USING A LOADTIME METAOBJECT PROTOCOL TO ENFORCE ACCESS CONTROL POLICIES UPON USER-LEVEL COMPILED CODE

A DISSERTATION
SUBMITTED TO THE SCHOOL OF COMPUTING
OF THE UNIVERSITY OF NEWCASTLE-UPON-TYNE
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Ian S. Welch

September 2004

Dedication

To my partner Jim and,
my family especially my parents Stan and Claire.

Acknowledgements

I would like to express my sincere gratitude to several people who have contributed in various ways to the completion of this thesis.

The advice and patience of my supervisor, Dr. Robert Stroud, was crucial in the completion of this work. Also, many thanks to the Defence and Research Agency and particularly Dr. Peter Y. Ryan for providing the financial support, advice and encouragement needed to finish my work.

While at Newcastle, the advice and help given to me by a large number of people, in particular Dr. G. Morgan, Mr J. Warne, Dr. A.S. McGough, Dr. B. Arief, Dr. A. Romanovsky, and Dr. C. Gaciek kept me going. Since leaving Newcastle and coming to work at Victoria University, I have been greatly helped by the contributions of my colleagues. Participation in the Circle of Guilt has been a great help in completing but also staying sane. Special thanks also to Margi Keys for helping proofread the final thesis and keeping me on the straight and narrow with the use of my hyphens!

Finally, I would not have started or completed this thesis if it hadn't been for the encouragement and support of my partner, Jim Whitman.

Contents

List of Tables

List of Figures

Listings

Abstract

This thesis evaluates the use of a loadtime metaobject protocol as a practical mechanism for enforcing access control policies upon applications distributed as user-level compiled code.

Enforcing access control policies upon user-level compiled code is necessary because there are many situations where users are vulnerable to security breaches because they download and run potentially untrustworthy applications provided in the form of user-level compiled code. These applications might be distributed applications so access control for both local and distributed resources is required. Examples of potentially untrustworthy applications are Browser plug-ins, software patches, new applications, or Internet computing applications such as SETI@home. Even applications from trusted sources might be malicious or simply contain bugs that can be exploited by attackers so access control policies must be imposed to prevent the misuse of resources. Additionally, system administrators might wish to enforce access control policies upon these applications to ensure that users use them in accordance with local security requirements. Unfortunately, applications developed externally may not include the necessary enforcement code to allow the specification of organisation-specific access control policies.

Operating system security mechanisms are too coarse-grained to enforce security policies on applications implemented as user-level code. Mechanisms that control access to both user-level and operating system-level resources are required for access control policies but operating system mechanisms only focus on controlling access to system-level objects.

Conventional object-oriented software engineering can be used to use existing security architectures to enforce access control on user-level resources as well as system-resources. Common techniques are to insert enforcement within libraries or applications, use inheritance and proxies. However, these all provide a poor separation of concerns and cannot be used with compiled code.

In-lined reference monitors provide a good separation of concerns and meet criteria for good security engineering. They use object code rewriting to control access to both user-level and system-level objects by in-lining reference monitor code into user-level compiled code. However, their focus is upon replacing existing security architectures and current implementations do not address distributed access control policies.

Another approach that does provide a good separation of concerns and allows reuse of existing security architectures are metaobject protocols. These allow constrained changes to be made to the semantics of code and therefore can be used to implement access control policies for both local and distributed resources. Loadtime metaobject protocols allow metaobject protocols to be used with compiled code because they rewrite base level classes and insert meta-level interceptions. However, these have not been demonstrated to meet requirements for good security engineering such as complete mediation. Also current implementations do not provide distributed access control.

This thesis implements a loadtime metaobject protocol for the Java programming language. The design of the metaobject protocol specifically addresses separation of concerns, least privilege, complete mediation and economy of mechanism. The implementation of the metaobject protocol, called Kava, has been evaluated by implementing diverse security policies in two case studies involving third-party standalone and distributed applications. These case studies are used as the basis of inferences about general suitability of using loadtime reflection for enforcing access control policies upon user-level compiled code.

Chapter 1

Introduction

"It is what I sometimes have called "the separation of concerns", which, even if not perfectly possible, is yet the only available technique for effective ordering of one's thoughts that I know of." (E. W. Dijkstra in "On the role of scientific thought" [Dijkstra 1976])

System administrators are responsible for ensuring site-specific access control policies are adhered to by all users to ensure that local resources are not misused. Ideally the access control policies should be enforced using the site's local security architecture [Butler, Welch, Engert, Foster, Tuecke, Volmer & Kesselman 2000] and should follow the security engineering principles of least privilege, complete mediation and economy of mechanism [Saltzer & Schroeder 1975]. Unfortunately, operating system security is insufficient to implement these principles with respect to applications distributed as compiled code, such as email attachments, dynamic web content, and so forth. Conventional object-oriented software engineering techniques cannot be used to modify the applications so they are constrained by the local security architecture because they do not provide a clean separation of concerns [Dijkstra 1982] or require access to source code. More recent approaches, such as in-lined reference monitors [Erlingsson & Schneider 2000, Evans & Twyman 1999, Sirer, Grimm, Gregory & Bershad 1999, Pandey & Hashii 1999, Hashii, Malabarba, Pandey & Bishop 2000] do implement the principles and provide a clean separation of concerns but do not allow use of the local site's security architecture and do not address distributed access control.

Metaobject protocols (MOPs) are object-oriented implementations of reflection [Maes 1987, Kiczales, des Rivieres & Bobrow 1991]and provide a mechanism which can be used to enforce access control for objects and distributed objects [Riechmann & Hauck 1997, Riechmann & Hauck 1998, Oliva & Buzato 1999, Caromel & Vayssiére 2001, Caromel, Huet & Vayssiére 2001, Caromel & Vayssiére 2003, Fabre, Nicomette, Perennou, Wu & Stroud 1995, Killijian, Fabre, Ruiz-Garcia & Chiba 1998, Killijian & Fabre 2000, Benantar, Blakley & Nadain 1996, Ancona, Cazzola & Fernandez 1999]. Furthermore, a loadtime MOP allows enforcement of access control policies upon compiled code using techniques that allow security engineering principles to be satisfied. However, existing loadtime MOPs do not explicitly address the implementation of complete mediation, or have been evaluated by using them to enforce access control policies upon remotely developed applications using a local security architecture.

This thesis describes the design and implementation of a loadtime metaobject protocol called Kava for Java designed to provide complete mediation and two case studies where Kava is evaluated by comparing conventional enforcement with using Kava for enforcement.

The remainder of this chapter is structured as follows. Section 1.1 introduces the thesis of this dissertation. Section 1.2 provides the motivation for the thesis. Section 1.3 details the contributions of this thesis. Finally, Section 1.4 provides an overview of the thesis itself.

1.1 The Thesis

The thesis of this work is that loadtime MOPs can implement a clean separation between concerns for compiled code allowing reuse of existing security architectures and are able to completely mediate all accesses to base-level objects. A clean separation allows the modularisation of security concerns providing the benefits of reuse, reduced development time and improved comprehensibility. While, complete mediation is required for assurance that access control policies are correctly enforced.

The thesis presents a loadtime MOP called Kava and the results of two case studies that contrast the use of Kava with conventional software engineering techniques. Although there are now other implementations of loadtime MOPs [Chiba 2000, Caromel et al. 2001], Kava was the first to be developed for security enforcement [Welch & Stroud 1999a], can enforce access control policies upon resources implemented by application Java library code, and the first to explicitly address how it meets the security engineering principles.

The design and implementation of Kava is discussed and an argument for complete mediation by metaobjects bound at loadtime is presented. The first case study shows that using Kava leads to a better separation of concerns compared to conventional techniques such as relying upon system-level enforcement or manually inserting enforcement code into application code. The second case study shows that this improvement is also found when the use of Kava is compared with conventional techniques such as proxies and inheritance.

1.2 Motivation

This section provides the motivation for the thesis and presents an overview of the arguments for using loadtime metaobject protocols for security enforcement. The discussion found in the overview is amplified in Chapter 2 that discusses related work and develops a set of goals for the thesis.

1.2.1 New Problem Domains

Security needs have evolved with the move away from the tightly controlled multi-user timesharing systems of the 1960s and 70s to today's wide-area, open distributed systems [Colouris, Dollimore & Kindberg 2001]. Today there is widespread use of applications developed remotely that may not conform to local security requirements whereas, in the past, there was tighter control over the development of applications because they were either developed in-house or in conjunction with trusted suppliers. Unlike centrally managed systems of the past, modern applications can be installed and executed by users without requiring the involvement of system administrators. This has created a requirement for tools that automatically enforce site-specific access control policies upon applications installed by either system administrators or ordinary users. Some contexts where this problem arises include:

Email attachments Most email programs allow users to receive applications in the form of compiled code contained within email attachments. These applications may be buggy or have been developed by malicious individuals for the purpose of damaging the user's system or stealing valuable details such as financial information. For example, the W32/Bagle-F virus [Sophos Anti-Virus 2004] sends itself as a password protected ZIP file. When executed it exploits the user's own privileges to harvest email addresses from the hard disk and re-sends itself via its own SMTP engine. It may also provide a backdoor for remote attackers who wish to access resources on the infected machine. Although viruses are a continuing problem email attachments remain an important and valued aspect of email functionality. An improved approach is to enforce site-specific access control policies that prevented or curtail exploitation of user privileges by an email attachment.

Dynamic web content Most browsers allow the viewing of dynamic web content in web pages consisting of static content and executable compiled, for example ActiveX controls or the popular Shockwave plug-in. Code distributed in this way may contain viruses or Trojan programs that appear to the user as benign but are performing malicious actions in the background. They usually execute with the same privileges as the user so they can misuse any resource that a user has access to. Viruses delivered in this way have been implemented in recent "Phishing" attacks in which attackers attempt to steal the account name and password for a user's electronic banking system [Anti-Phishing Working Group 2004]. These attacks succeed because rogue programs have unrestricted access to the user's hard-drive and can search for these details or collect them with dialog boxes that mimic those displayed in pages delivered to a browser by a bank's legitimate web server. Legitimate programs often require few of the many many system privileges allowed them and a solution to the general problem is the imposition of access control policies that curtail privileges granted to any downloaded program.

Mobile code The two examples provided earlier can be seen as examples mobile code. Mobile code is code produced on one host and executes on a different host. Mobile code may be used to dynamically extend a thin client's functionality, [Yoshikawa, Chun, Eastham, Vahdat, Anderson & Culler 1997] or it may be a convenient way to distribute an application across a group of hosts. The motivation for its use include better resource utilisation or improved performance [Chess, Harrison & Kershenbaum 1995]. In all of these cases, it is desirable that site-specific policies are imposed upon the code and, because mobile code executes as user-level code, imposing security enforcement at the operating system level is insufficient because code may need to directly invoke other code at this level rather than services at the kernel level of the system.

Standalone applications Organisations increasingly use third-party commercially-produced or open-source applications rather than developing their own applications. These applications may be untrustworthy or buggy, requiring site-specific policies to be enforced upon them to restrict the damage that they can cause. For example, a chat application with unrestricted functionality may be installed on users' PCs though security concerns required that access is limited to approved channels or chat servers. In order to enforce a suitable security policy, the application may need modification because it will not possess the necessary hooks for this to be accomplished externally since it has has been designed for a large and indeterminate user-base.

Component-oriented programming In the same way that use is made of third-party applications, organisations may also make use of sub-applications or components that are assembled together to build something larger [Szyperski 1998]. At runtime components would be linked into application code and execute in the same address space. Though programmers have access to specifications for the interface to these components, as with mobile code, interaction between these, other components and the application code cannot be mediated solely at the level of the operating system.

Computational grids Organisations may choose to share computational resources by allowing users to upload compiled code to remote hosts for execution, for example in a **computational grid** [Foster 2001]. A computational grid is made up of heterogeneous resources belonging to multiple administrative domains. Resources may be applications, data or computational services. Although individual organisations may be distrustful, they share resources in a limited way to achieve some mutual purpose. This can mean sending an application to a data source and returning the results to the application's originator. Obviously, the host where the application is executed would prefer to constrain its interaction with system resources, other local programs and with remote resources. This requires imposing site-specific policies upon applications that may not contain the appropriate enforcement code.

The security problem shared by all of these cases is how to enforce site-specific access control policies upon applications distributed as compiled code. To prevent misuse of resources, access to resources implemented as user-level code as well as resources implemented by kernel-level code must be controlled by access control policies. What constitutes misuse might be defined relative to what is considered the minimum required privileges for the application to perform as expected. Additionally, what constitutes misuse might also be defined in terms of an organisation's norms.

Generally, there are two types of access control policy considered in this thesis. The first are application-specific policies where the organisation imposes security policies governing access to resources provided by the application. The second are general policies where the organisation imposes security policies upon the use of resources provided by the organisation.

1.2.2 Man-machine Scale

Before discussing different approaches to enforcement of access control policies in these domains it is essential to establish a layered model of the computer system. Enforcement can be placed in any layer and the choice of layer governs the expressiveness of access control policies. Much of the rationale for loadtime MOPs depends upon which layer in the computer system that enforcement code is placed.

Gollmann calls this the **man-machine scale** because it is assumed that the upper layers of the computer system are more closely tailored to an individual's needs whilst the lower layers are general purpose and tied to the constraints of the hardware. Figure 1.1 shows a layered model of an IT system due to Gollman [Gollmann 1999]. Users run applications that make use of resources provided by other applications, services or libraries. The services or libraries may be middleware or provide access to the resources implemented by the underlying operating system. Applications and services both execute as user-level code, so their access to each other may not be mediated by the operating system. Depending upon the language runtime, the services may or may not be modifiable or protected from the application. The operating system executes as kernel-level code and is protected against tampering by either applications or services.

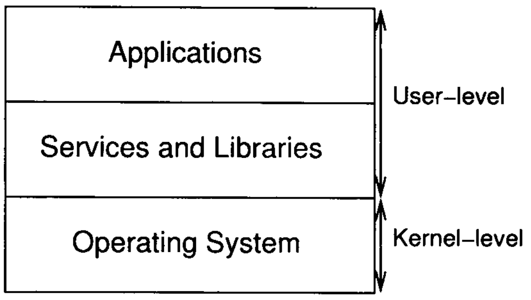

Figure 1.1: Layers of an IT system (based upon Figure 1.2 from Gollmann's book [Gollmann 1999])

1.2.3 Criteria for Evaluation

This thesis uses four main criteria for evaluating the effect of placing enforcement at different layers within the computer system. The first was was coined by Dijkstra [Dijkstra 1976] and the last three are Saltzer and Schroeder's famous principles for good security engineering [Saltzer & Schroeder 1975]:

Separation of concerns To provide reduced development time, improved system flexibility and increased comprehensibility the best way to tackle implementation of complex systems is to separate the system into separate concerns that can be addressed in isolation to each other. With regard to security enforcement this means separating enforcement code from application code to allow separate development, testing and reuse.

Least privilege Least privilege limits the impact of failure. Every program and every user of the system should operate using the least set of privileges necessary to complete the job.

Complete mediation Complete mediation ensures that policies are enforced as intended. Every access to every object subject to a security policy must be checked for authority.

Economy of mechanism Economy of mechanism deals with the practical problem of ensuring that enforcement mechanisms behave as intended. A recommendation is that code should be as simple and small in scale as is reasonable.

1.2.4 Operating System Enforcement

One approach to imposing site-specific access control policies is to rely upon the architecture of the operating system. Although operating systems separate mechanism from policy, which allows policies [Wulf, Cohen, Corwin, Jones, Levin, Pierson & Pollack 1974] to reflect user requirements, the approach suffers from three major drawbacks from the perspective of the application. The first is the principle of least privilege, the second is that economy of mechanism is violated and the third belongs to the realm of complete mediation. The first two are described by Schneider and his colleagues [Schneider, Morrisett & Harper 2000] and the third is related to problems associated with least privilege.

Least privilege is violated in the first instance because the privileges given to the application are usually the same as the user and access controls enforced by the kernel are too coarse grained. Complete mediation depends upon mediation by the kernel in all accesses to protected resources and this is assured because of address space partitioning. However, in many of the problem domains in Section 1.2.1, applications that may be untrustworthy are loaded into the same address space as the user-level code implementing the resources we wish to protect. Finally, economy of mechanism simply may no longer apply in many operating systems because more and more functionality is included within their kernels. This makes it difficult to ensure that they enforce security correctly.

1.2.5 Conventional Object-Oriented Techniques

Schneider argues [Schneider et al. 2000] in favour of language-level enforcement in order to meet the challenges presented by these new problem domains. By placing enforcement at the language level, access control can be formulated in terms of the operations provided by application or library objects rather than being restricted to the operations provided by the system-call interface. Object-oriented languages appear ideal for language-level enforcement because they provide information hiding because an object's encapsulate state is only accessible via the object's interface. Enforcement code can be placed at the interface by inserting access control checks into the object's methods. These access control checks can invoke components of the existing security architecture. This satisfies complete mediation because information hiding requires that all accesses take place via the object's interface. Least privilege is satisfied because access control policies can be specified in terms of the operations provided by the objects, and this is varied upon a per-application basis.

Where resources are implemented by libraries, or libraries are used to implement communication between applications (i.e. middleware), placing fixed enforcement code within libraries might seem attractive. However, this approach suffers from similar problems as those faced by enforcement at the level of the operating system. The end-to-end argument [Saltzer, Reed & Clark 1984] implies that libraries are designed to service a range of applications otherwise the functionality would be included at the application level. This means that libraries have coarse grained interfaces in order to satisfy a range of applications acting as clients to the library. This leads to the problem of least privilege. Implementing distributed access control may be easier to implement because all remote method invocations follow a generic pattern. However, as discussed by Blair and colleagues [Blair, Coulson, Andersen, Blair, Clarke, Costa, Duran-Limon, Fitzpatrick, Johnston, Moreira, Parlavantzas & Saikoski 2001], middleware tends to implement fixed strategies for non-functional concerns that cannot be adjusted on a per-application basis. For example, authentication and confidentiality protocols might be hardwired making it difficult to implement new application-specific protocols. Furthermore, local applications may still be able to invoke each other directly without mediation by the libraries, thereby violating the principles of complete mediation.

An improved approach to dealing with these problems is to manually add enforcement code as required. However, manually inserting enforcement code is a laborious and error-prone process. So object-oriented software engineering techniques such as capabilities [Dennis & Earl C. 1966], proxies [Redell 1974] and inheritance are typically used to simplify this programming task. However, each of these techniques has drawbacks. Inheritance leads to a tight coupling between enforcement and functional code. Similarly, capabilities and proxies both require modification of the application code and have other problems such as the confinement problem or self problem. The tight coupling and need to change application code means that there is not a clean separation of concerns between the enforcement code and application code. As discussed by Stroud and Wu [Stroud & Wu 1995] the lack of a clean separation of concerns between security and application functionality means that changing either requires changes to the other. This reduces system flexibility, maintainability, reduces reuse and reduces comprehensibility.

Aside from the lack of a clean separation of concerns, a major problem with applying these object-oriented engineering techniques is the requirement to have access to source code. For example, source code is not available for Java applets nor for some libraries. Since all conventional object-oriented techniques require some access to source code, these techniques only have limited application.

1.2.6 In-lined Reference Monitors

In-lined reference monitors and related tools pre-process compiled code before execution and insert enforcement code under the control of an access control policy [Pandey & Hashii 1999, Evans & Twyman 1999, Erlingsson & Schneider 2000, Sirer et al. 1999, Hashii et al. 2000]. Enforcement code monitors runtime behaviour and terminates execution if violation of an access control policy is detected. Unlike conventional object-oriented techniques, no access to source code is required.

Depending upon the particular implementation, low-level language events can be monitored allowing the property of least privilege to be satisfied. Also, as long as it can be shown that the enforcement code cannot be removed or tampered with, complete mediation can be shown. In fact, complete mediation is easier to show for in-lined reference monitors than conventional object-oriented techniques because the semantics of compiled code is simpler [Erlingsson & Schneider 2000]. Another benefit of in-lined reference monitors is a better economy of mechanism [Saltzer & Schroeder 1975] compared to conventional object-oriented techniques. This is because inserting access checks into source code relies upon trusting the language compiler to correctly implement the checks in the compiled code thereby making the language compiler part of the trusted computing base.

Although current applications of this technique are able to implement a wide-range of access control policies, for example Erlingsson shows how to re-implement Java's stack inspection policy [Úlfar Erlingsson & Schneider 2000], nonetheless, the focus is upon local access control rather than distributed access control. Implementing distributed access control would require re-implementing the tools, permitting them to change program outputs for example to implement encryption of network communications.

More fundamentally, the focus of this approach is upon replacing or emulating existing security architectures rather than modifying compiled code to make use of existing security architectures. The motivation is to make analysis of the correctness of the security enforcement more tractable. However, for many of the problem domains discussed earlier, system administrators prefer to apply existing security architectures to new application code because of the investment in existing code. Though this approach provides a clean separation of concerns and meets the requirements for good security engineering but is not flexible enough to allow reuse of existing security architectures.

1.2.7 Metaobject Protocols

An alternative approach to separating concerns between enforcement and code functionality is behavioural reflection. An advantage of behavioural reflection over code rewriting is that a wider range of policies can be enforced because the focus is at the level of program behaviour rather than program structure. Behavioural reflection [Maes 1987] is a technique for opening up the implementation of a system in a controlled way using a process called reification. An abstraction of the system's internal state, structure and behaviour is made visible via a meta-level programming interface. This abstraction is causally connected to the underlying system in the sense that any changes made to this model or reification of the system at the meta level, are reflected back into the actual system. Thus, it is possible to customise the behaviour of a system transparently by changing the abstraction of the system.

Metaobject protocols are an object-oriented implementation of reflection. Here, the meta level is implemented as a metaobject that is bound to a base-level object. The interface of the metaobject defines the metaobject protocol. Different metaobjects can be bound to different objects providing different customisations of object behaviour. It is therefore natural to consider placing enforcement code within a metaobject.

A clean separation of concerns is provided by the use of a MOP. The security metaobject can be developed independently of the base-level object and bound to the base-level before application execution. Modularising the different concerns leads to the improvement of software quality [Parnas 1972]. Development and testing time can be reduced because work can be divided between domain specialists. Reusability is improved because security concerns are implemented once and reused with multiple applications. Comprehensibility is improved because, with this approach, security concerns are considered independently of the application. System flexibility is also improved because changes at the meta level do not necessarily impact the base level and vice-versa.

1.2.8 Loadtime Metaobject Protocols

The requirement to be able to enforce access control policies upon compiled code means the binding between objects and metaobjects must take place after compilation. The most appropriate type of MOP providing this type of binding is referred to as a loadtime metaobject protocol [Welch & Stroud 1999a]. A loadtime MOP binds the metaobject to the object at loadtime, this can be achieved by pre-processing the compiled code before loading. Loadtime MOPs can be used where MOPs requiring access to source code are not and a loadtime MOP has better economy of mechanism and greater portability compared to a runtime MOP where modification to the interpreter is required. With the exception of the Simple Security MOP for Java [Caromel et al. 2001], most existing security architectures are not loadtime MOPs [Benantar et al. 1996, Riechmann & Hauck 1997, Riechmann & Hauck 1998, Ancona et al. 1999]. Though the Simple Security MOP provides a limited metaobject protocol and is a proof-of-concept of an improvement to loadtime metaobject protocols, it cannot be used to enforce access control policies on the use of system-resources by applications.

Making use of a loadtime metaobject protocol to implement security enforcement ensures that the requirements of least privilege, complete mediation and economy of mechanism are met. Kava [Welch & Stroud 1999a, Welch & Stroud 2000a, Welch & Stroud 2001] is a loadtime metaobject protocol designed with these requirements in mind, combining the best of MOPs and IRM techniques, making use of existing security architectures, and fulfilling the requirements of least privilege, complete mediation and economy of mechanism.

1.3 Contributions

This thesis makes three main contributions: development of a loadtime metaobject protocol, analysis of the limitations of conventional security engineering techniques for security enforcement and evaluation of how these limitations can be addressed using a loadtime metaobject protocol.

1.3.1 Design and Implementation of a Secure Loadtime Metaobject Protocol

First, the thesis describes the design and implementation of a loadtime metaobject protocol for Java called Kava. Arguments for least privilege, complete mediation and economy of mechanism are presented and analysed so that users of Kava can have confidence that security enforcement implemented using Kava can be trusted. Kava is a loadtime metaobject protocol for Java, implemented as part of the experimental work for this thesis, and represents a proof-of-concept for using behavioural reflection to enforce access control policies upon compiled code. Kava was the first loadtime metaobject protocol [Welch & Stroud 1999a] although other loadtime metaobjects have subsequently been developed [Caromel et al. 2001]. Java was chosen for the implementation because it is a popular language for exploring program mobility due to its first-class support for dynamic class loading and linking. Nonetheless, Java places some constraints upon implementation techniques that Kava must address, for example enforcing access control policies upon the use of system-level resources and dealing with Java's inheritance model.

Figure 1.2 shows the Kava architecture. The standard Java compiler is used to compile both the target Java program and Kava Metaobjects. The metaobjects allow the local redefinition of the semantics of the Java language, for example method execution can be locally redefined to include Java security permission checking before execution is permitted. The resulting byte code is then processed by Kava itself, either by applying Kava to the class files or by intercepting the loading of the compiled classes into the JVM. Kava uses a byte code rewriting toolkit to insert hooks into the program byte code that bring the runtime objects under the control of metaobjects. Where these hooks should be added is governed by a binding specification that is encoded using XML. The binding specification tells Kava which program classes and Java semantics should be under the control of an associated metaobject class. The transformed reflective program is then loaded as normal by the standard JVM.

Access control policies are represented in Kava by a combination of the binding specification and the enforcement code that is encapsulated by metaobjects. When used to implement security using the Java permission-based security model, the security policies are specified by a combination of a Java security policy file, the enforcement code encapsulated by metaobjects and a binding specification that effectively indicates when enforcement checks should be made.

Enforcement upon compiled code is possible because Kava requires no manual changes or access to source code of the application classes. This is a feature that has been absent from existing reflective implementations unless they have modified the JVM implementation. In addition, an approach based upon code rewriting has the benefit that it is easier to make the argument for complete mediation of security related operations by enforcement code compared to other approaches to implementing reflection in Java.

Reuse of access control policies is supported because the set of metaobjects containing enforcement code can be generic and parameterised for a particular application through the creation of an application-specific binding specification. This binding specification goes further than indicating individual methods to be controlled by a security policy but also allows control over field access, method invocation, method execution and exception handling. The ability to control the invocation of methods at the caller side in addition to the traditional control over method execution at the callee side allows security policies to be

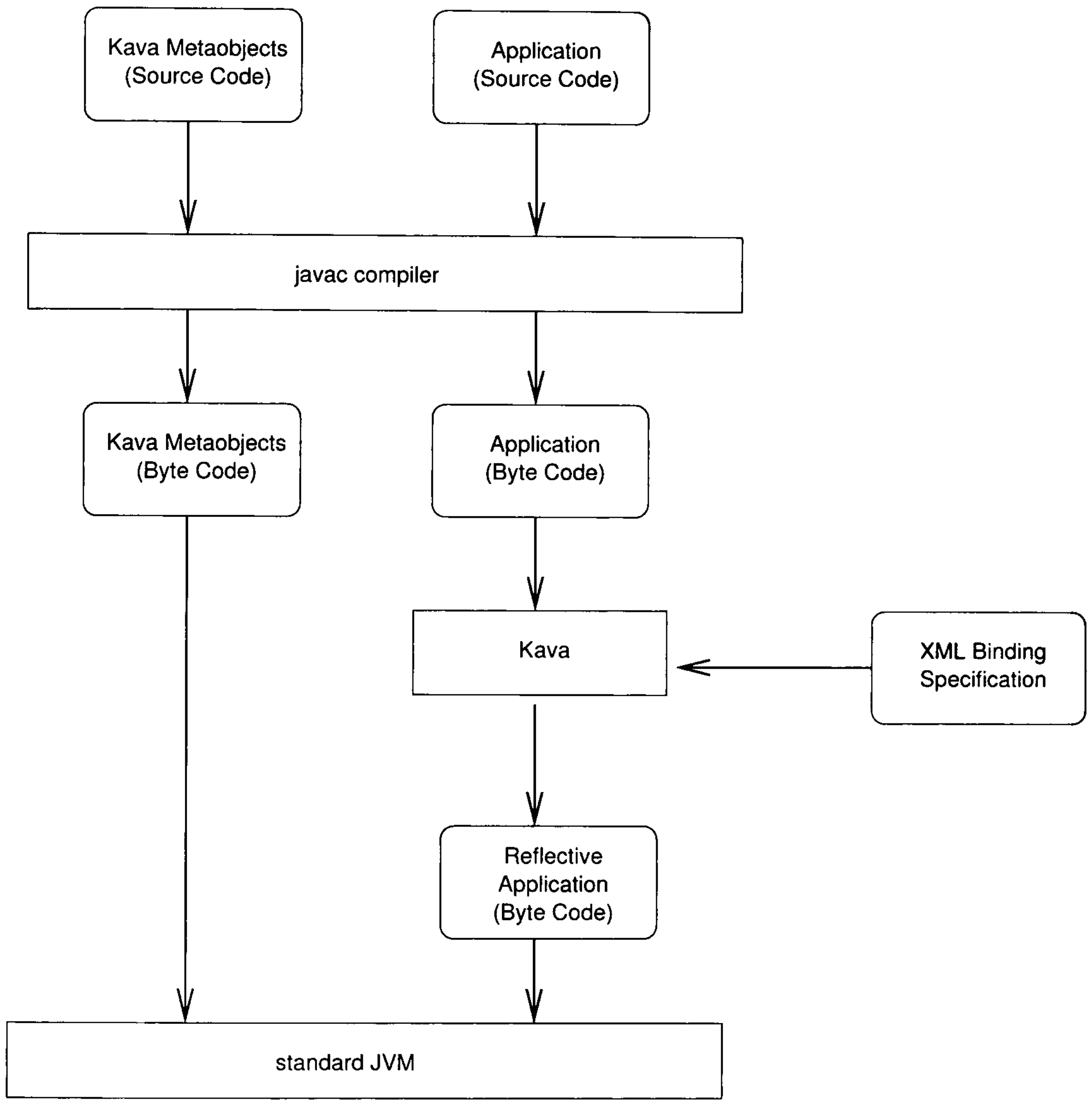
Kava Metaobjects (Source Code)
Application (Source Code)
javac compiler
Kava Metaobjects (Byte Code)
Application (Byte Code)
Kava
XML Binding Specification
Reflective Application (Byte Code)
standard JVM

Figure 1.2: 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.

applied to classes that cannot be rewritten by Kava due to constraints of the Java language.

