Using Reflection as a Mechanism for Enforcing Security Policies in Mobile Code

Ian Welch and Robert J. Stroud

University of Newcastle-upon-Tyne, United Kingdom NE1 7RU
{I.S.Welch, R.J.Stroud}@ncl.ac.uk,
WWW home page:http://www.cs.ncl.ac.uk/people/
{I.S.Welch, R.J.Stroud}

Abstract. Several authors have proposed using code modification as a technique for enforcing security policies such as resource limits, access controls, and network information flows. However, these approaches are typically ad hoc and are implemented without a high level abstract framework for code modification. We propose using reflection as a mechanism for implementing code modifications within an abstract framework based on the semantics of the underlying programming language. We have developed a reflective version of Java called *Kava* that uses bytecode rewriting techniques to insert pre-defined hooks into Java class files at load time. This makes it possible to specify and implement security policies for mobile code in a more abstract and flexible way. Our mechanism could be used as a more principled way of enforcing some of the existing security policies described in the literature. The advantages of our approach over related work (*SASI*, *JRes*, etc.) are that we can guarantee that our security mechanisms cannot be bypassed, a property we call *strong non-bypassability*, and that our approach provides the high level abstractions needed to build useful security policies.

1 Introduction

We are interested in applying ideas of behavioural reflection [11] to enforcing security mechanisms with mobile code. Mobile code is compiled code retrieved from across a network and integrated into a running system. The code may not be trusted and therefore we to need ensure that it respects a range of security properties. The Java security model provides a good degree of transparent enforcement over access to system resources by mobile code but it does not provide the same degree of transparency for control over access to application level resources. A number of authors [4][5] have tackled this problem and made use of code modification in order to add more flexible enforcement mechanisms to mobile code. However, although they have provided higher level means of specifying the security policies they wish to enforce, they have used code modification techniques that have relied upon structural rather than behavioural changes. We argue that reflection can be used to provide a model for behavioural change that is implemented using code modification. This provides a greater degree of separation

F. Cuppens et al. (Eds.): ESORICS 2000, LNCS 1895, pp. 309–323, 2000.

between policy and implementation than the current systems provide. It also addresses some of the drawbacks of existing schemes. In particular, it makes it possible to specify security policies at a more appropriate level of abstraction. Another advantage of our approach is that it provides a property we call *strong non-bypassability*. This guarantees the enforcement of security mechanisms by removing the opportunity to bypass them using the same mechanisms that were used to produce them. For example, approaches that use renaming are vulnerable to attacks that discover and exploit the real name of the underlying resource.

The paper is structured as follows. In section 2 we introduce the Java security model, describing its evolution and pointing out some of its drawbacks. In section 3 we describe our use of reflection to enforce security and introduce our reflective Java implementation *Kava*. In section 4 we provide two examples of how *Kava* can be used and show how it integrates with the existing Java security model. In section 5 we describe and evaluate some related work. Finally in section 6 we conclude with a discussion of the advantages and disadvantages of our approach.

2 Evolution of Java Security Model

Java [10] is a popular choice for researchers investigating mobile code technologies. Java has strong support for mobility of code and security. The Java class loader mechanism supports mobile code by allowing remote classes to be loaded over the network, and a security manager enforces checks on the use of local system resources by the mobile code. The ability to supply a user-defined class loader and security manager makes it possible to customise these mechanisms to a certain extent.

In the past few years the Java security model has undergone considerable evolution. In the JDK1.0 security model [9] any code run locally had full access to system resources while dynamically loaded code could only access system resources under the control of a security manager. System libraries have predefined hooks that cause `check access` methods provided by the security manager to be called before sensitive methods were executed. The default security manager sandbox provided minimal access and in order to support a different security model a new security manager would have to be implemented.

The concept of trusted dynamically loaded code was introduced in JDK1.1 [14]. Any dynamically loaded code that was digitally signed by a trusted code provider could execute with the same permissions as local code.

