Java Bytecode Modification and Applet Security*

Insik Shin† John C. Mitchell‡
ishin@cs.stanford.edu mitchell@cs.stanford.edu

Computer Science Department
Stanford University
Stanford, CA 94305

Abstract

While the Java Virtual Machine includes a bytecode verifier that checks bytecode programs before execution, and a bytecode interpreter that performs run-time tests such as array bounds and null-pointer checks, Java applets may still behave in ways that are annoying or potentially harmful to users. For example, applets may mount denial-of-service attacks, forge email or display misleading information in order to trick users. With these concerns in mind, we present techniques that may be used to insert additional run-time tests into Java applets. These techniques may be used to restrict applet behavior or, potentially, insert code appropriate to profiling or other monitoring efforts. The main techniques are class-level modification, involving subclassing non-final classes, and method-level modification, which may be used when control over objects from final classes is desired.

1 Introduction

The Java Language [12] has proven useful for a variety of purposes, including system development and the addition of "active content" to web pages. 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. To protect against execution of erroneous or potentially malicious code, the Java Virtual Machine verifies code properties before execution and performs additional checks at run time. However, these tests will not protect against certain forms of undesirable run-time behavior, such as denial-of-service attacks, irritating audio sounds, or violation of conventions regarding

*submitted as a research paper on language design and implementation, analysis and design methods, software engineering practices, experienced with object-oriented applications and systems, or security.

†contact: Tel: 650-723-9445

‡contact: Tel: 650-723-8634, Fax: 650-725-4671

hypertext links. Moreover, users cannot easily customize the tests that are performed since these are built into the Java Virtual Machine.

The goal of our work is to develop methods for enforcing applet properties, in a manner that may be customized easily. In this paper, we propose a technique, called *bytecode modification*, through which we put restrictions on applets by inserting additional bytecode instructions that will perform the necessary run-time tests. These additional instructions may monitor and control resource usage, limit applet functionality, or provide control over inaccessible objects. While our techniques bear some relation to software fault isolation [13], we check different properties and our code operations are specifically tailored to the file structure and commands of the Java Language. Our technique falls into two parts: *class-level* modification and *method-level* modification. In class-level modification, references to one class are modified to refer to another class. Since this method uses the class inheritance, it is simple and fast, but can not be applied to final classes and interfaces. In these cases, method-level modification is used since it may be applied on a method-by-method basis without regard to class hierarchy restrictions.

We have implemented these techniques in an HTTP proxy server that modifies classes before they are received by the browser. Our proxy server is controlled by a user interface that runs as a Java applet and may be configured to block access to a specific sites, redirect requests for special Java classes or eliminate tagged advertisements.

In Section 2, we discuss several example Java applet attacks that are outside the scope of the current Java verifier and security model. We explain the bytecode modification techniques in Section 3 and present some examples in Section 4. Experimental performance data appears in Section 5, with comparison to related work on safe execution of Java applets in Section 6. We conclude in Section 7.

2 Java Applet Safety

Before describing a series of techniques for modifying Java bytecode programs, we give some motivating examples of hazardous or undesirable Java applets. Each of the problems outlined in this section can be eliminated or contained using our approach.

2.1 Denial of Service Attack

The Java Virtual Machine provides little protection against denial of service attacks. An applet can make the system unstable by monopolizing CPU time, allocating memory until the system runs out, or starving other threads and system processes. For example, an applet may create huge black windows on the screen in such a way that the users cannot access other parts of the screen, or it may open a large number of windows [5]. Many machines have limits on the number of windows that can be open at one time and may crash if these limits are exceeded. Since the safety of the 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 Disclosure of Confidential Information Attack

Most browsers such as Navigator, Internet Explorer and HotJava, provide a network security mode which allows an applet to connect to the web server from which it was loaded. In spite of this security mode, an applet can send some confidential information out through various covert channels. 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 [3]. Another channel is available with an ability that an applet can send out an email message to any machine on the Internet [7]. If the web server is running an SMTP mail daemon, the applet can interact with sendmail after connecting to port 25 on the web server. This allows a hostile applet to forge email. One way to prevent this form of email forgery is to disallow connections to port 25.

Time-delayed access to files also can be used as a covert channel [10]. Specifically, if an applet, A, with access to private information is prohibited from accessing the net, information can still be sent out by another applet, B, which shares a file with applet A. Inter-applet communications using storage channels can be detected by monitoring the actions of applets through logging facilities.

