Polytechnic
UNIVERSITY
Brooklyn · Long Island · Westchester

DISSECT: DIStribution for SECurity Tool

Enriquillo Valdez Moti Yung

Department of Computer and Information Science

Technical Report
TR-CIS-2000-01
03/15/2000

DISSECT: DIStribution for SECurity Tool

Enriquillo Valdez [*]
Polytechnic University
Computer and Information Science Department
Brooklyn, NY, USA
rvaldez@photon.poly.edu

Moti Yung
CertCo Inc.
New York, NY, USA
moti@cs.columbia.edu

TR-CIS-2000-01

Abstract

A security threat that affects the Java environment is the reverse-engineering and code-understanding of the architecture-neutral bytecode format. In this paper, we present a novel decomposition strategy that protects the binary source of Java class files. Our strategy, which has been automated, decomposes "programmer selected" classes of a Java application into server classes and client classes. Server classes contain the actual class code and run only on trusted systems (which we call servers but can be other dedicated machines). Assumed to perform most of the task (but the sensitive part), client classes, on the other hand, execute on user systems and must interact with their corresponding server class in order to execute the sensitive code and provide the behavior of the original class. We implemented DISSECT, an architecture based on the decomposition strategy, for Java 1.1. Our protection architecture consists of an automated tool that generates decomposed classes and the supporting infrastructure for instantiating and executing classes remotely.

We conducted initial experiments to understand the impact of decomposed classes on performance, since the remote execution of classes increases the overhead and one has to understand the granularity and modularization of decomposition. We report initial performance results which show the overhead and demonstrate when it does not exist, when it is low and when it is high.

1 Introduction

Java has emerged as the technology of choice for harnessing the nascent network computing environment. In such an environment, applications need to be light-weight, flexible, and extendable. Applications download their software components as needed from the network at runtime. With built-in networking support, dynamic loading capability, and platform independent code, Java is suitable for such an environment. A Java word processor application, for example, would provide basic functionality, but when it needs to convert a document to a format that it does not know, it would automatically download the formatting component from the network to accomplish its task. However, supporting dynamic loading requires a significant amount of symbolic information (e.g., method names, signatures, access specifiers, etc.) to be maintained in the Java class files. To provide platform independent code, Java has configured its runtime system (i.e., Java Virtual Machine (JVM)) to operate in an environment that represents the lowest common denominator in all hardware architectures, a stack machine. The Java

[*]This work was supported by the National Science Foundation under Grant No. 9256688 and the NY State Center for Advanced Technology in Telecommunications.

stack machine operates in a straightforward manner and has a very simple bytecode (i.e., instruction) format. Unfortunately, these characteristics make Java code very easy to decompile [1, 20, 21].

The relative ease of decompiling Java code makes software developers apprehensive about producing their software products in Java. Software developers fear that their private knowledge components (e.g., trade secrets, proprietary algorithms, etc.) will be reverse-engineered. The reverse-engineering of Java code can be exploited by competitors to facilitate the construction of competing products and by malicious users who want to learn how to tamper with Java programs.

In this paper, we present the design and implementation of a network oriented protection architecture to address the reverse-engineering of Java applications. The basic idea behind this architecture is to remove valuable components from the application code delivered to users and to provide access to these distributed components via network resources. We call the methodology and the tool designed based on it DISSECT: DIStribution for SECurity Tool.

In our methodology and the tool implementing it, selected classes of a Java application are decomposed into client (local) and server (remote) classes. Consisting mostly of remote method calls, a client class encapsulates the visible behavior of the original class from a user's perspective (e.g., provides access to public methods and variables). The server class contains the original class code extended to support requests from the client class. A client object (i.e., an instantiation of a client class) must interact with its corresponding server object in order to provide the behavior of the original object. The client classes are distributed to and executed by users, and the server classes are delivered to trusted systems. Our architecture relies on Remote Method Invocation (RMI) to enable communication between client and server applications.

Our approach is applicable to situations where software developers want to prevent the disclosure of embedded secrets in programs or inner workings of programs (software protection). Suppose a program encrypts some data using an embedded cryptographic key. Applying our approach, the decomposed program maintains the cryptographic key on trusted servers and not embedded in the user's program (this is, by the way, in line with common practice regarding cryptographic co-processors). Thus, to get the encrypted data, a user interacts with a server that receives the user data and returns the encrypted data. Another example is a proprietary algorithm that solves an optimization problem which a developer does not want to reveal to users.

We note that selectivity has to be applied when adopting DISSECT methodology, since it may not be suitable for all types of Java applications. Decomposed applications will incur a communication penalty due to invoking methods on remote systems. For programs that already access an external resource, such as a database, this penalty does not exist and we may want to decompose according to such external accesses. In general, we try to compensate for added overhead communication by executing server objects on systems that offer advantages in terms of computing resources, such as a high-performance processor or special cryptographic hardware. Selecting classes for decomposition will require developers' discretion and foresight. Developers must structure their code with performance concerns in mind. An example is avoiding unnecessary method invocations or avoiding excessive method invocations between client and server objects. In the ideal situation, a decomposed application should spend most of its time executing either on the client or the server site and not waiting for remote method returns.

The paper is organized as follows: Section 2 reviews related work. Section 3 describes the decomposition methodology that we apply on Java classes. Section 4 presents the design and implementation of a Java protection architecture based on the decomposition strategy. Section 5 discusses the achieved security and performance of our protection architecture. The conclusions can be found in Section 6.