1.3.2 Demonstration of the Limitations of Conventional Object-Oriented Techniques

Second, this thesis provides a demonstration of the limitations of conventional object-oriented software engineering techniques for implementing access control policies. Two case studies are analysed where access control policies are enforced upon third-party applications, the first case study is a standalone application that uses the Java security architecture and the second case study is a distributed application that uses the **self-defence** security architecture. These case studies enforce security through a combination of manual editing of source code, enforcement placed in system-level libraries, proxies and inheritance.

1.3.3 Evaluation of Separation of Concerns

Third, the thesis evaluates whether Kava can be used to implement a clean separation of concerns between security enforcement code and application code. This allowed an evaluation of the effectiveness of using a loadtime metaobject protocol to enforce local and distributed access control upon remotely developed applications that may be distributed as compiled code. At the most general level, the approach offers a highly practical solution, even through there is a requirement to understand application semantics and structure for enforcing application-specific access control policies. Where access control policies aim to prevent abuse of local resources, there is a similar requirement to understand library semantics and structure. Additionally, to make the approach easier to use, work must be done on a more declarative binding specification and higher-level language for specifying policies.

1.4 Thesis Overview

The remainder of this thesis is structured as follows.

Chapter 2 reviews general security policy enforcement, conventional object-oriented enforcement, in-lined reference monitors and metaobject protocols before defining the goals of the thesis in greater detail. Some of the analysis of the limitations of conventional object-oriented techniques has been presented previously and published [Welch 1997, Welch & Stroud 1998a, Welch & Stroud 1999b].

Chapter 3 provides an overview of the design and implementation of Kava. Much of the work in this chapter has been published in a series of papers charting the evolution of Kava from an initial prototype called Dalang [Welch & Stroud 1998b, Welch & Stroud 1999a, Welch & Stroud 2000a, Welch & Stroud 2001, Welch, Stroud & Romanovsky 2001]. This chapter addresses goals related to enforcement of access control upon compiled user-level code and satisfying the requirements of separation of concerns, least privilege, complete mediation, economy of mechanism.

Two chapters describe case studies undertaken to demonstrate that using a loadtime metaobject protocol to enforce access control provides a better separation of concerns than conventional object-oriented techniques. Chapter 4 reports upon a case study where access control polices are enforced upon a third-party standalone IRC chat application named Lirc. It contrasts conventional object-oriented techniques based upon manually placing enforcement code with a loadtime metaobject approach. Chapter 5 reports upon a case study where access control policies are enforced upon a third-party distributed application named System K. It contrasts conventional object-oriented techniques based upon modifying system libraries, inheritance and proxies with a loadtime metaobject protocol approach. These two chapters draw upon papers describing the Lirc and System K case studies [Welch & Stroud 2000b, Welch & Stroud 2002, Welch & Stroud 2003].

Chapter 6 uses the experience of the case studies to make inferences about general applicability of loadtime metaobject protocol such as Kava to enforcing access control policies. It summarises the results of the two case studies and discusses constraints upon applying the approach to other applications or within the context of other languages.

Finally, Chapter 7 summarises the thesis, the contributions made by the thesis and discusses future work. Some of the future work relating to metaobject libraries for enforcement of Clark-Wilson policies has already been presented elsewhere [Welch 1999] and as has work on a prototype policy compiler for Kava [Lu 2004].

Chapter 2

Related Work

The goals of this thesis have been developed by systematically evaluating existing security engineering techniques against a set of criteria. The criteria used for evaluation are how well the technique provides a clean separation of concerns and conforms to the three classic principles of security engineering.

This chapter is organised as follows. Section 2.1 provides an overview of access control policies. The XACML/SAML framework is introduced, which is a framework for implementing a wide range of access control policies. This framework facilitates comparison of the different security engineering techniques discussed in this chapter. Section 2.2 explains the set of criteria used to evaluate the different security engineering techniques. Sections 2.3–2.6 apply the criteria to operating system enforcement, conventional object-oriented software engineering, in-lined reference monitors and metaobject protocols. Each of the existing techniques or applications of the techniques either have a different goal (in-lined reference monitors) or do not satisfy all of the criteria. This leads naturally to the development of a set of goals for the thesis in Section 2.7. The goals of the thesis are to develop a loadtime metaobject protocol for Java that allows enforcement of access control policies over objects and distributed objects. Unlike existing loadtime metaobject protocols, this metaobject protocol should demonstrate that it meets the requirements of least privilege, complete mediation and economy of mechanism. Furthermore, the metaobject protocol's applicability to enforcing access control policies upon user-level compiled code should be demonstrated through case studies involving enforcing existing security architectures upon third-party applications.

2.1 Access Control Policies

A security policy comprises of both security goals and rules [MAFTIA Project 2003]. Security goals correspond to the traditional view of security properties such as confidentiality, integrity and availability [Pfleeger 1997]. Security goals provide a high-level view of a secure system; security rules are lower-level constraints on system behaviour that ensure the system is robust against accidental or deliberate misuse. Rules may regulate both social and technical systems, where social rules consist of prohibitions, duties and obligations, and technical rules usually take the form of access control rules. The assumption is that both sets of rules will be correctly formulated to ensure that the security goals of the system are guaranteed.

The focus of this thesis are a subset of security policies called access control policies. Access control policies are enforced by access control mechanisms that mediate all access to resources [Lampson 1974, Lampson, Abadi, Burrows & Wobber 1992]. Conceptually, an access control mechanism can be thought of as a device that monitors system behaviour and blocks any action that would result in a violation of a technical rule [Schneider et al. 2000]. As evidenced by work on standards for building access control systems [Rutt, Curtis, Hopkins, Fairthorne, Hartman, Nessett, Vleck, Frantz, Lejeune, Blakley, Mukerji, Ammon, Salmond & Allred 1995, Anderson, Parducci, Adams, Flinn, Brose, Lockhart, Beznosov, Kudo, Humenn, Godik, Andersen, Croker & Moses 2003], access control policies are sufficient to describe the security requirements of many real world systems despite being unable to enforce availability and some confidentiality properties [Schneider 2000].

The OASIS XACML (eXtensible Access Control Markup Language) standard [Anderson et al. 2003] and SAML (Security Assertion Markup Language) standard [Lorch, Proctor, Lepro, Kafura & Shah 2003] provide a useful framework for understanding how access control policies are implemented. XACML/SAML can be used to enforce access policies upon both local and distributed objects. The framework's terminology is used throughout the rest of this chapter.

XACML provides a framework for implementing a wide range of security policies. XACML focuses upon the framework components. It defines the responsibilities of each component, the content of access control queries and responses. XMACL does not define the communication protocols used for communication, these are defined by the SAML framework. SAML is a standard XML-based framework for exchanging authentication and authorisation information [Philpott 2004]. How the authentication and authorisation information is derived is beyond the scope of the SAML framework [OASIS Group 2004]. For example, the SAML framework may describe how to represent the statement "X is authenticated" but it does not describe the protocol for authenticating X, this must be handled by some platform-specific component.

The XACML/SAML framework components act together to implement a reference monitor [Anderson 1972] that mediates all requests for access by an application to a resource according to an access control policy.

The main components in the framework are:

Policy Enforcement Point PEP performs access control by making decision request and enforcing authorisation decisions. The PEP is application-specific.

PDP Policy Decision Point evaluates the applicable policy and makes an authorisation decision. The PDP is application-independent.

Context Handler Context handler acts as a intermediary between the PEP and the PDP, it maps application-specific attributes associated with requests to an abstract syntax and adds additional context if required.

Figure 2.1 shows how the components work in concert with each other to enforce an access control policy. In the figure, an application attempts to access a resource. A resource may be a service provided by libraries or a service provided by another application program. Essentially each access is mediated by a PEP that delegates the access decision to a PDP. The general notion is that of policy/mechanism separation with policy being encoded within a PDP and mechanism implemented by a PEP.

Although XACML/SAML can be applied to controlling access to local resources by local applications the main application for the framework is securing distributed applications. When using XACML/SAML to secure a distributed system there are additional

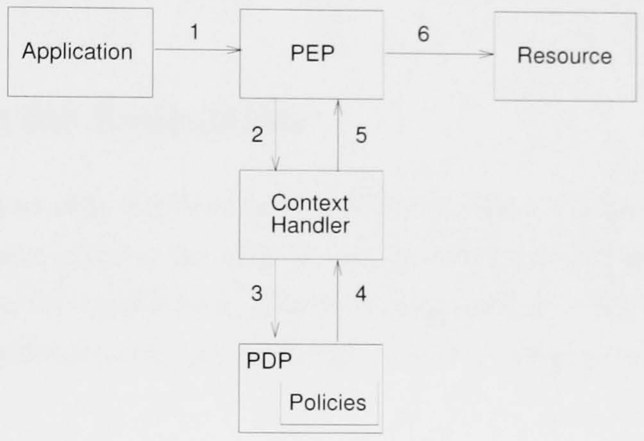

Figure 2.1: (1) An application program requests access to a resource from a PEP (Policy Enforcement Point). (2) The PEP sends the request to an appropriate context handler. (3) The context handler sends the translated request (decision request) to the PDP (Policy Decision Point). The PDP identifies the applicable policy (using the semantics defined by the framework) by searching its policies and determine which policy (or policies) apply to the request. The PDP evaluates the policy rules. (4) The PDP returns the response context (including the authorisation decision) to the context handler. The authorisation decision may be permit, deny or intermediate. (5) The context handler translates the response context to the native response format of the PEP. (6) The PEP fulfils the obligations. If access is permitted, then the PEP permits access to the resource; otherwise, it denies access.

problems that arise from the threats particular to a distributed system that must be solved by implementers. For example, the possibilities of eavesdropping, replay, message insertion, message deletion, message modification, etc. The SAML specification does not define protocols or mechanisms for preventing this but it does specify the support required from transport-level protocols that can defeat these threats. In particular the standard proposes using TLS/SSL [Dierks & Allen 1999] to provide message confidentiality and authentication.

2.2 Criteria for Evaluation

We are interested in security engineering techniques that allow the use of existing security architectures to enforce access control policies upon compiled user-level code. This section discusses the criteria for evaluating the different techniques reviewed in this chapter. An ideal technique should meet these criteria and allow reuse of existing security architectures. The criteria are:

- Clean separation of concerns.
- Least privilege.
- Complete mediation.
- Economy of mechanism.

Table 2.6 on page 55 summarises the result of evaluating operating system, conventional object-oriented, in-lined reference monitors and metaobject protocols against the criteria. The only technique allowing reuse of existing security architectures and meeting the majority of criteria are loadtime metaobject protocols. In-lined reference monitors also meet the majority of criteria but do not allow reuse of existing security architectures.

2.2.1 Separation of Concerns

According to Dijkstra [Dijkstra 1976], the ideal way to tackle implementation of a complex system is to partition the system into separate concerns that can be addressed in isolation to each other. Each concern can be designed, implemented and tested independently modules before being combined together within a single application. This ideal is known as a **clean separation of concerns**.

Besides making implementation tractable, a clean separation of concerns provides the benefits associated with good modularisation. These benefits are reduced development time, improved system flexibility and comprehensibility [Parnas 1972]. Development time is improved because the modules can be developed independently of each other. System flexibility is improved because changes to one module will not impact upon another module. Comprehensibility can be improved because one module can be studied in isolation without requiring understanding of the system as a whole.

A good example of a clean separation of concerns from an operating system perspective is the Hydra operating system [Wulf et al. 1974]. Hydra implemented policy decision making in user-level code and policy enforcement in kernel level code. This separation allowed new policies to be implemented without requiring modifications to the operating system. There are problems with this separation of concerns from an application perspective as will be discussed in Section 2.3.

The problem domains identified in Section 1.2.1 require a security technique that provides a clean separation of concerns. For example, enforcing a local access control policy upon a third-party application requires developing enforcement code separately and somehow imposing it upon the application. The concerns are already separate because the application has been developed independently of the security requirements. Ad-hoc modification of the application may not be possible because the source code of the application may not be available. Additionally, if the source code was available then upgrades to the application would require modification again. Besides being tedious, this would be an error-prone process. Ideally the two concerns should be kept separate and an appropriate technique used to automatically combine them together as required.

A security engineering technique providing this degree of separation of concerns may still require knowledge of the semantics of either the application or the resource being protected. Where there is a specification for the application, application-specific policies can be defined in terms of the application interface. Where there isn't a specification, for example dynamic web content, the policies can be defined in terms of the interfaces of the resource being protected.

Separation of concerns is a desirable property of any security engineering technique and the remainder of this chapter uses the concept as a means of evaluating a range of different security engineering techniques.

2.2.2 Security Engineering Principles

The XACML/SAML framework provides an architecture with a degree of separation of concerns for implementing security policies. However, the correct implementation of security policies in a computer system is difficult due to non-trivial dependencies upon the mechanism's components allied to the difficulties associated with finding the right mapping from policy abstractions to the concrete enforcement mechanisms [Samarati & Vimercati 2000]. However, there is some agreement upon basic principles that designers of enforcement mechanisms should follow to increase confidence in their design, these principles are the ones adopted in this thesis. They were first discussed by Saltzer and Schroeder [Saltzer & Schroeder 1975] in their paper on the basic principles of data protection and the main ones are listed below:

Least privilege Least privilege limits the impact of failure. Every program and every user of the system should be allocated the least set of privileges necessary to complete the job. Least privilege limits damage created when a subject abuses their privileges. For example, granting a doctor the right to request information only about the patients s/he treats protects the privacy of other patients on a hospital records database.

Complete mediation Complete mediation ensures that policies are always enforced as intended. Any access to an object subject to a security policy must be checked for authority. Complete mediation means that a reference monitor that enforces a security policy intercepts every request to a resource being protected. Unless this property holds, a request that leads to the violation of a security policy would be allowed.

Economy of mechanism The principle of economy of mechanism, by keeping the design as simple and small as possible, is an attempt to assure that the mechanism behaves as intended by limiting the unintended effects associated with needless complexity.

When applied to a reference monitor this means minimising its size makes formal verification of its correctness tractable.

These three principles provide criteria for evaluating the effectiveness of a particular security engineering technique.

2.3 Operating System Enforcement

The traditional approach to enforcing security in operating systems is to place PEPs within lower system layers. This has the advantages that it is easier to show complete mediation of accesses by the PEP and it reduces the performance overheads caused by security checks [Gollmann 1999]. In these circumstances there is no need for access to source code when enforcing an access control policy upon an application. Some operating systems, for example DTOS [Secure Computing Corporation 1997] or Flask [Spencer, Smalley, Loscocco, Hibler, Andersen & Lepreau 1999], provide policy/mechanism separation to provide system flexibility. Here, PEPs are protected from tampering by placing them within the operating system kernel and as a result policy decisions can be delegated to a component at the user level, allowing per-user access control policies [Wulf et al. 1974]. This then allows new policies to be specified without requiring changes to the kernel. Unfortunately, from an application-level perspective, operating system security enforcement violates the principle of least privilege and complete mediation is limited to operating system resources provided by the kernel. More generally, modern operating systems do not satisfy the principles of economy of mechanism. Each of these three problems will be dealt with in turn in the following sections.

2.3.1 Evaluation

Table 2.1 summarises the evaluation of operating system security techniques against the set of criteria. Although providing a good separation of concerns, from an application-level perspective, operating system mechanisms do not meet the requirements of least privilege, complete mediation or economy of mechanism.

Table 2.1: Evaluation of operating system enforcement.

Technique	**Description**	**Separation of Concerns**	**Least Privilege**	**Complete Mediation**	**Economy of Mechanism**
Operating system	Fixed placement of enforcement code within kernel.	Good	Coarse	Limited control over user-level code	Complex

Separation of Concerns

The security concern as represented by the enforcement code is completely separated from the functionality of the application that is the subject of an access control policy. Changes to the application do not require rewriting the enforcement code while changes to the enforcement code have no impact upon the application's code. No access to an application's source code is required to enforce access control policies upon the application.

Least Privilege

Least privilege is violated for two reasons. First, the privileges given to the application are the same as those of the user but the user needs many more privileges than the application. This permits the application to perform many actions that are not strictly necessary for it to implement its functionality. A misbehaving application will do far more harm than one with more constraints placed upon it. Second, access controls enforceable by the kernel may be too coarse grained. For example, the operating system can grant access to an individual file but cannot control what or how much data is written to the file or can grant access to the network but cannot distinguish between invocations sent across the network. This is because control can only be enforced at the kernel's interface and this interface is much coarser grained than the application's interface. This leads to the conclusion that the lower the layer in which the enforcement is placed then the less able security policies are to reflect user requirements [Gollmann 1999].

Complete Mediation

Complete mediation is violated because the operating system only mediates accesses to the kernel. Although address space isolation in traditional operating systems may force interactions between user-level code to go via the kernel, the advent of extensible systems such as Java or component-oriented programming leads to a situation where code is not isolated from other code in processes and can invoke each other without mediation by the kernel.

Economy of Mechanism

Depending upon the operating system implementation, the property of economy of mechanism may not be satisfied due to the migration of user-level functionality into system-level code. Schneider and his colleagues use the example of Windows 2000, which contains many millions of lines of code because of the inclusion of windowing and other libraries within the kernel [Schneider et al. 2000].

2.4 Conventional Object-oriented Security

As discussed by Gollman [Gollmann 1999], object-oriented systems provide some useful features to support security. Objects encapsulate data and only allow access via a well-defined interface. This is guaranteed by the language interpreter or a combination of checks provided by compilers and execution environment. Java [Gong, Mueller, Prafullchandra & R. 1997] is one example of a security architecture built using object-oriented principles that can enforce access control policies, and CORBA [Rutt et al. 1995] is an example of a security architecture for distributed object systems that can enforce distributed access control policies.

These features provide a useful basis for implementing security. However, there must also be some means of enforcing a specific access control policy. One way is to insert PEPs at the start of code implementing methods. These cannot be bypassed if all invocations are guaranteed to be via a well-defined interface and no other access to an object's code or state is possible.

2.4.1 Examples

This section evaluates different conventional object-oriented software engineering techniques for enforcing access control policies. For simplicity, it is assumed that each layer below the current layer being considered satisfies the principles of complete mediation and economy of mechanism.

Place Enforcement within Libraries

One approach to language-level enforcement is to place PEPs within the library code implementing the services that applications use, for example middleware or system libraries. The Java security architecture follows this model. Java provides extensible fine-grained access control for compiled user-level code while providing support for a separation between policy and enforcement. In previous versions, new security policies required the rewriting of the security manager but now policy is specified in a file that can be adjusted independently of the application allowing per-application access control policies.

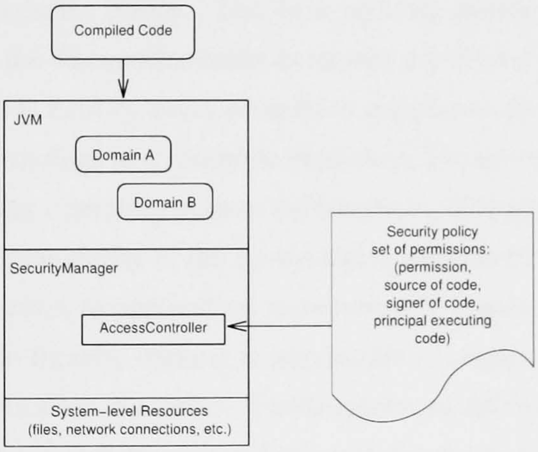

Figure 2.2: Overview of the Java security architecture

Figure 2.2 shows the Java Security Architecture. Compiled code in the form of classes is loaded by a class loader into the Java Virtual Machine (Java VM). A SecureClassloader loads each class and assigns it to a protection domain depending upon who signed the code, who is executing the Java VM, and where the class was loaded from. Unlike operating system security architectures, enforcement of protection domains is not achieved by isolating code within a process only able to communicate with other processes via the kernel. Instead, namespace management and type safety provide the isolation properties. This has the advantage of efficiency, since a process-based mechanism requires expensive context switches when communicating between processes. Requiring a process per-class would make Java unusable because of the communication overheads.

The Java core class libraries play a role similar to a kernel in operating systems such as DTOS or Flask. Java system classes must be invoked to access services or operating system resources and this provides a natural place to enforce access control policies. When a method of a compiled class tries to access resources such as the filesystem or the network, the SecurityManager is invoked. This is because each resource is encapsulated by a class whose methods contain hardwired security enforcement code that delegates the decision as to whether access is allowed by the SecurityManager. The SecurityManager itself delegates the access control decision to an AccessController class that makes the decision depending upon the current Java security policy. The Java security policy assigns permissions to protection domains and the AccessController evaluates the access decision by checking if the requested permission is held by every protection domain on the current call stack.

With regard to least privilege and complete mediation, library-level enforcement suffers from the same problems as operating system enforcement. The granularity of the library's interface determines the granularity of the access rights that can be granted to applications. Additionally, not all accesses to application resources will require invoking services containing enforcement code thereby making it impossible to impose access control policies governing access to application resources. Furthermore, middleware services will implement fixed protocols that can only be changed by rewriting the middleware [Coulson 2002].

The effect of these problems can be illustrated using two simple examples. First, consider preventing malicious email attachments from subverting the user-level email application and using it to send spam. Libraries such as Java's JavaMail API [Sun Microsystems 1999–2004] may mediate access to services allowing the sending of email but the services do not have the application-knowledge required to distinguish between user-initiated email and virus-initiated email. Second, consider using a remotely developed standalone application or component lacking the necessary access controls for enforcing an organisation's local policies. For example, an organisation might wish to enforce a local acceptable use policy upon the use of a third-party IRC (Internet Relay Chat) application. This might require restricting IRC access to approved chat channels. However, there is no way for the libraries used by the IRC application to distinguish between access to particular chat channels and simply providing access to a server.

Manually Place Enforcement within the Application

The solution to the problems stated above is to insert PEPs directly into the application itself. For example, Java's security architecture allows access control checks to be manually inserted into application code. The checks themselves can make use of the standard permission framework and architectural components such as the AccessController. But placing the access control checks at the level of the application's interface allows enforcement to be tailored to the application itself thereby providing least privilege and ensuring that all accesses are mediated by enforcement code. Unfortunately, manually inserting access control checks is laborious and error prone.

Treat Objects as Capabilities

One way to avoid requiring addition of PEPs to existing code is to adopt a capability-based architecture. Capabilities [Dennis & Earl C. 1966] represent access rights as unforgeable pointers to resources and their possession represents the right to invoke the services or methods of the resource. Originally developed for secure operating systems, capabilities have since been used to provide security for applications [Wallach, Balfanz, Dean & Felten 1997] and some languages provide special facilities for implementing capabilities. For example, Java provides a Guard class that hands out capabilities to subjects possessing the appropriate permissions. A client requests the capability by invoking a method of the Guard object, this method performs an access check to see if the client is allowed access to the capability before returning the capability. Once the client has the capability then it can invoke methods of the protected object.

There are two drawbacks to using a capability-based architectures. First, although it requires fewer changes to application source code than simply inserting PEPs within each method of an object, it still requires changes to the application. This doesn't provide a clean separation of concerns because changing the application still requires changes to enforcement code and vice-a-versa although it is more modularised and ad-hoc than manually inserting enforcement code. For example, using a Guard requires inserting code to request access to the object protected by the Guard. Second, the application must ensure confinement [Lampson 1973] of capabilities. But once a capability has been released to a client there is always a possibility that it can be in turn be given to a client who shouldn't have access or that an error in implementation might expose capabilities to clients who similarly should not have access. The problem is that possession allows clients to further delegate access as they wish.

Place Enforcement within Proxies

Redell's Caretaker pattern ([Redell 1974] as discussed in Miller and Shapiro's paper on implementing capabilities [Miller & Shapiro 2003]) is a specialisation of the proxy pattern [Gamma, Helm, Johnson & Vlissides 1995] that is similar to the Java Guard except allows revocation. A proxy object plays the role of a surrogate for an object implementing a resource. The client using the proxy believes it is invoking the methods of the resource object but it is really interacting with the proxy. The proxy enforces the security policy by performing an access check before delegating the method invocation to the real resource object. Hence, the proxy is described as playing the Caretaker role. Unlike a simple capability model, the Caretaker can revoke the access rights granted to a client at any time. Because the client must always go through the Caretaker there is no way for the client to simply bypass the proxy. Obviously the proxy is acting as a fine-grained PEP for individual objects. However, there a four main drawbacks to the use of Caretaker.

First, proxies raise the **self problem** that was first described by Lieberman [Lieberman 1986]. Lieberman claims inheritance-based languages cannot implement delegation. The problem is with the rebinding of the self[1] variable that takes place when the method invocation is delegated. It is possible to work around this by passing the self variable as an additional argument in all method invocations and then making all self invocations use this passed variable instead of the default. This requires that affected classes follow a particular programming convention. However, since the classes being wrapped are constructed independently of the proxy with which they are used, it is unlikely that they would support such a convention unless we could transform the compiled classes' methods.

Second, using proxies to add functionality to existing classes requires **logical wrapping** [Hölzle 1993] 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. 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.

Third, in a reasonably complex application, it is 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 [Lampson 1973] and is a possibility that always exists whenever a proxy is used.

Fourth, an **interface gap** 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, the interface to the proxy class must also be updated.

Successfully implementing proxies requires changes to the objects being proxied and this detracts from the separation of concerns gained. In addition, problems remain regarding the leakage of pointers to the unprotected object and this complicates implementation.

[1]In Java the *self* variable is represented by the *this* keyword.

Place Enforcement within a Superclass

Another approach to implement security functionality or security state is to implement these in a subclass and have classes implementing resources subject to a security policy inherit this functionality or state. For example, a SecureObject class could be implemented that includes a checkAccess method and fields able to track access-relevant state. All classes implementing resources subject to a security policy would inherit from SecureObject and call checkAccess wherever appropriate. This reduces the burden on programmers because they no longer need to re-implement checkAccess for every class or add extra fields to each class definition.

The main drawback is that this approach imposes an obligation on the implementer of the secure class that leads to tight coupling between functional and enforcement code [Stroud & Wu 1995]. In the example given, this consisted of calling inherited methods at particular points and it might also involve defining extra class-specific methods. This causes the code implementing the functionality of the secured class to become intertwined with the code enforcing security. As discussed earlier this makes it difficult to reuse enforcement code. It also hardwires choice about the types of enforcement that can be implemented. Changing enforcement requires changing the superclass. Changing the superclass is problematic because it forces all subclasses to change because the changes may introduce new methods or change the arguments of existing method calls.

2.4.2 Evaluation

Table 2.2 summarises the evaluation of conventional object-oriented techniques against the set of criteria. Conventional object-oriented techniques are able to satisfy least privilege and also complete mediation but do not necessarily provide good economy of mechanism nor provide a clean separation of concerns.

Separation of Concerns

Implementing PEPs within library code provides a clean separation of concerns because enforcement code is independent of application code. All of the others share a lack of clean separation of concerns. Changing either enforcement code or application code will

Table 2.2: Evaluation of conventional object-oriented techniques.

Technique	**Description**	**Separation of Concerns**	**Least Privilege**	**Complete Mediation**	**Economy of Mechanism**
Libraries	Fixed placement of enforcement code within kernel.	Good	Coarse	Limited	Language dependent
Manual insertion	Place enforcement as required within application	Poor	Fine	Complete	Language dependent
Capabilities	Treat object references as capabilities and control handing out capabilities.	Moderate	Fine	Limited	Language dependent
Proxies	Place enforcement code within proxies	Moderate	Fine	Limited	Language dependent
Inheritance	Place enforcement within superclass, all secure objects inherit behaviour and state.	Moderate	Fine	Complete	Language dependent

lead to some changes in both. Manual insertion leads to the worst separation of all because enforcement code is intertwined with application code. Capabilities and proxies provide cleaner separation because enforcement code is more modularised but still requires some changes to application source code: (1) Capabilities require changes to the objects that return references to protected objects and careful programming to avoid the confinement problem; (2) Proxies require the generation of entirely new classes and changes to objects that return references to protected objects to ensure references remain wrapped. Inheritance imposes additional obligations upon developers of application code and require modification of the source code in order to implement inheritance.

Least Privilege

Library-level enforcement suffers from the same problems as operating system enforcement of access control policies because PEPs are located at the wrong level of abstraction and cannot be easily changed. However, placing PEPs at the application level successfully provides least privilege because the access control policies can now specify fine-grained rights on a per-method basis for application or library code.

Complete Mediation

Complete mediation requires that the PEPs cannot be bypassed. Placing enforcement manually within application code or within the superclass provides complete mediation as the language ensures invocations can only take place across a well-defined interface. Capabilities and proxies are more problematic because both suffer from the confinement problem. Libraries only offer limited complete mediation because they cannot mediate in accesses to application resources.

Economy of Mechanism

Economy of mechanism is also quite dependent upon the development and execution environment. However, it is possible to make the observation that implementing PEPs by inserting source code into an application does require that the compiler is trustworthy as well as the execution environment. We must be able to trust that the compiler does not remove the PEPs at compilation time. This results in a larger trusted computing base than operating system enforcement because we now consider the compiler as well as the execution environment (and system libraries) as part of the trusted computing base.

2.5 In-lined Reference Monitors

In-lined reference monitors (IRMs) provide an alternative approach to operating system or conventional object-oriented security engineering. IRMs focus on replacing the existing security architecture rather than bringing applications under the control of existing security architectures. The main benefit of replacing existing security architecture is that the architecture implemented by the IRM may be easier to verify as being correct. Additionally, existing IRMs only enforce access control upon local objects rather than distributed objects. To implement distributed access controls, new mechanisms to support remote authentication and confidentiality would need to be added to existing IRM toolkits because current IRMs do not provide facilities to implement these concerns. Nonetheless, they are worth reviewing because they solve the problems of clean separation of concerns for compiled code and meet the criteria for access control for local objects.

IRM tools in-line automata that monitor execution of the application being secured and halt its execution if a violation of the security policy is detected [Erlingsson & Schneider 2000]. Figure 2.3 describes the IRM security architecture. An application provided as compiled code has the enforcement code for a security policy in-lined by an IRM Rewriter. The resulting application is now a secure application. The key notion here is that the enforcement and policy decision code can be automatically generated from a security policy written in terms of application abstractions. Essentially the PEP and PDP is merged into the target application and the term **in-lined reference monitor** reflects this because the notion is that the reference monitor is not in system libraries or kernels. Instead it is in-lined into the application itself. The inspiration for this work belongs to software based fault isolation [Wahbe, Lucco, Anderson & Graham 1993a] where address-space isolation is implemented by modifying user-level compiled code.

