Re-engineering Security as a Crosscutting Concern

IAN S. WELCH[1] AND ROBERT J. STROUD[2]

[1]*School of Mathematical and Computing Sciences, Victoria University of Wellington, New Zealand*
[2]*Centre for Software Reliability, School of Computing Science, University of Newcastle upon Tyne, UK*
Email: ian.welch@vuw.ac.nz

We have re-engineered a third-party application using a reflective security architecture that allows security to be treated as a crosscutting concern. This has resulted in a considerable reduction in tangling between application code and security code. Prior to the re-engineering, the application was secured using a conventional approach based upon the application of inheritance and the proxy pattern, and we are thus able to compare both approaches. Our experience highlights some general points that are applicable to any attempt to engineer security using advanced separation of concerns technology and some possible improvements to Kava, used to implement the crosscutting concerns.

Received 29 April 2002; revised 28 November 2002

1. INTRODUCTION

Current approaches to the development of security architectures for applications result in software that has application code tangled with security code. This tangling makes it difficult to change a security architecture once the software has been deployed. It also mitigates against reuse of security functionality as such functionality is tightly bound to the application it is securing. Advanced Separation of Concerns (ASoC) techniques such as reflection [1, 2], multidimensional separation of concerns [3], composition filters [4] and aspect-oriented software development [5] promise a clean separation of concerns between functional and non-functional code by being able to treat non-functional code as crosscutting functional code. Applying this to security code and treating it as a crosscutting concern promises increased maintainability of secure applications, the ability to change the security architecture independently of application code and increased security policy reusability.

This paper focuses on the application of reflection to re-engineering an existing security architecture for a reasonably complex third-party application that makes use of weak object mobility and distributed objects. Our third-party application, which we refer to as System K,[3] was developed as a security demonstrator that had the objective of illustrating a security architecture using a distributed, heterogeneous, hypertext document server written in Java. Inheritance and the proxy pattern were used to provide a degree of separation of concerns between application code and security code. However, a number of problems that arose from the use of these techniques can be avoided through the use of reflection. This paper describes these problems and how we avoided them by using reflection to re-implement the security architecture. In particular, using reflection to re-implement the security architecture provided two main advantages over the original architecture. First, unlike the existing implementation of the security architecture we can secure applications provided as compiled code as we no longer rely upon access to the source code in order to establish an inheritance relationship. Second, since our implementation technology does not rely upon the proxy design pattern, we implement a stronger encapsulation of objects for both distributed objects and mobile objects, and we remove the need for extra code to maintain the association between proxies and proxied objects.

This paper builds upon our own previous work [6, 7] and other authors' work on reflective security [8, 9, 10, 11, 12, 13, 14], aspect-oriented security [15, 16] and general approaches to implementing non-functional concerns using aspects [17, 18].

The structure of the paper is as follows. In Section 2 we provide an overview of the case study; in Section 3 we provide an overview of the current design; in Section 4 we critique the current design; in Section 5 we show how we implement security as a crosscutting concern; in Section 6 we discuss the results of our work and in Section 7 we discuss related work. Finally, Section 8 concludes the paper with a brief summary that highlights some areas for future research.

2. CASE STUDY

The functional part of System K is a CASE tool implemented as a hypertext document server. As System K was designed as a demonstration application it does not implement all the functionality that might be expected of a production application. Users manipulate CASE diagrams via viewers. A viewer allows a user to access a remote database and download a model for local editing.

[3]We cannot provide full details of the application as it was provided under a confidentiality agreement.

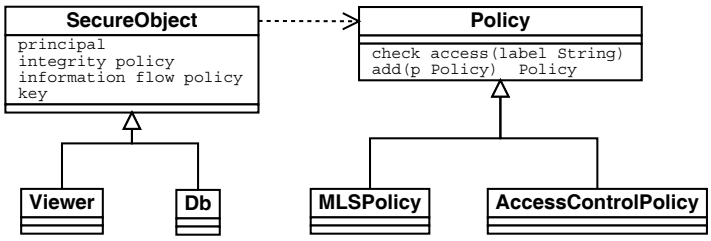

FIGURE 1. Class diagram for self-defence security architecture. Two classes under the control of the security architecture are shown: `Viewer` and `Db`. These extend `SecureObject` which maintains security related metadata. Each `SecureObject` contains pointers to an information flow policy and an integrity policy. The abstract `Policy` class defines a `check_access` method that is implemented by each type of policy. The `check_access` method is invoked to determine whether a particular action is authorized.

New elements and links to other models can be added or modified locally. Links may be made to models that exist in other databases. When a user has finished editing the model object then it can be uploaded via the viewer to the database. The functional part of System K is made up of 64 Java classes and approximately 4000 lines of code. System K is made up of multiple viewers that access multiple remote databases. The security architecture uses an authentication server to provide global authentication and multiple authorization servers to manage different security policies.

The security architecture used by the demonstrator is based on work introduced in [19]. The key idea is that each object is responsible for its own security—hence the model is termed 'self-defence'. This security architecture assumes that objects are strongly encapsulated and that all communication is via well-defined interfaces. Remote invocation requests between objects are encrypted and authenticated. Each object that provides access within the context of a security policy makes access control decisions on a per-request basis and may delegate this decision to other objects such as authorization servers. Delegation of authorization and authentication decisions to servers provides a global view of security policy, even though security decisions are made on a per-object basis. It also provides support for heterogeneous policies in the same system, as each object may either use its own policy to make an access decision or delegate the decision to an appropriate authorization server.

Each secured object in System K may be subject to both integrity and information flow security policies. The integrity policies applied to a security object are variations on access control lists. The information flow policy is based on the Bell–LaPadula multilevel security policy [20]. Essentially, the integrity policies involve simple allow or deny decisions, whereas the information flow policies involve a decision about access but also require updates to be made to the security state associated with the subjects and objects.

It is assumed that a viewer accesses a local trusted source such as a smartcard to locally authenticate the client as a principal and to retrieve the secret key that is used for communication with an authentication server. The authentication server is used to establish a shared symmetric session key between the viewer and other servers with which the viewer wishes to communicate over an encrypted channel. There are also several authorization servers, one for each policy type. This provides a global view for security and allows management of rights and clearances by security officers.

3. OVERVIEW OF THE CURRENT DESIGN

In this section we describe how the self-defence security architecture was implemented in System K using standard object-oriented techniques such as inheritance and the proxy design pattern [21]. The designers of System K chose these techniques because they offered a separation of concerns that would make it easier for future developers to change either the application or the security policies enforced by the security architecture. Although they did achieve this to a degree, the degree of separation of concerns was not as complete as would be provided by the use of ASoC techniques.

In our discussions we refer to both the static view of the architecture and the dynamic view of the architecture. The static view uses the class framework diagram shown in Figure 1, whereas the dynamic view uses the object collaboration diagram shown in Figure 2, which shows a secure invocation of a method on a remote `Db` object by a `Viewer` object.

