Method-Specific Access Control in Java via Proxy Objects using Annotations

by

Jeffrey Zarnett

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2010

Abstract

Partially restricting access to objects enables system designers to finely control the security of their systems. We propose a novel approach that allows granting partial access at method granularity on arbitrary objects to remote clients, using proxy objects.

Our initial approach considers methods to be either safe (may be invoked by anyone) or unsafe (may be invoked only by trusted users). We next generalize this approach by supporting Role-Based Access Control (RBAC) for methods in objects. In our approach, a policy implementer annotates methods, interfaces, and classes with roles. Our system automatically creates proxy objects for each role, which contain only methods to which that role is authorized.

This thesis explains the method annotation process, the semantics of annotations, how we derive proxy objects based on annotations, and how clients invoke methods via proxy objects. We present the advantages to our approach, and distinguish it from existing approaches to method-granularity access control. We provide detailed semantics of our system, in First Order Logic, to describe its operation.

We have implemented our system in the Java programming language and evaluated its performance and usability. Proxy objects have minimal overhead: creation of a proxy object takes an order of magnitude less time than retrieving a reference to a remote object. Deriving the interface—a one-time cost—is on the same order as retrieval. We present empirical evidence of the effectiveness of our approach by discussing its application to software projects that range from thousands to hundreds of thousands of lines of code; even large software projects can be annotated in less than a day.

Acknowledgements

I wish to acknowledge the guidance and contributions my thesis supervisors, Professors Patrick Lam and Mahesh Tripunitara, of the University of Waterloo. I would also like to thank Todd Veldhuizen, formerly of the University. My thanks also go to my readers, who kindly gave their time to review this document and suggested a number of improvements.

I would like to acknowledge the assistance of Christian Riege, software development manager of Riege Software International GmbH (Germany), for introducing the problem area and inspiring the use case.

I give thanks to the Springer Science & Business Media, for kindly granting permission to re-publish some material that appeared in the proceedings of the 5^{th} International Conference on Information Systems Security (2009) in Kolkata, India.

Finally, I would like to acknowledge the contributions of deadlines... because without them, nobody else could get anything done.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

We address the problem of fine-grained access control for security within the Java programming language. We target method-specific access controls (i.e., whether or not a particular method may be invoked on an object by a particular principal).

Java is an object-oriented programming language. A programmer declares and implements classes; objects are instances of classes. Classes have methods and data. A good programming practice is to ensure that all accesses to an object are via methods it exposes, thereby respecting the principles of encapsulation and information hiding.

Access control regulates accesses to resources by principals. It is one of the most important aspects of the security of a system. A protection state or policy contains all information needed by a reference monitor to enforce access control. An access control model describes the policy of the system.

Access control makes up a major building block in safety and security of systems, both electronically and in the real world. Many programming languages already incorporate the concept of access control, using keywords like `public` and `private`. Also, Java includes a *Security Manager*, which controls access to various external facilities, like file I/O.

In a larger context, developers have recently devoted much more attention to program security than before. In a world where software is at the heart of billions of dollars of transactions every hour, home computers have high-speed internet connections enabled 24 hours a day, and everyone's personal information resides in a multitude of company and governmental databases, security is a concern for many.

We propose a novel mechanism for access control to methods in Java objects. There are built-in mechanisms for program structuring in Java that resemble access control. For example, it is possible to specify that only some methods are public, while others are private. Private methods may be invoked only by other methods within the class. Note that access control based on method visibility is not secure: untrusted principals may circumvent the method visibility rules using Java reflection.

Method visibility mechanisms are coarse-grained: for instance, any method may invoke methods that are denoted as public. Several other proposed approaches augment the basic access control features in Java. Stack inspection [1], for example, can be used to provide method-specific access control, but struggles with remote clients.

Since its appearance, Role-Based Access Control (RBAC) [2] has emerged as the dominant access control model in enterprise settings. In RBAC, principals are called users. Users get permissions to access resources via membership in roles. We present a new way to realize RBAC in Java. Java is a widely-used programming language, and therefore is an important context in which to realize RBAC. We are not the first to make this observation (see, for example, [3]); however, our approach has several advantages over previous approaches.

We focus on clients that use Remote Method Invocation (RMI). Our approach works as follows. We use Java *annotations* [4] in our approach. Annotations enable developers to associate arbitrary metadata—in our case, access control metadata—with code. In particular, developers annotate methods, interfaces, and classes in the Java source code with roles from the developer-specified RBAC policy. We test for the presence of annotations after the source code is compiled.

At compile-time, our system builds interfaces according to the specified policy. At run-time, we create proxies matching these interfaces and give the proxies to clients instead of granting them access to the original objects.