2.3 Spoofing Attack

In a spoofing attack, an attacker creates a misleading context in order to trick a user into making an inappropriate security-relevant decision [4]. Some applets display the URL that will be accessed when the the mouse is held over a graphic or link. By convention, the URL is shown in a specific position on the status line. If an applet displays a fake URL, the user can be misled. This could allow an applet to mislead a user into connecting to a site that is hazardous in some way. Fortunately, this spoofing attack can be controlled by enforcing conventions about the URL displayed on the status line.

2.4 Annoyance Attack

An applet can annoy users with a very noisy sound which never ends. This form of sound attack exploits a useful feature of Java, the ability to play sound in the background. To eliminate an annoying sound, however, users typically must kill the thread playing sound, disable the audio, or quit the browser. All of these can be inconvenient. Another possible annoyance attack is to make the browser visit a given web site over and over, popping up a new copy of the browser each time. One way to manage annoying sounds is to provide the ability to turn the sound off. To do so, the Java runtime system must monitor and control objects with sound.

3 Java Bytecode Modification

This paper presents a safety mechanism for Java applets that is sufficient to solve the problems summarized above. The basic idea is to put restrictions on applets by inserting safeguarding code. In the examples we have implemented and tested, safeguarding code may monitor and control resource usage as well as limit the functionality of applets. This approach is a form of software fault isolation [13], 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$Window` that performs additional security and sanity checks. (We use the prefix `Safe$` to indicate one of our safe classes.) This safety mechanism must be applied before the applet is executed. For convenience in developing a proof of concept, applets are currently modified within an HTTP proxy server that sits between a web server and a client browser. This implementation does not require any changes in the web server, Java Virtual Machine or web browser. Since applets and the browser are not notified of changes in the applet, subsequent requests for safeguarded executable entities may be issued to the web server, which does not have them. This problem is handled by having the proxy redirect these requests to cites where the safeguarding entities are actually stored.

The following sections explain how modified executable entities are inserted in Java bytecode. The modifications may be divided into two general forms, class-level and method-level modifications.

3.1 Class-level Modification

A class such as `Window` can be replaced with a subclass of `Window` (which will be called `Safe$Window` in this example) that restricts resource usage and functionality. For example, `Safe$Window`'s constructor method can put a limit on how many windows can be open on the screen. The method allows new windows to be created until the number of windows exceeds the limit. If the limit is exceeded, the method throws an exception indicating that too many windows are open. Since `Safe$Window` is a subtype of `Window`, type `Safe$Window` can appear anywhere type `Window` is expected. Hence, the applet should not notice the change, unless it attempts to create windows exceeding the limit.

This example of class-level substitution is done by merely substituting references to class `Window` with references to class `Safe$Window`. In Java, all references to strings, classes, fields, and methods are resolved through indices into the constant pool of the class file [6],

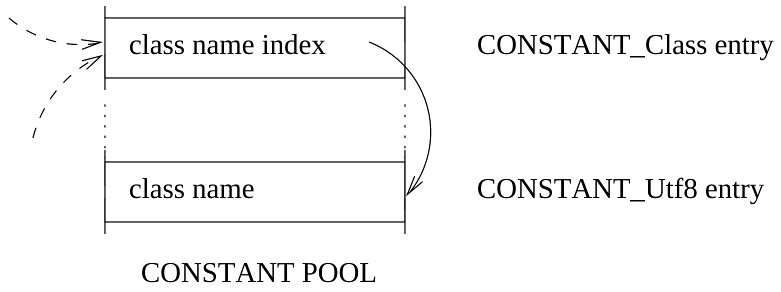


Figure 1: A class is represented with two entries in the constant pool

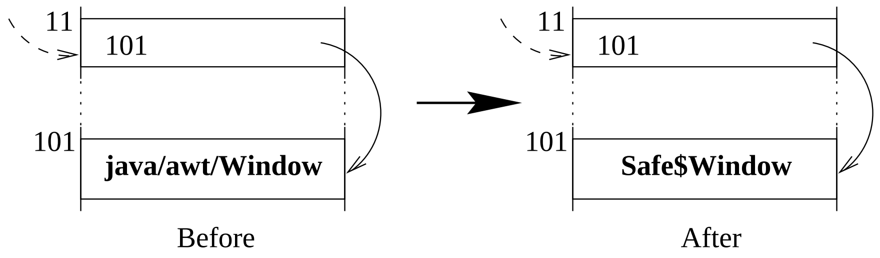


Figure 2: Class-level modification substitutes class reference

where their symbolic names are stored. Therefore, it is the constant pool that should be modified in a Java class file. In more detail, two entries are used to represent a class in the constant pool. A class is represented by a constant pool entry tagged as CONSTANT_Class which refers to a CONSTANT_Uft8 entry for a UTF-8 [1] string representing a fully qualified name of the class, as shown 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.