2 Related Work

In our work, we design a decomposition strategy that utilizes network resources to prevent the reverse-engineering of Java code.

In terms of preprocessing Java code to execute on remote systems, a related work appears in [12] which addresses malicious code threat for applets (the dual problem to ours!). Here, instead of running within a user's browser, a downloaded applet is preprocessed to execute in a sanitized environment where it cannot harm the user's system. The processed applet has its input/output (I/O) redirected to the web browser (that downloaded it).

Our work differs from [12] in the following aspects: (i) in our work users never get the actual bytecode of a protected class, but only a skeleton of the class. The processing of the Java code is done by the software developer instead of the user as in [12]. (ii) Our approach is fine grain where an application is decomposed on the class level to execute on remote systems. Thus, we can use our approach to execute the classes of an application on different remote systems. In [12], a coarse grain approach is presented where the entire Java program (i.e., the applet) executes on a remote system having its I/O redirected.

Other technical approaches to address reverse-engineering have included obfuscating Java classes, computing with encrypted functions, and executing objects inside a tamper proof environment.

Obfuscation techniques [5, 6, 10] have focused on manipulating the entries in the constant pool, the data structure in the Java class binary that maintains symbolic and type information. These techniques have added extraneous symbols, removed unnecessary symbols, or replaced symbols with confusing or illegal strings (e.g., strings which are not valid in the Java language) in the constant pool. Collberg et al. [3, 4] describe obfuscation techniques based on compiler optimization techniques. In their work, they introduce the notion of opaque constructs which are variables or predicates whose values (or properties) are difficult for deobfuscators to determine. These opaque constructs are used to prevent the removal of extraneous code that they add to Java programs. Their most resilient (i.e., resistant to reverse-engineering) opaque constructs use pointer-based structures and exploit the difficulty of pointer analysis. Their work lacks experimental results on how well opaque constructs (e.g., predicates) fare against known pointer analysis techniques. Obfuscation only serves to make it difficult to reverse-engineer. Another approach based on hiding programs uses cryptographic primitives. Sander and Tschudin [14] employ encrypted functions to hide function evaluations by mobile agents. Unfortunately, their theoretically appealing method applies only to very specific polynomial functions. Another approach links execution of classes with tamper resistance hardware. Wilhelm [18] presents an environment for executing objects that prevents disclosure of their inner working (and tampering). The classes are delivered to users encrypted; the classes are then decrypted and executed only within a tamper proof device. The tamper proof device contains the necessary secrets (e.g., decrypting keys) for revealing the encrypted classes. This approach requires special hardware installed on user systems and is not portable. In addition, developers must trust manufacturers to produce effective tamper proof devices and ensure periodically that these devices have not been tampered with (i.e., their secrets have not been broken).

3 Decomposition Methodology

The basic idea behind our decomposition methodology is to remove (valuable) code from the application's classes distributed to users. The following section discusses the details of the methodology.

3.1 Methodology

Before we discuss our decomposition strategy, we present some background information on Java. A Java application consists of a collection of classes. A class defines a set of elements consisting of variables and methods (i.e., functions). An interface defines a set of abstract methods. An object is an instantiation of a class. The type of an object is determined by the classes that it is an instance of, and by the interfaces implemented in these classes. The "behavior" of a class consists of all the declared variables and implemented methods.

The decomposition strategy works by extracting and encapsulating the behavior of a class which is visible to the user. We consider the visible behavior of a class to consist of all the (implemented) instance variables and methods that are accessible globally or within the package name space. These are the class

class first requests the server object and then invokes an initializing method. The client class implements an initializing method for each constructor method with arguments in the original class. Only the top-level class of the hierarchy makes the request for a server class instantiation on the remote system. The top-level class is the one that extends (i.e., is a subclass of) java.lang.Object.

We employ the Java's RMI mechanism to enable remote invocation of server objects' methods. Unfortunately, we need to provide some low level detail on the Java language interface to the RMI mechanism to enable understanding of features of decomposed classes. In Java, a remote object is an object whose methods can be invoked from objects running on different Java virtual machines. To use RMI, we need to define a remote interface (i.e., one that extends java.rmi.Remote directly or indirectly). This interface declares all the (remotely available) methods. Every declared method in the interface throws at least the exception java.rmi.RemoteException. This is because the network may fail, the remote system may crash, etc. Thus, an object that invokes a remote method will need to handle this type of exception. A remote object class implements the methods of the remote interface and must also extend the java.rmi.server.RemoteServer class. However, RemoteServer is an abstract class, and only the RMI server class provided with the Java Development Kit (JDK) is the java.rmi.server.UnicastRemoteObject class. The UnicastRemoteObject class provides the basic underlying support for a single server object (i.e., handles connection, etc.). Other classes derived from a remote class are stub and skeleton classes[3]. The stub class represents the remote object on the local system; the stub implements the same method as the remote interface. The stub object is responsible for marshaling and sending the arguments of a remote method invocation to the skeleton object. When a skeleton object receives a message, it unmarshals the arguments and calls the method of the remote object. If there is any return parameter, the skeleton object marshals the argument and returns it to the caller. When a client object has a reference[4] to a remote object, it invokes the methods of the stub object. Note that a client object needs access to the remote interface and stub classes to be able to make a remote method invocation.

RMI uses the Java Object Serialization mechanism to pass objects by value to (and from) another virtual machine. Thus, remote objects, such as our server classes, require that the method's arguments and return type be of primitive types[5], serializable objects (i.e., implement the java.io.Serializable interface), or remote references. Remote objects are not passed by value; RMI passes remote references (i.e., stub objects are passed) instead of the actual remote objects.