3.1. Association of security-related state with objects

Each secure object in System K requires knowledge of the principal's identity, the principal's secret key used for authentication and establishing secure communication, and the policies that apply to that object. These features are added by ensuring that each secure object subclasses the `SecureObject` class, which stores these details in its fields.

3.2. Authorization

The proxy pattern is used to implement both authentication and authorization concerns. It localizes the implementation of these concerns into client-side and server-side proxy objects. The client-side proxy implements the transmission of principal identity with remote method invocations and the server-side proxy enforces security policies on servers.

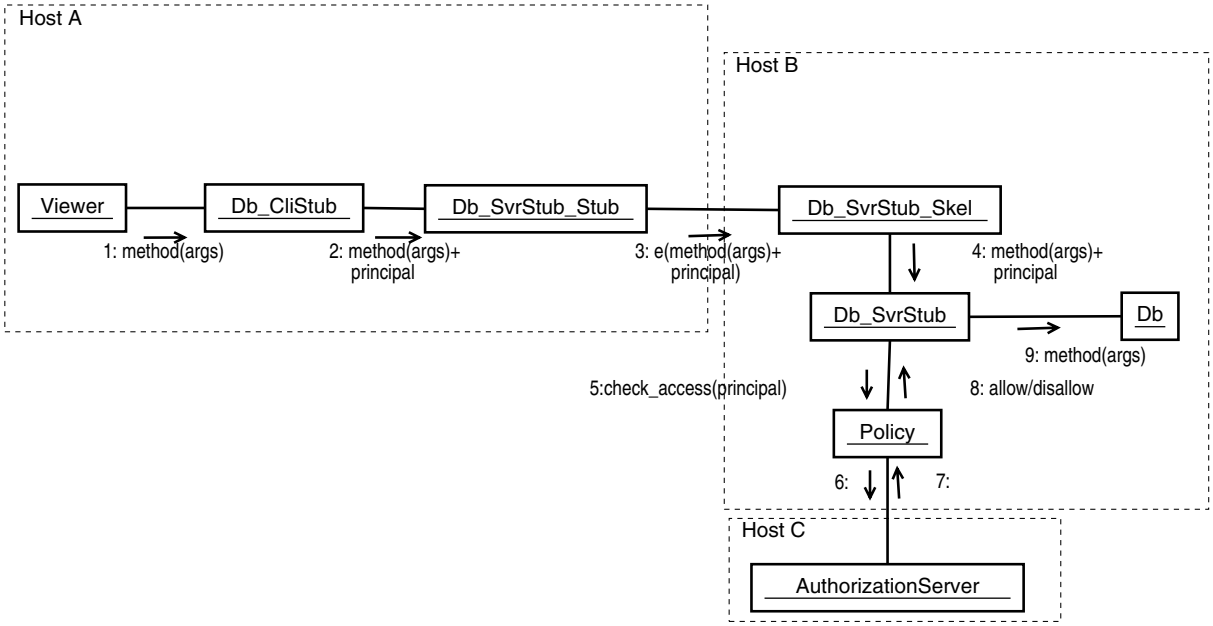

FIGURE 2. Collaboration diagram showing steps involved in a viewer invoking a method of a remote database under the self-defence security architecture.

The original implementation of System K required a proxy to be generated manually for each class that could be the target of an authorization check. The authors did suggest that a better approach would be to generate the proxy automatically so as to reduce the possibility of error when coding the proxies, but this approach was never implemented.

The normal flow of control for a secure policy-controlled invocation is shown in Figure 2. The `Viewer` object invokes a method on a remote `Db` object by invoking the method on a local client-side proxy (`Db_CliStub`). This proxy adds the principal identifier to the invocation and then invokes the method on the server side proxy (`Db_SvrStub`) using a Java Remote Method Invocation (RMI) stub. The RMI stub (`Db_SvrStub_Stub`) marshalls the method, the arguments and the principal's identity and encrypts it using a session key[4] shared between the `Viewer` and the `Db` object, before sending the encoded method call to the remote Java RMI skeleton for the server-side proxy (`Db_SvrStub_Skel`). The RMI skeleton decrypts the bytestream and unmarshalls the method, arguments and principal. It then invokes the server-side proxy (`Db_SvrStub`), which enforces any security policies before passing the method call on to the remote `Db` object. The steps involved in checking if access is allowed depend on the types of policies being enforced on the secured object by the proxies.

There are two main types of policy: integrity policies and information flow policies. Policies are implemented as classes (see Figure 1). The abstract `Policy` class defines a `check_access` method that takes an identity name as an argument and signals access denial by throwing an exception and an `add` method that is used to combine policies to reflect changes in object labelling. Policy implementations can make access decisions either on the basis of local information or by contacting authorization servers that maintain a global view of security information. How the proxy enforces policies depends upon the policy type and whether it is applicable to the protected object. This functionality is hardcoded into the proxy itself.

In the case of an integrity security policy, `Db_SvrStub` invokes the `check_access` method of the policy object using the principal's identity. If access is denied then the policy object will raise an exception, otherwise it will return silently. On return the appropriate method of the `Db` object is invoked. In the case of an information flow security policy the steps are the same as for an integrity security policy, except that if access is allowed then the security state of the server may need to be updated. This decision is hardcoded into the `Db_SvrStub`. If an update is required because the execution of the method may result in a change to the server state, then a new information flow policy is generated by adding the existing policy to one that is created using the principal's identity. Integrity policies require only that an access check is performed, whereas information flow policies require both an access check and updating of the server's security state.

3.3. Confidential communication

System K used inheritance to implement confidential communication and limited authentication for secure objects by adding capabilities to the standard RMI framework. Objects that can be accessed remotely are required to

[4]Note that we do not show how the session key is established. This takes place when the client contacts the server for the first time using public keys that have been distributed to both the client and server.

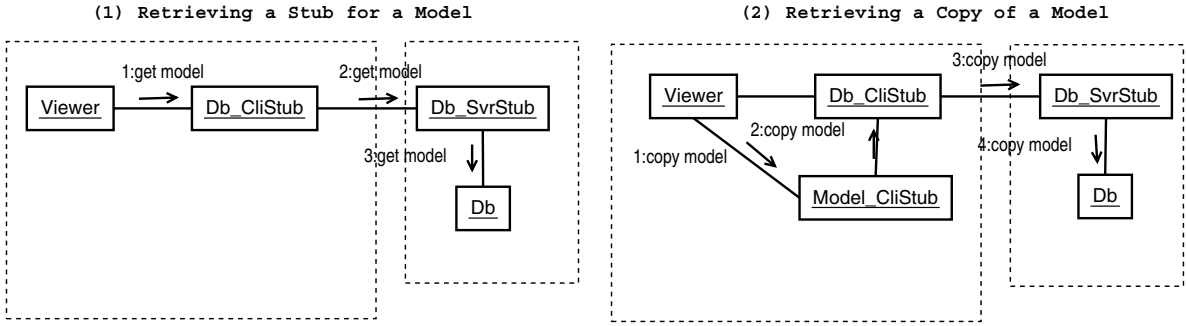

FIGURE 3. Steps involved in retrieving a `Model`. (1) A local stub for the `Model` is retrieved from the remote `Db`. (2) The local stub retrieves a copy of the `Model` for editing.

extend a modified `UnicastRemoteObject` class that invokes other classes that perform session key generation as part of client and server authentication, and ensures that the underlying sockets transparently encrypt and decrypt communications. Had the developers of System K been developing with the latest version of the Sun JDK, they could have achieved the same effect by using a customised socket factory to implement secure RMI, but this option did not exist at the time.