Class-level substitution requires a simple modification of a constant pool entry, since it takes advantage of the property of class inheritance. Obviously, however, the use of class inheritance prevents this approach from being applied to final classes or interfaces.

[1] The Unicode Standard, version 1.1, and ISO/IEC 10646-1:1993 jointly define a 16 bit character set which encompasses most of the world's writing system. UTF-8, one of UCS transformation formats, has been developed for the compatibility between the 16-bit characters and many applications and protocols for the US-ASCII characters. For more information regarding the UTF-8 format, see *File System Safe UCS Transformation Format (UTF-8)*, X/Open Preliminary Specification, X/Open Company Ltd., Document Number: P316.

3.2 Method-level Modification

To address the limitation of class-level modification, method-level modification replaces a method with a related method without making use of the class hierarchy structure. This approach provides more flexibility in that it can be used even when the method is final or is accessed through an interface, but requires more complicated modifications of method reference and method invoking instructions.

Before getting into more details, we show a brief description of a field and a method descriptor in 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 descriptor

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. A return descriptor represents 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 foo (Thread t, int i)` is `(Ljava/lang/Thread;I)V`.

Before explaining how method invoking instructions are modified, we also show how a method is compiled into a class file through the following example, which gives you the intuituion about what bytecodes look like.

The method

```
void foo (Thread t, int i) {
     t.setPriority (i);
}
```

compiles to

```
Method public foo(Ljava/lang/Thread;I)V
       push Ljava/lang/Thread;I)v
       push I
       invokevirtual Thread.setPriority(I)V
```

We are going to explain method-level modification with this example, trying to replace `Thread.setPriority(I)V` with a more restrictive method, for instance, called `Safe$-Thread.setPriority(Ljava/lang/Thread;I)V`, which does not allow an applet to have higher priority than a new upper limit defined in class `Safe$Thread`. Since the new safeguarding method invokes the instance method of class `Thread`, a reference to an instance of class `Thread` should be passed to the new method. For instance, `t.setPriority(5)` becomes `Safe$Thread.setPriority(t,5)`. The new method takes priority of type integer as one of its 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.

3.2.1 Method Reference Modification

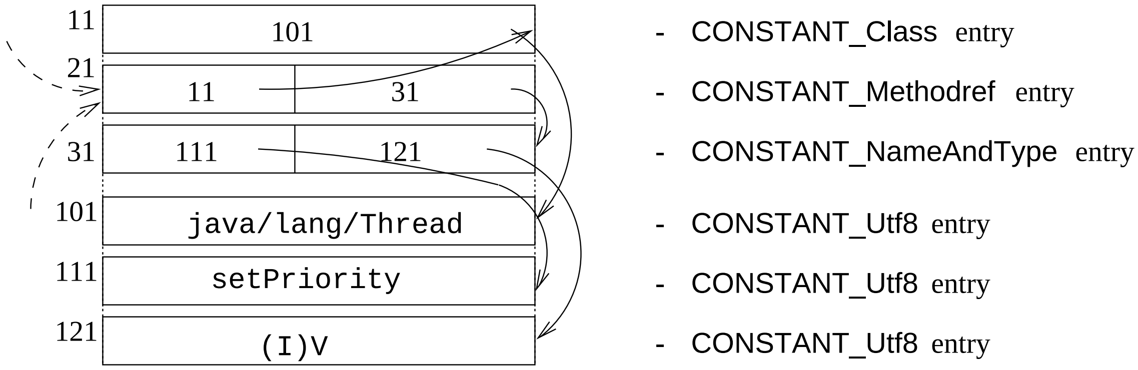
A method of a class (a static method) or of a class instance (an instance method) is represented by a constant pool entry tagged as CONSTANT_Methodref. The CONSTANT_Methodref entry refers to the CONSTANT_Class entry, representing the class of which the method is a member, and the CONSTANT_NameAndType entry, representing the name and descriptor of the method, as shown in Figure 3(a). In our example, the CONSTANT_Class entry and the CONSTANT_NameAndType entry refer to the CONSTANT_Uft8 entries representing `java/lang/Thread`, `setPriority`, and `(I)V`, respectively.