Examples of IRM architectures are SASI [Erlingsson & Schneider 2000], PoET/PSlang [Úlfar Erlingsson & Schneider 2000], Naccio [Evans & Twyman 1999], Ariel [Pandey

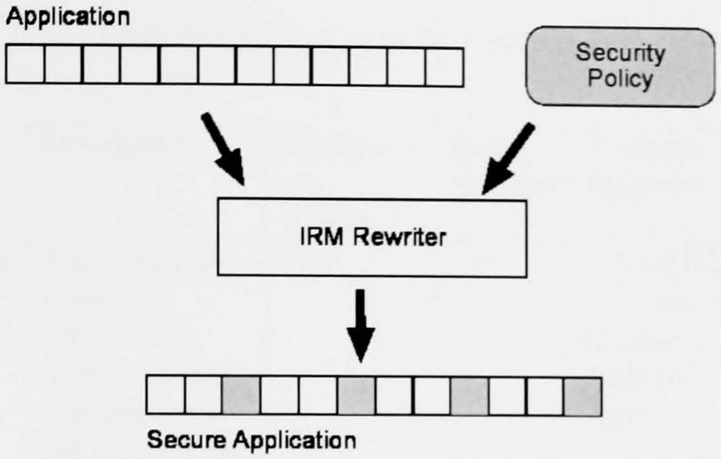

Figure 2.3: In-lined Reference Monitor architecture. Separation between enforcement and functionality. At runtime, an application is processed by an IRM rewriter that inserts enforcement code at appropriate places under the control of a security policy. This produces a secure application containing both enforcement and policy decision code (diagram based upon a diagram from [Úlfar Erlingsson & Schneider 2000].

& Hashii 1999, Hashii et al. 2000] and the Distributed Virtual Machine (DVM) [Sirer et al. 1999].

The following sections analyse how IRM techniques address separation of concerns, least privilege, complete mediation, economy of mechanism.

2.5.1 Evaluation

Table 2.3 summarises the evaluation of IRM techniques. Each example is not shown individually because of their similarity. IRM techniques provide least privilege and economy of mechanism. However, it is less flexible than conventional object-oriented software engineering because it replaces existing security architectures rather than providing a means to integrate applications with them. IRM techniques do provide least privilege with regard to enforcing access control upon user-level objects. However, current IRM implementations cannot enforce access control upon distributed objects.

Table 2.3: Evaluation of IRM techniques.

Technique	**Description**	**Separation of Concerns**	**Least Privilege**	**Complete Mediation**	**Economy of Mechanism**
IRM enforcement	Policy determines in-lining of enforcement within compiled code.**Replaces existing security architecture.**	Good	Fine	Control over user-level code but not distributed objects.	Good

Separation of Concerns

A clean separation of concerns can be achieved by in-lining security codes into user-level applications [Erlingsson & Schneider 2000, Evans & Twyman 1999, Sirer et al. 1999, Pandey & Hashii 1999, Hashii et al. 2000]. In-lined reference monitors (IRMs) realise access control policies by rewriting compiled applications and inserting enforcement code as necessary to effect the access control policy. A clean separation of concerns is provided because enforcement code can be changed independently of application code. A new access control policy can be implemented simply by modifying the policy definition and using the IRM to reapply the policy to the application. Similarly, when the application is changed the access control policy can be reimplemented simply by rerunning the IRM against the application.

Least Privilege

All of the examples of IRM except for Naccio provide fine-grained control for all user-level code. Naccio does not provide fine-grained control because it only intercepts access requests for resources implemented by library code and not application code.

None of the IRM implementations surveyed here provide control over access to distributed objects.

Complete Mediation

Schneider and Erlingsson's [Schneider 2000, Erlingsson & Schneider 2000, Úlfar Erlingsson & Schneider 2000] argument for complete mediation by IRMs depends upon showing that:

- Enforcement code mediates all the events relevant to the access control policy being enforced.
- Enforcement code and its variables are protected against tampering at runtime.

Ensuring All Relevant Events are Mediated. The enforcement code must mediate all events governed by the access control policy being enforced. This means the enforcement code must be capable of mediating an event and the IRM adding the enforcement code must be capable of identifying when mediation is required. Achieving confidence in the toolkit requires both an assessment of its code analysis capabilities and the scope of control of the IRM code. For example, consider an object-oriented language that allows direct access to an object's state without requiring the use of accessor methods. An IRM only able to identify method invocations or insert enforcement code to control method invocations could not be used with this language for controlling access to object state.

Resisting Tampering. The integrity of the enforcement code and its variables must be protected from subversion. The secured application should not be able to bypass, tamper with or modify the values of variables controlling execution of the enforcement code. PoET/PSlang uses the following techniques to prevent tampering with enforcement code:

1. The secured application is prevented from invoking methods belonging to the enforcement code or accessing enforcement variables because the enforcement code does not use names that exist within the application's namespace. For example, PoET/PSLang uses names containing characters that are illegal in Java source code and would be rejected by a compiler. Additionally, because these names are not mentioned in the application's original namespace, the application could not contain code using these names. Note though, that care has to be taken with Java because of

its introspection capabilities – these must be disabled to prevent generic code making use of these capabilities to access variables or invoke methods.

2. The secured application is prevented from bypassing the in-lined enforcement code because there is no way to branch around in-lined enforcement code. There are jump instructions included in the Java VM's byte code and used to implement exception handling and conditional branching but they cannot be used to either jump from one method into the body of another or make arbitrary branches within a method body.

   a. Bypassing enforcement code controlling access to a method requires being able to make an arbitrary jump into the middle of method from another method. This not possible because the Java verifier statically analyses the byte code and disallows jumps between methods.

   b. Bypassing enforcement code within a method requires being able to make an arbitrary jump within a method body. Any jump instruction that attempted to bypass enforcement code around an instruction that is the target of an access control policy would be unable to branch over the enforcement code because it does not exist yet. It could try to confuse the IRM by implementing a complex control flow but as long as the IRM can identify the correct placement of enforcement code then it can update the target of the jump instructions to point to the newly in-lined enforcement code.

3. The secured application is prevented from viewing code as data, otherwise it would be able to remove enforcement code at runtime. This is guaranteed by strong type checking enforced by the Java compiler. Additionally, the Java verifier that forms part of the Java virtual machines ensures that [Yellin 1996]: (1) There are no stack overflows or underflows. (2) All register accesses and stores are valid. (3) The parameters to all byte code instructions are correct. (4) There is no illegal data conversion.

This is not the complete story, it shows that inserted enforcement code cannot be bypassed but what if the attacker can confuse the IRM tool and avoid in-lining of enforcement code. Pandey and Hashii [Pandey & Hashii 1999] argue it is essential for an IRM tool, whose access control policy is based upon associating access checks with particular classes, to take account of Java's inheritance model [Pandey & Hashii 1999]. In Java, the compiled format of a class includes only those methods defined for that class. Any inherited methods are defined in the subclass and at runtime dynamic dispatch ensures that the correct method implementation is used. The implication of this for an IRM approach is that enforcement code must be added to the methods defined in superclasses as well as the class that is the subject of an access control policy. However, this leads to instances of both the superclass and the subclass being subject to access constraints. One solution is to create stub methods in the subclass that delegate calls to the superclass and provide a place to in-line enforcement code. However, Java allows classes to define some methods as final thereby preventing overriding. The solution to this problem is to insert an invocation in the superclass of an access check method defined as a null method in the superclass. The original method remains final but the access method can be overridden by the subclass and enforcement code in-lined.

These arguments for complete mediation can be extended to C++. What is required is the use of additional tools to ensure that access to code is only via well-defined interfaces. For example, SASI makes use of tools such as typed assembly language [Morrisett, Walker, Crary & Glew 1999a] to ensure type safety guarantees are maintained and Naccio relies upon tools implementing Software-Based Fault Isolation (SFI) [Wahbe, Lucco, Anderson & Graham 1993b].

Economy of Mechanism

The security engineering techniques discussed in this section provide good economy of mechanism compared to inserting PEPs into application source code. Preprocessors could be used to insert enforcement code but working with compiled code has distinct advantages. Compiled code is easier to parse than source code and doesn't require the co-operation of the compiler. This implies that the compiler does not need to be treated as part of the trusted computing base [Schneider et al. 2000].

2.6 Metaobject Protocols

Metaobject protocols are a software engineering technique that provides the strengths of conventional object-oriented engineering techniques while providing a clean separation of concerns. Metaobject protocols are an object-oriented implementation of reflection. Reflection [Maes 1987] allows a system's implementation to be exposed, while hiding its complexities, allowing it to be changed. A reflective system is comprised of a base and meta level. Reflection allows the system's base-level implementation to be opened up while hiding implementation details. This is achieved by providing a meta-level causally connected model, which is a reification of the base-level system. The model has two important features. First, the model is an abstraction that hides the complexities of the base-level implementation. Second, changing the model also changes the base-level system because they are causally connected together.

When discussing reflection there is a notion of functional and non-functional concerns. Normally, non-functional concerns are considered orthogonal features that can be found in many applications, for example persistence or distribution. In the context of this thesis, the non-functional concern is security and the functional concern is implemented by the application.

Malenfant and colleagues [Malenfant, Jacques & Demers 1996] define two types of reflection: structural and behavioural. Structural reflection is concerned with providing a reification of an application's structure. Behavioural reflection is concerned with reification of the language's semantics and of its implementation, as well as a reification of the state and implementation of the runtime system.

A metaobject protocol (MOP) [Kiczales et al. 1991] is an object-oriented implementation of reflection that allows for incremental, and constrained changes to a system. Object-oriented programming allows the meta-level model to be changed locally and incrementally. The base level is the object, and the meta level is implemented as a metaobject that is causally connected (bound) to the base-level object. The protocol defines the interaction between an object and metaobject, and constrains changes that may be made to the object's behaviour. Because the meta level is implemented as an object, techniques such as specialisation by inheritance can be used to structure it. Though metaobject protocols can be used to implement both behavioural and structural reflection, this thesis focuses upon the use of metaobject protocols for behavioural reflection.

One possible implementation of a metaobject protocol for behavioural reflection is shown in Figure 2.4. In this example, an object is bound to a metaobject. The metaobject provides a protocol that allows adjustment to the behaviour of method invocations at the receiver. This is implemented by transparently intercepting method invocations sent to the base-level object, reifying the method invocation as a first-class value and invoking the metaobject. This act of interception is known as a meta-level interception (MLI) [Zimmerman 1996]. In a CLOS style **before/after** metaobject protocol the metaobject can perform computations before and after the invocation is delegated to the base-level object for processing. Depending on the details of the metaobject protocol, a **before** computation could change the arguments of the method invocation and an **after** computation could change the result of the invocation before it is returned to the caller.

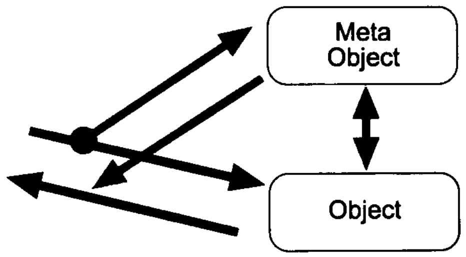

Figure 2.4: A metaobject can be conceptualised as an object associated with a base-level object that intercepts all operations invoked upon the base-level object. It can perform before or after processing and delegate the operation to the real object. Operations might be method invocation, state access and so on. It might also include operations performed by the object itself such as method execution.

A useful way of classifying MOPs is based upon when the binding between an object and metaobject is implemented. Table 2.4 shows the four main types [Welch & Stroud 1999a]: compile time, loadtime, runtime and just-in-time compile time.

The use of reflection and metaobject protocols to implement non-functional properties of a system in a clean way has been proposed for sometime [Stroud 1992]. This approach is based upon the fact that non-functional properties can be implemented as metaobjects and enforced upon base-level objects using a metaobject protocol. Use of metaobject protocols

Table 2.4: A major way that implementations of behavioural reflection differ from each other is their binding time. Binding times can be classified according to when metaobjects are bound to objects [Welch & Stroud 1999a] by the addition of meta-level interceptions (MLIs) [Zimmerman 1996]. In terms of the Java object lifecycle the possible binding times for behavioural reflection (and therefore the last moment at which new bindings can be established) are shown in this table

Binding time	**Description**
Source code	Preprocessor of class source code inserts MLIs
Compile time	Metaobject protocol customises the behaviour of compiler so that either a runtime metaobject protocol is implemented by adding MLIs or the compiler creates a customised compiled class that no longer has an explicit meta level but whose structure has been changed so that the non-functional concerns are realised at the base level.
Loadtime	Instead of pre-processing class source code the compiled representation of class is modified and MLIs are inserted. This can happen at any point from after compilation to the point at which the class is loaded for execution.
Runtime	Metaobject protocol exists that allows the semantics of the language and therefore the interpretation of byte code to be changed at runtime. This is done by changing the runtime interpreter.
Just-in-time compile time	Identical to compile time binding except that the compilation is from some intermediate object code into native object code.

make reuse of code easier because MOPs provide a clean separation of concerns between functional and non-functional features. MOPs allow metaobjects to be developed separately from the base-level objects, which are bound to the base level before use. Depending upon the type of MOP implementation binding can take place anytime from compile time to runtime. The notion here is that non-functional concerns such as persistence and security should be implemented once only and reused with different applications to customise their behaviour so that it meets non-functional requirements. This represents a flexible separation of concerns because not every base-level object needs to be bound to a metaobject, and not every behaviour of a base-level object requires interception by MLIs. There are many examples of using MOPs to implement non-functional concerns. For example, atomicity [Stroud & Wu 1995], concurrency [Briot, Doi, Honda, Ichisugi, Kodama, Ohsawa, Shibayama, Takada, Watanabe & Yonezawa 1990], fault tolerance applications [Fabre et al. 1995], and middleware services [Kon, Román, Liu, Mao, Yamane, a & Campbell 2000].

Using metaobjects to enforce security involves defining a special type of metaobject that implements security enforcement by checking accesses to their base-level object. Essentially, the metaobject plays the role of a PEP in relation to a base-level object by enforcing access checks upon accesses to the base-level object. Using a metaobject to implement security policy enforcement permits the enforcement to be applied to an existing object without requiring any co-operation by the object implementer. Late binding allows enforcement of security policy upon compiled applications (see Table 2.4 for a description of possible binding times). Implementation of enforcement code at the meta level allows the enforcement codes to be reused by requiring co-operation from the target applications.

There are similarities between an IRM approach and using MOPs for security enforcement. Both provide a clean separation between enforcement and application functionality that allows each to be varied independently of the other. Both mechanisms allow security enforcement to only be applied where needed. This means that you do not pay a performance hit for enforcement that you don't need. This is because with a code rewriting mechanism the code can be selectively injected into the application at the interface to the protected code. With a metaobject protocol, the behaviour of the application can be varied incrementally rather than as a whole. Finally, as for IRM, the principle of complete mediation of events relevant to the security policy by enforcement code must be satisfied. For

MOPs, this means that the binding between the base level and meta level should not be able to be bypassed or removed. In addition, metaobjects containing the enforcement code should not be able to be tampered with at runtime.

Unlike IRM techniques MOPs are a general purpose software engineering technique, sufficiently general to address multiple concerns within the same framework and capable of allowing existing applications to be integrated with existing security architectures. IRM techniques focus upon enforcing access control policies and implementing their own security architecture.

A security architecture based upon behavioural reflection and metaobjects, where enforcement is implemented in terms of changes to application behaviour, allows not only enforcement of security but also other concerns as the model for change is not targeted towards a particular concern. This is because the scope of what can be changed using the metaobject protocol is determined by what abstractions exist rather than the range of foreseen changes. The implementation of concerns is then specified in terms of the general language abstractions rather than abstractions relating to the concern itself. This implies that a wider range of concerns can be implemented using the metaobject protocol than simply security. The general idea is that differing concerns are implemented as different metaobjects and these can be combined together at the meta level to implement different concerns. For example, the Friends reflective architecture for fault-tolerant distributed systems can implement not only network security but also distribution and fault tolerance [Fabre et al. 1995].

2.6.1 Examples

Examples of the application of metaobject protocols to enforcing access control policies fall into one of two categories: (1) Those that enforce access control policies upon objects; (2) Those that enforce access control policies upon distributed objects.

The scope of this survey has been deliberately restricted to metaobject protocols that have been used to engineer security. This is because we were interested in those implementations of metaobject protocols that had considered at least some of our criteria. This rules out reviewing MOPs such as Iguana/J [Redmond & Cahill 2002] because they have been designed for a different application area or not been explored with security in mind.

Access to Objects

There are three examples of access control enforcement upon local applications using MOPs: Metaobjects for Access Control [Riechmann & Hauck 1997, Riechmann & Hauck 1998], Guaraná [Oliva & Buzato 1999] and Simple Secure MOP [Caromel & Vayssiére 2001, Caromel et al. 2001, Caromel & Vayssiére 2003].

Metaobjects for Access Control [Riechmann & Hauck 1997, Riechmann & Hauck 1998] defines security metaobjects (SMOs) that are used to implement a capability-based security architecture. In this security architecture, capabilities are references to objects. Security metaobjects are bound to these references and add behaviour such as controls on the delegation of capabilities, etc. Here, the security architecture is implemented by using a metaobject protocol provided by the MetaXa [Golm & Kleinoder 1999] reflective virtual machine. The metaobject is invoked whenever the base-level reference is passed as a return value, or a method is invoked using the reference. It can decide whether access is permitted, depending on the parameters of the invocation or the principal making the invocation. Principals are attached to domains of objects rather than arising out of authentication of subjects or association with threads. The binding between references and metaobjects is established at runtime using an interface to the metaobject protocol, and the association between objects and domains is similarly achieved. The protection of the metaobject, and the binding between instance and metaobject instance depends on the implementation of the MetaXa virtual machine. The main drawback of this mechanism is that a non-standard virtual machine is required.

Like MetaXa, Guaraná [Oliva & Buzato 1999] is a modified Java VM that includes features that enable runtime binding of metaobjects to objects. Although there are no examples of using Guaraná to implement enforcement of security policies, it has been designed with enforcement in mind. In particular, it addresses the problem of a base-level object being able to interfere with the correct execution of a meta-level object and thereby avoids having its requests mediated according to the current security policy. First, the identities of metaobjects bound to an object are hidden from the base-level object and other meta-level objects. This prevents direct tampering. Second, although Guaraná supports rebinding at runtime this can be controlled by the metaobject itself rather than the base level. Therefore, once the metaobject has been bound to the base-level object it cannot be arbitrarily removed.

The Simple Secure MOP [Caromel & Vayssiére 2001, Caromel et al. 2001, Caromel & Vayssiére 2003] is a loadtime MOP implemented using code rewriting techniques. It allows the Java security architecture to be used to implement enforcement of access control policies. It takes special steps, the saving of AccessController context, to ensure that metaobjects do not need to be granted an application permission in order to check that is held by the application objects. Recent work [Caromel & Vayssiére 2003] has extended this to providing a security framework for controlling the accesses the meta level may make to the base level. The use of code rewriting allows the MOP to enforce access control policies upon compiled code without sacrificing portability. However, the Simple Secure MOP cannot enforce access controls upon the use of Java services by user-level code because it can only control method execution by inserting MLIs into method bodies and this cannot be done for the Java core classes that provide the services.

Access to Distributed Objects

Friends [Fabre et al. 1995, Killijian et al. 1998, Killijian & Fabre 2000], DSOM [Benantar et al. 1996] and mCharm [Ancona et al. 1999, Cazzola 2000] are examples of enforcing distributed access control using MOPs. Each mechanism uses one of two ways to implement distributed access control: (1) Use a MOP to implement distributed access control by customising the behaviour of the components implementing remote method invocation implementation. (2) Provide a MOP for remote method invocations that hides implementation details and allows distributed access control policies to be specified declaratively.

In the first mechanism, the choice of abstraction level provided by the protocol constrains the types of policies that can be enforced. For example, distributed access control relies upon authentication of the remote client and this is normally done using some form of digital signature. Implementing this requires being able to manipulate messages as if they were collections of bytes. This is because a digital signature must be calculated using byte values. Furthermore, since the digital signature must travel with the message, the message abstraction must support concatenation so that the result can be added to the message before transmission across the network. This is relatively straightforward in a non type-safe language like C++ as you could simply cast a message as an array of bytes, for example in the Friends architecture. However, it requires careful choice of abstraction in a type-safe language like Java because type safety may prevent manipulating a message as an array of bytes. In a type-safe language, the remote method invocation framework might need to be modified to provide the necessary access. For example, mCharm uses a replacement for the standard Java RMI that provides access to remote method invocations as untyped byte codes.

The second mechanism doesn't expose all the details of the remote method invocation implementation. Instead, it provides an interface that allows the security policies to be specified declaratively and fixed enforcement code implements them. A reflective security architecture that takes this mechanism is DSOM, which is a reflective implementation of the CORBA security architecture. DSOM has a per-orb quality of protection object that specifies the type of protection for messages and access control policies to use when performing remote method invocations.

The choice of approach depends on the desired flexibility. The first mechanism allows arbitrary policies to be implemented but requires more skill on the part of the programmer. The second mechanism is easier to use but only offers a fixed set of policies because the programmer cannot write a metaobject that can directly manipulate messages. In particular it is not suitable when using MOPs as a technique to use an existing security architecture.

2.6.2 Evaluation

Metaobject protocols provide least privilege although existing MOPs tend to address either access control for objects or distributed objects rather than both. Complete mediation has been discussed in relation to compile-time and runtime MOPs but not for loadtime MOPs. None of the current implementations provide mediation of both accesses to local and distributed objects. Loadtime MOPs provide a better economy of mechanism than other types of MOPs. A clean separation of concerns is central to the idea of a MOP but has not been demonstrated with real-world, third-party applications. Table 2.5 summarises the evaluation of the applications of metaobject protocol techniques surveyed in this section.

The Simple Secure MOP for Java is the only example of a MOP that can be used to enforce access control policies upon compiled code without running into the problem of economy of mechanism faced by runtime MOPs. It differs from the loadtime MOP developed in this thesis because it is a deliberately simple MOP designed to explore issues of least privilege in regard to metaobjects. The Simple Security MOP for Java approach has not been applied to the problem of enforcing access control policies upon resources provided by libraries or distributed resources and doesn't explore the issues of complete mediation or economy of mechanism.

Table 2.5: Evaluation of MOP techniques.

Technique	**Description**	**Separation of Concerns**	**Least Privilege**	**Complete Mediation**	**Economy of Mechanism**
Compile-time MOP	Metaobjects enforce policy at runtime and bind to base level at compile time.	Good	Fine	Examples of control over distributed objects but not local.	Includes compiler and MOP implementation in TCB
Loadtime MOP	Metaobjects enforce policy at runtime and bind to base level at load time.	Good	Fine	Examples of control over local objects but not distributed.	Only includes MOP implementation in TCB
Runtime MOP	Metaobjects enforce policy at runtime and bind to base level at runtime time.	Good	Fine	Examples of control over local objects but not distributed.	Includes runtime and MOP in TCB

Separation of Concerns

Implementing enforcement code within the meta layer should allow for a clean separation of concerns. Metaobjects can be designed, implemented and tested separately from the objets to which they will be bound. The examples reviewed in this chapter do not address how successful or not this approach is when applied to real-world applications. Example applications developed to demonstrate the approach suffer from being developed with a particular purpose in mind. This is problematic because the problem domains from Chapter 1 make no assumptions about the structure of the application or component that will be subject to the policy.

Least Privilege

MOPs provide least privilege although the MOPs reviewed above only provide either access control for objects or distributed objects rather than both. At a minimum most MOPs provide access control at the level of individual methods. MOPs providing control over other aspects of object behaviour allow even finer-grained control. The Simple Security MOP takes the discussion of least privilege further than simply the granularity of control over access to base-level objects, it considers the privileges granted to base-level and meta-level objects. The position taken is that meta-level objects should not be granted the same privileges because they only need to check privilege rather than use it.

Complete Mediation

Only mCharm and Guaraná consider the problem of implementing complete mediation. Complete mediation is important because otherwise there is no guarantee that the desired access control policy can be enforced. This requires non-bypassable binding between the base-level object and the meta-level object and preventing the metaobject implementations or state being tampered with at runtime. Should the binding be bypassed then accesses could escape access control checks and should metaobjects be tampered with then the access checks could be removed. The mCharm architecture [Cazzola 2000] argues that if the metaobjects are located on separate hosts that are protected then they cannot be tampered with. There is no discussion of how the binding to the metaobject is achieved. Because mCharm relies upon a compile-time protocol, which customises the compiler, the binding relies upon correct rewriting of source code and the use of mechanisms such as class renaming and language-level visibility (private, protected, etc.) to prevent bypass. No specific argument is made about the effectiveness of the binding. Guaraná provides complete mediation by preventing access to the meta level by base-level objects. Metaobjects are invisible to the base level (preventing tampering) and reconfiguration of the binding between meta and base levels can be controlled thereby preventing the base level removing the binding between the base and meta level. However, the protection of the binding relies upon correct implementation of the Guaraná virtual machine. Requiring changes to the VM introduces the possibility of error and requires a large trusted computing base.

Economy of Mechanism

Regarding economy of mechanism, the type of behavioural reflection used has a direct impact upon the size and complexity of the enforcement code. Remembering the argument for object code rewriting versus source code rewriting it can be argued that source code and compile time binding requires re-implementation of compiler semantics because they both deal directly with application source code, a non-trivial task. Similarly, runtime requires modification of the execution environment, also a non-trivial task. This implies a relatively large and complex enforcement code when compared with code rewriting mechanisms. Loadtime MOPs should have a smaller economy of mechanism because, with regard to MLIs, the compiler does not have to be part of the trusted computing base. Also, it is easier to parse compiled code than source code leading to a smaller implementation of the tool that realises the MOP.

2.7 Goals of Thesis

The previous sections evaluate techniques and implementations of those techniques against criteria for good security engineering and a clean separation of concerns. The results of the evaluation are summarised in Table 2.6. Only in-lined reference monitors and loadtime metaobject protocols meet all the requirements. However, in-lined reference monitors do

Table 2.6: Evaluation of security engineering techniques.

Technique	**Description**	**Separation of Concerns**	**Least Privilege**	**Complete Mediation**	**Economy of Mechanism**
Operating systems	Fixed placement of enforcement code within kernel.	Good	Coarse	Limited control over user-level code	Complex
Libraries	Fixed placement of enforcement code within kernel.	Good	Coarse	Limited	Language dependent
Manual insertion	Place enforcement as required within application.	Poor	Fine	Complete	Language dependent
Capabilities	Treat object references as capabilities.	Moderate	Fine	Limited	Language dependent
Proxies	Place enforcement code within proxies.	Moderate	Fine	Limited	Language dependent
Inheritance	Place enforcement within superclass.	Moderate	Fine	Complete	Language dependent
IRMs	Policy determines in-lining of enforcement. within compiled code. **Replaces existing security architecture.**	Good	Fine	Control over local objects code but **not distributed**	Good
Compile-time MOPs	Metaobjects enforce policy at runtime and bind to base level at compile time.	Good	Fine	Examples of control over distributed objects **but not local**	Includes compiler and MOP impl. in TCB
Loadtime MOPs	Metaobjects enforce policy at runtime and bind to base level at load time.	Good	Fine	Examples of control over local objects **but not distributed**	Only includes MOP impl. in TCB
Runtime MOPs	Metaobjects enforce policy at runtime and bind to base level at runtime time.	Good	Fine	Examples of control over local objects **but not distributed**	Includes runtime and MOP impl. in TCB

not meet the requirement to enforce access control policies using existing security architectures and existing implementations cannot implement distributed access control. On the other hand, existing loadtime MOPs have not been demonstrated to meet all of the criteria and have not been used to implement distributed access control policies.

The goals of this thesis are to design and implement a loadtime MOP for Java that can be used to enforce both types of access control policy, enforce access control policies upon resources provided by applications or libraries while explicitly addressing the problem of complete mediation. The specific goals are to:

1. Implement a MOP that enforces access control policies upon compiled code.
2. Implement a MOP that controls access to both application, library and operating system resources.
3. Implement a MOP that satisfies the requirements for:
   - Least privilege
   - Complete mediation
   - Economy of mechanism
   - Clean separation of concerns
4. Demonstrate a clean separation of concerns compared with conventional software engineering techniques that still allows reuse of existing security architectures.

Chapter 3

Design and Implementation of Kava

This chapter provides an overview of the design and implementation of Kava. Kava is a metaobject protocol for enforcing access control policies upon compiled code. It is intended to allow standard Java security features to be used for enforcement so the meta level is written using the Java programming language. This allows it to make use of the Java language features and existing security architectures when implementing enforcement code.

There are three main sections to this chapter: (1) Section 3.1 provides an overview of the main elements of the Kava metaobject protocol; (2) Section 3.2 provides an overview of Kava's implementation; (3) Section 3.3 evaluates Kava against the goals from Section 2.7.

The following features of Kava specifically address the goals from Section 2.7:

- Kava's metaobject protocol includes extra metaobject methods to allow metaobjects to control access to application, library and operating system resources. This is discussed in Section 3.1.1.

- Special care has been taken to include accesses that are not represented by method invocations to ensure least privilege is provided. This is discussed in Section 3.1.2.

- MOPs provide a clean separation of concerns by allowing non-functional concerns to be modularised. Additionally, Kava allows specialisation through parameterisation of MLIs. This is discussed in Section 3.1.5.

- Kava exploits Java's loading and dynamic linking architecture to allow metaobjects to bind to classes provided as compiled code. This is discussed in Section 3.2.1.
- Kava's MOP satisfies the principle of least privilege because all accesses that are possible in Java can be controlled by a metaobject bound to a base-level object. This has required special care to include the control of static members as discussed in Section 3.2.8.
- The arguments for complete mediation discussed in Section 2.5.1 have been applied to Kava's implementation. However, additional questions particular to MOPs and the implementation of Kava must be considered such as binding, the Java reflection API, how inheritance is handled and so on. This is discussed in Section 3.2.9.

3.1 Language Design of Kava

This section provides an overview of the main elements of the Kava metaobject protocol. The role of metaobjects in the protocol is to allow per-instance changes to the Java language model. A meta-level programmer implements the standard Kava metaobject protocol or subclasses a default metaobject implementation. The metaobject protocol allows redefinition of method invocation by an object, method execution, field access, creation of new instances of classes by an object and exception raising.

Each object at the base level may have an associated metaobject. The association between objects and metaobjects is established at loadtime. This is declaratively specified in a special binding specification that specifies the associations between base-level and metaobject classes. Kava does not provide any special features for composing metaobjects together at the meta level, though it is possible to write meta-level programs using standard patterns for object collaboration such as the chain of responsibility pattern [Gamma et al. 1995].