To illustrate the decomposition strategy, we show the outputs on a simple Java class, Circle.java, given in Figure 1-(a). When decomposition is applied on a class, three classes are generated: a shared interface class, a client class, and a server class. The shared interface class (see Figure 1-(b)) is a remote interface and declares methods that correspond to the original instance methods (with public or package access) of the selected class. In Circle.java, the only method with such access is cirCumXArea(). The shared interface also provides methods for setting and retrieving instance variables with public and package access. The setValuePrimDouble() and getValuePrimDouble() are such methods. These methods only make accessible variables x, y, and z; see Figure 3 where these methods are implemented. In addition, the share interface declares methods for initializing the state of the server object, for example, the DDD() method is used for passing the arguments associated with the Circle class constructor method that has three arguments.

In the decomposition strategy, both the client and server classes implement the methods declared in the shared interface. For these methods, the client class implements proxy methods which invoke their

[3]The stub and skeleton classes are generated automatically by invoking the rmic program on the server class. The rmic program is distributed with the JDK.

[4]Section 4.3 discusses how we obtain such references in our protection architecture.

[5]There are eight primitive types: byte, short, int, long, float, double, char, and boolean.

```
(a)

  package Shape;

  public class Circle {

   public double x,y,r;
   private double Pi = 3.14159;

   public Circle()
   { x = 0.0; y = 0.0; r =0.0; }

   public Circle(double x, double y, double r)
   { this.x = x; this.y = y; this.r = r; }

   protected double circumference()
   { return 2 * Pi * r; }

   private double area()
   {  return  3.14159 * r * r; }

   public double cirCumXArea()
   {   return( circumference() *  area());  }

  }
```

```
(b)

package Shape;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Circle_face implements Remote
{
   public abstract void
     DDD(double d1, double d2, double d3)
       throws RemoteException;
   public abstract double
     cirCumXArea()
       throws RemoteException;
   public abstract void
     setValPrimDouble(double d, String string)
       throws RemoteException;
   public abstract double
     getValPrimDouble(String string)
       throws RemoteException;
}
```

Figure 1: (a) Shows a simple Java class and (b) shows the generated interface class

reciprocal methods on the server object. The proxy methods contain exception handling code (i.e., the code to handle RMI exceptions from invoking server methods). If the original selected class is a top-level class of the class hierarchy, then its corresponding client class contains a special constructor method. This special constructor is responsible for remotely requesting the instantiation of the server class on the remote system. If it is not the top-level class, the client class delegates the server class instantiation to its super class. Figure 2 shows the client class, Circle_client.java, which is a top-level class. An interesting point to observe is that only the first constructor method instantiates the server object (on the remote system). The second constructor method directly calls this initializing method. In the Circle_circle, the second constructor method invokes the initializing method before calling the DDD() method [6] on the server object. The server class implements the method of the shared interface. It has the original class's methods and variables. The server class extends the UnicastRemoteObject class. Figures 1-(b), 2, and 3 show actual classes generated by our tool that were subsequently decompiled by Mocha [20].

4 DISSECT: Architecture & Implementation

Running applications with decomposed classes requires a supporting infrastructure. We design and implement, DISSECT, a protection architecture that addresses the implications of decomposed classes for applications. We call an application with classes decomposed a client application and its corresponding set of server classes a server application. The protection architecture consists of a processing stage and an initialization stage. In the initialization stage, client objects link to their respective server objects (at runtime). The protection architecture also requires that certain classes of the Java API to be customized to function as remote objects.

From this point on, we call the system where a client application resides a home system and remote systems where server classes are instantiated host systems. We assume that both home and host systems

[6]Note that DDD() is null on the Circle_client class. It serves to make the client class consistent with the notion that it has implemented all the method of the interface (e.g., Circle_face).

```
package Shape;

import java.rmi.Naming;
import java.rmi.Remote;
import java.util.Random;
import java.util.Vector;
import server.HostLoaderX;

  public synchronized class Circle_client implements Circle_face
  {
   protected HostLoaderX AppRemoteServ;
   protected Remote Decomp_genfaceRemoteObj;
   protected api.lang.SystemXface AppSysObj;
   private Circle_face ShapeCircle_hostRemoteObj;

   // First constructor method establishes link with server object
   public Circle_client(String string1, String string2,
   Vector vector1, Vector vector2, Vector vector3)
   {
     ...
          string3 = (String)vector1.elementAt(0);
          AppRemoteServ =
        (HostLoaderX)Naming.lookup(
             (new StringBuffer
              (String.v( alueOf(string3)).append("/Loader").toString());
          Decomp_genfaceRemoteObj =
            AppRemoteServ.LoadByName
              (string1, string2,( vector2, vector3);
          ShapeCircle_hostRemoteObj =
            (Circle_face)Decomp_genfaceRemoteObj;
     ...
       return;
   }

   // Second constructor method
   public Circle_client
     (double d1, double d2, double d3,
       String string1, String string2,
       Vector vector1, Vector vector2, Vector vector3)
   {
       this(string1, string2, vector1, vector2, vector3);
     ...
   // Calls initializing method on server object
          ShapeCircle_hostRemoteObj.DDD(d1, d2, d3);
     ...
       return;
   }

  // A null method, not needed on client side
   public void DDD(double d1, double d2, double d3) {}

  // The proxy for invoking cirCumXArea on the server object
  public double cirCumXArea()
   {
     ...
          double d = ShapeCircle_hostRemoteObj.cirCumXArea();
          return d;
     ...
   }

   // Proxy methods for getting and setting variables
   public double getValPrimDouble(String string)
   {
     ...
          double d = ShapeCircle_hostRemoteObj.getValPrimDouble(string);
          return d;
     ...
   }

   public void setValPrimDouble(double d, String string)
   {

     ...
          ShapeCircle_hostRemoteObj.setValPrimDouble(d, string);
     ...
       return;
   }
  }
```