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

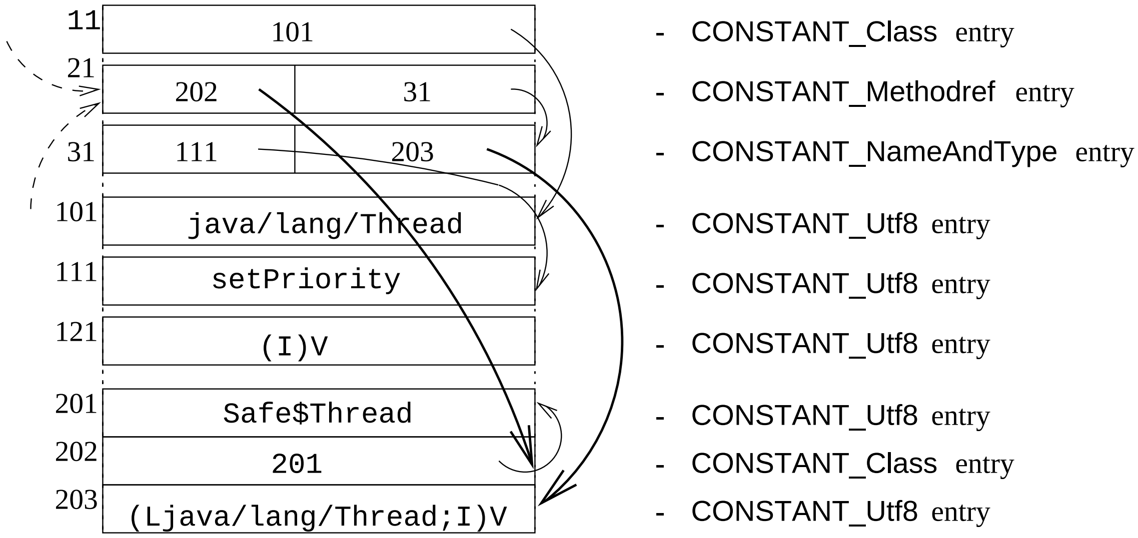
3.2.2 Method Invoking Instruction Modification

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 in this example. Both instructions take as an argument an index to a CONSTANT_Methodref constant pool entry, but their operand stacks are different.

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. Figure 4(a) shows the operand stack and instruction sequences for the



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



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

Figure 3: Method-level modification substitutes method reference

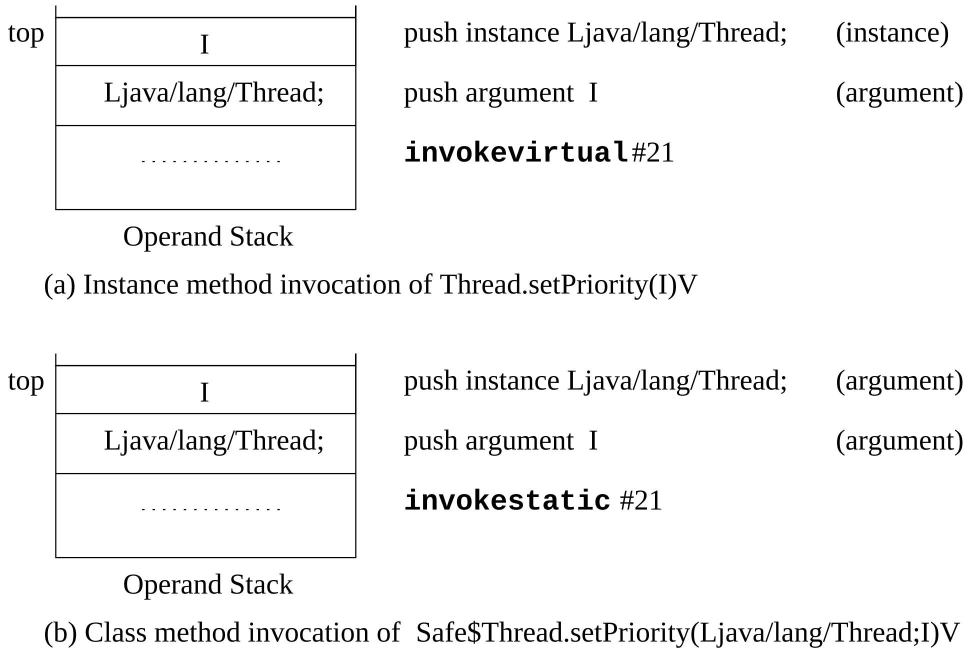


Figure 4: Operand stack and instruction sequences for method invoking instructions

instance method call to `Thread.setPriority(I)V`. The argument of `invokevirtual` is the index in Figure 3(a).

The class method invocation requires only arguments to be pushed onto the operand stack. The operand stack and instruction sequences for the instance method call to `Safe$-Thread.setPriority(Ljava/lang/Thread;I)V` are shown in Figure 4(b). The argument of `invokestatic` is the index in Figure 3(b).

While the operand stacks and `push` instructions and their arguments in Figure 4(a) and (b) are the same, the instruction for method invocation is different. Hence, the new method `Safe$Thread.setPriority(Ljava/lang/Thread;I)V` can be added into the bytecode program with a change from `invokevirtual` to `invokestatic`.

