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.

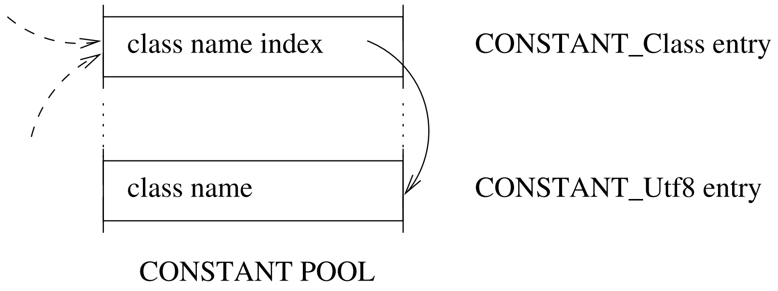


Figure 1. Class references in the constant pool

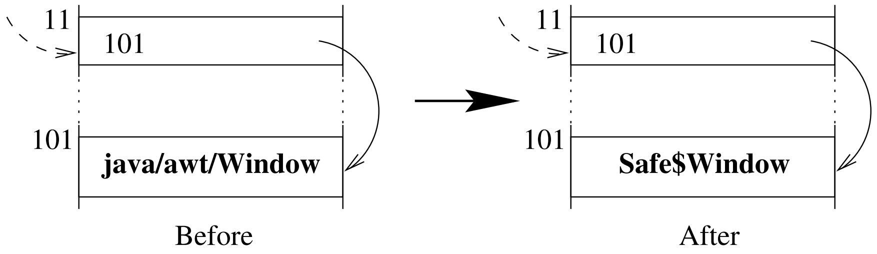


Figure 2. Modfying class references

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

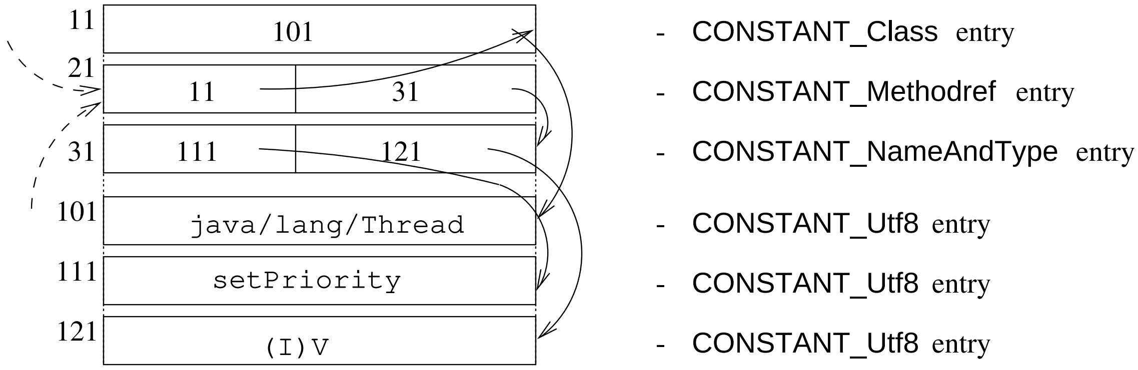
Our experiments with bytecode transfer and untrusted code execution were carried out in the context of Java applets and Jini service proxies. While the same instrumentation mechanisms apply in both cases (and, in general, for arbitrary bytecode), the transport mechanism is modified at different points.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

