active Documents

Traditional passive data, such as e-mail or web pages, is enhanced by active documents applications with the capability of executing the programs which have certain relationship to the content of the document, enabling enhanced presentation and interaction. Mobile code system is the premise for realizing these applications because it allows the embedding of code, stating the code into documents, and executing the dynamic contents during document fruition. A typical instance is represented by an application that uses graphic forms to compose and submit queries to a remote database. The interaction with the user is modeled by using the COD paradigm, i.e., the user first raises requests for the active document component to the server and then using the document as an interface to perform the computation. This type of application can be easily implemented by a technology which can fetch remote code fragments. A typical choice is a combination of WWW technology and Java applets.

2.3.3 Advanced Telecommunication Services

Support, management, and accounting of advanced telecommunication services, such as video conference, video on demand or tele-meeting, require a specialized "middleware" providing mechanisms for the dynamic reconfiguration and the user customization—advantages brought with code mobility. For example, in a tele-conference, the application components managing the setup, signaling, and presentation services could be dispatched to the users by a service broker. Examples of approaches exploiting code mobility can be found in [57and 74]. A special class of advanced telecommunications services supports mobile users. In this special circumstance, the autonomous components can provide support for disconnected operations, as discussed in [80].

2.3.4 Remote Device Control and Configuration

Remote device control applications are focusing on configuring a network of devices and monitoring their status. Several other applications are included in this domain, i.e., industrial process control and network management. Traditionally, monitoring is performed by randomly or periodically picking up the resource states while configuration is conducted by a predetermined set of services. This approach, based on the client-server paradigm, can bring a number of problems. Mobile code could be used in this incident to design and implement monitoring components that shared with the devices being monitored and report events that represent the evolution of the device state. Additionally, the management components migration to remote sites could improve both performance and flexibility.

2.3.5 Workflow Management and Cooperation

In a business or engineering process, workflow management applications support the cooperation of persons and tools involved. The workflow defines which activities must be carried out to accomplish a given task as well as how, where, when and at what distance these activities should involve each party. To represent activities as autonomous entities is a practice to this approach. During their evolution, they are circulated among the entities involved in the workflow. Mobile code could be used to provide support for mobility of activities that encapsulate their definition and state. For example, a mobile component could encapsulate a text document that undergoes several revisions. The component maintains information about the document state, the legal operations on its contents, and the next scheduled step in the revision process. An application of these concepts can be found in [22].

2.3.6 Active Networks

The concept of active networks is recently be proposed by several articles, which acts as a means to introduce flexibility into networks and provide more powerful mechanisms to weave or systemize the elements in the network according to applications' needs. They can be classified in tiers delimited by two extremes represented by the programmable switch and the capsule approaches. The *programmable switch* approach basically is an example of the COD paradigm, and it aims at providing dynamic extensibility of network devices through dynamic linking of code. On the other hand, the *capsule* approach aims to attach to every packet flowing in the network, some codes describing a computation that must be performed on packet data, at each node. Clearly, active networks aim at leveraging the advantages provided by mobile code in terms of deployment and maintenance, customization of services, and protocol encapsulation. As an example, in this scenario a multi protocol router could be downloaded on demand of the code needed to handle a packet corresponding to an unknown protocol, or even receive the protocol together with the packet.

2.3.7 Electronic Commerce

Electronic commerce applications make it possible to perform business transactions through the internet. A transaction may involve negotiation with remote entities and may require access to information that is continuously evolving, e.g., stock exchange quotations. In this context, there is the need to customize the behavior of the parties involved in order to match a particular negotiation protocol. Moreover, it is desirable to move application components close to the information relevant to the transaction. This makes mobile code appealing for this kind of applications. The term “mobile agent” is often related with electronic commerce. Another application of code mobility to electronic commerce can be found in [83].

2.4 Security of Mobile Code System

By its very nature, mobile code is fraught with inherent security risks. With the emergence of various forms of malicious active contents, users of mobile code systems are now aware of the increasingly serious security threats associated with mobile code computation. A malicious or faulty mobile code unit may tamper valuable data on local disks, covertly transmit sensitive information to another party, or masquerade as another trusted application.

Mobile code units may originate from unfamiliar sources, making it difficult for users to determine if a given code unit should be granted certain execution rights. The host user never writes them, nor does he know a lot about them, and sometime he does not know where they came from. Anonymity is a central reality of mobile code computing. A naive response will consider all mobile code as malicious and reject them or forbid all mobile code capabilities. Though that policy can give the host the maximum security, it is the most useless method because of the fact that there are many benefits of mobile code system and increasingly software infrastructures are built around mobile code technologies. The question is not to avoid downloading or using, but to protect the host from the downloaded mobile code running wild. Thus our objective is to verify the Java mobile code precisely as much as possible, that is, to let the mobile code causing no security problems (intentionally or involuntarily) pass our verification as many as possible.

2.4.1 Security Requirements

There are two classes of security issues in mobile code systems: *Host Security* and *Code Security*. The host security is concerned with the protection of the host from being attacked by malicious or faulty mobile programs, and with the avoidance of mutual interference among execution units. While the code security is concerned with the assurance of correctness and confidentiality for the computation that is delegated to a remote host. When an untrusted host carries out a computation on behalf of a client, the host may maliciously corrupt or expose the internal state of the client's execution units.

This dissertation is mainly devoted to the exploration of issues concerning the host security. There are three aspects concerned with the host security:

- **Integrity**: System resources should be protected from unauthorized modification, deletion, or other means of tampering.
- **Confidentiality**: Sensitive information should be protected from leaking to unauthorized parities through some channels.
- **Availability**: The services of computing system should be protected from monopolizing or denial.

In order to establish and evaluate the security of a computing system, one should refine the criteria above, and lay out exactly what the security requirements are in concrete terms. In general, the following attacks threat the host security in mobile code systems.

- **Denial of service:** The downloaded mobile program may monopolize shared the resources like the terminal screen, CPU time, threading services, etc. Such attacks destroy the availability of the host system.
- **Corruption:** Some malicious or faulty mobile code may modify or erase important data. Other may tamper with the internal state of the system, rendering the system state incoherent. Such attacks compromise the integrity of the system.
- **Leakage:** Some mobile codes may actively release sensitive information on an outside party. Other may engage in data processing activities from which malicious third parties can infer information that is supposed to be classified. Such attacks are direct violation of the system's confidentiality.

- **Masquerading:** Some malicious mobile programs may masquerade as another one by faking the UI of the latter, thus fooling the users into entrusting them with critical resources and data. Others may pretend to originate from a trusted origin. And malicious mobile programs may even fool the type system by appearing to be of another type, thus gaining access to the internal state of the system. Masquerading is a very subtle form of attack that could potentially lead to the compromising of all the three aspects of host security.

2.4.2 Evaluation Criteria of Protection Mechanisms

Protection mechanisms are technologies built into the computing environment for the sake of enforcing security policies. Protection is based on the notion of *separation*. Separation can be physical (allocating physically distinct resources to competing parties), temporal (scheduling competing processes to execute at a different time), logical (creating logical barrier to avoid interference), or cryptographic (encrypting sensitive information).

To design secure protection mechanisms, there are several principles can be referred [106 and 107].

- **Economy of mechanisms:** The design of the protection mechanism should be small and simple. A small and simple mechanism can be carefully analyzed and validated.
- **Fail-safe default:** The default condition should be denial of access. The designer of a protection mechanism should determine what is accessible instead of when access is denied.
- **Complete mediation:** The protection mechanism should be designed so that all possible access to system resources is covered. In a system that will be used continuously, and in which access rights may be revoked, every access attempt should be checked.
- **Open design:** The security of the protected system should not depend on keeping the design of the protection mechanism secret.
- **Separation of privilege:** Access on an object should depend on more that one condition. In this way, complete security breach will not occur when one protection system is defeated.

- **Least privilege:** The mobile code should be granted the bare minimum amount privilege necessary to complete the job.
- **Psychological acceptability:** If the users feel that protecting their system resources is too much work, they will not use it. The human interface should be designed for naturalness, ease of use, and simplicity, so that users will routinely and automatically apply the protection mechanisms.

2.4.3 Conventional Protection Techniques

In traditional operating systems, besides operating protection techniques such as CPU protection, *Memory Protection* and *Access Control* are other two protection mechanisms relevant to mobile code systems.

- Memory Protection

The purpose of memory protection is to prevent the malfunctioning of one execution unit from interfering other execution units or even the host. With the memory protection, the execution units are restricted and not able to interfere either with the execution states of other units or with the state of the global host.

There are three mechanisms in total that provide the memory protection in the traditional operating systems:

1. Processes are isolated in separate address spaces. No matter it is a data reference or a control transfer, the hardware will check the every address reference at run-time to see if the address space of the running process includes the location of the address reference. Or a memory exception will be generated to halt the process and return control to the operating system.

2. There are two types of executions provided by CPU which are named as the kernel mode and the user mode. Instructions that set the boundary of address space are protected and can only be executed in the kernel mode. User processes are then forbidden to redefining the boundary of their address spaces.

3. The kernel mode of operating system checks the control flowing outside of the

address space by a special interface which is usually achieved by providing a set of predefined system calls accessible by a special TRAP instruction. When a system call is invoked by outside execution, the CPU switches to kernel mode, and control is transferred to the operating system correspondingly, then the operating system starts to process the system call on behalf of the user process.

Furthermore, some operating systems provide complex mode to separate different security-levels by the group of concentric rings instead of a simple dichotomy of kernel and user modes. Usually it needs special hardware and operating system support. In the concentric ring mode, only the code running in the rings with higher trust level are allowed to access data and code in the rings with lower trust level, in another word it is a one-way flow. Accordingly the control will be hand over to the code in the ring of higher trust level via special entry points called gates.

- Access Control

Access control is achieved by protected information resources which identify the special execution units that can be granted the access to certain resources. In traditional operating systems, resources of the host are modeled as objects, while user processes are modeled as subjects. With permission, a subject can perform certain operations on an object. The permission of this kind of performance is called access right. Security policies are expressed as an assignment of rights to subjects. A protection domain is a collection of access rights. A user process acquires its access rights by being associated to a protection domain.

A matrix is introduced to describe the access rights in a multiprogramming system. In the matrix, protection domains are expressed in rows while objects are expressed in columns. Access rights are given to a protection domain (row) for the accessing of an object (column). It is easy to define the access right by labeling an entry in the matrix with access operations. In traditional operating systems, the access matrixes are usually implemented either of two ways. The access control list is one of the two ways. It is a list of <subject, right> pairs associated with every system resource. When an access occurs to a resource, the associated list will be checked to see if the accessing subject is in the list and access right is appropriately granted. Another is called capability. A capability is an unforgettable pointer to a system resource. The right was granted to a subject to access an object at the moment of received the pointer. In a sense, capability controls access through visibility which means it is impossible for a process to access a system resource if it is not even visible to the process.

In traditional operating systems, access control can be described in two related mechanisms which are *Authentication* and *Authorization*. Authentication is the process of establishing the identity of a user. Authorization is granting the right access to authenticated users according to the result of authentication. Under such a system, it is the user's identity that largely determines the right to perform an operation, or, be more precisely, it depends on the operating system's knowledge upon the user in a large extend.

2.4.4 Distinctiveness

Mobile code systems share many similarities with the traditional operating systems. In fact the security issues in them are all related to the multiprogramming, specifically resource sharing. But the mobile code is different from other multiprogramming languages used in traditional operating systems. The protection mechanisms in these traditional systems can not be directly transplanted to mobile code systems to address the similar security needs. Several distinguishing features of mobile code make the security needs different from those of traditional multiprogramming operating systems.

- **Layered Protection**

Traditional discretionary access control [108] relies on trusted resources which means a user should be a known party. The access control is based on the trust to the origination of the codes including the user's identity and the ownership of the resources. A straight simple implementation of this idea to mobile code security is to label every mobile code unit with a digital signature that indicating its origin. In this view, the idea of traditional operating system security is extended to mobile execution, and the access authorization is only issued to those codes whose origin is well-known to the host. This approach works well when the mobile program is developed by a famous brand name, or when it is sent from a credible source. However, the approach is dysfunctional when the foreign code is written by an author unknown to the host or comes from an uncertain origin. The key conception of the Internet computing is that any party can freely share information or actively contact with others who have access to the Internet. It is foreseeable that in the future more and more useful mobile programs are going to be developed and distributed by parties unknown to the average users. Security solely based on identity cannot afford to handle such complex demanding. This difficulty was first articulated by Ousterhout et al [94], and then found its full expression in a paper of Chess [28].

Based on the above understanding, the identity or the origin of the information should not restrict security engineers to authorize the access. No matter what the programs are anonymous or not, if they are trustworthy, they should be accepted by a sound security infrastructure. Therefore, it is accepted as an axiom the origin of the program should not hamper it from being download to a computing environment. Thus anonymous trust is the first fundamental challenge in mobile code security.

- **Layered Protection**

Another fundamental aspect of mobile code system is that a mobile code system creates a complete multiprogramming environment above the existing operating system. In the environment, mobile code is able to define its own computing model, provide its own set of services, maintain its own resources and hence define its own security model. As a result, it is not usually realistic to simply treat an execution unit as same as just another normal process in the operating system, running in just another protection domain. Furthermore, our desire for platform independence will conflict with any approach designed particularly to the security model of an operating system. On the other hand, as one of the users in the underlying platform, a mobile code computing environment may expose some of the operating system resources to the visiting execution units. The mobile code security model must comply with the security constraints imposed by the operating system.

In traditional operating systems, a process defines both the boundary for memory protection and the protection domain for access control. A mobile code system usually occupies only one process, which in turn hosts secondary threads representing execution units. In order to make it possible to protect the computing environment process from the execution units and to protect the execution units from interfering each other, we should set up two protection mechanisms. One is a memory protection to define secondary address spaces inside the address space of the computing environment process. And the other is the access control mechanism to define secondary protection domains inside the protection domain of the computing environment process. Thus a parent-child relationship is formed between the security model of the operation system and the security model of the mobile code system.

Layered protection is a characteristic feature in single-address-space operating systems like OPAL [27] and Mungi [62], also extensible operating systems like SPIN [17], VINO [111], and Exokernel [42]. In such kind of operating systems, untrusted code may be (dynamically) introduced into a privileged protection domain (e.g., the kernel) in order to prevent these units from exploiting the resources into that domain. Some recent works [30] focus on the Operating System community and endeavor to address the need for intra-address-space protection mechanisms motivated by software plug-ins, device drivers and data-driven security threats.

- **Implicit Acquisition**

Different from the traditional slow, manual, explicit software acquisition, the code mobility defines a new model of software acquisition. In the past, system administrators know exactly what package are installed on the system and announce any potential impact to users since all alternatives are reviewed and tested. In a mobile code system, software acquisition is completely different. A mobile code unit may arrive without the user's acknowledgment. Simple activities such as opening an email or browsing a webpage could invoke the installation of active mobile code unit. Acquisition is therefore implicit, which is also a design goal. In such an environment, only automatable checks are allowed such as signature checking, program analysis, type-checking and so on. All such checking should take only limited time to complete. It is this time constraint acquisition process established trust gradually. With the time constraint a computing environment has to establish the trustworthiness of a mobile program without going through the traditional evaluation cycle. In fact the time to establish the trust should be only a small part of the total execution time of the mobile program. Implicit acquisition is the third fundamental challenge of the mobile code security.

2.5 Protection Mechanisms for Mobile Code Systems

Discretion, verification, transformation, and arbitration are four kinds of approaches in mobile code systems protection. Most current protection mechanisms of existing mobile code systems can be considered as the combination of the four approaches.

2.5.1 Discretion

Discretion refers to the protection mechanisms which make security decisions based on identifying the “tokens” of trust. In particular, it turns to various authentication techniques [82] for help to establish the trust. Every mobile code unit is associated with certain digital signature(s). Once the host received a foreign mobile code unit the digital signature of the mobile code will be authenticated, and a (mechanical) process of authorization will authorize access privileges to the mobile code unit according to the result of authentication. The signature authentication in such kind of systems is assumed to be highly efficient. Discretion-based protection addresses the challenge of implicit acquisition pretty well because the signature authentication inherited in it is simple and the efficient. As a result, it has been studied as a general protection infrastructure [47and 63] and has been utilized in quite a few existing mobile code systems [54].

The core of discretion approach is the semantics of the signature. Eventually it is the meaning of a signature that determines which level of access rights is granted.

A digital signature is an unforgettable token that can denote the security property of the signed code unit. There are three potential denotations can be attached to the signatures of mobile code units.

Identity/Origin Semantics: This method is a direct translation of the traditional discretionary access control found in many operating systems. The signature of a mobile code unit discloses its origin or author. The computing environment keeps a record of the connection between known signatures and their relevant rights. The performance of the schemes which based on recognizing the owners or authors of programs is not satisfied in establishing the anonymous trust in mobile code systems.

Authoritative Endorsement Semantics: Giving a signature to a mobile code unit means that the signing party endorses the unit as being “safe”, normally it is in an informal sense. Certain trusted authorities will be responsible for certifying mobile code units in this approach. Developers submit their mobile programs to the trustworthy certification authorities to get the signature before the publication. Usually, what it means to be “safe” is informally defined by the signature, if it is properly defined at all. By this approach, a signature can only provide endorsement of the mobile code unit, but the endorsement has no formal semantics, which means it cannot be reduced to formally defined security properties. Because the endorsement is based on trust, therefore the security provided by it largely depends on the extent of trustworthy on the signing party.

Program-Analytic Semantics: The signature denotes a formal program-analytic property such as type safety or invariance of a particular assertion (program invariant). Only when the corresponding formal property can be found in the unit, signature is attached to the mobile code unit. There are three conditions that will result the attachment of the signature:

Code is trusted if it is generated by a trustworthy compiler [89 and 101].
Code is trusted if it has been properly rewritten by a trustworthy program transformer [17 and 111].
Code is trusted if it has been certified by a trustworthy program analyzer.

Compare to informal endorsement, a program-analytic semantics can be more reliable, because it builds the trust on a formally defined, publicly available program certifying algorithm instead of merely by human judgment. Unfortunately, currently, there are only small numbers of security properties have been processed by formalization. Memory safety and confidentiality are the rare cases that have been formalized into program-analytic terms. To further explore the space of application of this approach, studies are being carried on to translate more security properties into program-analytic terms.

2.5.2 Verification

In the verification approach to mobile code security, security policies are formulated as program analytic properties. Before landing the computation environment, in coming mobile code units must pass through a trusted program analyzer, usually named as a verifier whose job is to deny potentially unsafe programs from the various incoming units. Therefore the execution units that pass the analysis and reach the computation environment are guaranteed to satisfy certain security properties.

- **Verification for Memory Protection**

The application of the verification approach for memory protection is currently the most successful model. Following three examples give detail illustration.

First is Typed Intermediate Language. By the using a safe intermediate language memory protection is achieved in Java programs [55]. Java source programs are compiled into the format of Java Virtual Machine (JVM) bytecode [75]. The bytecode format is specially designed to protect execution units from interfering with each other and prevent them to access the JVM's internal state. Firstly, the JVM bytecode language is strictly typed. Secondly, pointer arithmetic is not allowed in the bytecode. Therefore, only in a type-safe manner could bytecode instructions access the memory. As a consequence, memory protection can be simplified into type-checking. All Java class files must be screened by a bytecode verifier before dynamically connecting to the JVM. Because the JVM bytecode is unstructured, data-flow analysis has to be introduced in to ensure that the type safety of the class file. In fact, dataflow analysis within the JVM also can be carried to check for other safety concerns such as operand stack overflow as well as to check for type safety. Therefore, runtime checks that would otherwise be needed to avoid operand stack overflow and ensure type safety can be safely avoided.

Second is Typed Assembly Language. While Java has to rely on an intermediate language in order to check the type information, and Necula and Lee have to resort to a highly expressive logical proof to capture similar information for machine code, Morrisett et al [52,84 and 85] demonstrated that type checking actually can be performed in an assembly language. Especially, it has been demonstrated by a typed assembly language (TAL) [84] which carries the type in formation of a rich, functional source language (a call-by-value variant of System F, the polymorphic λ-calculus augmented with products and recursion on terms). There are three important conclusions of this remarkable work. Firstly, it demonstrates that type safety can be achieved without using an abstract intermediate language, thus the run-time performance will be significantly reduced. In fact, type check of typed assembly code can be fully performed without referring to the original source program. Secondly, the typing construct imposes almost no restrictions on optimization, which makes it possible to exclude the safety property of the program from the code compiler. Thirdly, there is an effective type-preserving procedure that can interpreter the source language into TAL. Compared with this work, the approach of Necula and Lee [87] is more general and the verification is incomplete.

In summary, Java bytecode can be taken as a portable intermediate representation which allows attachment of type annotation in order to enforce memory protection statically. When it is applied solely to memory protection, proof-carrying code can use a very expressive logic to capture typing information for a target language, Therefore it can provide static typing without using an interpretive intermediate language. Last but not the least, static typing can be performed in a target language instead of resorting to an overly expressive formalism, which has been actually demonstrated in TAL.

- **Verification for Confidentiality**

Program-analytic approaches to the enforcement of confidentiality have received a lot of attention, and are relatively well-understood. Building on Bell and La Padula's security model [13 and 69], the work of Dorothy Denning [38, 39 and 40] has laid the foundation for the study of *Secure Information Flow* analysis.

The information flow model can be defined by

$$FM = \langle N, P, SC, \oplus, \rightarrow \rangle \qquad (2\text{-}1)$$

In the above model N is a set of logical storage objects or information receptacles. Elements of N may be files, or program variables. P is a set of process. SC is a set of security classes corresponding to disjoint classes of information. The class-combining operator “$\oplus$” is an associative and commutative binary operator. A flow “$\rightarrow$” relation is defined on pairs of security classes. For classes A and B, $A \rightarrow B$ means if and only if information in class A is permitted to flow into class B [15].

The security requirement of the model is that a flow model FM is secure if and only if execution of a sequence of operations cannot violate the relation “$\rightarrow$”. To comply with this policy, information at a given security-level is not allowed to flow to lower levels. A security system is composed of a set S of subjects and a disjoint set O of objects. Each subject $s \in S$ is associated with a fixed security class C(s), denoting it clearance. Likewise, each object $o \in O$ is associated with a fixed security class C(o), denoting its classification level. The security classes are partially ordered by a relation $\leq$, which forms a lattice. To avoid subjects with low clearance accessing sensitive data and subjects with high clearance to release sensitive data to low-clearance subjects, we need that a subject may only read objects with classification level no higher than its clearance, but may only write to objects with classification level no lower than its clearance. Information is always flowing unidirectionally from low classification source to high classification destination.

Information flow could be *explicit* or *implicit*. Given two variables X and Y, the information flow from Y to X is explicit in the following command:

$X := Y + 2;$

In that command the variable X gets the information of the data stored in the variable Y directly. Such information flow is called *explicit information flow*. Therefore the classification level of the data in variable X should be the classification level of the data in variable Y.

Information flow could also be implicit. Conditional statements may convert information into control flow just like the following commands:

```
if $Y > Z$ then
    $X := 0$;
else
    $X := 1$;
end if
```

In that conditional statement, the value of the data in the variable *X* depends on the values of the data in the variable *Y* and *Z*. Thus the variable *X* gets information from the data in the variable *Y* and *Z* indirectly and such information flow is called *implicit information flow*. Likewise, looping constructs can also cause implicit information flow.

```
$X :=0$;
while $Y < 10$ do
    $Y := Y + 1$;
    $X := X + 2$;
end while;
```

In the conditional statements above, the variable *X* gets information from the data in more than one variable. In such cases, the classification level of the data in the and thus the classification level of the data in variable *X* should be the one of the data in the variable *X* should be the *Least Upper Bounder* (LUB) of classification levels of the data in variables from which the variable *X* gets information. Assuming the classification levels of the data in the variable *Y* and *Z* are L_y and L_z, the classification level L_x of the data in the variable *X* should be $L_x = L_y \vee L_z$, where $\vee$ denotes the calculation of LUB.

To deal with explicit information flow, each expression is associated with a secure flow type, which represents the classification level of the data item. The lattice structure of the classification levels induces a natural sub typing relationship among the secure flow types: if type τ represents a classification level at least as high as that of type τ' then $\tau \geq \tau'$. An expression involving operands with distinct security types receives the least upper bound of the operands' types as its type. For example, if e and e' have security types τ and τ' respectively, and $\tau \leq \tau'$, then e + e' can be assigned security type τ'.Each variable also has a type τ var, indicating that it holds contents with type no higher than τ. Explicit leaking is then prevented by requiring that assignment of the form X := a is well-typed only if X has type τ var and a has type no higher than τ. To formally express this, we allow expression type τ to be coerced to any type τ' if τ ≤ τ', and then require that X := a is well-typed if and only if X has type τ var and a has type τ. With this arrangement, the above example code that explicitly leaks information will not be well-typed.

To handle implicit information flow, every command is associated with a type τ com. Intuitively, a command has type τ com if every variable that is being assigned in the command has type τ' var where τ ≤ τ'. That is, τ is a lower bound for the security-levels of the variables being assigned in the command. The idea is that if a conditional or iterative construct involves a condition expression of type τ then commands in the body should not assign to variables with security-levels lower than τ. To make this work, we need two more sub typing rules. For the variables, τ var ≤ τ' var if and only if τ ≤ τ'. For the commands, the opposite must hold: τ com ≤ τ' com if and only if τ' ≤ τ. Again, expressions can be freely coerced to their super types.

The verification can be done statically or dynamically. Static verification approaches analyze a program prior to the execution and judge whether the program is secure or not, while the dynamic ones implement the verification of the program during the run-time. The static approaches cannot achieve satisfying verification precision in implicit information transferring because of the inherent limitation of static verification approaches 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. While since the dynamic approaches implement the verification during the run-time, they can get better verification precision and trace the information flow in exception handling. The disadvantage of dynamic approaches is that they cost more run-time overhead than the static ones.

2.5.3 Transformation

Sometimes a mobile code representation may not be well tailored for execution although it is good for transportation (e.g., platform independent, compact for transport efficiency). In many mobile code systems, code units are transported in the byte-code form of virtual machine. The bytecode then is transformed into a native code for efficient execution just after it arrived to a host. Now such a just-in-time (JIT) compilation [120] becomes an important feature of mobile code systems like Java [55] and Omniware [84]. The Link-time code generation also adds portability to the mobile code systems [85]. Yet, dynamic code generation can also be considered as a protection mechanism. Mobile code units are expressed in a high level representation (e.g., a type-safe intermediate language as in Java) in which unsafe behavior cannot be expressed. While arriving at the host, the code units are converted to a format which can be executed on the host machine directly. Because the code generation is completed by a trusted compiler located on the host, and the unsafe behaviors cannot be expressed in the source code, the generated code can be considered as safe.

Transformation can also be used to tailor an untrusted code into a more secure form in a similar way. In contrast to the dynamic code generation, unsafe behaviors can be expressed in the migrated code. The code unit is statically analyzed while arriving at the host, and extra protection code is injected at program points where the security cannot be guaranteed.

- Transformation for Memory Protection

It was at early 1970's, the method of code rewriting has been applied to memory protection within a single address space [115]. Recently, the Omniware mobile code system [79] starts to use transformation to implement memory protection for untrusted mobile code units. Omniware mobile code units are transported as bytecode on the Omniware Virtual Machine (OmniVM) [128]. OmniVM is designed to resemble an RISC architecture, thus it provides efficient performance, simple implementation, and retarget ability. OmniVM divides its address space into segments, in order to ensure that execution units can only access those segments which they have been authorization to assess. Software-based Fault Isolation (SFI) is introduced in [83]. The basic idea of SFI is to rewrite untrusted mobile code units thus to turn it into versions cancel the access to unauthorized segments. Each memory address is divided into two parts, namely, a segment identifier and an offset within the segment. There are two possible rewriting rules can be formulated as below:

Segment Matching: For every memory reference, guard code is inserted before the reference has been initiated by the instructions. Initiatively the inserted code checks whether the referred segment matches the current segment. A memory fault will be raised if it failed in the check.

Sandboxing: For every memory reference, the segment identifier of the target address is dynamically overwritten by the identifier of the current segment.

The systematic application of either rule to every memory reference in a program guarantees that no interference occurs between disjoint segments.

Experience indicates that observable run-time overhead is caused by this approach because additional code is introduced by the transformation. Despite this overhead, native code which is executed in this way can run at a speed comparable to the speed of original code execution [83], though not as sane efficient as a the proof-carrying code version [88].

In extensible operating systems VINO [111] and Exo kernel [129], users are allowed to dynamically download untrusted extension code into the kernel address space to modify the behavior of the operating systems. Untrusted extension code units are subject to SFI transformation before downloading to protect the integration of the kernel address space.

2.5.4 Arbitration

Another way to completely protect a host is to cut the “direct” contact between the host and untrusted execution units. Once an untrusted execution unit requests the execution of an operation, the arbitrator, as a trusted party is called in to carry out the operation of the execution unit. Unsafe operations can be fully blocked by the arbitrator which can restrict the kind of operations visible to the execution unit, and can examine the client’s run-time state. The cost of such flexibility is usually a considerable run-time overhead.

Arbitration can be used to enforce both memory protection and access control. An interpreter is often used to enforce the memory protection. An interposition is frequently used to enforce the access control. Each of them will be examined in turn.

- Memory Protection by Interpreter

Using an interpreter to conduct computation in a safe and portable way has become very popular. Mobile code languages like Java [55], Safe Tcl [94], Scheme48 [100], and Telescript [62], JavaScript, all include the interpretation of some source or intermediate languages. The mechanism of interpreter approach to achieve memory protection can be explained in two ways:

Restricting expressiveness: A safe intermediate representation can be defined for mobile code units. With the limitation of the language, some unsafe operations cannot be expressed, while some can be statically checked. Take the JVM bytecode representation [75] as an example, in which privileged native instructions cannot be expressed; no pointer arithmetic; the language is strictly typed; interactions with host resources are performed through a public application programming interface (API). Therefore, memory interference can be avoided.

Dynamic checking: The interpreter can screen out all potentially dangerous moves by run-time checking because only through the arbitration of the interpreter could the execution unit interact with the host CPU. For an example, the JVM checks against null pointer dereferencing, out-of-bound array access, and illegal type-cast [75].

- Access Control by Interposition

Interposition means to insert trusted arbitration code in the form of a reference monitor [95] between a protected service and the entry point of the service. In a traditional operating system settings and processes usually access system resources via an on-bypass system which is called interface. Any attempts to access the protected resources are therefore subjected to the monitoring of the trusted arbitration code before they can reach the target services. Access control policies can be programmed into the arbitration code by which inappropriate access to the service can be screened out with flexibility. There are several implementations of interposition in mobile code systems: application wrappers, reference monitors, reference monitor in lining, and name resolution control.

Application wrappers. Application wrappers are software containers which are designed for controlling the interactions between untrusted programs and their execution environments. It was designed to retrofit arbitration code into a legacy software system in a non-intrusive manner.

Janus [111] is an application wrapper especially customized for protecting a host against insecure mobile code computing environments. The reflection of the design of Janus is that an untrusted process is not able to harm the host if its restriction of access to the underlying operating system has been placed appropriately. By using the process tracing facilities and the proc virtual file system in Solaris, Janus creates a user-level sandbox that put all system calls made by an untrusted process under the monitoring. Since legacy computing environments which have unreliable protection mechanisms (e.g., an old version of ghost view or a buggy, therefore Java-enabled web browser) can be executed inside the Janus sandbox, the Janus monitor can effectively block out unsafe system access initiated by the execution units running inside the legacy computing environment. Users may even supply their own policy module to specify which system calls should be allowed, which ones should be denied. A function must be called to determine what to do in deferent conditions.

Janus can provide effective protection to the host from any unreliable computing environment without requiring modification to the kernel and the computing environment. It is a good example to provide a practical solution to a very practical problem. However, even disregarding its platform-dependent nature, Janus can hardly address the layered protection problem. Firstly, Janus does not allow the computing environment to define a different protection domain for each execution unit. Secondly, the kind of security policy expressed by Janus is limited because it ignores the semantics of the computing environment. For an instance, when a JVM is running inside a Janus sandbox, the policy modules of Janus is not able to figure out the internal state of the JVM, and have to make the decision of their access control without understanding the state of JVM. In a word, layered protection can only be adequately addressed when interposing is a built-in feature of the computing environment instead of being a retrofitted patch of the operating system.

Reference Monitors. The security manager and stack inspection are the two mechanisms composed the Java reference monitor. All accesses to operating system services are isolated in the standard Java API. Whenever a service routine is invoked, the API transfers control to a corresponding monitor method of the global security manager object. The monitor method will inspect the Java run-time stack thus to conclude if the call is safe or not. If the monitor method does not allow the access, either an exception will then be created, or control is returned to the service routine to execute the original request. The security authority may oversee these monitor methods of the security manager class in order to customize the security policy of the JVM.

The Java security model allows one to define intricate security policies. For example, stack inspection allows the security manager use the micro control to decide what access level will be granted to the requestor. There are several drawbacks of this approach listed as follows. Firstly, the security manager needs to implement complex stack inspection logic to differentiate among accesses initiated by different execution units. From a software engineering point of view, both the construction and maintenance of this logic are difficult and fallible. Secondly, a procedure based definition of security policy is not easy to be understood. A popular solution is to introduce traditional access control lists in the arbitration code (asin Java [46] and Agent Tcl [56]). Subsequently, Netscape has attempted to extend the Java stack inspection mechanism by providing stack annotation which simplifies the logic for access right checking [123]. This extended version of stack inspection is later on proven by Wallach, Appel and Felten [121 and 122] to be equivalent to formal deduction in ABLP logic [1].

Reference Monitor in Lining. Code rewriting can be applied at load time to introduce monitoring code into an untrusted program. Here, the arbitration code does not reside at the entry points of privileged services, but instead is injected into the program itself to detect and avoid misuse of privileged services. Specifically, this strategy has been used for implementing the Java stack inspection [43and 122]. SFI has also been applied to enforce security policies expressed as security automata [44]. Besides a number of other efforts are involved in applying load time code rewriting to enforce high level access control policies [46, 102, 103 and125].

Name Resolution Control. In this approach, arbitration occurs while dynamic linking happens. The name resolution provides a relative simple way to offer the potential of centralizing all security logic into a single mechanism.

Safe-Tcl [94] is a security-aware extension of the popular Tcl scripting language [100]. Protection is achieved by three mechanisms — safe interpreters, aliases, and hidden commands. Similar to other shell scripting languages, Tcl is a command-based language which means the access to operating system facilities are provided through a set of commands. Safe-Tcl defines a padded cell security model, in which each individual execution unit is executed by its own interpreter. All system services are available in a trusted, master interpreter. When an untrusted script is executed, it is sandboxed in a separate, untrusted, safe interpreter. Who acts just as a separate name space. Privileged commands can be embedded in the safe interpreter in order to prevent untrusted script from unauthorized access to system resources. Additionally, to achieve the finer-grained control, a command may be aliased. Such as the name of a privileged command in the safe interpreter maybe "overshadowed" by a trusted arbitration routine in the master interpreter. If the access is granted the arbitration routine decides at run-time. If the access is permitted, it delegates the original call to the overshadowed command in the master interpreter.

The padded cell model refers to a form of interposition called name resolution control. In this approach the mechanism of name resolution is to control the selective access to the privileged services. In essence, name resolution control includes two component mechanisms. Firstly, granting of capabilities is realized by name visibility control. The notion of a safe interpreter, which is essentially a namespace, coincides with that of a protection domain. A privileged service can be accessed only if it can be named in the safe interpreter. It is easy for one to define a different access policy for each script because each script is assigned to separate name space and the name can be encrypted as well. Secondly, message interception selectively binds names of privileged services to wrapper code that protects the entry points of those services. Here, accessibility is not controlled by visibility, but instead by dynamic checking of the possession of rights.

Scheme [94 and 100] is another early mobile code system that set up its primary protection mechanism based on the approach of name resolution control. In Scheme, a procedure is considered to be a function closure, which contains a lambda expression and a binding environment. When a procedure is triggered, the only visible objects inside the lambda expression are the actual arguments and the values of the names in the lexical environment. Scheme48 allows programs to construct arbitrary binding environments, thus to execute untrusted code inside these carefully-crafted special environment. During the course of constructing such environments, the names of privileged procedures can be encrypted or be renamed to be invisible to arbitration routines.

Wallach et al [97] describe a way to implement name resolution control in the context of Java. In Java, a name space coincides with a class loader. A class name in one class loader represents a different class than another class with the same name in a different class loader. The class loader was originally conceived for name space partitioning so that there will be no name conflict among separate execution units. Taking advantage of this design, one may create a subclass of the standard class loader class, in which all requests for name resolution are monitored. As a result, if a privileged name is to be hidden, the class loader can throw an exception when the name is resolved. Aliasing can be simulated by resolving the names of privileged classes to arbitration classes.