In this section, we covered the details of how the two bytecode modification techniques work. While class-level modification requires a simple modification in the constant pool, method-level modification requires bytecode instruction modifications as well as constant pool modifications. Essentially, class-level modification requires only 5-35% of computation of method-level modification depending on the relative size of the constant pool. However, class-level modification can not be applied to final classes or interfaces which method-level modification may be applied to.

4 Examples for Applet Security

In this section, we outline several examples of using bytecode modification technique for protecting against malicious attacks mentioned in Section 2.

4.1 Window Consuming Attack

An applet can crash the system by creating more windows than the windowing system can handle. To protect against this resource consuming attack, the safety mechanism should keep track of window creation.

A Java library class, `Frame`, handles an optionally resizable top-level window. The constructor methods create a window. The key to the solution to this attack is to disallow an applet to invoke the constructor methods more than a certain number of times.

Since `Frame` is not final, a subclass `Safe$Frame` can be generated, in such a way that `Safe$Frame` can monitor and control every window generation. `Safe$Frame` can create windows using the constructor methods of `Frame` while counting the current number of open windows. It should not create a window if the number of windows exceeds its own limit. Class-level modification is used to substitute references to `Frame` with references to `Safe$Frame`. This technique may also restrict window size and window positions.

4.2 Email Forging Attack

An applet is able to disclose the user's confidential information through email, while its web server is running an SMTP mail daemon. To prevent access to 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 be familiar with how constructor method invocation is implemented in the Java Virtual Machine(JVM).

JVM 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

```
Method java.net.Socket create()
    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
```

`invokespecial` is the Java Virtual Machine instruction for instance initialization method invocations. It invokes instance methods requiring special handling, such as superclass, private, or instance initialization methods.

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

Since `Safe$Socket.init` is a static method, we 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 codes are as follows:

```
Method java.net.Socket create()
    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
```

4.3 URL Spoofing Attack

An applet can spoof a user with a fake URL display on the status line. This spoofing attack is protected by checking the consistency between the URL displayed and the URL from which a Web page is actually to be loaded.

A Java library interface, `AppletContext`, defines the methods that allow an applet to interact with the context in a Web browser or an applet viewer. The `showDocument` method requests that the browser or applet viewer show the Web page indicated by the URL argument. The `showStatus` method displays text in the Web browser or applet viewer's status line. `Safe$AppletContext.showStatus`, which is our static safe method for the `showStatus` method, saves the current text in addition to displaying it so that another our safe method, `Safe$AppletContext.showDocument`, can refer to the text later. When `Safe$AppletContext.showDocument` is invoked, it first examines whether or not the URL argument is equals to the text which is currently displayed on the status line. If so, the method requests the browser to bring the Web page indicated by the URL argument. If not, the method displays the URL argument on the status line, instead of passing on the request. In the latter case, the users may notice the inconsistency, and take an appropriate action. In general, the users can get the Web page loaded with one more mouse click. This guarantees that the users bring a new Web page with its URL displayed on the status line. Its positive side effect is that it displays the URL when the URL is not available.

Since the `AppletContext` interface is not inheritable, the two interface methods must be replaced through the method-level modification. References to `AppletContext.showStatus(S)V` and `AppletContext.showDocument(Ljava/net/URL;)V` is substituted with references to `Safe$AppletContext.showStatus(S)V` and `Safe$AppletContext.showDocument(Ljava/net/URL;)V` respectively.

`invokeinterface` is the instruction for invoking an interface method. Since the interface methods are substituted with the static methods, `invokeinterface` is also replaced with `invokestatic`. Since `invokestatic` does not have the last two operands of `invokeinterface`, the two operands should be assigned to the `nop` instruction.

4.4 Annoying Sound Attack

An applet can annoy the user with never-ending sounds. To prevent this annoyance attack, the user should be allowed to turn sounds off. The solution is to keep track of every sound object.

A Java library interface, `AudioClip`, describes the essential methods for playing a sound. `AppletContext.getAudioClip()` and `Applet.getAudioClip()` both return an object that implements this interface. The `loop` method of the object starts playing the audio clip in a loop, and the `stop` method stops playing the audio clip. The attack is implemented by looping the sound, but never stopping it.

Whenever an applet invokes the `loop` method of an object, the safety mechanism opens a window in which the users can turn off the sound, and keeps the object in order to invoke the `stop` method of the object when the users want to turn it off. Figure 5 shows a control window over a sound object.