4.1. Java Applets

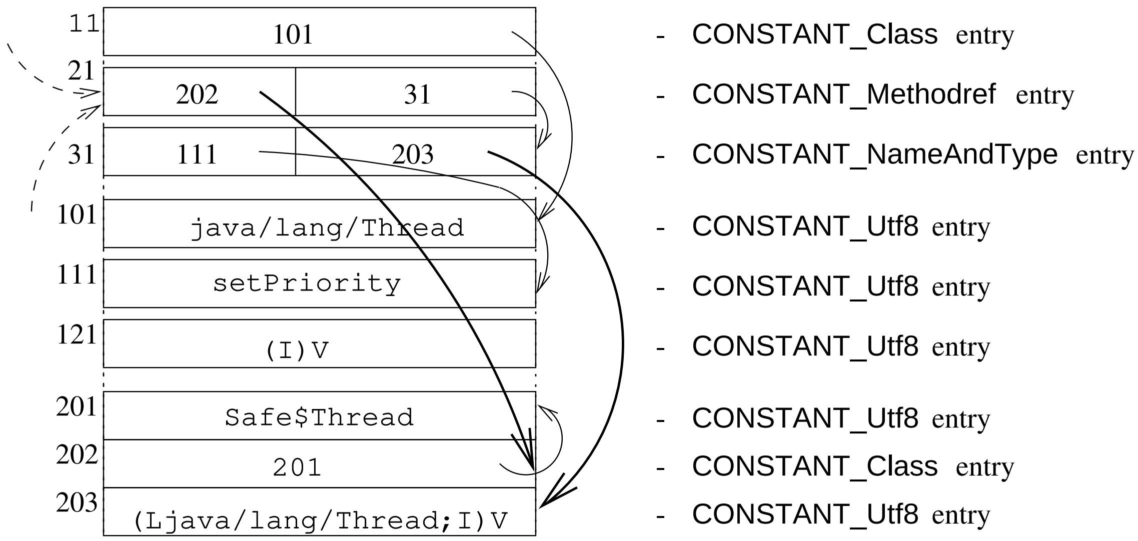
The ubiquity of Java applets and their usefulness comes at the price of an increased security risk owing to unintentional execution of malicious mobile code during web browsing.

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



(a) reference to `Thread.setPriority(I)V`



(b) reference to `Safe$Thread.setPriority(Ljava/lang/Thread;I)V`

Figure 3. Modifying method reference

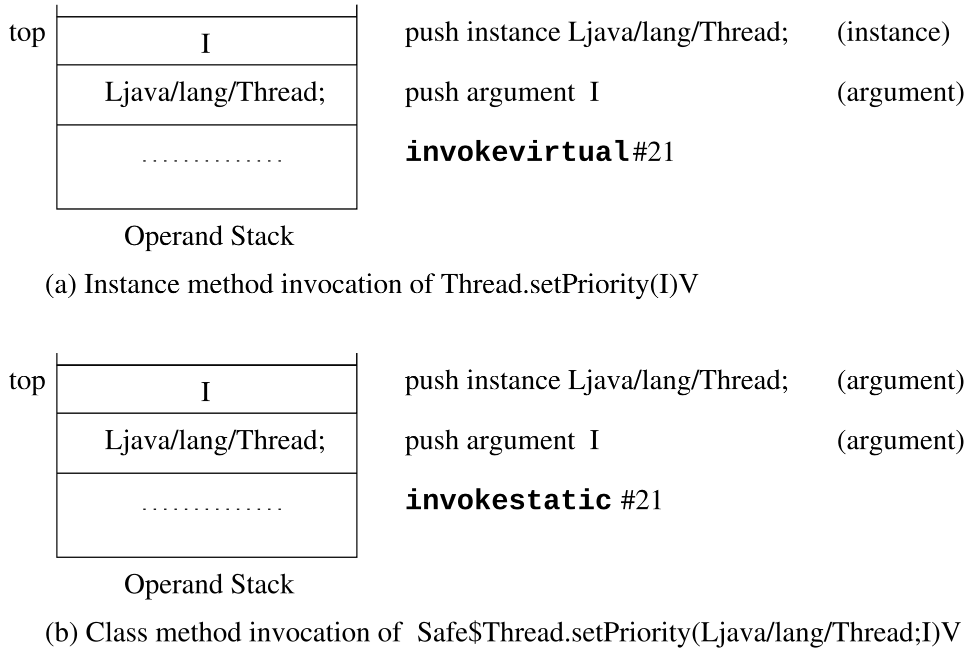


Figure 4. Modifying environments of method invocations

applet under control of the browser, allows the user to customize the security checks performed by the proxy and filter during a web session, without stopping or restarting the browser or Java virtual machine running under control of the browser.

4.2. Jini Proxies

The Jini architecture and application program interfaces provide a convenient and relatively general framework for dynamic discovery and configuration of distributed services. Although Jini is designed to support dynamic federation, the security aspects of Jini are extremely limited. The proliferation of networked small devices and the availability of lean JVMs makes Jini an obvious target for attacks in such a distributed system. We describe the basic Jini mechanisms below, and demonstrate the use of our techniques to ensure safety in this framework.