The extensible operating system SPIN [17] also models protection domains by name spaces. All extension code in SPIN is written in the type-safe language Modula 3. Capabilities are directly modeled as pointers. Therefore, if a name is well-typed in a code unit, then the resource or service it refers to will be accessible. Typing thus provides a means of expressing conditional visibility of a symbol. Fine-grained protection is achieved by allowing users to manipulate name spaces. Name spaces can be created dynamically, and code units are executed within the confine of that name space, thus restricting its capabilities. An interesting feature is that name spaces can be extended by the Combine operation, which creates a union of two name spaces. In general, a system that uses name resolution control for protection needs ways to construct and extend name spaces.

Besides the advantage of implementing name space in modeling protection domains, there are still some potential problems within this approach. One of them is that there is no way of revoking capability. The J-Kernel [101] is a Java security kernel that provides a capability revocation mechanism within a name-space domain framework.

3 Java Virtual Machine and Bytecode

3.1 Java Language

Platform independence, security, and network-mobility are three facets of Java's architecture that work together to make Java fit for the emerging distributed computing environment of mobile code systems. Among these three aspects the network-mobility of the code and objects is more important compared with the other two. The same code can be sent to all the computers and devices interlinked together in the network. Objects can be exchanged among the various components of a distributed system which can be executed on different kinds of hardware. The built-in security framework of Java also helps to make the software network-mobility more practical. By reducing the risks, the trust in a new paradigm of network-mobile software is build up with the help of the security framework.

A single Java program can run on various computers and devices without being changed to adapt itself to the running environment. Compared with the programs compiled specially for some certain hardware or an operating system, it is much easier and cheaper to develop, administrate and maintain the platform independence Java programs.

Networks provide a venue for malicious programmers to leak or tamper information, destroy computing resources, or simply do something annoying. Virus producers, for example, may place malicious piece of wares on the network which can be downloaded by unsuspecting users. Java addresses the security challenge by providing an environment in which programs downloaded across a network can be run with security in customizable degrees.

Robustness of simple program is one of the security aspects. Just like devious code written by malicious programmers, buggy code written by well-meaning programmers also can bring troubles such as potentially destroying information, monopolizing compute cycles, or causing systems to crash. Java's architecture guarantees a certain level of program robustness by preventing certain types of pernicious bugs, such as memory corruption, from ever occurring in Java programs. That guarantees that mobile code will not inadvertently crash.

By enabling the transmission of binary code in small pieces across networks, Java takes advantage of distribution computing. Compared with other programs that are not network-mobile, the special feature of Java program makes it easier and cheaper to be delivered.

The emerging of mobile code provides another opportunity that both code and state can transmits across the network with the mobile objects. Java achieved object mobility in its APIs for object serialization and RMI (Remote Method Invocation). Based on Java's underlying architecture, the object serialization and RMI together provide an infrastructure that allows the objects to be shared by various components of distributed systems. The network-mobility of objects makes new models possible for distributed systems programming, therefore the benefits of object-oriented programming are effectively brought to the network.

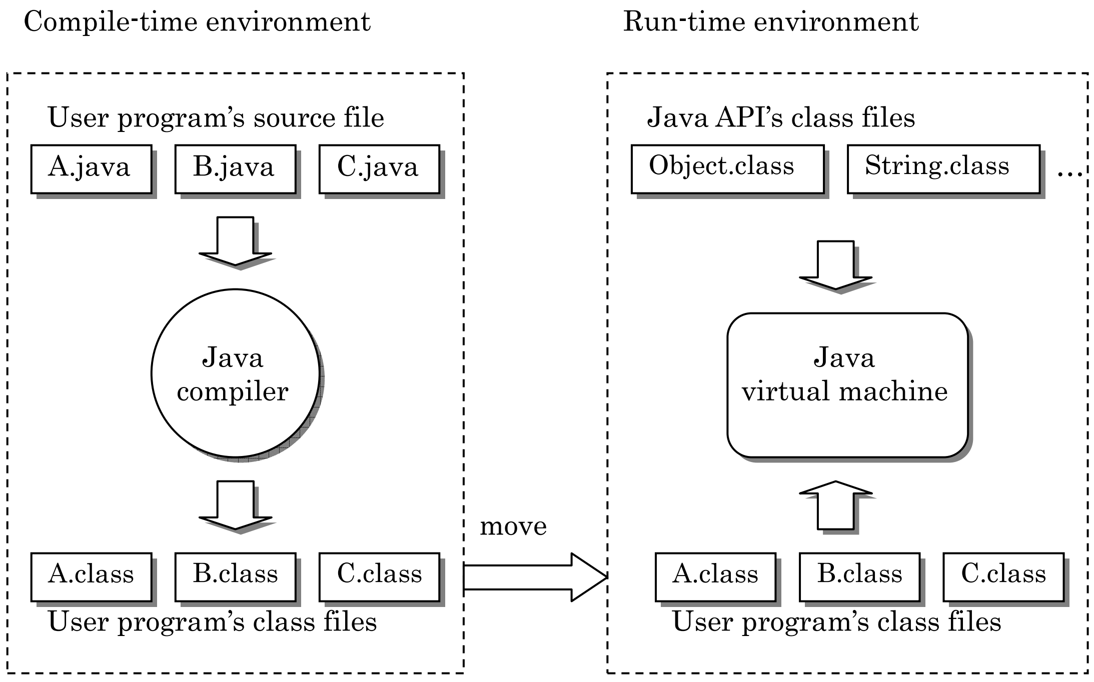


Figure 3-1. The Java programming environment.

Figure 3-1 shows the relationship among various parts of Java programs. Java program source files written in the Java programming language are compiled into Java class files in the form of bytecode. Then those class files are loaded and executed in Java virtual machine (the local JVM or a remote JVM). During the execution, the Java bytecode accesses system resources (such as I/O) by calling methods in the classes implementing the Java Application Programming Interface (Java API).

A “platform” is formed by the Java virtual machine and Java API together, on which all the Java programs are compiled. More than to be called as the *Java runtime system*, the combination of the Java virtual machine and the Java APIs is also called as the *Java Platform* (or, starting with version 1.2, the *Java 2 Platform*). It is because the Java platform can be implemented in software that makes it possible for Java programs to run on many different kinds of computers.

3.2 Java Virtual Machine

The core of Java's network-orientation is the Java virtual machine. All the three features, platform independence, security, and network-mobility, of Java's network-oriented architecture are supported by JVM.

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. Its specification defines certain essential features that every Java virtual machine must have, while leaves many options to the designers of each implementation. For example, all Java virtual machines must be able to execute Java bytecode programs, while developers can choose any technique to make it happen. Further, the feature of flexibility of the Java virtual machine's specification enables it to be implemented on a wide variety of computers and devices.

A major job of Java virtual machine is to load class files and execute the bytecodes contained in those files. As shown in Figure 3-2, the Java virtual machine contains a class loader, which loads class files from both the user’s program and the Java API, and a execute engine, which actually executes the bytecode loaded by the class loader. Only those class files from the Java API that are actually needed by a running program are loaded into the virtual machine. The bytecodes are executed in an execution engine.

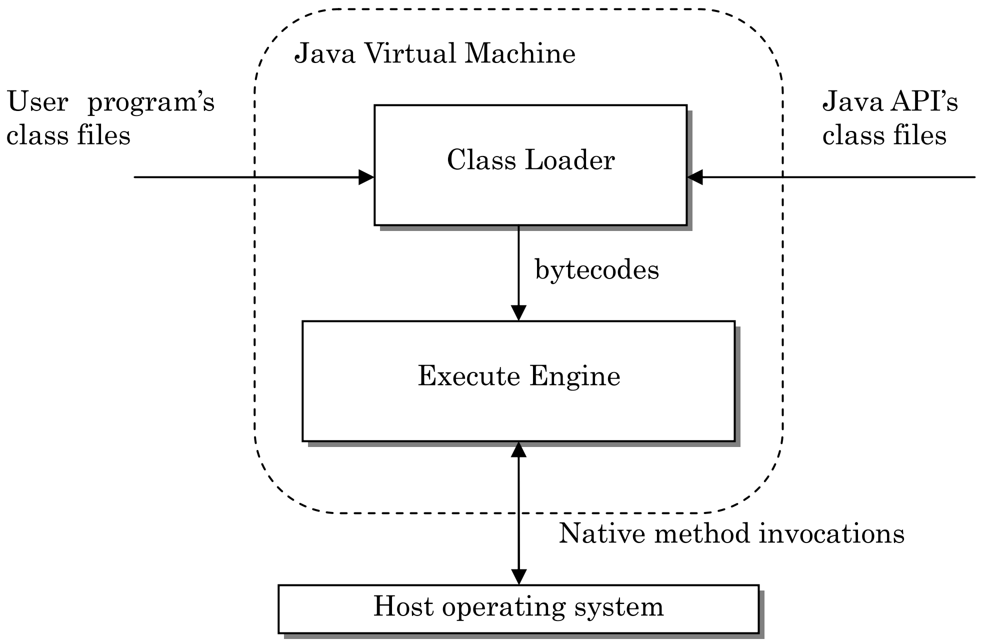


Figure 3-2. A basic block diagram of the Java virtual machine.

As a part of the virtual machine, the execution engine varies in different implementations. On a Java virtual machine implemented in software, the simplest kind of execution engine just interprets the bytecode once at a time. Just-in-time compiler is another kind of execution engine which is faster but requires more memory. In this scheme, the bytecode of a method are compiled to native machine code at the first call of the method. The native machine code for the method is then cached, and at the next time when the same method is invoked again it will be re-used. An adaptive optimizer is the third type of execution engine. By this approach, the virtual machine starts by interpreting bytecode, monitors the activity of the running program and identifies the most heavily used areas of the codes. Along with the running program, the virtual machine compiles to native and optimizes just these heavily used areas. The rest areas of the codes, which are not heavily used, remain as bytecode and still need to be interpreted by the virtual machine when be in use. This adaptive optimization approach enables a Java virtual machine to put typically 80 to 90% of its time at executing highly optimized native codes, while requiring it to compile and optimize only the 10 to 20% of the code that really matters to performance. Finally, in a Java virtual machine built on the top of a chip that executes Java bytecode natively, the execution engine is actually embedded in the chip.

All Java methods can be divided into two kinds: Java method and native method. A Java method is written in the Java language, compiled to bytecode, and stored in class files. A native method is written in other languages, such as C, C++, or assembly, and compiled to the native machine code of a particular processor. Java methods are platform independent, while native methods are stored in a dynamically linked library whose exact form is platform specific. During the execution of bytecode on a Java virtual machine that is implemented in software on the top of the host operating system, an interaction between the Java program and the host happens when Java program invokes the native methods. At that time the dynamic library that contains the native method will be loaded on the virtual machine and the native method then invoked. As it is shown in Figure 3-2, native methods are the connection between a Java program and an underlying host operating system.

3.3 Java Bytecode and Instruction Set

3.3.1 Bytecode

For analyzing bytecode program, we should understand the format of the program in the form of bytecode. Java programs consist of a set of classes. Each class is stored in one class file, which has the `ClassFile` structure as shown in Figure 3-3.

```
ClassFile {
    u4 magic;
    u2 minor_version;
    u2 major_version;
    u2 constant_pool_count;
    cp_info constant_pool[constant_pool_count-1];
    u2 access_flags;
    u2 this_class;
    u2 super_class;
    u2 interfaces_count;
    u2 interfaces[interfaces_count];
    u2 fields_count;
    field_info fields[fields_count];
    u2 methods_count;
    method_info methods[methods_count];
    u2 attributes_count;
    attribute_info attributes[attributes_count];
  }
```

Figure 3-3. The format of class file.

magic:

The `magic` item supplies the magic number identifying the class file format; it has the value 0xCAFEBABE.

minor_version, major_version:

The values of the `minor_version` and `major_version` items are the minor and major version numbers of this class file. Together, a major and a minor version number determine the version of the class file format. If a class file has major version number M and minor version number m, we denote the version of its class file format as M.m. Thus, class file format versions may be ordered lexicographically, for example, 1.5 < 2.0 < 2.1.

A Java virtual machine implementation can support a class file format of version v if and only if v lies in some contiguous range Mi.0 ≤v ≤Mj.m. Only Sun can specify what range of versions a Java virtual machine implementation conforming to a certain release level of the Java platform may support.

constant_pool_count:

The value of the `constant_pool_count` item is equal to the number of entries in the `constant_pool` table plus one. A `constant_pool` index is considered valid if it is greater than zero and less than `constant_pool_count`, with the exception for constants of type long and double.

constant_pool[]:

The constant_pool is a table of structures representing various string constants, class and interface names, field names, and other constants that are referred to within the ClassFile structure and its substructures. The format of each constant_pool table entry is indicated by its first "tag" byte.

The constant_pool table is indexed from 1 to constant_pool_count-1.

access_flags:

The value of the access_flags item is a mask of flags used to denote access permissions to and properties of this class or interface.

An interface is distinguished by its ACC_INTERFACE flag being set. If its ACC_INTERFACE flag is not set, this class file defines a class, not an interface.

If the ACC_INTERFACE flag of this class file is set, its ACC_ABSTRACT flag must also be set and its ACC_PUBLIC flag may be set. Such a class file may not have any of the other flags.

If the ACC_INTERFACE flag of this class file is not set, it may have any of the other flags. However, such a class file cannot have both its ACC_FINAL and ACC_ABSTRACT flags set.

The setting of the ACC_SUPER flag indicates which of two alternative semantics for its invokespecial instruction the Java virtual machine is to express; the ACC_SUPER flag exists for backward compatibility for code compiled by Sun's older compilers for the Java programming language. All new implementations of the Java virtual machine should implement the semantics for invokespecial documented in this specification. All new compilers to the instruction set of the Java virtual machine should set the ACC_SUPER flag. Sun's older compilers generated ClassFile flags with ACC_SUPER unset. Sun's older Java virtual machine implementations ignore the flag if it is set.

All bits of the access_flags item not assigned are reserved for future use. They should be set to zero in generated class files and should be ignored by Java virtual machine implementations.

this_class:

The value of the this_class item must be a valid index into the constant_pool table. The constant_pool entry at that index must be a CONSTANT_Class_info structure representing the class or interface defined by this class file.

super_class :

For a class, the value of the `super_class` item either must be zero or must be a valid index into the `constant_pool` table. If the value of the `super_class` item is nonzero, the `constant_pool` entry at that index must be a `CONSTANT_Class_info` structure representing the direct super class of the class defined by this class file. Neither the direct super class nor any of its super classes may be a final class.

If the value of the super_class item is zero, then this class file must represent the class Object, the only class or interface without a direct super class.

For an interface, the value of the super_class item must always be a valid index into the `constant_pool` table. The `constant_pool` entry at that index must be a `CONSTANT_Class_info` structure representing the class Object.

interfaces_count:

The value of the `interfaces_count` item gives the number of direct super interfaces of this class or interface type.

interfaces[]:

Each value in the interfaces array must be a valid index into the `constant_pool` table. The `constant_pool` entry at each value of interfaces[i], where $0 \leqslant i < \text{interfaces\_count}$, must be a `CONSTANT_Class_info` structure representing an interface that is a direct super interface of this class or interface type, in the left-to-right order given in the source for the type.

fields_count :

The value of the `fields_count` item gives the number of `field_info` structures in the fields table. The `field_info` structures represent all fields, both class variables and instance variables, declared by this class or interface type.

fields[] :

Each value in the fields table must be a `field_info` structure giving a complete description of a field in this class or interface. The fields table includes only those fields that are declared by this class or interface. It does not include items representing fields that are inherited from super classes or super interfaces.

methods_count:

The value of the `methods_count` item gives the number of `method_info` structures in the methods table.

methods[]:

Each value in the methods table must be a `method_info` structure giving a complete description of a method in this class or interface. If the method is not native or abstract, the Java virtual machine instructions implementing the method are also supplied.

The `method_info` structures represent all methods declared by this class or interface type, including instance methods, class (static) methods, instance initialization methods, and any class or interface initialization method. The methods table does not include items representing methods that are inherited from super classes or super interfaces.

attributes_count:

The value of the `attributes_count` item gives the number of attributes in the attributes table of this class.

attributes[]:

Each value of the attributes table must be an attribute structure.

The only attributes defined by the Java Virtual Machine specification as appearing in the attributes table of a `ClassFile` structure are the `SourceFile` attribute and the Deprecated attribute.

A Java virtual machine implementation is required to silently ignore any or all attributes in the attributes table of a `ClassFile` structure that it does not recognize. Attributes not defined in this specification are not allowed to affect the semantics of the class file, but only to provide additional descriptive information.

In Figure 3-4 and Figure 3-5, we give one example of the result of a Java program complied into bytecode and the definition of the class file.

Source code:

```
class Act {
   public static void doMathForever() {
            int i = 0;
            for (;;) {
                     i += 1;
                     i *= 2;
               }
         }
   }
```

$\Rightarrow$

Bytecode:

```
CA FE BA BE 00 03 00 2D 00 11 07 00 07
07 00 10 0A 00 02 00 04 0C 00 06 00 05 01
00 03 28 29 56 01 00 06 3C 69 6E 69 74 3E
01 00 03 41 63 74 01 00 08 41 63 74 2E 6A
61 76 61 01 00 04 43 6F 64 65 01 00 0D 43
6F 6E 73 74 61 6E 74 56 61 6C 75 65 01 00
0A 45 78 63 65 70 74 69 6F 6E 73 01 00 0F
4C 69 6E 65 4E 75 6D 62 65 72 54 61 62 6C
65 01 00 0E 4C 6F 63 61 6C 56 61 72 69 61
62 6C 65 73 01 00 0A 53 6F 75 72 63 65 46
69 6C 65 01 00 0D 64 6F 4D 61 74 68 46 6F
72 65 76 65 72 01 00 10 6A 61 76 61 2F 6C
61 6E 67 2F 4F 62 6A 65 63 74 00 20 00 01
00 02 00 00 00 00 00 02 00 09 00 0F 00 05
00 01 00 09 00 00 00 30 00 02 00 01 00 00
00 0C 03 3B 84 00 01 1A 05 68 3B A7 FF
F9 00 00 00 01 00 0C 00 00 00 12 00 04 00
00 00 05 00 02 00 07 00 05 00 08 00 09 00
06 00 00 00 06 00 05 00 01 00 09 00 00 00
1D 00 01 00 01 00 00 00 05 2A B7 00 03 B1
00 00 00 01 00 0C 00 00 00 06 00 01 00 00
00 02 00 01 00 0E 00 00 00 02 00 08
```

Figure 3-4. Java source code and bytecode.

```
hex bytes                 name
---------                 ----
CAFEBABE                  magic
0003                      minor_version
002D                      major_version
0011                      constant_pool_count
07                        tag
0007                      name_index
07                        tag
0010                      name_index
0A                        tag
0002                      class_index
0004                      name_and_type_index
0C                        tag
0006                      name_index
0005                      descriptor_index
01                        tag
0003                      length
282956                    "()V"     bytes[length]
01                        tag
0006                      length
3C696E69743E              "<init>" bytes[length]
01                        tag
0003                      length
416374                    "Act"              bytes[length]
01                        tag
000B                      length
736E697065742E6A617661  "Act.java"         bytes[length]
01                        tag
0004                      length
436F6465                  "Code"             bytes[length]
01                        tag
000D                      length
436F6E7374616E7456616C7565     "ConstantValue"          bytes[length]
01                        tag
```

```
000A                          length
457863657074696F6E73          "Exceptions"      bytes[length]
01                            tag
000F                          length
4C696E654E756D6265725461626C65      "LineNumberTable"    bytes[length]
01                            tag
000E                          length
4C6F63616C5661726961626C6573   "LocalVariables"           bytes[length]
01                            tag
000A                          length
536F7572636546696C65          "SourceFile"      bytes[length]
01                            tag
000D                          length
646F4D617468466F7265766572        "doMathForever"           bytes[length]
01                            tag
0010                          length
6A6176612F6C616E672F4F626A656374   "java/lang/Object"   bytes[length]
0020                          access_flags
0001                          this_class
0002                          super_class
0000                          interfaces_count
0000                          fields_count
0002                          methods_count
0009                          access_flags
000F                          name_index
0005                          descriptor_index
0001                          attributes_count
0009                          attribute_name_index
00000030                      length
0002                          max_stack
0001                          max_locals
0000000C                      code_length
033B8400011A05683BA7FFF9          code[code_length]
0000                          exception_table_length
0001                          attributes_count
```

000C	attribute_name_index	
00000012	attribute_length	
0004	line_number_table_length	
0000	start_pc	iconst_0, istore_0
0005	line_number	int i = 0;
0002	start_pc	iinc 0 1
0007	line_number	i += 1
0005	start_pc	iload_0, iconst_2, imul, istore_0
0008	line_number	i *= 2
0009	start_pc	goto 2
0006	line_number	while (true) {
0000	access_flags	
0006	name_index	
0005	descriptor_index	
0001	attributes_count	
0009	attribute_name_index	
0000001D	attribute_length	
0001	max_stack	
0001	max_locals	
00000005	code_length	
2AB70003B1	code[code_length]	
0000	exception_table_length	
0001	attributes_count	
000C	attribute_name_index	
00000006	attribute_length	
0001	line_number_table_length	
0000	start_pc	aload_0, invokespecial #3, return
0002	line_number	class Act {
0001	attributes_count	
000E	attribute_name_index	
00000002	attribute_length	
0008	sourcefile_index	

Figure 3-5. The definition of the class Act.

3.3.2 Instruction Set

A method's bytecode stream is a sequence of instructions for the Java virtual machine. Each instruction consists of a one-byte *opcode* followed by zero or more *operands*. The opcode indicates the operation to be performed. Operands supply extra information needed by the Java virtual machine to perform the operation specified by the opcode. The opcode itself indicates whether or not it is followed by operands, and the form the operands (if any) take. Many Java virtual machine instructions take no operands, and therefore consist only of an opcode. Depending upon the opcode, the virtual machine may refer to data stored in other areas in addition to (or instead of) operands that trail the opcode. When it executes an instruction, the virtual machine may use entries in the current constant pool, entries in the current frame's local variables, or values sitting on the top of the current frame's operand stack.

The JVM is a stack-oriented interpreter that creates a local stack frame of fixed size for every method invocation. The size of the local stack has to be computed by the compiler. Values may also be stored intermediately in a frame area containing *local variables* which can be used like a set of registers. These local variables are numbered from 0 to 65535, i.e. you have a maximum of 65536 of local variables. The stack frames of caller and callee method are overlapping, i.e. the caller pushes arguments onto the operand stack and the called method receives them in local variables.

The byte code instruction set currently consists of 204 instructions, 44 opcodes are marked as reserved and may be used for future extensions or intermediate optimizations within the Virtual Machine. The instruction set can be roughly grouped as follows:

- Stack operations: Constants can be pushed onto the stack either by loading them from the constant pool with the `ldc` instruction or with special "short-cut" instructions where the operand is encoded into the instructions, e.g. `iconst 0` or `bipush` (push byte value).

- Arithmetic operations: The instruction set of the Java Virtual Machine distinguishes its operand types using different instructions to operate on values of specific type. Arithmetic operations starting with `i`, for example, denote an integer operation and the instruction `iadd` adds two integers and pushes the result back on the stack. The Java types `boolean`, `byte`, `short`, and `char` are handled as integers by the JVM.

- Control flow: There are branch instructions like `goto` and `if icmpeq`, which compares two integers for equality. There is also a `jsr` (jump sub-routine) and `ret` pair of instructions that is used to implement the `finally` clause of `try-catch` blocks. Exceptions may be thrown with the `athrow` instruction. Branch targets are coded as offsets from the current byte code position, i.e. with an integer number.

- Load and store operations for local variables like `iload` and `istore`. There are also array operations like `iastore` which stores an integer value into an array.

- Field access: The value of an instance field may be retrieved with `getfield` and written with `putfield`. For static fields, there are `getstatic` and `putstatic` counterparts.

- Method invocation: Methods may either be called via static references with `invokestatic` or be bound virtually with the `invokevirtual` instruction. Super class methods and private methods are invoked with `invokespecial`.

- Object allocation: Class instances are allocated with the `new` instruction, arrays of basic type like `int[]` with `newarray`, arrays of references like `String[][]` with `anewarray` or `multianewarray`.

- Conversion and type checking: For stack operands of basic type there exist casting operations like `f2i` which converts a float value into an integer. The validity of a type cast may be checked with `checkcast` and the `instanceof` operator can be directly mapped to the equally named instruction.

Most instructions have a fixed length, but there are also some variable-length instructions: In particular, the `lookupswitch` and `tableswitch` instructions, which are used to implement `switch()` statements. Since the number of `case` clauses may vary, these instructions contain a variable number of statements.

αload x	Push the value with type α of the register x onto the operand stack
αstore x	Pop a value with a type α off the stack and store it to local register x.
αipush	Push a constant onto the operand stack
αconst d	Push constant d with type α onto the operand stack.
pop	Pop top operand stack element
dup	Duplicate top operand stack element
αop	Pop two operands with type α off the operand stack, perform the operation op ∈ {add, mult, compare .. }, and push the result onto the stack.
ifcond j	Pop 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.
getfield C.f	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 C.f	Pop a value k and a reference to an object of class C from the operand stack; set field f of the object to k.
invoke C.mt	Pop value k 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 k
αreturn	Pop the α 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 the stack

Figure 3-6. JVM Instructions set.

Figure 3-6 summarizes the instruction set of the Java virtual machine. A specific instruction, with type information, is built by replacing the α in the instruction template in the opcode column by the letter in the type column. For instance, *iload* represents loading an integer value, *aload* represents loading an object.

The abstract execution engine runs one instruction at a time during the execution of Java bytecode. This process takes place for each thread (execution engine instance) of the application running in the Java virtual machine. An execution engine fetches an opcode and, if that opcode has operands, fetches the operands. It executes the action requested by the opcode and its operands, and then fetches another opcode. Execution of bytecodes continues until a thread completes either by returning from its starting method or by not catching a thrown exception.

From time to time, the execution engine may encounter an instruction that requests a native method invocation. On such occasions, the execution engine will dutifully attempt to invoke that native method. When the native method returns (if it completes normally, not by throwing an exception), the execution engine will continue executing the next instruction in the bytecode stream.

One way to think of native methods, therefore, is as programmer-customized extensions to the Java virtual machine's instruction set. If an instruction requests an invocation of a native method, the execution engine invokes the native method. Running the native method is how the Java virtual machine executes the instruction. When the native method returns, the virtual machine moves on to the next instruction. If the native method completes abruptly (by throwing an exception), the virtual machine follows the same steps to handle the exception as it does when any instruction throws an exception.

A part of the job of executing an instruction is determining the next instruction to execute. An execution engine determines the next opcode to fetch in one of three ways. For many instructions, the next opcode to be executed directly follows the current opcode and its operands, if any, in the bytecode stream. For some instructions, such as `goto` or `return`, the execution engine determines the next opcode as part of its execution of the current instruction. If an instruction throws an exception, the execution engine determines the next opcode to fetch by searching for an appropriate catch clause.

Several instructions can throw exceptions. The instruction `athrow`, for example, throws an exception explicitly. This instruction is the compiled from of the throw statement in Java programming source code. Other instructions throw exceptions only when certain conditions are encountered. For example, if the Java virtual machine discovers that the program is attempting to perform an integer divide by zero, it will throw an `ArithmeticException`. This can occur while executing any of four instructions--`idiv`, `ldiv`, `irem`, and `lrem`--which perform divisions or calculate remainders on `int` or `long`.

Each type of opcode in the Java virtual machine's instruction set has a mnemonic. In the typical assembly language style, streams of Java bytecodes can be represented by their mnemonics followed by (optional) operand values.

Note that jump addresses are given as offsets from the beginning of the method. In Figure 3-5, the instruction `goto` causes the virtual machine to jump to the instruction at offset two (the instruction `iinc`). The actual operand in the stream is minus seven. To execute this instruction, the virtual machine adds the operand to the current contents of the pc register. The result is the address of the `iinc` instruction at offset two. To make the mnemonics easier to read, the operands for jump instructions are shown as if the addition has already taken place. Instead of saying "`goto` -7," the mnemonics say, "`goto` 2."

The central focus of the Java virtual machine's instruction set is the operand stack. Values are generally pushed onto the operand stack before they are used. Although the Java virtual machine has no registers for storing arbitrary values, each method has a set of local variables. The instruction set treats the local variables, in effect, as a set of registers that are referred to by indexes. Nevertheless, other than the instruction `iinc`, which increments a local variable directly, values stored in the local variables must be moved to the operand stack before being used.

For example, to divide one local variable by another, the virtual machine must push both onto the stack, perform the division, and then store the result back into the local variables. To move the value of an array element or object field into a local variable, the virtual machine must first push the value onto the stack, then store it into the local variable. To set an array element or object field to a value stored in a local variable, the virtual machine must follow the reverse procedure. First, it must push the value of the local variable onto the stack, then pop it off the stack and into the array element or object field on the heap.

4 Analysis of Information Flow in Bytecode

In this chapter, we will present the security model of protecting the host security in mobile code systems which is based on the secure information flow theory. Then we will analyze the information flow in Java bytecode and propose the mechanisms of tracing implicit information in Java bytecode.

4.1 Security Model

Different from traditional programs, the mobile code programs may move from host to host via network. In mobile code systems, the security approach should protect the host from malicious or defective mobile code programs. Since the low level security aspects like memory protection has been enforced well by the security characteristics of Java virtual machine, the work left is to protect the data *integrity* and *confidentiality*. In our research, we focus on the *confidentiality*.

Figure 4-1 shows the architecture of Java mobile code systems. When bytecode programs migrate from one remote host to the local system via the network, they are loaded to the local Java virtual machine. The JVM verifier checks whether the program is well typed to provide low level security. In such mobile code systems, the possible data-leaking process is shown in Figure 4-2 [77]. To detect such data leaking caused by mobile programs, some traditional protection-mechanisms used in operating systems could be utilized, such as *Authentication, Access Control* and *Secure Information Flow* theory. Authentication is usually implemented just after the mobile program arrives at the host (Step 1 in Figure 4-2). With some algorithm, the host can infer the mobile program's identity from the certain information carried by the mobile program. Depending on the mobile program's identity, the host judges whether the mobile program is secure or not. The Access Control is implemented when the mobile program tries to read some sensitive data from a host file (Step 2 in Figure 4-2). According to the mobile program's identity, the host grants certain read rights to the mobile program. The mobile program cannot get the data it is not authorized to access.

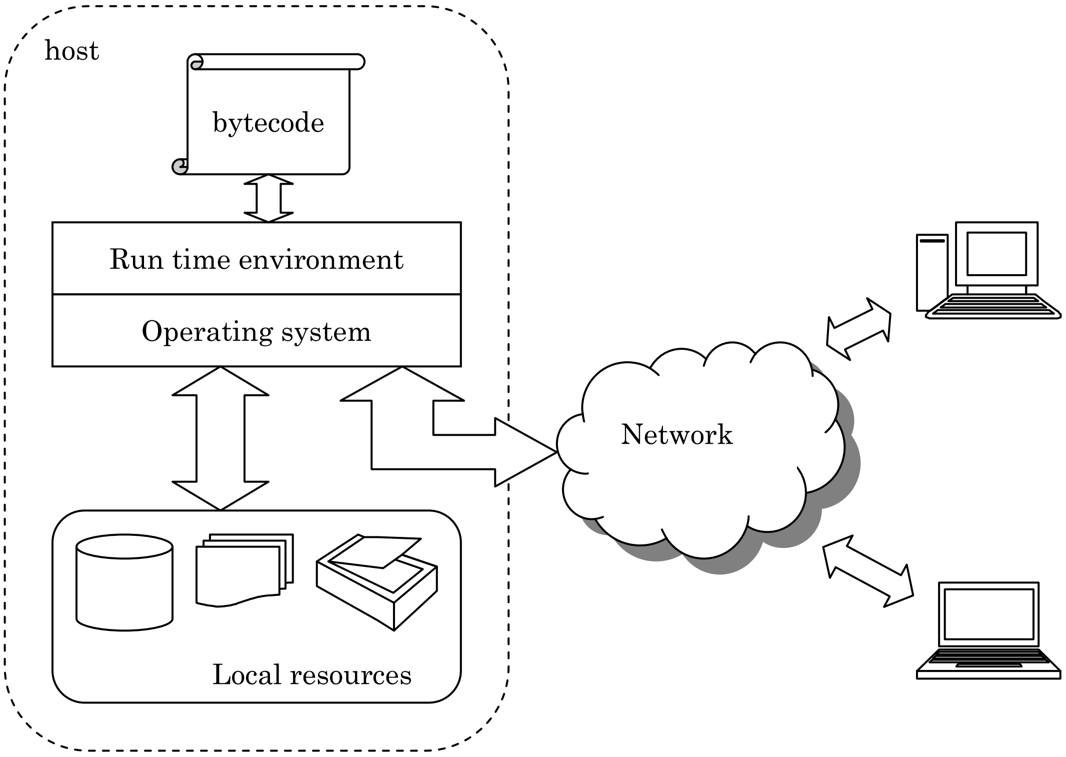


Figure 4-1. Framework of Java mobile code system.

The approaches based on authentication and access control confine the program from accessing the local system and network resources in order to ensure host's confidentiality. In other words, such approaches prevent data flowing from the local system or network resources to the program. While actually many mobile programs need the local information to perform their tasks. In those approaches, however, the programs cannot fulfill their tasks because of the access control mechanism, which makes it meaningless to download the mobile programs. And in fact it is not always true that the mobile program that gets the sensitive information of the local host will leak the information to some unauthorized parties. It is because the access control mechanism cannot trace and control the following propagation of the information that it denies the access to the sensitive information from unauthorized mobile programs. If a program gets sensitive data by some channel, the access control mechanism has no idea which party the information is transferred to, let alone controls the information transferring.

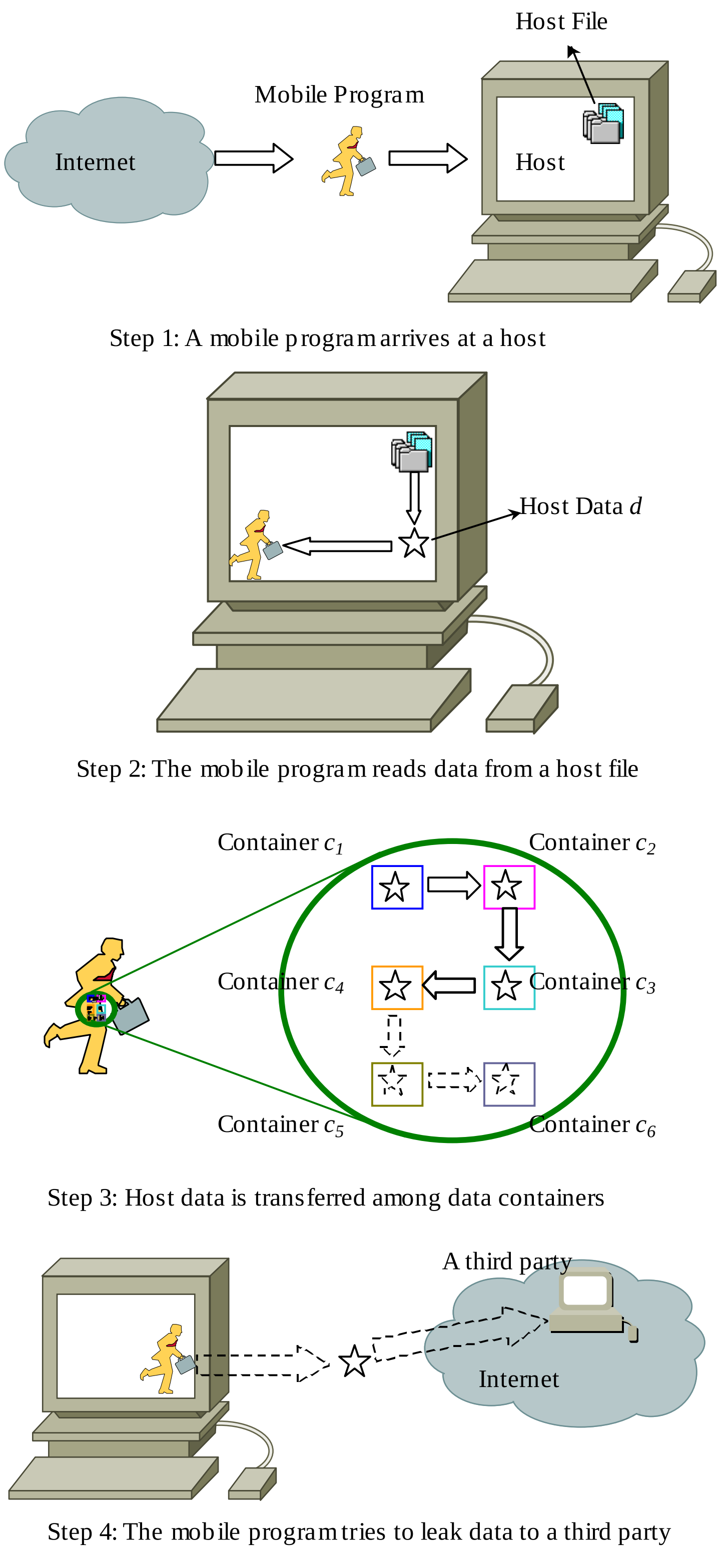

Host File
Mobile Program
Internet
Host
Step 1: A mobile program arrives at a host
Host Data d
Step 2: The mobile program reads data from a host file
Container c_1
Container c_2
Container c_4
Container c_3
Container c_5
Container c_6
Step 3: Host data is transferred among data containers
A third party
Internet
Step 4: The mobile program tries to leak data to a third party

Figure 4-2. The process of data-leaking in mobile code systems.