Since the `AudioClip` interface is not inheritable, the safety mechanism uses the method-level modification. There are two methods to be replaced. References to `AudioClip.loop()V` and `AudioClip.stop()V` are substituted with references to `Safe$AudioClip.loop()V` and

Figure 5: A pop-up window for controlling a sound object

`Safe$AudioClip.stop()V` respectively. As above, `invokeinterface` is replaced with `invokestatic`, with the two extra arguments replaced by `nop`'s.

We can extend this idea of bringing up a user interface to perform other forms of monitoring. For example, we could use our techniques to watch and control some internal variables in a variety of Java objects. For example, users might have a window that lists all threads and locks, and be allowed to kill threads. Or the window might be able to list all windows that have been created, and how big they are. Users could even be allowed to change (public) variables and call methods, making this a way to debug or experiment with applets.

5 Performance Results

Our safety mechanism imposes the extra overhead of inserting safeguarding code into applets and executing the additional safeguarding code. To evaluate the performance of our safety mechanism, we implemented and measured a prototype of our system which consists of safeguarding classes, an HTTP client and an HTTP proxy server.

The HTTP client is a Java program which sends a request to a web server, receives its reply from the server, and measures the time it takes to receive the reply. Our HTTP proxy server, written in Python, performs forwarding of messages between client and web server, as well as transformation of applets. Our HTTP client was running on a Sun Ultra 1 Model 170 which has one 170MHZ Ultrasparc processor, and our proxy server a Sun Ultra Enterprise 3000 which has two 248MHZ Ultrasparc processors. Both machines are on 10Mbit/s Ethernet links. We ran each test 1500 times to measure the performance of our safety mechanism.

5.1 Encapsulation Overhead

We evaluated the overhead of encapsulating Java classes in terms of loading time. We treated each Java class loaded from the network as untrusted, and encapsulated all of its bytecode. The proxy server performs the encapsulation while the classes are being loaded into the web browser. The encapsulation overhead increases the time it takes for the classes to be transferred to the browser. Table 2 shows the time($T_{encapsulation}$) it takes to encapsulate the classes. It shows that $T_{encapsulation}$ is linearly proportional to the size of the Java class.

Class size	Encapsulating Time (sec)	Loading time w/o encapsulation	Loading time w/ encapsulation	Overhead
1K	.024	.225	.249	10%
10K	.231	.226	.457	102%
20K	.438	.244	.682	180%
30K	.642	.261	.903	246%
40K	.874	.285	1.159	307%
50K	1.350	.316	1.666	427%
100K	2.652	.457	3.109	580%
150K	4.686	.570	5.256	822%
200K	6.336	.658	6.994	962%
250K	7.963	.797	8.760	999%

Table 2: Encapsulation Overhead

Since it is hard to measure the loading time using the browser, we use our HTTP client instead. The loading time is defined as the time to transfer the request to the web browser($T_{request}$), plus the time for the server to process the request(T_{server}), plus the time it takes the browser to receive the reply(T_{reply}). It does not include the time it takes for the browser to verify applets before displaying them. Table 2 also shows the loading time when the encapsulation is not applied. It increases by the proportion of the network speed.

Now, let us consider how $T_{encapsulation}$ affects the loading time. We define the encapsulation overhead is: $\frac{T_{encapsulation}}{T_{encapsulation}+T_{request}+T_{server}+T_{reply}}$. Table 2 shows the overhead. We tested class files up to 250K, a reasonable upper limit.[2] The measured cost of encapsulating an applet is substantial. Though $T_{encapsulation}$ and the loading time increase linearly, the overhead also increases linearly. While analyzing the overhead, we realized that our overhead mainly results from the proxy being written in Python. Python is an interpreted language which is easy to work with, but can be 100 times slower than C code for this kind of program, where individual bytes in the bytecode are being examined and modified, as

[2]The current Java compiler `javac` turned out to be unable to compile Java source files whose bytecode size is bigger than 260-270 Kbytes. It ends up with `java.lang.OutOfMemoryError`.