Recently JDK1.2/Java2 [15][16] (see figure 1) has introduced an extensible access control scheme that applies both to local code and dynamically loaded code. Fine-grained access to system resources by code can be specified in a policy file on the basis of the source of the code, the code provider (indicated by who cryptographically signed the code), and the user of the code. Unlike earlier versions of the JDK this policy file allows the security model to be adjusted without writing a new security manager. This is because the security manager has standard access control checkpoints embedded in its code whose behaviour is determined by the selection of permissions enabled in the policy file. The

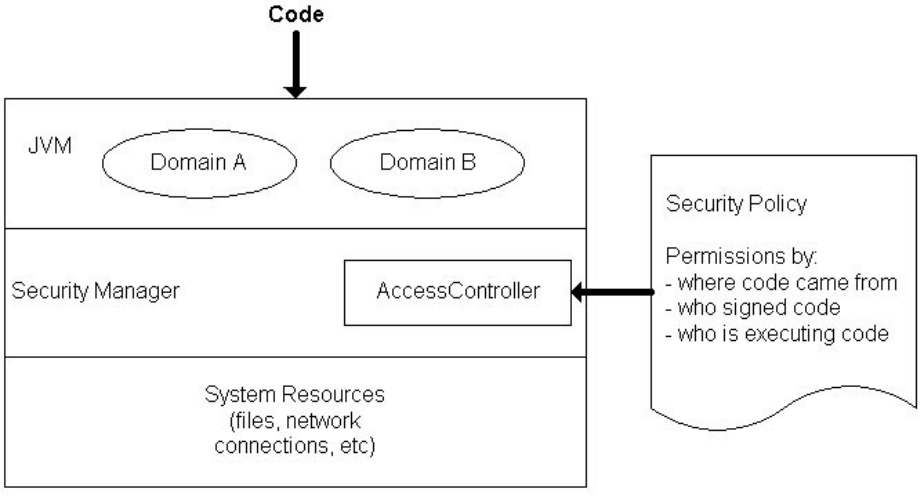

Fig. 1. Overview of Java2 Security Architecture

evaluation of the permissions is handled by an access controller that defines how different permissions are reconciled to give an overall access control decision. New permissions can be defined but explicit checks for the permissions must be added to the security manager or application code if the permissions apply to application resources rather than system resources.

2.1 Example: Extending the Java Security Model

To provide a flavour of the problems of the current Java security model we provide the following example of the definition of a customised security policy.

Imagine that an application developer has created a program for watching television broadcasts over the Internet called *WorldTV*. We may want to impose a security policy on the application to constrain which channels a user may watch. For example, if a machine is provided in a public place we might restrict the channels to a selection of local news channels.

The recommended steps for customising the Java security model in order to support such a policy [8] are:

- Define a permission class.
- Grant permissions.
- Modify resource management code.

A new permission class that represents the customized permission to *watch a channel* must be defined. It is realized by defining a class `com.WorldTV.ChannelPermission` that subclasses the abstract class `java.Security.Permission`.

Then the appropriate permission must be granted by adding entries into the security policy. In the example below we allow any application to watch channel 5.

```
grant
{
  permission com.WorldTV.ChannelPermission "5", "watch";
}
```

Finally, we must add an explicit check into the application's resource management code that calls `AccessController`'s `checkPermission` method using a `com.WorldTV.ChannelPermission` object as the parameter. If the application has not been granted the permission then an `AccessControlException` is raised. `AccessControlException` is a runtime exception so does not need to be declared in the class' interface.

```
public void watchChannel(String channel) {
  com.WorldTV.ChannelPermission tvperm = new
              com.WorldTV.ChannelPermission(channel, "watch");
  AccessController.checkPermission(tvperm);
  ...
}
```

2.2 Discussion

The ability to define application specific permissions makes the Java security model easily extensible. In previous versions of the Java security model the only way to implement application specific policy was to create a new `SecurityManager` class. For the example above a new method `checkChannel` would have had to been added to the `SecurityManager` class. By the time all possible checks had been added to `SecurityManager` the resulting interface would be too large and unwieldy for use and analysis. Through the use of typed access-control permissions and an automatic permission handling mechanism (implemented in the `AccessController` class) only a single method `checkPermission` is required. This represents an extensible and scalable architecture.

However, the application developer must still identify where the permission checks should be added into the application code and manually insert the checks. This means that security code is tangled with application code and this makes management and maintenance difficult. Whenever a new permission type is added then the application developer must access the source code of the application and modify then recompile. This raises the possibility of error as the modifications are made, and it is possible in the case of mobile code that the source code itself is not available.

A better approach would be to use something similar to the `SecurityManager` approach for system classes where hooks are added to the system classes

that force the check methods of the `SecurityManager` to be invoked when certain critical methods are executed. Essentially it should be possible to take application code and automatically add hooks that invoke security mechanisms at appropriate places. For example, instead of manually modifying `watchChannel` the application developer should just be able to specify that the permission `ChannelPermission` is checked before this method can be invoked. This would result in a better separation of concerns between application code and security code.