Meta-level interceptions (MLIs) cause control to switch from the base level to meta level at runtime, the context of the behaviour being brought under control is reified and made available to the metaobject. This allows the metaobject to choose a strategy based

Listing 3.1: IMetaObject Java interface representing the Kava metaobject protocol.

```
public interface IMetaObject {
  // behavioural reflection
  public void beforeInvokeMethod(IInvokeContext context);
  public void afterInvokeMethod(IInvokeContext context);
  public void beforeExecuteMethod(IExecuteContext context);
  public void afterExecuteMethod(IExecuteContext context);
  public void beforePutField(IFieldContext context);
  public void afterPutField(IFieldContext context);
  public void beforeGetField(IFieldContext context);
  public void afterGetField(IFieldContext context);
  public void beforeNewOperator(INewContext context);
  public void afterException(IExceptionContext context);
  // introspection
  public Object getBase();
  public boolean boundToClass();
  public String getClassname();
}
```

upon the context of the MLI and modify the context if required. For example, changing the results of a method invocation or field access.

Reuse is facilitated in Kava because each MLI may have a meta parameter associated with it. This allows specialisation of a metaobject for a new object by modifying the binding specification and goes beyond the usual notion of specialisation of metaobjects through inheritance.

3.1.1 Overview

The Kava metaobject protocol is represented by a Java interface as shown in Listing 3.1. Each Java method on the interface represents a base-level behaviour that can be redefined by Kava.

To provide least privilege and complete mediation, all possible accesses to an object must be represented. Only controlling callee-side invocation (method execution) is insufficient because the Java VM allows direct access to an object's state and there is a technical difficulty in binding metaobjects to Java libraries or services provided to Java applications.

Therefore, the Kava MOP includes methods that do not necessarily appear upon a C++ MOP or other Java MOPs.

In addition to methods that allow redefinition of base-level behaviour, others support introspection using the java.lang.reflect package.

The Kava MOP uses the convention that each type of access (or behaviour that may be changed at the meta level) has a **before** and **after** metaobject method. The before metaobject method defines what happens before the base-level behaviour and the after metaobject method defines what happens after the base-level behaviour.

Every Kava metaobject must implement this interface. To ease the programming task for a meta-level programmer a default implementation of the interface is provided that implements the introspection methods. Normally, programmers will extend the (MetaObject) class and override the before/after methods that correspond to the base-level behaviour that they want to modify. Note that each method takes a single argument representing the reified base-level context. The context is particular to the type of interception, for example when a switch to the meta level takes place upon interception of a field access, the context is reified as a context object with an IFieldContext.

Kava metaobjects can invoke any code written in Java because they are themselves written in Java. In particular the metaobjects can make use of any existing security architecture implemented in Java.

3.1.2 Kava Metaobject Protocol

The following sections discuss each type of metaobject method and how they relate to enforcement of access control policies. The metaobject methods are:

- Invoke metaobject methods
- Execute metaobject methods
- PutField and GetField metaobject methods
- New metaobject method
- Exception metaobject method

- Introspection metaobject methods

Invoke Metaobject Methods

The metaobject methods beforeInvokeMethod() and afterInvokeMethod() allow a meta-level programmer to write code that executes before an invocation takes place and after an invocation has taken place, and the result of the invocation is about to be returned to the caller. This allows policies to be imposed that change the arguments being used in an invocation, control whether a method can be invoked, change the result of an invocation or prevent an invocation from taking place.

Invoke and execute metaobject methods are duals of each other; invoke methods impose constraints upon which methods a base-level object may invoke, and execute methods impose constraints upon which methods belonging to the base-level object may be invoked by other objects. In theory, either could be used in place of the other. Enforcing controls over what methods may be invoked by others is more straightforward because its implementation simply requires no more than the transformation of the method being invoked, rather than all possible methods performing this action. However, the requirement for an invoke metaobject method arises because it is impossible in Java to intercept the loading of some classes and transform them. The reasons for this are discussed in Section 3.2.1.

Execute Metaobject Methods

The metaobject methods beforeExecuteMethod() and afterExecuteMethod) allow a meta-level programmer to write code that executes before and after a method body is executed. It allows implementation of policies, which are similar to the invoke methods, but enforces them on the object being invoked rather than the object making the invocation. This method implements the same functionality as the invoke metaobject methods in reflective languages such as OpenC++ [Chiba & Masuda 1993]. Note that a constructor of an object is an ordinary method, so this method can be used to redefine object initialisation.

PutField and GetField Metaobject Methods

Some access control policies require control over access to the state of the target object. For example, a Bell-LaPadula [Bell & LaPadula 1976] style policy may require that the label associated with an object is changed, depending upon the clearance of the subject associated with an update of an object's state.

The metaobject methods beforePutField() and afterPutField() allow code to be executed before and after fields are updated. For example, the code can be used to update the labels associated with objects when their state is changed. The metaobject methods beforeGetField() and afterGetField() allow code to be executed before and after fields are read.

Note that the fields that are the target of these methods do not necessarily belong to the object that is bound to the metaobject defining PutField or GetField. This is an artefact of the Kava implementation. There is no way to intercept state update other than wrapping the byte code instructions for PutField or GetField. Subject to the visibility of the target field (public, protected, package or private), any object can access the state of another object without invoking an accessor method belonging to the target object's interface. Therefore to implement least privilege and complete mediation, the metaobject protocol must allow these operations to be redefined at the meta level. The metaobject can redefine how the object to which it is bound accesses other object's state.

New Metaobject Method

As mentioned above, the execution of a constructor can be intercepted and redefined at the meta level by redefining the execute metaobject method. This is useful when adding extra initialisation steps to a base-level object or initialising a metaobject. However, it may be impossible to do this if the object is implemented as system-level code or the allocation of space for the new object might itself represent a violation of an access policy.

Alternatively, another approach might be to specify the constructor of the target object's class when binding a metaobject that redefines the invoke metaobject method. The problem is that the byte code for creating a new instance is different from simply invoking a constructor. The Java VM will create a new instance of an object and then calls its constructor. This could result in static code being executed and this code might subvert the access control policy.

Therefore, the Kava MOP goes beyond controlling execution and invocation. It allows requests for new instances of objects to be intercepted and redefined. This is done by implementing the beforeNewOperator() metaobject method, which allows code to be executed before a new instance of an object is created. The metaobject redefining the creation of new instances is bound to any object attempting to create one of the target objects.

Exception Metaobject Method

The Kava MOP allows exception raising to be redefined because not including exception raising as a redefinable behaviour would introduce a covert channel. Otherwise, the developer of an object that will be have its behaviour constrained by the Kava MOP could bypass the MOP simply by raising exceptions and using them to return results to a client.

Kava allows exception raising to be intercepted at runtime but only permits changes to the raised exception rather than allowing exception raising to be suppressed. This is because programmer's develop their programs with a termination model in mind and Kava should not confound programmer's expectations. This is why, in Kava, only an afterException metaobject method is provided. This allows additional behaviour to occur when an exception is raised and the exception instance to be substituted by another that is a subtype of the original instance.

Introspection Metaobject Methods

In order to implement an access control policy, a metaobject may need to introspect upon the state of the object to which it is bound. For example, in a health record context where access decisions must take into account the age of the patient recorded within the record itself.

The three introspection methods are designed to support these operations and default implementations for these methods are provided in Kava. They do not provide the introspection capability themselves, but provide the information needed to use the java.lang.reflect package for introspection.

As a metaobject can be bound either to a class (to support static members, see Section 3.2.8) or an instance of a class there needs to be a means of detecting the fact. A metaobject can determine what it is bound to by invoking boundToClass(). Either true will be returned, indicating that it is bound to a class, or false will be returned, indicating that it is bound to an instance of a class.

A metaobject bound to an object can obtain a reference to the object that it is bound to by invoking getBase(). The metaobject can then use the java.lang.reflect package to introspect upon the state of the instance. A metaobject bound to a class can get the name of the class by invoking getClassname(). As for the case of being bound to an instance, the java.lang.reflect package can be used to introspect upon the state of the class. Note that in Java, classes can define static members that maintain per-class state.

To support complete mediation, metaobjects are hidden from the base level to prevent tampering (see Section 3.2.9) although meta-level programmers can implement their own mechanism if they wish. For example, a registry class can be used for metaobjects to register themselves with when initialised. Base-level objects could lookup their associated metaobject by using their own object pointer as a key. The important point, however, is by default, Kava does not expose the meta level that is enforcing security policies.

3.1.3 Context Parameters

All base-level behaviours have a context that must be accessible for the implementation of access control policies. Depending upon the base-level behaviour being redefined, the context's content will vary. Kava uses the concept of Context objects to simplify the metaobject protocol, which allows the possibility of lazy reification [Masuhara, Matsuoka, Watanabe & Yonezawa 1992]. In Kava there is a partial implementation of lazy reification in that runtime instances of java.lang.reflect.Method, java.lang.Class or java.lang.reflect.Field are only created on demand. This is achieved by reifying only the minimum information needed to derive them at runtime. Ideally, even this minimal information would have its reification deferred until the meta-level program actually needs to access them but because the Java language does not provide direct access to a thread's stack, the current implementation of Kava cannot do this.

Kava differs from other Java MOPs that simply represent context explicitly in the metaobject methods. For example, Javassist [Chiba 2000] reifies the context of intercepting field accesses as the field name and value being set or read. This was explored in Dalang [Welch & Stroud 1999a], a predecessor of Kava, but it led to two problems. First, it leads to method interfaces that were unwieldy because of the number of context parameters. Second, it meant that it would be more difficult to implement lazy reification if the parameters were ordinary Java classes such as Object or String, because there is no way to redefine their implementations. By providing a generic Context object it is possible to provide implementations that can take advantage of Java VM support for lazy reification.

Each metaobject protocol method is passed an instance of a Context object and accesses it using the appropriate interface (see Figure 3.1 to see the hierarchy). For example, Listing 3.2 shows the interface IFieldContext that provides access to the reified context of a field operation (either a get or a put field operation). A meta-level programmer uses the IFieldContext interface to access the reified context of a field operation. S/he is able to find out the name of the field (getFieldName()), the type of the field (getFieldType()), the value either being returned from the field access, or the value that is being written to the field (getFieldValue()) and is able to change the value that is either being returned or being used to set the field value (setFieldValue()).

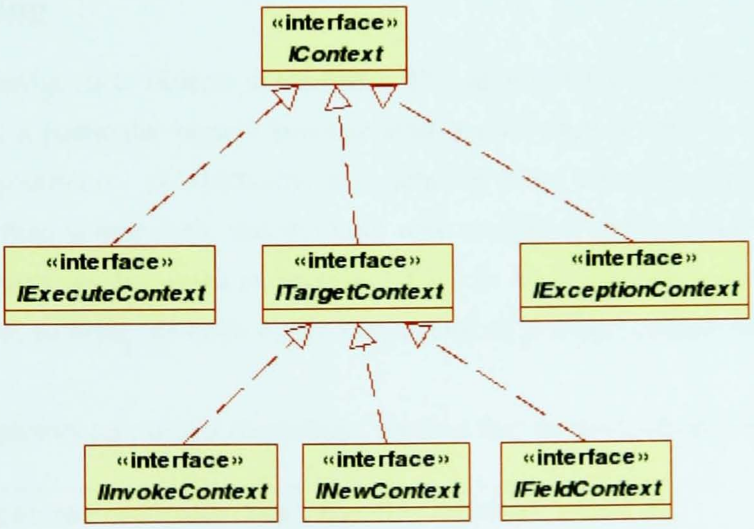

Figure 3.1: Context hierarchy. All contexts implement IContext.

Listing 3.2: IFieldContext interface.

```
public interface IFieldContext extends ITargetContext {
  public String getFieldName();
  public Type getFieldType();
  public Object getFieldValue();
  public void setFieldValue(Object o);
}
```

As the metaobject protocol works with a uniform model of objects, primitive types are not used for methods that introspect upon values or allow the setting of values. Instead, if the field is one that has a primitive value type then wrapper classes must be used to coerce primitive values into objects and vice-versa.

Unlike a standard CLOS before/after metaobject protocol [Attardi, Bonini, Boscotrecase, Flagella & Gaspari 1989], Kava allows **some** behaviours to be overridden. Base-level behaviours can be overridden where the behaviour's reified context object implements the method overrideBase(). For example, the Listing 3.3 shows how method execution can be overridden. Kava ensures that if the overrideBase() method is invoked, the matching after metaobject method will not be invoked.

3.1.4 Binding

Kava binds metaobjects to objects at loadtime. This allows Kava to be used with compiled user-level code, a particular benefit because source code may not always be available to meta-level programmers. Additionally, it is actually more straightforward to parse Java compiled code than source code and working with compiled code provides a better economy of mechanism (as discussed in Section 2.5.1). In Java, all code is encapsulated by a class. Therefore, to bring all code under the control of policies implemented at the meta

Listing 3.3: Implementation of a metaobject method that overrides execution of base-level methods.

```
public void beforeExecuteMethod(IExecuteContext context) {
  context.overrideBase();
}
```

level, each class loaded should be bound to a metaobject class. To allow per-instance policies at runtime, each instance should also be associated with a metaobject class. As Java also supports static members and methods, the implementation of Kava for Java also allows the binding of a metaobject to a class itself as this allows static methods and field accesses to be brought under the control of the meta level.

Unlike compile-time metaobject protocols, Kava does not allow programmers to annotate source code with binding information. Kava works with compiled code so the binding information is supplied separately to the base-level program. The binding specification is represented at loadtime by a BaseMetaConfiguration object and a default implementation has been provided that parses an XML file located in the local file system. Alternative implementations can be used that download specifications from a central server so that an organisation's use of metaobjects to enforce security could be centrally managed. The DTD for the XML binding specification is shown in Listing 3.4.

The Kava binding specification allows metaobjects to be bound on a per class and per behaviour basis. This means that for an object of one class, the invocation behaviour can be intercepted and redefined by the metaobject bound to the object, while for another object of another class, both the invocation and execution behaviours can be intercepted and redefined. Since only those behaviours that are of interest to the meta level are intercepted, the application will perform far better than one where all behaviours are intercepted. This notion of fine-grained binding of metaobjects to objects is inspired by the Iguana metaobject protocol [Gowing & Cahill 1996] that allows the customisation of metaobject protocols.

The Kava binding specification is also used to parameterise metaobjects by allowing meta parameters to be specified for all behaviours, these are discussed in Section 3.1.5. In addition, the specification allows optimisation hints to be passed to Kava (for example, if not all the context is required to be reified).

3.1.5 Meta Parameters

Each metaobject is an instance of a metaobject class. Inheritance can be used to specialise metaobjects. In addition, each meta-level interception can be parameterized to allow metaobjects to change their tolerances or strategies according to parameters encoded

Listing 3.4: DTD for Kava binding specification

```
<?xml version='1.0' encoding='UTF-8'?>
<!--
  DTD for kava binding specification.
  Version 0.94a
-->
<!ELEMENT binding (class*)>
<!ELEMENT class (intercept*)>
  <!ATTLIST class classname CDATA #REQUIRED>
  <!ATTLIST class metaclass CDATA #REQUIRED>
  <!ATTLIST class extends CDATA "">
<!ELEMENT intercept (execute?, get?, put?, raise?, invoke?, new?)>
<!ELEMENT execute (optimisations?, method?)>
  <!ATTLIST execute metaparameter CDATA "">
<!ELEMENT get (optimisations?, method?, field?)>
  <!ATTLIST get metaparameter CDATA "">
<!ELEMENT put (optimisations?, method?, field?)>
  <!ATTLIST put metaparameter CDATA "">
<!ELEMENT raise (optimisations?, method?)>
  <!ATTLIST raise metaparameter CDATA "">
<!ELEMENT invoke (optimisations?, method?, targetmethod?)>
  <!ATTLIST invoke metaparameter CDATA "">
<!ELEMENT new EMPTY>
  <!ATTLIST new targetclass CDATA #REQUIRED>
<!ELEMENT optimisations EMPTY>
  <!ATTLIST optimisations reify (reify|noreify) "reify">
  <!ATTLIST optimisations reflect (reflect|noreflect) "reflect">
  <!ATTLIST optimisations before (before|nobefore) "before">
  <!ATTLIST optimisations after (after|noafter) "after">
<!ELEMENT method (arguments?)>
  <!ATTLIST method name CDATA "*">
<!ELEMENT arguments (type*)>
<!ELEMENT type EMPTY>
  <!ATTLIST type name CDATA #IMPLIED>
<!ELEMENT field EMPTY>
  <!ATTLIST field class CDATA "*">
  <!ATTLIST field name CDATA "*">
  <!ATTLIST field type CDATA #IMPLIED>
<!ELEMENT targetmethod (arguments?)>
  <!ATTLIST targetmethod class CDATA "*">
  <!ATTLIST targetmethod name CDATA "*">
<!ELEMENT metaparameter CDATA "*">
```

within the binding specification file. Each binding specification includes meta parameters that are passed to the metaobject when a meta-level interception takes place. This allows easier reuse of metaobjects because a metaobject can implement several different strategies and the binding specification can encode which strategy is most appropriate for a particular program. For example, it may be necessary to distinguish between methods that update the state of an object from those that don't. A metaobject could be implemented that hard-codes the names of the methods that update the state of an object. Reusing the metaobject requires updating its list of methods to include any new methods belonging to the classes implementing the program it is going to be reused with. In Kava, parameterisation allows the list of methods to be encoded in a binding specification file that is written for the particular program the metaobject is being reused with. At runtime, the parameters are passed to the metaobject. This avoids the need to rewrite the metaobject. This notion of parameterisation is based upon CodA's [McAffer 1995] approach to parameterisation.

3.1.6 Meta-Level Cooperation

Implementing access control policies may require multiple metaobjects with each metaobject addressing a different security concern. Although Kava only allows one metaobject to be explicitly bound to each object, Kava does not preclude composing metaobjects together at the meta level to cooperate in changing the behaviour of a base-level object. This is because the **chain of responsibility pattern** [Gamma et al. 1995] can be used to compose metaobjects together at the same meta level to create a complex meta architecture (similar to the approach described in Mulet et al. [Mulet, Malenfant & Cointe 1995], or Olivia [Oliva & Buzato 1999]. Another approach uses reflection recursively to build a tower of metaobjects (the approach adopted by *FRIENDS* [Fabre et al. 1995, Fabre & Perennou 1996]). The advantage of using chaining is that each metaobject may introspect on the base-level object whereas with a metaobject tower, only the base metaobject may introspect on the base-level object. In addition, with chaining, the cost of interception and reification of method calls is only incurred once, because the metaobject classes involved in the tower do not need to have meta-level interceptions added. Perhaps the tower approach is appropriate only when moving to a higher level of abstraction such as in the *ABCL/R*

[Briot et al. 1990] approach.

3.1.7 A Misbehaved Applet

This simple example demonstrates how Kava can be used in practice. The basic scenario is that a badly-designed or maliciously-designed applet may crash a host system by consuming all its resources by creating an unbounded number of top-level windows. In order to protect against this attack, the system should track the number of windows created and either block or throw an exception when a predetermined limit is exceeded. The class used to generate top-level windows is the Java library class Frame.

Using Kava, a simple preventative approach is to implement a mechanism limiting the creation of new instances of Frame objects. This can be done by creating an implementation of a Metaobject that monitors creation of Frame objects. A simplified implementation of the metaobject is shown below:

```
import kava.*;

public class ResourceMonitor extends Metaobject
{
  public void beforeNewInvocation(INewContext context)
  {
    if (context.getTarget() instanceof Frame) {
      incrementFrameUsage();
      if (exceededMaximumFrame())
      {
        throw new
          RuntimeException("Too␣many␣frames␣created␣by␣applet");
      }
    }
  }
  ...
}
```

Kava performs a static analysis of the applet as it is loaded and binds a ResourceManager metaobject to any object invoking a new operation and the metaobject method beforeNewInvocation redefines how new instances are created. The process is as follows. First, it checks if the target of the invocation is an instance of a Frame class. If this is the case it calls a method incrementUsage that increments a global count of the number of instances of that class and its superclasses. To avoid a malicious program subclassing Frame to avoid being controlled, superclasses are counted as an instance of a subclass of a monitored class that might be created. Then it calls a method exceededMaximum that checks if the maximum limit for the number of instances of any monitored class has been exceeded. If the limit has been exceeded, a runtime exception is thrown and the execution of the applet halts.

The binding specification defining this binding is shown below:

```
<binding>
  <class classname="*" metaclass="ResourceMonitor">
    <new targetclass="*"/>
  </class>
</binding>
```

3.2 Kava Implementation

This section provides an overview of the implementation of the Kava metaobject protocol. The main elements in the architecture are the application-level class loader used to intercept class loading, the meta-configuration object that parses the binding specification and the transformer objects that insert meta-level interceptions according to declarations in the binding specification. The main steps are to add metaobject pointers to the base-level class, add MLIs around blocks of byte codes such as methods and add meta-level interceptions around individual byte codes. Choosing where to add the MLIs is driven by the binding specification. The program being brought under the control of the metaobject protocol must be analysed class-by-class and bytecode-by-bytecode to see if any of the declarations in the binding specification apply. If they do then the appropriate MLI is added.

3.2.1 Overview

Kava is written purely in Java and runs using an unmodified version of the Java Development Kit[1] and uses the BCEL (Byte code Engineering Library) [Dahm 1998] to realise the binding between metaobjects and objects.

The implementation of Kava exploits this ability to intercept class loading of application classes by providing an opportunity to instrument class files before passing the class file to the Java VM for instantiation. Kava was the first implementation of a metaobject protocol for Java that exploits this feature [Welch & Stroud 1998b, Welch & Stroud 1999a] and builds upon existing work on adding new functionality to existing classes [Dahm 1998, Keller & Hölzle 1998, Czajkowski & von Eicken 1998].

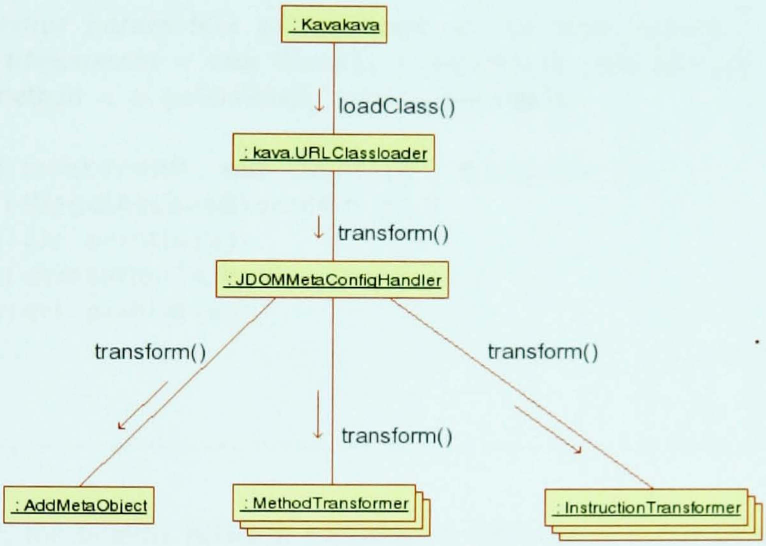

Figure 3.2: High-level object collaboration diagram showing objects and their interactions when a class is loaded into the Java VM.

Figure 3.2 shows the collaborations between the main objects making up Kava that realise the binding. Kavakava is a replacement for the java SDK tool (see Listing 3.5). It is responsible for loading an application's root class and implementing the Kava metaobject protocol before invoking the main() method of the class. Classes are loaded using the

[1]The most recent version of JDK used is 1.4. but most of the work described in this thesis was done using JDK 1.2.

kava.URLClassloader class loader. Because the root class is loaded using the URLClassloader, subsequent application-level classes are also loaded using this class loader.

Listing 3.5: Kavakava replacement for Java.

```
public class Kavakava {
  public static void main(String args[]) throws Exception {
    // parse arguments
    ...
    URLClassLoader loader =
      new kava.runtime.URLClassLoader(...);
    Class c = loader.loadClass(args[0]);

    // construct arguments
    ...
    // construct parameters so and look up the main method
    Class[] parameters = new Class[] { arguments.getClass()};
    Method method = c.getMethod("main", parameters);
    try {
      method.invoke(null, new Object[] { arguments });
    } catch (IllegalAccessException e) {
      System.err.println(e);
    } catch (InvocationTargetException e) {
      System.err.println(e);
    }
  }
}
```

At runtime, the binding between metaobjects and objects is represented by a meta-configuration policy object. All meta-configuration policy objects are instances of classes that extend the BaseMetaConfiguration class. Each policy implementation can determine the bindings in different ways. For example, in Figure 3.2 the JDOMMetaConfigHandler object is an instance of a class that extends BaseMetaConfiguration and determines the metaobject and object bindings by parsing an XML binding specification resident on the local file system. As an alternative to loading the binding specification from the local file system, an implementation of BaseMetaConfiguration might fetch the binding specification from a remote server and verify it using a digital signature. The meta-configuration object determines how to modify the class by parsing the binding specification and will invoke the AddMetaObject transformer to add the fields and methods necessary to bind an object to a metaobject. It then inserts hooks into method bodies (using MethodTransformer) and around individual instructions (using InstructionTransformer). These hooks switch control from the base level to the meta level (the associated metaobject) when the base-level objects carry out certain behaviours. Finally, after rewriting the class to include these traps, the class loader passes an internal representation of the class to the Java VM's byte code verifier. The significance of this is rewritten classes that must still honour rules such as type safety.

3.2.2 Intercepting Class Loading

Kava can intercept class loading because Java allows redefinition of class loading. Class loaders support dynamic loading of classes on the Java platform [Liang & Bracha 1998]. Classes are distributed as binary representations known as class files. Class files do not need to be real files. A class file may be stored in a database, memory or downloaded across a network. The Java VM uses class loaders to load class files and create class objects. Class loaders are ordinary objects that can be defined in Java code. Every class loader is an instance of the class Classloader. The Classloader class provides a loadClass(String name) method that loads a specified class into the Java VM. This involves reading the byte stream representing the compiled class and passing it to a Java native method (defineClass) that instantiates the class within the Java VM itself.

A primordial class loader used to bootstrap class loading and load all Java core system libraries is implemented within the Java VM. The application-level class loaders are implemented by extending the Classloader class and overriding loadClass() with an implementation specific method for loading classes. Should a class loader not be able to locate a named class, it delegates the class loading to its parent class loader. Application-level class loaders cannot override the loading of Java's system libraries because this would allow an attacker to substitute an untrustworthy class for a system class such as java.lang.Object or java.lang.String.

Having to call loadClass explicitly to load classes would make programming difficult, so Java provides a way to implicitly invoke loadClass upon an appropriate class loader.

For example, an application class referred to within a class loaded by an application-level class loader X will be implicitly loaded by X without requiring an explicit invocation of loadClass.

Kava provides a class loader based upon the Java URLClassLoader class for loading and rewriting application classes. This class overrides the default loadClass method and delegates the loading of classes across the network or local filesystem to a new method called defineClass. The defineClass method is shown in Listing 3.6. The method has two parameters: the name of the class to define and an instance of a Resource as parameters. The Resource is a representation of the unloaded class that implements a method for fetching the bytes representing the class and a method for getting the security certificates associated with the class. This array of bytes is then converted into a JavaClass, which is a BCEL representation of an unloaded class that allows manipulation of the class structure. This is then passed to a Kava BaseMetaConfiguration class that performs the changes to the class necessary to implement the Kava metaobject protocol. The JavaClass returned from this method is then converted back to an array of bytes before finally being passed to the Java VM with a set of security certificates that are used to authenticate classes.

3.2.3 Adding Meta-Level Interceptions

The Kava metaobject protocol is implemented using the technique of byte code rewriting. Kava makes use of the Byte Code Engineering Library [Dahm 1998] toolkit to implement the standard transformations that add the hooks necessary to switch control from the base level to the meta level at runtime. The byte code rewriting toolkit deals with technical details such as maintaining relative addressing when new byte codes are inserted into a method, or determining the number of arguments a method supports before it has been instantiated as part of a class. An earlier version of Kava used the Java Instrumentation Object Environment (JOIE) [Cohen & Chase 1998] but this toolkit was discontinued and Kava was rewritten to use BCEL. The particular toolkit is unimportant as long as it allows instrumentation and deals with updating instructions that use relative addressing when new byte codes are introduced. An abstraction layer can be used to hide the dependency upon a particular toolkit so that it is easier to change toolkits if required to do so in the future.

Listing 3.6: Kava's class loader.

```
/*
* Defines a Class using the class bytes obtained from the specified
* Resource. The resulting Class must be resolved before it can be
* used.
*/
private Class defineClass(String name, Resource res) throws IOException {

  // package validation
  ...

  // Now read the class bytes and define the class
  byte[] b = res.getBytes();
  ByteArrayInputStream is = new ByteArrayInputStream(b);

  // create a class object
  JavaClass clazz = new ClassParser(is, name).parse();

  // invoke Kava to rewrite the class to add binding
  JDOMMetaConfiguration.setMetaConfigFile(binding);
  BaseMetaConfiguration metaConfig =
    JDOMMetaConfiguration.getMetaConfiguration();
  KavaClass kc = metaConfig.transform(
    new KavaClass(new ClassGen(clazz)));
  b = kc.getClassGen().getJavaClass().getBytes();

  java.security.cert.Certificate[] certs = res.getCertificates();
  CodeSource cs = new CodeSource(url, certs);
  return defineClass(name, b, 0, b.length, cs);
}
```

Standard byte code rewritings are used to add hooks for individual methods and individual byte code instructions. These hooks reify the context of a behaviour being trapped, invoke the metaobject associated with an object and reflect any changes to the context back to the base level. The metaobjects invoked are completely separate from the byte code hooks and are developed entirely in Java. This separation means that the runtime meta level can be adjusted dynamically at runtime although determining which behaviours are trapped is determined at loadtime.

Kava only performs **binary compatible** byte code rewriting. This ensures that a rewritten class can still be linked against other classes that it references without requiring recompilation or being rejected by the Java verifier. There are nine possible binary compatible changes [Arnold & Gosling 1998]. Kava will implement only two of these, namely:

1. Re-implementing existing methods, constructors, and initializers by inserting MLIs.
2. Adding new fields, methods, or constructors to an existing class or interface to bind metaobjects to objects.

Like compile-time reflection Kava reflects upon the structure of the code in order to implement reflection. However, Kava works at a level much closer to the Java machine than most compile-time approaches that deal with the higher-level language. Although this means it is impossible to extend the syntax of the higher-level language, it does mean that Kava can implement some kinds of reflection more easily than in a traditional compile-time MOP. For example, in the application of OpenC++ version 2 adding fault tolerance in the form of checkpointing CORBA applications [Killijian et al. 1998], data flow analysis is performed on the source code to determine when the state of the object is updated. With Kava no such analysis is necessary; all that is required is to intercept the update of state of an object by changing the behaviour of the update field operation in the Java runtime object model. When an update has been completed a "dirty" flag is set indicating that the current state should be checkpointed.