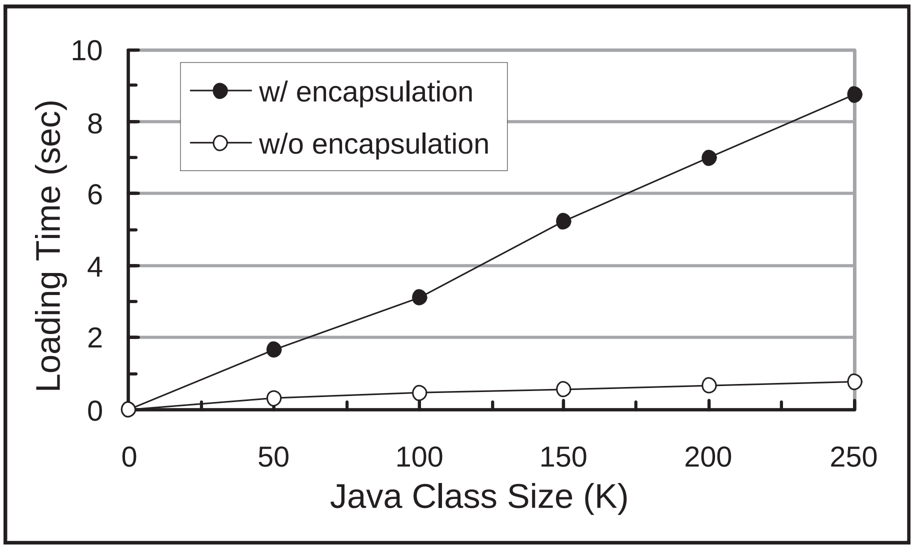


Figure 6: Encapsulation overhead

shown in the Appendix. It the proxy server were to be rewritten in C, the encapsulating overhead would be minor.

5.2 Safeguarding Code Execution Overhead

We evaluated the cost of running safeguarding code in terms of executing time. We implemented four kinds of safeguarding classes which are explained in Section 4 and measured the execution overheads respectively. The safeguarding code was running on a Sun Ultra 1 Model 170 with Netscape $Navigator^{TM}$ Gold 3.01. Table 3 shows the overhead of each safeguarding code.

Safeguarding code	Overhead
`Safe$Frame` for Window Attacks	4%
`Safe$Socket` for Network Accesses	4%
`Safe$AppletContext` for URL Spoofing	5%
`Safe$AppletContext` for Sound Object Control	55%

Table 3: Execution Overhead

The safeguarding code falls into two general categories. The first type performs additional security checks and raise an exception if the check fails. Safeguarding codes for window attacks, network accesses, and URL spoofing are included in this group. As shown in the table, the security checks against those attacks can be done with a 4% overhead.

The other types keeps track of an object and provides control over it. This protects against both faulty programming and malicious attacks that loses control of the resource. Our experimental data show that such a problem may be handled with a half execution time overhead.

6 Related Work

There are three general approaches which have been proposed for the safe execution of mobile code. Lucco, *et al.*, introduced *software fault isolation* [13] 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. Java uses a simple *sandbox* security model for executing untrusted applets in a restricted execution environment. This sandbox model was supposed to prohibit untrusted applets from using any sensitive system services, but failed to do even with small implementation errors [3]. Malkhi, *et al.*, proposed a concept of *playground*(sandbox) [8] for executing untrusted mobile code on a remote protected domain(machine), called playground. Prior to execution the applet is transformed to use the downloading user's web browser as a graphics terminal for its input and output. The way in which the applet is transformed is *class-level modification* explained in Section 3.1. They just substituted the names of AWT classes to the names of the representative stubs of the corresponding remote AWT classes. As long as the AWT classes are all inheritable and have no final method, the class-level modification is good for this approach. Our approach is related to software fault isolation. We encapsulate applets through *bytecode modification*, in order to perform more security and sanity checks and provide controls over objects which happen to be inaccessible.

Language semantics can be used to enforce safety by guaranteeing that a program can't affect resources that it can't name [2]. 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* [11], where the mobile code carries a proof that it complies with certain invariants or requirements. This can be treated as an effort to provide a formal method to specify and check the extended semantics. Research is underway to provide the formal method.

Another approach for securing hosts from mobile code is to import and run only trusted mobile code from the network. For example, digital signature mechanism enables a user to download applets written by only trusted authors. Princeton research group proposed a *Java Filter* [1] for preventing untrusted applets from entering the user's computer. A user can download Java applets only from trusted servers using the Java Filter. For the browser to employ the Java filter, they made changes to the browser's class library which is the class file of the `AppletClassLoader`. Firewalls can be used to filter out all outside applets [9], while allowing trusted internal applets to run. A few techniques are considered to try to block Java applets at the firewall. One idea is to look for `<applet>` tags in the downloaded stream and delete or replace such tags. The firewall should scan almost all the different mechanisms(HTML, FTP, gopher, mail, news) which can be used to deliver applets by encapsulating them properly. This technique imposes a great deal of traffic loads on the firewall. Also Javascript can be used to build `<applet>` tags on the fly. Although there is no such tags in the HTML document, the browser's executing of Javascript will cause them to be inserted at the time the document is viewed. Another idea is to detect Java class files at the firewall by a magic byte sequence that is required at the beginning of every class file or by their name which will end in `.class`. However, this technique can not detect class files which are passed through an encrypted (SSL) connection, which will make them indistinguishable from ordinary files to the firewall, or are a part of compressed archive(Jar or Zip). This idea is also used at the proxy for bytecode modification, so our proxy suffers from the same limitations. However, if bytecode modification were incorporated into the browser or virtual machines, these limitations would not apply.