4.2.1 Protocols

The Jini API extends the Java environment from a single virtual machine to a network of machines. In general, a Jini system contains a set of services each of which offers some functionality to any member of a federation. The initial handshake between the clients and the services is facilitated by *lookup services*. The location and installation of services via this lookup mechanism involves network transfer of Java bytecode. Specifically, if a process on machine A wishes to communicate with a process on machine B, then machine A installs a surrogate stub object that communicates with a corresponding object on machine B through remote method invocation. This surrogate stub object is called a *proxy*. If the object on machine B provides a lookup service, then the stub object installed on machine A is called a *lookup proxy*. If the object on machine B provides a general distributed service, then the stub object installed on machine A is called a *service proxy*.

A Jini client obtains a lookup proxy and a service proxy through the following steps:

Discovery The client broadcasts a request for a lookup service. The lookup service responds with a lookup proxy. This stub object (proxy) allows the client to make further queries of the lookup service.

Query The client queries the lookup service for a service with specific attributes. The lookup service responds with a list of services and supplies a list of attributes associated with each service. v

Selection The client requests and receives a service proxy for a specific service. The result of selection may be a signed or unsigned jar (java archive) file.

The first phase, used in discovery, involves communication with a subnet or multicast group, in order to locate a lookup service. In the second phase, which includes query and selection, the agent communicates with the lookup service through a lookup proxy that was obtained in the discovery

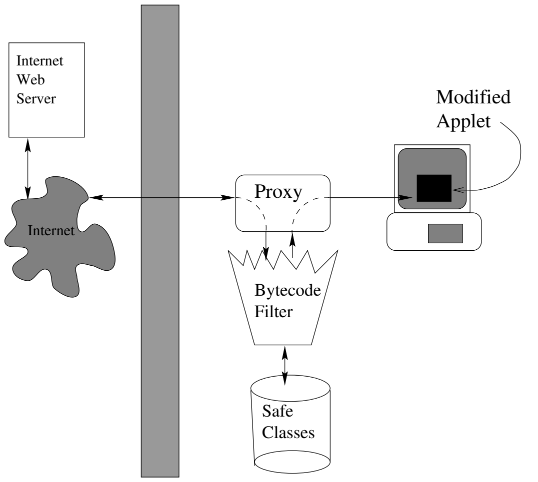


Figure 5. Architecture for instrumenting Java applets

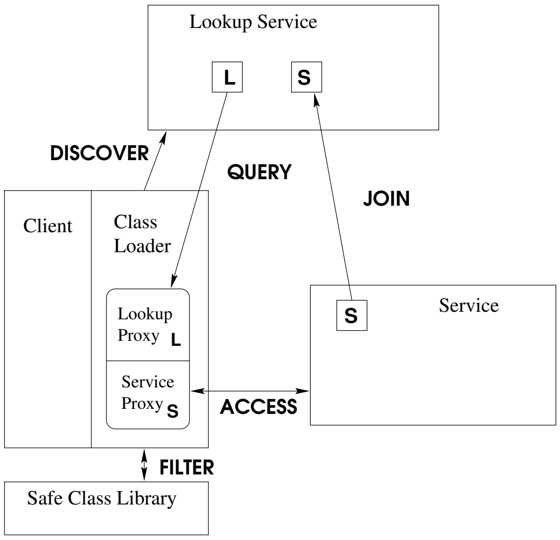


Figure 6. Architecture for instrumenting Jini proxies

phase. While standard Jini uses remote method invocation, other forms of communication could conceivably be substituted. Since the agent must install and execute a lookup proxy (interface or driver), the agent is susceptible to errors or attacks resulting from corrupted or malicious proxy code. In the third phase, the agent interfaces with the installed service through a service proxy. Again, standard Jini relies on remote method invocation, but if the service is distributed, then other communication mechanisms could be appropriate. As with installation of the lookup proxy, the agent is again subject to risks associated with execution of service proxy code received over the network.