3.2.4 Realising MLIs

The following sections discuss the different types of transformation Kava uses to realise MLIs.

Adding Metaobject Pointers

The first transformation adds the metaobject pointers required for the metaobject bound to each object and the metaobject bound to the class. A static private field is added that points to an instance of a metaobject responsible for overriding class-based behaviour and a private field that points to a per-instance metaobject is added. In addition, the get$Meta() method that is used to support Kava's approach to integrating with the Java inheritance model is added. The role of this method is discussed in more detail in Section 3.2.9.

Wrapping Blocks of Byte Code

The second transformation makes use of the structure of a class file to identify blocks of byte code representing class methods, initialization methods, and finalization methods and then adds wrappers around them. This allows the method execution and exception raising to be intercepted and control handed to a meta layer. Method execution is simpler than exception raising to reify. Method execution requires only the insertion of hooks at the entry and exit points of a method whereas exception raising requires a try ... catch clause to be synthesised for the entire method.

In order to safely insert new byte code instructions into a class, some difficult technical issues must be solved. For example, how to handle the effects of inserting instructions on branch instructions and on the exception table. Fortunately, the BCEL framework takes care of these low-level issues.

Wrapping Individual Byte Codes

The third transformation is applied to byte code instructions such as those dealing with invocation and access to state. Here fine-grained wrappers are applied around individual instructions. This allows the MLI to intercept base-level behaviour when the class itself makes an invocation, accesses state or creates a new instance of a class. The ability to use a byte code rewriting toolkit to insert code into method bodies makes these types of interceptions possible. Other Java MOPS, for example Reflective Java [Wu & Schwiderski 1997] that rely upon proxying method bodies, could not provide control over operations such as field access or new invocations.

3.2.5 Static Analysis

As a class is loaded by Kava it is analysed to determine if a metaobject should be bound to its instances and whether any of its behaviours should be brought under the control of the metaobject by inserting MLIs. The binding specification is used to determine the answer to these questions. Whether a binding should be established is determined by the top-level binding declaration and specifies, for a Java class or classes, which metaobject is bound to that class' instance. The binding specification also specifies which base-level behaviours should be brought under the control of the metaobject. Each class method and byte code is inspected to see if it implements any of the specified behaviours. For example, a PUTFIELD instruction's target class and field name is tested against a <putfield> declaration to check that it refers to the same class and field.

Where a declaration uses a type to specify whether a behaviour should be brought under the control of meta level, for example the type of the class defining a field, the instanceof operator is used to determine if the target is the type itself or a subclass. Here, Kava is required to determine the inheritance relationship between classes before they are loaded into the Java VM to ensure that classes are instrumented before they are made immutable. The current version of Kava requires this information to be encoded in the binding specification but a better approach would be to use static analysis to determine the relationship between classes before loading (assuming that all classes are available or can be retrieved at runtime). This is discussed again in Section 3.2.9.

Where a declaration uses a simple name, for example the name of a field, syntactic matching is used. Here, names may match, or if a wildcard has been used in the declaration, then any name may be deemed to match the declaration.

3.2.6 Instantiating Metaobjects

Normally this is achieved by adding two fields: a static private field that points to an instance of a metaobject that is responsible for overriding class-based behaviour, and a private field that points to a per-instance metaobject. In addition, an accessor method is added that uses a naming convention normally disallowed by the Java compiler.

3.2.7 Reification

The state of a base-level object is available to a metaobject via the standard Java features for introspection provided by the java.lang.reflect package. This is accessible to the metaobject because Kava is implemented in the Java language, making it a meta circular reflective language [Cointe 1987] ensuring that all the features of the base-level language are available to the meta level. All that is required is access to a pointer to the base-level object and standard Java introspection can then be used to access the state of a base-level object or invoke its method. This is not passed with each MLI, however, instead the Kava metaobject protocol provides methods for metaobjects to discover the identity of their base-level object.

Beyond the state of the base-level object, there are five types of reified context associated with the switch to the meta level. The context is available to the metaobject at runtime and can be changed by the metaobject. On return to the base level, the changes to the context are reflected in changes to the base-level state. The types of context are:

1. The Invoke context reifies the target object, the method name, arguments and return value associated with a method invocation.

2. The Execute context reifies the method name, arguments and return value associated with a method execution.

3. The PutField and GetField context reifies the field name and values associated with a field access.

4. The Raise context reifies the exception being raised.

5. The New context reifies the name of the target class and arguments associated with the creation of a new instance.

There is only one implementation of Context. This results from an optimisation choice. At the start of each method, a new Context object is allocated. It is then reused for different types of MLIs to avoid the cost of object initialization (an expensive operation that should be minimised[Roulo 1998]). There is a context object created for each method so that multi threading can be supported. An alternative (and possibly more efficient) way is to use the per-thread variables that are now available in Java.

The metaobject protocol reifies each element of context as a java.lang.Object. The problem is that Java also supports primitive data types such as int, byte, etc. As these primitive types are not objects, extra work must be done to allow them to be reified as objects. In the initial prototypes, this was done by inserting byte code instructions that wrapped them using the Java wrapper classes such as Integer, Short etc. This adds considerable overhead that is not justified if the meta-level program never uses the reified arguments. To reduce the overhead, these primitive types are reified using the ContextBuilder helper class. This allows them to be stored as primitive types that are not wrapped unless the meta-level programmer actually uses a Context object interface to access the argument.

3.2.8 Static Members

Java supports static members in addition to per-instance members. Therefore, to ensure least privilege and complete mediation, the implementation of Kava had to take account of static methods and fields.

Since the context of behaviours involving static members is different from those involving instances, an approach is to provide a different metaobject protocol when dealing with static members. In addition, there is the question of whether a metaobject or a metaobject class with static methods should be bound to a class with static members. This also leads to a different metaobject protocol that has an interface made up of static methods.

One of the aims of Kava is to provide a uniform metaobject protocol, and developing a separate metaobject protocol to deal with static members complicates this. Instead, the same metaobject protocol is used in all cases. This has the following impact upon the design:

- Some context types reify a target reference, static members do not have targets so the convention adopted uses a default value of null where there is no target reference.
- Each class (which is an object in Java) has a metaobject bound to it in the same way that an instance of the class would have a metaobject bound to it. By changing the implementation of the metaobject, the behaviour of the classes' static members can be changed.

3.2.9 Implementing Complete Mediation

The argument for Kava providing complete mediation is presented in the following sections. The argument is based upon the following assumptions:

Trusted computing base The Java Runtime Environment, the Kava implementation, the enforcement metaobjects, the binding specification and the Java access control policies are trusted.

Ensuring Kava is always invoked Kava always has access to classes being loaded in the Java VM so it can add MLIs as required.

Enforcement mechanism is protected Attackers cannot tamper with MLIs or metaobjects.

Inheritance is accounted for by Kava Attackers cannot confuse Kava's static analysis by subclassing protected objects.

Each of these assumptions is justified in the following sections.

Trusted Computing Base

Kava assumes that the Java Runtime Environment (which includes the Java system classes, the Java VM and Java byte code verifier), the Kava implementation itself, the metaobjects encapsulating the permission checks, the binding specifications, and the Java access control policy files, can only be correct.

The current implementation protects metaobjects against tampering by using a combination of permissions and code-signing techniques. All metaobjects are packaged into JAR files that are signed by the security developer using his or her private key, which is associated with a X.509 certificate. The certificate must be distributed via a trusted channel to the destination system. The metaobject class has a check in its class initializer that the current thread holds a KavaPermission permission named "initialiseMetaobjectClass". This requires the base-level program code and the metaobjects to have been granted this permission in the host's Java access control policy file. The entry in this file for the metaobjects should grant the "initialiseMetaobjectClass" permission on the basis of the identity of the signer of the JAR file they are contained within. If the JAR file has been corrupted, or if the metaobjects are actually provided by another provider then the permission will not hold. The permission is checked when the metaobject class is loaded and initialised.

The Java Runtime Environment, the Kava implementation, the binding specifications and Java access control policy files are assumed to be protected using operating-system level protection such as ACLs on files. This at least matches the current situation with the Java security architecture and future work might consider using cryptographic signatures to protect the Kava implementation and the binding specifications.

Ensuring Kava is Always Invoked

Kava needs control of class loading in order to rewrite classes at loadtime. However, if a class loader is loaded at runtime it may be possible to use it to bypass Kava's class loader. Fortunately it is possible to use Kava's metaobject protocol in order to avoid this problem. The general idea is to use a metaobject to intercept the act of class definition and ensure that Kava is invoked before class definition takes place. This raises two issues: dealing with application-level class loaders, and Java system library class loaders.

The first type is an application-level class loader. All application class loaders must call the native method defineClass and pass it a byte code array that represents the class to be instantiated within the Java VM. A special metaobject that is bound to the invocation of this method by any class loader can be used to prevent the bypassing of the Kava class loader. This metaobject simply applies Kava to the byte code and returns the transformed byte code to the base level where the class loader passes it to the Java VM for instantiation.

This means that any application-level class loader can be transformed into a class loader that brings classes under the control of Kava.

The second type is a system class loader such as an URLClassloader. Since Kava cannot rewrite system classes, this would seem to be a problem. However, a metaobject can be bound to any invocation of a method on a system class loader. This can redirect such method calls to a Kava version of the system class loader that actually does the job of fetching the bytes and instantiating them as a class. When the malicious program calls getClass, the method is not invoked on the system class loader but is dispatched by the metaobject to a Kava version of getClass() that ensures the correct rewritings are applied.

Another benefit of using Kava recursively to bring all class loading under its control is that this prevents malicious use of a byte code rewriting tool to remove Kava's hooks. As Kava intercepts all calls to defineClass, it will always be the last tool to add byte code before the class is instantiated. This means that even if another byte code rewriting tool inserts its own byte code, these cannot have any effect on the hooks that Kava inserts. Note that (as guaranteed by the Java platform) we assume that there is no way at runtime to inject byte code into a loaded class.

An unresolved issue are native methods. Java permits native code to be invoked directly by classes and this code bypasses all Java security controls. Currently, Kava does not provide any control over these although a metaobject approach might be used to detect the use of native methods and halt execution if these are used.

Protecting the Enforcement Mechanism

As highlighted by Erlingsson and Schneider [Erlingsson & Schneider 2000, Úlfar Erlingsson & Schneider 2000] in their work on in-lined reference monitors, complete mediation requires protection of the integrity of the mechanism enforcing the policy. In the context of metaobjects and metaobject protocols, this means that the integrity of binding between the metaobject and base-level object must be protected, and pointers to the metaobject bound to a base-level object must be unavailable to base-level objects to prevent subversion of the enforcement mechanism. Otherwise a program might remove the bindings at runtime, and either branch past the meta-level interceptions implementing the bindings, or tamper with the metaobject by invoking its methods or replacing it with a null metaobject. Ensuring non-bypassability, but not preventing an enforcement metaobject to be switched for a metaobject that always grants access, means that access control policy enforcement will not take place.

Erlingsson and Schneider protect the integrity of their enforcement mechanism in two ways [Erlingsson & Schneider 2000, Úlfar Erlingsson & Schneider 2000]. First, they use a base-level language that prevents code from being treated as data and enforces an encapsulation boundary preventing arbitrary branching. Preventing code from being treated as data ensures that the program containing the in-lined reference monitor cannot remove the enforcement code at runtime by dynamically rewriting parts of the program. Preventing arbitrary branching ensures that the program cannot simply branch around enforcement code. Second, access to the enforcement mechanism is prevented by choosing names and types for interception code that are invisible to the base-level code. This can be achieved by choosing names that either are not in the base-level program's namespace and/or using names that are illegal in source code but legal in object code.

Kava applies these techniques to the implementation of meta-level interceptions (MLIs). MLIs are in-lined into the target class, these must be non-bypassable and the base-level application must not be able to tamper with the metaobject invoked by the MLIs. First, Java was chosen as the base-level language because it offers several low-level security guarantees such as lack of pointer arithmetic and type safety [Yellin 1996]. The lack of pointer arithmetic prevents code being accessed as data, and type-safety prevents arbitrary jumps to instructions. Furthermore, because MLIs are inserted after compilation and cannot be removed at runtime, any attempts to write programs branching within a method in order to avoid MLIs implementing either invoke or field behaviours, will be defeated because Kava will locate all the places MLIs need to be inserted and adjust branches so that they branch to the MLIs before the target instruction. Second, base-level access to the metaobject is prevented through namespace control. When the base-level program is written, the field names for the metaobject pointers are not in the program's namespace. This means that no compiler can compile code that tries to use these names as they are not defined yet. In addition, the field names contain the $ character and this character is not allowed to be used in Java source code (although it is valid at the byte code level). However, Java does provide the ability to introspect upon classes. This makes it possible to access a field without requiring the field's name, for example the list of fields associated with an instance can be obtained as an array of fields and code can iterate through the array looking for one of type IMetaObject. Therefore we assume that the use of java.lang.reflect package must be prevented or controlled to avoid introspection being used to dynamically access fields. Either the Java security architecture can be used to turn off access to metaobject fields, or Kava can itself be used to prevent requests for access to these fields by binding a metaobject to classes making use of classes from the java.lang.reflect package.

Dealing with Inheritance

When implementing Kava, the design required consideration of how Kava's binding model would interact with Java's inheritance model. First, it was decided that Kava must ensure that metaobject bindings are inherited from parent classes because choosing not to make bindings inheritable provides a way for an attacker to bypass an enforcement metaobject by simply subclassing the protected class. This implementation must take account of Java's method of implementing inherited behaviour. Second, a conflict might occur between inherited and explicit bindings.

Implementing Inheritance of Bindings One approach to implementing inheritance of bindings is to override every inherited method in the subclass by a stub method that simply invokes the superclass's method. MLIs are then added to the proxy methods. This will not work where the superclass has defined a method as final because the Java verifier rejects attempts to override a final method. This could be addressed by changing the superclass' methods to be non-final though this might introduce a security vulnerability.

Kava uses an alternative approach derived from the work on using byte code rewriting to enforce access control policies by in-lining enforcement code [Pandey & Hashii 1999]. The basic idea is to edit each method of the superclass and insert MLIs as specified for the subclass. This addresses problem one and two described above. For example, if the execute behaviour of all methods of the subclass are being brought under the control of a metaobject then the superclass R would appear as in Listing 3.7.

Listing 3.7: Superclass containing MLI for method execution.

```
class R {
  private myMeta$Object = new MetaObject("meta1");
  final public void f() {
    get$Meta().beforeExecuteMethod(...);
    <code for f>
  }
  private get$Meta() {
    return myMeta$Object;
  }
}
```

Listing 3.8 shows how a class, class RC in this case, subclasses then overrides the get$Meta() method to return the appropriate metaobject.

Listing 3.8: Subclass defining its own MetaObject to use for method executions.

```
class RC extends R {
  private myMeta$Object = new MetaObject("meta2");
  private get$Meta() {
    return myMeta$Object;
  }
}
```

This means invoking f() on an instance RC will result in methods implemented by RC and R invoking the MetaObject instance "meta2". On the other hand invoking f() on an instance of R will result in the MetaObject instance "meta1" being invoked instead. This is because Java method resolution always chooses the most specific method definition to use [Gosling, Joy & Steele 1996].

Inserting MLIs into superclasses because instances of subclasses are bound to metaobjects is problematic if the superclass is loaded before the subclass. After it has been loaded, Kava cannot modify its implementation. Therefore, during the program analysis stage Kava must identify all subclass/superclass relationships before loading any classes. This can be done through static analysis, for example the BCEL toolkitlong-term allows classes to be introspected upon before loading them and the BCEL Repository class allows subclass relationships to be examined and analysed without loading a class into the Java VM.

This is quite expensive depending upon the length of inheritance relationships. The current version of Kava simplifies this task by placing the burden upon the meta-level programmer who must specify the relationships between classes in the binding specification. Static analysis is a better long-term solution because the meta-level programmer may not be able to anticipate what classes will be loaded by Kava. This is particularly true of mobile code or other code where a detailed specification of the application is not available. This use of static analysis should be the focus of future work.

Dealing with Conflict The second problem of possible conflicts between explicit bindings and inherited bindings needs to be considered in the context of the different possible combinations of bindings and inheritance relationships. The relevant combinations of bindings are: (1) the superclass R has a binding but the subclass RC does not. (2) the superclass R does not have a binding but the subclass RC does. (3) both R and RC have bindings. Another issue to consider is whether Kava should allow bindings on the basis of objects implementing interfaces given that Java permits an object to implement multiple interfaces. This can be resolved by treating it as another form of case three.

Each case is considered below:

Case one The metaobject bound to RC is used when methods defined only in RC are invoked upon either instances of the subclass or superclass. In the second case, the metaobject bound to R is used when methods defined either in R or RC are invoked upon instances of the subclass R. Attempts to invoke methods defined in RC upon instances of RC will fail because there is no metaobject bound to RC. This can be corrected by either checking if the metaobject pointer is null before attempting to invoke a method or associating a metaobject with RC instances that does nothing when invoked. Kava currently does the latter although further investigation will determine whether the former is more efficient. In both cases there is an overhead because reification is done even though it may not be needed.

Case two In this case there is no conflict. The subclass retains it binding and the superclass is not affected.

Case three In this case a conflict may occur. According the scheme described so far, this would result in the subclass's binding overriding the superclass's binding. This may result in a **metaclass incompatibility problem** because an inherited method may rely upon being able to invoke the metaobject bound to instances of its defining class rather than the metaobject bound to instances of the subclass [Graube 1989, Forman, Danforth & Madduri 1994, Forman & Danforth 1999]. Note that the base-level programmer is unaware of the meta level but the meta-level programmer may have intended to modify the behaviour of the superclass methods in a way that is not implemented by the subclass' metaobject. For example, the subclass might bind to a less restrictive metaobject thereby creating a security hole.

We could address these conflicts by applying multiple inheritance rules, as described by Forman and Danforth [Forman et al. 1994, Forman & Danforth 1999], to the metaobjects and synthesise a new metaobject or simply invoke each metaobject in turn. However, we have to question whether this automatic composition of metaobjects is always appropriate and ask "What if the behaviours of the metaobjects are incompatible?" The current version of Kava resolves these questions by refusing to load a class if a possible conflict is detected and leaves it to the meta-level programmer to manually resolve the metaclass incompatibility. Note that null metaobjects are assumed not to conflict with any other metaobject. An improved approach might be to allow multiple inheritance and generate warnings on the assumption that in most cases multiple inheritance will not cause a problem.

3.2.10 Performance

The focus of this thesis is upon improved software engineering rather than performance requirements. However, some measures of the overheads introduced by using Kava were obtained. The performance of the execution speed of an application where enforcement code is in-lined manually and where the enforcement code is implemented by a metaobject bound to application objects using Kava was measured. Where enforcement code used multiple parameters to determine if access was allowed or denied, the metaobject implementation was on average 156% slower than in-lined code (341ms compared to 133ms). Where enforcement code made the decision using a single parameter, the metaobject was on average 7% slower than in-lined code (403ms compared to 377ms).

Differences in performance are related to the cost of the indirection introduced by the MLI and the cost of reifying context. In both cases the cost of indirection should be identical because the same code is in-lined. Therefore, the governing factor in this simple analysis was the cost of reification and reflection. The first fragment of enforcement code required reification and reflection of a String object and the second required reification of multiple primitive objects requiring expensive wrapping and unwrapping.

Initial performance results indicated the impact that reification and reflection has upon Kava's performance. Any unnecessary reification and reflection should be avoided. These findings have led to changes to Kava. Kava now allows hints to be passed via the binding specification that can be used to determine if reification or reflection is required. Many permission checks need neither and removing unnecessary code would improve performance.

As discussed in Section 7.3, future work must include a comprehensive performance analysis to help identify areas for improvements to performance. It should be possible to reduce the costs of this approach through the application of techniques such as partial evaluation[Masuhara, Matsuoka, Asai & Yonezawa 1995] or lazy creation/reification of metaobjects [Maes 1987] as has been done with other reflective languages.

3.2.11 Example: Redefining Field Access

The following example provides a flavour of how Kava uses MLIs to bring object behaviours under the control of a meta level. In the following example, the PutField behaviour is brought under control of the meta level and to avoid requiring the reader to parse the byte code, this is presented using the Java language.

Consider the field access shown below[2]:

```
myGreeting = "Kia␣ora␣te␣ao";
```

After Kava is used to bind a metaobject to the class containing the field access, the field access is rewritten at byte code level as shown in Listing 3.9.

In Listing 3.9 the act of updating a field is reified and handled at the meta level, which requires two MLIs, one before and one after the actual field access. Code is added directly

[2]"Kia ora te ao" can be loosely translated as "Hello World".

Listing 3.9: A rewritten field access.

```
my$Context.clear();                                          1
ContextBuilder.pushValue(my$Context, "Kia␣oro␣te␣ao");       2
ContextBuilder.pushTarget(my$Context, this);                 3
ContextBuilder.pushTarget(my$Context, "meta-parameter");     4
this.get$meta().beforePutField(my$Context);                  5
if (my$Context.getContinue()) {                              6
 myGreeting = (String)my$Context.getValue();                 7
 this.get$meta().afterPutField(my$Context);                  8
}                                                            9
```

before and after the field assignment instruction (PUTFIELD in byte code). At runtime the following takes place:

1 The current context object associated with the current invocation is cleared. This object is reused to avoid initialising a new object for each MLI.

2 Prior to execution of the instruction, the stack has a reference to the object instance encapsulating the field and the value to be stored. The value to be stored is popped off the stack and stored in a context object making it available to the metaobject when the metaobject methods are invoked. Note that it is not stored directly. Rather a helper class (ContextBuilder) is used that simplifies the task of writing interceptions. The helper class is written using the Java language and then called by inserting the byte code instructions for invoking the necessary method. All helper class methods are static and should this should lead to the Java JIT in-lining them.

3 The target is popped off the stack and stored in the context object.

4 Meta parameters associated with the MLI are added to the context. This enables metaobject behaviour to be parameterised and specialised to a particular application.

5 The beforePutField() metaobject method of the metaobject associated with the object is invoked causing a switch to the meta level.

6 Because the meta level can override the base-level behaviour, this step determines if the base-level behaviour is allowed to take place. To override the base-level behaviour, the meta-level programmer invokes my$Context.overrideBase() before returning the base level. This will cause my$Context.getContinue() to return **false** thereby causing steps 7–8 to be skipped.

7–8 These steps are executed only if the base-level behaviour has not been overridden. Step 7 both restores the stack and invokes the original base-level behaviour. Step 8 invokes the afterPutField() metaobject method bound to the base-level behaviour.

3.3 Evaluation against Goals

This section reviews whether Kava meets the goals defined in Section 2.7.

3.3.1 Enforcement upon Compiled Code

Kava exploits Java's loading and dynamic linking architecture to allow interception of class loading. A byte code rewriting toolkit is used to insert MLIs into classes before linking. The MLIs redirect control at runtime to metaobjects associated with objects through a binding specification. The binding specification is specified in a separate file because lack of source code and a desire for a clean separation mean code annotation cannot be used. This allows Kava to be used with compiled code which is discussed in Section 3.2.1.

3.3.2 Control Access to Application, Library and Operating System Resources

Application and library code both reside in user-level space and are loaded using Java's class loader. However, Java prevents direct modification of its core classes preventing metaobjects from being bound to objects implementing library services. Therefore Kava not only intercepts invocation of application methods at the callee-side but also intercepts invocations **made** by applications. This allows metaobjects to perform access checking before a library is invoked. This is discussed in Section 3.1.2.

All operating system resources are invoked via a Java library. This means that the controls applied to invocation of Java libraries are able to control access to operating system resources.

3.3.3 Clean Separation of Concerns

In addition to the separation of concerns provided by a MOP, Kava promotes reusability of metaobjects by allowing their parameterisation at the time they are bound to base-level objects. This allows specialisation beyond what is provided by inheritance. An aspect of the clean separation of concerns is how easily a set of metaobjects can be reused with a new application. Traditionally, reuse of metaobjects is achieved through specialisation by inheritance. Kava also allows specialisation through parameterisation. Meta parameters can be associated with MLIs via the binding specification. This allows generic metaobjects to be developed that are tailored to a particular application object by declaring meta parameters to be passed to the metaobject when a MLI is activated at runtime. This is discussed in Section 3.1.5.

3.3.4 Least Privilege

Kava provides least privilege because each object can be bound to a metaobject and the metaobject can redefine the semantics of all operations listed as part of the object's interface. Therefore all possible operations that may be executed by any code can be brought under the control of the meta level. However, Java allows declaration of **static** methods that are part of classes rather than objects and therefore these need to be controlled as well as ordinary methods. Kava handles static methods (and fields) by binding metaobjects to instances of class objects as well as objects, this is discussed in Section 3.2.8. Also, not all possible operations are declared as part of an object's interface. For example, field access does not need to be via a method belonging to the encapsulating object. Therefore the Kava metaobject protocol provides metaobject methods allowing these operations to be reified and handled at the meta level. This is discussed in Section 3.1.2.

3.3.5 Complete Mediation

This section discusses how the principle of complete mediation is provided by Kava. The principle of complete mediation means that access to every object must be checked for authority. This requires that the enforcement mechanism's integrity is not compromised. In the context of Kava this means ensuring that the infrastructure, meta-level interceptions and metaobjects are not compromised either at or during runtime.

The argument for Kava providing complete mediation is presented Section 3.2.9. The argument can be summarised as follows:

- It is assumed that the Java Runtime Environment, the Kava implementation, the enforcement metaobjects, the binding specification and the Java access control policies are trusted.
- It is assumed that Kava always has access to classes being loaded in the Java VM so it can add MLIs as required.
- Once MLIs have been added to a class, it is assumed that the MLIs or the metaobjects cannot be tampered with. This is achieved by applying the techniques described in Section 2.5.1.
- Attackers cannot confuse Kava's static analysis by subclassing protected objects. This requires Kava's design to take account of how Java implements inheritance.

3.3.6 Economy of Mechanism

The Kava metaobject protocol provides improved economy of mechanism compared to either source code or runtime MOPs. Source code MOPs require re-implementation of the compiler to allow source code to be pre-processed and bindings added. This is a more complex task than rewriting byte code because byte code has a more constrained semantics than source code. Runtime MOPs require changes to the Java VM. Again, this is a more complex task than rewriting byte code and requires revalidation of the entire runtime environment.

3.4 Summary

This chapter provided an overview of the design and implementation of Kava before evaluating Kava against the goals from the previous chapter. The next three chapters motivate the use of Kava and address the goal of evaluating the use of a loadtime metaobject protocol for security engineering.

Chapter 4

Case Study One: Standalone Application

This chapter demonstrates that using a loadtime metaobject protocol to enforce Java security policies provides a better separation of concerns than using conventional object-oriented techniques.

Section 4.1 introduces the application that is the subject of the case study and the scenario for its use. Section 4.2 describe a conventional object-oriented approach to enforcement and Section 4.3 describes using Kava, a loadtime metaobject approach. Section 4.4 identifies qualitative improvements to the separation of concerns gained by using Kava.

4.1 Overview

The case study uses a third-party Java standalone application developed by Jason von Nieda, called Lirc [von Nieda 2001]. Lirc is an IRC (Internet Relay Chat) client. The experiments were performed with version 1.0 of Lirc, which is licensed under the GNU General Public Licence agreement. A benefit of this licensing arrangement is the source code is available.

The scenario is an organisation that provides support for one of its products wants to permit their technical staff to provide support to customers via an IRC channel. For this purpose the staff are given permission to run Lirc on their own workstations. However, the organisation must impose upon staff the following local access control policy:

1. Users of Lirc should only be allowed to join specifically named channels, for example

"#help",

2. Users of Lirc should have resource limits placed on their network usage.

The organisation's security officer wishes to use the Java security architecture to enforce these policies upon users of Lirc. There are two types of resources subject to the access control policy: application resources and system resources. Resources of the first type are implemented by the application itself. Resources of the second type are implemented by Java system libraries. For the purposes of this case study, the access control policy is assumed to be specified using a standard Java policy file.

4.2 Conventional Object-oriented Security

This section describes the conventional object-oriented approach to enforcing access control policies upon application and system resources using the Java security architecture. Access to both types of resource is controlled by manually placing enforcement code within the Lirc application.

4.2.1 Application Resources

In this section the conventional approach to using the Java security architecture to control access to named IRC channels by manually placing enforcement code within the application is described. Unlike system resources provided by Java core libraries there are no standard Java permissions for use in granting rights to access IRC channels. Therefore the following steps for customising the Java security architecture in order to support such a policy [Gong 1999] must be followed:

1. Define a permission class.

2. Grant permissions.

3. Place permission checks within the code implementing the resource.

The rest of this section explains each of these steps in turn.

Define a ChannelPermission

First, a permission class representing the right to access a permission must be defined. Generally, a new permission class should subclass either the java.security.Permission class or the java.security.BasicPermission class. Subclassing from Permission allows permissions that have names and actions to be defined. For example, java.net.SocketPermission extends java.security.Permission with a constructor that requires a host specification (hostname or IP address and a port range) and a set of "actions" specifying ways to connect to the host. To control what rights that an application has over an IRC channel, a class example.lirc.ChannelPermission is defined that subclasses java.security.Permission and has a constructor that requires the name of an IRC channel and a set of actions specifying what rights a user has over that channel.

Listing 4.1: Defining a channel permission by declaring a new class

```
package example.lirc;
public final class ChannelPermission extends
  java.security.Permission {
    public ChannelPermission(String channel,
    String actions) {
      super(channel);
      this.actions=actions;
    }
    public boolean equals(Object obj) {...}
    public int hashCode() {...}
    public String getActions() {return actions;}
    public boolean implies(Permission permission) {...}
   ...
}
```

The implementation of the new permission is shown in Listing 4.1. For reasons of space, only the constructor and not the other methods, such as equals or implies that must also be implemented are shown. The implies method is used to capture the notion that the granting of one permission implies the granting of another permission.

Grant Access to a Channel

Once the permission class has been implemented, then the permission must be granted by creating a Java security policy file on the host where the application will be used. An example policy file granting the Lirc application right of access to the "#help" IRC channel is shown in Listing 4.2. The application is code-signed by an internal developer to ensure that only that application is granted this right. For the purposes of this case study, the internal developer is identified by the name "Bob". There should be an X.509 certificate in the local keystore (in this case, "teststore") that associates "Bob" with a public key that can be used to check the application's signature. The policy file must be protected against tampering by users so operating system rights must be used to make it immutable by users. In a Unix environment this can be achieved by the system administrator making the file read-only.