Obviously, a program-analytic semantics may be more reliable than such informal endorsement like authentication and access control. The *Secure Information Flow* theory in Denning [38, 39, and 40] is a kind of program-analytic mechanism and has been adopted in many approaches for mobile code systems security [11, 14, 15 and 16]. In order to make it as much as possible that mobile code programs can complete their functions without impairing the host security, we build the security model base on the secure information flow theory.

As mentioned in Chapter 2, in the secure information flow 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 it clearance. Likewise, each object $o \in O$ is associated with a fixed security class $C(o)$, denoting its classification level. The security classes are partially ordered by a relation $\leq$, which forms a lattice. To avoid subjects with low clearance accessing sensitive data and subjects with high clearance to release sensitive data to low-clearance subjects, we need that a subject may only read objects with classification level no higher than its clearance, but may only write to objects with classification level no lower than its clearance. Information is always flowing unidirectionally from low classification source to high classification destination.

In mobile code systems, the objects are same to those in traditional operation systems, which are the system resources on the host we need to protect (we refer to that host as the local-host). But the subjects in mobile code systems are different from the ones in traditional operation systems. In traditional operating system the subjects are the processes running on the local-host, while the subjects in mobile code systems are all the other hosts trying to get information from the local-host (we refer to them as observer-hosts). That difference makes that the approaches used in traditional operating systems cannot be adopted in mobile code systems without any ameliorating. The data-leaking in mobile code systems thus means that an observer-host gets some information on the local host it is not authorized. Therefore security classes denoting classification levels should be assigned to the local-host's files, and security classes denoting the clearance should be assigned to observer-hosts. As for the mobile code, it is just the intermediate transferring information between objects and subjects. It is not necessary to assign any security class to the mobile code or its information carriers.

Before the mobile program leaves the local-host or sends information out, the information being transferred in the mobile code is not leaked yet. It is when the mobile code tries to send information to observer-hosts that the data-leaking may be caused. Whether a data-leaking arises is not decided by that the mobile code gets some sensitive information but that it transfers the information to an unauthorized observer-host. It is not necessary to set any restriction or do any checking when information is being transferred in mobile code. 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 could understand the information's classification level and check whether the observer-host has the right to get the information.

The approaches in [11, 15, and 16] based on the secure information flow theory neglect the difference of security demand between mobile code systems and traditional operating systems. They treat the mobile code as the subject in the way that the processes are treated in traditional operating systems. In [11 and 15] the approaches assign security-levels (denoting the clearance) to information carriers (objects, method's parameters and return value, etc.) in the mobile code and the host rejects any mobile program causing an illicit information flow which means that information at a given security-level flows to lower levels. In [16] the authors adopt a security policy that grants access to private data based on the program's need and check if data with high security-level can ever propagate to observers with low security-level, that is, the approach makes the judgment at Step 3 in Figure 4-2. These approaches are more precise than the ones that only use authentication and access control. But they make two mistakes that they assign security-levels to the mobile code and consequently they detect data-leaking when the information is still being transferring in the mobile code. The two mistakes result in unnecessary restrictions in verification procedure and reduce the verification precision. Considering two data containers H with a higher security-level and L with a lower security-level, such statement

$H := L;$

$L := H$

will not do any harm to the host confidentiality. However the statement will be considered to cause an illicit information flow by the approaches in [11, 15, and 16]. The mobile programs that pass the verification of those approaches are only a part of all the mobile programs that will not do any harm to the local host security.

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

Security-level. In our approach, we refer to the security class denoting classification as *security-level*. The security-level indicates the host system resources' sensitivity. The higher the security-level is, the more sensitive the resource file is. All the information gotten from a resource has the same security-level as the resources. The system resources that should be protected include:

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

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. At present, the clearance-level is assigned to an observer-host according to its network address in our approach.

Distribution Map of Security-level. During the execution of the mobile program, the mobile program reads data from files of the local-host and transfers the data among its data containers. We maintain a distribution map to represent the security-levels of the local-host information in the mobile program's containers. When the execution starts, all elements in the distribution map have no value since there is no local-host information held by any container of the mobile code program. Each time the data in a container of the mobile code program changes, the corresponding element in the distribution map updates its value to the security-level of the new data.

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

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 a network link and 2) the mobile code moves to next destination We should compare the security-level of the information to be sent with the clearance-level of the observer-host to receive the information.

Definition 1. Let *DLC* be a data-leaking channel in one mobile code. Let *I* be the information to be sent at the *DLC* 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.

In summary, we assign security-levels to system resources of the local-host and clearance-levels to observer-hosts at first. Then during the local-host information is being transferred in the mobile code (Step 3 in Figure 4-2), we set no restriction to the information flow in the mobile code and just record the information flow it in the distribution map of security-level. When the mobile code tries to send information out (to an observer-host at Step 4 in Figure 4-2 or write to local-host files), we say that the execution encounters data-leaking channel and we check if a data-leaking is caused according to the record stored in the distribution map of security-level.

Our security model can be implemented by static approaches (as we have done in [78 and 77]) or dynamic approaches. In both kinds of approaches, to maintain a distribution map of security-level for recording the information flow is the core of the implementation. In the following section, we introduce the semantics rules used to update the distribution map of security-level according to the information flow.

4.2 Semantics Rules

In our security model, the key to verify the mobile code precisely is maintaining a correct distribution map of security-level during the execution of the mobile code, that is, tracing and recording the information flow in the mobile code correctly. In this section we will give the semantics rules that indicate the relationship between Java bytecode instructions and the change of the distribution map of security-level.

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 [75]. So the data container in bytecode could be the element of operand stack, the local register or the field of an object instance on the heap. We denote by *S* the aggregation of all security-levels of the data in operand stack. Similarly we denote by *R* the aggregation for registers and by *F* the aggregation for objects' fields. In this way the distribution map of security-levels could be represented by a tuple (*S*, *R*, *F*).

We denote *S* by the alphabet "*S*" followed by a sequence of numbers separated by the marker "·". The first number after "*S*" represents the security-level of the data in the top element on the stack, and the last one represents the security-level of the data in the bottom element on the stack. Given an index *j*, we denote by *R*(*j*) the security-level of the data in the local register with the index *j* (*j* should be less than the maximal number of registers). Given an object reference *oref* and a constant pool index *cpi*, we denote by *F*(*oref.cpi*) the security-level of the data held in the object's field resolved from the object reference *oref* plus the constant pool index *cpi* (the item at index *cpi* should be `CONSTANT_Fieldref` indicating the field). For a container of operand stack or a local register, if the data in the container is type *long* or *double*, *i* or *j* is the index of the first one of the two successive words used to store the data.

Furthermore we denote by $R(j \leftarrow l)$ the operation on *R* that updates the element's value at index *j* in *R* to *l* while keeps all other elements in *R* unchanged. Similarly we define $F(oref.cpi \leftarrow l)$ for *F*, too.

The change on the distribution map of security-levels results from the information flow among data containers of the mobile code, which could be divided into explicit flow and implicit flow as mentioned in Chapter 2. Even the same instructions will cause different change on the distribution map of security-level in explicit information flow and implicit information flow. Therefore we define the semantics rules for the explicit information flow and the implicit information flow respectively.

- **Semantics Rules for Explicit Information Flow**

The explicit information flow is quite simple and easy to trace. In one explicit information flow, the information in the used data is transferred to the defined data. Thus the security-level of the defined data should be assigned the security-level of the used data or, if the used data are more than one, the LUB of the security-levels of all the used data.

We list a subset of JVM bytecode instructions causing explicit information flow in Figure 4-3. Such instructions can cause explicit information flow among the registers, operand stack and class fields.

For example, consider the instruction `iload 4` which pushes the data in the local register R_4 at the index 4 to the top element S_0 of the operand stack. By executing the instruction, the data in S_0 gets the information of the data in R_4. We say that an explicit information flow is caused between the operand stack element S_0 and the local register R_4, and the security-level of the data in S_0 has the same value with security-level of the data in R_4. Consequently the element in the distribution map of security-level representing the security-level of the data in S_0 should be updated to the new value.

`pop`	Pop the top operand stack element.
α`op`	Pop two operands with type α off the operand stack, perform the operation $op \in$ { *add, cmpg, cmpl, div, mul, rem, sub*}, and push the result onto the stack.
α`const_c`	Push constant c with type α onto the operand stack.

α`load` x	Push the value with type α at the index x onto the operand stack
α`store` x	Pop a value with type α off the operand stack and store it into local variable at index x.
`getfield` x_1, x_2	Pop a reference to an object off the operand stack; fetch the value of the object's field resolved from the reference plus the constant pool item at $(x_1<<8)\| x_2$ and put it onto the operand stack.
`putfield` x_1, x_2	Pop a value and a reference to an object from the operand stack; store the value into the object's field resolved from the reference plus the constant pool item at $(x_1<<8)\| x_2$.

Figure 4-3. A subset of JVM instructions causing the explicit information flow.

`pop`	$(S(n)\cdot S(n\text{-}1)\cdot\ldots S(1)\cdot S(0)\cdot S, R, F) \rightarrow$ $(S(n)\cdot S(n\text{-}1)\cdot\ldots S(2)\cdot S(1)\cdot S, R, F)$
α`op`	$(S(n)\cdot S(n\text{-}1)\cdot\ldots S(1)\cdot S(0)\cdot S, R, F) \rightarrow$ $(S(n)\cdot S(n\text{-}1)\cdot\ldots S(2)\cdot (S(1)\vee S(0))\cdot S, R, F)$ $(\alpha\in\{d, f, i, l\}, op\in\{add, div, mul, rem, sub\})$
α`const_c`	$(S(n)\cdot S(n\text{-}1)\cdot\ldots S(1)\cdot S(0)\cdot S, R, F) \rightarrow$ $(S(n)\cdot S(n\text{-}1)\cdot\ldots S(0)\cdot 0\cdot S, R, F)$ $(\alpha\in\{d, f, i, l\})$
α`load` x	$(S(n)\cdot S(n\text{-}1)\cdot\ldots S(1)\cdot S(0)\cdot S, R, F) \rightarrow$ $(S(n)\cdot S(n\text{-}1)\cdot\ldots S(0)\cdot R(x)\cdot S, R, F)$ $(\alpha\in\{a, d, f, i, l\})$
α`store` x	$(S(n)\cdot S(n\text{-}1)\cdot\ldots S(1)\cdot S(0)\cdot S, R, F) \rightarrow$ $(S(n)\cdot S(n\text{-}1)\cdot\ldots S(1)\cdot S, R(x \leftarrow S(0)), F)$ $(\alpha\in\{a, d, f, i, l\})$
`getfield` x_1, x_2	$(S(n)\cdot S(n\text{-}1)\cdot\ldots S(1)\cdot S(0)\cdot S, R, F) \rightarrow$ $(S(n)\cdot S(n\text{-}1)\cdot\ldots S(0)\cdot F(oref.cpi)\cdot S, R, F)$ (*oref* is the reference to an object held by the top element on stack; $cpi = (x_1<<8)\| x_2$)
`putfield` x_1, x_2	$(S(n)\cdot S(n\text{-}1)\cdot\ldots S(1)\cdot S(0)\cdot S, R, F) \rightarrow$ $(S(n)\cdot S(n\text{-}1)\cdot\ldots S(3)\cdot S(2)\cdot S, R, F(oref.cpi \leftarrow S(0)))$ (*oref* is the reference to an object held by the second top element on stack; $cpi = (x_1<<8)\| x_2$)

Figure 4-4. Semantics rules for instructions in explicit information flow.

Denoting by $\vee$ the least upper bound (LUB) operation and considering the security-level of a constant is 0 (the lowest security-level), we define the semantics rules for Java bytecode instructions in an explicit information flow in Figure 4-4.

- **Semantics Rules for Implicit Information Flow**

The implicit information flow is much more complex than the explicit ones. We denote the data used as the condition of the implicit information flow by *conditional data*. In one implicit information flow, beside the information in the used data the defined data will also get the information in the conditional data of the implicit information flow. Therefore the security-level of the defined data should be assigned as the LUB of the security-levels of the used data and the conditional data of the implicit information flow.

We list a subset of the Java bytecode instructions that may cause implicit information flow in Figure 4-5. In Java bytecode, the family of *if*-instructions (e.g. `if_acmp<cond>`, `if<cond>` and `ifnull`) and the instructions of the *switch* statement (`tableswith` and `lookupswitch`) will generate conditional control transfer and thus cause implicit information flow. (The exception handling in Java bytecode may cause implicit information flow, too. We will discuss it later in Chapter 6.) The implicit information flow usually has two or more execution branches, some of which may be blank, that is, there is no instructions on the branch except instructions at the *fork* and *join* points. All the data that may be changed in the scope of the any branch will get the information of the conditional data of the implicit information flow additionally.

`if_acmp <cond> j`	Pop 2 values of type `ref` off the operand stack and compare them. Branch to offset *j* if the result of the comparison satisfies the condition `<cond>` $\in$ {eq, `ne`}.
`if_icmp <cond> j`	Pop 2 values of type `int` off the operand stack and compare them. Branch to offset *j* if the result of the comparison satisfies the condition `<cond>` $\in$ {eq, `ne`, `lt`, `le`, `gt`, `ge`}.
`if<cond> j`	Pop a value off the top of the operand stack, and compare it against zero. Branch to offset *j* if the result

	of the comparison satisfies the condition `<cond>` ∈ {eq, `ne`, `lt`, `le`, `gt`, `ge`}.
`ifnonull j`	Pop a value of type `ref` off the top of the operand stack. If the value is not null, branch to offset *j*.
`ifnull j`	Pop a value of type `ref` off the top of the operand stack. If the value is null, branch to offset *j*.
`lookupswitch`	Pop a value *key* of type `int` from the operand stack and compare *key* against the *match* values. If it is equal to one of them, a target address is calculated by adding the corresponding *offset* to the address of this `lookupswitch`. If the *key* does not match any of the *match* values, the target address is calculated by adding *default* to the address of this `lookupswitch`. Execution continues at the target address.

Figure 4-5. A subset of JVM instructions causing the implicit information flow.

For example, consider the following section of Java bytecode.

```
 0:iload_1
 1:iload_2
 2:if_icmple 9
 5:iload_4
 7:istore_3
 8:goto 12
11:iload_5
13:istore_3
14: … …
```

The section of Java bytecode compares the value of data in local register at index 1 and 2, and then stores the greater one into the local register at index 3. The instruction `if_icmple 9` at address 2 causes a conditional control transfer according to the result of comparing the data in local register at index 1 and 2, and thus it causes implicit information flow whose conditional data are the data in local register at index 1 and 2. In the branches of that implicit information flow, the data in the local register at index 3 may be changed and it gets information of the used data (the data in local register at index 4 or 5 depending on which branch is executed) and information of the conditional data (the data in local register at index 1 and 2). Therefore the security-level of the data in the local register at index 3 should be assigned to the LUB of the security-levels of the data in the local register at index 1, 2, 4 and 5. Here we define the *environment security-level* of one implicit information flow as the security-level of the conditional data or, if there are more than one conditional data, the LUB of the security-levels of all conditional data of the implicit information flow. Thus the security-level of the data changed in one implicit information flow should be assigned to the LUB of the conditional security-level of the implicit information flow and the security-level(s) of its used data.

Denoting by L_{env} the conditional security-level of implicit information flow, we could rewrite the rules in Figure 4-4 to define semantics rules for Java bytecode instructions in the implicit information flow as shown in Figure 4-6.

`pop`	$(S(n)\cdot S(n\text{-}1)\cdot\ldots S(1)\cdot S(0)\cdot S, R, F) \rightarrow$
	$(S(n)\cdot S(n\text{-}1)\cdot\ldots S(2)\cdot S(1)\cdot S, R, F)$
α`op`	$(S(n)\cdot S(n\text{-}1)\cdot\ldots S(1)\cdot S(0)\cdot S, R, F) \rightarrow$
	$(S(n)\cdot S(n\text{-}1)\cdot\ldots S(2)\cdot (S(1)\vee S(0)\vee L_{env})\cdot S, R, F)$
	$(\alpha\in\{d, f, i, l\}, op\in\{add, div, mul, rem, sub\})$
α`const_c`	$(S(n)\cdot S(n\text{-}1)\cdot\ldots S(1)\cdot S(0)\cdot S, R, F) \rightarrow$
	$(S(n)\cdot S(n\text{-}1)\cdot\ldots S(0)\cdot L_{env}\cdot S, R, F)$ $(\alpha\in\{d, f, i, l\})$
α`load` x	$(S(n)\cdot S(n\text{-}1)\cdot\ldots S(1)\cdot S(0)\cdot S, R, F) \rightarrow$
	$(S(n)\cdot S(n\text{-}1)\cdot\ldots S(0)\cdot (R(x) \vee L_{env})\cdot S, R, F)$ $(\alpha\in\{a, d, f, i, l\})$
α`store` x	$(S(n)\cdot S(n\text{-}1)\cdot\ldots S(1)\cdot S(0)\cdot S, R, F) \rightarrow$
	$(S(n)\cdot S(n\text{-}1)\cdot\ldots S(1)\cdot S, R(x \leftarrow (S(0) \vee L_{env})), F)$ $(\alpha\in\{a, d, f, i, l\})$
`getfield` x_1, x_2	$(S(n)\cdot S(n\text{-}1)\cdot\ldots S(1)\cdot S(0)\cdot S, R, F) \rightarrow$
	$(S(n)\cdot S(n\text{-}1)\cdot\ldots S(0)\cdot (F(oref.cpi) \vee L_{env})\cdot S, R, F)$
	(*oref* is the reference to an object held by the top element on stack; $cpi = (x_1{<<}8) \mid x_2$)
`putfield` x_1, x_2	$(S(n)\cdot S(n\text{-}1)\cdot\ldots S(1)\cdot S(0)\cdot S, R, F) \rightarrow$
	$(S(n)\cdot S(n\text{-}1)\cdot\ldots S(3)\cdot S(2)\cdot S, R, F(oref.cpi \leftarrow (S(0) \vee L_{env}))$
	(*oref* is the reference to an object held by the second top element on stack; $cpi = (x_1{<<}8) \mid x_2$)

Figure 4-6. Semantics rules for instructions in implicit information flow.

4.3 Implicit Information Flow Analysis

As we have analyzed above, the same Java bytecode instructions will cause different change on the distribution map of security-levels in explicit information flow and implicit information flow. Thus we need to divide the Java mobile code into explicit information transferring block and implicit information transferring block in order to update the distribution map of security-levels correctly and verify the mobile code precisely. Though the explicit information flow is quite simple, the scope of an explicit information transferring block cannot be located directly because only the instruction that is not in any implicit information transferring block may cause explicit information flow. In other words, locating the scopes of implicit information transferring blocks in Java mobile code is the precondition of locating the scopes of explicit ones. As soon as we get the location of implicit blocks, the problem of locating explicit blocks becomes quite simple. The scope of an explicit information transferring block is the instruction causing the explicit information flow, which is not in any implicit information transferring block. (In some meanings, we can consider the explicit information flow as one kind of special implicit information flow whose conditional security-level is the lowest one.) Thus we will analyze the implicit information flow in Java bytecode and give the algorithms to locate the scopes of the implicit information transferring blocks. As mentioned above, the implicit information flow could be cause by the family of *if*-instructions (e.g. `if_acmp<cond>`, `if<cond>` and `ifnull`) and the instructions of the *switch* statement (`tableswith` and `lookupswitch`) in Java bytecode. In the following we discuss those two cases respectively.

4.3.1 Implicit Information Flow Caused by *if*-instructions

The *if*-instructions in Java bytecode could be compiled from loop constructs (*for*, *while* and *do-while*) or *if-else* constructs in Java programming language. The *if-else* constructs in Java programming language is complied into bytecode straightly. Java *if-else* constructs could be complied into the bytecode formats shown in Figure 4-7 and Figure 4-8. The formats may have one or two non-blank branches, which depends on whether the construct has an *else* clause or not. While the case of compiling a loop construct is a little complex. A Java language conditional loop could be complied into two bytecode formats: the *if*-instruction is at the bottom of the loop or at the top of the loop as shown in Figure 4-9. In both formats of the bytecode, the conditional loop has only one non-blank branch composed by the instructions of the loop construct.

```
public void test
(int a, int b)
{
   if (a > b)
       int c = a;
   else
       int c = b;
}
```

$\Longrightarrow$

```
 0:iload_1
 1:iload_2
 2:if_icmple 8
 5:iload_1
 6:istore_3
 7:goto 12
10:iload_2
11:istore_3
12:return
```

$\Longrightarrow$

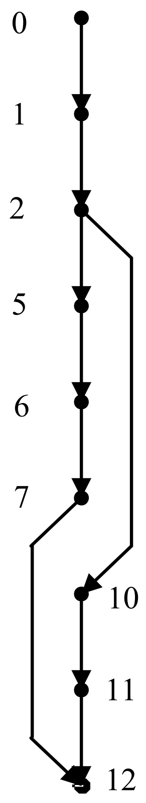


Figure 4-7. The bytecode and CFG of *if-else* construct with *else* clause.

```
public void test
(int a, int b)
{
    if (a > b)
       int c  = a;
}
```

$\Longrightarrow$

```
0:iload_1
1:iload_2
2:if_icmple 6
5:iload_1
6:istore_3
8: return
```

$\Longrightarrow$

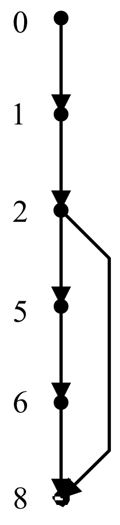


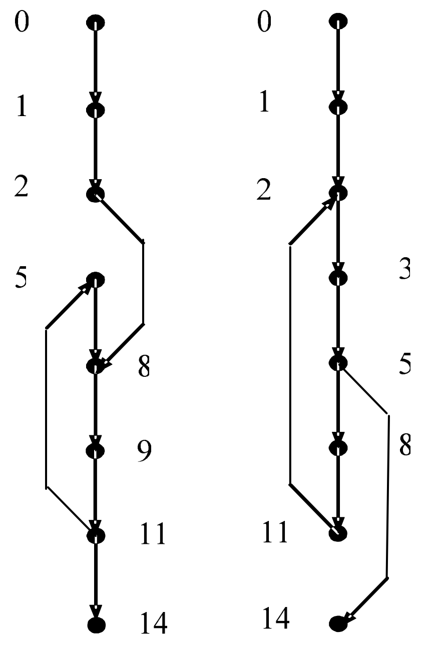
Figure 4-8. The bytecode and CFG of *if-else* construct without *else* clause.

```
for (int i = 1; i < 10; i++)
{;}
```

a. A Java loop construct .

```
0: iconst_1              0:iconst_1
1: istore_1              1:istore_1
2: goto 6                2:iload_1
5: iinc 1, 1             3:bipush 10
8: iload_1               5:if_icmpge 9
9: bipush 10             8:iinc 1, 1
11:if_icmpge -6          11:goto -9
14: return               14:return
```

b. JVM bytecode



c. CFG

Figure 4-9. The bytecode and CFG of the conditional loop construct.

Since Java programming language has no *goto* clause, the loop is the only construct that could be compiled into the bytecode format including one instruction (either an *if*-instruction or an instruction `goto`) that transfers the control flow backward to some instruction before it. If an *if-else* construct has two non-blank branches, an instruction `goto` will be used to separate the two branches in the corresponding bytecode. Furthermore if there is a *return* clause at the first non-blank branch's end of the Java program, the instruction `goto` will be replaced by an instruction α`return` or `return` in the bytecode, which is just like an instruction `goto` jumping to the method's end. Based on these facts, we give the algorithm to locate the non-blank branch's scope in the *if*-instruction construct in Figure 4-10.

```
Given an if-instruction, $i$: if<op><cond> $j$, the $i\_max$ is the address of the last
instruction in the method, the $n$ is the number of the branches and the $S_i$ is the
scope of the branch $b_i$.
If $j < 0$ then
    $n = 1$ and $S_1 = [i+j, i)$    //the if-instruction forms a loop
else
    If the instruction just before $i+j$ is $i'$: goto $j'$ then
        If $j' < 0$ then
            $n = 1$ and $S_1 = (i, i')$    //the if-instruction forms a loop
        else
            $n = 2$ and $S_1 = (i, i')$, $S_2 = [i+j, i'+j')$
        end if
    else
        If the instruction just before $i+j$ is $i'$: αreturn or return then
            $n = 2$ and $S_1 = (i, i')$, $S_2 = [i+j, i\_max)$
        else
            n = 1 and $S_1 = (i, i+j)$
        end if
    end if
end
```

Figure 4-10. The algorithm to locate the branch's scope of *if*-instructions.

4.3.2 Implicit Information Flow Caused by *switch* Statement

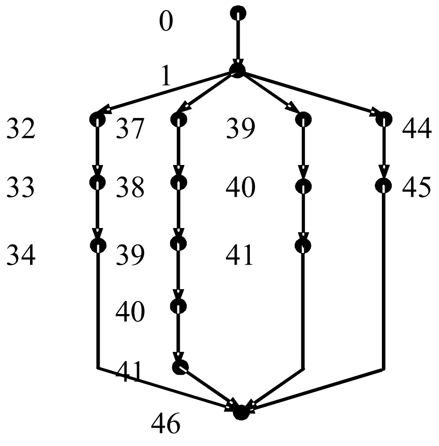
Java programming language's the *switch* structure is another kind of instructions that may cause conditional control transfer. They are compiled using `tableswitch` and `lookupswitch` instructions. Each *case* block of a normal Java language *switch* statement should be ended with a *break* clause and the *default* block (if there is one) should be the last block of the *switch* statement. However, this ideal format is optional and a disordered Java *switch* statement as shown in Figure 4-11 could also be compiled correctly. So we could not use the target addresses to locate the scopes of branches in a *switch* statement directly.

```
public void test ( int a, int b)
{
      switch (a)
      {
            case 0:
                  b = 0;
                  break;
            default:
                  b = 5;
            case 2:
                  b = 2;
                  break;
            case 3:
                   b = 3;
      }
}
```

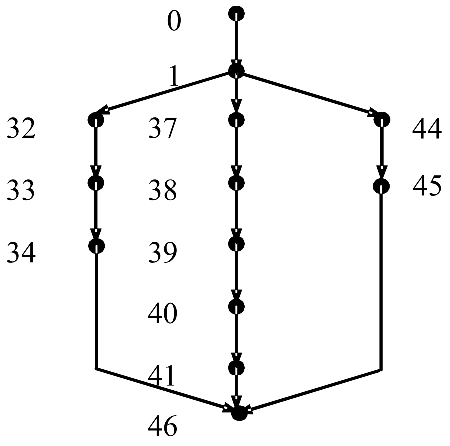
a. A Java language *switch* statement

```
0: iload_1
1: tableswitch{ //0 to 3
            0: 31;
            1: 36;
            2: 38;
            3: 43;
            default: 36 }
32: iconst_0
33: istore_2
34: goto 12
37: iconst_5
38: istore_2
39: iconst_2
40: istore_2
 41 :goto 5
44 :iconst_3
45 :istore_2
46 :return
```

b. The JVM bytecode



c. The original CFG



d. The simplified CFG

Figure 4-11. The bytecode and CFG of the Java language *switch* statement.

```
Given an instruction tableswitch or lookupswitch at the address $i$, form
the target offsets into an array $T[m]$ in ascending order. The $i\_max$ is the address
of the last instruction in the method, the $n$ is the number of the branches and the
$S_n$ is the scope of the branch $b_n$.
$n = 0$, $ret\_exist$ = false, $begin\_address = T[0] + i$ and $end\_address = 0$
For each pair of elements $T[a]$ and $T[a+1]$ in array $T$ ($0 <= a < m - 1$), in forward
order loop
    If the instruction just before $T[a+1] + i$ is $i'$: $\alpha return$ or $return$ then
        $n = n + 1$, $S_n = [begin\_address, i')$,
        $ret\_exist$ = true, $begin\_address = T[a+1] + i$, $end\_address = i\_max$
    else
      If the instruction just before $T[a+1] + i$ is $i'$: $goto\ j$ then
          If $ret\_exist$ is true then
              $n = n + 1$, $S_n = [begin\_address, i') \cup [i' + j, end\_address)$
          else
              $n = n + 1$, $S_n = [begin\_address, i')$, $end\_address = j$
              $begin\_address = T[a+1] + i$
          end if
      end if
    end if
end loop
If $T[m] + i < end\_address$ then
    $n = n + 1$, $S_n = [T[m] + i, end\_address)$
end if
If $n = 0$ then
    $n = 1$, $S_1 = [T[0] + i, T[m - 1] + i)$
end if
```

Figure 4-12. The algorithm to locate the branch’s scope in *switch* statement.

When the control flow gets to the *default* block in Figure 4-11, the block of *case 2* will also be executed since there is no *break* clause at the end of the *default* block. Thus the branch from *default* block (1→37→38→39→40→41→ 46) includes the block of *case 2*. The branch from the block of *case 2* (1→39→40→41→46) could be omitted because it is a part of another branch. In this way the CFG could be simplified as shown in Figure 4-11. The *break* clause in the *switch* statement can be replaced by a *return* clause. Based on the simplified CFG we could draw a conclusion that the *break* clause (compiled into the instruction `goto` *j*) and the *return* clause (compiled into the instruction α`return` or `return`) are the boundary between two branches in *switch* statement. Thus we give the algorithm to locate the non-blank branch's scope in the *switch* statement in Figure 4-12.

4.3.3 Nested Implicit Information Transferring Blocks

In Java programming language the *if-else* construct and *switch* statements could be nested, that is, one *if-else* construct is in the branch of another *if-else* construct or *switch* statements, and vice versa. Therefore the implicit information transferring blocks in Java bytecode could also be nested. Since the scope of each branch in one implicit information transferring block can be calculated by our algorithms shown in Figure 4-10 and Figure 4-12, the nested implicit transferring blocks can also be resolved. The algorithm calculating inner implicit transferring is given in Figure 4-13. With those algorithms, we can divide a section of Java mobile code into explicit transferring blocks and implicit transferring blocks and then apply the proper semantics rules for the instruction in those blocks.

```
Given an branch $b_o$ with the scope $S[s,e]$of the outer implicit transferring block,
$b_i$ is one branch of the inner implicit transferring block in $b_o$.
Use the proper algorithm in Figure 4-11 or Figure 4-12 to calculate the coarse
scope of $b_i$, $S'_i[i, j]$
If s > $i$ then
    $i$ = s
endif
If e < j then
    $j$ = e
endif
The real scope of $b_i$ is $S[i, j]$.
```

Figure 4-13. The algorithm to locate the branch's scope of inner implicit blocks.

5 Method of Bytecode Modification

5.1 Overview

5.1.1 Motivation of Dynamic Verification

The verification of mobile code can be done statically or dynamically. By now the works on mobile code verification for the host security are almost static verification approaches [11, 14, 15, 16, 18, 19, 77 and 78]. The static approach verifies the mobile code from remote hosts before the local JVM executes the code as shown in Figure 5-1. The advantage of the static approaches is that they will not cause additional runtime overhead and will not slow down the execution. But the static approaches have an inherent limitation that it could not get any runtime information of the mobile code execution. This limitation affects the verification precision badly and may make the static approach to lose its practicality. For example, it is impossible for a static approach to get to know which branch of a conditional structure will be executed, where an exception will happen, whether the invoking of a method returns normally or exceptionally, and so on. Therefore the static approach has to verify all the branches of a conditional structure in order to find all potential violation of the security policy, which means that a mobile program may be rejected by the local host for an instruction that will not be executed actually in runtime. As for the exception handling in the mobile code, the static approach has no idea to deal with the information flow in exception handling precisely. What the static approach could do is only to find out all the instructions that may raise exceptions potentially and verify all the instructions possibly executed in runtime. Therefore the same misjudgment occurring in the verification of conditional structure may arise here, too. Obviously, such misjudgment impairs the verification precision and diverges from our research objective.

To get better verification precision, we implement our security model by dynamic approach, which means that the verification is done during the execution of the mobile code. Compared with the static ones, the dynamic approaches have the merit that they can get enough execution information to trace the information flow correctly and to further verify the mobile code more precisely. For example when the execution encounters a conditional instruction, the dynamic approach can get to know which branch will be executed. Thus the approach just verifies instructions to be executed, and eliminates the possibility of verifying the mobile code as malicious for some instructions that will not be executed. Similarly, the dynamic approach can master the change of control flow caused by exception throwing and trace the information flow in exception handling. And all those are too difficult for static approaches to achieve. That merit of dynamic approaches that makes they can get better verification precision than static ones, which is consistent with our objective. While the cost of better verification precision is the additional runtime overhead caused by the verification work done in execution, the development of hardware techniques provides more and more fast calculating speed. We may also need to reduce the additional overhead in runtime caused by the dynamic verification.

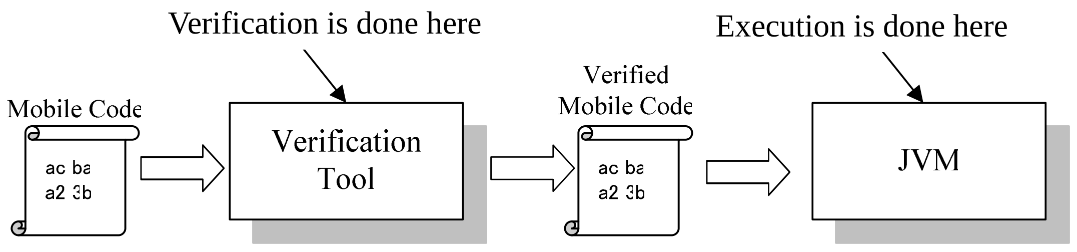


Figure 5-1. Static verification approaches.

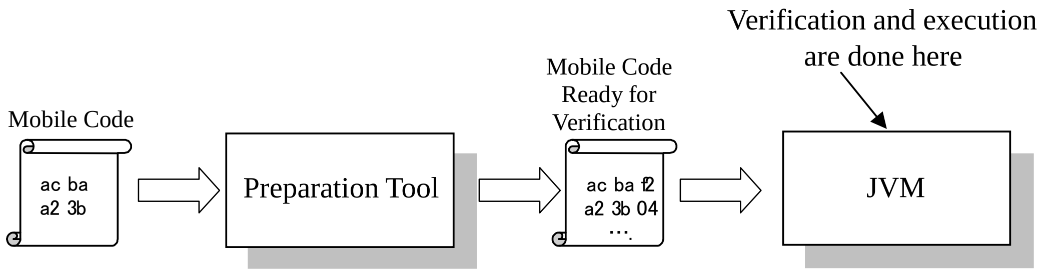

Verification and execution are done here
Mobile Code
ac ba
a2 3b
Preparation Tool
Mobile Code Ready for Verification
ac ba f2
a2 3b 04
….
JVM

Figure 5-2. Dynamic verification approaches.

5.1.2 Bytecode Modification Technique

Java bytecode modification presents the opportunity to change the execution semantics of java programs. A wide range of possible applications have been discussed, ranging from the addition of performance counters, to the support of orthogonal persistence, agent migration, and new security semantics. Here we list some related projects.

Access Control. By intercepting or wrapping calls to potentially dangerous Java methods, systems by Pandey and Hashiiip [96], Erlingsson and Schneider [45], and Chander et al. [26] can apply desired security policies to arbitrary codelets without requiring these policies to be built directly into the Java system code, as done with Java's built-in security system.