4.2.2 Security

The current Jini security model is simply the Java security model. The final jar file received as the result of selection may be signed. The client Java security manager may be configured to provide each lookup proxy or service proxy with specific access rights, according to the signing key used in creation of the jar file. This provides some protection, but suffers from the familiar shortcomings of signature-based security. Specifically, authenticated code may have unintentional safety loopholes, which are only adequately addressed by a content-based protection mechanism. Instrumenting the proxy bytecode at the client's end, before it is loaded by the class loader for execution, provides for client-specified safe behavior.

Figure 6 illustrates the augmented Jini architecture. Note that specific kinds of bytecode filtering may be appropriate for Jini. For example, if Jini service proxies (consisting of java bytecode) are not expected to contain loops, then filtering can be used to detect this and prevent looping. Another desirable property might be that the proxy only engages in secure authenticated network connections with the service it came from, and the code may be filtered to require this. Other client and service specific tests may be performed by instrumenting the proxy with sentinel code, which performs the required code analysis prior to execution of the original code. Jini-specific filtering can be achieved either by writing appropriate configuration files for our existing filter, or extending the filter implementation.

The second piece of the augmented Jini infrastructure is an interception mechanism for proxy bytecode. As described above, for Java applets this was achieved by using a network proxy that intercepted http communication. For Jini, this must now be done using a network proxy or some other mechanism that has access to arguments and return values of Java RMI calls.

4.2.3 Bytecode Interception

The Jini specification only requires the protocols described above to be based on TCP and UDP transport and object serialization, and implementations are free to choose a communication mechanism. In particular, the specification makes no particular reference to the RMI registry or wire protocols. The only requirement is that Jini interfaces preserve the RMI remote interface semantics, which means that they need to only declare `RemoteException` on its methods. This makes the job of identifying a well-known, constant path of remote-class delivery rather difficult. We note though that most Jini communities of services and clients actually use RMI, on account of it being used in Sun's default implementation.

We focus on the relevant parts of the Jini discovery protocol. After the lookup server is found, it sends to the client an object implementing the `net.jini.core.lookup.ServiceRegistrar` interface. This is done in the last phase of the discovery protocol, via a `java.rmi.MarshalledObject`. This `ServiceRegistrar` object is the lookup proxy, and will receive all service proxies on the clients behalf. Accordingly, there are three places where one could intercept the service proxy bytecode:

- during transport from the lookup server to the lookup service proxy,
- within the `ServiceRegistrar` before the serialized object is reconstructed, and
- within the `ServiceRegistrar` after object reconstruction.

The lack of a common wire protocol makes it infeasible to intercept the service object along the network path, thus we intercept the object bytecode in the `ServiceRegistrar`. This is best done before the serialized object is reconstructed, since modifying the bytecode of an object after reconstruction presents problems. For example, it is difficult to maintain the state of the object.

In Java, in order to load a class into the JVM execution, one needs a `java.lang.ClassLoader` object which does the actual loading via the protected final `defineClass` method. Loading a class in the JVM proceeds as follows. If the class name is referenced implicitly during the code execution, the JVM tries to locate in its internal runtime table a class with such name, that was loaded by the same ClassLoader which loaded the class making the reference. If there is no such class, the JVM calls the method `loadClass()` on the `ClassLoader` which loaded the class making the reference. This method first tries to delegate the loading of the class to its parent `ClassLoader` by calling its `loadClass()`, and so forth, up until the bootstrap. If all parents fail to locate the class (not previously loaded by them, and unable to find its .class file), then `loadClass()` calls `findClass()` in the ClassLoader that initiated the loading process. If `findClass` finds the

.class file, it loads it into a byte array, and then calls the final, protected method `defineClass`. If `findClass` fails to locate the class, it throws a `ClassNotFoundException`. In the Java API, there is only one network-aware ClassLoader, java.net.URLClassLoader. Note also that `java.rmi.server.RMIClassLoader` relies internally on a subclass of `URLClassLoader`.

This leads us to two possible ways of intercepting service objects. The first one is to assume that the classes for the downloaded service objects will always be loaded into the JVM by using either the `URLClassLoader` or the `RMIClassLoader`, and then replace all references to `java.net.URLClassLoader` with our extension of it, `jinifilter.SafeURLClassLoader`. The second one is to replace all calls to `java.lang.ClassLoader.defineClass()` with the static `jinifilter.SafeClassLoader.defineClass()`. In our current implementation, we chose to assume that the classes for the downloaded service objects will always be loaded into the JVM by using either the `URLClassLoader` or the `RMIClassLoader`. Note that this is not restrictive, since classes on the disk may be referenced using the file:// protocol.