Figure 2: Shows parts of the client class (i.e., we omit exception handling code) generated from the class in Figure 1-(a)

```
package Shape;

import java.rmi.server.UnicastRemoteObject;
import java.util.Vector;

public synchronized class Circle_server
   extends UnicastRemoteObject implements Circle_face
   {
   public double x;
   public double y;
   public double r;
   private double Pi;

   // called by the ApplicationClassLoader
   public Circle_server(Vector vector1, Vector vector2)
   {
       Pi = 3.141590118408203;
       x = 0.0; y = 0.0; r = 0.0;
   }

   public Circle_server() {}

   public void DDD(double d1, double d2, double d3)
   {
       Pi = 3.141590118408203;
       x = d1; y = d2; r = d3;
   }

   protected double circumference()
   { return 2.0 * Pi * r;}

   private double area()
   { return 3.141590118408203 * r * r;  }

   public double cirCumXArea()
   { return circumference() * area();}

   public double getValPrimDouble(String string)
   {
       double d;
       if (string.equals("x"))
           d = x;
       else if (string.equals("y"))
           d = y;
       else if (string.equals("r"))
           d = r;
       ...
       return d;
   }

   public void setValPrimDouble(double d, String string)
   {
       if (string.equals("x"))
           x = d;
       if (string.equals("y"))
           y = d;
       if (string.equals("r"))
           r = d;
       return;
   }
}
```

Figure 3: Shows the server class generated from the class in Figure 1-(a)

have access to the Internet. We also assume that connections over the Internet are reliable and secure. In addition, we require host systems to be trusted, available, and willing to participate in client application executions. We assume the existence of Server Name Services (SNS)s. SNSs maintain a list of addresses of available host systems. The addresses of SNSs are assumed to be known; they can be retrieved, for example, from a central web site. We require HTTP servers to be running on host systems. HTTP servers are used by the RMI mechanism for downloading classes, such as the stub classes, from the host system to the home system.

4.1 Processing Stage

In the processing stage, an application is transformed into a client application and a server application. This consists of applying the decomposing strategy on selected classes, adapting the application to work with the decomposed classes, and generating classes for supporting the execution of decomposed applications. The inputs to this stage are the class files of an application and a list of selected classes

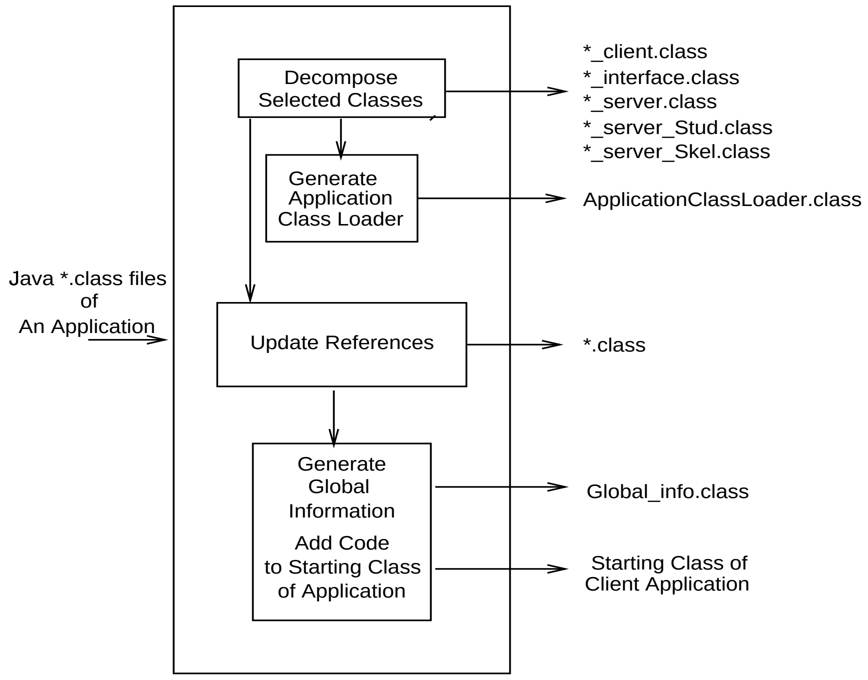
Decompose Selected Classes
Generate Application Class Loader
Update References
Generate Global Information
Add Code to Starting Class of Application
Java *.class files of An Application
*_client.class
*_interface.class
*_server.class
*_server_Stud.class
*_server_Skel.class
ApplicationClassLoader.class
*.class
Global_info.class
Starting Class of Client Application

Figure 4: The processing stage. The *_server.classes and the ApplicationClassLoader class are distributed only to host systems. The *_client.classes and other class files are distributed to users. The *_interface.classes are distributed to users and host systems.

to decompose. Figure 4 shows the class files generated in the processing stage. Next, we discuss the supporting classes: classes that provide server classes with I/O capabilities of the home system, and the class responsible for instantiating server classes on host systems.