Resource Management and Accounting. J-kernel [59] and J-SEAL2 [20] both focus primarily on isolation of codelets. Bytecode modification is used to prevent codelets from interferring in operations of each other. JRes [35] focuses more on resource accounting; bytecode modification is used to instrument memory allocation and object finalization sites.

Optimization. Cream [32] and BLOAT (Bytecode-Level Optimization and Analysis Tool) [91] are examples of systems, which employ Java bytecode modification for the purpose of optimization. Cream uses side-effect analysis, and performs a number of standard optimizations, including dead code elimination and loop-invariant code motion. BLOAT uses Static Single Assignment form (SSA) [34] to implement these and several other optimizations.

Profiling. BIT (Bytecode Instrumenting Tool) [71] is a system which allows the user to build Java instrumenting tools. The instrumentation itself is done via bytecode modification. Other generic bytecode transformation frameworks, such as JOIE [33] and Soot [116], also have hooks to instrument Java code for profiling.

Other Semantics. Sakamoto et al. [105] describe a system for thread migration implemented using bytecode modification. Marquez et al. [81] describe a persistent system implemented in Java entirely using bytecode transformations at class load time. Notably, Marquez et al. also describe a framework for automatically applying bytecode transformations, although the status of this framework is unclear. Kava [126] is reflective extension to Java. That is, it allows for run-time modification and dynamic execution of Java classes and methods.

All of those systems could also be implemented with customized JVMs (and many such customized JVMs have been built.) Of course, fully custom JVMs can outperform JVMs with semantics "bolted on" via bytecode modification because changes can be made to layers of the system that are not exposed to the bytecode, such as how methods are dispatched, or how memory is laid out.

But the price of building custom JVMs is the loss of the portability that is one of the most important advantages of Java. While the strongest argument in favor of bytecode modification is its portability: changes made exclusively at the bytecode level can be moved with little effort from on Java virtual machine to another, so long as the modified bytecode still complies to the JVM specification [75]. To preserve Java's promise "Write Once, Run Anywhere", we adopt the bytecode modification technique rather than the custom JVMs to implement our dynamic verification. An additional benefit is that code added by bytecode modification can still be optimized by the underlying JVM.

And Java has two 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. To modify the Java bytecode, we need reflection functionality to get the structure of a class file, such as the symbolic information, fields, methods, interfaces and attributes. The runtime reflection functionality is added into the 1.1 release of the Java Developer's Kit (JDK). However the Java reflection API is available only after the class has been loaded into the JVM, which is too late for us to do any modification. And the reflection was not designed to extend functionality, and so it does not make available the implementation of class methods. Method implementations are accessible through the `javap` disassembler included in the standard Java Developer's Kit, but `javap` runs from the shell and prints to its standard output; it is not integrated into the Java reflection API, nor does it produce a data structure that can be manipulated by the program. Thus we need more powerful reflection toolkit for our bytecode modification. The reflection functionality of Java bytecode had been studied in [124], and some toolkits such as JOIE and BCEL [21] are available for our modification.

5.1.3 Load-time Modification

There are a number of stages in the program lifecycle during which a program author or user can specify the functionality of a class or set of classes. Some examples of tools used at different stages are detailed in Table 5-1. Originally the base functionality is declared by the class author in the source code, and that source code is translated into an executable image by a compiler.

Stage	**Example Use**	**Example Tool**
Pre-processor	macros or conditional compilation	cpp
Compiler	translation from source to classfile	javac
Post-processor	Instrumentation	ATOM, BIT
Component Integration	Setting text, color	Bean Builder
Load-time	User-supplied modification, templates	ClassLoader, KOIE
Just-In-Time compilation	Compilation to native code	JIT

Table 5-1. Stages in the program development life cycle.

Authors or users can employ post-processors such as instrumentation tools to insert new method calls into an existing executable image. A popular example of that is the tool ATOM [114], which works on executable images for Alpha processor; similar functionality is available for Java with BIT [70]. Most often, this instrumentation is used for performance analysis or as an interface to platform simulation. An important guarantee typically made by instrumentation tools is that the semantics of the original program are not changed. However Shasta [109] processes executable images to run on distributed shared memory systems. Object Design Incorporateds's Object Store PSE [92] also uses a post-processor, to insert persistence methods into existing code. Rational Software Corporation's tool Purify [99] changes code to detect memory leaks.

Multiple third-party components (classes or more often collections of interacting classes) are integrated during application composition. In Java, these components are known as Beans and are often handled in visual builders. This composition allows consumers of code – either end-users or programmers using components in their own application – to modify certain properties of the component. However, users can only modify those properties foreseen by the original author. They cannot independently add features except through the basic object-oriented techniques of inheritance.

After application composition, the classes are eventually loaded into the environment. During execution, the bytecode can be translated into native local platform instructions by Just-In-Time compiler (JIT). JITs only re-implement the bytecode in a different language. They do not add new functionality (although JITs may transform the code for optimization, for example unrolling loops or recording instructions.)

The JVM loads Java classes from disks or elsewhere through class loaders, invoked as part of Java's dynamic linking mechanism. The process of loading a class through a class loader is shown in Figure 5-3. When an already loaded class (the class `Vehicle`) uses an undefined class (the class `Car`), either by accessing a static member or creating an instance of the class, the JVM traps the undefined reference and send a request for the class loader to load the class. The class loader fetches the class file (Car.class) from the files system. Then the input class is loaded into the JVM and the execution continues.

In the program development life cycle, we choose the *Load-time* to apply the bytecode modification. The architecture of JVM, in which classes are loaded on demand by a user-extensible class loader, offers a complementary alternative to the previous steps: load-time modification meaning that the class loader is responsible not only for locating the class, but for modifying the bytecode in ways specified by the user. Therefore in the process of loading class files, after the class loader fetches the class file it implements the modification of the bytecode and then sends the modified class file to the JVM.

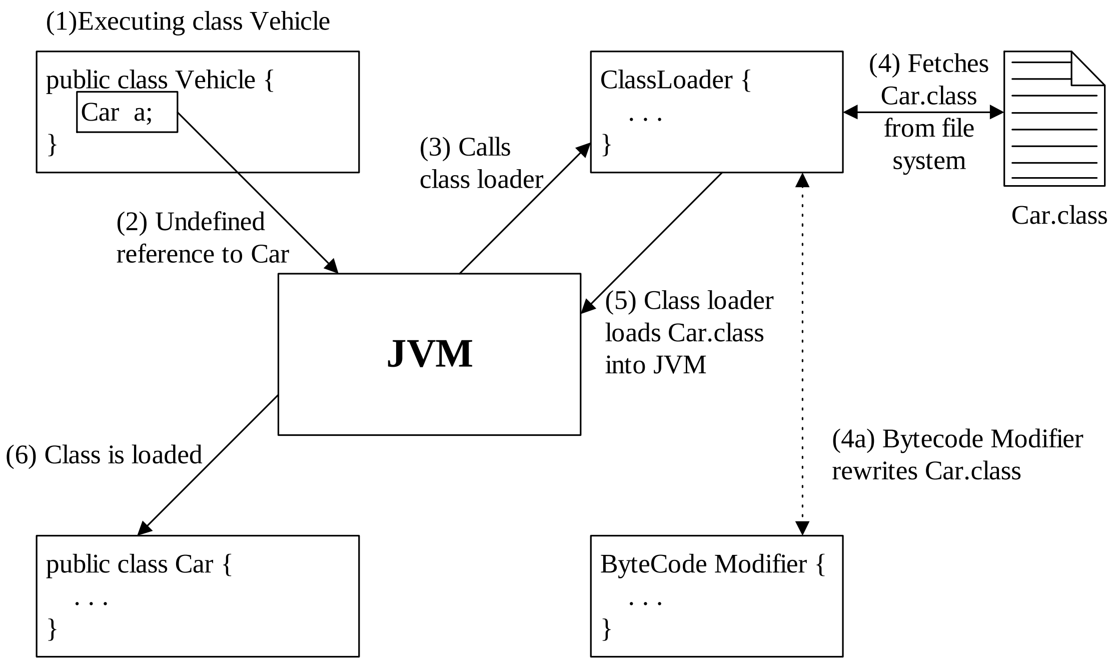

(1)Executing class Vehicle
public class Vehicle {
Car a;
}
ClassLoader {
. . .
}
(4) Fetches Car.class from file system
Car.class
(3) Calls class loader
(2) Undefined reference to Car
JVM
(5) Class loader loads Car.class into JVM
(6) Class is loaded
(4a) Bytecode Modifier rewrites Car.class
public class Car {
. . .
}
ByteCode Modifier {
. . .
}

Figure 5-3. The process of loading class files through the 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. A modification registered with a class loader can be applied to all classes that are eventually loaded into the JVM.

5.1.4 Modification Contents

As mentioned in Chapter 4, to use our approach to verify a mobile code program, we should maintain a distributed map of security-level during the execution in order the trace and record the information flow. And then at each data-leaking channel, we compare the security-levels and clearance-level to check whether a data-leaking arises according to the security-level distribution map.

In order to achieve the dynamic verification of mobile code programs, we chose the technique of bytecode modification to implement our approach. At first to construct a distribution map of security-level, we need to allocate additional containers to store the security-level of the data held by the mobile code's information carriers, which are called *security-level containers*. Then to maintain the distribution map of security-level during the execution of the bytecode, we need to insert additional instructions to calculate the change of the security-levels caused by information transferring, both the transferring among carriers in one method and the transferring between methods in order to update the distribution map of security-levels. At last to check whether any data-leaking rises, we need to insert instructions of comparing the security-levels and clearance-levels to each data-leaking channel. All those work can be achieved by modifying the parameters, methods and classes' declaration in the mobile code program. In general, the bytecode modification in our approach can be divided into two main parts, *class redefinition* and *instruction insertion*. The former includes adding additional data containers, modifying the parameters and return type of methods, while the latter includes inserting proper instructions to calculate security-levels and check data-leaking.

5.2 Class Redefinition

We have discussed the structure of the class file and the information flow in Java bytecode in Chapter 3 and Chapter 4. The information transferring in the Java bytecode may be among the information carriers in one method, or between the caller and callee methods. Thus we discuss the class redefinition necessary for the two kinds of information transferring respectively.

To explain the modification more clearly, we give an example class here. The Java program in Figure 5-4 defines a class named as *Circle*. The Java bytecode and the class file structure of the class *Circle* are shown in Figure 5-5 and Figure 5-6.

In the class file of the class *Circle*, the section `Header` includes the magic number and the version information. The section `Constant Pool` represents various string constants, class and interface names, field names, and other constants, such as the initialization method's name, the field *radius*'s type and the constant 3.14 defined in the class *Circle*. The section of `Access Right` gives the value used to denote access permissions to and properties of the class *Circle*. The section

`Fields` gives a complete description of the fields *center_x*, *center_y* and *radius*, which are defined directly in the class *Circle*. The section `Methods` gives a complete description of all methods declared by the class *Circle*, including instance methods *area* and the instance initialization methods *Circle*. At last the section `Class Attributes` defines some attributes of the class *Circle* such as `SourceFile` attribute and the `Deprecated` attribute.

The following discussion of class redefinition will take the class *Circle* as an example.

```
public class Circle{

        private float center_x;
        private float center_y;
        private float radius;
        final float pi = 3.14;

        Circle(float x, float y, float r){
                center_x = x;
                center_y = y;
                radius = r;
        }

        float area(){
                return pi*radius*radius;
        }

        boolean isInCircle(float x, float y){
                boolean result = false;
                float dis = (x – center_x)* (x – center_x)
                                + (y – center_y)* (y – center_y);
                if (dis <= radius*radius){
                        result = true;
                }
                return result;
        }
}
```

Figure 5-4. The Java program of the class *Circle*.

```
Compiled from "Circle.java"
public class Circle extends java.lang.Object{
final float pi;

Circle(float, float, float);
   Code:
     0:    aload_0
     1:    invokespecial      #1; //Method java/lang/Object."<init>":()V
     4:    aload_0
     5:    ldc         #2; //float 3.14f
     7:    putfield  #3; //Field pi:F
     10:   aload_0
     11:   fload_1
     12:   putfield  #4; //Field center_x:F
     15:   aload_0
     16:   fload_2
     17:   putfield  #5; //Field center_y:F
     20:   aload_0
     21:   fload_3
     22:   putfield  #6; //Field radius:F
     25:   return

float area();
   Code:
     0:    ldc         #2; //float 3.14f
     2:    aload_0
     3:    getfield  #6; //Field radius:F
     6:    fmul
     7:    aload_0
     8:    getfield  #6; //Field radius:F
     11:   fmul
     12:   freturn

boolean isInCircle(float, float);
   Code:
     0:    iconst_0
     1:    istore_3
     2:    fload_1
     3:    aload_0
     4:    getfield  #4; //Field center_x:F
     7:    fsub
     8:    fload_1
     9:    aload_0
     10:   getfield  #4; //Field center_x:F
```

```
      13:   fsub
      14:   fmul
      15:   fload_2
      16:   aload_0
      17:   getfield  #5; //Field center_y:F
      20:   fsub
      21:   fload_2
      22:   aload_0
      23:   getfield  #5; //Field center_y:F
      26:   fsub
      27:   fmul
      28:   fadd
      29:   fstore     4
      31:   fload      4
      33:   aload_0
      34:   getfield  #6; //Field radius:F
      37:   aload_0
      38:   getfield  #6; //Field radius:F
      41:   fmul
      42:   fcmpg
      43:   ifgt       48
      46:   iconst_1
      47:   istore_3
      48:   iload_3
      49:   ireturn

}
```

Figure 5-5. The Java bytecode of the class *Circle*.

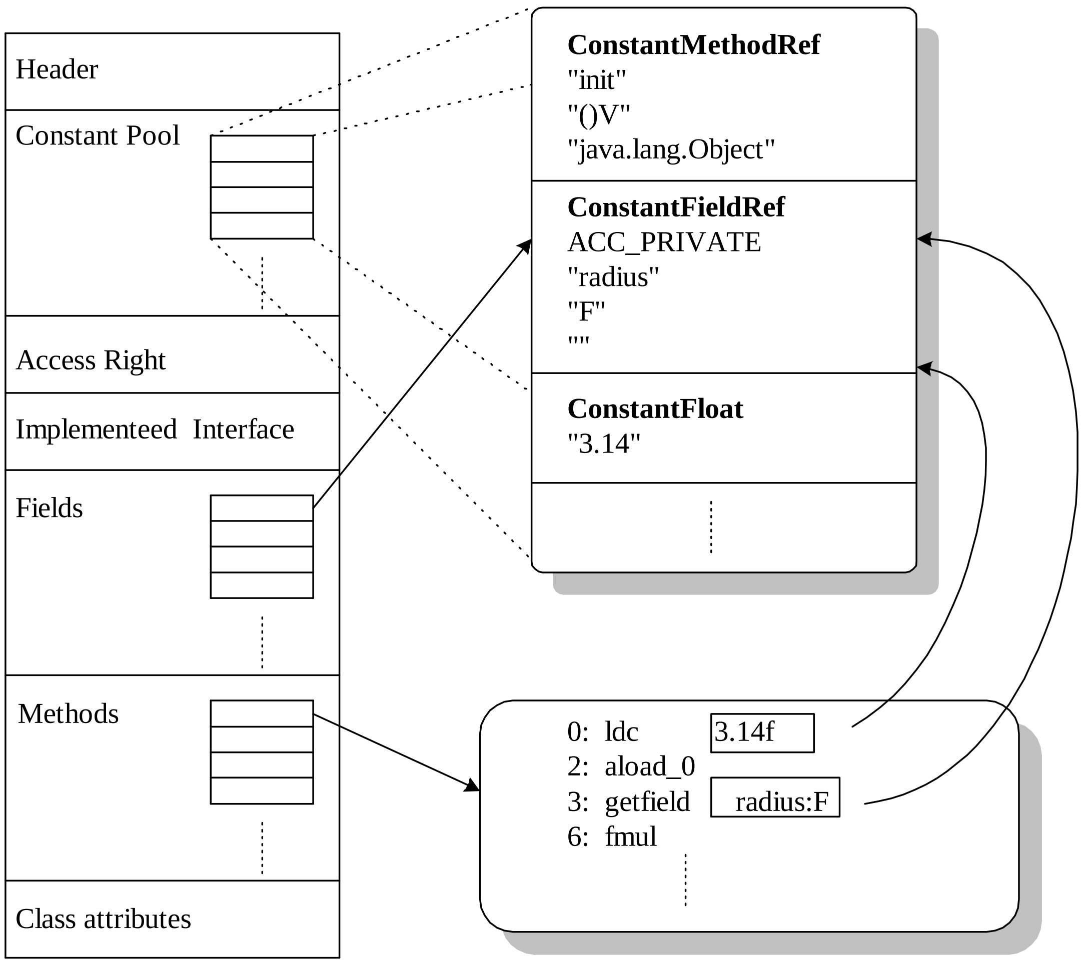


Figure 5-6. The class file structure of the class *Circle*.

5.2.1 Redefinition for Intra-procedural Information Transferring

The Java Virtual Machine 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 all the information transferring in one method can be considered as the information transferring among the three kinds of elementary information carriers. Therefore we need to add security-level containers for the information in those elementary carriers respectively.

- **Local Register**

For the local register (which is used to store the local variables of the method), we allocate an additional register as the security-level container of the information in the original register. Adding new local registers is trickier than adding new entries to the Constant Pool. In particularly the JVM specification requires that the arguments to the method appear in order at the low local registers before the local variables appear. Considering that we will add new arguments to a method (which will be discussed later) and the local registers' indices in instructions should be recalculated, we store the security-level of the information in a local variable to the register just after the one storing the local variable. 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 or two according to the length of the local variable in the 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`. And we also should reset the value of the item `max_locals` to make the JVM allocate additional local registers used as security-level containers for the method. We show the allocating new registers as seceurity-level containers for the method *isInCircle* in Figure 5-7 as an example.

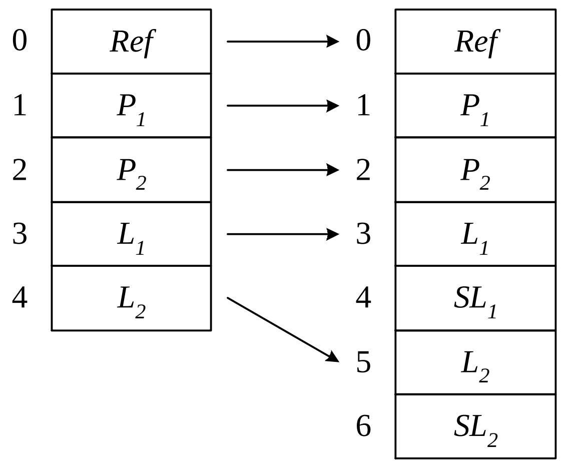


Figure 5-7. The security-level containers for information in local registers. *Ref* is a reference to the method's instance; P_1 and P_2: the parameters of the method; L_1 and L_2: the local variables in the method; SL_1 and SL_2: the security-levels of the information in V_1 and V_2 respectively.

- **Operand Stack**

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. We store the security-level of the information in one stack element to the element just before the original one in the direction counted from the top of the stack, that is, we keep an internal order of variables and their security-levels on the stack. By this way, we make the JVM push the security-level of one variable to the stack after it push the variable, and pop the security-level from the stack before it pop the variable. Similar to the local registers, we also need to reset the value of the item `max_stack` defining the maximum size of the stack since the elements pushed to the stack during the execution of the modified bytecode increase. We give an example of allocating new stack elements as security-level containers in Figure 5-8.

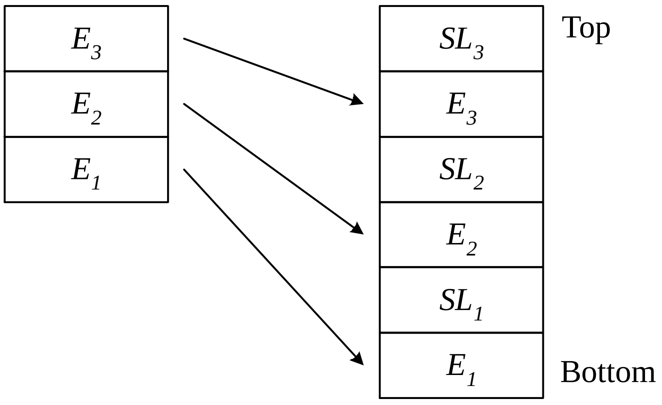


Figure 5-8. The security-level containers for information in the elements on the operand stacks. E_1, E_2 and E_3: the elements on the operand stack; SL_1, SL_2 and SL_3: the security-levels of the information in E_1, E_2 and E_3 respectively.

- **Class Fields**

Adding security-level containers for the fields of one class is much different from adding containers for local variables. We should decide the new field's type, name, access flag and the position we insert it.

To decide the types of the new fields (used as the security-level containers for the original fields), we divide the original fields into three kinds according to their types: the fields of primary types, the fields of class types and the fields of array types. For the field of primary type, it can only hold one data in it. Thus, we add a new field of type `byte` (because all security-levels are integers) as the security-level container of the original field. For the field of class type, it can concern with a lot of information since the data held in it is a reference. Thus we add a new field of the same class type as the original field, and the security-levels of the original field's members are stored in the added field's corresponding members (not the members have the same names, but those added as security-level containers when the class of the field' type is modified). So that the new field (security-level container) can hold the same number of data as the original one. For the field of array types, the field also holds one reference like the field of class type. Thus we also add a new field of the same array type as the original one, which is used as the security-level container for the original field.

The name of the new field used as the security-level containers is the original field's name suffixed with "*_SL*". And the new 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.

Beside the original fields of one class, the class instance itself (the reference) also holds information and can be used in the information flow. That fact makes it necessary for us to do two things: first is that we need to add a new field of type `byte` as the security-level container of the class reference itself; second, which has been mentioned above, is that we use the added members in the added field rather than the added members in original field to store security-level because the original field may be null and we cannot use the member of a null reference to store the security-level of the reference itself.

We give an example of adding new fields as security-level containers in Figure 5-9.

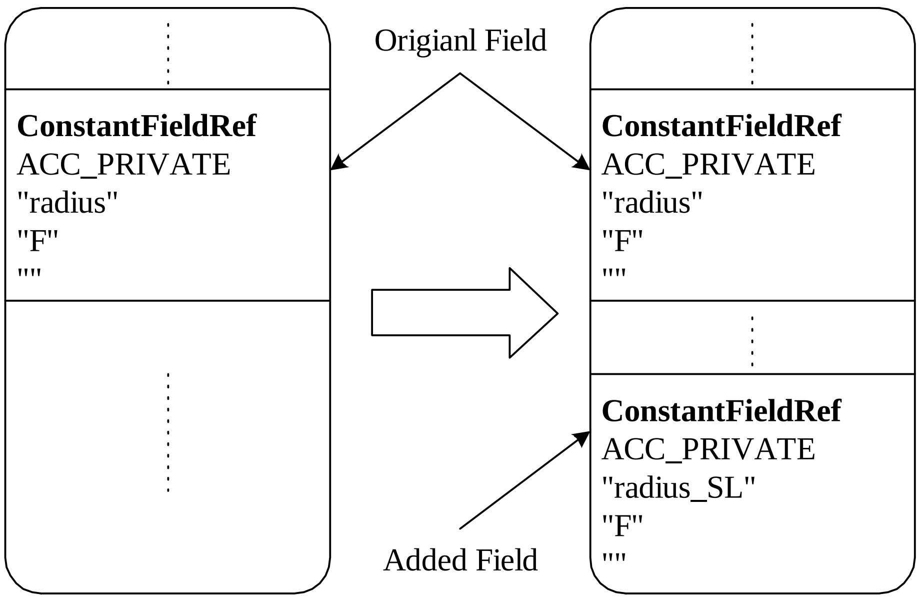


Figure 5-9. The security-level containers for information in the class fields.

5.2.2 Redefinition for Inter-procedural Information Transferring

The information can 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.

Adding security-level container for parameters is the combine of locating new local registers and adding new fields. Given one method having one or more arguments, we add one new argument as the security-level container for each original argument. The type of the new arguments is decided in the same way as we decide the type of the new fields. As for the new parameter’s order in the sequence of all parameters, we insert the new argument just after the original argument. By this way keep the alternate order of the local registers and their security-level containers since the arguments will be loaded to the local registers. Because the arguments’ names are not saved in the class file, we need not name the new arguments. For example, considering the method void *Circle*(*float x*, *float y*, *float r*) with the descriptor (*FFF*)*V* in the class *Circle* shown in Figure 5-4, we add one argument of type `byte` after each original argument. Thus the descriptor of the modified method *Circle* is (*FBFBFB*)*V*.

For the return value of one method, we cannot deal with it as we do for parameters since one method may have multiple parameters but it can only return one value. The only thing that we can do is to change the type of the return value. Given one method with a return value of primitive type or class type, 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 container for the return value. (For the security-level container of the return value of primary types, we convert the security-level to the type `byte`.) If the return type is an array $T[n_1][n_2]...[n_m]$, we alter the return type to the array type of $T[2][n_1][n_2]...[n_m]$. The first element of the first dimension is the original return value, and the second element is the security-level container for the original one.

By this way, we keep consistent with the rules of adding local registers, fields and parameters. For example, considering a method *float area*() 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*.

5.3 Instructions 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 offset of conditional instruction should be recalculated so that they can branch to the correct instruction. Similar to the discussion of class redefinition, we discuss the instructions insertion for the information transferring in one method and the information transferring between the caller and callee methods respectively. And we also discuss the insertion of the instruction for checking data-leaking.

5.3.1 Intra-procedural Information Transferring

The information transferring can be divided into explicit transferring and implicit transferring. By the algorithms given in Chapter 4, we can partition the bytecode of one method into explicit blocks and implicit blocks. In explicit blocks, the information flow is explicit flow and 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 flow is implicit flow and the information is transferred from the conditional variables of the implicit flow to the defined variables additionally. Since the implicit blocks can be nested, one implicit information flow may consist of several implicit blocks and in the case the conditional data of the implicit information flow include all the conditional variables of the implicit blocks. We define the *conditional security-level* of one implicit transferring block as the security-level of the conditional variable or, if there are more than one conditional variable, the LUB of the security-levels of all conditional variable of the implicit transferring block. And we can get the formula 5-1, in which L_{env} is the environment security-level of one implicit information flow, L_{coni} is the conditional security-level of the *i*th block of implicit transferring blocks composing the implicit information flow, and *m* is the number of the implicit transferring blocks.

$$L_{env} = L_{con1} \vee L_{con2} \vee \ldots \vee L_{conm} \qquad \text{....................} \; 5\text{-}1$$

Thus at the beginning of one implicit transferring block, we should first insert proper instructions to calculate the conditional security-levels of the current implicit block, and then calculate the environment security-level and store it (in order to make it easier to calculate the environment security-level of the inner implicit information flow). Then we should insert proper instructions to assign the LUB of the environment security-level and the all security-levels if used variables to the security-level container of the defined variable.

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. In the way, when the JVM executes one original bytecode instruction, the operands used by the instruction on the stack are laid as if no instruction is inserted, which assure that the inserted instructions has no side affect on the original functionality of the bytecode.

We give an example of inserting instructions in Figure 5-10. 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 instructions at address 1, 3, 32 and 35 are inserted for loading instructions, the instructions at address 27 and 26 are inserted for storing instructions, and the instructions at address 4 to 22, 25, 37 to 54 and 57 are inserted for operating instructions in original bytecode respectively. And in the modified bytecode, the indices of local variables have been recalculated and the new indices of original local variable are 1, 3, 5 and 7.

In implicit blocks besides the instructions inserted in explicit blocks, we should insert additional instructions to calculate and store the environment security-level.

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_{cenv} 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 intermediate 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 environment 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.

```
public void cal(int a, int b){
        int c = a + b;
        int d = a * c;
        return;
}
```

a. Java source code

```
0: iload_1
1: iload_2
2: iadd
3: istore_3
4: iload_1
5: iload_3
6: imul
7: istore 4
9: return
```

b. Java bytecode

```
 0: iload_1          32: iload_2
 1: iload_2          33: iload 5
 2: iload_3          35: iload 6
 3: iload_4          37: istore 9
 4: istore 9         39: istore 10
 6: istore 10        41: iload 9
 8: iload 9          43: If_icmple 9
10: If_icmple 9      46: pop
13: pop              49: istore 9
14: istore 9         51: goto 6
16: goto 6           54: istore 9
19: istore 9         56: pop
21: pop              57: iload 10
22: iload 10         59: iadd
24: iadd             60: iload 9
25: iload 9          62: istore 8
27: istore 6         64: istore 7
29: istore 5         66: return
31: iload_1
```

c. modified bytecode

Figure 5-10. An example of instructions insertion.

5.3.2 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 its security-level to an array of proper type. 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 for the return value of primary types.

5.3.3 Data-leaking Checking

As mentioned above, 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.

6 Information Flow in Exception Handling

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

6.1 Motivation

When a program violates the semantic constraints of the Java programming language, the Java virtual machine signals this error to the program as an *exception*. Besides the implicit occurrence, the exception can also be explicitly caused by the statement *throw* in Java programming language. 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 to. Obviously, the occurrence of an exception and the corresponding exception handling will change the control flow and thus affect the information flow. A failure to account for the effects of exception occurrence and exception handling constructs in performing analyses system can result in incorrect analysis information, which in turn can result in unreliable security verification systems.

The additional expense that is required to perform analyses accounting for the effects of exception handling constructs may not be justified unless these constructs occur frequently in practice. In [112], Sinha and Harrold examined a number of non-trivial, real-life Java programs from a diverse group of applications in order to determine the frequency with which java programs use exception handling statements. The result of the study is shown in Table 6-1, which includes the description of each program group, the number of programs examined and the usage of exception handling statements.

Program Group		Numbers of Programs	Programs that Contain	
Name	Description		*try* Stmts	*throw* Stmts
jacorb	ORB implementation in java.	1062	271	229
javacup	LALR-parser generator for Java.	34	5	17
jdk	Sun's JDK 1.1.5.	1256	342	372
jlex	Lexical-analyzer generator for Java.	1	1	1
swing	Sun's Swing API 1.0.2.	692	87	106
tdb	Debugger for Java.	8	3	5
toba	Java bytecode-to-C translator.	43	13	27
Total		3096	722	757

Table 6-1. Presence of exception handling statements in Java programs.

As the tables illustrates, 23.3% and 24.5% of the examined programs contained *try* and *throw* statements respectively. Within a program group, ignoring the values for `jlex`, these percentages varied from 12.6% to 37.5% for *try* statements and, 15.3% to 62.8% for *throw* statements. Several programs contained both *try* and *throw* statements, and over all program groups, there were 497 such programs. Therefore, there were 982 programs, which comprise 31.7% of all examined programs, which contained either a *try* statement or a *throw* statement. The study supports that the use of exception handling statements in real-life programs is significant enough that it should be considered during various analyses.

The discussion above proves that the information flow in Java bytecode exception handling cannot be ignored in our analyses for the Java bytecode verification. The ignorance of exception handling will cause unreliable verification result and make the verification approach unpractical.

6.2 Exception Handling in Java Language

In java, exceptions can be synchronous or asynchronous. *Synchronous* exceptions occur at particular program points and are caused by expression evaluation, statement execution, or explicit *throw* statements. Synchronous exceptions can be checked or unchecked: for *checked* exceptions, the compiler must find a handler or a signature declaration for the method that raised the exception; for unchecked exceptions the compiler does not attempt to find such an associated handler or a signature declaration. Synchronous exceptions are further classified as pre-defined or user-defined: *pre-defined* exceptions are defined by the Java language; *user-defined* exceptions are defined by users of the language. For example, the method `write`() defined in `java.io.DataOutputStream` can raise a pre-defined checked exception `IoException`. While the method `pop`() defined in `java.util.Stack` can raise a pre-defined unchecked exception `EmptyStackException`. Users can define a checked exception by extending `java.lang.Exception` or `java.lang.Throwable`. Similarly, users can define an unchecked exception by extending `java.lang.Error` or `java.lang.RuntimeException`.

Asynchronous exceptions occur at arbitrary, non-deterministic points in a program's execution, and are unchecked. Asynchronous exceptions occur when either the Java Virtual Machine raised an instance of `InternalError` (because of faults in the virtual-machine software, the host-system software, or the hardware), or a thread invokes the method `stop`() that raised an instance of `ThreadDeath` in another thread. Figure 6-1 shows the types of Java exceptions.

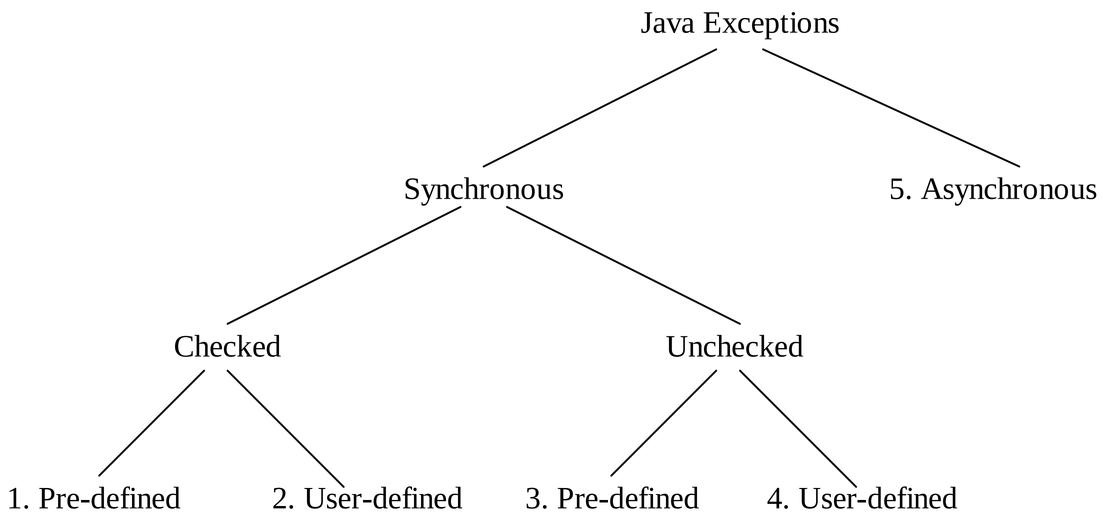


Figure 6-1. Exception types in Java.

In Java programs, all synchronous, pre-defined exceptions (type 1 and 3 in Figure 6-1) are raised as a result of expression evaluations, statement executions or *throw* statements. While synchronous, user-defined exceptions (type 2 and 4 in Figure 6-1) are raised by the *throw* statements only. In Java all thrown exceptions are instances of classes derived from the class `java.lang.Throwable`.

In Java language, a *try* statement is the exception-handling construct. A *try* statement consists of a `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 will cause *intra-procedural* control transferring (if the exception is handled in the method where it is raised) or *inter-procedural* control transferring (if the exception is not handled in the method where it is raised and thrown to the caller method). We summarize the exception handling process in Figure 6-2. The figure shows a *try* statement and its components blocks; the conditions triggering the control flow between the blocks are numbered and listed under to the figure. In the following, we list all possible types of path within a *try* statement.

Path 1 is taken if the `try` block raises no exception and no `finally` block is specified in this *try* statement.

Path 2 is taken if the `try` block raises exception and no matching `catch` block can be found in this *try* statement. No `finally` block is specified in this *try* statement.

Path 3-10 is taken if the `try` block raises no exceptions. The `finally` block is specified in this *try* statement and raises no exception.

Path 3-11 is taken if the `try` block raises no exceptions. The `finally` block is specified in this *try* statement and raises exception.

Path 4-11 is taken is if the `try` block raises exception and no matching `catch` block can be found in this *try* statement. The `finally` block is specified in this *try* statement. If the `finally` block raises no exception, the exception raised in the `try` block is propagated to the outer *try* statement. If the `finally` block raises exception, the exception raised in the `finally` block is propagated to the outer *try* statement.

Path 5-6-10 is taken if the `try` block raises exception and the matching `catch` block is found in this *try* statement. The `catch` block raises no exception and the exception is handled. The `finally` block is specified in this *try* statement and raises no exception.

Path 5-6-11 is taken if the `try` block raises exception and the matching `catch` block is found in this *try* statement. The `catch` block raises no exception and the exception is handled. The `finally` block is specified in this *try* statement and raises exception.

Path 5-7-11 is taken if the `try` block raises exception and the matching `catch` block is found in this *try* statement. The `catch` block raises exception and the exception is not handled. The `finally` block is specified in this *try* statement. If the `finally` block raises no exception, the exception raised in the `try` block is propagated to the outer *try* statement. If the `finally` block raises exception, the exception raised in the `finally` block is propagated to the outer *try* statement.