In summary, our augmented Jini framework for bytecode instrumentation works as follows. A Jini service registers its proxy with a lookup service during the process of registering with it. The discovery and selection of this service by a client results in a serialized object form of the service proxy being sent to the client. The modified class loader at the client invokes the bytecode filter, which instruments classes and methods in the service proxy so that safe classes from the safe class library are used in place of standard classes. This allows us to trace file and network access, prevent denial of service attacks by limiting use of specified resources, and instrument bytecodes for other profiling efforts. In our experiments we have found it straightforward to construct safe classes for these purposes.

5. Examples

We present an example each of class-level and method-level modification in the context of containing bytecode behavior.

5.1. Resource abuse (Screen)

As mentioned in Section 3, foreign code can crash the system by creating more windows than a certain system-dependent limit. To protect against this resource-hog attack, the safety mechanism should keep track of the window creation process.

Since the Java library class `Frame` handles an optionally resizable top-level window, the attack may be contained by restricting the number of times mobile code can invoke the window constructor method.

Since `Frame` is inheritable, this may be achieved by replacing all references to the `Frame` class by references to a safe class subclassed from it, say `Safe$Frame`. `Safe$Frame` creates windows using the constructor methods of `Frame`, while keeping track of the number of windows already open.

```
public class SafeFrame extends Frame {
  private static int numOfFrames = 0;
  public SafeFrame (String title){
    super(title);
    if(numOfFrames < FRAMEMAX)
       numOfFrames++;
    else {
          numOfFrames = 0;
          throw new AWTError
             ("Number of Window Frames
                  exceeded");
         }
  }
};
```

As the code fragment shows, `Safe$Frame` guards the creation of a new window with a check against the current value of `numofFrames`, the number of frames currently open. Since references to `Frame` are substituted with references to `Safe$Frame` before bytecode execution, the applet wont notice any change until it tries to exceed these resource limits.

This technique can clearly be extended into a full-blown tunable resource monitor, customizable to a client's security needs. Safeguarding against resource abuse has a common template:

1. Identify the Java classes which control the resource we are interested in monitoring.

2. Subclass this resource class (assuming it is not final) to create a safe version of the original class, which has provisions for user defined checks.

3. Invoke the bytecode filter on the resource class, so that all references to the original class are replaced with references to the new (safe) subclass.

Note that carrying out the bytecode modification in a network proxy instead of the browser allows many diverse security policies to be managed by a single proxy.

5.2. Compromise of Privacy

An applet is able to disclose the user's confidential information through email, while its web server is running an

SMTP mail daemon. To get rid of this covert channel, the applet should not be able to connect to port 25 on the web server.

A Java library class, Socket, implements a socket for interprocess communication over the network. The constructor methods create the socket and connect it to the specified host and port. Since we want to put restrictions on the constructor methods, we should familiarize ourselves with how they are implemented in Java Virtual Machine.

Java Virtual Machine class instances are created using the JVM's new instruction. Once the class instance has been created and its instance variables have been initialized to their default values, an instance initialization method of the new class instance(<init>) is invoked. At the level of the JVM, a constructor appears as a method with the special compiler-supplied name <init>. For example:

```
Socket create() {
        return new Socket(host_name,
port_number);
}
```

compiles to

0	*new #1*	*Class* java.net.Socket
3	*dup*	
4	*getfield*	*Field* this.host_name java.lang.String
7	*getfield*	*Field* this.port_number I
10	*invokespecial #4*	*Method* java.net.Socket.<init> Ljava/lang/String;I)V
13	*areturn*	

The JVM instruction invokespecial invokes instance methods requiring special handling, such as superclass, private, and instance initialization methods.

Since Socket is a final class in the browser, we replace the constructor methods via method-level modification. Our safe method, Safe$Socket:init, can monitor and control every socket connection. Safe$Socket:init establishes the socket connection upon every request excluding a request to port 25, and returns a new socket object. Safe$Socket:init takes the same argument type as whatever the constructor of Socket takes, but a different return type since it returns the new socket object. So references to Socket.<init>(Ljava/lang/String;I)V are replaced with references to Safe$Socket:init(Ljava/lang/String;I)Ljava/net/Socket;.