3 A Reflective Approach Using *Kava*

Our approach is based on the use of metaobject protocols to provide flexible fine-grained control over the execution of components. The metaobject protocol implements the security mechanisms that enforce security policies upon application code. This effectively allows security checks to be inserted directly into compiled code, thus avoiding the need to recode applications in order to add application specific security checks. Figure 2 below presents the *Kava* reflective security architecture. We discuss each aspect of the architecture in the following sections.

3.1 Reflective Object Oriented Model of Computation

A reflective computational system [12] is a system that can reason about and make changes to its own behaviour. Such a system is composed of a base level and a meta level. The base level is the system being reasoned about, and the meta level has access to representations of the base level. Manipulations of the representations of the base level at the meta level result in changes to the behaviour of the base level system.

These notions of reflection have been extended to include the concept of the metaobject protocol [11] where the objects involved in the representation of the computational process and the protocols governing the execution of the program are exposed. A *metaobject* is bound to an object and controls the execution of the object. By changing the implementation of the metaobject the object's execution can be adjusted in a principled way.

In order to use reflection as a mechanism to enforce security properties we need to be able to control all interactions between the object and its environment. Therefore we need to be able to control all interactions with an object. This includes self-interactions. Thus, we need to control the following behaviours:

- Method invocation by an object.
- Method execution.
- Setting and getting of state.
- Object instantiation.
- Object construction.
- Exception raising.

The metaobject bound to the object defines the object's behaviour. Security enforcing mechanisms can be implemented by the metaobject in order to realise a security policy. In order to provide a guarantee that the security properties are honoured it must be impossible to bypass the metaobject. We call this property strong non-bypassability. We have implemented a reflective Java that implements this reflective model of object oriented computation and also has the property of strong non-bypassability.

In the next two sections we introduce the reflective version of Java we have developed and describe how it achieves this property we call strong non- bypassability.

3.2 Kava Metaobject Protocol

We have developed a reflective Java called *Kava* [19] that gives the control over the behaviour of objects that is required to add security enforcement at the meta layer. It uses byte code transformations to make principled changes to a class' binary structure in order to provide a metaobject protocol that brings object execution under the control of a meta level. These changes are applied at the time that classes are loaded into the runtime Java environment. The meta layer is written using standard Java classes and specifies adaptations to the behaviour of the components in a reusable way. Although neither bytecode transformation nor metaobject protocols are new ideas, our contribution has been to combine them. Byte code transformation is a very powerful tool but it is in general difficult to use, as it requires a deep knowledge of class file structure and byte code programming. What we do is use a load-time structural metaobject protocol (such as provided by Joie [1] or JavaClass [3]) in order to implement a runtime metaobject protocol. Working at the byte code level allows control over a wide range of behaviour. For example, the sending of invocations, initialisation, finalization, state update, object creation and exception raising are all under the control of *Kava*.

3.3 Meta Level Security Architecture

Security policy enforcement (see figure 2) is built on top of the runtime metaobject protocol provided by *Kava*. Metaobjects implement the security mechanisms that enforce the policy upon the application. Each object has a metaobject bound to it by *Kava*. In effect each metaobject acts a reference monitor for each application object. The binding is implemented by adding hooks directly into the binary code of the classes. As this binding exists within the component itself instead of in a separate wrapper class we argue that we are achieving a strong encapsulation of components. Outside parties cannot bypass the wrapping and therefore the security implemented in the metalevel by simply gaining an uncontrolled reference to the object because no such references exist. This type of binding we refer to as *strong non-bypassibility*. There are two common techiques for adding interceptions to Java classes : creation of a proxy class, or renaming methods in the class and replacing them with proxy methods. The proxies add

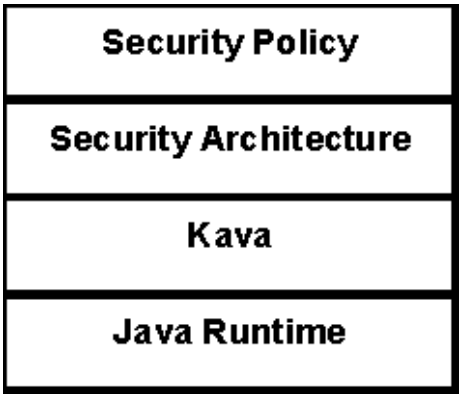

Fig. 2. Overview of *Kava* Security Architecture

the security enforcement. These approaches only support weak non-bypassability as there is the possibility that a reference to the real class might escape or the name of the real method might be discovered. This would make it possible to bypass the security enforcement.