4. CRITIQUE OF THE CURRENT DESIGN

Inheritance and the proxy pattern have their drawbacks in terms of flexibility and maintainability. In this section we discuss them in the context of the design of System K.

4.1. Inheritance

Inheritance is used to associate security states with objects and to provide a secure implementation of RMI. Developers of new classes that must be secured must subclass the `SecureObject` class. This should provide no problems when implementing classes from scratch. However, if an existing class is to be re-engineered then the programmer requires access to the source code in order to make the target class subclass `SecureObject`. This means that classes supplied only as compiled code cannot be re-engineered to be secure using an approach based on inheritance, although one way to address this would be to use wrappers and delegation.

Another problem is that the decision as to which policies to associate with an object is hardwired into the implementation of the `SecureObject` class. It would be more elegant if the details of the policies that apply to an object were separated out into some form of configuration file. An example of this problem is shown in the difference between securing a `Db` class and a `Model` class. System K implements both integrity and information flow security policies for `Models`, but only an integrity security policy for `Dbs`. However, both classes inherit both policies as part of their metadata. If an external configuration file was used then it would be possible to specify different policies for each that could be adjusted at loadtime without having to change any source code.

Another problem with the use of inheritance arises when engineering secure RMI functionality by changing the system class `UnicastRemoteObject`. By changing this class the client and server classes require no changes at the source code level as they already subclass `UnicastRemoteObject`. However, this is an inflexible solution as it means that any RMI application must now use secure RMI. Also, access to the Java runtime classes is required in order to perform this replacement, which may not be possible in all environments for good security reasons.

4.2. Proxy pattern

The advantage of using the proxy pattern is that no changes are required to a class so that the principal's identity can be transparently added and removed from method invocations and authorization checks can be applied before method invocations are executed. However, the following disadvantages stem from the introduction of the extra proxy class into the application: a requirement for 'logical wrapping', the confinement problem; and an interface gap. We discuss each of these problems below.

Using proxies to add functionality to existing classes requires 'logical wrapping' [22] to ensure that the proxy and proxied instances are always associated with each other. Any class that returns an instance of the proxied class must be modified (or itself proxied) to ensure that it returns a proxy instance. This requires programmers to introduce logic into the application in addition to the definition of the proxy class itself. In the implementation of System K we can see this when considering how access to a mobile object such as the `Model` object is achieved. The security architecture depends upon a proxy mediating all access to a secured object, therefore when a mobile object is downloaded, a proxy must also be downloaded with the object to ensure that all accesses to the mobile object continue to be mediated. The implementation in System K is shown in Figure 3. As shown, instead of requesting a copy of a `Model` object directly from a `Db` instance, a proxy for the remote object must first be downloaded, and this proxy is then responsible for downloading the `Model` object. Other implementations are possible, for example, the proxy could be automatically generated at the client side. However, there must always

be additional processing in order to maintain the association between the proxy and the proxied class.

The second problem is related to the first. In a reasonably complex program, it is always possible that an instance of a proxied class returned by a method invocation might not be replaced with an instance of its proxy. Such an unwrapped instance would then allow the proxied instance to be invoked directly, bypassing the proxy. This is simply a variant of the confinement problem [23] and is a possibility that always exists whenever a proxy is used.

The third problem is the 'interface gap' that exists because of the introduction of an extra proxy class that must maintain the same interface as the proxied class. Whenever the interface of the proxied class changes, then the interface to the proxy class must also be updated. The non-reflective version of System K required that this be done manually, which introduced the possibility of missing changes to the interface of the proxied class.

5. SEPARATING OUT THE CONCERNS

There are three security issues that need to be addressed.

- Authorization. How are security policies enforced?
- Confidentiality. How do clients communicate confidentially with remote servers?
- Authentication. How do clients associate their identity with invocations?

Implementing these mechanisms involves adding security state and making behavioural changes throughout the application—thus, these security issues are genuine crosscutting concerns. Furthermore, these concerns are not restricted to this particular application, they crosscut all secure applications. Therefore, an ideal implementation of these concerns should be reusable for other applications. However, as we have seen, implementing these concerns using conventional object-oriented techniques such as inheritance and the proxy pattern leads to a number of problems that are summarized below.

- Use of inheritance for associated security state with application objects requires access to the source code of the application object.
- Inheritance forces all subclasses to inherit the same state or behaviour irrespective of whether it is relevant to them or not. In the particular instance of System K, we can see that all subclasses of `SecureObject` inherit state associated with both integrity and information flow policies irrespective of whether a particular object requires them. Also, changing the system class `UnicastRemoteObject` forces all RMI to be confidential irrespective of whether a particular message should be confidential or not.
- Proxies require the 'logical wrapping' code in order to maintain the association between proxies and proxied classes. This has an impact upon the implementation of security for mobile code, for example the `Model` class. To maintain the 'logical wrapping', extra steps must be taken to ensure that the proxy is always downloaded with the proxied class.
- Because there is a separation between proxies and proxied classes, there is always the possibility that a reference to a proxied instance might be exposed, making it possible to bypass the checks performed by the proxy.
- The separation between proxy and proxied class creates an additional maintenance burden as changes to the interface of the proxied class must always be reflected in changes to the proxy class.

We have used reflection to address these problems. In particular, meta-object protocols allow incremental and local changes that avoid the problems associated with inheritance, and the use of reflection removes the need for a proxy. In the rest of the paper, we show how we used reflection to re-implement the existing security architecture as a set of crosscutting concerns.

In Section 5.1 we introduce Kava, which is the implementation of reflective Java that we used for our experiments. We then describe in Section 5.2 how we modularized the crosscutting concerns described above into meta-objects and then show in Section 5.3 how we compose these concerns together at runtime to provide a secure system.

5.1. Kava

Kava [24] is used as a platform for our experiments with security and reflection. The Kava meta-object protocol provides control over the semantics of method execution, the invocation of methods upon other objects, field access, exception raising and object instantiation. Object behaviour can be adjusted on a per-instance basis. The binding between objects and meta-objects is contained in a binding specification file that uses a declarative syntax. In order to allow parameterization of meta-objects, we have extended the binding specification to allow meta-parameters to be associated with object behaviours that are brought under the control of meta-objects.

Kava implements different crosscutting concerns as meta-objects that are bound to base-level objects according to a binding specification. As Kava implements behavioural reflection at loadtime this binding can occur as the base-level application is loaded into the Java Virtual Machine. Expressed in terms of aspect-oriented programming concepts [5], the binding language is a weaving language expressing pointcuts and the meta-objects are separately compilable advice aspects that can be applied to compiled code.

