Mobile Code Security by Java Bytecode Instrumentation*

Ajay Chander
Computer Science Department
Stanford University
ajayc@cs.stanford.edu

John C. Mitchell
Computer Science Department
Stanford University
mitchell@cs.stanford.edu

Insik Shin
Department of Computer and Information Science
University of Pennsylvania
ishin@cis.upenn.edu

Abstract

Mobile code provides significant opportunities and risks. Java bytecode is used to provide executable content to web pages and is the basis for dynamic service configuration in the Jini framework. While the Java Virtual Machine includes a bytecode verifier that checks bytecode programs before execution, and a bytecode interpreter that performs run-time tests, mobile code may still behave in ways that are harmful to users. We present techniques that insert run-time tests into Java code, illustrating them for Java applets and Jini proxy bytecodes. These techniques may be used to contain mobile code behavior or, potentially, insert code appropriate to profiling or other monitoring efforts. The main techniques are class modification, involving subclassing non-final classes, and method-level modifications that may be used when control over objects from final classes is desired.

1. Introduction

Since its early beginnings in the Green project, the Java language [26] has come a long way in its applicability and prevalence. While its initial adoption was fuelled by the ability to add "active content" to web pages, Java has also become a predominant system and application development language, providing useful capabilities over and above the language features through an extensive set of application programming interfaces (APIs). The APIs simplify programming by providing a rich set of domain-dependent libraries, as well as enabling new programmatic and computational paradigms. As an example, the Java Cryptography API makes it possible for applications to easily implement security protocols for their own needs, while the Jini API provides a specification for Java bytecode based distributed programming. One of the keys to Java's success and appeal is its platform independence, achieved by compilation of source code to a common intermediate format, namely Java Virtual Machine (JVM) bytecode, which can then be interpreted by various platforms. The ability to transport bytecode between JVMs is most commonly encountered while browsing the net, and Java's platform independence ensures a client-independent experience.

Although previous language implementations, such as Pascal and Smalltalk systems, have used intermediate bytecode, the use of bytecode as a *medium of exchange* places Java bytecode in a new light. A networked computer can import and execute Java bytecode in ways that are invisible or partly invisible to the user. For example, a user (or his browser) may execute a Java applet embedded within a page as part of the HTTP protocol, or a client may execute a lookup service proxy as it prepares to join a Jini community. To protect against execution of erroneous or intentionally malicious code, the JVM verifies bytecode properties before execution and performs additional checks at run time. However, these checks only enforce some type correctness conditions and basic resource access control. For example, these tests will not protect against large classes of undesirable run-time behavior, including denial-of-service, compromise of integrity, and loss of sensitive information from password or credit card information files, say. The introduction of new security architectures [8] for Java has allowed for digital signature verification and resource access control through the Permissions framework, but suffers from lack of specificity. A more expressive and fine-grained mechanism which can be customized to a user's security needs and is flexible enough to respond to security holes as they

*Partially supported by DARPA contract N66001-00-C-8015 and ONR grant N00014-97-1-0505.

are discovered, is needed.

The goal of our work is to develop methods for enforcing foreign bytecode properties, in a manner that may be customized easily. In this paper, we propose a technique, called *bytecode instrumentation*, through which we impose restrictions on bytecode by inserting additional instructions that will perform the necessary run-time tests. These additional instructions may monitor and control resource usage as well as limit code functionality. This approach is essentially a form of software fault isolation [24], tailored to the file structure and commands of the Java language. Our technique falls into two parts: *class-level* modification and *method-level* modification. Class-level modification involves substituting references to a class by another class subclassed from it. As this method employs inheritance, it can not be applied to final classes and interfaces. In these cases, method-level modification, which may be applied on a method-by-method basis without regard to class hierarchy restrictions, enforces the safe behavior that we hope from foreign code.

We have implemented these techniques within two contexts, each of which has a different bytecode delivery path. For the case of Java applets transported via the HTTP protocol, instrumentation is done by a network proxy, which in addition can also function as a GUI-customizable firewall to specific sites, Java classes, and tagged advertisements. For Jini service proxies, for which there are only transport interfaces but no specific transport mechanisms, we chose to modify the bytecode at the client's ClassLoader end, before its execution. Figures 5 and 6 summarize the system architecture for these two cases.

The rest of the paper is organized as follows. Section 2 gives examples of mobile code risks which cannot be checked within the scope of the current Java verifier and security model, and discusses the extent of our technique. The bytecode instrumentation technique itself is presented in Section 3. Section 4 explains how the mobile code transport frameworks for Java applets and Jini proxies are augmented to instrument the component bytecodes. Section 5 presents examples of the techniques presented in Section 3, with one illustrative example for each of class-level and method-level modification. We make comparisons with existing work in Section 6, and conclude in Section 7.