Path 5-8 is taken if the `try` block raises exception and the matching `catch` block is found in this *try* statement. The `catch` block raises no exception and the exception is handled. No `finally` block is specified in this *try* statement.

Path 5-9 is taken if the `try` block raises exception and the matching `catch` block is found in this *try* statement. The `catch` block raises exception and the exception is not handled. No `finally` block is specified in this *try* statement.

Path 12-6-10 is taken if an unhandled exception is propagated from nested blocks and the matching `catch` block is found in this *try* statement. The `catch` block raises no exception and the exception is handled. The `finally` block is specified in this *try* statement and raises no exception.

Path 12-6-11 is taken if an unhandled exception is propagated from nested blocks and the matching `catch` block is found in this *try* statement. The `catch` block raises no exception and the exception is handled. The `finally` block is specified in this *try* statement and raises exception.

Path 12-7-11 is taken if an unhandled exception is propagated from nested blocks and the matching `catch` block is found in this *try* statement. The `catch` block raises exception and the exception is not handled. The `finally` block is specified in this *try* statement. If the `finally` block raises no exception, the exception raised in the `try` block is propagated to the outer *try* statement. If the `finally` block raises exception, the exception raised in the `finally` block is propagated to the outer *try* statement.

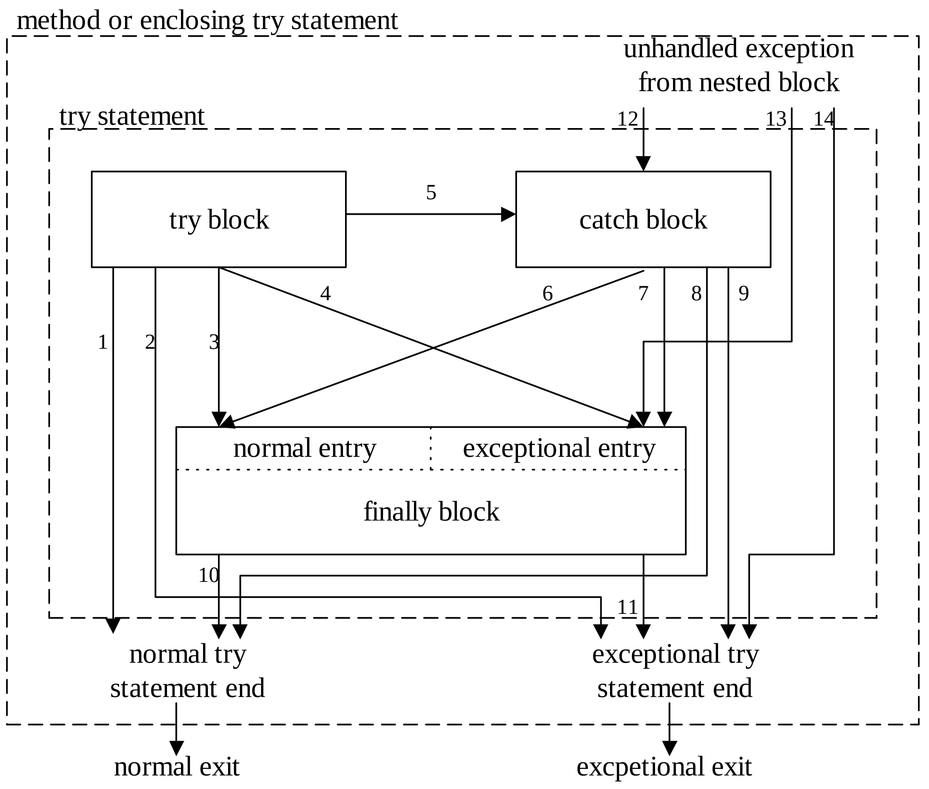
Path 12-8 is taken if an unhandled exception is propagated from nested blocks and the matching `catch` block is found in this *try* statement. The `catch` block raises no exception and the exception is handled. No `finally` block is specified in this *try* statement.

Path 12-9 is taken if an unhandled exception is propagated from nested blocks and the matching `catch` block is found in this *try* statement. The `catch` block raises exception and the exception is not handled. No `finally` block is specified in this *try* statement.

Path 13-11 is taken if an unhandled exception is propagated from nested blocks and no matching `catch` block is found in this *try* statement. The `finally` block is specified in this *try* statement. If the `finally` block raises no exception, the exception raised in the `try` block is propagated to the outer *try* statement. If the `finally` block raises exception, the exception raised in the `finally` block is propagated to the outer *try* statement.

Path 14 is taken if an unhandled exception is propagated from nested blocks and no matching `catch` block is found in this *try* statement. No `finally` block is specified in this *try* statement.

Within `catch` blocks, all handlers are examined in the order in which they appear to find one handler that is a super-type of a raised exception. And no priority is given to an exact match handler over one requiring the application of an inheritance relationship. A raised exception *E* matches a `catch` handler *H* if *E* and *H* are of the same type or *H* is a super-class of *E*.



1. no exceptions raised in `try` block; no `finally` block
2. exception raised in `try` block; no matching `catch` block; no `finally` block
3. no exceptions raised in `try` block; `finally` block specified
4. exception raised in `try` block; no matching `catch` block; `finally` block specified
5. exception raised in `try` block; matching `catch` block specified
6. exception handled; `finally` block specified
7. `catch` block raises exeption; `finally` block specified
8. exception handled; no `finally` block
9. `catch` block raises exeption; no `finally` block
10. no exceptions raised in `finally` block
11. `finally` block propagates previous exception or raised another exception
12. unhandled exception from nested block; matching `catch` block specified
13. unhandled exception from nested block; no matching `catch` block; `finally` block specified
14. unhandled exception from nested block; no matching `catch` block; no `finally` block

Figure 6-2. Control flow in Java exception handling constructs.

6.3 Exception Handling in Java Bytecode

In Java bytecode, exceptions can be thrown explicitly by the instruction `athrow` or implicitly by some specific instructions such as those shown in Figure 6-3. The exception handling In Java bytecode has the same procedure as that in Java programming language, and thus has all the possible paths shown above. However, the presentation of *try* statements in Java bytecode is not so clear as that in Java programming language. In the latter, the scope of each block (`try` block, `catch` block and `finally` block) in a try statement can be located easily by the enclosing symbols "{" and "}". While the location in Java bytecode is not so straight. In next section, we will discuss the presentation of *try* statements in Java bytecode and give the algorithm to locate the scope of each block in a *try* statement.

Instruction	*Stack*	*Exceptions Thrown*
aaload	*arrayref, index* => *v*	ArrayIndexOutOfBoundsException, NullPointerException
bastore	*arrayref, index, v* =>	ArrayIndexOutOfBoundsException, NullPointerException
iaload	*arrayref, index* => *v*	ArrayIndexOutOfBoundsException, NullPointerException
idiv	*value1,value2* =>*result*	ArithmeticException
instanceof	*objectref* =>*restult*	Resolution Exceptions
invokestatic	[*arg1*,[*arg2…*]]	Resolution Exceptions
Ldc	…=>*item*	Resolution Exceptions
newarray	*count* =>*arrayref*	NegetiveArraySizeException
putfield	*objectref,value*=>	Resolution Exceptions, NullPointerException

Figure 6-3. Some Java bytecode instructions that can throw exceptions.

6.3.1 Compilation of *try* Statement in Java Bytecode

Different from the straight and clear presentation format of *try* statements in

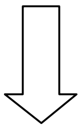
Java programming language, the presentation format in Java bytecode is a little complex.

- **Compilation of `try` and `catch` Blocks**

The compilation of Java's `try-catch` construct is straightforward. Figure 6-4 give a simple example of Java program with `try-catch` construct being compiled into Java bytecode. The `try` block is compiled just as it would be if the `try` were not present. If no exception is thrown during the execution of the `try` block, it behaves as though the `try` were not there. Following the `try` block is the Java bytecode implementing the single `catch` block. The contents of the `catch` block are also compiled like a normal method. However, the presence of a `catch` clause caused the compiler to generate an instruction between the bytecode of `try` and `catch` blocks, which can change the control flow to avoid the unconditional execution of the `catch` block such as `return`, `jsr`, `goto` and so on. Furthermore the compiler will generates an *exception table* entry for each `catch` block to indicate the scope of `try` block that the `catch` block deals with by the index pairs [*from*, *to*], the beginning index of the `catch` block by the column *target* and the exception type that the `catch` block can handle by the column *type*. In the example shown in Figure 6-4, if some value that is an instance of *TestExc* is thrown during the execution of the instructions between indices 0 and 4 (inclusive), the control is transferred to the instruction at index 5, which is the beginning of the `catch` block.

Multiple `catch` blocks of a given *try* statement are compiled by simply appending the Java bytecode for each `catch` block one after the other, and adding entries to the exception table. If during the execution of the `try` block, an exception is thrown that matches the handler type of one or more of the `catch` blocks, the first such `catch` block is selected. Control is transferred to the bytecode for the `catch` block. Here no priority is given to the `catch` block with the exact matching exception type over one with super-class exception type. If no such `catch` block can be found, the JVM re-throws the exception without invoke the bytecode in any `catch` block. And nested *try* statements are compiled very like a *try* statement with multiple `catch` blocks. The nesting of `catch` blocks is represented only in the exception table. When an exception is thrown, the innermost `catch` block containing the site of the exception and having a matching handler type is selected to handle the exception. It is so even that the exception occurs within the bounds of the outer `catch` block and even the outer `catch` block might otherwise have been able to handle the thrown exception.

```
void catchOne() {
    try {
        tryItOut();
    }
    catch (TestExc e) {
        handlExc (e);
    }
}
```



```
Method void catchOne()
 0 alod_0              //Beginning of try block
 1 invokevirtaul #6  //Method tryItOut()V
 4 return              //End of try block
 5 astore_1            //Beginning of catch block
 6 aload_0
 7 aload_1
 8 invokevirtual #5 //Method handleExc(LTestExc;)V
11 return

 Exception table:
 From     To    Target  Type
 0          4      5        Class TestExc
```

Figure 6-4. An example of `try-catch` construct's compilation.

- **Compilation of `finally` Block**

In Java bytecode, the `finally` block can be compiled as an embedded routine or as bytecode appended to the `try` block and `catch` blocks (if there is any). The Java program in Figure 6-5 has a *try* statement with `try`, `catch` and `finally` blocks. We give the bytecode compiled from the program in the two compilation ways in Figure 6-6 and Figure 6-7.*

* The bytecode in Figure 6-6 is generated by JDK 1.4.1 and that in Figure 6-7 by JDK 1.4.2.

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

Figure 6-5. An example of Java program with the `finally` block.

```
void
testOfcatch(int[],int);
Code:
  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
```

Figure 6-6. The `finally` block is compiled as a subroutine.

```
void testOfcatch(int[],int);
Code:
  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   12   32    any
  19   25   32    any
  32   34   32    any
```

Figure 6-7. The `finally` block is compiled as code appended to `try` and `catch` blocks.

In the compilation format of subroutine as shown in Figure 6-6, an instruction `jsr` is added at the end of `try` block and `catch` blocks (if there is any) in order to transfer the control to the code implementing the `finally` block if the execution of the `try` or `catch` block ends normally (edges 3 and 6 in Figure 6-2). In more detail, the subroutine call works as follows: The instruction `jsr` (at indices 10 and 22 in Figure 6-6) pushes the address of the following instruction onto the operand stack before jumping. The first instruction (`astore` 6 at index 36 in Figure 6-6) in the code implementing the `finally` block stores the address on the operand stack into local registers. The following code (instructions from index 38 to 42 in Figure 6-6) is run. Assuming the execution of the `finally` block completes normally, the instruction `ret` (at index 42 in Figure 6-6) at the bottom of the subroutine retrieves the address stored to local registers and resumes execution at that address.

Furthermore in order to deal with the exceptional exit of the `try` and `catch` blocks (edges 4 and 7 in Figure 6-2), the compiler generates one exception table entry for each `try` or `catch` block with the handler type *any*, which can handle any type of exceptions thrown with the scope of `try` or `catch` block. (Usually such a entry is also generated for the `finally` block itself.) When the `try` or `catch` block throws an exception and the matching `catch` block is not found, the exception table entries for the `finally` block is searched for an appropriate exception handler. Then the control is transferred to the instruction (at index 28 in Figure 6-6) indicated by the column *target*. After storing the reference value of the thrown exception to local registers (the instruction `astore` 5 at index 28 in Figure 6-6), the following instruction `jsr` does a subroutine call to the code implementing the `finally` block. Assuming that code returns normally, the reference value of the thrown exception is pushed back to the operand stack (the instruction `aload` 5 at index 35 in Figure 6-6) and re-thrown by the following instruction `athrow` (at index 35 in Figure 6-6).

In the other compilation format, the code implementing the `finally` block is appended to the `try` block and `catch` blocks (if there is any) as shown in Figure 6-7. If the execution exits the `try` or `catch` block normally (edges 3 and 6 in Figure 6-2), the execution continues to work on the following code, which implements the `finally` block and is appended to the `try` or `catch` block by the compiler. In Figure 6-7, the code appended to the `try` block is the instructions between [72, 110] and the code appended to the `catch` block is those between [37, 3]. For the exceptional exit of to the `try` or `catch` block, the compiler generates one exception table entry for each `try` or `catch` block with the handler type *any* just like the compilation in the subroutine format. The instructions from the index indicated by the column *target* do similar work as those in the subroutine format except that the subroutine call is replaced by executing the code implementing the `finally` block directly.

6.3.2 Locating the `try` Block

Since the scope of the `try` block is indicated by the columns *from* and *to* in exception table entries, it is easier to locate the scope of the `try` block compared with locating the scope of the `catch` or `finally` block. What we should pay attention to is that the compiler generates exception table entries for `catch` blocks if the *try* statement contains a `finally` block and such entries should be excluded.

If the `catch` block is specified in a *try* statement, the compiler generates a exception table entry with a particular handler type for each `catch` block. Thus if there are one or more entries whose handler type is not *any*, the `catch` block is specified and the scope of the `try` block is indicated by the index pair [*from*, *to*) in those entries. Each different pair corresponds to one `try` block in a *try* statement. For example, the scope the `try` block of in Figure 6-6 is [0, 10) and that in Figure 6-7 is [0, 12).

If no `catch` block is specified in a *try* statement, all the entries in the exception table are generated for the `finally` block by the compiler and the handler types of all entries should be *any*. These entries are generated to deal with the exception thrown either in the `try` block or the `finally` block. Since the entries generated for the latter case should have equal values in columns *from* and *target*, the scope of the `try` block is indicated by the index pair [*from*, *to*) in the entry whose column *from*'s value is not equal to column *target*'s value.

6.3.3 Locating the `catch` Block

If the `catch` block is specified in a *try* statement, the compiler generates an exception table entry for each `catch` block, in which the beginning index of the `catch` block is indicated by the column *target*. What we need to do is to locate the ending index of the `catch` block.

If no `finally` block is specified in the *try* statement, the control is transferred to the immediate post-dominator instruction of the *try* statement after the execution exits the `try` block normally. Thus the immediate post-dominator instruction of the `try` block, which is indicated by the column *to* in the exception table, should be the instruction `goto` if there are instructions left to be executed in the method or the instruction `return` (α`return`) if the end of the *try* statement is the bottom of the method. In the former case the end index of the last `catch` block is indicated by the branch index of the instruction `goto`, and in the latter case the end index is the index of the last instruction in the method. The end of other `catch` blocks (if there are more than one `catch` block in the *try* statement) can be located by the beginning index of the next `catch` block since the `catch` blocks are compiled into successive bytecode. For example the in bytecode shown in Figure 6-9, the scope of the first `catch` block is [6, 14) indicated by values in the column *target* and the scope of the last `catch` block is [14, 19) indicated by the value in the column *target* and the branch index of the instruction `goto` at index 3 which is the immediate post-dominator of the `try` block.

If the `finally` block is specified in the *try* statement, the compiler generates for `finally` block one exception table entry to deal with the exception thrown in the `try` block that the `catch` block cannot deal with, one entry to deal with the exception thrown in each `catch` block and one entry to deal with the exception thrown in the `finally` block itself. Thus the scopes of the `catch` blocks are indicated by the columns *from* and *to* in those entries to deal with exception thrown in `catch` blocks, just like the scope of the `try` block is indicated by the entry generated for the `catch` block. The entry generated to deal with exception in the `try` block can be found by compare the value of columns *from* and *to* with the scope of the `try` block, and the one generated to deal with exception in the `finally` block has equal values in the columns *from* and *target*. Excluding those entries, the left ones are generated to deal with the exception in the `catch` block and can be used to locate the scope of the `catch` block by the index pair [*from*, *to*). For example in the bytecode shown in Figure 6-7, the third exception table entry is generated for the `finally` block to deal with the exception in the `catch` block, and the scope of the `catch` block is [19, 25) indicated by the columns *from* and *to*.

6.3.3 Locating the `finally` Block

Locating the scopes of the `catch` and `finally` blocks is complicated because of the `finally` block’s compilation. The compiler generates an exception table entry for each `catch` block and one or more entries with type `any` for the `finally` block. The exception table entry with handler type *any* is generated and can 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 this *try* statement.

In the case of the `finally` block being compiled into subroutine, the last instruction in the `try` block (whose index is indicated by the value of the column *to* in the exception table entry) should be the instruction `jsr` *i*. The instruction `jsr` *i* transfers the control to the subroutine compiled from the `finally` block, whose starting index is *i*. Assuming the index of the instruction `ret` in the subroutine is *i*', the scope of the `finally` block is (*i*, *i*'). As for locating the scope of the `catch` block, if there is any, the index of the first instruction in the `catch` block is indicated by the column *target* in the exception table entry whose handler type is not *any*. At the end of the `catch` block there should be one instruction `jsr` transferring the control to the subroutine of the `finally` block. Since the scope of the `finally` block has been located as (*i*, *i*'), the end of the `catch` block is the first `jsr` *i* post-dominating the first instruction of the `catch` block. Assuming the value of the column *Target* in the exception table entry indicating the `catch` block is *k* and the index of the instruction `jsr` *i* transferring the control to `finally` block is *k*', the scope of the `catch` block is [*k*, *k*'].

For example, in Figure 6-5 the `finally` block is specified because there are 3 entries with the handler type *any* in the exception table. The first entry in the exception table has a handler type of `java/lang.NullPointerException` and indicates the scope of the `try` block as [0, 10] by the columns *from* and *to*. Since the last instruction in the `try` block is `jsr` 36 at index 10, the subroutine complied from the `finally` block starts from the instruction at index 36. And the index of the instruction `ret` is 42, thus the scope of the `finally` block is (36, 42). As the scope of the `catch` block, the starting index had been indicated as 16 by the column *target* in the first entry of the exception table. Since the instruction transferring control to the `finally` block is `jsr` 36, the first such instruction post-dominating the instruction at index 16 (the beginning of the `catch` block) is the instruction at index 22. Thus the scope of the `catch` block is [16, 22].

In the other case, the `finally` block is compiled into the code appended to the `try` and `catch` blocks respectively. That is, the same code compiled from the `finally` block will appear after each `try` and `catch` blocks, and the last instruction in the `try` block could not be the instruction `jsr`. To divide the `try` block from the following `catch` or `finally` blocks, the complier generates the instruction `return` (or α`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*'].

As for locating the scope of the `catch` block, 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 found instruction blocks is the `finally` block appended to the `catch` block. If there are any entries whose handler types are not *any* in the exception table, the `catch` block is specified. The locating of the `catch` block's the scope is based on the scope of the `finally` block. 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 Figure 6-6 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 Figure 6-7 the scope of the `try` block is [0, 12], the scope is of the `finally` block is [12, 16], [25, 26] and [34, 35] respectively, and the scope of the `catch` block is [19, 25).

6.4 Implicit Information Flow in Exception Handling

In Java bytecode, exceptions can be thrown explicitly by the instruction *athrow* or implicitly by some specific instructions such as those shown in Figure 6-3. For the exceptions raised implicitly by one Java bytecode instruction, whether the exception occurs depends on the values of the variables operated in the instruction. In other words, the occurrence of one exception carries the information of the data affecting the exception's generation. Therefore it is reasonable to assign a security-level to each exception. We define the security-level of one exception as the LUB of security-levels of the data 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 data in the variable *arrayref*.

Considering that exception handling will not only cause 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 when it is handled in the caller method.

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 conditional branch node and it may have some of the branches shown in Figure 6-2. 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 may be raised by the PEI (or the LUB of the security-levels of all exceptions that may be raised by the PEI). To distinguish it from the conditional security-level of *if*-instructions, we call that 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 possibly 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 (we call such one PEI as single-exception PEI) and the others can throw two or more types of exception (we call such one PEI as multiple-exception PEI). Thus for the exceptions that can be raised by multiple-exception PEIs, there are three kinds of exception handling: 1) all of they may be handled in the same method where they are raised (only intra-procedural transferring may be caused); 2) none of they may be handled in the same method where they are raised (only inter-procedural transferring may be caused); or 3) some of they may be handled and the others can not be handled in the same method where they are raised (both intra-procedural and inter-procedural information transferring may be caused). 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 had 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` blocks 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 6-2. The scope of the normal branch (1 or 3 in Figure 6-2) 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 6-2) 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 6-2. The scope of the normal branch (1 or 3 in Figure 6-2) 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

6-2) 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 exit the method. If there is one `finally` block specified in the *try* statement where the exception is raised, we can insert 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 field *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 start 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 calculated just before the PEI. As for the exceptional branch, we should backup 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 field *SL* of the exception. At the end of the exceptional branch we should restore the SL_c to the original one.

- **Both kinds of 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 cannot 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 6-2. The scope of the normal branch (1 or 3 in Figure 6-2) 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 6-2) is from the start 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 do in case of inter-procedural implicit information transferring. As for 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.

- **Procedure of Dealing with Implicit Information Flow Caused by PEIs**

Based on the analysis above, we can 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.

a Locate the scopes of all the `try` blocks, `finally` blocks and `catch` blocks in *m*.

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

c 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.

d If any PEIs in one `try` block is inter-procedural PEIs, at start 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 start of the `finally` block.

e At the start 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.

f 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.

g 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.

Here we give an example. Consider the section of Java program whose bytecode and CFG are shown in Figure 6-8. Using the procedure above we could deal with implicit transferring caused by PEIs in that Java bytecode. We give the modified bytecode in Figure 6-9. 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 Figure 6-8. The PEIs are `arraylength` at index 1 and `iaload` at the index 5 in Figure 6-8. (Here to simplify the example we assume that the instruction invokevirtual itself will not raise any exception.). We insert instructions at the address 2, 3 and 14-20 in Figure 6-9 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 at the address 71-83 in Figure 6-9 to the `finally` blocks to set the correct current environment security-level. At the start of the `catch` block we insert instructions at the address 44-55 in Figure 6-9 to check whether the field *SL* of exception instance caught is 0 and set the current environment security-level to the correct value.

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

```
void testOfcatch(int[],int);
Code:
  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: astore5
 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   12    32    any
  19   25    32    any
  32   34    32    any
```

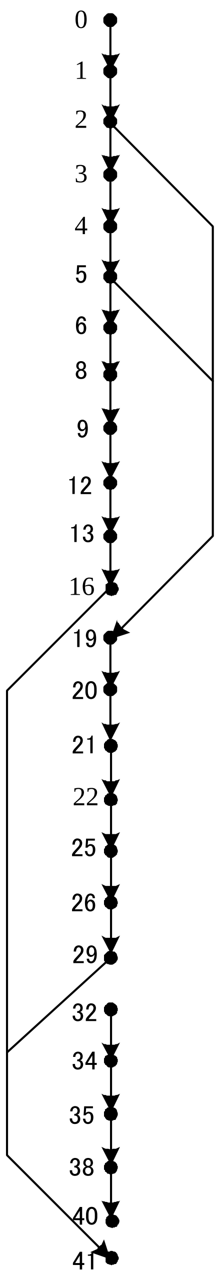


Figure 6-8. An example of implicit information transferring caused by PEIs.

```
void testOfcatch(int[],
byte[], int, byte);
Code:
  0: aload_1
  1: iload_3
  2: dup
  3: istore 13
  5: istore 7
  7: arraylength
  8: istore 6
 10: aload_1
 11: aload_2
 12: iload 4
 14: iload 5
 16: iload_3
 17: jsr 80
 20: istore 13
 22: swap
 23: iload 4
 25: iaload
 26: istore 9
 28: iaload
 29: istore 8
 31: aload_0
 32: invokevirtual #5
 35: aload_0
 36: invokevirtual #6
 39: goto 54
 42: astore 6
 44: aload 6
 46: getfield #9;
     //Field SL; B
 49: dup
 50: ifne 36
 53: iload 13
 55: istore 12
 57: aload_0
 58: aload_0
 59: invokevirtual #8
 62: aload_0
 63: invokevirtual #6
 66: goto 27
 69: astore 10
 71: aload 10
 73: getfield #9;
     //Field SL; B
 76: ifne 10
 79: aload 10
 81: iload 12
 83: putsield #9;
     // Field SL; B
 86: aload_0
 87: invokevirtual #6
 90: aload 10
 92: athrow
 93: return

 95: istore 14
 97: dup2
 98: if_icmple 86
101: istore 15
103: pop
104: istore 15
106: goto 68
109: pop
110: ret 14

 Exception table:
 from   to   target   type
    0   25     30     Class jave/lang/NullPointerException
    0   25     45     any
   30   49     45     any
   45   52     45     any
```

Figure 6-9. The modified bytecode of that shown in Figure 6-8.

6.5 Explicit Information Flow in Exception Handling

In Java bytecode the exception can be thrown by the instruction `athrow` explicitly. The instruction `athrow` transfers the control from it to the point where the exception thrown by it can be handled, and thus 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 depending on where the exception thrown. But different from PEIs, the instruction `athrow` causing explicit information transferring acts as an unconditional control-trsferring instruction `goto` in the information flow. The effect of the instruction `athrow` on the information flow is that the environment security-level of the block where the instruction `athrow` throws an exception is transferred to the block where the exception is handled. Therefore it is quite simple to deal with the explicit information transferring caused by the instruction `athrow`. What we should do is to set the field *SL* of the exception instance that will be thrown by the `athrow` to the current environment security-level SL_c. As for the block where the exception is handled, we compare the current environment security-level with the security-level in the field *SL* of the exception instance and updates to the current environment security-level to the value of the higher one, which is just like we do for implicit information flow at the handler block.

7 Implementation and Evaluation

7.1 System Architecture

A prototype verification system implementing the approach described in this thesis has been developed, and it is named as BMOS (Bytecode MOdification System). This system is written in Java language so as to be adopted by various mobile systems. The input of BMOS is a class file in memory. Modified by the modifier, the bytecode containing verification code is delivered to the local runtime platform. During the execution of the bytecode, an information-leak exception will be thrown if there is any violation of host security policies. In this way, the verification system interrupts the process causing information-leak to protect the local host security. Of course, the user can choose to ignore the exception in order to make the bytecode finish its job.

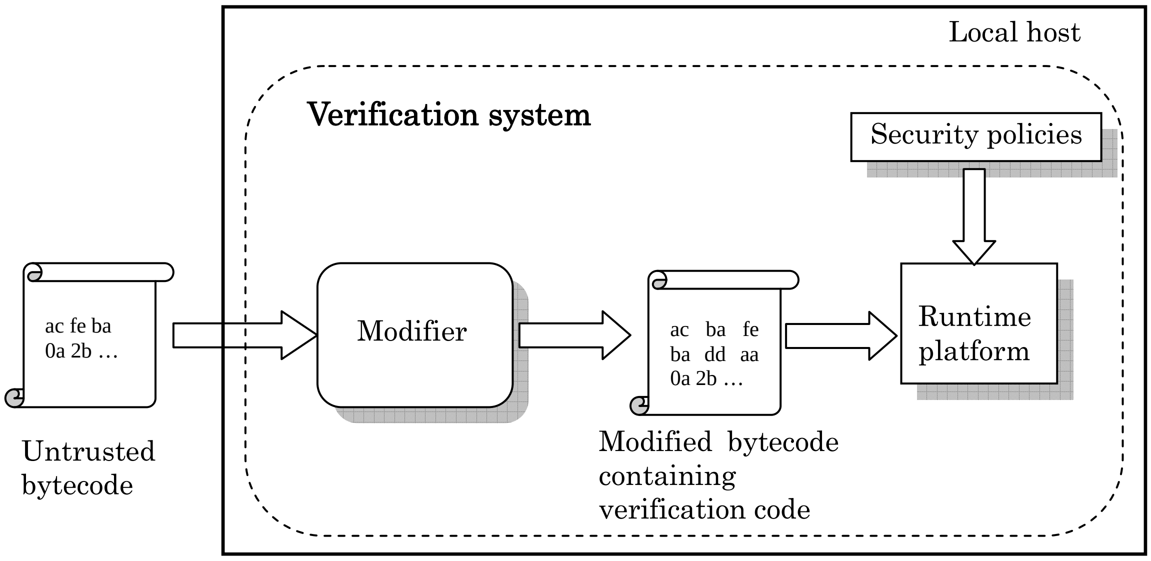


Figure 7-1. BMOS architecture.

The BMOS is built using the techniques described in this thesis. As shown in Figure 7-1, the system consists of two main parts: the security policies and the modifier. The security policies define the security requirement of the local host. These policies are described by the security-levels of resources and the clearance-levels of third-party hosts, and stored in a configuration file. The core of the system is the modifier, which redefines the class and inserts the verification code for the dynamic verification during the execution.

7.1.1 Security policies

In BMOS, the security policies are described by the security-levels and the clearance-levels.

A security-level defines a host file's 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, and all the files in one directory get the security-level of the directory.

Security-level definition:

SL.level=res 1, res 2, …

e.g.   SL.3=/home/temp, /home/usr/

A clearance-level defines 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. The clearance-level is assigned to an observer-host according to its network address.

Clearance-level definition:

CL.level= res1, res2,…

e.g.   CL.2=www.abc.com, ftp.xy.com, 201.118.23.234

7.1.2 Modifier

The modifier is the core part of this system. It performs the modification of bytecode used for the dynamic verification during the execution. The modification can be divided roughly into three steps. The first step is to parse the class file from bytecode to the instance of `ClassInfo` class defined beforehand. The second step is class-level modifications and method-level modifications, including adding fields, rewriting the method-descriptors, adding local variables, changing the stack size and inserting verification instructions, etc. The final step is to generate the bytecode of the modified class. In Figure 7-2, we give the process steps of the modifier.

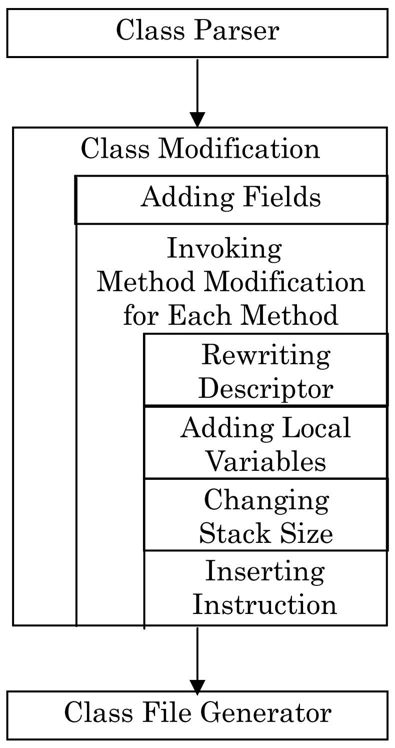


Figure 7-2. Processing steps of Modifier.

ClassModifier.java

```
public ClassModifier(File classFile)
      throws FileNotFoundException, IOException{

  classInfo = new ClassInfo(classFile);
  cp = classInfo.getConstantPool();//Constant Pool
  indexOfByteInCP = cp.getUtf8(String.valueOf
      (TypeDiscriptorParser.BYTE)).getIndex();

}

public void modifyClass(){

  //adding fields as Security-Level Container for the original fields
  addFields();

  //modify each method of the ClassFile
  modifyMethods();

}
```

Figure 7-3. The Program of Class-level Modification.

MethodModifier.java

```
public void modifyMethod() {

  // Rewrite the descriptor: add parameters as Security-Level Container
  // for original, and modify the return type
  addParameters();

  // add local variables as Security-Level Container for original
  addLocalVariable();

  // add stack as Security-Level Container for original
  addStack();

  // add verification code
  initiaInstruction();
  addVerificationCode();
}
```

Figure 7-4. The Program of Method-level Modification.

We illustrate part of the source code of the modifier in Figure 7-3 and 7-4. In the constructor of class `ClassModifier`, the class file to be modified is parsed to an instance of class `ClassInfo`. The method `modifyClass` of class `ClassModifier` performs the class-level modification. And in the method `modifyMethods`, one instance of class `MethodModifier` is generated for each method in the class file, and the method `modifyMethod` shown in Figure 7-4 is invoked to perform the method-level modification.

7.2 Implementation

7.2.1 Class Parser

In order to analyze and modify the class file, the class file is transformed to the format to meet our needs. This process is implemented by a Class Parser, which reads information from the class file and then converts it to instances of classes defined beforehand. Some tools, such like Bytecode Engineering Library (BCEL) and Java Object Instrument Environment (JOIE), have been developed to implement such class parsing. But to implement our approach, we need not only parse the class file to instances, but also modify the instances and regenerate the class file. Thus we adapted the JOIE to meet our need of class file modification.

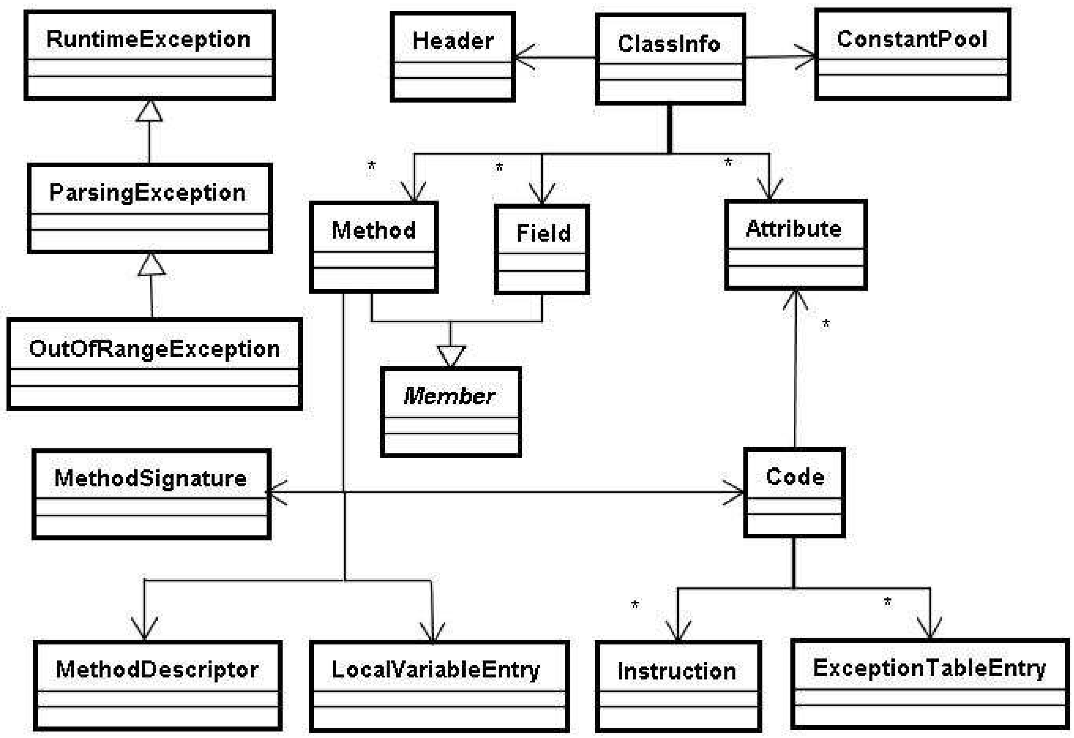


Figure 7-5. UML diagram of the classes representing class file.