An overview of the Kava architecture is shown in Figure 4. An application-level class loader is used to intercept class loading and pass the class to Kava. Kava uses a bytecode rewriting toolkit (BCEL [25]) to embed hooks into the bytecode for the compiled application classes. At runtime, these hooks switch control from the base level to the metal level. Only those classes specified in the binding specification have these hooks inserted. This means that the

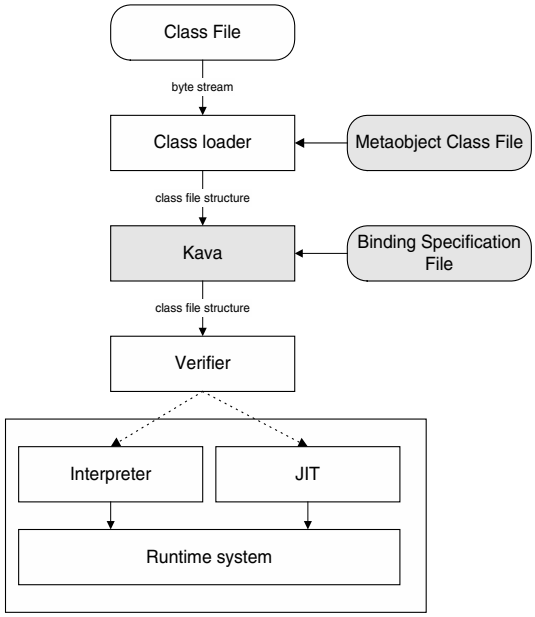

FIGURE 4. Overview of Kava. Classes are loaded by an application-level class loader. The class file structure is then passed to Kava for rewriting. After hooks have been added the class is verified and loaded as normal.

```
public interface IMetaObject {
  public void beforeExecuteMethod(
    IExecuteContext context);
  public void afterExecuteMethod(
    IExecuteContext context);
  public void beforeInvokeMethod(
    IInvokeContext context);
  public void afterInvokeMethod(
    IInvokeContext context);
  ...
}
```

FIGURE 5. Excerpt of the meta-object interface. There is a 'before' and 'after' method for each language behaviour such as method execution or invocation. When the behaviour is intercepted and handled at the meta level then the context is reified. There is a different type of context for each reifiable behaviour.

cost of a switch to the meta level is only paid where it is absolutely necessary.

Kava provides programmers with the ability to implement changes to class behaviour through bytecode rewriting without having a knowledge of class file structure or bytecode programming. It does not provide all the capabilities of bytecode rewriting such as changing the superclass or adding a completely new method but it does allow type safe changes to the semantics of an existing class.

All Kava meta-object classes subclass the `MetaObject` class which implements the interface shown in Figure 5. Like the CLOS meta-object protocol [2], there is a 'before' meta-method and an 'after' meta-method for every base-level operation that can have its semantics changed. When these methods are invoked they are passed a context parameter that encapsulates the base-level context as it was when the switch to the meta level took place. For example, a context of type `IExecuteContext` would encapsulate a reified representation of the method about to be executed, the arguments passed to that method and any meta-parameter associated with the method in the binding specification.

A benefit of using bytecode rewriting is that Kava is highly portable across JDKs and does not need access to source code. This distinguishes it from a number of other reflective Java implementations that rely on extensions to the JRE or require access to source code [26, 27, 28, 29, 30]. Furthermore, whereas most reflective systems are implemented internally using proxies, Kava's use of bytecode rewriting techniques ensures that the interception of operations by meta-objects is non-bypassable, which makes it easier to argue that the overall application is secure. We refer readers to [6] for a discussion of how we achieve non-bypassability and our trust model.

5.2. Modularizing the crosscutting concerns

In this section we introduce the meta architecture and discuss how the various meta-object classes we introduced address the different concerns in a modular manner. Figure 6 shows the meta-object classes and the application classes that they are bound to at loadtime. The `MetaAuthorization` meta-object implements the authorization concern by enforcing security policy checks on operations, the `MetaConfidentiality` meta-object implements the confidentiality concern by ensuring that the Java RMI implementation uses the Secure Socket Layer (SSL), and the `MetaAuthentication` meta-object implements the authentication concern by ensuring that the principal's identity is associated with each method invocation.

5.2.1. Authorization

The `MetaAuthorization` meta-object maintains pointers to the information and integrity policies associated with a server. The pointers are to the current information and integrity policies governing the server's participation in invocations. The initial policies and their initial values are specified as parameters for the meta-object constructor and are hard-coded in the binding specification. The meta-object controls access to server methods and applies the information and integrity security policies as appropriate. As methods differ in the security policies they apply, the binding specification file parameterizes the meta-object protocol and indicates whether the method is under the control of an information flow or integrity security policy. In addition, in the case of an information flow policy, it indicates whether the execution of the method could result in a change to the security state of the server. This is necessary for security policies such as Bell–LaPadula, where the security state of the server must be updated to reflect the interaction with a principal. Note that this approach to authorization relies upon a correctly-specified binding specification as it indicates which methods are controlled by which security policies.

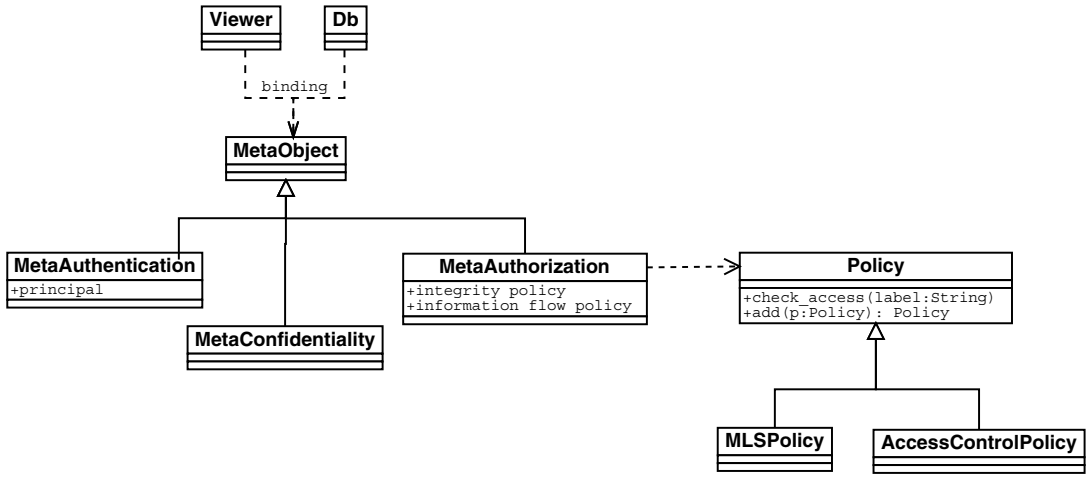

FIGURE 6. Class diagram for reflective security architecture.