The *Kava* system, binding specification and the metaobjects must form part of the trusted computing base. The *Kava* system and binding specification are installed locally and can be secured in the same way as the Java runtime system. However, the metaobjects may exist either locally or be retrieved across the network. This raises the possibility that the metaobjects themselves might be compromised. In order to counter this threat we use a specialised version of a classloader that verifies the identity and integrity of the metaobject classes using digital signing techniques. Each metaobject is digitally signed using a private key of the provider of the metaobject. The public key of the provider exists in the local public key registry on the host where the *Kava* system is installed. The digital signature of the downloaded metaobject is then verified using the local copy of the provider's public key. If there is discrepancy then a security exception is raised and the system halts. This prevents malicious combinations of application objects and metaobjects.

4 Example

In this section we provide two examples of how *Kava* can be used as the basis for implementing security enforcement mechanisms using metaobjects. The first example reworks the simple example from our discussion of the Java security model (an example of static permissions), and the second example is of a security policy that limits the total number of bytes that can written to the local file system by an application (an example of dynamic permissions).

4.1 Overview of Approach

Our approach leverages upon the existing Java security model. As pointed out earlier in section 2, the main problem with the Java security model is the lack of automatic addition of enforcement code to application code. *Kava* provides a principled way of doing this.

The enforcement *Kava* adds depends on the particular security policy being enforced, and the structure of the application. There are two particular phases in the *Kava* system. These are loadtime and runtime.

Loadtime. At loadtime *Kava* must determine what operations are trapped. These decisions are encapsulated by a `MetaConfiguration` class. There should be one for each security policy to be enforced. For example, there might be one configuration for a multilevel policy where all interactions between object must be trapped and another configuration for a simple access control policy where only method invocations are trapped. The `MetaConfiguration` class is responsible for parsing the *policy file* which provides additional information about the application that the security policy is being applied to and the particular policy settings for that application. For example, what metaobjects to bind to which classes, and what types of operation to trap. The *policy file* uses an extended form of the standard JDK1.2 syntax for security policies.

Runtime. At runtime the traps inserted under the control of the `MetaConfiguration` class switches execution for the base level (the application code) to the meta level (the metaobject associated with each object). The metaobject performs the permission checks necessary to implement the particular security policy. A specialised `Policy` object associates the permissions with the loaded classes. This is a specialisation of the default `Policy` class because it has to map additional permissions against classes in order to support the security policy.

4.2 Example: *WorldTV*

Using the *Kava* approach the developer carries out the first two steps of defining a permissions class and granting permissions as necessary. However, instead of taking the application code and editing it the application programmer defines a new `Metaobject` class and places the enforcement code here. For example,

```
import kava.*;
public class EnforcementMetaobject implements Metaobject
{
  public boolean beforeReceiveMethod(Reference source,
    Method myMethod, Value[] args)
  {
    com.WorldTV.ChannelPermission tvperm = new
              com.WorldTV.ChannelPermission(
```

```
                (String)args[0].getValue, "watch");
    AccessController.checkPermission(tvperm);
    return Constants.INVOKE_BASE;
  }
}
```

This redefines how a method invocation received by an object is handled. It enforces a check before the execution of the method invocation that the correct `ChannelPermission` is held by the thread executing the code.

The next step the application programmer must do is to specify which methods of which class are controlled by this enforcement metaobject. This is included in the expanded version of the standard Java policy file.

```
bind
{
  kava.EnforcementMetaobject *::watchChannel(String);
}
grant
{
  permission com.WorldTV.ChannelPermission "5", "watch";
}
```

The *bind* specification indicates to the `MetaConfiguation` class which methods of which class should be trapped. In this case any method named `watch` Channel with a single parameter of type `String` belonging to any class will be trapped and have security checks enforced upon it.

4.3 Example : LimitWrite

The previous example is a traditional fairly static access control security policy. *Kava* can also enforce dynamic security policies that depend upon changing state. The following example shows that *Kava* could be used to enforce a policy that places a million-byte limit on the amount of data that may be written to the file system.