Since Safe$Socket:init is a static method, we substitute replace invokespecial with invokestatic. In addition, we should remove a socket object created by new from the stack, since the new method returns a socket object. The modified bytecodes are as follows:

0	*new #1*	*Class* java.net.Socket
3	***pop***	
4	*getfield*	*Field* this.host_name java.lang.String
7	*getfield*	*Field* this.port_number I
10	***invokestatic** #4*	*Method* Safe$Socket.<init> (Ljava/lang/String:I) Ljava/lang/Socket;
13	*areturn*	

5.3 Client Safety Policies

Our example prototypes use a simple safety specification language which is used by the filter to determine the user's requirements. These may be expressed either as lists of classes and methods that need to be modified, or through a GUI which records the "safe" ranges of resource usage for many commonly accessed system resources (screen, audio, network connections, threads, etc.). In terms of the safety specification language, these provide for a conjunction of resource bound conditions. We are studying policy languages which allow for the expression of other logical constructs, for example, XOR. A client may specify that a piece of code opens a network connection or reads a file on the local disk, but not do both. Other temporal constraints which ensure the privacy of the client from mobile code also seem useful to capture within a safety specification language. Given a particular specification language, bytecode instrumentation allows us to bootstrap mobile code with sentinel code which checks for policy compliance before the code is executed.

6. Related Work

There are three general approaches which have been proposed for the safe execution of mobile code. On one end of the spectrum are digital signature based schemes, which associate trust in code with the ability to authenticate the signature provided with it. For example, digital signature mechanisms in the Java Crypto API enable a user to download applets written only by known (and trusted) authors. While signed code provides a greater degree of comfort, the confidence in its safety is based on our knowledge of its author, and not on its content.

On the other end of spectrum are formal-method based approaches which guarantee that mobile code meets certain specifications. Language semantics can be used to enforce safety by guaranteeing that a program can't affect resources that it can't name [3]. However, such semantics should be extended to include the exact conditions and requirements that a security protocol should satisfy, such as resource consumption or information about communication. Necula and Lee introduced *proof-carrying code* [20], where the mobile code carries a proof that it complies with certain invariants or requirements. While such a scheme greatly increases the confidence placed in mobile code environments, the excessive overhead involved with the creation of proofs, and the difficulty of formally specifying security concerns renders these methods somewhat impractical.

Lucco, *et al.*, introduced *software fault isolation* [24] for transforming untrusted mobile code so that it can not escape its fault domain. They showed that memory accesses could be encapsulated with a 5-30% slowdown. The original Java *sandbox* security model prohibited untrusted applets from using any sensitive system service, but the sandbox model turned out to be compromised even with small implementation errors [5]. These observations were followed by some efforts to try and prevent untrusted applets from getting into a local machine. Malkhi *et al.*, proposed the concept of a *playground* [17] for executing untrusted mobile code on a remote protected domain(machine), while using the user's browser as an I/O terminal. The Secure Internet Programming group at Princeton proposed a *Java Filter* [2] for preventing untrusted applets from entering the user's computer; a user could download Java applets only from trusted servers using the Java Filter. Another approach is to use firewalls to filter out all outside applets [18], while allowing trusted internal applets to run.

Our approach is related to software fault isolation. We transform applets through *bytecode instrumentaion*, such that they effectively execute in an environment with guarded access to security-sensitive objects and methods. Such content-based security provides a middle ground between the complexity of formal methods approaches and the all-or-nothing paradigm of signature based security. We believe that bytecode instrumentation is a practical means of achieving specific security ends, and complements other techniques for raising trust in mobile code.

We briefly summarize other approaches to modifying bytecode. The Princeton group modified the browser's `AppletClassLoader` such that it could employ the Java Filter. Malkhi *et al.*, substituted the names of AWT classes with the names of remote versions of corresponding AWT classes, by symbolic manipulation. BIT [15] and JTrek [28] provide Java libraries for modifying Java classes at link-time. The BIT framework allows the insertion of a new method invocation, but doesn't allow the replacement of removal of method calls. The JTrek class library enables developers to add a new method invocation, and to get the value of a variable for modifying and monitoring the activity of a Java class. BCA [11] and JOIE [4] are tools for rewriting Java bytecodes at load-time. The BCA system allows Java class components to be adapted and transformed at the Java class file format level, by adding a method to a class and performing symbolic manipulations, without any operations at the bytecode instruction level. The JOIE toolkit enables modifies bytecode at load time, using a user-extensible class loader. In contrast to our proxy approach, the JOIE toolkit is dependent on current versions of the class `java.lang.ClassLoader`.