Using a meta-object to perform authorization avoids the problems associated with using a separate proxy class. As there is no proxy and no reference to be leaked, we can provide direct access to the protected server without the need for a security manager object. Also, in the case of the `Model` class, there is the benefit that there is no need to add extra steps that maintain the presence of a proxy even when the `Model` object is downloaded for local editing. This is because the meta-objects are automatically downloaded with the base-level object because they are pointed to by the base-level object and defined to be serializable.

5.2.2. *Confidentiality*

The `MetaConfidentiality` meta-object ensures confidential communications between clients and servers. As we are implementing our security architecture using a more recent version of the JDK than the developers of System K, we can take advantage of support for secure RMI. This allows SSL sockets to be used for communication between client and server. The `MetaConfidentiality` meta-object is bound to the RMI startup code for the remote object on the server, and intercepts the `Naming.rebind` method to ensure that a secure socket factory is registered for the remote object. This ensures that SSL sockets are used for both client- and server-side connections. As a side effect of using SSL, clients and servers are authenticated before confidential communication takes place. Note that the role of the `MetaConfidentiality` meta-object is simply to intervene in the initialization of the RMI layer and the meta-object does not otherwise intercept method invocations between application objects.

5.2.3. *Authentication*

The `MetaAuthentication` meta-object implements the passing of a principal's identity[5] from the client to the server with a remote invocation. The meta-object is designed to be bound to the server's RMI stub and skeleton because being bound to these objects exposes interfaces for marshalling and unmarshalling arguments that are passed with an RMI method invocation. At the client side, the `MetaAuthentication` meta-object adds the principal to the bytestream representing the marshalled arguments before it is sent over the encrypted socket connection to the skeleton; at the server side, it removes the principal from the bytestream before unmarshalling the arguments from the decrypted bytestream. It does not appear to be possible to customize the RMI implementation and achieve the same effect by using socket factories, because only bytestreams and not individual method calls are visible at the socket level. Furthermore, our implementation is forced to use JDK1.1 compliant proxies because JDK1.2+ proxies do not provide the opportunity to intercede in the marshalling and unmarshalling of arguments and results. This makes our implementation technique dependent upon current features of Java; a better approach would be to use a generic interface that provided access to representations of remote method invocations as bytestreams. However, this abstraction is not provided by the current version of Java and we must therefore rely upon particular characteristics of the RMI implementation (although we could insulate our meta-objects from these dependencies by hiding the implementation details behind such an interface).

[5]The principal's identity is read from the subject's smartcard.

5.3. Composing the crosscutting concerns

In this section we describe how the meta-objects described in the preceding section are composed to address the separated concerns in a composite manner. Figure 7 shows how the meta-objects are composed and co-operate to provide confidential RMI method invocation between a `Viewer` object (the client) and a `Db` object (the server) under the control of information and integrity security policies.

The `MetaAuthorization` meta-object is bound to methods of classes that must be brought under the control of a security policy. The `MetaConfidentiality` meta-object is bound to any class that registers a server object with the RMI registry. The `MetaAuthentication` meta-object is bound to the marshalling and unmarshalling

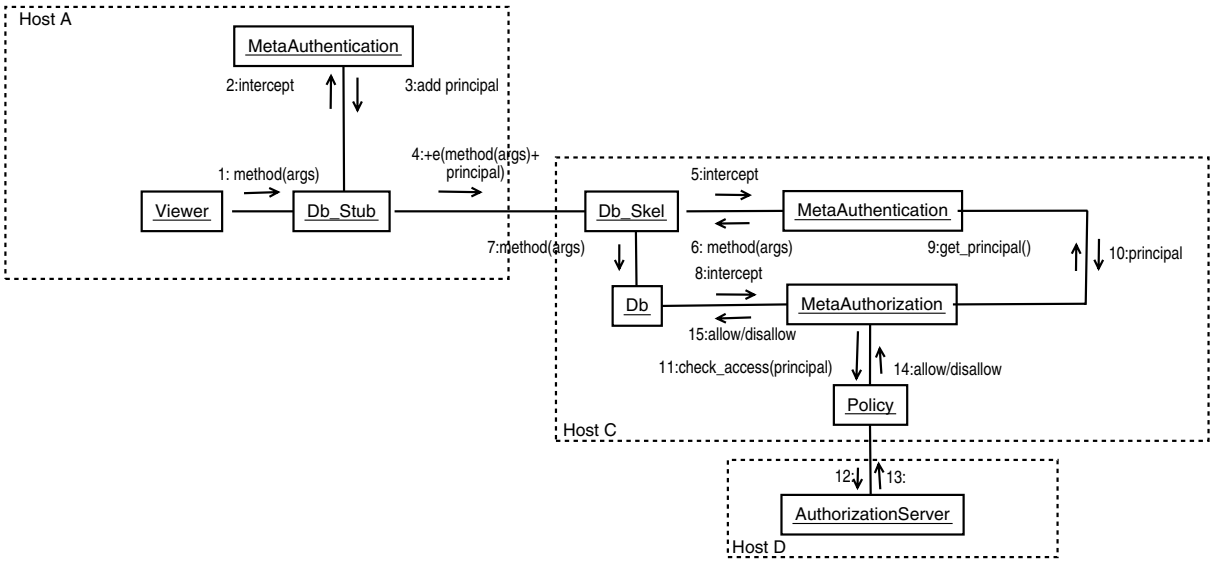

FIGURE 7. Collaboration diagram showing the steps involved in invoking a method of a remote database under the reflective security architecture.

methods of the stubs and skeletons that implement RMI for server classes.

In Figure 6 we illustrate our approach for two of the application classes, the `Viewer` and `Db` classes which are respectively client and server. Since their interactions are governed by a security policy, they are shown as being bound to `MetaObject` instances that implement different security concerns. The particular meta-object that they are bound to at loadtime is determined by the binding specification.

Under the reflective security architecture an RMI method invocation goes through the following steps (see Figure 7).

On the client side, the `Viewer` object invokes a method on the local stub (`Db_Stub`) that represents the remote `Db` object. Before the `Db_Stub` marshalls the arguments, control is switched to the `MetaAuthentication` meta-object which first marshalls the principal's identity. Control is returned to the `Db_Stub`, which then sends the bytestream to the remote skeleton (`Db_Skel`) across an encrypted socket connection. Note that secure communications are established as a result of the `MetaConfidentiality` meta-object's intervention during the startup of the remote `Db` object and the meta-object is not otherwise involved in the communication between the client and server objects.

On the server side, the `Db_Skel` unmarshalls the arguments and invokes the appropriate `Db` method. The `MetaAuthentication` meta-object takes control and unmarshalls the principal's identity which it stores. Control is returned to the `Db_Skel` and the appropriate `Db` method is invoked using the unmarshalled arguments.

If the method is specified as being under reflective control in the binding specification then its execution is reified and handled at the metalevel by the `MetaAuthorization` meta-object.