2. Mobile Code Risks

We preface our techniques for enforcing Java bytecode properties by examples of harmful behavior to illustrate the risk associated with untrusted mobile code. While these attacks have been around for a while, recent interest in peer to peer computing has added value to individual machine cycles, and one may presume, incentive to deploying mobile code attacks.

The categorisations below should be taken only as indicative, and not exhaustive. We situate the extent of our techniques w.r.t. the various kinds of attack threats posed by mobile code; Section 5 presents more detail for specific examples.

2.1. Denial of Service

The current Java security model provides a Permissions framework to specify the host resources that mobile code may access. However, the extent of use is not monitored, and code which has legitimate use for a certain resource, say the screen, or the audio driver, may abuse this privilege. The system may be rendered useless by greedy techniques: monopolizing and stealing CPU time, grabbing all available system memory, or starving other threads and system processes. Many variants on this theme exist, a common scheme is for the foreign code to spawn a "resource consuming" thread. The runaway thread redefines its `stop` method to execute a loop and effects an "infinite access" to the resource, which may result in annoying to crippling behavior, for example through screen flooding. Often a complete browser or system shutdown becomes the only viable option.

Since the safety of Java runtime system may be threatened by inordinate system resource use, it is useful to have some mechanism to monitor and control resource usage.

2.2. Information Leaks

An applet may subvert its constrained channels of information flow through various means. A possible third-party channel is available with the URL redirect feature. Normally, an applet may instruct the browser to load any page on the web. An attacker's server could record the URL as a message, then redirect the browser to the original destination [5]. Another scenario exploits the ability of an applet to send out email messages [10]. If the web server is running an SMTP mail daemon, a hostile applet may forge email after connecting to port 25.

Time-delayed access to files also can be used as a covert channel [19]. Specifically, if mobile code fragment A, with access to private information is prohibited from accessing the net, information can still be sent out by another mobile code fragment B, which shares a file with A. Intercode communication via storage channels may be detected by system logs, but these are hard to analyze in real time.

It is generally accepted that theoretically feasible covert channels like refreshing a page at uneven time intervals to transmit a sequence of bits, are hard to detect. We ignore such arbitrary and unpredictable information channels, while using our techniques to plug more tractable pathways as in the case of email forgery.

2.3. Spoofing

In a spoofing attack, an attacker creates a misleading context in order to trick a user into making an inappropriate security-relevant decision [7]. For example, some applets display URLs as the mouse navigates over various components of a web page, like a graphic or a link. By convention, the URL is shown in a specific position on the status line. If an applet displays a fake URL, the user may be misled into connecting to a potentially hazardous website. It is also possible to abuse weaknesses in mobile code-fetch conventions to spoof the real place of origin of a code fragment, laying client-side security policies regarding network connections to naught.

Bytecode instrumentation is an effective technique against well-specified attacks, which include denial of service and information leaks via specific pathways. In this sense, its scope is monotonic; newly discovered attack specifications can be added to a client's policy files and any additional bytecodes that match them can be instrumented to enforce safety properties. The reader may like to think of this in virus checking terms, where the safety net widens with addition of new entries in the virus signature files. Bytecode instrumentation thus allows for *content-based* protection, since the modification is a function of the bytecode and the client's safety policy.

We now move on to the technical details of our scheme.

3. Bytecode Instrumentation

Our goal is to design a safety mechanism for Java bytecode that extends the signature based security manager with user-controlled content-based control. The basic idea is to restrict bytecode by the insertion of *sentinel code*. In the examples we have implemented and tested, sentinel code may monitor and control resource usage as well as limit functionality. This approach is a form of software fault isolation [24], adapted to the specific structure and representation of Java bytecode programs.

Our safety mechanism substitutes one executable entity, such as a class or a method, with a related executable entity that performs additional run-time tests. For instance, a class such as `Window` can be replaced with a more restrictive class `Safe$Window` that performs additional security and sanity checks. This replacement must occur before the transported bytecode is loaded within the JVM of the client, and we achieve this at different points in the transport path in our experiments with Java applets and Jini proxies (see Section 4). Note that we will use the prefix `Safe$` to indicate a safe class.

The following sections explain how modified executable entities are inserted in Java bytecode. The modifications may be performed at the level of the class or the method, by modifying the constant pool to replace references to substituted entities by their safe substitutes.

3.1. Class-level Modification

A class such as `Window` can be replaced with a subclass of `Window` (say `Safe$Window`) that restricts resource usage and functionality. For example, `Safe$Window`'s constructor can limit the number of windows that can be open at one time, by calling `Window`'s constructor, and raising an exception when the number of windows opened currently (stored as a private variable in the method) exceeds the limit. Since `Safe$Window` is defined to be a subclass of `Window`, the applet should not notice the change, unless it attempts to create windows exceeding the limit.