4.2 Customized API Classes

In the Java class library, there are API classes that implement neither the java.io.Serializable interface (i.e., are not serializable) nor the remote interface. An example is the java.lang.System class which provides access to system resources of the JVM and has some of its methods implemented in native code. Some classes are not serializable because they contain references (addresses) to native code that are only valid in the originating JVM. Other classes do not implement the remote interface because of security concerns. However, in certain situations having access to such API classes is appropriate for server classes; the original unprocessed class may contain such a reference (e.g., a call to the java.System.out.println() method). In the following subsections, we discuss Remote Host Objects (RHOS) that provide I/O capabilities to the server classes from the home system.

4.2.1 langSystemServer

We provide server classes with access to the home system's standard input, output, and error capabilities. This involves providing access to the java.lang.System class's in, out, and err fields. Because the System class is a final class (i.e., cannot be subclassed), we create a wrapper class, langSystemServer, that calls the methods of the System class. The langSystemServer class implements the remote interface api.lang.SystemXface (see Figure 5). The SystemXface interface defines a method for (almost) every

```
package api.lang;

import java.rmi.Remote;
import java.rmi.RemoteException;
import java.io.*;

public interface SystemXface extends Remote {

// standard out
 ...

 public void outClose() throws java.rmi.RemoteException;
 public void outFlush() throws java.rmi.RemoteException;

 public void outPrint(boolean b)  throws java.rmi.RemoteException;
 public void outPrint(int i)      throws java.rmi.RemoteException;
 public void outPrint(long l)     throws java.rmi.RemoteException;
 public void outPrint(float f)    throws java.rmi.RemoteException;
 public void outPrint(double d)  throws java.rmi.RemoteException;
 public void outPrint(char[] s)   throws java.rmi.RemoteException;
 public void outPrint(String s)   throws java.rmi.RemoteException;
 public void outPrint(Object obj) throws java.rmi.RemoteException;

 public void outPrintln()          throws java.rmi.RemoteException;
 public void outPrintln(boolean b)  throws java.rmi.RemoteException;
 public void outPrintln(int i)      throws java.rmi.RemoteException;
 public void outPrintln(long l)     throws java.rmi.RemoteException;
 public void outPrintln(float f)    throws java.rmi.RemoteException;
 public void outPrintln(double d)  throws java.rmi.RemoteException;
 public void outPrintln(char[] s)   throws java.rmi.RemoteException;
 public void outPrintln(String s)   throws java.rmi.RemoteException;
 public void outPrintln(Object obj) throws java.rmi.RemoteException;

 public void outWrite(int b) throws java.rmi.RemoteException;
 ...
 }
```

Figure 5: Shows parts of the abstract remote methods of the SystemXface class

method in the fields. For the err and out fields, access is provided for the print(), close(), and flush() methods. For the in field, access is allowed to all java.io.InputStream methods, except for the in.read() methods that read bytes of data into a byte array.

We do not provide access to all the methods of the System class. Some methods like the arrayCopy() method, which copies the value of a source array to a destination array, are unnecessary since the server classes have access to their own System class on their systems. Other methods are not provided because of security concerns, such as the getProperties() method which reveals the system parameters of a JVM.

4.2.2 ioServer

The java.io package consists of a large number of classes that handle I/O from memory, file, and stream. Host systems are allowed access to a small subset of the java.io classes that operate on files, and on the standard input, output, and error files (i.e., the fields of the java.lang.System class) on home systems. Of course, this does not include abstract classes such as Reader. We do not provide remote classes for retrieving I/O from memory (e.g., ByteArrayInputStream, CharAarrayReader, etc.), since this can be handled by the server classes on their systems. We also do not provide a remote class for the FileDescriptor class (which represents low level open file or socket), since it cannot be instantiated and is not serializable.

We provide an ioServer class that implements the remote interface and services requests for instantiating classes of the java.io package on the home system. The source and destination of an I/O operation can be specified by using a file name, by creating a File Object, or by denoting one of the System class's stream fields. A normal method invocation in the original class, for example, new BufferedReader(new

FileReader (“foo.bar”)) appears in a server class as a request to the home system’s ioServer object to instantiate BufferedReader;FileReader;“foo.bar”. When the ioServer object receives the request, it instantiates the FileReader class with “foo.bar” and passes this created input stream to an instantiated BufferedReaderSr class which implements a remote object version of the java.io.BufferedReader class. The ioServer object then passes the reference of the BufferedReaderSr back to the server object. When input or output streams are chained, as in the previous example, we only provide remote access to the outer stream in the chain.

4.2.3 Application Class Loader

```
import Shape.Circle_face;
....

public synchronized class typeClassmsppCir implements typeClassX
{
   public Remote InvokeNewInst
           (String string, Vector vector1, Vector vector2)
   {
....
           object2 = Class.forName(object2);
           Object aobject[] = new Object[2];
           aobject[0] = vector1;
           aobject[1] = vector2;
           Class aclass[] = ...
           if (string.equals("Shape/Circle_server"))
           {
               Constructor constructor =
                 object2.getConstructor(aclass);
               Circle_face circle_face =
     (Circle_face)constructor.newInstance(a( object);
               object1 = circle_face;
               return object1;
       }
....
}
```

Figure 6: Shows parts of the Circle’s Application Class Loader

The ApplicationClassLoader class is responsible for dynamically instantiating the server classes on host systems when requested by client objects. The ApplicationClassLoader class provides a method, InvokeNewInst(), that receives as parameters the name of the server class to instantiate and references to RHOS. This method has a simple structure. It compares the class name received with its internal list of classes it services. If it finds a match, the InvokeNewInst() method instantiates the requested server class and returns a remote reference to this object. Otherwise, the method returns a null reference. The ApplicationClassLoader class uses Java’s reflection mechanism for instantiating a server class with two constructor arguments. The first argument is a java.util.Vector containing names of RHOS and the second is a Vector containing references to RHOS. The Vector of names is used for retrieving references of RHOS (by name) in the Vector of RHOS; corresponding entries in both Vectors have the same index. The server class is responsible for initializing its references to RHOS when instantiated by the ApplicationClassLoader class. Note that the calling of initializing methods (e.g., Circle_client’s second constructor method invokes the Circle_server’s DDD() method) is left up to the client objects on home systems. Figure 6 shows the ApplicationClassLoader for the Circle class. An ApplicationClassLoader class is automatically generated for each decomposed application.