Listing 4.2: Granting access to join a specific channel

```
keystore "teststore";
grant signedBy "Bob" {
  permission example.lirc.ChannelPermission
    "#help", "join";
};
```

Place Permission Checks in the Application

Finally, the enforcement code shown in Listing 4.3 must be added into the application just before a channel "join" action. By inspection of the source code we determined that the channel "join" action for the application resource "IRC Channel" is implemented in the method createChannel() of the class Lirc. If no source code is available then either a debugger could be used to determine what methods are executed when a user joins a channel or the byte code could have been inspected.

The enforcement code consists of a calling the SecurityManager method checkPermission() to check if Lirc has been granted permission to "join" the IRC channel represented by the argument channel. The method returns silently if access is granted, otherwise an AccessControlException exception is raised. Because this is a runtime exception, it does not need to be declared in the method signature and so the exception is propagated upwards through the call stack until either a handler is found or the thread terminates with an uncaught exception.

Listing 4.3: Explicit check for permission to create a channel.

```
public void createChannel(String channel) {
  SecurityManager sm =
   System.getSecurityManager();
  if (sm != null) {
    sm.checkPermission(new example.lirc.ChannelPermission(channel,
      "join"));
  }
  ...
}
```

4.2.2 System Resources

Placing resource limits on network usage is an example of a dynamic access control policy for ensuring availability [Millen 1992]. Availability is guaranteed by limiting the total number of bytes an application may transmit across the network. Java represents access to the network as an OutputStream object created by opening a network socket.

The conventional object-oriented technique for enforcing access control upon system resources is to place enforcement code within the libraries implementing the resource. As discussed in Section 2.4.1, Java applies this technique to control access to system resources. Unfortunately, Java provides neither permissions nor enforcement code within its system libraries that allows a dynamic access control policy to be expressed and enforced.

The problem here is that the enforcement for access of system resources is not fine-grained enough for this application-specific policy. Instead the security enforcement technique described in Section 4.2.1 must also be applied to controlling Lirc's access to the network. The rest of this section describes how this is achieved.

Define a WritePermission

The right to write a certain number of bytes using a resource is represented by the permission class WritePermission. This is similar to the permission defined to control access to an IRC channel. However, unlike ChannelPermission, access is not a binary decision. What is required is a mechanism that allows the AccessController to determine whether holding a permission to write x bytes implies that a request to write x or fewer bytes is allowed. Note that the permission to write x bytes is immutable and granted in the Java policy file.

The java.security.Permission class provides an abstract implies() method for comparing granted permissions with requests for permissions that can be overridden by any subclass that needs to define itself in relationship to other permissions. For example, holding one type of permission might imply a set of other permissions. In this case it is used to define a relationship between granting the right to write a certain quota of bytes with a request to check if the current number of bytes has been exceeded or not.

Listing 4.4, shows the implementation of implies() for WritePermission. The permission being checked at runtime is only implied as being held if it is a WritePermission granting access to the network and the quota of bytes written so far has not been exceeded. When determining if the permission being checked is a WritePermission, the name is taken into account. This is to allow the WritePermission to be used to control access to streams other than just the network. This provides some future compatibility, because the same permission could be used to impose quotas on access to files as well as networks in the future.

Grant a Write Limit

The next step is to define a Java policy file as shown in Listing 4.5. This policy file grants the application Lirc the right to write a maximum of 1.000.000 bytes to the network is shown below. Again, we assume the application developer is "Bob" and the system administrator should make the policy file write protected to prevent users increasing their own quotas.

Listing 4.4: Check for implication of permissions in class WritePermission.

```
public final class WritePermission extends java.security.Permission
{
  // check if permission is implied
  public boolean implies(Permission p) {
    if (!(p instanceof WritePermission)) {
      return false;
    }
    WritePermission wp = (WritePermission)p;
    if (!getName().equals(wp.getName())) {
      return false;
    } else if (getNumBytes() < (wp.getNumBytes()) {
        return false;
    }
  return true;
  }

  // convert the string representation of number of bytes to long
  private long getNumBytes() {
    ...
  }
  ...
}
```

Listing 4.5: Java policy limiting Lirc's access to the network.

```
keystore "teststore";
grant signedBy "Bob" {
  permission example.general.WritePermission "network", "1000000";
};
```

Place Permission Checks in the Application

The final step is to identify where the application invokes the system resource and insert enforcement code. Writing to a socket is achieved by invoking one of two write methods on an output stream associated with a socket. Again by inspecting the source code, the places where the application writes bytes to a socket connection are located and permission checks added as shown in Listing 4.6.

Listing 4.6: Check for permission to write to the network.

```
SecurityManager sm = System.getSecurityManager();
if (sm != null) {
 Lirc.currBytes += requestedBytes;
 sm.checkPermission(new WritePermission("network", Lirc.currBytes));
}
return true;
```

Here, the number of bytes to be written must be added to a running total of bytes written by the application **as a whole**. Therefore a static field belonging to the application (the Lirc class), currBytes, is used. This is updated by the number of bytes to be written. This running total is used to create an instance of a WritePermission representing a request to continue writing to the network. The implies() method will return silently if the quota has not been exceeded, otherwise a runtime exception will be raised and the application will halt.

The use of a static field to maintain the total bytes written to the network would allow a user to defeat security enforcement by simply restarting the application afresh when the user's quota is reached. A better solution would be a centralised quota manager on initialisation and update it as the application executes. This would prevent this loophole being used and also allow a quota to be imposed over all instances of Lirc used within the organisation.

4.3 Loadtime MOP

The conventional approaches discussed above suffer from an intertwining of application and enforcement code. Changing one requires changing the other, for example should a new version of Lirc be released then the enforcement code would have to be added again manually. Also, the process is laborious and error-prone. It relies upon manual inspection of application code and it would be quite easy to miss out a necessary piece of enforcement code. Overall the conventional approach leads to increased development time, decreased system flexibility and comprehensibility.

An alternative security engineering technique is to use a loadtime MOP, in particular Kava because it meets the other criteria for least privilege, complete mediation and economy of mechanism. The aim is to reuse the permissions, enforcement code and policies instead of replacing them.

This section shows how Kava can be used to achieve this by implementing the same policies described above using the Java security architecture but instead of checks being manually added they are implemented by metaobjects. The binding configuration file specifies where the access control checks should be enforced in the application and the security policies are specified using a Java security policy file. Section 4.3.1 describes how this can be applied to an application resource while Section 4.3.2 describes how this can be applied to a system resource.

4.3.1 Application Resources

To control who can join an IRC channel when using the Lirc application, Kava must be used to ensure that possession of ChannelPermission is checked before the base-level application attempts to access an IRC channel. This requires that the meta-level programmer must implement a metaobject to encapsulate this enforcement code and bind it to the right base-level object. Additionally, the Java policy file must be modified to take into account the use of Kava and the metaobjects. Each of these steps is detailed below.

Define a Metaobject

The ChannelEnforcementMetaobject class redefines beforeExecuteMethod provided by the standard Kava metaobject class, thus ensuring that the check is performed before the method is executed. The code for the actual security check is similar to the code shown in Listing 4.3. However, the enforcement code shown earlier refers to the variable representing the channel name whereas this code takes the first argument from the execution context and uses it to construct the permission.

Listing 4.7: ChannelEnforcementMetaObject controlling access to IRC channels.

```
package example.lirc;
public class ChannelEnforcementMetaObject
   extends kava.metaobjects.MetaObject {
   ...
   public void
     beforeExecuteMethod(IExecuteContext context) {
       String joinChannel
         = (String)context.getArgs()[0];
       SecurityManager sm =
         System.getSecurityManager();
       if (sm != null) {
         sm.checkPermission( new ChannelPermission(joinChannel,
         "join"));
       }
     }
}
```

Specify a Binding

The meta-level programmer must define a binding specification that ensures the ChannelEnforcementMetaObject is bound correctly at loadtime. Instead of providing the XML version of the binding specification, an English version is shown for readability below.

> "Bind the metaobject class ChannelEnforcementMetaObject to the base-level class Lirc and intercept calls to the method createChannel."

Modify the Policy File

Finally, the meta-level programmer must write a new policy file. The policy file must grant access to all the code involved in the permission checking at runtime because Java requires that the permissions checked at the meta level must be granted to every protection domain that is on the call stack when the check is performed. The policy file must therefore grant appropriate permissions to the application itself plus the metaobject and Kava implementation.

The new policy file is shown in Listing 4.8. Note the inclusion of principals representing the application developer (named "Bob"), the meta-level programmer (named "Alice") and the developer of Kava itself (named "Kava"). The policy file is structured in three parts: (1) "Bob" is granted the same rights as before; (2) "Alice" is granted the same rights as "Bob" as well as the right to allow any metaobjects signed by "Alice" to be initialised ("initialiseMetaobjectClass"); (3) "Kava" is granted the same rights as "Alice" plus any rights it needs such as the right to install an application-level class loader.

4.3.2 System Resources

To control resource usage by the Lirc application using Kava, the same steps as taken in Section 4.3.1 must be applied. However, unlike the previous section this requires the use of a metaobject that redefines two base-level behaviours: method invocation and method execution. The metaobject is then bound to the application classes that create and write to network sockets.

Define a Metaobject

Writing to a socket is achieved by invoking one of the write methods on an OutputStream associated with a socket and Kava ensures that a permission check is made before any of these methods are invoked. The simplest approach would be to redefine the execution of the write methods. However, Kava cannot do this because the OutputStream is implemented as a system class and cannot be rewritten. However, this problem can be overcome by intercepting invocations of the write methods made by Lirc. Unlike the application-specific IRC

Listing 4.8: Policy file for enforcing channel access control using Kava.

```
keystore "teststore";
grant signedBy "Bob" {
  permission example.lirc.ChannelPermission
    "#help", "join";
};
grant signedBy "Alice" {
  permission example.lirc.ChannelPermission
    "#help", "join";
  permission kava.runtime.KavaPermission
    "initialiseMetaobjectClass";
};
grant signedBy "Kava" {
  permission example.lirc.ChannelPermission
    "#help", "join";
  permission kava.runtime.KavaPermission
    "initialiseMetaobjectClass";
  ...
  permissions required only by Kava
  ...
};
```

channel access policy, intercepting calls to standard system classes does not require knowledge of the semantics of the application that is being constrained because any application that attempts to communicate over a socket must use these standard methods.

Unfortunately, redefining all invocations of OutputStream write methods will not only affect writing to sockets but also writing to files and memory as well because these also create OutputStream instances for writing and reading bytes. Therefore, when determining whether to check permission to write a number of bytes the parent of the OutputStream must be taken into account to distinguish those used to access sockets from other types. Unfortunately, this is not a straightforward task because an ObjectStream method or field that provides a reference to its parent does not exist. The solution is to implement extra meta-level code using a table to track the creation of OutputStream instances by sockets. When an invocation of an OutputStream write method takes place the object identity of the target object can be checked against the table of known socket-created output streams. The code implementing this is shown in Listing 4.9. The metaobject method afterInvokeMethod watches for invocations of getOutputStream on a socket. The returned value is stored in a hashtable along with a reference to the socket that created it. This allows us to watch for the creation of output streams and determine the socket that created it.

Listing 4.9: Meta-level enforcement code tracking the creation of OutputStream instances by sockets.

```
public class WriteEnforcementMetaObject extends
  kava.metaobjects.MetaObject {
  // keep track of the output streams created by the socket
  private static Hashtable ostreams = new Hashtable();
  // keep track of the current number of bytes private
  static long currBytes = 0;
  ...
  public void afterInvokeMethod(IInvokeContext context) {
    if (context.getMetaParam().equals("open␣output␣stream")) {
      WriteEnforcementMetaObject.ostreams.
      put(context.getReturnValue(), context.getTarget());
    }
  }
}
```

A check of the security policy is performed before any write method belonging to an OutputStream object is invoked. As described earlier, this only takes place if the OutputStream was created by a socket. Listing 4.10 shows the enforcement code encapsulated within the metaobject. The details of the enforcement is identical to Listing 4.6 except that the number of bytes to be written is extracted from the reified context.

Listing 4.10: Meta-level enforcement code checking whether Lirc has exceeded its network quota.

```
public class WriteEnforcementMetaObject extends
  kava.metaobjects.MetaObject {
  ...
  public boolean beforeInvokeMethod(IInvokeContext context) {
    Object target = context.getTarget();
    if (isSocketOStream(target)) {
      Object[] args = context.getArgs();
      if (context.getMetaParam().equals("write␣bytes")) {
        currBytes += ((byte[])args[0]).length;
      } else if (context.getMetaParam().equals("write␣byte")) {
        currBytes += 1;
      }
      SecurityManager sm = System.getSecurityManager();
      if (sm != null) {
        sm.checkPermission(
          new WritePermission("network", Long.toString(currBytes)));
      }
      return true;
    }

    // is the output stream in the table?
    protected static boolean
    isSocketOStream(Object ostream) {
      return ostreams.containsKey(ostream);
    }
  }
}
```

Specify a Binding

Again, the meta-level programmer must define a binding specification that ensures the WriteEnforcementMetaObject is bound to the correct base-level objects at loadtime. In this case, the binding is to objects creating sockets and to objects invoking the write methods of the ObjectStream objects. Unlike a manual approach this can be done automatically because Kava can statically analyse the application code. A high-level view of the binding specification is shown below:

> "Bind the metaobject class WriteEnforcementMetaObject to any application base-level class, where either:
>
> 1. the method getOutputStream() is invoked on a class Socket (associate this with the meta parameter "open output stream");
> 2. the method write(byte[] ...) is invoked on a class OutputStream (associate this with the meta parameter "write bytes");
> 3. the method write(byte ...) is invoked on a class OutputStream (associate this with the meta parameter "write byte")."

Modify the Policy File

Finally, the Java security policy must be specified. As before the policy file must allocate the appropriate rights to "Bob", "Alice" and "Kava". The policy file is shown in Listing 4.11.

4.4 Improvements to the Separation of Concerns

The conventional security engineering technique of placing enforcement code within the application manually led to an intertwining of enforcement and application code. This case study showed, for two different types of resources and access control policies, that the enforcement code could be modularised and implemented as metaobjects. This leads to the following benefits:

- No access to application source code is required.

Listing 4.11: Policy file for enforcing network quota using Kava.

```
keystore "teststore";
grant signedBy "Bob" {
  permission example.general.WritePermission
    "network", "1000000";
};
grant signedBy "Alice" {
  permission kava.runtime.KavaPermission
    "initialiseMetaobjectClass";
  permission example.general.WritePermission
    "network", "1000000";
};
grant signedBy "Kava" {
  permission kava.runtime.KavaPermission
    "initialiseMetaobjectClass";
  permission example.general.WritePermission
    "network", "1000000";
  ...
  permissions required by Kava
  ...
};
```

- The application code can now be read and modified independently of the enforcement code. The enforcement code can also now be read and modified without requiring changes to the application code.
- Should the application code change, there is no need to reapply enforcement code manually. Kava will be able to insert the necessary MLIs automatically. Automatic rather than manual changes reduce the chances of introducing errors into the implementation.
- The metaobjects can be reused with a new application; all that is required is to modify the Java policy file and possibly the binding specification. Whether the binding specification changes depends upon the generality of the policy. The IRC channel policy is closely tied to the application and would require rereading the application specification and making appropriate changes. However, the network quota policy is application-independent requiring no changes to the binding specification.

4.5 Summary

This chapter demonstrates that using a loadtime metaobject protocol provides a cleaner separation of concerns than manually placing enforcement code within applications. The next chapter demonstrates that using a loadtime metaobject protocol provides a cleaner separation of concerns than placing enforcement within libraries or proxies.

Chapter 5

Case Study Two: Distributed Application

This chapter demonstrates that using a loadtime metaobject protocol to enforce access control policies upon a distributed application provides a better separation of concerns than using conventional object-oriented techniques such as modifying library code, inheritance or proxies.

Section 5.1 introduces the System K distributed application that is the subject of the case study and the scenario for its use. Section 5.2 describes a conventional object-oriented approach to enforcement and Section 5.3 describes using Kava, a loadtime metaobject approach. Section 5.4 identifies qualitative improvements to the separation of concerns gained by using Kava. Finally, Section 5.5 summarises this chapter.

5.1 Overview

This section introduces the third-party application, referred to as System K[1] developed as a security demonstrator with the the objective of illustrating a security architecture using a distributed, heterogeneous, hypertext document server written in Java.

The functional part of System K is a Computer Aided Software Engineering (CASE)

[1]The full details of System K are not available because the system was provided under a confidentiality agreement

tool implemented as a hypertext document server. Users manipulate CASE models represented as diagrams via viewers. A viewer allows a user to access a remote database and download a model for local editing. New elements and links to other models can be added or modified locally. Links may also 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.

System K requires a security architecture able to enforce access control policies upon client-server interactions (between Viewers and Databases) and weakly-mobile code (Models are downloaded to the same host as the viewer and edited locally). For the purposes of this case study, it is assumed that the local host is trusted so that security can be enforced locally and that the remote host is also trusted. However, the network is assumed to be untrusted.

The functional part of System K comprises sixty-four Java classes, and approximately 4,000 lines of code. At runtime, System K consists of multiple viewers interacting with multiple remote databases.

The security architecture used by the demonstrator is based on work introduced by Bull and his colleagues [Bull, Gong & Sollins 1992] and has the characteristic that each object is responsible for its own security – hence the model is termed **self-defence**. 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 authorisation servers. Additionally, each object may delegate authentication tasks to authentication servers. This provides a global view, even though security decisions are made on a per-object basis. It also provides support for heterogeneous policies in the same system because each object may either use its own policy to make an access decision or delegate the decision to an appropriate authorisation server.

Security for a distributed system relies upon authentication of communicating parties whilst protecting the integrity and/or confidentiality of communications. Like Kerberos [Neuman & Ts'o 1994], System K provides a trusted third party to assist clients in setting up a secure channel with servers. A trusted authentication server maintains an association between each principal's unique identifier and a unique symmetric key used for authentication and communication between the authentication server and the principal.

The symmetric key is established using a modified version of the Otway-Rees protocol [Otway & Rees 1987] and communications are encrypted using the IDEA algorithm [Schneier 1995].

Again like Kerberos, System K allows delegation of authorisation decisions to trusted third-parties to allow for scalability and ease of administration. There are several authorisation servers, one for each policy type. This approach provides a global view for security and allows management of rights and clearances by security officers.

Each secured object in System K may be subject to access control policies that attempt to ensure the integrity of objects and prevent undesirable information flow. Originator-Control (ORCON) [McCollum, Messing & Notargiacomo 1990] access control policies are used to ensure that integrity is maintained, these are essentially variants on access control lists where the owner of an object can decide which principals may access the object. Undesirable information flow is implemented by an access control policy based upon the Bell-laPadula multilevel security policy [Bell & LaPadula 1976].

5.2 Conventional Object-oriented Security

The following sections describe how the self-defence security architecture was implemented in System K using standard object-oriented techniques such as modifying system libraries, using inheritance and the proxy design pattern [Gamma et al. 1995]. The designers of System K chose these techniques because they offered a better separation of concerns than manually placing enforcement code within applications.

5.2.1 Overview

The following sections describe how secure RMI, association of security state with secure objects and authorisation have been implemented by placing enforcement code within libraries, within superclasses and within proxies.

5.2.2 Secure RMI

System K places enforcement code within system libraries to implement authentication, integrity, confidentiality and initial authorisation checks for remote invocations and requires adherence to a programming convention that differs from the standard conventions when using Java RMI.

A modified RMI protocol stack is created that mutually authenticates clients and servers, generates session keys, manages the passing of security parameters with invocations and transparently encrypts and decrypts byte streams encoding remote invocations. Classes representing remote invocations, remote references and the UnicastObject have been modified to include extra code and members to support secure RMI.

The changes are almost transparent to the developer of a server because the modified protocol stack automatically encrypts and decrypts the remote method invocations with the current session key to provide authenticity, integrity and confidentiality. Also from the point-of-view of a client, the changes are transparent because the modified protocol stack automatically encrypts and decrypts the remote method invocations to provide authenticity, integrity and confidentiality.

Had System K been developed with JDK1.2 or greater, Secure RMI over SSL [Java Team 1996-1999] could have been used to provide authentication, integrity and confidentiality although it would require some changes to the implementation of servers.

5.2.3 Security State

Each client and server in System K has security-related behaviour and state associated with it, namely a unique identifier, a secret key for each principal and active labels used to implement authorisation. This is achieved by requiring these classes to extend the SecureObject class as shown in Figure 5.1.

Principal Identifier

The principal identifier is a string that must be globally unique and persistent. The case study assumes that this can be achieved by issuing a smartcard with a global identifier. The identifier is read from a smartcard at the time of object creation and associates the

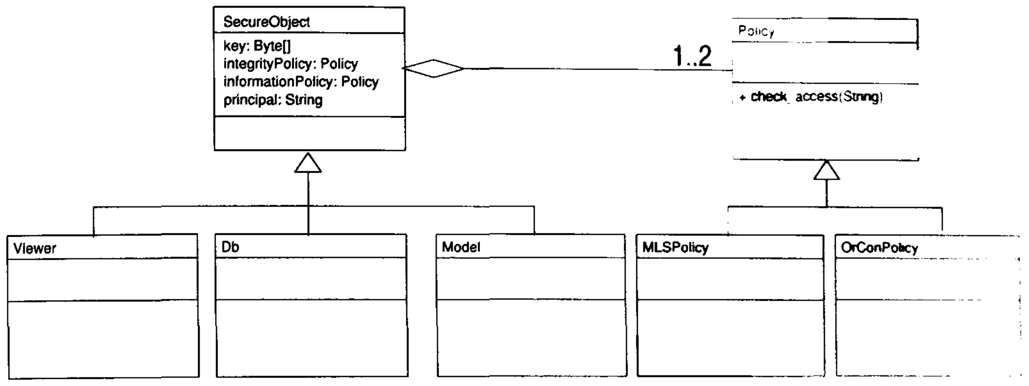

Figure 5.1: Class diagram for self-defence security architecture. Two classes under the control of the security architecture are shown: Viewer and Db. These extend SecureObject which maintains security-related meta data. Each SecureObject contains pointers to an information flow policy and an integrity 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 authorised.

object with its creator. This provides each object such as a server or mobile objects with a principal identifier allowing authorisation servers to model access control policies based upon the identity of a principal. For example, all objects created by Alice may access all objects created by Bob.

The principal identifier is passed with all remote method invocations sent by clients or servers acting as clients. It is not added to invocations by the RMI protocol stack. Instead a client-side proxy is used to add the principal identifier to invocations as discussed in Section 5.2.4.

Secret Key

A secret key associated with the user is read from the user's smartcard at the time of object creation. The secret key is a symmetric key shared with an authentication server that acts as a trusted third-party between the client and servers. The RMI protocol stack uses the key in a mutual authentication and key establishment protocol.

Active Labels

Active labels are a concept particular to System K. They share features similar to systems such as those specified in the Orange book [US Department of Defense 1985], where each object protected by the system reference monitor has a label. But unlike a traditional system, where the interpretation of the labels is fixed within the infrastructure and are simply state, active labels encapsulate both state and logic for interpreting their meaning. This is achieved by requiring each label to inherit from a Policy class that defines a check() method with a string argument representing the name of a principal. The check method determines whether the object with the particular label will allow access to a named principal.

Self-defending objects encapsulate active labels and whenever an object's method is invoked it implements authorisation policies by calling the check methods of its active labels. Some security policies such as MLS require the labels associated with objects to change following a policy check, active labels implementing these types of policies require implementation of an add() method that allows a new label to be created by adding the current label to another label.

System K has implemented two types of policies that may be used as active labels: OrConPolicy and MLSPolicy representing Originator Control and Multilevel Security respectively. Both policy classes delegate their decisions to remote servers – a GroupId server in the case of OrConPolicy and MLSServer in the case of MLSPolicy. Each server has a maintenance interface that allows a security officer to implement a particular organisation's security policy. Servers must be protected so that only authorised users can change the current policy in effect, achieved by applying the self-defence architecture recursively.

5.2.4 Authorisation

System K uses proxies to implement the Caretaker pattern discussed in Section 2.4.1. For each server that has security policies enforced upon it there is a client-side and server-side proxy. The client-side proxy implements transmission via RMI of principal identity, and the server-side proxy acts as a PEP enforcing security policies on servers. The original implementation of System K required a proxy to be generated manually for each class that is the target of an authorisation 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 proxies, but this approach was never implemented.

The normal flow of control for a secure policy-controlled invocation on a database server is shown in Figure 5.2. Note, it is assumed that a reference to the database's security proxy has already been obtained by invoking the open() method of the database's manager object. The diagram also shows the proxies and skeleton classes automatically generated by the Java RMI compiler that provide distribution and implement authenticity, integrity and confidentiality.

The Viewer object invokes a method on a remote Db object by invoking the method on a client-side security proxy (Db_CliStub). This proxy adds the principal identifier to the invocation and then invokes the method on the client-side RMI proxy (Db_SvrStub_Stub). The RMI stub marshals the method, the arguments and the principal's identity and encrypts it using a session key obtained during the initial interaction with the remote Java RMI skeleton for the server's security proxy (Db_SvrStub_Skel) before sending the encrypted method call. The RMI skeleton decrypts the byte stream and unmarshals the method, arguments and principal. After unmarshalling the RMI skeleton invokes the server's security proxy (Db_SvrStub) that enforces any security policies before passing the method call on to the Db server 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.

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, 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, the security state of the server may need to be updated. This decision is hard-coded into the Db_SvrStub. If an update is required because the execution of the method results in a change to the server state, a new information flow policy is generated by adding the existing policy to one that is created using the principal's identity. Integrity policies such as OrConPolicy require only that an access check is performed, whereas information-flow policies such as MLSPolicy require both an access check and update to the server's security state.

The conventional implementation requires these checks and updates to be hard coded

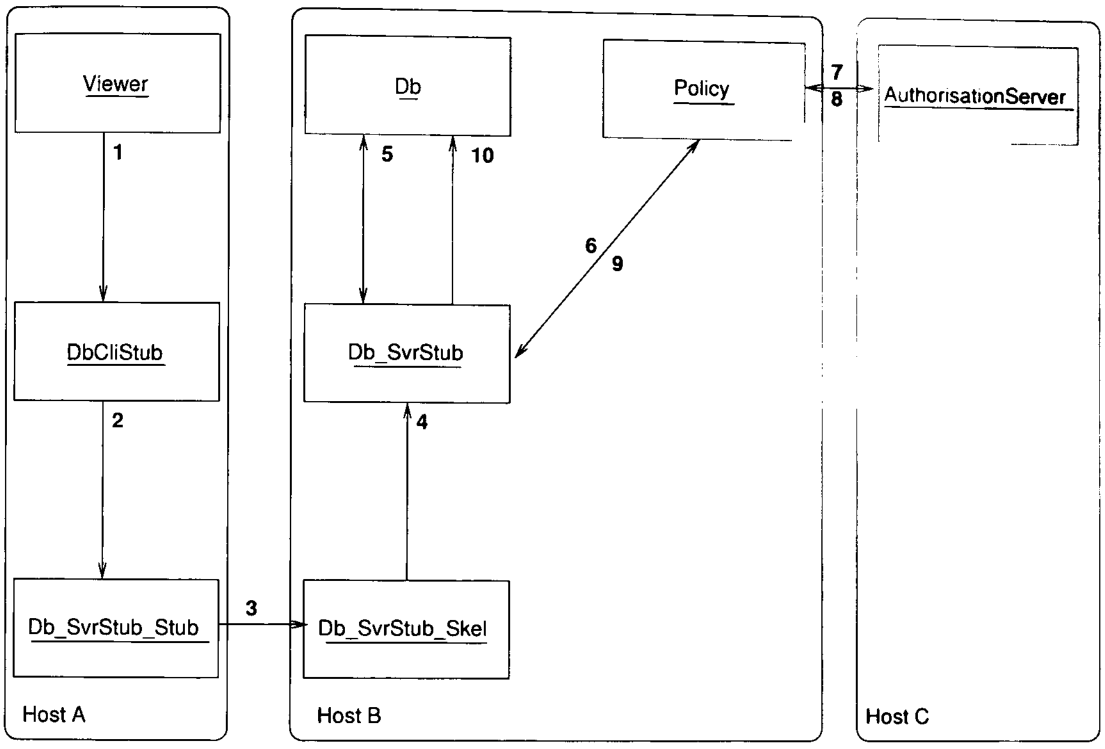

Figure 5.2: Collaboration diagram showing steps involved in a viewer invoking a method of a remote database under the self-defence security architecture. Step 1: Viewer object invokes a Db object method. Step 2: A DbCliStub proxy receives the invocation and adds the identity of the principal associated with the Viewer. The DbCliStub proxy is only responsible for access control, not with enforcing confidentiality, it invokes the DbSvrStub_Stub proxy. Step 3: The DbSvrStub_Stub proxy encrypts the invocation and sends it across the network. Step 4: The Db_SvrStub_Skel receives the encrypted invocation and decrypts it before invoking the Db_SvrStub. Step 5: The Db_SvrStub retrieves a reference to the Policy object associated with the Db object. Step 6: The check_access method of the Policy object is invoked to determine if the method may be invoked. Step 7: The Policy object delegates the access check to the appropriate AuthorisationServer object. This may actually be a stub used to contact a remote authorisation server. Step 8: Either the access is allowed or disallowed. Step 9: The Policy object returns the decision. Step 10: Assuming that access is granted, the Db_SvrStub invokes the appropriate method of the Db object.

into the security proxy such as Db_SvrStub.

System K uses the same approach as used for protecting remote servers to protect Model objects that are downloaded for local editing. Although this allows the same infrastructure to be used for both local and distributed access control it leads to an overcomplicated implementation because security proxies must be downloaded with the Model objects. This requires that when a request is made to download a Model from a database, a security proxy is returned instead. The security proxy then downloads the real Model object and acts as a local proxy for it. Subsequently, when the Model object is to be uploaded, the security proxy ensures that the Model object is uploaded to the database and the proxy is garbage collected.

5.3 Loadtime MOP

This section describes how Kava can be used to implement access control policies without modifying system libraries or using inheritance or proxies. Section 5.3.1 provides an overview of the Kava implementation of access control and the different metaobjects used. Section 5.3.2 describes how a metaobject can be used to ensure that secure RMI is implemented and Section 5.3.4 describes how authorisation is implemented through a combination of access checking and passing information about principals across secure RMI.