The first task is to define a new permission type that has a dynamic behaviour. We define a permission class `FileWritePermission` that subclasses `java.security.Permission`. This new permission's constructor defines the maximum number of bytes that may be written to the file system. It also adds a new method `incrementResourceCounter(long n)` that increments the global count of the number of bytes written to the file system. Finally it defines the `implies` method so that when the `AccessController` calls the `implies` method to see if the permission being checked is held, the current number of bytes written is compared with the maximum to determine if this is true or not.

The second step is to specify the enforcement metaobject. It has a straightforward structure as the security policy decision is specified within the `Permission` class.

```
import kava.*;
public class FileEnforcementMetaobject
  implements Metaobject
{
  public boolean beforeSendMethod(Reference source,
   Method myMethod, Value[] args)
  {
   FileWritePermission perm = new
              FileWritePermission();
   perm.incrementResourceCounter(Integer.toLong(args[2].
   getValue());
   AccessController.checkPermission(perm);
   return Constants.INVOKE_BASE;
  }
}
```

Here the behaviour of an object sending an method invocation to another object is redefined. We do this because *Kava* cannot rewrite library classes unless the JVM is changed. A new `FileWritePermission` is constructed with a throwaway value. Then the context for the permission is updated by calling `setPermissionContext` using the number of bytes written to the file. Here we are exploiting the knowledge that the third argument always the number of bytes to be written to the file.

The third step is to specify the policy file :

```
bind
{
  kava.FileEnforcementMetaobject
      (* extends FileWriter).write(*, int, int);
}
grant {
  FileWritePermission "1000000";
}
```

The policy file determines which methods of which classes are brought under the control of the metaobject. It specifies that any invocation of `write` method of any subclass of `FileWriter` is to be trapped and handled by the metaobject `FileEnforcementMetaobject`. In this way we can ensure that no checks are accidentally omitted from the source code because of a software maintenance oversight.

Unlike the previous example we trap invocations made by an object rather than the execution of a particular method of an object. This is because *Kava* cannot rewrite system classes without the use of a custom JVM and so we trap calls made to the controlled object rather than modify the implementation of the object itself.

5 Related Work

The principle of separating security policy and dynamically enforcing security on applications is not new. In this section we discuss and evaluate four approaches to implementing this principle.

5.1 Applet Watch-Dog

Applet Watch-Dog [6] exploits the ability of the execution environment to control code execution. Here the threads spawned by applets are monitored and controlled in order to protect hosts from denial of service attacks. It is a portable approach that requires no changes to the Java platform in order to work. When applets are loaded in conjunction with the *Applet Watch-Dog* their use of memory, priority of threads, CPU usage and other resources is monitored and displayed in a window. The user can choose to stop or suspend threads as required. A security policy for resource usage can also be specified so that a thread is automatically stopped if it exceeds the prescribed maximum usage of a resource.

The *Applet Watch-Dog* approach can prevent a large class of denial-of-service attacks. However, it cannot prevent other attacks such as privacy attacks. The example given by the authors is that it cannot prevent an applet from forging mail as this would require monitoring port usage. The scope of policies enforceable by a Watch-Dog is obviously limited by the scope of control the execution environment has over code execution. For example, if the capability to monitor ports does not exist then attacks exploiting port access cannot be controlled. Another problem is that specifying a new type of security policy requires the rewriting of the *Applet Watch-Dog*.

5.2 Generic Wrappers

Generic wrappers use wrappers to bring components under the control of a security policy. The wrappers act as localised reference monitors for the wrapped components. A well developed example of this approach is found in [7]. Here the emphasis is on binary components and their interaction with an operating system via system calls. Wrappers are defined using a Wrapper Definition Language (WDL) and are instantiated as components are activated. The wrappers monitor and modify the interactions between the components and the operating system. Generic policies for access control, auditing, intrusion detection can be specified using the WDL.

The use of *generic wrappers* and a wrapper definition language is an attractive approach as it is flexible and is generalisable to many platforms. However, there are some drawbacks. Wrappers can only control flows across component interfaces and cannot control internal operations such as access to state or flows across outgoing interfaces. Also the wrappers are not at the right level of abstraction. The level of abstraction is at a lower level than the application level. This makes it difficult to specify security policies that control both access to system resources and application resources.

5.3 SASI - Security Automata SFI Implementation

SASI [4] uses a security automaton to specify security policies and enforces policies through software fault-isolation techniques. The security automaton acts as a reference monitor for code. A security automaton consists of a set of states, an input alphabet, and a transition relationship. In relation to a particular system the events that the reference monitor controls are represented by the alphabet, and the transition relationship encodes the security policy enforced by the reference monitor.