Our approach provides more fine-grained access control than previous approaches: a proxy object exposes only those methods to which the client is authorized. Consequently, an RMI client is unaware of the existence of other methods in the class. This has an additional efficiency side benefit: we preclude the client from even invoking a method to which it is not authorized, and thus, large arguments need not be sent over the network. Other approaches, such as bytecode editing [5], check at the server whether the call is authorized after the client has invoked the method and the arguments have been transmitted.

The remainder of this document is organized as follows. Chapter 2 contains the background information and related work. In Chapter 3, we present our first technique for access control, which partitions methods into "safe" and "unsafe" categories based on simple Java Annotations. In Chapter 4, we generalize our system to allow arbitrary roles, in a hierarchy, according to the Role-Based Access Control Standard. We present an overview of the performance analysis in Chapter 5, and some future enhancements in Chapter 6. Finally, we present related work in Chapter 7, and conclude in Chapter 8.

Chapter 2

Background

We define *Security* in this context as preventing damage to the system or environment through various methods and mechanisms. In the programmatic sense, it may be useful to say that a program is secure when it does not allow access not explicitly authorized, and does not leak data. A typical way of controlling access is authentication, similar to the use of fingerprint scanners to control entry to a facility. Just because something is secure does not mean it is safe: many dangerous things are secure, such as nuclear weapons – access controls prevent most people from being able to look at or manipulate them.

Java Remote Method Invocation (RMI) is the Java language method for creating distributed applications. This is the equivalent of the remote procedure call (RPC) mechanism, but in Java. As with the virtual machine concept, RMI allows clients on different platforms to interoperate without protocol translation. RMI also allows the passing of complex objects between clients and servers [6].

Enterprise Java Beans (EJB) is a specification for the server-side architecture for Java Enterprise Edition (Java EE). It is intended as an architecture for component-based business applications, and offers features as scalability, transactions, persistence, and concurrent users [7]. Applications run inside an application server (such as JBoss Application Server), which provides the supporting services, such as data persistence.

This chapter surveys the various ways access controls are used in the Java programming language. The survey touches on access controls, specifying security policy, stack inspection, and dynamic code generation and loading.

We start the examination of Java security with some basic concepts from access controls and its principles. We continue with a description of how Java allows developers to use both explicit and implicit access controls. Next, we examine how these controls might be implemented using stack inspection. This technique often relies on bytecode analysis. Finally, we look at a complication which significantly affects all the preceding techniques: dynamic code generation and loading. We explore how dynamic code generation works, why it may be disruptive, and how it can even be used to increase security.

2.1 Java Access Controls

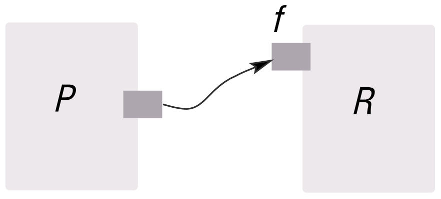
Java provides several levels of access control for its fields and methods as a built in feature of the language. Fields and methods may be marked `public`, `private`, `protected`, or package private (no keyword). Sun Microsystems provides a brief overview of the keywords in [8]. These are effectively only relevant at compile time, and are intended to help programmers obey the principle of encapsulation.

Java also makes use of policy files to build access matrices that determine authorization. Such files are mostly a set of grant statements, specifying whom receives which permissions to what resources. Programmers specify policies before program startup. Security policies can be changed at runtime by loading a new policy, but the process of refreshing the security policy is implementation-dependent. [9].

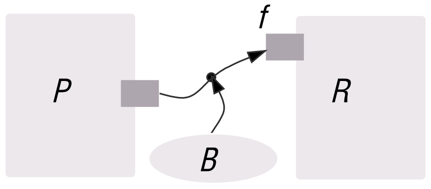
Two necessary definitions: A *principal* is an entity that can take or authorize some action(s), and *target* is some entity we wish to protect [10].

Permissions for a method invocation are the intersection of the permissions of all principals within the call sequence [9]. This enforces the principle of least privilege, because the effective privileges are those shared by all active principals. In a simple example, the file management service has read and write access to the directory `files/user1/` directory, and the user wishing to write a file only has read access to `files/user1/`. The intersection of these two permissions is read access, since it is the only one shared between the two principals. The intersection policy prevents privilege escalation when low-privilege code calls high-privilege code, and prevents privilege leakage when high-privilege code calls low-privilege code.

Access Control Principles Three basic ways to look at Java security are the identity-based, multi-level, and role-based approach, none of which have language support. This naturally introduces risk, because security implementations may themselves be insecure when the application programmer writes them. Consider the example of encryption: the Advanced Encryption Standard (AES) is a well-understood method of encryption, yet it is far better to use a pre-built and tested AES library than to create your own, because your own implementation does not benefit from the rigourous testing the pre-built library has received.