If the execution of the method has been specified in the binding specification as being under the control of an integrity policy then the `MetaAuthorization` meta-object invokes the `check_access` method of the policy object. This may cause it to contact an authorization server. If access is denied then a runtime exception is raised and this is converted to a `RemoteException` and propagated all the way back to the `Viewer` object. Otherwise, the `MetaAuthorization` meta-object silently returns, which implies that execution is permitted. Control returns to the base level and the method is executed.

If the execution of the method has been specified as being under the control of an information flow policy then the `MetaAuthorization` object performs the same steps as above for checking if access is allowed or not. However, if access is allowed then the information flow policy associated with the server is updated by invoking the policy `add` method.

Note that both an integrity security and information flow policy may apply. In this case the integrity security policy is applied first, followed by the information flow policy.

6. DISCUSSION

In this section we critique the conventional methods applied within System K for adding security, provide some analysis of our approach to re-engineering System K to use a reflective security architecture and summarize some of our findings.

6.1. Critique of conventional methods

Although the Java security model provides a number of mechanisms for security enforcement, these suffer from a variety of drawbacks and could not have been used to provide security enforcement for System K.

The Java `Permissions` framework requires changes to the application code. Another drawback is that it is class-based not object-based, so information flow policies that are

object-based cannot be used. Defining customized security managers is not straightforward and in general is only worthwhile for system resources. The use of `Guard` classes also involves restructuring the application. Guards provide simple wrappers that suffer from the same problem as the proxies used in the self-defence architecture. Even worse they require changes to the clients that use the objects protected by the `Guard` classes. The biggest drawback is the lack of integration with RMI.

The self-defence architecture was an attempt to provide an extensible architecture for Java but it too had drawbacks and was grounded in conventional techniques. The main problem was the use of proxies which require additional application-level artifacts and introduce a number of well-known problems such as bypassability, and require updating when the protected object changes. Using reflection avoids these problems as a reflective approach removes the need for a proxy and any changes can be specified in a declarative manner.

6.2. Analysis

As a result of our re-engineering of System K to use reflection, we achieved a reduction in code size for application classes of between approximately 15% and 45% through the removal of tangled code and unnecessary classes. In order to reduce the impact of programmer style we normalized both the original and re-engineered versions of the System K codebase. Normalizing the code involved removing all whitespace and comments from the code. We then compared the size of the original and re-engineered classes. However, we did not compare the size of the supporting framework for the security architecture, because this is common to both versions. We found that the main types of reductions were due to the removal of explicit access checks, removal of proxies and removal of code to update metadata.

When implementing the security architecture using reflection, we manually identified security related classes and methods that required access checks and encoded this information in the binding specification. The problem with this approach is that we could easily miss methods. This could be avoided if we had a more abstract/expressive language for specifying bindings.

We found that the nature and abstraction level of security policies that can be enforced reflectively is governed by the capabilities of the reflective language and the granularity of interfaces offered by the application and system classes. For example, Kava cannot make Java system classes reflective, although it can intercept calls to Java system classes from application code. This has had an impact on our design. Ideally, for implementing the confidentiality and authentication concerns, we would have liked to have had access to bytestream representation of invocations. However, although we could apply meta-objects to the RMI stubs and skeletons which allowed us to intercept marshalling and unmarshalling, we could not gain access to bytestream representations of the marshalled invocations. This prevented us from implementing our own encryption/decryption and authentication algorithms and led us to rely upon SSL socket support. We could have had the required access if we could reflect upon system classes that provide access to bytestream representations of RMI invocations. A related problem with respect to our implementation of authentication was that we had to generate JDK1.1 compatible proxies to get access to object marshalling and unmarshalling. This may not be an option in future versions. One area of future work that could address this problem would be an abstract layer that provides access to marshalling and unmarshalling. The actual implementation of the layer would vary according to the underlying JDK platform without affecting our meta-object protocol.

Dealing with exceptions correctly is another source of complexity. When exceptions are raised at the server side they are unchecked exceptions. These are converted into `RemoteExceptions` at the metalevel and returned to the client. The client then raises them locally as unchecked exceptions. This could lead to a halt of the program rather than a simple 'access denied' exception. To avoid this problem, it is necessary to intercept raised exceptions at the client side and include application-specific exception handling at the meta level. This can be quite complex as we are adding a whole new error semantics.

We discovered an area of future work with respect to weaving. Although the situation did not arise in this application, it might be necessary, in general, to bind multiple meta-objects (or aspects) to a single application object. Although Kava, in common with many other reflective systems, only allows an object to be bound to a single meta-object, we can get round this difficulty by applying the Chain of Responsibility pattern [21]. Whenever there is a possibility that a binding specification may require two meta-objects to be associated with a single base level object then we bind a `MetaChain` meta-object class to the base level object. At runtime this meta-object will invoke a chain of meta-objects that extend the `MetaChain` class. The membership of the chain of meta-objects and the binding information specific to each is passed as a parameter to the root `MetaChain` meta-object. The problem with this approach is that it relies on parameters encoded in the binding specification to specify the object behaviours in which each meta-object in the chain of meta-objects is interested. Although this removes the need for hardcoded checks in the `MetaChain` meta-object, it still incurs an unnecessary overhead. This is because each meta-object in the chain is invoked even when a behaviour in which it has no interest is intercepted at the base level. It may ignore the behaviour but it is still invoked because our weaver intercepts the union of all behaviours that the meta-objects in the chain may need to control. This is a consequence of implementing composition at the meta level rather than in the weaver. A more efficient approach would be to modify the weaver so that it chained the meta-objects together. This would mean that meta-objects would not be invoked unnecessarily. On the other hand, embedding weaving rules into Kava might limit future changes to weaving rules.

This could be avoided by providing the ability to customize weaving behaviour (see, for example, JAC [31]).

Finally, with respect to performance, we note that there is a trade-off between the generality and portability of the interception technology used (reflection or aspects), and the efficiency of the implementation. A reflective approach involves constructing a reification of a method call at the meta level, and although techniques such as 'lazy reification' are possible, this still incurs a substantial overhead. For simple behavioural extensions (aspects), a full reification may not be necessary, but this limits the expressive power of the aspects or meta-objects. Achieving the right balance is an engineering compromise. Flexibility and extensibility inevitably incur a run-time penalty, but improvements in the underlying implementation technology are continually reducing that cost.

6.3. Summary

Successful re-engineering of an existing application for security using reflection requires a well-structured application level. Unless the objects that are to be brought under the control of security policies are properly encapsulated then it is difficult to provide enforcement as we rely upon a well-defined interface that cannot be bypassed. In the case of System K the application level was well-designed and already some crosscutting concerns were modularized but were manually woven into the application. In a sense we provided a way of automatically performing weaving of these concerns into the application.

7. RELATED WORK

In this section we consider related work by other researchers into the application of reflection, aspect-oriented programming approaches and the implementation of other non-functional concerns.