4.2.4 Adapting the Application

To adapt the application to work with the decomposed classes, our tool performs three different operations. First, references in decomposed and undecomposed classes are updated. References to original classes that are now decomposed are changed to reflect the appropriate classes (i.e., _client.class, _server.class, or _interface.class). References to supported resources on the home system that appear in the server classes are also changed accordingly. An example is a call to the System.out.print() method that appears in the original class and in the server class. Such references are replaced with references to RHOS. Second, a Global_Info class is generated. This class maintains essential information, such as the name of the Application class loader and references to RHOS, which is needed by the client object when instantiating server classes. The Global_Info class facilitates the instantiation of client classes from anywhere in the application's class tree. Third, special code is added to the main method of the starting class of the application. This special code consists of three different code segments. The first segment sets up the RMISecurityManager. This is required to enable classes associated with remote objects to be loaded (e.g., stub classes, return type, etc.). The second segment retrieves host system addresses from SNSs. This segment looks up Internet addresses of SNSs in an address profile file contained in the directory where the user executes the client application. The last segment instantiates the RHOS. This segment also creates Vectors of names and RHOS, and sets the appropriate field in the Global_Info class.

4.3 Initialization Stage

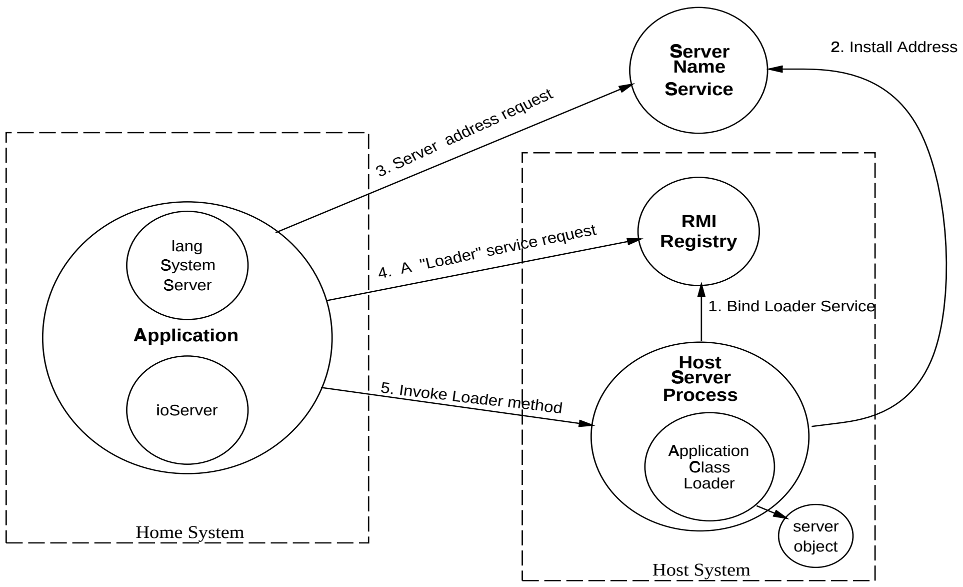

Figure 7: Initialization Stage

In the initialization stage, the objects of a client application establish links to their corresponding server objects on host systems. Figure 7 shows the required initial steps when a decomposed application starts running. We assume that host systems are running before client applications start executing.

A host system executes a Host Server Process (HSP) which services client object requests. The HSP supplies methods that instantiate and return references of server objects running on the host system. At start up, the HSP registers a reference of itself on a locally running registry (e.g., RMI registry) with the name Loader (see Figure 7, $\boxed{1}$). A request for the Loader service on the registry returns a remote reference to the HSP. The HSP then forwards its machine address and the port identification (i.e., port number) of the locally executing registry to SNSs $\boxed{2}$.

In a home system, when a client application starts executing, it retrieves the addresses of host systems from SNSs $\boxed{3}$ [7]. The client application then requests the registries at these locations for references to Loader services $\boxed{4}$. Within the client application, every client object invokes a Loader service passing the following parameters: the name of the ApplicationClassLoader, the name of the server class (to instantiate) and references to RHOS $\boxed{5}$. Once HSP receives these parameters, it instantiates the specified ApplicationClassLoader. HSP then invokes the ApplicationClassLoader.InvokeNewInst() method to instantiate the requested server class. If the server class instantiation is successful, the client object gets a reference to its remote counterpart (i.e., server object). Once the client application classes obtain the remote references to their application server classes, the execution of the application can proceed.