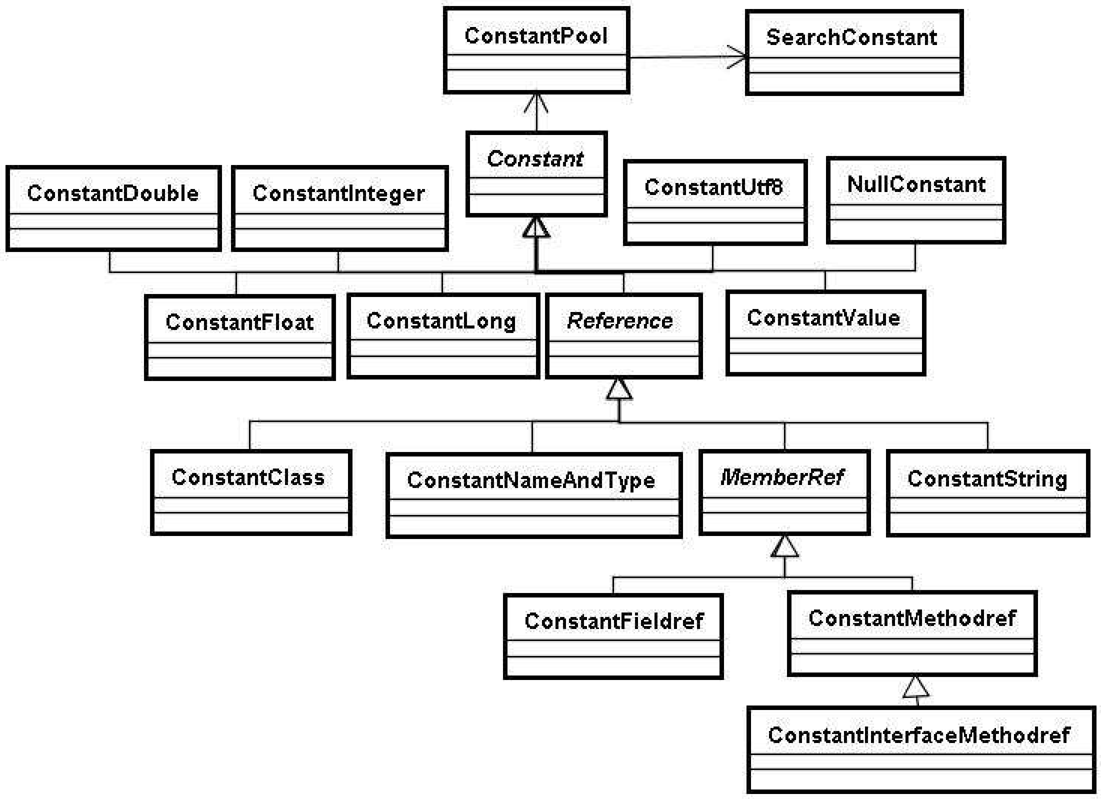


Figure 7-6. UML diagram of the classes representing the constant pool.

JOIE is a framework for safe Java bytecode transformation. It provides both low-level and high-level functionality to extend or adapt compiled Java classes. The low-level interface allows manipulating the bytecodes itself whereas the high-level interface provides methods for inserting new interfaces, fields, methods or whole code splices. In order to modify the class file more easily, we ameliorate some classes of JOIE. Figure 7-5 shows the UML diagram of the classes used to represent the class file, and Figure 7-6 shows the UML diagram of the classes used to represent the information in the constant pool.

The input of the class parser should be one class file in the format of bytecode file or `InputStream`. The constructor of class `ClassInfo` will invoke proper methods to read information of the class file from bytecode or `InputStream`, generate instances of necessary classes (such as `ConstantPool`, `Field`, `Method`, `Code`, `Instruction` and so on) and construct the instance of class `ClassInfo` containing all information of the class file. For example, the class file “c:\workspace\bin\bms\Tester.class” can be parsed by the following code:

```
File file = new File(“c:¥workspace¥bin¥bms¥Tester.class”);
ClassInfo ci = new ClassInfo(file);
```

As we get `ClassInfo` instance of one class file, we can get all information we need about the class file. For example, the methods and fields of the class file “Tester.class” can be obtained by the following code:

```
Method[] methods = ci.getMethods();
Field[] fields = ci.getFields();
```

For an instance of class `Method`, the information necessary for modification, such as descriptor, code, local variables and instructions can be obtained easily as following code:

```
Code code = method.getCode();
LocalVariableTable lv= code.getLocalVariableTable();
CodeIterator iter = code.getSplice().getCodeIterator();
```

As for the Java bytecode instructions, a base class `Instruction` represents a single JVM instruction. Some subclasses, such as `Load`, `Branch`, are defined to represent the instructions performing the similar operation. Such subclasses hide subtle distinctions among different forms of the same instruction. For example, the JVM specification defines fifty separate bytecode forms that can load or store a value, depending on the size and type of the value and its location in the frame. Unified `Load` and `Store` instruction classes can generate the correct bytecode form for the operands. Other subclasses of Instruction represent the few instructions with multiple or a variable number of operands, including table switches, multidimensional object array creation, and interface method invocation.

Instruction operands and arguments are represented as logical references to other objects rather than as numeric offsets into tables. For example, a `Branch` instruction instance contains a reference to the target `Instruction` instance, rather than a byte offset. Also references to methods are represented by instances of the class `Methodref`, instead of as raw integers as in the byte stream interface. To achieve this, a class `Label` is generated to represent the offset of branch instruction, such as `goto`, `if_icmpne`. And the size of instruction Label is defined as 0 in order to cause no side effect on original bytecode.

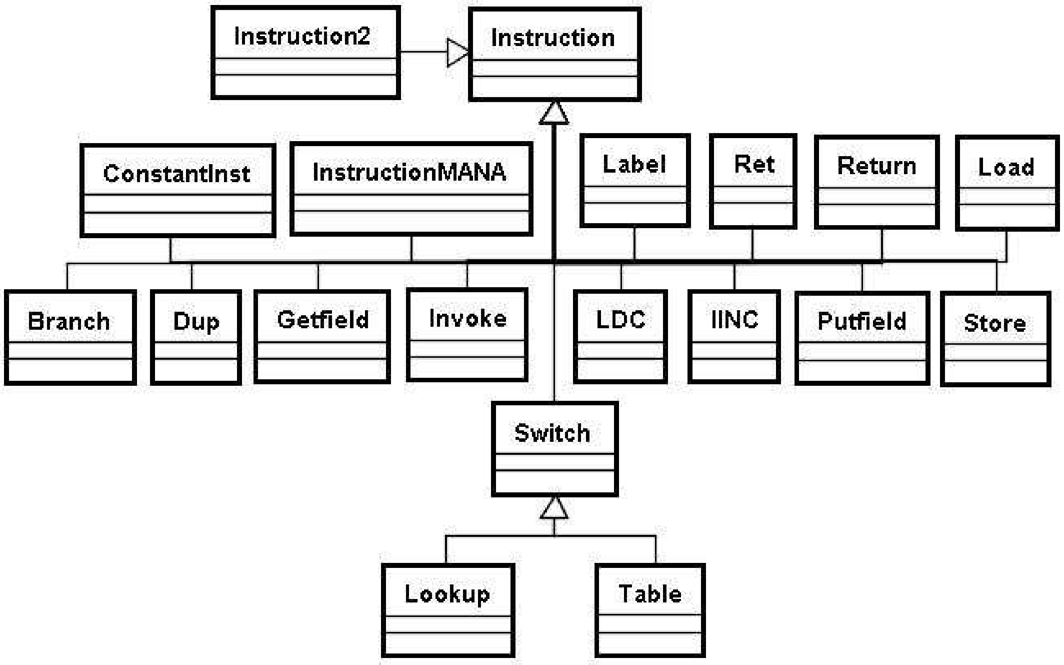


Figure 7-7. UML diagram of the classes representing the bytecode instructions.

Instruction classes contain logic to preserve referential integrity across changes to the class file. For example, a Branch instruction automatically updates its offset field if new instructions appear between the branch and its target. Figure 7-7 shows the hierarchy of the classes representing the bytecode instructions.

7.2.2 Class Modification

The jobs performed at class-level modification are adding new fields as security-level container for the original fields and the class itself, and invoking method-level modification for each method in the class.

For the original field of primary types, we add one field of type `byte`. For the original field of an array for primary types, we add one field of `byte` array with the same dimension as the original one. And for the field of classes or array for classes, we add one field of the same type as the original one. Beside these, we also add one field of type `byte` as the security-level container for the class itself. 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 of Java class file, any new fields and entries could be appended rather than inserted in the middle in order to preserve the indices of existing entries.

In order to judge the type of the field and thus add new field of proper type, we generated a class `TypeParser` to parse the method's descriptor to an array of variables' descriptors, and judge the type that one variable's descriptor represents. Figure 7-8 shows part of the program of adding fields.

```
Field[] fields = classInfo.getFields();//All original fields

if (fields != null && fields.length > 0){
   for (int i=0; i < fields.length; i++){
      Field field = fields[i];
      String descriptor = field.getDescriptor();

      int typeOfFiled = TypeParser.judgeType(descriptor);

      //Add new field
      Field newField = null;
      String fieldName = field.getName();
      String newFieldName = fieldName+ TypeParser.SL_SUFFIX;
```

```
        indexOfName = cp.getUtf8(newFieldName).getIndex();
        int indexOfDescriptor;
        if (typeOfFiled == TypeParser.WRONG_TYPE){
           throw new IllegalArgumentException();
        }else if (typeOfFiled == TypeParser.PRIMARY_TYPE){//primary
           indexOfDescriptor = indexOfByteInCP;
           newField = new Field(newFieldName, Type.BYTE);
        }else if (typeOfFiled%10 == TypeParser.PRIMARY_TYPE){//array
           ArrayType arrayType =
              new ArrayType(Type.BYTE, typeOfFiled/10);
           indexOfDescriptor =
              cp.getUtf8(arrayType.getDescriptor()).getIndex();
           newField = new Field(newFieldName, arrayType);
        }else {//class or array of class
           newField = new Field(newFieldName, field.getType());
           indexOfDescriptor = cp.getUtf8(descriptor).getIndex();
        }

        newField.setDesc_index(indexOfDescriptor);
        newField.setName_index(indexOfName);
        newField.setClassInfo(classInfo);
        classInfo.addField(newField);
    }
}
```

Figure 7-8. Part of the program of adding fields.

After the fields adding, the method-level modification should be invoked for each method in the class file, which is performed by the method `modifyMethods` shown in Figure 7-9.

```
private void modifyMethods(){
    //all the methods of the Class file
    Method[] methods = classInfo.getMethods();

    if (methods != null && methods.length > 0){
        for(int i = 0; i < methods.length; i++){
             MethodModifier mm = new MethodModifier(methods[i], cp);
             mm.modifyMethod();
        }
    }
}
```

Figure 7-9. The program of invoking method-level modification.

7.2.3 Method Modification

Method-level modification is the main part of the whole modification. The tasks performed here are adding security-level containers for parameters, modifying the return type, adding security-level containers for local variables, increasing the stack size, locating the implicit blocks' scopes, and the most important one, inserting the verification instructions.

- Modifying method's descriptor

  Adding security-level containers for parameters means to modify the descriptor of the method, that is, to insert the descriptor of the new parameter into the method's original descriptor. In order to keep the alternate order of variables and their security-level containers, we insert the descriptor of the new parameter just after the descriptor of the original one. The rules used to decide types of new parameters are the same as the ones used when we add new fields.

```
if (numOfParams > 0) {
    for (int i = 0; i < numOfParams; i++) {
        String paramType = paraRetTypes [i];
        newDescriptor.append(paramType);
        int type = TypeParser.judgeType(paramType);

        if (type == TypeParser.WRONG_TYPE){
            throw new IllegalArgumentException();
        }else if (type == TypeParser.PRIMARY_TYPE){
            newDescriptor.append(TypeParser.BYTE);
        }else if (type % 10 == TypeParser.PRIMARY_TYPE){//array
            newDescriptor.append(paramType.substring(0, type / 10)
                + TypeParser.BYTE);
        }else {//class or array of class
            newDescriptor.append(paramType);
        }
    }
}
// modify the return type if it is not Void;
String returnType = paraRetTypes[numOfParams];
int type = TypeParser.judgeType(returnType);
if (TypeParser.VOID_TYPE == type) {
    newDescriptor.append(returnType);
} else {
    newDescriptor.append('[');
    newDescriptor.append(returnType);
}
// Set new Descriptor to Constant Pool
int constantIndex = method.getDesc_index();
cp.getUtf8(constantIndex).setString(newDescriptor.toString());
```

Figure 7-10. Part of the program of modifying the method's descriptor.

The return type of one method is also defined in the method's descriptor. In order to return the security-level of the result, we assemble an array by the result and its security-level and return the array as the execution result of the method. At Figure 7-10 shows part of the program modifying the method's descriptor.

- Adding local variables

  Adding security-level containers for local variables is almost same as adding security-level containers for parameter on theory. But they are quite different in implementation and the former is much more difficult than the later. All parameters of one method are defined in one item in the constant pool, while the local variables are defined in the `LocalVariableTable` attribute separately. To add a local variable, we need to define its available scope in instructions, its name index and descriptor in constant pool, and its index in all local variables. Similar to parameters, in order to keep the alternate order of variables and their security-levels, we insert new local variables into the original ones rather than append them. Thus we also need to recalculate the index of all local variables and the operands of the instructions using local variables. Figure 7-11 shows part of the program adding local variables and recalculating the index.

```
if (lvType == TypeParser.PRIMARY_TYPE) {//primary type
    descriptorIndex = indexOfByteInCP;
}else if(lvType%10 == TypeParser.PRIMARY_TYPE){//array
    String descriptor = lve.getDescriptor()
        .substring(0,lvType/10)+TypeParser.BYTE;
    descriptorIndex = cp.getUtf8(descriptor).getIndex();
} else {//class or array of class
    descriptorIndex = cp.getUtf8(lve.getDescriptor()).getIndex();
}

LocalVariableEntry newLve = new LocalVariableEntry(lve.getStart(),
    lve.getLength(), nameIndex, descriptorIndex, lvPositon + 1);
newLve.setStart_inst(lve.getStart_inst());
newLve.setEnd_inst(lve.getEnd_inst());
newLve.setCpool(cp);
lvEntries.add(lvPositon + 1, newLve);

// Modify the frame index of the original lv
lve.setFrame_index(lvPositon);

// Deal with the LocalVarible
code.addLocalAt(lvPositon + 1, lvPositon + 1);
```

Figure 7-11. Part of the program of adding local variables.

- Increasing stack size

For the operands on the stack, we also need to keep the variables and their security-levels in alternate order. This order is kept by using correct instructions to push and pop operands to and from stack. We need not do anything to the stack except double the size of the stack, since we push the security-level of each operand to the stack.

- Locating implicit blocks' scopes

  In order to calculate the correct security-levels of variables, we need to know which instructions are in the implicit blocks and what are the environment security-levels of those implicit blocks. As we have discussed in Chapter 4, the *if*-instructions and *switch* instructions can cause implicit blocks. And we also give the algorithm to locate the scope of implicit blocks, including nest ones.

  To implement the algorithms, we use one variable `impBlkScps` of type `Stack` to manage the scopes of implicit blocks. We also use two arrays of type `int` to represent the implicit block and all the instructions' implicit-block layer-number as shown in Figure 7-12. We loop all instructions of the method to find the *if*-instructions and `switch`. Before we locate the scopes of the implicit blocks caused by one if-instruction or `Switch`, we need to decide the current implicit block, that is, the instruction is in any outer implicit blocks. We peek the scope on the top from `impBlkScps`. If the address of current instruction is in the scope, the scope of the current implicit block is the one just peeked. Or the loop has exited the scope, and we pop the implicit block since it will not be used any more.

```
int[] impBlkScope = { startAddr, endAddr, LayerNo };
```

The first element represents the start address, the second element represents the end address, and the third element represents the layer number.

```
int[] instrLayerNo = new int [maxAddr]
```

maxAddr is the max address of instructions in the method.

Figure 7-12. The arrays representing the implicit block and instructions' layers.

When the loop encounters one *if*-instruction or `switch`, we located the scopes of implicit blocks according to the algorithms in Chapter 4. Then we push the scopes to the variable `impBlkScps` ascending (at first push the scope of the block appearing later in the instruction sequence, then the scope of the one appearing earlier). At last we set the implicit-block layer-number of the instructions in one implicit block to its layer number. Figure 7-13 shows part of the program of deciding current implicit block and locating the scopes of implicit blocks.

```
while (iter.hasNext()) {
   Instruction instr = iter.nextInstruction();
   int addr = instr.getAddr();

   // find the current implicit scope
   while (!impBlkScps.isEmpty()) {
      nextImpBlkScp = (int[]) impBlkScps.peek();
      if (addr <= nextImpBlkScp[1] && addr >= nextImpBlkScp[0]) {
         curImpBlkScp = nextImpBlkScp;
         break;
      } else {// exit nextImpBlk
         curImpBlkScp = null;
         impBlkScps.pop();
      }
   }

   if (curImpBlkScp != null) {
      outerBlkStart = curImpBlkScp[0];
      outerBlkEnd = curImpBlkScp[1];
      curLayerNo = curImpBlkScp[2];
   }else{
      outerBlkStart = 0;
      outerBlkEnd = maxAddr;
      curLayerNo = 0;
   }

   // if-instruction
   if (Instruction.isIfInstruction(instr)) {
      int addr = instr.getAddr();// I
      int op = instr.getOp();// j

      if (op < 0) {// loop
         int[] tempScope = { addr + op, addr - 1 };
         pushImpScp(tempScope);
      } else {
         Instruction instrBfreTar =
            splice.getPrevious(instr.getTarget());
         int addr2 = instrBfreTar.getAddr();// i'

         if (instrBfreTar.getOpcode() == Opcode.GOTO
              || instrBfreTar.getOpcode() == Opcode.GOTO_W) {
            int op2 = instrBfreTar.getOpcode();// j'
            if (op2 < 0) {// loop
               int[] tempScope={ addr + 1, addr2 - 1, curLayerNo + 1};
            pushImpScp(tempScope);
            } else {// two branches, common if-instruction
```

```
                int[]tempScope2={addr+op,addr2+op2-1,curLayerNo+1};
                pushImpScp(tempScope2);
                int[] tempScope1={addr+1,addr2 - 1, curLayerNo + 1 };
                pushImpScp(tempScope1);
            }
        } else if (instrBfreTar.getOpcode() == Opcode.ARETURN
            || instrBfreTar.getOpcode() == Opcode.RETURN) {
            int[] tempScope2={addr + op, maxAddr - 1, curLayerNo + 1};
            pushImpScp(tempScope2);
            int[] tempScope1 = { addr + 1, addr2 - 1, curLayerNo + 1 };
            pushImpScp(tempScope1);
        } else {
            int[] tempScope={addr + 1, addr + op - 1, curLayerNo + 1 };
            pushImpScp(tempScope);
        }
    }...
  }else if (instr.getCategory() == ByteCode.CAT_SWITCH) {
  ...
  ...
  ...
  }
}
```

Figure 7-13. Part of the program locating implicit blocks.

- Tracing information flow

Inserting instructions to trace the information flow is the most important part of the method-level modification. What we should do here is to insert proper instructions to implement that loading security-level to stack, calculating LUB of security-level and store the security-level back to local variables or fields. During the operation be the inserted instruction, we should keep two rules: 1. When the JVM executes one original bytecode instruction, the operands used by the instruction on the stack should be laid as if no instruction is inserted, which assure that the inserted instructions has no side affect on the original function of the bytecode; 2. When the JVM executed the original instruction and the instructions inserted for it, the operands on the stack should keep the alternate order of variables and their security-levels, which assure that we may insert instructions for single original instruction and need not to consider the context.

To keep the two rules, we load the security-level after the variable, and store the security-level before the variable. As for the instruction operating two or more operands, we add several local variables and use them as temporary containers for security-level or variables when we arrange the operands on the stack, and insert a little complicate instruction to keep the two rules above. Figure 7-14 shows part of the program inserts instructions for loading security-levels and calculating security-levels.

```
private void insertCodeForCompare(Instruction instr){
   splice.insertBefore(instr, getCodeForESL());
   splice.insertAfter(instr, loadTempSL);
}

private void insertCodeForLoad(Load instr){
   Instruction load = null;
   int lvNo = instr.getOperandIndex();
   if (instr.getOpcode() == Opcode.ALOAD){
      load = new Load(Opcode.ALOAD, lvNo + 1 , code);
   }else{
      load = new Load(Opcode.ILOAD, lvNo + 1 , code);
   }
   splice.insertAfter(instr, load);
   iter.forward(1);
}

private Splice getCodeForESL(){
   Splice splice = new Splice();
   //Store the second operand to temp-Operand local variable
   int operandType = ((Integer)operandTypes.peek()).intValue();
   int index = indexOfTempOperand[operandType] ;
   short formOp = (short)(Opcode.ISTORE + operandType);
   Instruction instrStoreOp = new Store(formOp, index, code);
   splice.append(instrStoreOp);
   //Load  the second operand from temp-Operand local variable
   Instruction instrLoadOp =
      new Load((short)(formOp - 33), index, code);

   //insert instructions
   splice.append(storeTempSL);
   splice.append(instrStoreOp);
   splice.append(loadTempSL);
   splice.append(spliceOfLUB);
   splice.append(instrLoadOp);
   return splice;
}
```

Figure 7-14. Part of the program inserting instructions.

Furthermore, we add a local variable of type int[] to store the environment security-level of each implicit block layer. So that we can know correct environment security-level every original instruction according to the implicit-block layer-number of the instruction. Thus we also need to insert instructions to manage the environment security-level’s array. Figure 7-15 shows an example of calculating the environment security-level and storing it.

```
/**
  * insert verification code for switch-instruction
  *
  * @param instr
  */

public void insertInstrForSwitch(Switch instr) {

    splice.insertBefore(instr, storeTempSL);

    // calculate the LUB of ESL and outer layer ESL
    int layerNo = instrLayerNo[instr.getAddr()];
    if (layerNo > 0) {
        Splice spliceESL = new Splice();
        Instruction instrPushIndex = new ConstantInst(layerNo - 1);
        Instruction instrLoadESL = new Load(Opcode.BALOAD, code);
        spliceESL.append(loadTempSL);
        spliceESL.append(loadESLsRef);
        spliceESL.append(instrPushIndex);
        spliceESL.append(instrLoadESL);
        spliceESL.append(spliceOfLUB);

        splice.insertBefore(instr, spliceESL);
    }

    // store the ESL to ESLs
    Instruction instrPushIndex = new ConstantInst(layerNo);
    Instruction instrStoreESL = new Store(Opcode.BASTORE, code);

    Splice spliceESLs = new Splice();

    spliceESLs.append(loadESLsRef);
    spliceESLs.append(instrPushIndex);
    spliceESLs.append(loadTempSL);
    spliceESLs.append(instrStoreESL);

    splice.insertBefore(instr, spliceESLs);

    // set ESL to curESL
    Instruction instrStoreCurESL =
        new Store(Opcode.ISTORE, indexOfCESL, code);
    splice.insertBefore(instr, loadTempSL);
    splice.insertBefore(instr, instrStoreCurESL);
}
```

Figure 7-15. Part of the program dealing with environment security-level.

7.3 Examples of Applying BMOS

To prove the validity of our new security model and dynamic approach, we give examples of applying our verification system BMOS to an agent system AgentSpace to verify the security of agents.

7.3.1 Applying MOBS in AgentSpace

- AgentSpace system

We use an agent system called AgentSpace (http://research.nii.ac.jp/~ichiro /agent/agentspace.html) to demonstrate how to apply our verification method to one agent system.

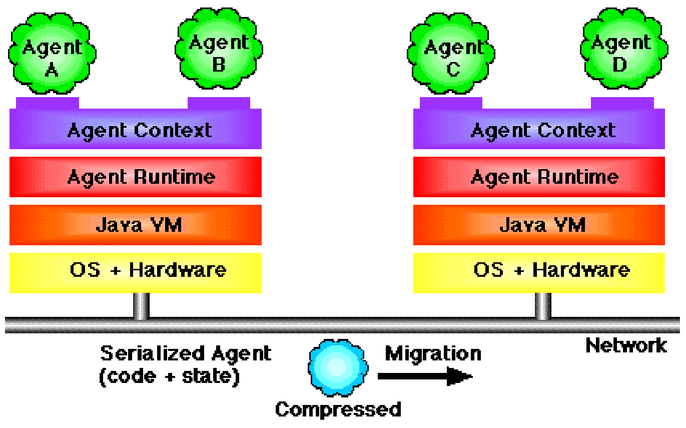


Figure 7-16. Architecture of AgentSpace system.

The runtime system of AgentSpace is shown in Figure 7-16. In AgentSpace, a mobile agent is a Java object containing code and state, and it can be transmitted to a remote host and then be executed.

In AgentSpace system, the AgentServer works as the platform of the mobile agent system. And AgentReceiver, AgentMonitor and AgentSender, are used to perform the agent operations. The AgentReceiver listens to the socket in order to receive any agent from other hosts. If the AgentReceiver gets any agent, the AgentLoader will transform the serialized data to one object representing the agent and register the agent to the AgentManager. The AgentManager manages all the agents running on the host, and will initialize a thread for each agent and invoke the method `arrive()` to make the agent begin its work. AgentMonitor watches over and executes the operation done on the agents by the host user. And before the AgentSender sends the agent to its next destination, the method *dispatch*() of the agent is invoked to do the wrapping operation. The receiving process and the sending process of one agent on a host in the AgentSpcae system are shown in Figure 7-17 and Figure 7-18.

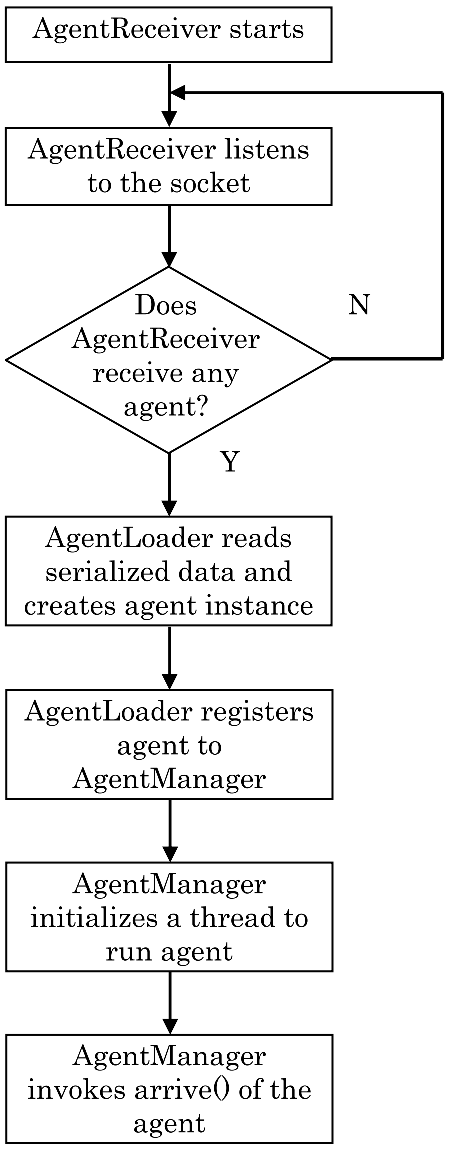

AgentReceiver starts
AgentReceiver listens to the socket
Does AgentReceiver receive any agent?
N
Y
AgentLoader reads serialized data and creates agent instance
AgentLoader registers agent to AgentManager
AgentManager initializes a thread to run agent
AgentManager invokes arrive() of the agent

Figure 7-17. Agent receiving process in AgentSpace.

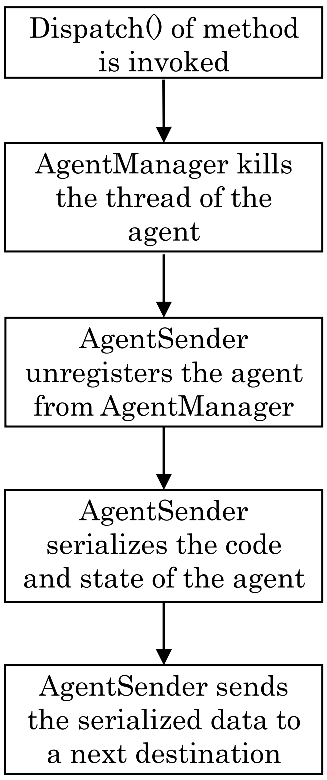

Dispatch() of method is invoked
AgentManager kills the thread of the agent
AgentSender unregisters the agent from AgentManager
AgentSender serializes the code and state of the agent
AgentSender sends the serialized data to a next destination

Figure 7-18. Agent sending process in AgentSpace.

- AgentSpace system with embedded modifier

In the AgentSpace system, the security issues are not considered. Therefore, we embed our verification system into AgentSpace in order to protect the host security from agents. The modified architecture of AgentSpace is shown in Figure 7-19.

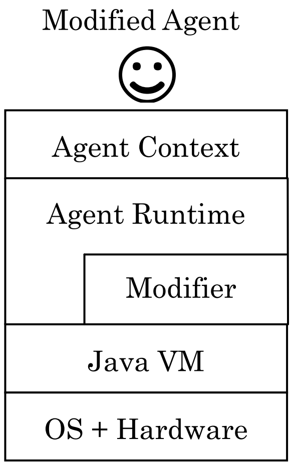


Figure 7-19. Architecture of AgentSpace system with modifier.

As discussed in Chapter 5, the task of the modifier is to insert verification code into classes of mobile programs, which are agents in the AgentSpace system. The inserted code includes the code calculating security-levels and the code detecting data-leaking. And since the data-leaking channels have two types (data-leaking through network connection built by the mobile code itself and data-leaking through the movement of the mobile code), the code detecting data-leaking can be divided into two parts detecting the two types of data-leaking further.

The code calculating security-levels and the code detecting data-leaking through network connection built by the mobile code should be inserted into all the classes of the mobile program and such insertion is independent of the mobile code system. Thus such codes should be inserted by the modifier to the agent's classes in AgentSpace system.

The insertion of the code detecting the data-leaking through the movement of the mobile code is dependent on the architecture of the mobile code system. In the AgentSpace system, the state of one agent can also be transferred with the agent between hosts. When the agent moves to the next destination, the information held in its fields will be taken out of the local host. Thus, according to our security model, the information in the fields should be checked before the agent moves to the next destination. Since the method `dispatch()` will be invoked by the AgnetManager before the AgentSpace system sends out the agent, the modifier should insert such checking codes into the method `dispatch()`.

The timing of invoking the modifier should be late enough that the code of the agent is read from the serialized data, and also early enough that the object of the agent has not been created. In AgentSpace, the method `agentClassDataLoad` of the class `AgentClassLoader` is used to unzip and read the code of the agent from serialized data. Thus we add the code of invoking our modifier to the method `agentClassDataLoad` as shown in the box of Figure 7-20. In this way, every agent loaded into the local host will be modified and inserted with verification codes; and the agent will be verified during its execution.

```
public Hashtable agentClassDataLoad(byte[] data) {
    try {
        ByteArrayInputStream byteInStream =
            new ByteArrayInputStream(data);
        ZipInputStream zipInStream =
            new ZipInputStream(byteInStream);
        ZipEntry zipEnt;
        while ((zipEnt = zipInStream.getNextEntry()) != null) {
            String entryName = zipEnt.getName();
            if (entryName.startsWith("META-INF")) {
                continue;
            }
            int count;
            ByteArrayOutputStream byteOutStream =
                new ByteArrayOutputStream();
            byte[] classBytes = new byte[1024];
            while ((count = zipInStream.read(classBytes)) != -1) {
                byteOutStream.write(classBytes, 0, count);
            }
            byte[] bytes = byteOutStream.toByteArray();

            //Code added to embed BMOS to AgentSpace
            ClassInfo ci = new ClassInfo(bytes, true);
            ClassModifier cm = new ClassModifier(ci);
            cm.modifyClass();
            bytes = ci.writeToBytes();

            cache.put(entryName, bytes);
        }
        zipInStream.close();
        byteInStream.close();
    } catch (IOException e) {
        System.err.println(e);
    }
    return cache;
}
```

Figure 7-20. Code added into AgentLoader to invoke the modifier.

The execution and verification process of the agent in AgentSpace system with embedded modifier is shown in Figure 7-21.

When one agent arrives at a host, the AgentLoader reads the serialized data and then invokes the modifier to modify the classes of the agent. Verification code is inserted into the agent, especially the code checking the information in all the fields into the method `dispatch()`. And then the AgentLoader creates one instance of the modified agent. After the AgentManager initializes a thread and makes the instance of the agent run, the method `arrive()`is invoked by the AgentManager and the agent begins its work. During the execution of the modified agent, the security-levels are calculated. And if the agent tries to send out any data through the network connection, the data to be sent will be checked to detect data-leaking. If any data-leaking is detected, an exception will be thrown out and the execution is interrupted to prevent the data-leaking from happening.

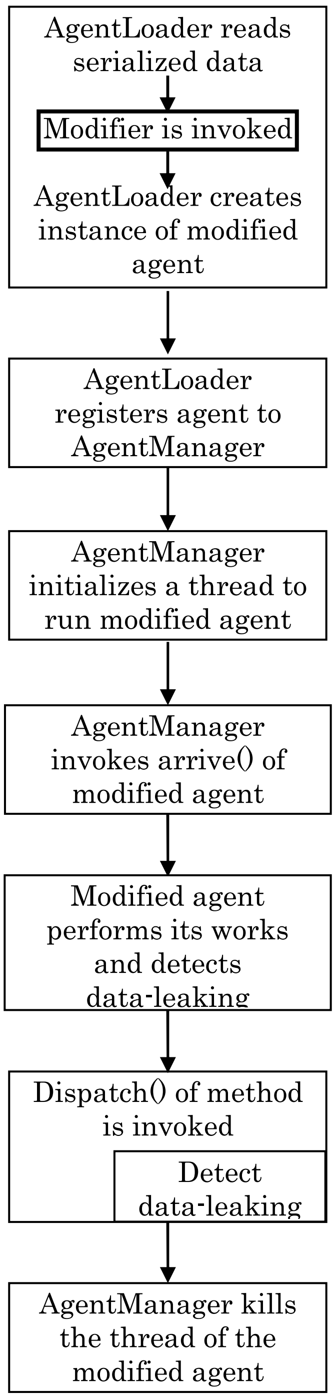

AgentLoader reads serialized data
Modifier is invoked
AgentLoader creates instance of modified agent
AgentLoader registers agent to AgentManager
AgentManager initializes a thread to run modified agent
AgentManager invokes arrive() of modified agent
Modified agent performs its works and detects data-leaking
Dispatch() of method is invoked
Detect data-leaking
AgentManager kills the thread of the modified agent

Figure 7-21. Agent receiving process in AgentSpace with modifier.

When the work of the agent finished, the agent will move to its next destination. Before the agent is sent out, the method `dispatch()` will be invoked by the AgnetManager. The code inserted into the method `dispatch()`will check the information stored in all the fields of the agent. If the information is sent to one unauthorized host, an exception will be thrown out and the execution is interrupted to prevent the data-leaking from happening. If no data-leaking is detected, the agent will be sent to its next destination.

7.3.2 Example Agent

Here we give an example agent of AgentSpace system to show how the modifier works. The agent `Evaluation` is a quite simple tool to calculate the estimated salary based on the education, working years and the age of the user. And if the user agrees, the agent will take or send the result back to its original host, or it will discard the result.

We implemented two versions of the agent `Evaluation`. The difference between the two versions is the way of sending the estimated salary. One version uses the field to take the estimated salary back to the original host; while the other version will send the estimated salary back to the original host directly by socket built by the agent itself. We discuss the verification process of the two versions in the AgentSpace system with embedded modifier respectively.

- Agent `Evaluation` of the version I

In this version, the agent `Evaluation` will use the field `salary` to take the estimated salary back to the original host. The Java source code is shown in Figure 7-22.

```
public class Evaluation extends Agent{
   private int salary = 0;

   public void arrive(){
       evaluation(28, 2, 3, 1);
   }

   public void evaluation(int age, int workingYears,
      int education, int sendFlag){
          int result = 0;
          int baseSal = 0;
          int ageToWork = 0;
          switch (education) {
              case 0:// High schoole
                     baseSal = 200;
                     ageToWork = 18;
                     break;
              case 1:// Bachelor
                     baseSal = 300;
                     ageToWork = 22;
                     break;
              case 2:// Master
                     baseSal = 400;
                     ageToWork = 24;
                     break;
              case 3:// Doctor
                     baseSal = 500;
                     ageToWork = 27;
                     break;
              case 4:// post doctorate
                     baseSal = 550;
                     ageToWork = 30;
          }
         result = baseSal+(workingYears*2-(age-ageToWork))*20;

          if (sendFlag == 1) {// the user agrees to send
              salary = result;
          }
   }

   public void dispatch(){}
}
```

Figure 7-22. Java source ode of the sample agent `Evaluation` of the version I.