7.1. Reflective security

Several other authors have also explored implementing security using reflection. With the exception of Ancona *et al.* [12] who introduce the concept of channel reification, the general approach is to have a meta-object acting as a fine-grained reference monitor for the object it is bound to. Earlier approaches investigated implementing security using reflection with specialized languages or virtual machines [8, 9, 10, 11, 12] whereas our work has focused on providing a portable implementation of reflective security for an existing language. We have also focused on providing more fine-grained control over object behaviour and used a form of bytecode rewriting that allows easy demonstration of non-bypassability. Our earlier work [6] examined how to apply the Java security architecture in a more flexible way by using meta-objects to enforce Java security permission checks on base-level objects that are executed locally. As a demonstration we applied this to a third-party Internet Relay Chat client and showed how to enforce dynamic and static security policies through the use of reflection to implement Java security permission checks [7]. Although we achieved the aim of a clear separation of concerns, a drawback of our approach was that the meta level required the same permissions as the base level. Other researchers have proposed an approach that maintains the principle of least privilege and therefore does not require the meta level to have the same permissions as the base level [13, 14]. This is not an issue for this paper as we do not apply the standard Java security architecture.

7.2. Aspect-oriented security

Researchers have also investigated implementation of security using aspect-oriented techniques, in particular Lasagne [15] and JavaPod [16]. Both implement authorization and authentication concerns. Lasagne's case study is the implementation of security for a distributed dating application that was developed to show Lasagne's capabilities and JavaPod's case study is the implementation of security for a virtual learning assistant application that was developed by third parties but manually rewritten to use the JavaPod platform.

Both case studies only implement security for distributed applications, although the possibility of providing protection for mobile objects is discussed in the JavaPod case study but is discounted as the assumption is that the client is totally untrusted and may therefore bypass any authorization that is locally applied. In contrast, we provide security for mobile code. This is possible because our trust model assumes that there is a trusted execution environment at the client side. Given that the clients in our case study are within an organizational boundary, this approach is reasonable. In contrast, the clients considered by JavaPod were totally untrusted.

Lasagne and JavaPod provide a coarse-grained component view of applications, rather than Kava's fine-grained class view of applications, and provide the ability to specify high-level composition policies that address some of the weaving problems we have encountered. Lasagne also provides a framework for hiding the complexity of 'logical wrapping' and simulates true delegation. This allows it to avoid many of the problems associated with wrappers. Both features could be useful in a future version of Kava.

The authors of JavaPod report similar results to our own in that they achieved a good separation of concerns between security code and application code. They also pointed to problems with the 'granularity' of application interfaces and the impact of them upon the security policies that could be enforced. For example, in order to provide censored results for certain clients they had to add extra methods that would be used by these clients and use access control lists to restrict the clients from using the methods that returned uncensored data. We did not encounter this problem, possibly because System K already had interfaces of the correct granularity. Should there have been a requirement for 'censoring' then we could possibly have achieved this by attaching a meta-object to the consumer that controlled the consumer's access to the returned values.

7.3. Other non-functional concerns

There are two case studies of implementing non-functional concerns that are relevant to our work. The first is of the implementation of transactions using aspects and the second is a more general study that attempted to evaluate aspect-oriented programming.

Kienzle and Guerraoui [18] look at the problems of implementing transactional semantics using aspect-oriented programming. They make the case that transactional semantics can only be implemented using separation of concerns techniques if there is some application knowledge; otherwise, non-trivial programs will exhibit deadlock and only backward recovery can be used, which may not be appropriate for the application. They do see some value in using separation of concerns in conjunction with application knowledge to reduce the amount of tangling between code delimiting transaction boundaries and application code. However, they point out that hiding the transaction boundaries actually causes problems. For example, it is harder for application programmers to perform optimizations, leading to worst-case performance. Also because there is a decoupling between application code and transactions aspects then it is possible for changes to be made to the applications without the necessary corresponding changes being made to the transaction aspects.

There are similarities to our experience. We agree that this semantic decoupling can cause problems, although the degree of problems depends upon the power of the weaving language and how it is used. For example, if the binding specification simply lists methods that are under the control of the access control policy, then any additions to the base-level class must be reflected in changes to the binding specification. However, if the binding specification is more declarative and states that any method that updates a certain field is under the control of the access control policy, then this ensures that additions at the base level are automatically caught by the binding specification.

Walker *et al.* [17] carried out some experiments that looked at ease of change and debugging of code when aspect-oriented programming is used. They asked groups of programmers to carry out tasks involving code that dealt with synchronization and debugging concerns and observed their performance. They concluded that a narrow aspect-core interface is 'better', defining a narrow aspect-core interface as one where the scope of the effect of an aspect across a boundary is well-defined, so that the aspect can be reasoned about without extensive analysis of the source code. We feel that in some respects we have managed to design a relatively narrow aspect-core interface. For example, our meta-objects do not contain hard-coded or application-specific code as they rely upon parameters that are included in the binding specification. However, the fact that the policies at the meta level are encoded using Java means that abstract reasoning may be difficult. A domain specific policy language might be more applicable here, as it would allow encoding of policy and enforcement using a security-specific language that allows more abstract reasoning.

8. CONCLUSIONS AND FUTURE WORK

We have described our experience in re-engineering the security aspects of a third-party application using meta-objects. We achieved a significant reduction in code size due to the removal of code tangling, but this was only possible because the application was already well-structured. Our analysis of our experience highlighted some general points about implementing security as a crosscutting concern and identified some areas for future work. In particular, the relationship between weaving of aspects and binding of meta-objects deserves further exploration. For ease of use, a more abstract/declarative language for specifying security policy is required, but this would effectively be domain dependent.

ACKNOWLEDGEMENTS

This work was sponsored by United Kingdom DERA Contract CSM/547/UA and the EU Project Malicious- and Accidental-Fault Tolerance for Internet Applications (IST-1999-11583).

REFERENCES

[1] Maes, P. (1987) Concepts and experiments in computational reflection. In *Proc. OOPSLA 87, Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, Orlando, FL, October 4–8, pp. 147–155. ACM Press, New York.

[2] Kiczales, G., des Rivieres, J. and Bobrow, D. G. (1991) *The Art of the Metaobject Protocol*. MIT Press, MA.

[3] Tarr, P., Ossher, H., Harrison, W. and Sutton, J. S. M. (1999) *N* degrees of separation: multidimensional separation of concerns. In *Proc. ICSE 99, 20th Int. Conf. on Software Engineering*, Los Angeles, CA, May 16–22, pp. 107–119. IEEE Computer Society, Washington, DC.

[4] Askit, M., Bergmans, L. and Vural, S. (1992) An object-oriented language-database integration model: the composition-filters approach. In *Proc. ECOOP 92, Eur. Conf. on Object-Oriented Programming*, Utrecht, Netherlands, June 29–July 3. *Lecture Notes in Computer Science*, **615**, 372–395. Springer, Berlin.

[5] Kiczales, G., Irwin, J., Lamping, J., Loingtier, J.-M. and Lopez, C. (1997) Aspect-oriented programming. In *Proc. ECOOP 97, Eur. Conf. on Object-Oriented Programming*, Jyväskylä, Finland, June 9–13. *Lecture Notes in Computer Science*, **1241**, 220–242. Springer, Berlin.