4.3.1 Failure Modes

Decomposition introduces new failure modes to applications. The decomposed application must deal with exceptions associated with remote method invocation. RMI exceptions may result from network uncertainty (packets get lost), failure on host systems (e.g., crashes), server objects being prematurely collected by the garbage collector on server systems, RMI classes, such as stubs classes, not being found at runtime, etc. From the perspective of the client side of the decomposed application, we address RMI exceptions caused by server objects being unavailable. Events that generate such type of RMI exception are: when a client object is unable to link with its corresponding server object or when client objects' references to server objects become invalid.

We provide two approaches for handling RMI exceptions. The first approach is simple and general. When a RMI exception occurs, the decomposed application reports the error (by writing to the standard error of the home system) and terminates. This, however, may not be reasonable for some (long running) applications. The second approach attempts to recover from RMI exceptions caused by server objects being unavailable. When a decomposed application encounters such type of RMI exception, it first tries to reconnect with the same host system (if it originally provided the server object) or with a different host system. If unsuccessful, it repeats the same procedure with another host system. If unsuccessful after a predetermined number of tries, the decomposed applications reverts to the first approach. The second approach requires that client objects cache the list of server names they initially receive at instantiation. In addition, if recovery is for a client object to reestablish connection with its corresponding server object, then the client object must (if necessary) restore the state of the server object (i.e., all non final static variables which were set in the previous execution must be forwarded to the newly instantiated server object).

4.4 Implementation

We implemented a prototype of the protection architecture as described in this Section for Java 1.1. Our tool takes as input the name of the starting class of the application, option parameters, a list of class names to decompose, and the Java binary class files of the application. We used freely available software to aid in the decomposition of Java classes. Supplied in [19], classes of the Jcf package were used for reading Java class binaries into a data structure suitable for manipulation. To this package we added classes to enable us to read the bytecode of a Java class binary. Because generating a Java class binary directly is a cumbersome process due to the indirection associated with entries in the constant

[7]By caching addresses of hosts systems from previous executions, applications can avoid interacting with SNSs.

pool, we used Jasmin [9], a Java assembler, to generate Java class binaries. Thus, our tool generates ASCII descriptions [8] of Java classes, which are then assembled to (Java class) binaries by using Jasmin.

Additionally, we utilize two network programs to help implement host systems. The first program, ClassServer [15], is a simple webserver which enables classes to be loaded over the network via the RMI mechanism. It provides the loading of stub classes from host system to home systems; it is also used by the HSP for retrieving the network classes (e.g., skeleton classes) on the host system. Supplied with the JDK, the second program is the rmiregistry. This program provides the bootstrap registry service for locating services of the HSP (by name) on a host system. The client application can request the service of HSP by knowing the IP address (or the machine name) where the registry executes and the port it (registry) listens for servicing. (In our prototype, SNSs maintain a list of consisting of machine addresses and ports associated with the rmiregistries.)

5 Discussion

In this section, we discuss the achieved security and performance.

5.1 Security

In our approach, we have converted a Java class into a black box where users have no way of accessing or observing the internals of the class. Our approach eliminates the reverse-engineering threat to Java class elements declared with private or protected access. Client classes distributed to users do not include these types of class elements or will users be aware of their existence. We have also significantly reduced the susceptibility to reverse-engineering of class elements declared with package or public access. Even for these types of class elements, users do not get direct access to them. Access to instance class elements is provided via proxy methods that invoke their counterpart methods on a server object running on a trusted system. Only static elements declared with public or package access in the original class appear in the client class. However, software developers using our approach will not use these types of static class elements for maintaining their secrets or proprietary information.

5.2 Performance

We evaluate the performance of our approach by measuring and comparing the execution time of method invocations of decomposed applications with original applications. Our performance experiments were conducted on Sun ULTRAs running Solaris 2.7 connected by a 10 Mbps Ethernet LAN on the same subnet. The systems designated as host systems were lightly loaded. For our experiments, we used Java 1.1.6 version 3 with the JIT option set off. We obtained explicit timing measurements by instrumenting our applications with calls to System.currentTimeMillis() (before and after a method invocation). The timing results shown in Table 2 represent the average of 5000 method invocations except for constructor methods (these are the entries with a star in the method column in Table 2) which were calculated from the average of 100 invocations. We evaluate the performance of three applications.

The first application, CompPi, calculates pi to the precision of 50 digits. We select the Pi class which performs the Pi calculation for decomposition. The Pi.Pi() constructor method receives the parameter for setting up the precision, and the Pi.execute() method computes Pi by using Machin's formulas. The Pi.execute() method returns an object of the type java.lang.BigDecimal. The second application, DataB, simulates the functionality of a database. Here, we decompose the DataApp class which maintains the

[8]The ASCII descriptions of Java classes were written using a simple Java assembly language.

Application	Method	Description	Remote (ms)	Local (ms)	Remote/Local
CompPi	Pi.Pi()*	1 arg/no result	860.19	0.21	4096
	Pi.execute()	no args/result	24.58	17.09	1.43
DataB	DataApp.DatApp()*	no arg/no result	941.07	24.14	39
	DataApp.method4()	no args/result	9.05	0.082	110
	DataApp.method32()	no args/result	19.73	0.449	44
	DataApp.method64()	no args/result	23.06	0.872	26
MainXor	SimpleXor.SimpleXor()*	no args/no result	1548.1	9643.37	.160
	SimpleXor.rcvData()	2 args/80 words array	3.91	1.03	3.8
	SimpleXor.retKey()	no args/80 words array	2.89	N/A	N/A

Table 2: Average execution time of remote and local method invocations (5000 calls) (* 100 calls)

data. The DataApp.DatApp() constructor method reads a local data file to initialize its state. The DataApp class provides methods (method64(),method32(), and method4()) for retrieving data stored in memory. These methods randomly pick items to return; the return parameter of these methods is a Vector containing java.lang.String objects[9].