When the agent `Evaluation` of the version shown in Figure 7-22 arrives at a host, the AgentListener passes the serialized data to the AgentLoader to create the instance of the agent. After the AgentManager initializes a thread and make the instance of the agent `Evaluation` run, the method `arrive()`is invoked by the AgentManager and the agent begins its works. In method `evaluation()`, the salary is calculated based on the age, working years and education; and the result is stored in the temporary variable `result`. Thus variable `result` will get the information of the variables `age`, `workingYears` and `education`. And then the value of `result` is transferred to the field `salary` if the variable `sendFlag` is “1”; in this case the field `salary` gets the information of the variables `age`, `workingYears` and `education` indirectly. After the execution of the method `evaluation()`, the AgentManager will invoke the method `dispatch()`and then the AgentSender will send the agent `Evaluation` to its next destination with the field `salary`. Obviously, the information held in the field `salary` will be taken out of the local host and may be leaked to some unauthorized hosts.

Our approach can be used to trace the information flow and detect possible data-leaking in the agent `Evaluation` of version I. When the agent `Evaluation` shown in Figure 7-22 arrives at a host where the modifier is embedded into the AgentSpace system as shown in Figure 7-20, the AgentReceiver of the local AgentSpace system detects it and passes the agent to the AgentLoader. After the AgentLoader reads bytecode of the agent, the modifier is invoked to modify the agent and insert the verification codes into the agent. Then the AgentLoader will generate the agent instance based on the modified bytecode instead of the original bytecode.

In Figure 7-23, we give the original bytecode of the method `evaluation()` in the agent `Evaluation` of version I and the modified bytecode generated by our modifier described above. In the modified bytecode shown in Figure 7-23(b), the bold codes are the original codes of the method `evaluation()` (the indices of local registers have been changed) and others are the codes inserted to calculate the security-levels. Especially, the codes in shadow are the codes used to calculate the LUB of two security-levels. And in Figure 7-24, we give the code inserted into the method `dispatch()`to detect data-leaking through the movement of the agent.

```
0 iconst_0
1 istore 5
3 iconst_0
4 istore 6
6 iconst_0
7 istore 7
9 iload_3
10 tableswitch
  0: 44 (34)
  1: 56 (46)
  2: 68 (58)
  3: 80 (70)
  4: 92 (82)
  default: 101(91)
44 sipush 200
47 istore 6
49 bipush 18
51 istore 7
53 goto 48
56 sipush 300
59 istore 6
61 bipush 22
63 istore 7
65 goto 36
68 sipush 400
71 istore 6
73 bipush 24
75 istore 7
77 goto 24
80 sipush 500
83 istore 6
85 bipush 27
87 istore 7
89 goto 12
92 sipush 550
95 istore 6
97 bipush 30
99 istore 7
101 iload 6
103 iload_2
104 iconst_2
105 imul
106 iload_1
107 iload 7
109 isub
110 isub
111 bipush 20
113 imul
114 iadd
115 istore 5
117 iload 4
119 iconst_1
120 if_icmpne 9
123 aload_0
124 iload 5
126 putfield 12
129 aload_0
130 invokevirtual 32
133 return
```

a. original bytecode.

```
0 sipush 255
3 newarray 8
5 astore 16
7 iconst_0
8 iconst_0
9 istore 11
11 istore 10
13 iconst_0
14 iconst_0
15 istore 13
17 istore 12
19 iconst_0
20 iconst_0
21 istore 15
23 istore 14
25 iload 6
27 iload 7
29 aload 16
31 iconst_1
32 bastore
33 tableswitch
  0: 67(34)
  1: 127(94)
  2: 187(154)
  3: 247(214)
  4: 307(274)
  default: 364(331)
67 sipush 200
70 iconst_0
71 aload 16
73 iconst_1
74 baload
75 istore 18
77 istore 19
79 iload 19
81 iload 18
83 if_icmplt 7
86 iload 19
88 istore 18
90 iload 18
92 istore 13
94 istore 12
96 bipush 18
98 iconst_0
99 aload 16
101 iconst_1
102 baload
103 istore 18
105 istore 19
107 iload 19
109 iload 18
111 if_icmplt 7
114 iload 19
116 istore 18
118 iload 18
120 istore 15
122 istore 14
124 goto 240
127 sipush 300
130 iconst_0
131 aload 16
133 iconst_1
134 baload
135 istore 18
137 istore 19
139 iload 19
141 iload 18
143 if_icmplt 7
146 iload 19
148 istore 18
150 iload 18
152 istore 13
154 istore 12
156 bipush 22
158 iconst_0
159 aload 16
161 iconst_1
162 baload
163 istore 18
165 istore 19
167 iload 19
169 iload 18
171 if_icmplt 7
174 iload 19
176 istore 18
178 iload 18
180 istore 15
182 istore 14
184 goto 180
187 sipush 400
190 iconst_0
191 aload 16
193 iconst_1
194 baload
195 istore 18
197 istore 19
199 iload 19
201 iload 18
203 if_icmplt 7
206 iload 19
208 istore 18
210 iload 18
212 istore 13
214 istore 12
216 bipush 24
218 iconst_0
219 aload 16
221 iconst_1
222 baload
223 istore 18
225 istore 19
227 iload 19
229 iload 18
231 if_icmplt 7
234 iload 19
236 istore 18
238 iload 18
240 istore 15
242 istore 14
244 goto 120
247 sipush 500
250 iconst_0
251 aload 16
253 iconst_1
254 baload
255 istore 18
257 istore 19
259 iload 19
261 iload 18
263 if_icmplt 7
266 iload 19
268 istore 18
270 iload 18
272 istore 13
274 istore 12
276 bipush 27
278 iconst_0
279 aload 16
281 iconst_1
282 baload
283 istore 18
285 istore 19
287 iload 19
289 iload 18
291 if_icmplt 7
294 iload 19
296 istore 18
298 iload 18
300 istore 15
302 istore 14
304 goto 60
307 sipush 550
310 iconst_0
311 aload 16
313 iconst_1
314 baload
315 istore 18
317 istore 19
319 iload 19
321 iload 18
323 if_icmplt 7
326 iload 19
328 istore 18
330 iload 18
332 istore 13
334 istore 12
336 bipush 30
338 iconst_0
```

```
339 aload 16
341 iconst_1
342 baload
343 istore 18
345 istore 19
347 iload 19
349 iload 18
351 if_icmplt 7
354 iload 19
356 istore 18
358 iload 18
360 istore 15
362 istore 14
364 iload 12
366 iload 13
368 iload 4
370 iload 5
372 iconst_2
373 iconst_0
374 istore 18
376 istore 20
378 iload 18
380 istore 18
382 istore 19
384 iload 19
386 iload 18
388 if_icmplt 7
391 iload 19
393 istore 18
395 iload 20
397 imul
398 iload 18
400 iload_2
401 iload_3
403 iload 14
405 iload 15
407 istore 18
409 istore 20
411 iload 18
413 istore 18
415 istore 19
```

```
417 iload 19
419 iload 18
421 if_icmplt 7
424 iload 19
426 istore 18
428 iload 20
430 isub
431 iload 18
433 istore 18
435 istore 20
437 iload 18
439 istore 18
441 istore 19
443 iload 19
445 iload 18
447 if_icmplt 7
450 iload 19
452 istore 18
454 iload 20
456 isub
457 iload 18
458 bipush 20
461 iconst_0
462 istore 18
464 istore 20
466 iload 18
468 istore 18
470 istore 19
472 iload 19
474 iload 18
476 if_icmplt 7
479 iload 19
481 istore 18
483 iload 20
485 imul
486 iload 18
488 istore 18
490 istore 20
492 iload 18
494 istore 18
496 istore 19
```

```
498 iload 19
500 iload 18
502 if_icmplt 7
505 iload 19
507 istore 18
509 iload 20
511 iadd
512 iload 18
514 istore 11
516 istore 10
518 iload 8
520 iload 9
522 iconst_1
523 iconst_0
524 istore 18
526 istore 20
528 iload 18
530 istore 18
532 istore 19
534 iload 19
536 iload 18
538 if_icmplt 7
541 iload 19
543 istore 18
545 aload 16
547 iconst_1
548 iload 18
550 bastore
551 iload 20
553 if_icmpne 20
556 aload_0
557 aload_1
558 iload 10
560 iload 11
562 istore 18
564 swap
565 iload 18
567 putfield 22
570 putfield 12
573 return
```

b. modified bytecode.

Figure 7-23. Java bytecode of the method `evaluation()` of version I.

In the execution of the modified method `evaluation()` shown in Figure 7-23(b), the codes at address from 33 to 362 will assign the initial values to the variable `baseSal` and `ageToWork` based on the value of the variable `education`, and calculate the security-levels of them. The codes at address from 364 to 514 will calculate the estimated salary and store it in the variable `result`;

the security-level of the estimated salary is calculated as well by these codes. If the `sendFlag` is “1”, the codes at address from 556 to 570 will set the estimated salary to the field `salary` and update the security-level of the field. And in the modified code, the codes at address from 29 to 32 and from 524 to 550 are used to calculate the environment security-level of the implicit blocks caused by the instructions *tableswitch* and *if_icmpne*.

```
0 aload_0
1 getfield 22
4 getstatic 29
7 if_icmple 11
10 new 50
13 dup
14 invokespecial 52
17 athrow
18 return
```

Figure 7-24. The bytecode of the modified method `dispatch()`.

In such a way, the security-levels of all the variables can be calculated. Furthermore, different from static approaches, our approach will update the security-level of the field `salary` only when the filed does get the information in the variables `age`, `workingYears` and `education`. Thus the information flow in the method `evaluation()` can be traced precisely and the preparation of detecting data-leaking can be done well.

As mentioned above, the information held in the fields of one agent will be taken out of the local host when the agent moves to next destination. In the case of the agent `Evaluation`, the information held in the field `salary` will be taken to the next destination host. The codes inserted into the method `dispatch()`as shown in Figure 7-24 compare the security-level of the information in the field `salary` (stored in the added field indicated by the index 22 in the constant pool) with the clearance-level of the next destination host (stored in static field indicated by the index 29 in the constant pool). And the codes will throw an exception if the security-level is greater, which means that a data-leaking happens.

- Agent `Evaluation` of the version II

In this version, the agent `Evaluation` will send the estimated salary back to the original host through a socket directly. The Java source code is shown in Figure 7-25.

```
public class Evaluation extends Agent{
   private final String hostName = "192.168.21.3";
   private final int hostPort = 8080;

   public void arrive(){
       evaluation(28, 2, 3, 1);
   }

   public void evaluation(int age, int workingYears,
      int education, int sendFlag){
          int result = 0;
          int baseSal = 0;
          int ageToWork = 0;
          switch (education) {
              case 0:// High schoole
                     baseSal = 200;
                     ageToWork = 18;
                     break;
              case 1:// Bachelor
                     baseSal = 300;
                     ageToWork = 22;
                     break;
              case 2:// Master
                     baseSal = 400;
                     ageToWork = 24;
                     break;
              case 3:// Doctor
                     baseSal = 500;
                     ageToWork = 27;
                     break;
              case 4:// post doctorate
                     baseSal = 550;
                     ageToWork = 30;
          }
          result = baseSal+(workingYears*2-(age-ageToWork))*20;

          if (sendFlag == 1) {// the user agrees to send
              Socket socket = new Socket(hostName, hostPort);
              OutputStream out = socket.getOutputStream();
              out.write(salary);
          }
   }

   public void dispatch(){}
}
```

Figure 7-25. Java source ode of the sample agent `Evaluation` of the version II.

When the agent `Evaluation` of the version shown in Figure 7-25 arrives at a host, the process of the agent is the same as the one shown in Figure 7-22 until the value of the variable `sendFlag` is checked. Different from version I, the agent of version II will send the variable `result` out through a socket if the variable `sendFlag` is “1”. Obviously, the information held in the variable `result` will be taken out of the local host and may be leaked to some unauthorized hosts. And in version II, there are no fields used to take data out of the local host.

Our approach can be used to trace the information flow and detect possible data-leaking in the agent `Evaluation` of version II, too. When the agent `Evaluation` shown in Figure 7-25 arrives at a host where the modifier is embedded into the AgentSpace system as shown in Figure 7-20, the AgentReceiver of the local AgentSpace system detects it and passes the agent to the AgentLoader. After the AgentLoader reads bytecode of the agent, the modifier is invoked to modify the agent and verification code is inserted into the agent. Then the AgentLoader will generate the agent instance based on the modified bytecode instead of the original bytecode.

In Figure 7-26, we give the original bytecode of the method `evaluation()` in the agent `Evaluation` of version II and the modified bytecode generated by our modifier described above. In the modified bytecode shown in Figure 7-26(b), the bold codes are the original codes of the method `evaluation()` (the indices of local registers have been changed) and others are the codes inserted to calculate the security-levels. Especially, the codes in shadow are the codes used to calculate the LUB of two security-levels; and the codes in the black box are used to detect data-leaking. Since no field is used to take data out of the local host, no verification code is inserted into the method `dispatch()`.

```
0 iconst_0
1 istore 5
3 iconst_0
4 istore 6
6 iconst_0
7 istore 7
9 iload_3
10 tableswitch
  0: 44 (34)
  1: 56 (46)
  2: 68 (58)
  3: 80 (70)
  4: 92 (82)
  default: 101(91)
44 sipush 200
47 istore 6
49 bipush 18
51 istore 7
53 goto 48
56 sipush 300
59 istore 6
61 bipush 22
63 istore 7
65 goto 36
68 sipush 400
71 istore 6
73 bipush 24
75 istore 7
77 goto 24
80 sipush 500
83 istore 6
85 bipush 27
87 istore 7
89 goto 12
92 sipush 550
95 istore 6
97 bipush 30
99 istore 7
101 iload 6
103 iload_2
104 iconst_2
105 imul
106 iload_1
107 iload 7
109 isub
110 isub
111 bipush 20
113 imul
114 iadd
115 istore 5
117 iload 4
119 iconst_1
120 if_icmpne 31
123 new 44
126 dup
127 ldc 8
129 sipush 8080
132 invokespecial 46
135 astore 8
137 aload 8
139 invokevirtual 49
142 astore 9
144 aload 9
146 iload 5
148 invokevirtual 53
151 return
```

a. original bytecode.

```
0 sipush 255
3 newarray 8
5 astore 16
7 iconst_0
8 iconst_0
9 istore 11
11 istore 10
13 iconst_0
14 iconst_0
15 istore 13
17 istore 12
19 iconst_0
20 iconst_0
21 istore 15
23 istore 14
25 iload 6
27 iload 7
29 aload 16
31 iconst_1
32 bastore
33 tableswitch
  0: 67(34)
  1: 127(94)
  2: 187(154)
  3: 247(214)
  4: 307(274)
  default: 364(331)
67 sipush 200
70 iconst_0
71 aload 16
73 iconst_1
74 baload
75 istore 18
77 istore 19
79 iload 19
81 iload 18
83 if_icmplt 7
86 iload 19
88 istore 18
90 iload 18
92 istore 13
94 istore 12
96 bipush 18
98 iconst_0
99 aload 16
101 iconst_1
102 baload
103 istore 18
105 istore 19
107 iload 19
109 iload 18
111 if_icmplt 7
114 iload 19
116 istore 18
118 iload 18
120 istore 15
122 istore 14
124 goto 240
127 sipush 300
130 iconst_0
131 aload 16
133 iconst_1
134 baload
135 istore 18
137 istore 19
139 iload 19
141 iload 18
143 if_icmplt 7
146 iload 19
148 istore 18
150 iload 18
152 istore 13
154 istore 12
156 bipush 22
158 iconst_0
159 aload 16
161 iconst_1
162 baload
163 istore 18
165 istore 19
167 iload 19
169 iload 18
171 if_icmplt 7
174 iload 19
176 istore 18
178 iload 18
180 istore 15
182 istore 14
184 goto 180
187 sipush 400
190 iconst_0
191 aload 16
193 iconst_1
194 baload
195 istore 18
197 istore 19
199 iload 19
201 iload 18
203 if_icmplt 7
206 iload 19
208 istore 18
210 iload 18
212 istore 13
214 istore 12
216 bipush 24
218 iconst_0
219 aload 16
221 iconst_1
222 baload
223 istore 18
225 istore 19
227 iload 19
229 iload 18
231 if_icmplt 7
234 iload 19
236 istore 18
238 iload 18
240 istore 15
242 istore 14
244 goto 120
247 sipush 500
250 iconst_0
251 aload 16
253 iconst_1
254 baload
255 istore 18
257 istore 19
259 iload 19
261 iload 18
263 if_icmplt 7
266 iload 19
268 istore 18
270 iload 18
272 istore 13
274 istore 12
276 bipush 27
278 iconst_0
279 aload 16
281 iconst_1
282 baload
283 istore 18
285 istore 19
287 iload 19
289 iload 18
291 if_icmplt 7
294 iload 19
296 istore 18
298 iload 18
300 istore 15
302 istore 14
304 goto 60
307 sipush 550
310 iconst_0
311 aload 16
313 iconst_1
314 baload
315 istore 18
317 istore 19
319 iload 19
321 iload 18
323 if_icmplt 7
326 iload 19
328 istore 18
330 iload 18
332 istore 13
334 istore 12
336 bipush 30
338 iconst_0
```

```
339 aload 16
341 iconst_1
342 baload
343 istore 18
345 istore 19
347 iload 19
349 iload 18
351 if_icmplt 7
354 iload 19
356 istore 18
358 iload 18
360 istore 15
362 istore 14
364 iload 12
366 iload 13
368 iload 4
370 iload 5
372 iconst_2
373 iconst_0
374 istore 18
376 istore 20
378 iload 18
380 istore 18
382 istore 19
384 iload 19
386 iload 18
388 if_icmplt 7
391 iload 19
393 istore 18
395 iload 20
397 imul
398 iload 18
400 iload_2
401 iload_3
403 iload 14
405 iload 15
407 istore 18
409 istore 20
411 iload 18
413 istore 18
415 istore 19
417 iload 19
419 iload 18
421 if_icmplt 7
424 iload 19
426 istore 18
428 iload 20
430 isub
431 iload 18
433 istore 18
435 istore 20
437 iload 18
439 istore 18
441 istore 19
443 iload 19
445 iload 18
447 if_icmplt 7
450 iload 19
452 istore 18
454 iload 20
456 isub
457 iload 18
458 bipush 20
461 iconst_0
462 istore 18
464 istore 20
466 iload 18
468 istore 18
470 istore 19
472 iload 19
474 iload 18
476 if_icmplt 7
479 iload 19
481 istore 18
483 iload 20
485 imul
486 iload 18
488 istore 18
490 istore 20
492 iload 18
494 istore 18
496 istore 19
498 iload 19
500 iload 18
502 if_icmplt 7
505 iload 19
507 istore 18
509 iload 20
511 iadd
512 iload 18
514 istore 11
516 istore 10
518 iload 8
520 iload 9
522 iconst_1
523 iconst_0
524 istore 18
526 istore 20
528 iload 18
530 istore 18
532 istore 19
534 iload 19
536 iload 18
538 if_icmplt 7
541 iload 19
543 istore 18
545 aload 16
547 iconst_1
548 iload 18
550 bastore
551 iload 20
553 if_icmpne 47
556 new 44
559 dup
560 ldc 8
562 sipush 8080
565 invokespecial 46
568 astore 21
570 aload 21
572 invokevirtual 49
575 astore 23
577 aload 23
579 iload 10
581 ilaod 11
583 getstatic 33
586 if_icmple 11
589 new 50
592 dup
593 invokespecial 52
596 athrow
597 invokevirtual 53
600 return
```

b. modified bytecode.

Figure 7-26. Java bytecode of the method `evaluation()` of version II.

In the execution of the modified method `evaluation()` shown in Figure 7-26(b), the information flow is the same as the one in the version I until the instruction at address 553, which checks the value of the variable `sendFlag`. If the variable `sendFlag` is “1”, the agent will send the value of the variable `result` out of the local host through the socket built by the instruction at address 565. Thus the inserted codes at address from 581 to 596 compare the security-level of the information in the variable `result` (stored in the local register 11) with the clearance-level of the destination host (store in stored in static field indicated by the index 33 in the constant pool). And the codes will throw an exception if the security-level is greater, which means that a data-leaking happens.

7.3.3 Evaluation

- Efficiency evaluation

For dynamic verification approach, the preparation time (the modification time in our method) and the execution time increment are important evaluation factors. We take the agent `Evaluation` shown above as the evaluation example and give the result in Figure 7-27.

Test Environment:
OS: Windows Vista
JVM: JDK 1.4.2.16
CPU: Intel Core2 6320(1.86GHz)
Memory: 2.5G

	Original	Modified	Increase to
Number of Instruction	53	293	553%
Execution Time(ms)	92.2×10^{-7}	441.5×10^{-7}	478%
Modification Time (ms)	12		

a. Evaluation result of version I

	Original	Modified	Increase to
Number of Instruction	60	294	490%
Execution Time(ms)	25.9	26.0	100.4%
Modification Time (ms)	13		

b. Evaluation result of version II

Figure 7-27. Performance evaluation result of the agent `Evaluation`.

```
public class AgentRuntime extends Thread {
      ...
      ...
   public void run() {
      theAgent = theInfo.getAgent();
      if (theAgent == null) {
         return;
      }
      if(theInfo.getAgentStatus().equals(AgentStatus.PREINIT)) {
         theInfo.setAgentStatus(AgentStatus.ONCREATE);
         theAgent.create();
         theInfo.setAgentStatus(AgentStatus.NORMAL);
      } else if (theInfo.getAgentStatus().
        equals(AgentStatus.ONTRANSMIT)) {
         theInfo.setAgentStatus(AgentStatus.ONARRIVE);
         for (int i = 0; i < 10000000; i ++){
            theAgent.arrive();
            theAgent.dispatch();
         }
         theInfo.setAgentStatus(AgentStatus.NORMAL);
      } else if (theInfo.getAgentStatus().
        equals(AgentStatus.PERSISTENT)) {
         theInfo.setAgentStatus(AgentStatus.ONRESUME);
         theAgent.resume();
         theInfo.setAgentStatus(AgentStatus.NORMAL);
      } else if (theInfo.getAgentStatus().
        equals(AgentStatus.CHILD)) {
         theInfo.setAgentStatus(AgentStatus.ONCLONE);
         theAgent.child(theInfo.getParentIdentifier());
         theInfo.setAgentStatus(AgentStatus.NORMAL);
      } else if (theInfo.getAgentStatus().
        equals(AgentStatus.ERROR)) {
         theInfo.setAgentStatus(AgentStatus.ONEXCEPT);
         if (!(theAgent.except(theInfo.getAgentError()))) {
            theInfo.setAgentStatus(AgentStatus.DEATH);
            theInfo.setAgentError(null);
            return;
         }
         theInfo.setAgentError(null);
         theInfo.setAgentStatus(AgentStatus.NORMAL);
      } else {
         System.out.println("unknown status: " +
            theInfo.getAgentStatus());
      }
      ...
      ...
```

Figure 7-28. The code used to execute the agent of version I 10 million times.

Since the execution time of the agent (no matter the original one or the modified one) of the version I is too short to be measured, we add a loop in the AgnetManager to execute the original agent and modified agent 10 million times;

and then calculate the average time as the execution time of the original one and modified one. And we also add code to invoke the method `dispatch()` in order to make the result time involve the execution time of the method `dispatch()`. The codes added into the AgentManager are the ones in the black box in Figure 7-28.

As for the execution time of the agent of version II shown in Figure 7-27(b), we execute the original agent and modified agent 10 times respectively, and use the average value as the result.

From the result shown in Figure 7-27, we can find that the execution times of version I and version II are quite different though the numbers of the instructions of them are almost same. The reason is that there is one instruction to build a socket connection in version II. Such an operation will not finish until the other party responds it through the next work; and costs quite more time than common instructions that can be finished by the local JVM.

From the result of version I, we can get to know that the modification time is short. For a class composed of 4000 instructions (the average number of instructions in one Java core class is about 150), the modification will be finished less than one second. While the number of instructions and the execution time increase to about 5 times. This will slow down the execution speed. But considering the time itself, the execution time of the modified agent is still too short to be felt though it has increased to 5 times of original one. From the result, we can estimate that: even for one agent composed of 600,000 instructions, the execution of the modified agent will finish in 0.5 second. Therefore in some degree the additional overhead caused by the code inserted is acceptable.

The result of version II also proved the execution time increment caused by verification code is so small that it is can even be omitted. The number of modified instructions increased to about 5 times of the original one. But the execution time of original code and modified code is almost the same. The reason is that connecting the socket (the instruction at address 132 in Figure 7-26(a)) occupied most (almost 100%) of the execution time; even the execution time of other codes is increased to about 5 times by the verification codes, this part of execution time is still so small and can be omitted compared with the time of connecting socket. Thus the execution times of original code and modified code are almost the same.

As for the codes detecting data-leaking in version I and II, we can estimate that the execution time of these codes only takes up a small part of the total execution time because 1. the codes do not include any instructions costing much time; 2. the number of these codes only take a small part of total instructions (3% in version I and 2.3% in version II).

- Security verification evaluation

In Figure 7-29, we compare our approach with type-system approaches and static approaches of non-type-system on the aspect that whether the agent can be verified correctly, that is, verified as malicious when data-leaking happens and as secure when data-leaking does not happen.

#		1	2	3	4
Value of `sendFlag` is “1”		○	○	×	×
$CL \geq SL$		×	○	×	○
Data-leaking happens		○	×	×	×
Agent is Malicious or Secure		M	S	S	S
Agent is verified as	by Type-system approaches	M	M	M	M
	by Static approach of non-type-system	M	S	M	S
	by Our approach	M	S	S	S

Figure 7-29. The comparison of our approach with type-system and static approaches. (*CL* denotes the destination host’s clearance-level; *SL* denotes the LUB of the security-levels of the age, workingYears and education; M denotes the agent is verified as malicious; S denotes the agent is verified as secure.)

Compared with the type-system verification approaches, our approach is more precise. For example, the variable `baseSal` in the method `evaluation()`will get the information of the variables `education`. If the security-level of the variable `baseSal` is set lower (in fact the security-level of such temporary variable is usually set to the lowest one in type-system) than the variables `education`, the type-system approaches will reject the agent since they consider that the instruction at address 44 in Figure 7-23(a) and Figure 7-26(a) causes one data-leaking. Obviously such a judgment is wrong in the case 2, 3 and 4 in Figure 7-29. (In the case 1, though the type-system approaches can get a correct judgment, they do not find the instruction that really caused data-leaking.) In our approach, only the variable to be sent out of the local host will be checked whether a data-leaking is caused. Thus the variable `baseSal` is not the checking target in our approach in the all cases. In this way, the problem of imprecise verification is overcome in our approach.

And as analyzed above, in the `evaluation()` shown in Figure 7-22, the field `salary` will get information of variables `age`, `workingYears` and `education` only when the `sendFlag` is “1”. But in static approaches, the field `salary` is considered to get the information no matter the `sendFlag` is “1” or not. If the clearance-level of the destination host is set lower than any one of the variables `age`, `workingYears` and `education`, the static approaches will consider that the agent causes one data-leaking since it will leak data to unauthorized host, no matter the `sendFlag` is “1” or not. Obviously such judgment is wrong when the `sendFlag` is not “1” (the case 3 in Figure 7-29). While in our approach such misjudgment is avoided since our approach is dynamic and updates the security-level of the information in the field `salary` only when the `sendFlag` is “1”. Thus our approach will not make the misjudgment of considering one agent as malicious for the unexecuted code; while such misjudgment is not avoidable in static approaches. In this way, the verification precision is improved further. As for the version II shown in Figure 7-25, the static approaches will also make similar misjudgment while our approach can avoid such mistake.

Based the analysis above, we can find that our verification approach resolved the inherent problems of type-system verification approaches and the static verification approaches and improved the verification precision. While the cost of the improvement is the additional execution overhead caused by the code inserted to calculate security-levels and detect data-leaking, which can usually be omitted.

7.4 Discussion

7.4.1 Improvements

- Less restriction on mobile code program

In our approach, we only check and restrict the output operation of the bytecode. The mobile programs can obtain any information (input) from the host to perform their tasks. The restriction on mobile code program is less than that of traditional access control approaches. A lot of mobile code program can be implemented.

- Better verification granularity

In static approaches, all instructions of the bytecode are verified and the bytecode will be rejected if there is any malicious section of instructions. Different from static approaches, our dynamic approach achieved the runtime verification of bytecode, and only verifies the sections of instructions that are executed. In our approach, the bytecode will not be rejected by reason of its code that will not be executed in runtime. Thus the verification precision is improved.

- Recursive method invoking

For static verification approaches, the analysis will never stop if a recursive method is invoked. To solve this problem, some static approaches assume that the return variable of one recursive method depends on all of the arguments, which reduces the verification precision. In our approach, the verification of recursive methods is resolved without any additional effort. Just like other common methods, verification code is inserted into recursive methods. The verification for recursive invoking starts when the recursive method begins to invoke itself, and the verification finishes when recursive invoking stops. Thus in our approach, the verification precision will not be reduced because of the existence of recursive invoking.

- Dynamic variables determination.

For static approaches, it is impossible to disclose the information flow caused by the variables that are not determined until the program is executed. For example, the name of a file used to read information from is obtained from the console when the program is executed, so it is impossible to determine the information flow between the file and other objects.

The similar case is the elements of the array. Not until the program is executed, it is impossible to know which element of the array will be used. All the elements of the array have to share the same security class, which causes the security class of the element is too high and impairs the verification precision.

Our approach eliminates such limitation because it is dynamic verification approach. The value of all variables can be determined when the verification is performed. Thus our approach achieved better verification precision than static approaches.

7.4.2 Preconditions

In order to implement my approach, some preconditions must be satisfied as follows.

- Knowledge of API Definition

The first precondition is the knowledge of API definition used in the mobile code. The definition here means the relation between the input and output. And from it, the security-level relation between the input and output can be derived. The APIs used in Java mobile code include the common Java API and the original API.

As for the Java API, the definition has been defined clearly in the Java platform specifications. Therefore, the security-level relation between the input and the output could be derived. So in my approach, the Java API is considered as a black box. The class files of Java APIs will not be modified, and the security-level relation is used to calculate the output security-level. A library of the security-level relation should be built for the Java API.

Here is an example. For the method `min()` in the class math, the definition is "Returns the smaller of two `int` values." Therefore, I can derive the security-level relation is that the security-level of the output should be the LUB of security-levels of the two input.

As for the original API, if the Java class files are available, it will be modified as same as other class files. If definition is available, it will be considered as black box and the security-level relation library should be updated to support the API. If nothing is available, the mobile code could not be supported.

- Platform-independency of mobile code

The second precondition is the Platform-independency of mobile code. In Java program, all methods can be divided into Java methods and native methods. In the execution, the mobile code class files and necessary Java API class files are loaded by the class loader. Then the execution engine will execute these class files and invoke necessary native methods. Usually, the mobile code invokes the proper Java APIs and then Java APIs invoke the native methods. This kind of mobile code is called platform-independent code. Since my approach supports all JVM instructions and considers the Java API as a black box, the platform-independent mobile code is supported by my approach. While, some mobile code invokes the native methods directly through the Java Native Interface. And such mobile code is called platform-dependent code. Since the native methods are written in other language rather than Java, this kind of mobile code is not support by our approach.

- Knowledge of Migration Method

The third precondition is Knowledge of Migration Method. Since information could be taken out when the mobile code moves to next destination, data-leaking is checked before the mobile code migration. So where and when the mobile code migrates is necessary to my approach.

- Configuration files of security-level and clearance-level

The fourth precondition is the configuration files of security-level and clearance-level. To judge a data-transferring is data-leaking or not, the security-level of the data and the clearance-level of the destination host are necessary. The configure file of security-level could be set by the local host user. The user can assign proper security-level to his local resources to protect them. In my approach, the security-level is defined in the format as shown in Figure 7-30. And some examples defining the security-levels of the personal information are also shown in the figure. While the configure file of clearance-level could also be set by the local host user if the data is transmitted in a limited scope of hosts. Or, the clearance-level should be decided by the negotiation between hosts. The format and examples of clearance-level are shown in Figure 7-31.

```
res.level = res 1, res 2,…
e.g.   res.255 = /personal/income, /personal/address
       res.150 = /personal/birthday, /personal/telNo
       res.100 = /personal/loanStatus
       res.50 = /personal/maritalStatus
```

Figure 7-30. Security-level configuration file format.

```
type.level = res 1, res 2,…
      e.g. url.150 = www.abc.com:8080
           ip.100 = 202.118.34:15016
```

Figure 7-31. Clearance-level configuration file format.

7.4.3 Limitations

- Input/Output operation disclosing

Output operation disclosing is difficult because it contains a sequence of operation. The following is an example program to write a line to a file. This program consists of 5 method invocations, in which the information flows from the string `strInfo` to the file `strFile`. In the source code level, it is not easy to analyze the information flow. In the bytecode program, it is obviously that the information flow analysis comes to more difficult. Figure 7-32 shows the bytecode of this file writing operation. In order to get the security-level of the local file, the input operations should also be determined. Similar to output operation, the input operation disclosing is difficult too. The operation pattern and information flow behavior need to be studied further.

```
FileOutputStream fos =
  new FileOutputStream(strFile, true);
PrintWriter prt =
  new PrintWriter(new OutputStreamWriter(fos), true);
prt.println(strInfo);
Prt.close();
```

```
new <java.io.FileOutputStream>
dup
aload_1
iconst_1
invokespecial java.io.FileOutputStream.<init>
astore 4
new <java.io.PrintWriter>
dup
new <java.io.OutputStreamWriter>
dup
aload  4
invokespecial java.io.OutputStreamWriter.<init>
iconst_1
invokespecial java.io.PrintWriter.<init>
astore  5
aload  5
aload_2
invokevirtual java.io.PrintWriter.println
aload  5
invokevirtual java.io.PrintWriter.close
```

Figure 7-32 The Java program and bytecode of writing operation

- No semantic analysis

In the approach described in this thesis, the security-level of output data is calculated from all the security-levels of the data that the output data depends on. That is, our approach requires that the sensitive data should not affect the output data to be sent to the third-party hosts. In many cases, the third-party host cannot retrieve the sensitive data from the output data it received, even the sensitive information affect the output data. Though the bytecode causing such output data should be considered as secure code, our approach will determine the bytecode as malicious code and reject it.

In the following example, the s1, s2, s3 represent the salary of three persons. Suppose these data is sensitive for the host. The average salary can be achieved by the computation of the statement. Although the information flows from the sensitive data to the output data average salary, it is impossible to obtain someone's salary from the average salary. Obviously even the average salary is sent to one third-party host that has no privilege to know any person's salary, the security policy will not be violated. While the action of sending average salary to the third-party host will be detected as one data-leaking in our approach.

```
average: =(s1+s2+s3)/3
```

- Additional Overhead

Though the additional overhead caused by the inserted verification code can be omitted in most cases, it is still a problem for some applications of mobile code where the execution time is a critical factor such as some real-time systems, or the hardware is not so powerful such as mobile phones.

Thus the modification method should be revised to decrease the number of instructions inserted into the mobile code. For example, in the Figure 7-23(b) the instructions at address 378 and 380 can be deleted without causing any impair on the verification. Such redundant instructions are inserted because our modification method analyzes the original instructions and insert code for them one by one. The modification method should be revised to avoid such redundant insertion.

And the modification method should also be revised to reduce the execution time of verification code. For example, in a simple loop without branches, the verification code will executed the same times as the original code. While in such a loop, it is enough to calculate the security-level only once since the relation between variables cannot change. In the case that the number of the loop's execution times is quite large, the execution time can be reduced in a large scale.

Finally the current modification method of our approach inserts verification code into mobile program; so that the original functions and the verification function are executed on the same stack. This situation caused that many instructions have to be inserted to arrange the operands on the stack, so that the original function and the verification function will not affect each other. And these inserted instructions slow down the execution further. If the original function and verification function are executed on different stack, the number of inserted instructions can be reduced.

7.4.4 Applications

Our dynamic verification approach can be implemented to protect the confidential information on the local host in many mobile code systems, such as the agent-based e-commerce systems.

Electronic commerce (e-commerce) is increasingly assuming a pivotal role in many organizations. It offers opportunities to significantly improve the way that businesses interact with both their customers and suppliers. Recently agent-based e-commerce has been researched widely [132, 133 and 134].

In general, according to the nature of the transactions, the following types of e-commerce are distinguished: business-to-business (B2B), business-to-consumer (B2C), consumer-to-business (C2B) and consumer-to-consumer (C2C). In all of the types, the agent can be used as the medium between the two sides of the e-commerce. An example of the agent-based e-commerce system is shown in the Figure 7-33.