5.3.1 Overview

This section outlines how the Kava metaobjects co-operate together to implement the self-defence security architecture. Figure 5.3 shows how the metaobjects co-operate to provide secure RMI between a Viewer object (the client) and a Db object (the server) under the control of access control policies. Two metaobjects cooperate in realising distributed access control: (1) The MetaAuthorisation metaobject bound to methods of classes that must be brought under the control of a security policy enforce security decisions; (2) The MetaAuthentication metaobject is bound to the marshalling and unmarshaling methods of the stubs and skeletons that implement RMI for server classes. A RMI method invocation goes

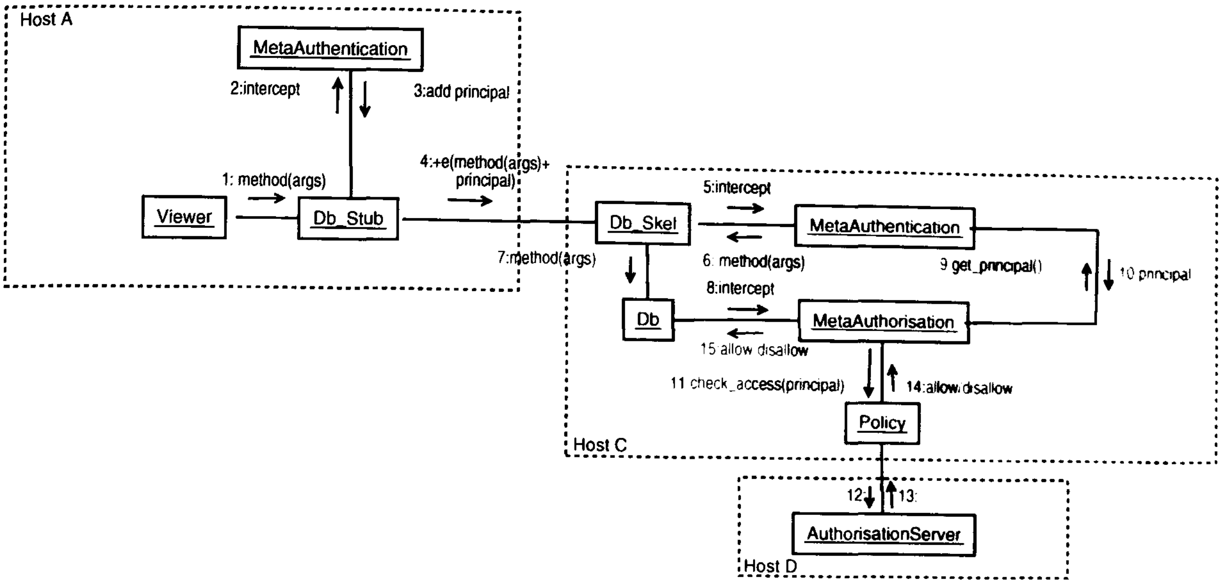

Figure 5.3: Collaboration diagram showing the steps involved in invoking a method of a remote database under the control of Kava metaobjects.

through the following steps:

1. 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 marshals the arguments, control is switched to the MetaAuthentication metaobject, which marshals the principal's identity. Control is returned to the Db_Stub, which then sends the byte stream to the remote skeleton (Db_Skel) across an encrypted socket connection. Note that secure communications are established as a result of the MetaConfidentiality metaobject's intervention during the startup of the remote Db object, and that the metaobject is not otherwise involved in the communication between the client and server objects.

2. On the server side, the Db_Skel unmarshals the arguments and invokes the appropriate Db method. The MetaAuthentication metaobject takes control and unmarshals the principal's identity which it stores. Control is returned to the Db_Skel and the appropriate Db method is invoked using the unmarshaled arguments.

3. If the method is specified as being under reflective control in the binding specification, its execution is reified and handled at the meta level by the MetaAuthorisation

metaobject.

4. If the execution of the method has been specified in the binding specification as being under the control of an integrity or information flow policy, the MetaAuthorisation metaobject invokes the check_access method of the policy object.

   (a) In the case of an integrity policy, invoking check_access may cause the access decision to be delegated to a remote authorisation server. If access is denied, a runtime exception is raised, and this is converted to a RemoteException and propagated all the way back to the Viewer object. Otherwise, the check_access method silently returns.

   (b) In the case of an information-flow policy, the check_access method is invoked as described above. However, if the method returns silently (indicating access is allowed), the information-flow policy associated with the server is updated by invoking the policy object's add method.

   (c) In the case that both types of policy apply, then both of the steps described above apply. As long as an exception is not raised due to an access control check, control will return to the base level and the method is executed.

Note that the diagram doesn't show the MetaConfidentiality metaobject that is bound to any class that registers a server object with the RMI registry. This metaobject ensures that a socket factory implementing SSL/TLS is used for RMI.

5.3.2 Secure RMI

The MetaConfidentiality metaobject ensures confidential communications between clients and servers. It uses standard Secure RMI facilities provided by JDK 1.2 [Sun Microsystems 2000] for securing communications and providing authentication rather than the infrastructure developed as part of System K. Using Secure RMI is a more portable solution making use of the Transport Level Security protocols [Dierks & Allen 1999] that have been widely tested.

The developers of System K did not use Secure RMI for two reasons. First, System K was developed using JDK 1.1 rather than the more recent development kits that can provide facilities for secure RMI. Second, developers wished to avoid US export restrictions upon the use of public key technology [Apache Week 1997] by using a key establishment protocol based upon oneway functions. These restrictions were lifted in September 1999.

The standard approach for using secure RMI is to ensure that a secure sockets layer (SSL) factory is used by the RMI protocol stack instead of a standard socket factory. Unfortunately, this cannot be done by setting a Java property in the version of JDK[2] used for the case study and requires extra code to be added to any client that is going to communicate securely with a server. To avoid changes to the application source code, we encapsulate this code in a metaobject class and bind it 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 metaobject is simply to intervene in the initialisation of the RMI layer, and that the metaobject does not otherwise intercept method invocations between user-level objects.

5.3.3 Security State

Instead of placing enforcement code relating to security state in a superclass, the MetaAuthorisation metaobject controls access to server methods and applies the information and integrity security policies as appropriate. As methods differ as to which security policies are applied, the binding specification file parameterises the metaobject 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 results 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 interaction with a principal.

[2]JDK1.2 was used.

5.3.4 Authorisation

Using a metaobject to perform authorisation avoids problems associated with using a separate proxy class for this purpose. As there is no proxy and no reference to be leaked, it is possible to 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 metaobjects are automatically downloaded with the base-level object because they are pointed to by the base-level object and defined to be serializable.

The MetaAuthentication metaobject implements the passing of a principal's identity from the client to the server with a remote invocation. The metaobject is designed to be bound to the server's RMI stub and skeleton because being bound to these objects exposes interfaces for marshalling and unmarshaling arguments that are passed with RMI method invocation. At the client side, the MetaAuthentication metaobject adds the principal's identity to the byte stream, representing the marshalled arguments, before it is sent over the encrypted socket connection to the skeleton, and at the server side it removes the principal from the byte stream before unmarshaling the arguments from the decrypted byte stream.

The current implementation must use JDK1.1-compliant proxies because the metaobject requires access to the byte stream representation of remote methods calls and JDK1.2 does not generate proxies at all. This makes this implementation technique dependent upon current features of Java. An improved approach is to use a generic interface that provides access to representations of remote-method invocations as byte streams. Even if this is not provided by Java, the metaobjects can be insulated from these dependencies by hiding the implementation details behind a fixed interface to the RMI protocol stack. In early versions this would be implemented as described here whilst later versions could take advantage of any opening up of the Java protocol stack.

5.4 Improvements to the Separation of Concerns

This section discusses how using Kava provided a better separation of concerns than the conventional techniques used in this chapter.

Before discussing the improvements on a case-by-case basis it is useful to reflect upon the reduction on code size gained by using Kava to secure System K rather than conventional object-oriented techniques. The Kava version reduced the size of application classes by between approximately 15% and 45% through the removal of explicit access checks, proxies and removal of code to update meta data.

The conventional and Kava versions of the System K codebase were normalised to reduce the impact of programmer style. Normalising the code involved removing all whitespace and comments from the code. This allowed the size of the codebases to be compared without worrying about different programmer styles with respect to code formatting or commenting. Additionally, because the focus of the case study was to see if the engineering of application code was improved, only the size of application code used to invoke either secure RMI or authorisation was compared. The code implementing the secure RMI protocol stacks was excluded from the counts because this is assumed to be system-level code and the authorisation servers were excluded because they are common to both the conventional and Kava versions of System K.

The rest of this section discusses the improvements gained by using Kava on a case-by-case basis.

5.4.1 Place Enforcement within Libraries

In this case study the RMI protocol stack was changed to implement authenticity, integrity and confidentiality. Although providing a good separation of concerns from an application point-of-view because it requires no changes to applications, in the context of the case study there are three main drawbacks to this approach:

- All servers must use the RMI protocol stack. This means that all server implementations, even those that do not require Secure RMI were forced to make use of it. It is true that this could have been avoided by adding options to standard RMI classes that could control which protocol implementation was used but it does point out the difficulties in engineering a solution that will fit all possible applications.

- Changing the RMI protocol stack was a non-portable solution requiring changes to the Java runtime environment. This may not be possible in all environments for good security reasons.
- Inserting enforcement code into the protocol stack may overcomplicate its design and lead to problems when it changes at a later stage.

Kava was used to address implementing Secure RMI by using standard facilities and invoking them by encapsulating the necessary code within a metaobject.

This addresses the first problem because the metaobject did not have to be bound to all server objects. Some server objects could make use of it to implement Secure RMI but other objects did not have to make use of it. This is a benefit of the clean separation of concerns provided by a loadtime MOP. The second problem is addressed because Kava doesn't require modification of system libraries due to its ability to redefine invocations by inserting MLIs into application code. The third problem is actually an indication that adding enforcement code leads to an intertwining of enforcement code and library code that makes maintainability difficult. Again, by modularising the changes to the RMI protocol stack within a metaobject protocol the separation of concerns is improved.

5.4.2 Place Enforcement within Proxies

As discussed in Section 5.2.4, System K makes use of the Caretaker pattern and implements it using proxies. Using proxies avoids changing the implementation of the proxied object. However, it does require writing additional classes and manually inserting code to update the active labels as required for the type of policy being implemented. The following describes how the problems described in Section 2.4.1 can be seen in this case study:

Self problem. Self-invocations within System K are not intercepted by the security proxy. Should an implementation error permit a method that should be subject to an authorisation check to be invokable via an unprotected method then authorisation would be bypassed.

Logical wrapping problem System K suffers from the logical wrapping problem because the architecture is complicated by the need to add extra code to support enforcement of access control on mobile objects as described in Section 5.2.4.

Confinement problem. Should a direct reference to a Db or Model object escape then the security proxies protecting them might be bypassed. This only occurs with a Db object if it implements a remote interface because it is only invoked using RMI. On the other hand, Model objects are downloaded into the local JVM so possessing an object reference allows its security proxy to be bypassed.

Interface gap problem. System K suffers from the interface gap that exists because of the introduction of an extra proxy class. Whenever the interface of an object that is the subject of security proxy changes, the security proxy must also be modified so it reflects the changes to the secured object. This complicates maintenance because two classes must be updated and points to a poor separation of concerns.

Essentially, using Kava avoids these problems because there is no longer any need to maintain a separate proxy. The role of proxy is played by a metaobject but this is not visible from the point of view of any client of the base-level object. The problems are addressed as follows:

Self problem. Self-invocations within System K are always intercepted by the metaobject. Therefore other methods that are not subject to an access control policy cannot be used to bypass protection.

Logical wrapping problem Avoiding the use of a separate proxy removes the requirement to maintain extra code to support enforcement of access control.

Confinement problem. Should a direct reference be made to a Db or Model object, the security of the system is not compromised. There simply is no protected direct reference to a base-level object that can escape.

Interface gap problem. There is no need to change the interface to ensure that proxied versions of objects are returned because the metaobject is associated with the object itself and will always be invoked.

5.4.3 Place Enforcement within a Superclass

In System K, inheritance is used to associate security state with objects. Developers of new classes that must be secured must subclass the SecureObject class. This should present no problems when implementing classes from scratch. However, if an existing class is to be re-engineered, 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.

The decision as to which policies to associate with an object are hardwired into the implementation of the SecureObject class. It would be more elegant if the details of policies applying 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 and a Model. System K implements both integrity and information-flow security policies for Model objects, but only an integrity security policy for Db objects. However, both classes inherit both policies as part of their meta data. If an external configuration file was used, it would be possible to specify different policies for each. These could be adjusted at loadtime without having to change any source code.

The use of Kava avoids the requirements for inheritance because the binding between the metaobject class and base-level class is established at loadtime allowing even compiled classes to be brought under the control of the meta level. Additionally, different metaobject classes providing different security functionality can be bound to those program classes that require them rather than all classes.

5.5 Summary

This chapter contrasted conventional object-oriented techniques with use of Kava for enforcing access control policies upon distributed and mobile objects. It showed that Kava provided a better separation of concerns than conventional techniques and led to a number of improvements.

The next chapter uses the experience of the two case studies to draw some general observations about using loadtime metaobject protocols to enforce access control policies.

Chapter 6

Inferences from Case Studies

This chapter makes some inferences from the specific case studies about the general use of a loadtime MOP such as Kava for security engineering.

6.1 MOPs Provide Better Separation of Concerns than Conventional Object-Oriented Techniques

Chapter 4 discusses the application of Kava to a standalone application and Chapter 5 discusses the application of Kava to an application composed of distributed and mobile objects showed that it led to a cleaner separation of concerns. Furthermore it showed concrete benefits in terms of simplification when compared with conventional object-oriented techniques. This supported the abstract reasoning in Chapter 2 that loadtime metaobject protocols would provide a better approach for security engineering than conventional object-oriented techniques. Given that both case studies were real applications and used standard techniques for security it seems reasonable to argue that Kava could be used with other applications with similar success.

6.2 Constraints to Consider when Using MOPs for Enforcement

The following constraints must be considered when using MOPs to enforce access control policies upon compiled user-level code: (1) Detailed specification may be necessary; (2) Expressiveness of the binding specification influences success in enforcing an access control policy; (3) Granularity of interfaces also influences the success in enforcing an access control policy; (4) Security exceptions will break the transparency of the technique and increase the complexity of a solution; and (5) Applying a loadtime MOP such as Kava in the context of another language requires the presence of type safety, rewriting tools and an ability to intercept loading of classes.

6.2.1 Detailed Specifications may be Necessary

Experience with the case studies show that using Kava to enforce access controls upon either application resources or system resources requires identifying the appropriate base-level operations to bring under the control of the meta level because these form the vocabulary of the access control policy. Application-specific policies such as controlling access to application resources requires knowing the semantics and structure of an application. Other policies controlling access to system resources or operating system resources provided by library code requires knowing the semantics and structure of the library code.

6.2.2 Expressiveness of Binding Influences Complexity of Use

There is a trade-off between the expressive power of the binding specification and the complexity of programming metaobjects. For example, implementation of the Lirc resource control example required extra code to identify socket streams created to access the network from other streams. The binding specification language could be extended to allow extra context to be taken into account such as call graphs or aggregation relationships. This would relieve the programmer from having to write code to detect these relationships at runtime. However, it would increase the complexity of the enforcement mechanism making verification of Kava's correctness more difficult.

Extending Kava's binding specification language would bring the power of the language closer to AspectJ's notion of a pointcut [Kiczales, Hilsdale, Hugunin, Kersten, Palm & Griswold 2001]. AspectJ is an aspect oriented programming (AOP) language. Like MOPs, AOP allow a clean separation of concerns. The core idea is some concerns crosscut an application's functionality and that these concerns can be separated out to provide a better modularity than existing approaches [Kiczales, Irwin, Lamping, Loingtier & Lopez 1997]. AspectJ allows the crosscutting concerns to be implemented by defining code excerpts called advice and using declarative descriptions, called joinpoint, that determine where the code excerpts should be woven into the final application. The joinpoints are similar to Kava's binding specification. However, MOPs differ in that they provide a more constrained way to modify the behaviour of an application because the protocol defines the range of possible changes that can be applied through modifications of the language semantics.

6.2.3 Granularity of Interfaces Constrain Policies

The case studies showed that the nature and abstraction level of security policies that can be enforced using a MOP 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 the design. Ideally, secure RMI and authorisation would have been implemented by manipulating byte stream representations of remote method invocations. For example, subject identifiers could have been directly inserted and removed, access checks could have been applied by inspecting the target, method name and parameters and byte streams representing individual remote method invocations could have been encrypted or decrypted. However, although metaobjects could be bound to the RMI stubs and skeletons, which allowed interception of marshalling and unmarshalling, access was not possible to the byte stream representations of the marshalled invocations (although the level of abstraction that was available allowed extra parameters to be inserted and removed from the remote messages). This prevented the implementation of my own encryption/decryption and authentication algorithms and led us to rely upon SSL socket support that provides transport-level security but not middleware-level security. If Kava was able to reflect upon system classes, metaobjects could be used to provide access to byte stream representations of RMI invocations.

6.2.4 Security Exceptions Break Transparency

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 meta level 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 a whole new error semantics was added.

One area that is worth considering for future work is how to handle exceptions. In this implementation, a runtime exception is raised when a security policy is violated. Existing exception handlers catch the exception and print a stack trace on the console. What if a handler did not exist that handled runtime exceptions in a sensible way? Of course, what is sensible is to a degree application sensitive. In some cases the virtual machine should be halted. In others a message should be displayed and normal execution continued. This is a complex issue that we have considered in another paper where we explore some extensions to Kava that would support better interactions between behavioural reflection and exceptions [Welch et al. 2001].

6.2.5 Constraints upon Applying Kava to other Languages

Could Kava be used to enforce access control policies upon compiled code other than Java VM byte code? Although Kava itself could not be, the general approach to designing a MOP suitable for using for security enforcement could be applied to other languages. The main requirements are type safety, rewriting tools for compiled code and the ability to intercept class loading. A possible target language might be C# because it is object-oriented and uses a high-level compiled format for its classes.

6.3 Summary

This chapter attempted to stand back from the individual case studies and make some inferences about using a loadtime MOP such as Kava to enforce access control policies applications.

Chapter 7

Conclusions and Future Work

This chapter provides a summary of the thesis, the contributions it makes, discusses the overall contributions of the thesis, provide an overview of possible future work and makes some concluding remarks.

7.1 Summary of Thesis

Chapter 1 argued that enforcing security policies upon compiled code that makes use of application and system resources requires more than operating system-level enforcement. To enforce access control upon both system resources and upon application resources requires application-level enforcement. Conventional object-oriented techniques can be used to implement existing security architectures but provide neither a clear separation of concerns nor can they be used with compiled code. In-lined reference monitors are be used with compiled code but are unable to make use of existing security architectures or (currently) implement distributed access control policies. An approach that provides a clean separation of concerns involves the use of metaobject protocols. Existing loadtime metaobject protocols that can work with compiled code are able to provide sufficient least privilege but do not explicitly address complete mediation and economy of mechanism. Additionally they have not been used to implement distributed access control. This thesis describes an implementation of a loadtime metaobject protocol that addresses these problems and investigates whether a clean separation of concerns is achievable by carrying out two case studies with third-party applications.

Chapter 2 introduces a set of criteria for an ideal security engineering technique for enforcing access controls upon compiled user-level code, that is both library or application code. The criteria are systematically applied to four candidate security engineering techniques: operating system enforcement, conventional object-oriented engineering, in-lined reference monitors and metaobject protocols. Table 7.1 reproduces the summary of results. Only loadtime metaobject protocols and in-lined reference monitors meet most of the criteria. In particular in-lined reference monitors have well-developed arguments for their satisfaction of complete mediation and economy of mechanism. However, loadtime metaobject protocols have the advantage, with respect to the aims of this thesis, of allowing existing security architectures to be used instead of replacing them. Existing loadtime metaobject protocols do not meet all the criteria and this is used to develop a set of goals for the thesis. The goals are:

1. Implement a MOP that enforces access control policies upon compiled code.
2. Implement a MOP that controls access to both application, library and operating system resources.
3. Implement a MOP that satisfies the requirements for:
   - Least privilege
   - Complete mediation
   - Economy of mechanism
   - Clean separation of concerns
4. Demonstrate a clean separation of concerns compared with conventional software engineering techniques that still allows reuse of existing security architectures.

Chapter 3 provides an overview of Kava, which is an implementation of the metaobject protocol for Java designed to meet the first three goals. The first goal is satisfied because Kava's exploitation of byte code rewriting to insert meta-level interceptions at loadtime allows Kava to be used with compiled code. The second goal is satisfied because Kava Java's

Table 7.1: Summary of evaluation of security engineering techniques.

Technique	**Description**	**Separation of Concerns**	**Least Privilege**	**Complete Mediation**	**Economy of Mechanism**
Operating systems	Fixed placement of enforcement code within kernel.	Good	Coarse	Limited control over user-level code	Complex
Libraries	Fixed placement of enforcement code within kernel.	Good	Coarse	Limited	Language dependent
Manual insertion	Place enforcement as required within application.	Poor	Fine	Complete	Language dependent
Capabilities	Treat object references as capabilities.	Moderate	Fine	Limited	Language dependent
Proxies	Place enforcement code within proxies.	Moderate	Fine	Limited	Language dependent
Inheritance	Place enforcement within superclass.	Moderate	Fine	Complete	Language dependent
IRMs	Policy determines in-lining of enforcement. within compiled code. **Replaces existing security architecture.**	Good	Fine	Control over local objects code but **not distributed**	Good
Compile-time MOPs	Metaobjects enforce policy at runtime and bind to base level at compile time.	Good	Fine	Examples of control over distributed objects **but not local**	Includes compiler and MOP impl. in TCB
Loadtime MOPs	Metaobjects enforce policy at runtime and bind to base level at load time.	Good	Fine	Examples of control over local objects **but not distributed**	Only includes MOP impl. in TCB
Runtime MOPs	Metaobjects enforce policy at runtime and bind to base level at runtime time.	Good	Fine	Examples of control over local objects **but not distributed**	Includes runtime and MOP impl. in TCB

facility intercept the loading of application classes and Kava allows access to other types of classes to be controlled by intercepting application invocations of classes implementing library and operating system resources. The third goal is satisfied because of the fine-grained control provided by a MOP, the reuse of arguments for complete mediation and economy of mechanism developed for in-lined reference monitors and Kava's ability to modularise enforcement code within reusable metaobjects.

Demonstration of the last goal required two case studies to be carried out and these are described in Chapter 4 and Chapter 5. The purpose of the case studies was to contrast conventional object-oriented security engineering with a loadtime metaobject protocol approach. Each chapter describes both approaches and evaluates whether the benefits of a good separation of concerns was used by using a loadtime metaobject protocol as opposed to conventional techniques. The first case study application is a third-party standalone IRC client. It allows comparison of approaches to enforcing access control policies controlling an application's access to application and system resources. The second case study is a third-party distributed CASE tool that also uses weakly-mobile objects. This allows comparison of approaches to implementing secure remote method invocations and authorisation. In both case studies, Kava was found to provide a better separation of concerns than conventional techniques.

Finally, Chapter 6 makes some inferences from the specific case studies about the general use of a loadtime MOP such as Kava for security engineering. First, it argues that the case studies showed both an improved separation of concerns was provided by using a loadtime MOP and these improvements would also apply if a loadtime MOP was used with another application. Second, it discusses a set of constraints to consider when using a loadtime MOP such as Kava with other applications. There are five constraints to consider: (1) Detailed specification may be necessary; (2) Expressiveness of the binding specification influences success in enforcing an access control policy; (3) Granularity of interfaces also influences the success in enforcing an access control policy; and (4) Security exceptions will break the transparency of the technique and increase the complexity of a solution; and (5) Applying a loadtime MOP such as Kava in the context of another language requires the presence of type safety, rewriting tools and an ability to intercept loading of classes.

7.2 Overall Contributions

The overall contributions of thesis go beyond exploring the issues in implementing a loadtime metaobject protocol for Java and applying it to two case studies. These contributions are: (1) The lessons from the case studies allow us to show that the security challenges from Chapter 1 are met for each problem domain described there; (2) The loadtime metaobject protocol approach can be used to implement other non-functional concerns besides security; and (3) The loadtime metaobject approach is applicable to languages other than Java.

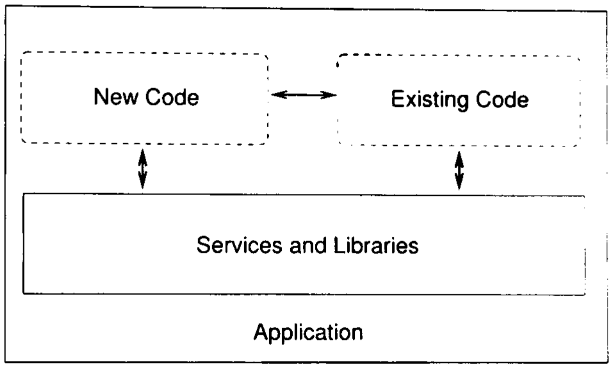

Figure 7.1: A model of applications that may have security policies enforced upon them by Kava. New code may or may not be untrusted. It may interact with existing trusted code in the form of system libraries and services. The interactions that may need to be controlled by a security metaobject are shown as double-headed arrows.

First, we argue that the experiences with the case studies are generalisable to the security challenges presented by new problem domains, as identified in Chapter 1. The metaobject protocol approach allows all accesses to both user-level and kernel-level resources by user-level code to be mediated by metaobjects that enforce security policies. This can be seen as enforcing a security policy upon an application composed of trusted and untrusted compiled code components that make use of local libraries and services. Such an application is shown in Figure 7.1. We assume that trusted code or libraries honour their interfaces and their semantics are well specified. These specifications may be supplied with code or derived using reverse engineering or visualisation. Security policies are enforced by controlling invocation of the trusted code's methods. New code components that may be untrusted are not assumed to be well specified, security policies are enforced by controlling their access to trusted code.

Table 7.2 revisits each of the problem domains described in Chapter 1 and specifies which code components are required to be well specified in order to allow a loadtime metaobject protocol to enforce security policies. In all the cases there is new code that is being introduced into an environment that contains other user-level code executing within the same address space. In each problem domain knowledge differs about the specification of the different code components. For example, untrusted mobile code can be prevented from misusing the resources available in its new environment because Kava can prevent malicious invocations of known methods whereas third-party code where there is a specification of its own methods allows us to enforce security policies particular to the functionality of the third-party code. Both these points are illustrated in the two case studies.

Second, we argue that the experiences of this thesis are applicable to non-functional requirements other than security. Security is a notoriously difficult [Saltzer & Schroeder 1975] non-functional concern to implement successfully and therefore has provided a good target for exploring the capabilities and limitations of a loadtime metaobject protocol. Further contributions of this thesis concern how the constraints identified in Chapter 6 apply more widely. These inferences restricted themselves to the use of a loadtime MOP such as Kava for security engineering. In fact, Kava could be used to implement other non-functional concerns. For example, fault tolerance, real time constraints and dynamic adaptation.

The basic requirements of least privilege, complete mediation and economy of mechanism also apply to the successful implementation of these other non-functional concerns. For example, consider checkpointing. Least privilege translates to a requirement to only enforce checkpointing where required in the target application as otherwise unnecessary saving of state will lead to reduced performance. Complete mediation translates to a requirement to intercept every possible state update. Economy of mechanism is still required because trust in the implementation of checkpointing requires validation of the implementation and this is made easier if the implementation is small.

The argument about the need for specification and its relationship to trust also apply to other non-functional concerns. Again, checkpointing requires some specification of the

Table 7.2: This table shows how Kava can be used to enforce security policies in all the new problems domains described in Chapter 1. Where the code is well specified, Kava can enforce policies related to the semantics of the code – for example the policies controlling access to IRC channels. Where no specification exists, Kava can enforce policies upon the code that limit the use of other well specified code – for example, mobile code can have its access to other code within the same address space controlled.

Problem domain	Specification			Policy Scope
	New Code	Application	Services and Libraries	
Email attachments	N	Y	Y	Control access to applications, services libraries.
Dynamic web content	N	Y	Y	Control access to applications, services libraries.
Mobile code	N	Y	Y	Controls access to applications, services libraries.
Third-party application	Y	Y	Y	Control access to new code, applications, services libraries.
Component-oriented programming	Y	Y	Y	Control access to new code, applications, services libraries.
Computational grids	Y	Y	Y	Control access to new code, applications, services libraries.

classes where state changes take place so that they can be identified. An expressive binding specification allowing declarative specifications such as bind a metaobject class to all classes that update a particular field simplify implementation of checkpointing. The classes making up the application must be at the right level of granularity to allow interception of state changes, too coarse grained and checkpoints become to unwieldy. Exceptions related to checkpointing failures will break transparency and require handling at the metalevel. Enforcement requires the presence of type safety, rewriting tools and an ability to intercept loading of classes.

Thirdly, the approach taken in this thesis is generalisable beyond Java. As stated about implementation of a loadtime metaobject requires three features: (1) type safety to guarantee interfaces are respected; (2) rewriting tools to allow the insertion of metalevel interceptions; and (3) an ability to intercept loading of classes to allow insertion of interceptions at loadtime. This could be achieved for a language such as C++ by using (1) type-safe code enforced using a toolkit such as TAL [Morrisett, Walker, Crary & Glew 1999b]; (2) use of instrumentation libraries; and, modification of static linkers and dynamic library linking routines.

7.3 Future Work

The main future work is to:

- Improve static analysis.
- Provide least privilege for metaobjects.
- Improve metaobject composition.
- Treat distribution as a first-class entity.
- Improve complete mediation.
- Investigate and improve performance.
- Implement a policy language.

7.3.1 Improve Static Analysis

As discussed in Section 3.2.5, Kava's static analysis needs to be extended to allow the superclass-subclass relationships to be determined statically. At present, the developer of the binding specification must manually specify these relationships. This is not a good approach when imposing access control policies upon unknown code. For example, in the programming with Kava example (Section 3.1.7) the ResourceManager metaobject must be bound to all classes creating an new instance of any object. The ResourceManager must determine at runtime whether the instance is either a Frame object or a subclass. This imposes a considerable overhead upon the application that is the subject of the access control policy. Completing the implementation of static analysis would ensure that this could be determined at loadtime, thereby improving Kava's performance.

Additionally, Kava's static analysis could be made more expressive. For example, implementing a binding keyword allowing call graph relationships to be considered when determining whether to make a binding. In many cases this could be determined statically.

7.3.2 Provide Least Privilege for Metaobjects

The metaobject protocol approach provides least privilege except that metaobjects need to have the same rights as the base-level objects so that checkPermission() will succeed. Should a metaobject not have the same rights then the check will fail because the rights or permissions must be held by every protection domain on the stack.

In our earlier work [Welch & Stroud 2000b, Welch & Stroud 2002] it was proposed to avoid this problem by changing the Policy implementation so that metaobjects would always be granted the same rights as the base-level objects. However, this leads to a violation of least privilege because metaobjects do not need to exercise these rights other than to check the base level is entitled to use them. Granting them to the metaobject creates a vulnerability because a hostile or faulty metaobject might go beyond checking that they are held by abusing these rights.