In general, the idea of modifying executables is not a new one, and has appeared in other contexts. We list some approaches here for completeness. Using microcode modification, ATUM [1] and MPTRACE [6] generate data and instruction traces and collect them for performing trace-driven simulations in the process of architecture evaluation. By modifying a fully-linked executable, Pixie [29], Epoxie [25], and QPT [13] record the sequence of instructions and data references for profiling and tracing systems. There exist many general purpose modification tools for executables. OM [23] and ATOM [22] are tools running on Alpha AXP under OSF/1 for disassembling the binary into an intermediate form, modifying the intermediate in a machine-independent way, and translating the modified intermediate from into the target binary format. EEF [14] is a C++ library running on SPARC processors under SunOS and Solaris which provides abstractions for analyzing and modifying executable programs in a machine-independent way. Etch [21] is a binary rewriting tool for Win32 applications on Intel x86 processors.

7. Conclusion

This paper presented a technique for modifying bytecode programs, through which users may customize the behavior of foreign java bytecode, and its prototype implementation for protecting against certain kinds of hazardous run-time behavior. Our safety system transforms Java applets and Jini proxies through bytecode instrumentation, in order to perform additional security and sanity checks and provide control over inaccessible objects. We demonstrated through some examples that bytecode modification may address security concerns regarding denial of service, email-forging, URL spoofing, and annoyance attacks.

The bytecode modification techniques explained in this paper fall into two categories, class-level modification and method-level modification. Class-level modification involves replacing references to one class by references to another. In our examples, we replaced references to standard library classes with references to our own customized "safe" subclasses of these classes. Class-level modification involves relatively simple manipulation of bytecode since symbolic names of classes are conveniently stored in the constant pool. Method-level modification provides more flexibility in that it can be used when the method is final or is accessed through an interface, but requires more complicated modifications of method reference and method invoking instructions.

Since our techniques are general enough to be applicable to any Java bytecode, we implemented these techniques within two contexts, each of which has a different bytecode delivery path. In the case of Java applets, instrumentation was done by a network proxy whereas 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 class loader before its execution.

Our experiments suggest a low performance hit of 5-20% based on the mixture of class and method-modifications required in the bytecode, using publicly available libraries (JavaClass) to extract information from, and reconstruct bytecode. Currently, versions of the proxy exist in Java and Python, and we are working towards identifying domain-specific security policies. The design of more expressive safety specification languages is an interesting line of further research. We encourage the reader to visit http://iswim.stanford.edu/bytecode-instrumentation/ for a current version of our work.

Although we presented our technique in the context of the Java security model, we believe that it certainly has a wider range of applicability than the simple security-related examples presented in this paper. For example, it may be useful to explore ways to utilize the technique in other settings, such as interacting with normally inaccessible objects. We also believe that with the flexibility of current Java security models, such as descibed in [8], it may be fruitful to use bytecode modification to enforce chosen restrictions on usage of the security manager.

Acknowledgments

Several people contributed at various points to this work. Amit Patel wrote an initial version of the Java proxy in Python. Vijay Ganesh and Jun Zhai participated in a port of the bytecode filter and proxy to Java. Martin Gavrilov and Lucas Ryan helped with examples and code for instrumenting Jini proxies. Our thanks go to all of them, for their help, and useful discussions.

References

[1] Anant Agarwal, Richard Sites, and Mark Horwitz. ATUM: A New Technique for Capturing Address Traces Using Microcode. In *Proceedings of the 13th International Symposium on Computer Architecture*, 119-127, June 1986.

[2] Dirk Balfanz and Edward W. Felten. A Java Filter. Technical Report 97-567, Department of Computer Science, Princeton University, 1997.

[3] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson Modula-3 language definition. *SIGPLAN Notices*, 27(8), August 1992.