The security automaton is merged into application code by a *rewriter*. It adds code that implements the automaton directly before each instruction. The rewriter is language specific (the authors have produced one for x86 machine code, and one for Java bytecode). Partial evaluation techniques are used to remove unnecessary checks.

The current system does not have any means for maintaining security related state which makes some application level security policies difficult to express. The authors propose extending *SASI* to include the ability to maintain typed state.

One of the problems the authors found when applying *SASI* to x86 machine code was the lack of high level abstractions. For example, the lack of a concept of *function* or *function calls* meant that the *SASI* rewriter had to be extended to include an *event synthesizer*.

SASI is very powerful and can place controls on low level operation such as `push` and `pop` allowing rich security policies to be described. However, the security policy language is very low level with the events being used to construct the policies almost at the individual machine language instruction level. The Java implementation was at a slightly higher level, mainly because the Java machine code is a high level machine code for an object oriented machine, but still the policies were quite low level. The authors plan to investigate a Java implementation that exposes more high level abstractions and make use of high level security policies. We would argue that reflection provides an appropriate model for solving this problem.

5.4 Naccio - Flexible Policy-Directed Code Safety

Naccio [5] allows the expression of safety policies in a platform-independent way using a specialised language and applies these policies by transforming program code. A *policy generator* takes resource descriptions, safety policies, platform interface and the application to be transformed and generates a policy description file. This file is used by an *application transformer* to make the necessary changes to the application. The application transformer replaces system calls in the application to calls to a policy-enforcing library. *Naccio* has been implemented both for Win32 and Java.

Naccio relies on wrapping methods, the original method is renamed and a wrapper method with the same name added. The wrapper method delegates the actual work to the renamed method but can perform policy checking before and after the call to the renamed method.

Naccio provides a high level way of specifying application security that is platform-independent but it is limited in what can be controlled. For example, *Naccio* cannot specify a safety policy that prevents access to a particular field of an object by other objects. Also because *Naccio* relies on renaming of methods there is the possibility that the enforcement mechanisms could be bypassed.

5.5 Evaluation

The *Applet Watch-Dog* approach makes good use of existing capabilities in the execution environment to prevent denial-of-service attacks. However, it is limited in the scope of security policies it can support because it relies upon the capabilities already present in the execution environment. It also is difficult to specify new types of security policy as this requires the rewriting of the *Applet Watch-Dog*.

Generic wrappers, *SASI* and *Naccio* provide greater control over code execution and more flexible policy specification. *SASI* and *Naccio* extend earlier work that used code rewriting for security enforcement that was more ad hoc in nature and focused on specific classes of security policy. For example, Java bytecode rewriting has been used to implement fine grained access control [13], and resource monitoring and control policies [2].

However, there are problems with the level of abstraction and expressiveness of these approaches.

Generic wrappers work at a low level of abstraction, essentially the level of the operating system. This limits them to enforcing security policies that control access to system resources. Although it is possible that a number of application level security policies could be expressed, the lack of high level abstractions makes this task difficult.

SASI operates at the level of machine code which provides it with a lot of power. However, it has difficulties when dealing with application level abstractions where the operations that need to be intercepted are related to the object-oriented computational model. With the Java version there is the concept of higher level operations because the Java virtual machine bytecode explicitly uses object-oriented concepts. A higher level approach would be to base the security policy automata primitives on an abstract model of object oriented computation. This could be mapped to required behavioural changes which would then be realized in a platform dependent way.

To some extent *Naccio* supports application level abstractions. However, it lacks a rich model for expressing the program transformations. If it had a model based on behavioural change then it could specify richer policies but still in a platform independent way.

In our opinion the metaobject protocol [11] approach provides a good basis for the implementation of security policies. It provides both a high level, abstract model of the application but also a principled way to describe and implement changes to the behaviour of the application. The approaches discussed here implement the security policies at too low a level. Instead of implementing traps for individual machine code instructions or system calls the better approach

is to work at the level of the object oriented computational model. For example, instead of trapping Java `invokevirtual` instructions and adding security enforcement mechanisms at this level, the metaobject approach would trap invocations sent from an object and specify before and after behaviour that invoked required security mechanisms. The actual mapping to code rewriting would be handled by the metaobject protocol allowing the security policy developer to work at a high level. This is the approach that we are taking with our system *Kava*.