Subsequent work on the Simple Security-Aware metaobject protocol shows how this can be avoided. The idea is to change the MLIs so that the base level AccessControlContext is reified and passed as context to the metaobject. The access decision can then be evaluated using this access control context rather than the meta level's access control context.

This problem raises a general question, which is "What privileges should a meta level possess?". It seems reasonable to suppose that a set of privileges corresponding to the range of behaviours that can be redefined at the meta level to be defined and enforced by the Java security architecture. Caromel and Vayssiére [Caromel & Vayssiére 2003] have proposed a security framework for limiting the privileges of the meta level to reduce the possibility of a buggy or malicious meta level subverting the behaviour of an application. Kava could be modified to take advantage of both of these proposals.

7.3.3 Improve Metaobject Composition

The second case study required multiple metaobjects to co-operate in changing the runtime behaviour of single object. Although the chain of responsibility [Gamma et al. 1995] can be used with Kava to implement co-operation, it does lead to some inefficiencies because both metaobjects do not always need to be invoked for the same MLI. This is because each metaobject in the chain is invoked even when a behaviour that it has no interest in is intercepted at the base level. It may ignore the behaviour but it is still invoked because Kava intercepts the union of all behaviours that the metaobjects in the chain may need to control. This is a consequence of implementing composition at the meta level rather than in the construction of MLIs. A more efficient approach is to modify the Kava so that it chained the metaobjects together when implementing MLIs. This means that metaobjects would not be invoked unnecessarily. On the other hand, embedding composition rules into Kava would limit future changes to composition rules for metaobjects. However, this can be avoided by providing the ability to customise composition (see for example, JAC [Pawlak, Steinturier, Duchien & Florin 2001]).

7.3.4 Make Distribution First-Class

This thesis uses a single metaobject protocol to address both object and distributed object access control. The case study shows that this provides a working solution but leads to an unwieldy-looking design that may be hard to maintain. This situation can be improved if the concern of object distribution was made first-class in Kava, thereby making it easier to apply Kava to securing distributed objects. Making distribution first-class means that the binding specification can reflect the notion of binding to the interfaces offered by distributed objects. A first step would be to provide access to marshalling and unmarshalling of remote method invocations via an abstract layer. This would make the implementation of the control over RMI independent of the any particular Java RMI implementation.

7.3.5 Improve Economy of Mechanism

Although using code rewriting to implement Kava provides a good economy of mechanism because left the compiler out of the trusted computing base, it does rely upon a byte code rewriting toolkit written in Java that is roughly 20Kloc in size. Complete mediation would be improved by implementing a smaller byte code rewriting toolkit composed of only the functions depended upon by Kava. Furthermore, the toolkit could be implemented in a language with a more well-defined formal semantics than Java to make verification of its correctness tractable.

7.3.6 Investigate and Improve Performance

There will always be a trade-off between the generality of an approach like behavioural reflection and specific approaches such as manual implementation. Our case is that using a general approach has benefits, in terms of reusability and reducing programmer error, that outweigh performance costs. Only some basic performance measures have been performed and there is scope to carry out a more thorough performance analysis. Additionally, the current version of Kava incorporates some simple optimisations but considerably more can be done. For example, static analysis of metaobjects to determine if they need access to context or not. This way the costly reification of context can be avoided automatically. Additionally, lazy reification [Masuhara et al. 1992] can also be explored.

7.3.7 Provide a Policy Language for Kava

At the moment security policies are expressed through a combination of Kava's binding specification (specifying what language-level operations to control) and the policy representation used by the security model being enforced by Kava. Separation between the enforcement and policy is something that the current approach provides at the implementation level but is something that should not be separate from the user's point of view. Keeping them separate, when defining a security policy, raises the possibility of controlling an operation from a policy point of view but forgetting to include the operation in the binding specification. An improved approach is to use a single high-level policy language that automatically generates the binding specification and policy representation particular to the security model being enforced by Kava.

As a proof-of-concept, a prototype implementation of a policy compiler that maps Ponder [Damianou 2002] policies to Kava binding specifications and Java security policies has been developed [Lu 2004]. The main problem that had to be solved were providing a means to translate Ponder targets and actions to the required Java policy targets and permissions. This was solved by requiring the developers of application or library resources to provide a resource description file that provides a resource-specific Ponder vocabulary for specifying policies. This notion of a resource description file is based upon the concept of resource descriptions found in Naccio [Evans & Twyman 1999] where these a dictionary providing translations between platform-neutral abstractions and platform-specific implementations. The translator uses this to determine what permissions to add to the Java policy file and to generate the binding specification. At present the binding specification is fixed and encoded in the resource file and future work would be to automatically generate the binding specification based upon the rights being granted to programs using the user-level resources.

Ponder could also be used to specify other policies such as policies such as Clark-Wilson [Clark & Wilson 1987]. As part of earlier work, libraries of metaobjects for implementing the Clark-Wilson [Welch 1999] security policy was designed. Using the Ponder to Kava policy translator requires defining new resource descriptions to allow Ponder abstractions to be mapped to Kava binding specifications and appropriate Java security policies.

7.4 Concluding Remarks

Security is a notoriously difficult non-functional concern to implement successfully. Furthermore, existing attempts to implement security while maintaining a separation of concerns has led to inflexible or untrustworthy implementations. The loadtime metoabject approach described in this thesis avoids either pitfall by combining code rewriting and metaobject protocols. Additionally, this thesis describes an approach that could be applied to other non-functional concerns and other languages. This is a significant step forward in allowing developers to meet the challenges arising in new problem domains such as mobile code or component-oriented programming.

Bibliography

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

Anderson, A., Parducci, B., Adams, C., Flinn, D., Brose, G., Lockhart, H., Beznosov, K., Kudo, M., Humenn, P., Godik, S., Andersen, S., Croker, S. & Moses, T. (2003). eXtensible access control markup language (XACML) version 1.0, *Technical report*, OASIS group.

Anderson, J. P. (1972). Computer Security Technology Planning Study, *Technical Report ESD-TR-73-51*, Electronic Systems Division, Deputy for Command and Management Systems, HQ Electronic Systems Division (AFSC).

Anti-Phishing Working Group (2004). What is phishing? [online], Available from: `http://www.antiphishing.org`. [Accessed August 2004].

Apache Week (1997). Apache and secure transactions, Available from: `http://www.apacheweek.com/features/ssl`. [Accessed August 2004].

Arnold, K. & Gosling, J. (1998). *The Java programming language (2nd ed.)*, ACM Press/Addison-Wesley Publishing Co.

Attardi, G., Bonini, C., Boscotrecase, M. R., Flagella, T. & Gaspari, M. (1989). Metalevel Programming in CLOS, *ECOOP'89*.

Bell, E. D. & LaPadula, L. J. (1976). Secure computer system: Unified exposition and Multics interpretation, *Technical Report ESD-TR-73-306*, MITRE Corporation, Bedford, MA.

Benantar, M., Blakley, B. & Nadain, A. J. (1996). Approach to object security in distributed SOM, *IBM Systems Journal* **35**(2): 192–203.

Blair, G., Coulson, G., Andersen, A., Blair, L., Clarke, M., Costa, F., Duran-Limon, H., Fitzpatrick, T., Johnston, L., Moreira, R., Parlavantzas, N. & Saikoski, K. (2001). The design and implementation of Open ORB 2, IEEE Distrib. Syst. Online 2, 6 (Sept. 2001). Available from: `http://computer.org/dsonline`. [Accessed August 2004].

Briot, J.-P., Doi, N., Honda, Y., Ichisugi, Y., Kodama, Y., Ohsawa, I., Shibayama, E., Takada, T., Watanabe, T. & Yonezawa, A. (1990). *ABCL: An Object-Oriented Concurrent System*, Computer Systems Series, MIT Press.

Bull, J. A., Gong, L. & Sollins, K. R. (1992). Towards security in an open systems federation, *Proceedings of ESORICS 92, 2nd European Symposium on Research in Computer Security*, Vol. 648 of *Lecture Notes in Computer Science*, Springer, Toulouse, France, pp. 3–20.

Butler, R., Welch, V., Engert, D., Foster, I., Tuecke, S., Volmer, J. & Kesselman, C. (2000). A national-scale authentication infrastructure, *Computer* **12**(33): 60–66.

Caromel, D., Huet, F. & Vayssiére, J. (2001). A simple security-aware MOP for Java, *Proceedings of REFLECTION01, Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, Vol. 2192 of *Lecture Notes in Computer Science*, Springer, Kyoto, Japan, pp. 118–125.

Caromel, D. & Vayssiére, J. (2001). Reflections of MOPs, components, and java security, *Proceedings of ECOOP 2001, European Conference on Object-Oriented Programming*, Vol. 2072 of *Lecture Notes in Computer Science*, Springer, Budapest, Hungary, pp. 256–274.

Caromel, D. & Vayssiére, J. (2003). A security framework for reflective Java applications, *Software Practice and Experience* **33**(9): 821–846.

Cazzola, W. (2000). *Communication-Oriented Reflection: A Way to Open Up the RMI Mechanism*, PhD thesis, Universita degli Studi di Milano, Italy.

Chess, D. M., Harrison, C. G. & Kershenbaum, A. (1995). Mobile agents: Are they a good idea?, *Technical Report RC19887*, T. J. Watson Research Centre, IBM. Originally written 12/1994, declassified 3/1995.

Chiba, S. (2000). Load-time structural reflection in Java, *Proceedings of ECOOP 2000, European Conference on Object-Oriented Programming*, Vol. 1850 of *Lecture Notes in Computer Science*, Springer, Sophia Antiopolis and Cannes, France, pp. 313–336.

Chiba, S. & Masuda, T. (1993). Designing an extensible distributed language with a meta-level architecture, *Proceedings of 7th European Conference on Object-Oriented Programming (ECOOP'93)*, Kaiserslautern, pp. 482–501.

Clark, D. D. & Wilson, D. R. (1987). A comparison of commercial and military computer security policies, *IEEE Symposium on Security and Privacy*, IEEE, pp. 184–194.

Cohen, G. A. & Chase, J. S. (1998). Automatic Program Transformation with JOIE, *USENIX Annual Technical Symposium*, USENIX, New Orleans, Louisiana, pp. 167–178.

Cointe, P. (1987). Metaclasses are first class: The ObjVlisp model, *Conference proceedings on Object-oriented programming systems, languages and applications*, ACM Press, pp. 156–162.

Colouris, G., Dollimore, J. & Kindberg, T. (2001). *Distributed Systems: Concepts and Design*, third edition edn, Addison-Wesley.

Coulson, G. (2002). What is reflective middleware?, IEEE Distrib. Syst. Online 2, 8 (Dec. 2001); available from: `http://computer.org/dsonline`. [Accessed May 2002].

Czajkowski, G. & von Eicken, T. (1998). JRes: A resource accounting interface for Java, *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ACM Press, Vancouver, British Columbia, pp. 21–35.

Dahm, M. (1998). Byte code engineering with the JavaClass API, *Technical Report B-17-98*, Friei Universitat, Berlin.

Damianou, N. C. (2002). *A Policy Framework for Management of Distributed Systems*, PhD thesis, Department of Computing, Imperial College of Science, Technology and Medicine.

Dennis, J. B. & Earl C., V. H. (1966). Programming semantics for multiprogrammed computations, *Communications of the ACM* **9**(3): 143–155.

Dierks, T. & Allen, C. (1999). The TLS protocol version 1.0. Internet request for comment RFC 2246., Internet Engineering Task Force. Available from: `http://www.ietf.org/rfc/rfc2246.txt`. [Accessed February 2003].

Dijkstra, E. W. (1976). *A Discipline of Programming*, Prentice Hall, Englewood Cliffs, NJ.

Dijkstra, E. W. (1982). *Selected Writings on Computing: A Personal Perspective*, Springer-Verlag. Manuscript "EWD 447: On the role of scientific thought" (30th August 1974) was published as pages 60-66.

Erlingsson, Ú. & Schneider, F. B. (2000). SASI enforcement of security policies: a retrospective, *Proceedings of the 1999 Workshop on New Security Paradigms*, ACM Press, pp. 87–95.

Evans, D. & Twyman, A. (1999). Flexible policy-directed code safety, *IEEE Symposium on Research in Security and Privacy*, IEEE Computer Society, Technical Committee on Security and Privacy, IEEE Computer Society Press, Oakland, CA, pp. 32–45.

Fabre, J.-C., Nicomette, V., Perennou, T., Wu, Z. & Stroud, R. J. (1995). Implementing fault-tolerant applications using reflective object-oriented programming, *Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing (FTCS-25)*, Pasadena, USA, pp. 489–498.

Fabre, J.-C. & Perennou, T. (1996). Friends – a flexible architecture for implementing fault-tolerant and secure distributed systems, *Proceeding of Dependable Computing - EDCC-2, Second European Dependable Computing Conference*, Vol. 1150 of *Lecture Notes in Computer Science*, Springer, Taormina, Italy, pp. 3–20.

Forman, I. & Danforth, S. (1999). *Putting Metaclasses to Work*, Addison Wesley.

Forman, I. R., Danforth, S. & Madduri, H. (1994). Composition of before/after metaclasses in SOM, *Proceedings of the ninth annual conference on Object-oriented programming systems, language, and applications*, ACM Press, pp. 427–439.

Foster, I. (2001). The anatomy of the Grid: Enabling scalable virtual organizations, *Proceedings of the 1st International Symposium on Cluster Computing and the Grid*, IEEE Computer Society, p. 6.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, Massachusetts.

Gollmann, D. (1999). *Computer Security*, John Wiley & Sons Ltd.

Golm, M. & Kleinoder, J. (1999). Jumping to the meta level: Behavioural reflection can be fast and flexible, *Proceedings of REFLECTION01, Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, Vol. 1616 of *Lecture Notes in Computer Science*, Springer, Kyoto, Japan, pp. 22–39.

Gong, L. (1999). *Inside Java(tm)2 Platform Security*, Addison-Wesley.

Gong, L., Mueller, M., Prafullchandra, H. & R., S. (1997). Going beyond the sandbox: An overview of the new security architecture in the Java Development Kit 1.2, *USENIX Symposium on Internet Technologies and Systems*, USENIX, Berkeley, CA, Monterey, California, pp. 103–112.

Gosling, J., Joy, B. & Steele, G. L. (1996). *The Java Language Specification*, The Java Series, Addison-Wesley.

Gowing, B. & Cahill, V. (1996). Meta-Object Protocols for C++:The Iguana Approach, *Reflection ' 96*, pp. 137–152. R49.

Graube, N. (1989). Metaclass compatibility, *Conference proceedings on Object-oriented programming systems, languages and applications*, ACM Press, pp. 305–315.

Hashii, B., Malabarba, S., Pandey, R. & Bishop, M. (2000). Supporting reconfigurable security policies for mobile programs, *Computer Networks/Proceedings of the 9th World Wide Web Conference*, Vol. 33, Elsevier, Amsterdam, The Netherlands, pp. 77–93.

Hölzle, U. (1993). Integrating independently-developed components in object-oriented languages, *Proceedings of the 7th European Conference on Object-Oriented Programming*, Springer-Verlag, pp. 36–56.

Java Team (1996-1999). JDK(tm) 2 SDK documentation. Available from: http://java.sun.com.

Keller, R. & Hölzle, U. (1998). Binary component adaptation, *Proceedings of the 12th European Conference on Object-Oriented Programming*, Springer-Verlag, pp. 307–329.

Kiczales, G., des Rivieres, J. & Bobrow, D. G. (1991). *The Art of the Metaobject Protocol*, Massachusetts Institute of Technology.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. & Griswold, W. G. (2001). An Overview of AspectJ, *ECOOP 2001*, Vol. LNCS 2072, Springer, Budapest, Hungary, pp. 327–353.

Kiczales, G., Irwin, J., Lamping, J., Loingtier, J.-M. & Lopez, C. (1997). Aspect-oriented programming, *Proceedings of ECOOP 97, European Conference on Object-Oriented Programming*, Vol. 1241 of *Lecture Notes in Computer Science*, Springer, Jyväskylä, Finland, pp. 220–242.

Killijian, M.-O. & Fabre, J. C. (2000). Implementing a reflective fault-tolerant CORBA system, *19th IEEE Symposium on Reliable Distributed Systems*, IEEE Computer Society, Nurnberg, Germany, pp. 154–163.

Killijian, M.-O., Fabre, J.-C., Ruiz-Garcia, J.-C. & Chiba, S. (1998). A metaobject protocol for fault-tolerant corba applications, *The Seventeenth Symposium on Reliable Distributed Systems 1998*, IEEE Computer Society, West Lafayette, Indiana, USA.

Kon, F., Román, M., Liu, P., Mao, J., Yamane, T., a, C. M. & Campbell, R. H. (2000). Monitoring, security, and dynamic configuration with the dynamicTAO reflective orb, *IFIP/ACM International Conference on Distributed systems platforms*, Springer-Verlag New York, Inc., pp. 121–143.

Lampson, B., Abadi, M., Burrows, M. & Wobber, E. (1992). Authentication in distributed systems: theory and practice, *ACM Trans. Comput. Syst.* **10**(4): 265–310.

Lampson, B. W. (1973). A note on the confinement problem, *Communications of the ACM* **16**(10): 613–615.

Lampson, B. W. (1974). Protection, *SIGOPS Operating System Review* **8**(1): 18–24.

Liang, S. & Bracha, G. (1998). Dynamic class loading in the Java virtual machine, *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ACM Press, pp. 36–44.

Lieberman, H. (1986). Using prototypical objects to implement shared behaviour in object-oriented systems, *Proceedings of OOPSLA'86*, ACM Press.

Lorch, M., Proctor, S., Lepro, R., Kafura, D. & Shah, S. (2003). First experiences using XACML for access control in distributed systems, *Proceedings of the 2003 ACM workshop on XML security*, ACM Press, pp. 25–37.

Lu, F. (2004). Implementing Ponder policies using Kava (MCompSci report). University of Victoria at Wellington, School of Mathematics, Statistics and Computer Science (Te Tatou Kura). Supervised by Ian Welch.

Maes, P. (1987). Concepts and experiments in computational reflection, *Conference proceedings on Object-oriented programming systems, languages and applications*, ACM Press, pp. 147–155.

MAFTIA Project (2003). Conceptual model and architecture of MAFTIA, *Technical Report D21*, MAFTIA Project.

Malenfant, J., Jacques, M. & Demers, F.-N. (1996). A tutorial on behavioural reflection and its implementation, *First International Conference on Reflection and Metalevel Archtectures (Reflection '96)*, ACM Press, San Francisco, California, USA, pp. 1–20.

Masuhara, H., Matsuoka, S., Asai, K. & Yonezawa, A. (1995). Compiling away the meta-level in object-oriented concurrent reflective languages using partial evaluation, *Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, ACM Press, pp. 300–315.

Masuhara, H., Matsuoka, S., Watanabe, T. & Yonezawa, A. (1992). Object-oriented concurrent reflective languages can be implemented efficiently, *conference proceedings on Object-oriented programming systems, languages, and applications*, ACM Press, pp. 127–144.

McAffer, J. (1995). *A Meta-Level Architecture for Prototyping Object Systems*, PhD thesis, University of Tokyo.

McCollum, C. J., Messing, J. R. & Notargiacomo, L. (1990). Beyond the pale of MAC and DAC - defining new forms of access control, *IEEE Computer Society Symposium on Research in Security and Privacy*, Oakland, California, pp. 190–200.

Millen, J. K. (1992). A Resource Allocation Model for Denial of Service, *IEEE Computer Society Symposium on Research in Security and Privacy*, IEEE, Oakland, California, pp. 137–147.

Miller, M. S. & Shapiro, J. S. (2003). Paradigm regained: Abstraction mechanisms for access control, *in* V. A. Saraswat (ed.), *Advances in Computing Science – ASIAN 2003*

Programming Languages and Distributed Computation, 8th Asian Computing Science Conference, Vol. 2896 of *Lecture Notes in Computer Science*, Springer, Mumbai, India.

Morrisett, G., Walker, D., Crary, K. & Glew, N. (1999a). From system F to typed assembly language, *ACM Transactions on Programming and Languages Systems* **21**(3): 527–568.

Morrisett, G., Walker, D., Crary, K. & Glew, N. (1999b). From system F to typed assembly language, *ACM Trans. Program. Lang. Syst.* **21**(3): 527–568.

Mulet, P., Malenfant, J. & Cointe, P. (1995). Towards a methodology for explicit composition of metaobjects, *Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, ACM Press, pp. 316–330.

Neuman, B. C. & Ts'o, T. (1994). Kerberos: An authentication service for computer networks, *IEEE Communications* **32**(9): 33–38.

OASIS Group (2004). Security and privacy considerations for the OASIS security assertion markup lanaguage (SAML) v2.0, *Technical report*, OASIS Group. Edited by Frederick Hirsch, Rob Philpott and Eve Maler.

Oliva, A. & Buzato, L. E. (1999). The Design and Implementation of Guaraná, *Usexnix COOTS*, Usenix, San Deigo, California, USA, pp. 203–216.

Otway, D. & Rees, O. (1987). Efficient and timely mutual authentication, *SIGOPS Operating System Review* **21**(1): 8–10.

Pandey, R. & Hashii, B. (1999). Providing fine-grained access control for Java programs, *European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science, Springer, Lisbon, Portugal, pp. 450–473.

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules, *ACMcomms* **15**(12): 1053–1058.

Pawlak, R., Steinturier, L., Duchien, L. & Florin, G. (2001). JAC: a flexible solution for aspect-oriented programming in Java, *Proceedings of REFLECTION01. Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, Springer, Kyoto, Japan, pp. 1–24.

Pfleeger, C. P. (1997). *Security in Computing*, second edn, Prentice-Hall International, Inc.

Philpott, R. (2004). Technical overview of the OASIS security assertion markup language (SAML) v1.1, *Technical report*, OASIS Group.

Redell, D. (1974). *Naming and Protection in Extendable Operating Systems*, PhD thesis, MIT. Available from: `http://www.lcs.mit.edu/publications/specpub.php?id=708`.

Redmond, B. & Cahill, V. (2002). Supporting unanticipated dynamic adaptation of application behaviour, *Proceedings of the 16th European Conference on Object-Oriented Programming*, Springer-Verlag, pp. 205–230.

Riechmann, T. & Hauck, F. J. (1997). Meta objects for access control: extending capability-based security, *Proceedings of the 1997 workshop on New security paradigms*, ACM Press, pp. 17–22.

Riechmann, T. & Hauck, F. J. (1998). Meta objects for access control: a formal model for role-based principals, *Proceedings of the 1998 workshop on New security paradigms*, ACM Press, pp. 30–38.

Roulo, M. (1998). Accelerate your java apps!, Available from:`http://www.javaworld.com/javaworld/jw-09-1998/jw-09-speed.html`. [Accessed February 2004].

Rutt, T., Curtis, D., Hopkins, A., Fairthorne, B., Hartman, B., Nessett, D., Vleck, T. V., Frantz, D., Lejeune, H., Blakley, B., Mukerji, J., Ammon, C., Salmond, K. & Allred, L. (1995). Corba security specification, *Specification*, OMG.

Saltzer, J. H., Reed, D. P. & Clark, D. D. (1984). End-to-end arguments in system design, *ACM Transactions on Computer Systems* **2**(4): 277–288.

Saltzer, J. H. & Schroeder, M. D. (1975). The protection of information in computer systems, *Proceedings of the IEEE* **63**(9): 1278–1308. S64.

Samarati, P. & Vimercati, S. (2000). *Foundations of Security Analysis and Design (Tutorial Lectures)*, Springer, chapter Access Control: Policies, Models, and Mechanisms, pp. 137–196.

Schneider, F. B. (2000). Enforceable security policies, *ACM Transactions on Information System Security* **3**(1): 30–50.

Schneider, F., Morrisett, G. & Harper, R. (2000). A language-based approach to security, *in* R. Wilhelm (ed.), *Informatics – 10 Years Back, 10 Years Ahead. Conference on the Occasion of Dagstuhl's 10th Anniversary.*, Vol. 2000, Springer, Saarbrücken, Germany, pp. 86–101.

Schneier, B. (1995). *Applied cryptography (2nd ed.): protocols, algorithms, and source code in C*, John Wiley & Sons, Inc.

Secure Computing Corporation (1997). DTOS general system security and assurability assessment report, *Technical Report CDRL A011*, Secure Computing Corporation, Roseville, MN.

Sirer, E. G., Grimm, R., Gregory, A. J. & Bershad, B. N. (1999). Design and implementation of a distributed virtual machine for networked computers, *Proceedings of the seventeenth ACM symposium on Operating systems principles*, ACM Press, pp. 202–216.

Sophos Anti-Virus (2004). W32/bagle-f virus analysis, Available from: `http://www.sophos.com/virusinfo/analyses/w32baglea.html`. [Accessed June 2004].

Spencer, R., Smalley, S., Loscocco, P., Hibler, M., Andersen, D. & Lepreau, J. (1999). The Flask security architecture: System support for diverse security policies, *Proceedings of the Eighth USENIX Security Symposium*, USENIX, pp. 123–139.

Stroud, R. (1992). Transparency and reflection in distributed systems, *Proceedings of the 5th workshop on ACM SIGOPS European workshop*, ACM Press, pp. 1–5.

Stroud, R. J. & Wu, Z. (1995). Using metaobject protocols to implement atomic data types, *Proceedings of the 9th European Conference on Object-Oriented Programming*, Springer-Verlag, pp. 168–189.

Sun Microsystems (1999–2004). JavaMail API, Avaliable from: `http://java.sun.com/products/javamail/`. [Accessed September 2004].

Sun Microsystems (2000). The scoop on rmi and ssl, Available fom: `http://java.sun.com/j2se/1.3/docs/guide/rmi/SSLInfo.html`. [Accessed January 2004].

Szyperski, C. (1998). *Component Software – Beyond Object Oriented Programming*, Addison-Wesley Longman Limited.

Úlfar Erlingsson & Schneider, F. B. (2000). IRM enforcement of Java stack inspection, *IEEE Symposium on Research in Security and Privacy*, IEEE Computer Society, Technical Committee on Security and Privacy, IEEE Computer Society Press, Oakland, CA, pp. 246–255.

US Department of Defense (1985). DoD trusted computer system evaluation criteria (the Orange book), *Technical Report DOD 5200.28-STD*, US Department of Defense.

von Nieda, J. (2001). Lirc – an IRC client for Java, `http://www.vonnieda.org/Lirc`. Last accessed 17/6/02.

Wahbe, R., Lucco, S., Anderson, T. E. & Graham, S. L. (1993a). Efficient software-based fault isolation, *Proceedings of the fourteenth ACM symposium on Operating systems principles*, ACM Press, pp. 203–216.

Wahbe, R., Lucco, S., Anderson, T. E. & Graham, S. L. (1993b). Efficient Software-Based Fault Isolation, *Fourteenth ACM Symposium on Operating Systems Principles*, Asheville, North Carolina, pp. 202–216.

Wallach, D. S., Balfanz, D., Dean, D. & Felten, E. W. (1997). Extensible security architectures for java, *Proceedings of the sixteenth ACM symposium on Operating systems principles*, ACM Press, pp. 116–128.

Welch, I. (1997). Adding security to commercial-off-the-shelf software, *European Research Seminar on Advances in Distributed Systems (ERSADS)*.

Welch, I. (1999). Reflective enforcement of the Clark-Wilson integrity model, *2nd Workshop on Distributed Object Security, OOPSLA'99*.

Welch, I. S. & Stroud, R. (1999a). From Dalang to Kava – the evolution of a reflective Java extension, *Second International Conference on Meta-Level Architectures and Reflection*, Vol. 1616, Springer, Saint-Malo, France.

Welch, I. S., Stroud, R. J. & Romanovsky, A. (2001). Aspects of exceptions at the meta-level, *Metalevel Architectures and Separation of Crosscutting Concerns, Third Internation Conference, REFLECTION 2001*, Vol. 2192 of *Lecture Notes in Computer Science*, Springer, Kyoto, Japan, pp. 280–282.

Welch, I. & Stroud, R. (1998a). Using metaobject protocols to adapt third-party components, `http://www.comp.lancs.ac.uk/computing/middleware98/wips.html`, last accessed 14/07/02. Work in Progress paper at Middleware'98.

Welch, I. & Stroud, R. (2003). Re-engineering security as a crosscutting concern – experience with a third party application, *The Computer Journal* **46(5)**: 578–589.

Welch, I. & Stroud, R. J. (1998b). A reflective Java class loader, *ECOOP Workshops 1998*, Springer Verlag, pp. 374–375.

Welch, I. & Stroud, R. J. (1999b). Supporting Real World Security Models in Java, *7th IEEE International Workshop on Future Trends of Distributed Computing Systems*, Cape Town, South Africa, pp. 155–159.

Welch, I. & Stroud, R. J. (2000a). Kava – a reflective Java based on bytecode rewriting, *in* W. Cazzola, R. J. Stroud & F. Tisato (eds), *Reflection and Software Engineering*, Vol. 1826, Springer–Verlag, Heidelberg, Germany, pp. 157–169.

Welch, I. & Stroud, R. J. (2000b). Using reflection as a mechanism for enforcing security policies in mobile code, *Proceedings of ESORICS 2000, 6th European Symposium* *on Research in Computer Security*, Vol. 1895 of *Lecture Notes in Computer Science*, Springer, Toulouse, France, pp. 309–323.

Welch, I. & Stroud, R. J. (2001). Kava – using byte-code rewriting to add behavioral reflection to Java, *Proceedings of COOTS 2001, USENIX Conference on Object-Oriented Technologies and Systems*, USENIX, Berkeley, CA, San Antonio, Texas, pp. 119–130.

Welch, I. & Stroud, R. J. (2002). Using reflection as a mechanism for enforcing security policies on compiled code, *Journal of Computer Security* **10**: 399–432.

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

Wulf, W., Cohen, E., Corwin, W., Jones, A., Levin, R., Pierson, C. & Pollack, F. (1974). Hydra: the kernel of a multiprocessor operating system, *Commun. ACM* **17**(6): 337–345.

Yellin, F. (1996). Low level security in Java, Available from: `http://java.sun.com/sfaq/verifier.html`. [Accessed August 2002].

Yoshikawa, C., Chun, B., Eastham, P., Vahdat, A., Anderson, T. & Culler, D. (1997). Using smart clients to build scalable services, *Proceedings of the USENIX 1997 Annual Technical Conference*, USENIX Association, Anaheim, CA, pp. 105–117.

Zimmerman, C. (1996). Metalevels, MOPs and What all the Fuzz is All About, *in* C. Zimmermann (ed.), *Advances in Object-Oriented Metalevel Architectures and Reflection*, CRC Press.