In the third application, MainXor, a local text file is read and passed as data to the SimpleXor class which performs a simple encryption. We select the SimpleXor class for decomposition. In SimpleXor's constructor method an instance of the java.security.SecureRandom class is instantiated. The SecureRandom object takes approximately an average of 7 seconds to generate its random seed. The rcvData() method (of SimpleXor) receives two parameters, a data array of 80 integers and the length of the array to process. This method requests 80 random bytes via a call to the java.security.SecureRandom.nextByte() method, applies an exclusive OR operation on the data array with the randomly generated key, and returns an encrypted version of the data array. The retKey() method returns the randomly generated key used in the last invocation of the rcvData() method.

Our initial performance results show that remote method invocations that do useful work, such as SimpleXor.rcvData(), Pi.execute(), and DataApp.method##(), differ from their local versions by 1-2 orders of magnitude. Our results also show that local method invocation is significantly better than remote when no (hard) work is done in the methods. Examples of these cases are: when a method simply returns data such as SimpleXor.retKey(), or when a method sets instance variables of an object, such as in Pi.Pi(). We also expected our results to show that constructor method invocation to be more costly for decomposed classes than for the original classes. (Recall that decomposed classes have the added overhead associated with linking with their corresponding server objects.) The Compi and DataB applications exhibit this type of performance. However, MainXor illustrates a special case where instantiating classes remotely yields better performance than locally! In the SimpleXor object, successive invocations of the java.security.SecureRandom class on the host system used the same pseudo-random number generator for their seed data while for the local execution we instantiated this class for every run.

From these performance experiments, we conclude the following: first, instantiating computationally intensive classes (e.g., data base computations, cryptographic tools like pseudo-random generators) on remote systems where successive invocations can make use of the previously computed state is advantageous, and second, the cost of remote method invocations is excessive when compared with its local versions when no useful work is performed in the methods. We also note that our approach is suitable

[9]The number at the end of the method indicates the number of String objects contained in the Vector Object.

for situations where the users have to access data on remote machines (e.g., remote databases). In these cases the communication cost is unavoidable. It is also appropriate for large granularity when server objects (on host systems) are not called frequently.

6 Conclusion

In this paper, we presented a decomposition methodology that protects the classes of Java applications from reverse-engineering and code-understanding. We implemented, DISSECT, a tool that employs the decomposition methodology and provides runtime support for running decomposed applications. There are many future directions, such as addressing security issues, scalability of host systems, and specifying system designs which employ decomposition as a built-in component.

References

[1] Ahpah Software, Inc. SourceAgain. http://www.ahpah.com.

[2] A. D. Birrell and B. J. Nelson. Implementing Remote Procedure Calls. In ACM TOCS, 2(1):39-59, Feb. 1984.

[3] C. Collberg, C. Thomborson, and D. Low. A Taxonomy of Obfuscating Transformations. http://www.cs.auckland.ac.nz/~collberg/Research/Publications/CollbergThomborson97a/index.html, July 1997.

[4] C. Collberg, C. Thomborson, and D. Low. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. In Proceedings of POPL 1998, pages 184-196, Jan. 1998.

[5] DashO Obfuscation Edition. http://www.preemtive.com/products.html.

[6] N. Eastridge. Java Shrinker & Obfuscator v1.04. http://www.e-t.com/jshrink.html, March 1999.

[7] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going Beyond the Sandbox: An Overview of the New Security Architecture in the *Java*^{TM} Development Kit 1.2. In Proceedings of the USENIX Symposium on Internet Technologies and Systems, Dec. 1997.

[8] C. S. Horstmann and G. Cornell. Core Java 1.1, Volume II-Advanced Features. Sun Microsystems Press, 1998.

[9] Jasmin. http://www.cat.nyu.edu/meyer/jasmin/.

[10] KB Sriram. HashJava.
http://www.sbtech.org/, Sept 1997.

[11] G. McGraw and E. W. Felten. Securing Java: getting down to business with mobile code, 2nd edition. John Wiley & Sons, 1999.

[12] D. Malkhi, M. Reiter, and A. Rubin. Secure Execution of Java Applets using a Remote Playground. In Proceedings of the 1998 IEEE Symposium on Security and Privacy, 1998.

[13] J. Meyer and T. Downing. Java Virtual Machine. O'Reilly & Associates, 1997.

[14] T. Sander and C. Tschudin. Towards Mobile Cryptography. In Proceedings of the 1998 IEEE Sym. on Security and Privacy, pages 215-224, 1998.

[15] Sun Microsystems, Inc. ClassServer. ftp://java.sun.com/pub/jdk1.1/rmi/class-server.zip, 1997

[16] Sun Microsystems, Inc. Java Object Serialization Specification, Revision 1.2, Dec. 1996.

[17] Sun Microsystems, Inc. Java Remote Method Invocation, 1997.

[18] Uwe G. Wilhelm. Cryptographically Protected Objects.
http://lsewww.epfl.ch/˜wilhelm/CryPO.html, May 1997.

[19] Matt T. Yourst. Inside Java Class Files.
http://www.laserstars.com/articles/ddj/insidejcf/.

[20] Hanpeter van Vliet. Mocha, the Java Decompiler.
http://www.brouhaha.com/˜eric/computers/mocha.html, Aug 1996.

[21] WingSoft Corporation. WingDis Java Decompiler.
http://www.wingsoft.com/.