This class substitution is done by merely substituting references to class `Window` with references to class `Safe$Window`. When `Safe$Window` is a subtype of `Window`, type `Safe$Window` can be used anywhere type `Window` is expected.

In Java, all references to strings, classes, fields, and methods are through indices into the constant pool of the class file [16]. Therefore, it is the constant pool that should be modified in a Java class file. More specifically, two entries are used to represent a class in the constant pool. A constant pool entry tagged as CONSTANT_Class represents a class while referencing a CONSTANT_Uft8 entry for a UTF-8 string representing a fully qualified name of the class, as in figure 1.

If we replace a class name of a CONSTANT_Uft8 entry, `Window`, with a new class name, `Safe$Window`, the CONSTANT_Class entry will represent the new class, `Safe$Window`, as shown in figure 2.

Substituting a class requires just one modification of a constant pool entry representing a class name string. This is straightforward since a subclass may appear anywhere a superclass is used without any modifications to the program. However, this approach cannot be applied to a final class or an interface class.

3.2. Method-level Modification

In class-level modification, the basic idea is to substitute a potentially harmful method (for example, those that provide direct access to system resources) with a safer version that provides for customized control. Unlike class-level modification, however, there is no relationship between the two methods. This provides more flexibility in that it can be used even when the method is final or is accessed through an interface, but requires more modifications than a simple substitution of methods.

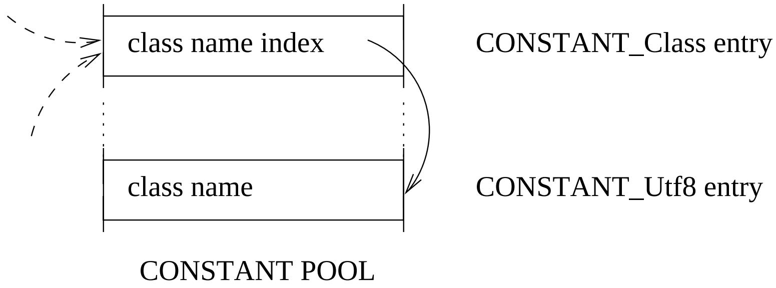

Figure 1. Class references in the constant pool

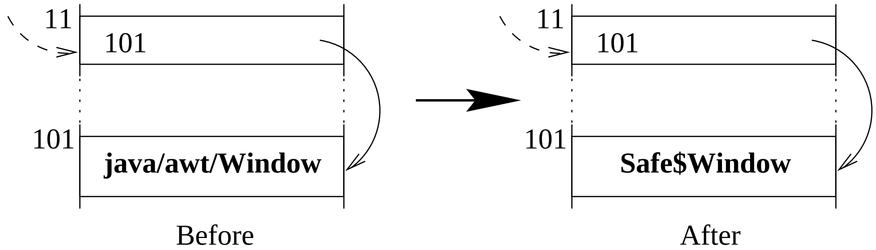

Figure 2. Modfying class references

Before getting into the general mechanism, let us consider a field and a method descriptor within the Java class file format. The field descriptor represents the type of a class or instance variable. For example, the descriptor of an `int` instance variable is simply `I`. Table 1 shows the meaning of some field descriptors.

Descriptor	Type
C	character
I	integer
Z	boolean
L<classname>;	an instance of the class

Table 1. The meaning of the field & method descriptors

The Method descriptor represents the parameters that the method takes and the value that it returns. A parameter descriptor represents zero or more field types, and a return descriptor a field type or `V`. The character `V` indicates that the method returns no value(`void`). For example, the method descriptor for the method

```
void setPriority (Thread t, int i)
```

is

```
(Ljava/lang/Thread;I)V
```

We will describe an instance method in the form <ClassName.MethodNameAndType> and an class(static) method in the form <ClassName:MethodNameAndType>, to distinguish them, though they are not distinguished in the constant pool.

A method such as `Thread.setPriority(I)V` can be replaced with a safer version, say `Safe$Thread:-setPriority(Ljava/lang/Thread;I)V`, which does not allow threads spawned by mobile code to have higher priority than a user-specified upper limit defined in class `Safe$Thread`. The new safeguarding method takes priority of type integer as one of the arguments, and compares it with its upper limit. If the argument is higher, the argument is set to the upper limit. Eventually, the new method invokes `Thread.setPriority(I)V` with the verified argument. Since the new method invokes an instance method mentioned before, a reference to an instance of class `Thread` should be passed to the new method.