[4] Geoff A. Cohen, Jeffrey S. Chase, and David L. Kaminsky. Automatic Program Transformation with JOIE. In *Proceedings of the 1998 USENIX Annual Technical Symposium*, 1998.

[5] Drew Dean, Edward W. Felten, and Dan S. Wallach. Java Security: From Hotjava to Netscape and beyond. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, May 1996.

[6] Susan J. Eggers, David R. Keppel, Eric J. Koldinger, and Henry M. Levy Techiques for Efficient Inline Tracing on a Shared-Memory Multiprocessor. In *Proceedings of the 1990 ACM Sigmetrics Conference on Measurement and Modelings of Computer Systems*, 8(1), May 1990.

[7] Edward W. Felten, Dirk Balfanz, Drew Dean, and Dan S. Wallach. Web spoofing: An Internet Con Game. Technical Report 540-96, Department of Computer Science, Princeton University, February 1997.

[8] Li Gong. JDK 1.2 Security Architecture. Sun Microsystems, Inc., March 1998.

[9] James Gosling, Bill Joy, and Guy Steels. *The Java Language Specification.* Addison Wesley Publishing Company, Reading, Massachusetts, 1996.

[10] Gary McGraw and Edward W. Felten. Securing Java: Getting Down to Business with Mobile Code. John Wiley & Sons, 2000.

[11] Ralph Keller and Urs Holzle. Binary Component Adaption. In *Proceedings of the 1998 ECOOP*, 1998.

[12] Mark LaDue. Hostile applets home page. http://www.rstcorp.com/hostile-applets/index.html.

[13] James R. Larus and Thomas Ball. Rewriting Executable Files to Measure Program Behavior. *Software, Practice and Experience*, 24(2), February 1994.

[14] James R. Larus and Eric Schnarr. EEL:Machine-Independent Executable Editing. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation* (PLDI), 291-300, June 1995.

[15] Han Bok Lee and Benjamin G. Zorn BIT: A Tool for Instrumenting Java Bytecodes. In *Proceedings of the 1997 USENIX Symposium on Internet Technologies and Systems*, 73-82, December 1997.

[16] Tim Lindholm and Frank Yellin. The Java Virtual Machine Specification. Addison Wesley, 1996.

[17] Dahlia Malkhi, Michael Reiter, and Avi Rubin. Secure Execution of Java Applets using a Remote Playground.

[18] David M. Martin Jr., Sivaramakrishnan Rajagopalan, and Aviel D. Rubin. Blocking Java Applets at the Firewall. In *Proceedings of the 1997 Internet Society Symposium on Network and Distributed System Security*, February 1997.

[19] Nimisha V. Mehta and Karen R. Sollins. Expanding and Extending the Security Features of Java. In *Proceedings of the 7th USENIX Security Symposium*, January 1998.

[20] G.C. Necula and Peter Lee. Safe kernel extensions with run-time checking. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation*, October 1996.

[21] Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, Brian Bershad, and Brad Chen. Instrumentation and Optimization of Win32/Intel Executables Using Etch. In *Proceedings of the 1997 USENIX Windows NT Workshop*, 1-7, August 1997.

[22] Amitabh Srivastava and Alan Eustace. ATOM A System for Building Customized Program Analysis Tools. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation* (PLDI), 49-60, June 1994.

[23] Amitabh Srivastava and David W. Wall. A Practical System for Intermodule Code Optimization at Link-Time. *Journal of Programming Languages*, 1(1), 1993.

[24] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the 14th Symposium on Operating Systems Principles*, December 1993.

[25] David W. Wall. Systems for late code modification. In Robert Gieberich and Susan L. Graham, eds, Code Genration - Concepts, Tools, Techniques, 275-293, Springer-Verlag, 1992.

[26] The Java Language Environment: A White Paper. Sun Microsystems Computer Company, May 1995.

[27] The JDK Class Loader Extensions Framework. `http://java.sun.com/products/jdk/1.2/docs/guide/extensions/spec.html`

[28] DIGITAL JTrek. `http://www.digital.com/java/download/jtrek/index.html`

[29] Tracing with pixie. Technical Report CSL-TR-91-497, Computer Systems Laboratory, Stanford University, November 1991