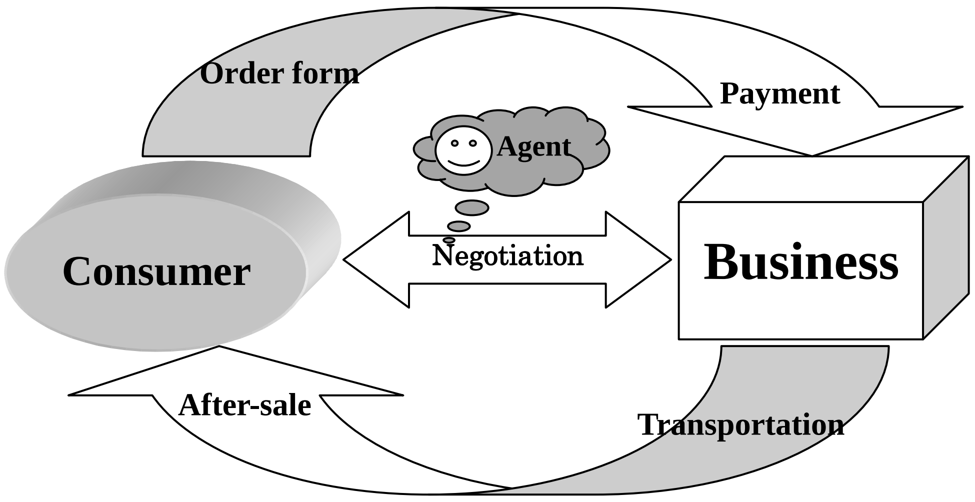


Figure 7-33. An example of agent-based e-commerce system.

Like other agent systems, the security problems in agent-based e-commerce systems can also be divided into two categories: the problem of protecting the host from agent and the problem of protecting the agent from the host. Our approach can deal with the security problem of the former one. As discussed above, our approach can verify the agent correctly and precisely; and make it possible that the user can use more agent-based e-commerce services securely. The actions of the agent arrives at a local host will be traced precisely (by calculating the security-levels of variables in the agent). The agent may access any confidential information on the host to perform its work and provide services to the user, that is, makes a negotiation between the two sides of the trade. When the agent takes confidential information out of the local host, no matter how, by the way of building network directly or by the way of the agent’s movement to other host, our approach can check whether the destination host has the right to get the confidential information. In this way, the user can use the e-commerce service securely without worrying about the leak of his confidential information.

For example, an agent-based personal loan application system is shown in Figure 7-34. The consumer uses an agent to apply for a loan from some company. When the agent arrives at the intermediary server, it submits the loan application to the server. Then the server judges whether to permit the loan application under the help of the credit management server. At last the agent goes back to the consumer with the application result. In this process, our approach can be applied to protect the confidential information of the loan company from being leaked to some malicious consumers. Our approach will not block any agent even it access some confidential information when the agent is executed on the host. Instead, our approach traces the information flow in the agent and records what information the agent gets from the host. Before the agent leaves the intermediary server or sends information out, our approach will check whether the confidential information is sent out. In this way, the loan company can receive more loan application without any loss of confidentiality.

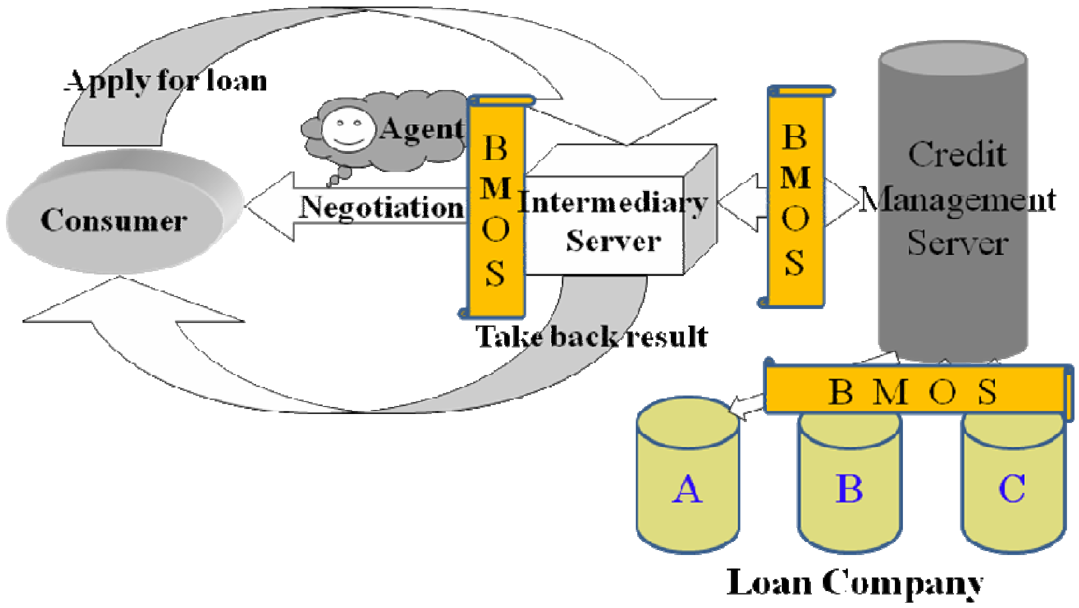


Figure 7-34. An example of applying BMOS in e-commerce system.

8 Conclusion

8.1 Summary

An innovative dynamic approach of information security is described in this thesis. This approach is able to provide protection of data confidentiality of the host by verifying the Java mobile code downloaded dynamically in the runtime. It analyses and traces information flow inside the Java bytecode and checks if there are any data-leaking caused in the Java mobile code that may potentially destroy the data confidentiality of the host. Our dynamic verification approach improves the verification precision and practicability. With our approach, the user can use more mobile code without worrying about the leak of his information.

Traditional host protecting approaches, such as type-system approaches, tend to confine untrusted mobile code from doing harm to the host by restricting the action that the code can do on the host. These security policies are helpful in keeping untrusted code in checked but unfortunately they have the side effect of precluding a large number of useful applications of mobile code.

The existing verification approaches for mobile code are almost static ones which verify the mobile code before the execution. Because of their inherent limitation, static approaches will verify some mobile programs as malicious for the code that will not be executed in the runtime, and reject such secure mobile programs. Such mistake made by static approaches will also preclude many useful application of mobile code.

To overcome the two verification precision problems above, we put forward one dynamic verification approach based on the theory of secure information flow. Compared with the traditional type-system approaches, the advantage of our approach is that our approach protects the host confidentiality while put less restriction on mobile code. We analyze the security requirement in mobile code systems well and put forward a security model suitable to the mobile code environment. In our security model the information flow in the mobile code is just traced and recoded. We do not set any restriction to the information transferring in the mobile code. Only when the mobile code tries to send information out to some third party, we check whether the action causes a data-leaking based on the information we collected. To implement our security model, we define the semantics rules used to trace and record information and give the algorithms to locate implicit and explicit blocks in Java bytecode. Considering that the data-leaking can only be caused by the output operation, the verification precision is improved by our approach.

Compared with those static verification approaches, our approach is dynamic and implements the verification during the execution of the mobile code. By this way, our approach only verifies the code that is actually executed during the runtime; and avoids the misjudgment of considering the mobile code as malicious for some code what will not be executed. Thus compared with static approaches, the verification precision is improved further. The dynamic verification of our approach is implemented by the technique of bytecode modification. That is, the verification code is inserted into the mobile code; and the verification function is executed as well as the original function in the runtime.

Furthermore the information flow in the exception handling and the recursion calling can also be traced and verified by our approach, which is too difficult to achieve for static approaches. The exception handling has been studied in many works by now. However those works are almost exception analysis in terms of high-level languages. In this thesis, we analyze the exception handling in the Java bytecode and give the algorithms to locate the blocks in the *try* statement in Java bytecode. We analyze the information flow in the exception handling and give the methods to deal with both intra-procedural and inter-procedural information transferring caused by the exception handling. The ability to deal with the exception handling and the recursion calling makes our approach more practicable.

In this thesis, we introduce the prototype verification system implementing our verification approach, which is called BMOS. And an example of applying BMOS in one agent system is discussed, too. By studying the verification of several example agents by our approach, the performance efficiency and security verification precision of our approach are proved.

In this thesis, we focus our research on the Java mobile code. In fact, the security model and the modification method can also be applied to other mobile code in the bytecode format. As for as defining the code inserted for each kind of instruction, our dynamic approach can be used to verify the mobile code build in other language.

We believe that our research, especially the research of dynamic verification, is an instructive attempt for bytecode verification.

8.2 Future Work

This thesis describes a novel dynamic approach to verify the mobile code security, which achieves better verification precision than static approaches. While there are still a plenty of work to be done for developing a practical system and enforcing more security properties by our approach.

- Input/output operation disclosing. In order to verify the bytecode program, the security-level of the objects should be retrieved when input operations occur and the security verification rule should be certified when output operations occur. However these input/output operations are not a simple instruction. The input/output operations consist of a sequence of operations. There are various patterns perform these operations. The disclosing of these behaviors will be the next research topic.

- Meliorate the modification method to trace all the information in the array. By now most of the information held by one array can be dealt with in our approach. But the information held by the array reference itself and the array's length can not be traced. Although we can add special security-containers for the one-dimension array to solve the problem, it become hardly difficult to add such security-level containers in the case of multi-dimension array because the numbers of elements in each dimension are arbitrarily different and it is not easy to calculate, load and store the security-levels of the arrays in one multi-dimension array. To build an appropriate construct to transfer the security-levels with the information in the array is the key to deal with this problem.

- Reduce the execution time increment. As mentioned above, the additional execution overhead caused by the inserted verification code should be reduced in order to improve the practicability of our approach. The modification method should be revised to reduce the number of inserted instructions, the number of execution times of inserted instructions and separate the operand stacks of the original function and the verification function.

- In this thesis, only the conditional transfer instruction is concerned when detecting implicit information flow. In a real situation, there are many other kinds of covert channels that may also lead to implicit information flows, such as termination channels, timing channels, probabilistic channels, resource exhaustion channels and power channels. All these channels should be considered in the future study for host security.

- Multiple mobile code owner policies. In our security model, the security policy is set for all mobile code from other hosts. In a more precise model, the security policies should be defined according to the origin of the individual mobile code, in another word, different mobile code programs are applied to different security policies even they immigrate from the same host.

References

[1] Abadi, Michael Burrows, Butler Lampson, and Gordon D. Plotkin, “A Calculus for Access Control in Distributed Systems.” ACM Transactions on Programming Languages and Systems, 15(4), pp.706–734, September 1993.

[2] Adl-Tabatabai, Geoff Langdale, Steven Lucco, and Robert Wahbe, “Efficient and Language-Independent Mobile Programs”, Proceedings of ACM SIGPLAN’96 Conference on Programming Language Design and Implementa-tion (PLDI’96), pp.127–136, May 1996.

[3] Aho, R. Sethi, and J. D. Ullman, “Compilers Principles, Techniques, and Tools”, Addison-Wesley, 1986.

[4] Amtoft and Anindya Banerjee, “Information Flow Analysis in Logical Form”, 11th Static Analysis Symposium, Verona, Italy, Springer-Verlag, pp.100–115, August 2004.

[5] Andrews and Richard P. Reitman, “An Axiomatic Approach to Information Flow in Programs”, ACM Trans. Program. Lang. Syst., 2(1), pp.56–76, 1980.

[6] Avvenuti, C. Bernardeschi, and N. D. Francesco, “Java Bytecode Verification for Secure Information Flow”, SIGPLAN Not., 38(12), pp.20–27, 2003.

[7] Avvenuti, Cinzia Bernardeschi and Nicoletta De Francesco, “Java Bytecode Verification for Secure Information Flow”, SIGPLAN Not., 38(12), pp.20–27, 2003.

[8] Aycock, “A Brief History of Just-In-Time”, ACM Computing Surveys, 35(2), pp. 97–113, June 2003.

[9] Banatre, C. Bryce and D. Le M’etayer, “Compile-Time Detection of Information Flow”, Sequential Programs, 1994.

[10] Banerjee and D. Naumann, “Secure Information Flow and Pointer Confinement in a Java-Like Language”, 2002.

[11] 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, March 2002.

[12] Bell and L. J. LaPadula Leonard J. La Padula, “Secure Computer Systems: A Mathematical Model”, volume II, 1975.

[13] Bell and Leonard J. LaPadula, “Secure Computer Systems: Mathematical Foundations”, Technical Report 2547 (Volume I), MITRE, March 1973.

[14] Bernardeschi and N. D. Francesco, “Combining Abstract Interpretation and Model Checking for Analyzing Security Properties of Java Bytecode”, Third International Workshop on Verification, Model Checking and Abstract Interpretation Proceedings, pp.1–15. LNCS 2294, Venice, January 2002.

[15] 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), pp.391–398, 2002.

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

[17] Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, David Becker, Marc Fiuczynski, Craig Chambers, and Susan Eggers, "Extensibility, Safety and Performance in the SPIN Operating System", Proceedings of the 15th ACM Symposium on Operating System Principles, pp.267–284,Copper Mountain, Colorado, December 1995.

[18] Bian, Ken Nakayama, Yoshitake Kobayashi, and Mamoru Maekawa, "Java Mobile Code Security by Bytecode Analysis", ECTI Transactions on Computer and Information Technology.

[19] Bian, Ken Nakayama, Yoshitake Kobayashi, and Mamoru Maekawa ,"Mobile Code Security by Java Bytecode Dependence Analysis", Proceedings of the International Symposium on Communications and Information Technologies 2004 (ISCIT 2004), Sapporo, Japan, pp.923-926 October 26- 29 2004.

[20] Binder, W. "Design and implementation of the J-SEAL2 mobile agent kernel", 2001 Symposium on Applications and the Internet, San Diego, CA, USA 2001.

[21] Bytecode Engineering Library (BCEL), http://bcel.jakarta.jp/

[22] Cai, P. Gloor, and S. Nog, "DataFlow: A Workflow Management System on the Web Using Transportable Agents", Technical Report TR96-283, Dept. of Computer Science, Dartmouth College, Hanover, N.H., 1996.

[23] Carzaniga, G. Pietro Picco, and G. Vigna, "Designing distributed applications with mobile code paradigms", Proceedings of the 19thInternational Conference on Software Engineering(ICSE'97), 1997.

[24] Chambers, Igor Pechtchanski, Vivek Sarkar, Mauricio J. Serrano, Harini Srinivasan, "Dependence Analysis for Java", LCPC, pp.35-52, 1999

[25] Chander, J. C. Mitchell, and I. Shin, "Mobile Code Security by Java Bytecode Instrumentation", DARPA Information Survivability Conference and Exposition (DISCEX II'01), Volume II-Volume 2.

[26] Chander,A., Mitchell, J.C., Shin, I, "Mobile code security by Java bytecode instrumentation", 2001 DARPA Information Survivability Conference & Exposition (DISCEX II), Anaheim, CA, USA, 2001.

[27] Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska, "Sharing and Protection in A Single-Address-Space Operating System", ACM Trans-actions on Computer Systems, 12(4): 271–307, November 1994.

[28] Chess, "Security Issues in Mobile Code Systems", Mobile Agents and Security, volume 1419, Lecture Notes in Computer Science, Springer-Verlag, 1998.

[29] Chess, B. Grosof, C. Harrison, D. Levine, and C. Paris, "Itinerant agents for mobile computing", IEEE Personal Communications, vol. 2, no. 5, pp.34-49, Oct. 1995.

[30] Chiueh, Ganesh Venkitachalam, and Prashant Pradhan, "Integrating Segmentation and Paging Protection for Safe, Efficient and Transparent Software Extensions", Proceedings of the 17th ACM Symposium on Operating Systems

Principles, pp.140–153, Charleston, South Carolina, December 1999.

[31] Cifuentes, “Reverse Compilation Techniques”, PhD thesis, Queensland University of Technology, 1994.

[32] Clausen, L.R, “A Java bytecode optimizer using side-effect analysis”, Concurrency: Practice and Experience 9, pp.1031–1045, 1997.

[33] Cohen, G., Chase, J., Kaminsky, D., “Automatic program transformation with JOIE”, Proceedings of the 1998 Usenix Annual Technical Symposium, New Orleans, Louisiana, pp.167–178, 1998.

[34] Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K., “Efficiently computing static single assignment form and the control dependence graph”, ACM Transactions on Programming Languages and Systems 13, pp.451–490, 1991.

[35] Czajkowski, G., von Eicken, T.: JRes, “A resource accounting interface for Java”, Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, Vancouver, British Columbia pp.21–35, 1998.

[36] Darvas, Reiner H¨ahnle and David Sands, “A Theorem Proving Approach to Analysis of Secure Information Flow”.

[37] Denning and P. J. Denning, “Certification of Programs for Secure Information Flow”, Communications of the ACM, 20(7), pp.504–513, 1977.

[38] Denning and Peter J. Denning, “Certification of Programs for Secure Information Flow”, Communications of the ACM, 20(7), pp.504–513, July 1977.

[39] Denning, “A Lattice Model of Secure Information Flow”, Comm. ACM, 19(5), pp.236-243, 1976.

[40] Denning, “Cryptography and Data Security”, Addison-Wesley, 1982.

[41] Deutsch and C. A. Grant, “A Flexible Measurement Tool for Software Systems”, Information Processing, 71, pp.320–326, 1972.

[42] Engler, M. Frans Kaashoek, and James O’Toole Jr, “Exokernel: An Operating System Architecture for Application-Level Resource Management”, Proceedings of the 15th ACM Symposium on Operating System Principles, Copper Mountain, Colorado, December 1995.

[43] Erlingsson and Fred B. Schneider, “IRM Enforcement of Java Stack Inspection”, Proceedings of the 2000 IEEE Symposium on Security and Privacy, pp.246–255, Berkeley, California, May 2000.

[44] Erlingsson and Fred B. Schneider, “SASI Enforcement of Security Policies: A Retrospective”, Proceedings of the 1999 New Security Paradigms Workshop, pp. 87–95, Caledon Hills, Ontario, Canada, September 1999.

[45] Erlingsson.U, Schneider,F.B. “IRM enforcement of Java stack inspection”, Proceedings of the 2000 IEEE Symposium on Security and Privacy, Berkeley, California, pp.246–255, 2000.

[46] Evans and Andrew Twyman, “Flexible Policy-Directed Code Safety”, Proceedings of the 1999 IEEE Symposium on Security and Privacy, pp.32– 47, Oakland, California, May 1999.

[47] Farmer, Joshua D. Guttman and Vipin Swarup, “Security for Mo-bile Agents: Authentication and State Appraisal”, Proceedings of the Fourth European Symposium on Research in Computer Security (ESORICS’96), vol-ume 1146, Lecture Notes in Computer Science, pp.118–130, Springer-Verlag, Rome, Italy,

September 1996.

[48] Faulkner and Ron Gomes, “The Process File System and Process Model in UNIX System V”, Proceedings of the USENIX Winter 1991 Conference, pp. 243–252, Dallas, Texas, January 1991.

[49] Florio, R. Gorrieri, and G. Marchetti, “Coping with Denial of Service due to Malicious Java Applets”, Computer Communications, 23(17), pp.1645–1654, November 2000.

[50] Focardi and S. Rossi, “Information Flow Security In Dynamic Contexts”, 2002.

[51] Franz, “Code-Generation On-the-Fly: A Key to Portable Software”, Doc-toral Dissertation No. 10497, ETH Zurich, 1994.

[52] Glew and Greg Morrisett, “Type-Safe Linking and Modular Assembly Language”, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL’99), pp.250–261, San Antonio, Texas, January 1999.

[53] Goldszmidt and Y. Yemini, “Distrisbuted Management by Delegation”, Proc. 15th Int’l Conf. Distributed Computing, June 1995.

[54] Gong, Gary Ellison and Mary Dageforde, “Inside Java 2 Platform Security: Architecture”, API Design, and Implementation, Addison-Wesley, 2nd edition, 2003.

[55] Gosling, Bill Joy, Guy Steele, and Gilad Bracha, “The Java Language Specification”, Addison-Wesley, 2nd Edition, 2000.

[56] Gray, “Agent Tcl: A Flexible and Secure Mobile Agent System”, Pro-ceedings of the 4th Annual USENIX Tcl/Tk Workshop, pp.9–23, Monterey, California, July 1996.

[57] Gray, D. Kotz, S. Nog, D. Rus and G. Cybenko, “Mobile Agents for Mobile Computing”, Proc. Second Aizu Int’l Symp. Parallel Algorithms/Architectures Synthesis, Fukushima, Japan, Mar 1997.

[58] Harrison, D.M.Chess, A.Kershenbaum, “Mobile Agents: Are they a good idea?”, IBM Research Report, T.J.Watson Research Center, NY, 1995.

[59] Hawblitzel, C., Chang, C.C., Czajkowski, G., Hu, D., von Eicken, T, “Implementing multiple protection domains in Java”, USENIX Annual Technical Conference, New Orleans, Louisiana, USENIX, 1998.

[60] Hawblitzel, Chi-Chao Chang, Grzegorz Czajkowski, Deyu Hu and Thorsten von Eicken, “Implementing Multiple Protection Domains in Java”, Proceedings of the 1998 USENIX Annual Technical Conference, New Orleans, Louisiana, June 1998.

[61] Heintze and J. G. Riecke, “The SLam Calculus: Programming with Secrecy and Integrity”, Proc. ACM Symp. on Principles of Programming Languages, pp. 365–377, Jan. 1998.

[62] Heiser, Kevin Elphinstone, Jerry Vochteloo, Stephen Russell and Jochen Liedtke, “The Mungi Single-Address-Space Operating System”, Software Practice and Experience, 28(9), pp.901–928, July 1998.

[63] Jaeger, Atul Prakash, Jochen Liedtke and Nayeem Islam, “Flexible Control of Downloaded Executable Content”, ACM Transactions on Information and System Security, 2(2), pp.177–228, May 1999.

[64] Joshi and K. Rustan M. Leino, “A Semantic Approach to Secure Information

Flow”, Science of Computer Programming, 37(1–3), pp.113–138, 2000.

[65] Knabe. “Language Support for Mobile Agents”, PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA15213-3891, USA, December 1995.

[66] Kobayashi, Keita Shirane, “Type-based Information Flow Analysis for a Low-level Languages”, Computer Software, Vol.20, No.2 pp.2-21, 2003.

[67] 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, Lecture Notes in Computer Science 2229, pp. 292-303, Nov. 2001.

[68] 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), pp.961-973, Oct. 2002.

[69] LaPadula and D. Elliot Bell, “Secure Computer Systems: A Mathematical Model”, Technical Report 2547 (Volume II), MITRE, May 1973.

[70] Lee and Benjamin G. Zorn. “BIT: A Tool for Instrumenting Java Bytecodes”, The USENIX Symposium on Internet Technologies and Systems, pp.73-82, 1997.

[71] Lee, H.B., Zorn, B.G., “BIT: A tool for instrumenting java bytecodes”, USENIX Symposium on Internet Technologies and Systems, Monterey, California, USA ,1997.

[72] Leroy, “Java Bytecode Verification: Algorithms and Formalizations”, J. Autom. Reasoning 30(3-4), pp.235-269, 2003.

[73] Liang and Gilad Bracha, “Dynamic Class Loading in the Java Virtual Machine”, Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA’98), pp.36–44, Vancouver, British Columbia, October 1998.

[74] Limongiello, R. Melen, M. Roccuzzo, A. Scalisi, V. Trecordi, and J. Wojtowicz, “ORCHESTRA: An Experimental Agent-Based Service Control Architecture for Broadband Multimedia Networks”, GLOBAL Internet‘96, Nov. 1996.

[75] Lindholm and Frank Yellin, “The Java Virtual Machine Specification”, Ad-dison Wesley, 2nd edition, 1999.

[76] Lu, K. Nakayama, Y. Kobayashi, and M. Maekawa, “Java Mobile Code Dynamic Verification by Bytecode Modification for Host Confidentiality”, International Journal of Network Security.

[77] Lu, K. Nakayama, Y. Kobayashi, and M. Maekawa, “Verification for Host Confidentiality by Abstract Interpretation in Mobile Code Systems”, IEE Mobility Conference 2005, Guangzhou, China, Nov. 2005.

[78] Lu, K. Nakayama, Y. Kobayashi, M. Maekawa, “Abstract Interpretation for Mobile Code Security”, IEEE Proceedings of International Symposium on Communications and Information Technologies 2005 (ISCIT 2005), pp.1068-1071, Beijing, China, Oct. 2005.

[79] Lucco, Oliver Sharp and Robert Wahbe, “Omniware: A Universal Sub-strate for Web Programming”, Proceedings of the 4th International World Wide Web Conference, Boston, Massachusetts, December 1995.

[80] Magedanz, K. Rothermel and S. Krause, “Intelligent Agents: An Emerging Technology for Next Generation Telecommunications?”, INFOCOM’96, San Francisco, Mar 1996.
[81] Marquez, A., Zigman, J.N., Blackburn, S.M, “A fast portable orthogonally persistent Java”, Software: Practice and Experience Special Issue: Persistent Object Systems 30, pp.449–479, 2000.
[82] Menezes, Paul C. van Oorschot and Scott A. Vanstone, “Handbook of Applied Cryptography”, CRC Press, 1996.
[83] Merz and W. Lamersdorf, “Agents, Services, and Electronic Markets: How Do They Integrate?”, Proc. Int’l Conf. Distributed Platforms, IFIP/IEEE, 1996.
[84] Morrisett, David Walker, Karl Crary and Neal Glew, “From System F to Typed Assembly Language”, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL’98), pp.85–97, SanDiego, CA., January 1998.
[85] Morrisett, Karl Crary, Neal Glew and David Walker, “Stack-Based Typed Assembly Language”, Workshop on Typesin Compilation, pp.95–118, Kyoto, Japan, March 1998.
[86] Myers and B. Liskov, “A Decentralized Model for Information Flow Control”, Proc. ACM Symp. on Operating System Principles, pp.129–142, Oct. 1997.
[87] Necula and Peter Lee, “Safe kernel Extensions without Run-Time Check-ing”. Proceedings of the Second Symposium on Operating System Design and Implementation (OSDI’96), pp.229–243, Seattle, Washington, October 1996.
[88] Necula and Peter Lee, “Safe, Untrusted Agents Using Proof-Carrying Code”, Mobile Agent Security, volume 1419, Lecture Notes in Computer Science Springer-Verlag, 1998.
[89] Necula and Peter Lee, “The Design and Implementation of A Cer-tifying Compiler”, Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI’98), pp.333–344, Montreal, Quebec, November 1998.
[90] Necula, “Proof-Carring Code”, ACM Symposium on Principles of Programming Languagues(POPL), 1997.
[91] Nystrom, N.J, “Bytecode level analysis and optimization of Java classes”, Master’s thesis, Purdue University, 1998.
[92] Object Design Inc. Object-Store PSE Resource Center, 1998. http://www.odi.com/ content/products/PSEHome.html.
[93] Ousterhout, “Tcl and the Tk Toolkit”, Addison-Wesley, 1994.
[94] Ousterhout, Jacob Y. Levy, and Brent B. Welch, “The Safe-Tcl Secu-rity Model”, Mobile Agents and Security, volume 1419, Lecture Notesin Computer Science, Springer-Verlag, 1998.
[95] Palsberg and P. Ørbæk, “Trust in the λ-Calculus”, Proc. Symposiumon Static Analysis, pp. 983 in LNCS, pp. 314–329, Springer-Verlag, Sept. 1995.
[96] Pandey, R., Hashii, B., “Providing fine-grained access control for Java programs”, 13th Conference on Object-Oriented Programming (ECOOP’99), No.1628, Lecture Notes in Computer Science, Lisbon, Portugal, Springer-Verlag, 1999.
[97] Paoli, Andre L. Dos Santos and Richard A. Kemmerer, “Web Browsers and

Security", Mobile Agents and Security, volume 1419, Lecture Notes in Computer Science, Springer-Verlag, January 1998.

[98] Podgurski and Lori A. Clarke, "A Formal Model of Program Dependences and Its Implications for Software Testing, Debugging, and Maintenance". IEEE Transactions on Software Engineering, 16(9), pp.965–979, September 1990.

[99] Rational Software Corporation. Purify,1998. http://www.pure.com/products/purify.

[100] Rees, "A Security Kernel Based on the Lambda-Calculus", A. I. Memo 1564, MIT, 1996.

[101] Rouaix, "A Web Navigator with Applets in Caml", Proceedings of the 5th International World Wide Web Conference, pp.1365–1371, Paris, France, May 1996.

[102] Rudys and Dan S. Wallach, "Termination in Language-Based Systems", ACM Transactions on Information and System Security, 5(2), pp.138–168, May 2002.

[103] Rudys and Dan S. Wallach, "Transactional Rollback for Language-Based Systems", Proceedings of the International Conference on Dependable Systems and Networks (DSN'02), pp.439–448, Washington, D.C., June 2002.

[104] Sabelfeld and A. C. Myers, "Language-Based Information-Flow Security", IEEE Journal on Selected Areas in Communications, 21(1), pp.5--19, January 2003.

[105] Sakamoto, T., Sekiguchi, T., Yonezawa, A., "Bytecode transformation for portable thread migration in Java", Proceedings of the Joint Symposium on Agent Systems and Applications /Mobile Agents (ASA/MA). pp.16–28, 2000.

[106] Saltzer and J.H, "Protection and the Control of Information Sharing in MULTICS", Communications of ACM, 17(7), pp.388-402, July, 1974.

[107] Saltzer and M. Schroeder, "The Protection of Information in Computer Systems", Proceedings of the IEEE, 63(9) pp.1278—1308, Sep. 1975.

[108] Samarati and Sabrinade Capitanidi Vimercati, "Access Control: Policies, Models, and Mechanisms", Foundations of Security Analysis and Design: Tutorial Lectures, volume 2171, Lecture Notes in Computer Science, Springer-Verlag, January 2001.

[109] Scales and Kourosh Gharachorloo, "Towards Transparent and Efficient Software Distributed Shared Memory", The Sixteenth ACM Symposium on Operating Systems Principles, 1997.

[110] Sekar, V. N. Venkatakrishnan, Samik Basu, Sandeep Bhatkar, Daniel C. DuVarney, "Model-Carrying Code: A Practical Approach for Safe Execution of Untrusted Applications", SOSP, pp.15-28, 2003.

[111] Seltzer, Yasuhiro Endo, Christopher Small and Keith A. Smith, "Deal-ing with disaster: surviving misbehaved kernel extensions", Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation, Seattle, Washington, October 1996.

[112] Sinha and Mary Jean Harrold, "Analysis and testing of programs with exception-handling constructs". IEEE Trans. on Software Engineering, 26(9):849-871, 2000.

[113] Smith and D. Volpano, "Secure Information Flow in A Multithreaded Imperative Language", Proc. ACM Symp. on Principles of Programming

Languages, pp. 355–364, Jan. 1998.

[114] Srivastava and Alan Eustace. “ATOM: A System for Building Customized Program Analysis Tools”, Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation, pp.196-205, June 1994.

[115] Tennenhouse, J.M. Smith, W.D. Sincoskie, D.J. Wetherall and G.J. Minden, “A Survey of Active Network Research”, IEEE Comm., vol. 35, no.1, pp.80–86, Jan 1997.

[116] Vall´ee-Rai, R., Hendren, L., Sundaresan, V., Lam, P., Gagnon, E., Co, P., “Soot – a Java bytecode optimization framework”, Proceedings of CASCON 1999, Mississauga, Ontario, Canada, pp.125–135, 1999.

[117] Venkatakrishnan and R. Sekar, “Empowering Mobile Code Using Expressive Security Policies”, 10th New Security Paradigms Workshop (NSPW), 2002.

[118] Venners, “Inside the Java Virtual Machine”, 1998.

[119] 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.

[120] Wahbe, Steven Lucco, Thomas E. Anderson and SusanL. Graham, “Efficient Software-Based Fault Isolation”, Proceedings of the 14th ACM Symposium on Operating Systems Principles, pp.203–216, Asheville, North Carolina, De-cember 1993.

[121] Wallach and Edward W. Felten, “Understanding Java Stack in Spection”, Proceedings of 1998 IEEE Symposium on Security and Privacy, Oakland.

[122] Wallach, Andrew W. Appel and Edward W. Felten, “SAFKASI: A Security Mechanism for Language-Based Systems”, ACM Transactions on Software Engineering and Methodology, 9(4), pp.341–378, October 2000.

[123] Wallach, Dirk Balfanz, Drew Dean and Edward W. Felten, “Extensible Security Architectures for Java”, Proceedings of the 16th ACM Symposium on Operating Systems Principles, pp.116–128, Saint Malo, France, October 1997.

[124] Welch and R. Stroud, “Using Reflection as a Mechanism for Enforcing Security Policies in Mobile Code”, Proceedings of the Sixth European Symposium on Research in Computer Security, 2000.

[125] Welch and Robert J. Stroud, “Using Reflection as a Mechanism for Enforcing Security Policies on Compiled Code”, Journal of Computer Security, 10(4), pp.399– 432, 2002.

[126] Welch, I., Stroud, R., “Kava–a reflective Java based on bytecode rewriting”, Lecture Notes in Computer Science 1826. Springer-Verlag, 2000.

[127] White, “Mobile Agents”, In Jeffrey Bradshaw, Editor, Software Agents, chapter 19, pp.437–472. AAAI Press/MIT Press, 1996.

[128] Yemini and S. da Silva, “Towards Programmable Networks” IFIP/IEEE Int’l Workshop Distributed Systems: Operations and Management, L’Aquila, Italy, Oct. 1996.

[129] Yemini, “The OSI Network Management Model”, IEEE Comm., pp.20–29, May 1993.

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

[131] Zhao, “Dependence Analysis of Java Byte-code”, Proc. 24th IEEE Annual

International Computer Software and Applications Conference (COMP-SAC'2000), pp.486-491, IEEE Computer Society Press, Taipei, Taiwan, October 2000.
[132] B. Banerjee, A. Biswas, and M. Mundhe, S. Depnath, and S. Sen, "Using Bayesian Networks to Model Agent Relationships", Applied Artificial Intelligence, vol. 14, no. 9, pp. 867-879, 2000.
[133] A. Byde, C. Preist, and N.R. Jennings, "Decision Procedures for Multiple Auctions", Proc. First Int'l Joint Conf. Autonomous Agents and Multi-Agent Systems, pp. 613-620, July 2002.
[134] M. Dastani, N. Jacobs, C.M. Jonker, and J. Treuer, "Modeling User Preferences and Mediating Agents in Electronic Commerce", Agent Mediated Electronic Commerce, F. Dignum and C. Sierra, eds., pp. 163-193, 2001.

Acknowledgments

This dissertation is my doctoral study from October 2003 to March 2007 at the Graduate School of Information System of the University of Electro-communications, Tokyo, Japan. I am grateful to a large number of people who have helped me finish this work.

My sincerest thanks go to my supervisory committee and my examiners for their insight advice and great support. And my most heartful thanks go to Professor Maekawa, my senior supervisor, for being my advisor for this thesis. He led me into this research area and whose insight and vision has directly guided my research over the past four years. Without the benefit from his generous financial support, care and encouragement, this dissertation would have never been written out, and it would be impossible for me to finish my PhD career.

Special thanks to the members in Maekawa lab and Ohsuga lab, for their technical expertise, talent dedication to the craft, and honest feedback.

This dissertation is dedicated to my family. I want to express my gratitude to my parents, who have been supporting me such a long time since I was a little boy. And I also want to thank my wife for helping me throughout the course of my study, especially in the last one year. I cannot finish this dissertation without the love and support from my family.

Author Biography

Dan Lu was born in Heilongjiang China, on October 10th, 198. He graduated from Harbin Engineering University in 2001 and he received the M.S. degree in computer control and application from Harbin Engineering University in 2003. He has been with the Graduate School of Information Systems, University of Electro-communications, Tokyo, Japan, working towards the PhD degree. His research interests include Java Virtual Machine, Java bytecode, and Mobile code.

List of Publication Related to the Thesis

1. Dan Lu, K. Nakayama, Y. Kobayashi, M. Maekawa, "**Abstract Interpretation for Mobile Code Security**", IEEE Proceedings of International Symposium on Communications and Information Technologies 2005 (ISCIT 2005), pp.1068-1071, Beijing, China, Oct. 2005.

2. Dan Lu, K. Nakayama, Y. Kobayashi, and M. Maekawa, "**Verification for Host Confidentiality by Abstract Interpretation in Mobile Code Systems**", IEE Mobility Conference 2005, Guangzhou, China, Nov. 2005.

3. Dan Lu, K. Nakayama, Y. Kobayashi, and M. Maekawa, "**Analysis of Information Flow in Exception Handling of Java Bytecode**", Applied Science and Technology, Vol.34, No.2, 2007, pp. 28-30

4. Dan Lu, K. Nakayama, Y. Kobayashi, and M. Maekawa, "**Java Mobile Code Dynamic Verification by Bytecode Modification for Host Confidentiality**", International Journal of Network Security, Vol. 7, No. 3, 2008, pp. 416-427