3.2.1 Method Reference Modification

It is more difficult to represent a method than a class; consequently the bytecode modification procedure for methods is more involved. A constant pool entry tagged as CONSTANT_Methodref represents a method of a class(a static method) or of a class instance(an instance method). The CONSTANT_Class entry representing the class of which the method is a member and the CONSTANT_NameAndType entry representing the name and descriptor of the method are referenced by CONSTANT_Methodref. In our example, the CONSTANT_Class entry and the CONSTANT_NameAndType entry reference the CONSTANT_Uft8 entries representing java/lang/Thread.setPriority and (I)V, respectively.

Since a new class appears, we should add a new CONSTANT_Uft8 entry representing a new classname string, Safe$Thread, and another new CONSTANT_Class entry referencing the new CONSTANT_Uft8 entry, and then modify the CONSTANT_Methodref entry to refer to the new CONSTANT_Class entry instead of an old CONSTANT_Class entry (which represents the class java/lang/Thread.) Since a method descriptor changes, we also need to add a CONSTANT_Uft8 entry representing a symbolic name for the new method descriptor, (Ljava/lang/Thread;I)V, and then modify the CONSTANT_NameAndType entry to refer to the new CONSTANT_Uft8 entry for the method descriptor. Now the CONSTANT_Methodref entry represents a new method, Safe$Thread:setPriority(Ljava/lang/Thread;I)V, as shown in figure 3.

3.2.2 Method Invocation Modification

Among various Java Virtual Machine instructions implementing method invocations, we are interested in invokevirtual for an instance method invocation and invokestatic for a class(static) method invocation. Both instructions require an index to a CONSTANT_Methodref constant pool entry, but they require slightly different environments. The instance method invocation is set up by first pushing a reference, to the instance which the method belongs to, onto the operand stack. The method invocation's arguments are then pushed onto the stack. The contents of the stack at this point (the environment), which include the reference to the method and the operand stack of the call to Thread.setPriority(I)V, is shown in Figure 4(a). The class method invocation requires an environment much like that of the instance method invocation, except that a reference to the instance is not pushed onto the operand stack. The environment of a call to Safe$Thread:setPriority(Ljava/lang/Thread;I)V is shown in Figure 4(b).

A visual comparison of the two method invocation environments in figure 4 makes the modification required very clear. The contents of the operand stacks are the same, though an instruction for method invocation changes from invokevirtual to invokestatic.

The distinct nature of methods and classes, and their distinct representation in bytecode thus leads to different mechanisms for their respective modification. Method-level modification requires a change in bytecodes in addition to some modifications in the constant pool, whereas class-level modification requires only one change in the constant pool. The difference in the costs of these operations is made up by the difference in the applicability of the schemes; method-level modifications provide finer-grained control, and are the only choice for final classes and interfaces.

4. Application Frameworks

Our experiments with bytecode transfer and untrusted code execution were carried out in the context of Java applets and Jini service proxies. While the same instrumentation mechanisms apply in both cases (and, in general, for arbitrary bytecode), the transport mechanism is modified at different points. In the following, we refer to the code which carries out the bytecode instrumentation as the *bytecode filter*.

4.1. Java Applets

The ubiquity of Java applets and their usefulness comes at the price of an increased security risk owing to unintentional execution of malicious mobile code during web browsing.

There are two obvious ways of inserting a Java bytecode filter into the network and browser architecture. One approach would be to modify the class loader of the Java virtual machine used by the browser. The other is to capture and modify Java bytecode before it enters the browser. The latter provides an easier experimental framework, since a user can easily configure his or her browser to obtain web content through a piece of software called a network proxy. This can be done by a simple modification to a standard browser dialog box. In contrast, modifying the class loader of the Java virtual machine requires installation of special-purpose code in every browser. Moreover, using a standard proxy interface allows us to install a Java-based "security-tuner" interface in every browser, which allows the user to specify their security constraints. Thus a proxy interface provides a simple, customizable and flexible framework for developing and testing Java bytecode filters.

The basic architecture of our system is shown in Figure 5. When the web browser requests a web page or applet, this request goes through the network proxy. The proxy forwards the request to the web server and receives the desired display or executable content. When the web server sends a Java applet, the proxy will pass the applet code to the bytecode filter. The bytecode filter will examine the bytecode for potential risks and modify the bytecode before sending the code for execution to the web browser. In this way, the web browser only receives bytecode that has been screened. The proxy also has access to a repository of Java classes, including secure safe classes that can be substituted for standard library classes and implementations of user-interface methods. The user interface, written in Java and run as an

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS

Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS

Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS

Sync your system to PACER to automate legal marketing.

WHAT WILL YOU BUILD? | **sales@docketalarm.com** | **1-866-77-FASTCASE**

fastcase® Smarter legal research.