7 Conclusion

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

The encapsulating overhead shown in Section 5 seems considerable, but it results primarily from the Python proxy server. As an interpreted language, Python can be 100 times slower than C code where individual bytes in the bytecode are being examined and modified. If the proxy server were to be rewritten in C, the encapsulating overhead would be minor. Other experimental performance results show that encapsulated code for additional security checks is executed with a 5% slowdown, and code for controling inaccessible objects with a 55% slowdown.

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. In the future, we plan to explore ways to utilize the technique in other settings, such as interacting with normally inaccessible objects.

References

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

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

[3] 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.

[4] 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.

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

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

[7] Gary McGraw and Edward W. Felten. Java Security: Hostile Applets, Holes, and Antidotes. John Wiley & Sons, 1997.

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

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.

Appendix

Comparison between the Performance of C and Python

We implemented a bubble sort algorithm in C and Python to compare their performance. Table 4 and Figure 7 show the CPU time in sorting hundreds of variable-length words using

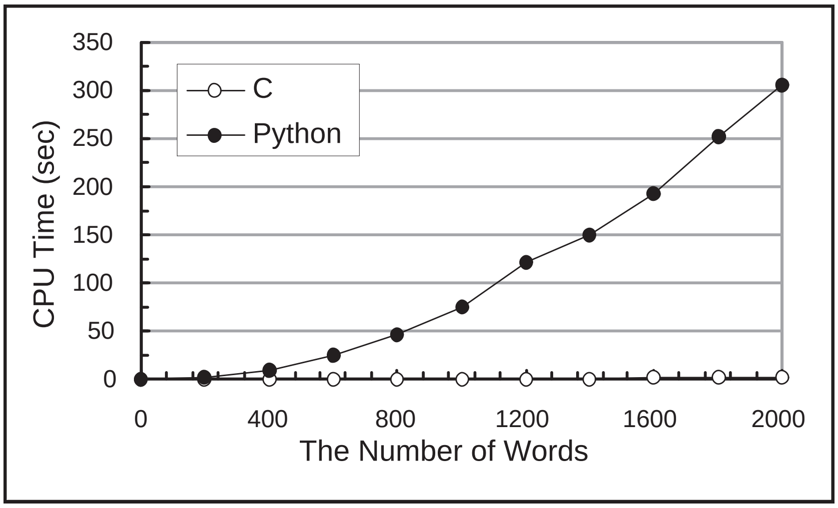


Figure 7: A program in C is much faster than in Python.

C and Python programs respectively. In this example, we may say that a program in C is 100 times faster than in Python.

Word Number	CPU time in C	CPU time in Python
200	.02	2.10
400	.08	9.79
600	.16	24.35
800	.25	45.92
1000	.41	74.22
1200	.61	120.77
1400	.82	150.73
1600	.98	192.97
1800	1.26	252.34
2000	1.60	305.40

Table 4: A program in C is much faster than in Python.

With the data in Table 4, we can estimate the encapsulating time when code for encapsulating is written in C. Based on this comparison, Figure 8 shows the estimated encapsulating overhead.

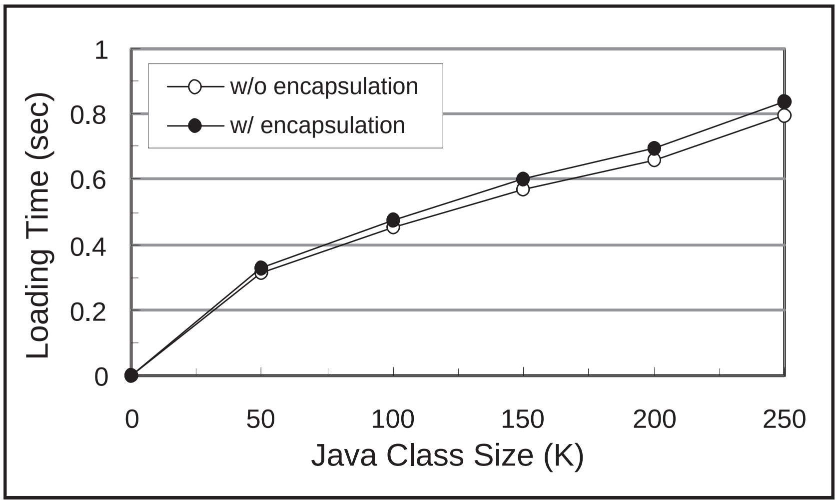


Figure 8: If the proxy were to be rewritten in C, the encapsulating overhead would be minor.