[6] Welch, I. and Stroud, R. J. (2000) Using reflection as a mechanism for enforcing security policies in mobile code. In *Proc. ESORICS 2000, 6th Eur. Symp. on Research in Computer Security*, Toulouse, France, October 4–6. *Lecture Notes in Computer Science*, **1895**, 309–323. Springer, Berlin.

[7] Welch, I. and Stroud, R. J. (2002) Using reflection as a mechanism for enforcing security policies on compiled code. *J. Comput. Security*, **10**, 399–432.

[8] Härtig, H., Kowalski, O. and Kühnhauser, W. E. (1993) The BirliX security architecture. *J. Comput. Security*, **2**, 5–21.

[9] Benantar, M., Blakley, B. and Nadain, A. J. (1996) Approach to object security in distributed SOM. *IBM Syst. J.*, **35**, 192–203.

[10] Riechmann, T. and Hauck, F. J. (1997) Meta objects for access control: extending capability-based security. *New Security Paradigms Workshop*, Langdale, Cumbria, September 23–26, pp. 17–22. ACM Press, New York.

[11] Riechmann, T. and Hauck, F. J. (1998) Meta objects for access control: a formal model for role-based principals. *New Security Paradigms Workshop*, Charlottesville, VA, September 22–25, pp. 30–38. ACM Press, New York.

[12] Ancona, M., Cazzola, W. and Fernandez, E. B. (1999) Reflective authorization systems: possibilities, benefits and drawbacks. In Vitek, J. and Jensen, C. (eds), *Secure Internet Programming: Security Issues for Distributed and Mobile Objects*. *Lecture Notes in Computer Science*, **1606**, 35–50. Springer, Berlin.

[13] Caromel, D. and Vayssiére, J. (2001) Reflections of MOPs, components, and Java security. In *Proc. ECOOP 2001, Eur. Conf. on Object-Oriented Programming*, Budapest, Hungary, June 18–22. *Lecture Notes in Computer Science*, **2072**, 256–274. Springer, Berlin.

[14] Caromel, D., Huet, F. and Vayssiére, J. (2001) A simple security-aware MOP for Java. In *Proc. REFLECTION01, Third Int. Conf. on Metalevel Architectures and Separation of Crosscutting Concerns*, Kyoto, Japan, September 25–28. *Lecture Notes in Computer Science*, **2192**, 118–125. Springer, Berlin.

[15] Truyen, E., Vanhaute, B., Joosen, W., Verbaeten, P. and Jørgensen, B. N. (2001) Dynamic and selective combination of extensions in component-based applications. In *Proc. ICSE 01, 23rd Int. Conf. on Software Engineering*, Toronto, Ontario, May 12–19, pp. 233–242. IEEE Computer Society, Washington, DC.

[16] Bruneton, E. and Riveill, M. (2001) Experiments with JavaPod, a platform designed for the adaptation of non-functional properties. In *Proc. REFLECTION01, Third Int. Conf. on Metalevel Architectures and Separation of Crosscutting Concerns*, Kyoto, Japan, September 25–28, pp. 52–72. Springer, Berlin.

[17] Walker, R. J., Baniassad, E. L. A. and Murphy, G. C. (1999) An initial assessment of aspect-oriented programming. In *Proc. ICSE 99, 20th Int. Conf. on Software Engineering*, Los Angeles, CA, May 16–22, pp. 120–130. IEEE Computer Society, Washington, DC.

[18] Kienzle, J. and Guerraoui, R. (2002) AOP: does it make sense? The case of concurrency and failures. In *Proc. ECOOP 2002, Eur. Conf. on Object-Oriented Programming*, Málaga, Spain, June 10–14. *Lecture Notes in Computer Science*, **2374**, 37–61. Springer, Berlin.

[19] Bull, J. A., Gong, L. and Sollins, K. R. (1992) Towards security in an open systems federation. In *Proc. ESORICS 92, 2nd Eur. Symp. on Research in Computer Security*, Toulouse, France. *Lecture Notes in Computer Science*, **648**, 3–20. Springer, Berlin.

[20] Bell, D. E. and LaPadula, L. J. (1976) *Secure Computer System: Unified Exposition and Multics Interpretation*. MITRE Technical Report 2997. The MITRE Corporation, Bedford, MA.

[21] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995) *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA.

[22] Hölzle, U. (1993) Integrating independently-developed components in object-oriented languages. In *Proc. ECOOP 93, Eur. Conf. on Object-Oriented Programming*, Kaiserslautern, Germany, July 26–30. *Lecture Notes in Computer Science*, **707**, 36–56. Springer, Berlin.

[23] Lampson, B. W. (1973) A note on the confinement problem. *Commun. ACM*, **16**, 613–615.

[24] Welch, I. and Stroud, R. J. (2001) Kava—using byte-code rewriting to add behavioral reflection to Java. In *Proc. COOTS 2001, USENIX Conf. on Object-Oriented Technologies and Systems*, San Antonio, TX, January 29–February 2, pp. 119–130. USENIX, Berkeley, CA.

[25] Dahm, M. (1998) *Byte Code Engineering with the JavaClass API*. Technical Report B-17-98, Friei Universitat, Berlin.

[26] Golm, M. (1997) *Design and Implementation of a Meta Architecture for Java*. MSc, Erlangen.

[27] Wu, Z. and Schwiderski, S. (1997) *Reflective Java*. Technical Report APM.1931.01, Architecture Projects Management Limited (ANSA), Cambridge, UK. Available from http://www.ansa.co.uk.

[28] de Oliveira Guimarães, J. (1998) Reflection for statically typed languages. In *Proc. ECOOP 98, Eur. Conf. on Object-Oriented Programming*, July 20–25. *Lecture Notes in Computer Science*, **1445**, 440–461. Springer, Berlin.

[29] Golm, M. and Kleinoder, J. (1999) Jumping to the meta level: behavioural reflection can be fast and flexible. In *Proc. REFLECTION01, Third Int. Conf. on Metalevel Architectures and Separation of Crosscutting Concerns*, Kyoto, Japan, September 25–28. *Lecture Notes in Computer Science*, **1616**, 22–39. Springer, Berlin.

[30] Chiba, S. (2000) Load-time structural reflection in Java. In *Proc. ECOOP 2000, Eur. Conf. on Object-Oriented Programming*, Sophia Antiopolis and Cannes, France, June 12–16. *Lecture Notes in Computer Science*, **1850**, 313–336. Springer, Berlin.

[31] Pawlak, R., Steinturier, L., Duchien, L. and Florin, G. (2001) JAC: a flexible solution for aspect-oriented programming in Java. In *Proc. REFLECTION01, Third Int. Conf. on Metalevel Architectures and Separation of Crosscutting Concerns*, Kyoto, Japan, September 25–28. *Lecture Notes in Computer Science*, **1616**, 1–24. Springer, Berlin.