6 Conclusions and Further Work

Using *Kava* to implement security mechanisms in Java allows security policy to be developed separately from application code and then be combined at loadtime. This makes it ideal for flexible security for securing mobile code where the policies that the code must obey are defined by the host and the code is delivered in a compiled form.

As we have shown *Kava* can be integrated with the current Java security model and uses high level abstractions in order to specify policy. The difference between using the standard Java security model and using *Kava* is that the permissions checking takes place in metaobjects that are separate from the application objects. The metaobjects are only bound at loadtime allowing security policy to be changed independently of the application code.

Due to the use of bytecode rewriting *Kava* achieves a strong degree of non-bypassability than other systems proposed. This is important for making the case that the metaobject can act as a non-bypassable reference monitor for the associated object.

The *Kava* metaobject protocol allows control over more aspects of the behaviour than a system such as *Naccio*, *generic wrappers*, or the *Applet Watch-Dog* and at the same time provides higher level abstractions than a system such as *SASI*.

A direction for future work is the development of general policy frameworks for use with *Kava*. Currently, as shown in the examples, the security policy is developed manually. This is a useful feature in some situations but ideally there should be policy frameworks available that free the developer from having to develop their own set of permissions and metaobjects . We have proposed elsewhere some frameworks for implementing the Clark-Wilson security model [17] and a Resource Management security model [18]. We are currently integrating this work with *Kava* to provide high level support for application security.

Acknowledgements. This work has been supported by the UK Defence Evaluation Research Agency, grant number CSM/547/UA and also the ESPIRIT LTR project MAFTIA.

References

[1] Cohen, G. A., and Chase, J. S. : Automatic Program Transformation with JOIE. Proceedings of USENIX Annual Technical Symposium 1998

[2] Czajkowsik, G., von Eicken, T., JRes: A Resource Accounting Interface for Java, ACM OOPSLA Conference, October 1998.

[3] Dahm, M. : Bytecode Engineering, Java Informations Tage 1999

[4] Erlingsson, U., Schneider, F. : SASI Enforcement of Security Policies: A Retrospective. Proceedings New Security Paradigms Workshop, 1999

[5] Evans, D., Twyman, A. : Flexible Policy-Directed Code Safety. IEEE Security and Privacy, Oakland, CA., May 9-12, 1999

[6] Florio, M.F., Gorrieri, R., Marchetti, G. : Coping with Denial of Service due to Malicious Java Applets. Computer Communications Journal, August 2000

[7] Fraser, T., Badger, L., Feldman, M. : Hardening COTS Software with Generic Software Wrappers. IEEE Security and Privacy, Oakland, CA., May 9-12, 1999

[8] Gong, L. : Inside Java(TM) 2 Platform Security. Addison-Wesley, 1999

[9] Gosling, J., Frank Yellin, and the Java Team, "Java API Documentation Version 1.0.2", Sun Microsystems, Inc., 1996

[10] Gosling, J., Joy, B., Steele, G. L. : The Java Language Specification, The Java Series, Addison-Wesley, 1996

[11] Kiczales G., des Rivieres J. : The Art of the Metaobject Protocol. MIT Press, 1991.

[12] Maes, P. : Concepts and experiments in computational reflection, OOPSLA, 1987

[13] Pandey, R., Hashii, B., Providing Fine-Grained Access Control for mobile programs through binary editing, Technical Report TR98-08, University of California, Davis, August 1998

[14] Java Team, JDK 1.1.8 Documentation", Sun Microsystems, Inc., 1996-1999

[15] Java Team, Java 2 SDK Documentation", Sun Microsystems, Inc., 1996-1999

[16] Java Security Team, "Java Authentication and Authorization Service", Sun Microsystems, Inc., http://java.sun.com/security/jaas/index.html, 1999

[17] Welch, I. : Reflective Enforcement of the Clark-Wilson Integrity Model, 2nd Workshop on Distributed Object Security, OOPSLA, 1999.

[18] Welch, I., Stroud, R. J. : Supporting Real World Security Models in Java. Proceedings of 7th IEEE International Workshop on Future Treads of Distributed Computing Systems, Cape Town, South Africa, December 20-22, 1999

[19] Welch, I., Stroud, R. J. : Kava : A Reflective Java based on Bytecode Rewriting. Springer-Verlag Lecture Notes in Computer Science LNCS 1826, 2000