(a) Default method invocation semantics: a call is always allowed



(b) Security constraints on method invocations

Figure 2.1: Method Invocation with and without Access Control [5]

In identity-based security, a triplet of (Code Base, Signer, Principals) represents rights. If the code base belongs to `example.com`, the signer is `alice`, and the prinicpal is `bob - /user/bob - rw`, code from example.com, signed by Alice, and executed by Bob has read and write permissions to the directory `/user/bob`. Identities should be mapped 1:1 to real-life users of the system.

Multi-level security is based upon the Bell-Lapadula model of security—informally, “read down, write up" [11]. A principal (like Alice from the preceding paragraph) is explicitly granted rights for various levels, such as read access to level 1, read and write access to level 2, and write access to level 3. Unfortunately, this approach requires that the programmer insert security checks (e.g., assert level 1 access) into the beginning of every method in the program. This is cumbersome, and requires decompilation to apply to third party code for which we do not possess the source (such as libraries).

Role-based security associates various roles with principals. Roles are changeable, unlike identities, but both could be principals in Java. An example of a role would be a secretary in the sales department. Whoever assumes the secretary role receives permission to view and edit schedules, add and update contact information, et cetera. Roles can also have subsumption, such as management subsuming members – continuing the example, the manager has the secretary’s abilities, plus whatever managerial rights he receives (e.g., publish the schedule). Ultimately, roles are more flexible than identities as a method of security, since various users may assume various roles as needed.

Access constraints are not always widely employed in Java. The sole exception is the Java security manager’s control of access to various external resources. The goal of fine-grained access control is to allow invocation of particular methods only when permitted. In Figure 2.1, object P wishes to invoke function f on object R. The call proceeds if condition B holds; otherwise access is denied.

In a role-based access control system, principals are called users. Users get permissions to access resources by being a member of one or more roles.

Sandhu et al [2] described various levels of RBAC, from 0 (simplest) to 3 (most features). We have chosen to implement RBAC_1: the basic model with role inheritance. We chose this model because we support role hierarchies, but choose not to support the enforcement of separation of privilege that RBAC_2 requires. Sandhu et al describe the RBAC_1 model as having the following components:

- U, R, P, S (Users, Roles, Permissions, Sessions)
- $PA \subseteq P \times R$ (A many to many permission to role assignment)
- $UA \subseteq U \times R$ (A many-to-many user to role assignment)
- $user : S \rightarrow U$ (A function mapping a session s to a single user u)
- $RH \subseteq R \times R$ (A partial order on R; the role hierarchy)
- $roles : S \rightarrow 2^R$ (A function mapping each session s_i to a set of roles $roles \subseteq \{r \mid (\exists\, r' \geq r)\ \{(user(s_i),\ r) \in UA\}\}$ (which can change with time) and session s_i has the permissions $U_{r \in roles(s_i)} \{p \mid \{(\exists\, r'' \leq r')\ \{(p,\ r'') \in PA\}\}$

Defining Access Control Rules One option for defining access rules is to collect specific constraints which a compiler prepares separately from the compilation of the main code. We can then use a bytecode editor to weave the constraint bytecode into the main program bytecode as in Pandey and Hashii's system [5]. This system has global access constraints and selective access constraints. Selective constraints depend on who the caller is, while global constraints always apply. Pandey and Hashii acknowledge that a system with a default permit policy is less secure than one with a default deny policy, but more efficient for developers, because they need only mark invocations known to be dangerous. Explicit marking is risky, because missing even one dangerous invocation can invalidate the rest of the program's security. For this reason, marking is probably infeasible for large programs.

An abstract example of a global constraint is to deny the instantiation of object O when condition B is true, and an example of a selective constraint would be to deny the instantiation of object O by an object P when condition B is true. This could apply to method invocation as well as object instantiation. These constraints may be used with either a default permit or default deny policy. A concrete example would be to deny all writing to the local disk, by specifying the rule `deny (invoke File.Write)`. A boolean predicate can be much more detailed: to prevent reading of the UNIX /etc/passwd (password) file, use a rule that says to deny (File.Read) when the file name equals /etc/passwd.

The concepts of allow and deny are fairly flexible. Since we evaluate the conditions at the time of invocation, they can be used to, for example, cap the number of concurrent instantiations of a class, or limit the total number of times a method may be invoked. These are more applicable to general security than simple access control and authentication, because they could be used as a safeguard against a denial of service attack based upon repeatedly invoking expensive methods to consume all available processor time, or spawning so many objects that there is no more free memory.

Each object inherits all constraints from its superclasses. To safeguard against bypassing of security, a subclass may not have weaker access constraints than its superclass. It may, of course, have stronger constraints. Finally, a subclass may not be instantiated for a class that may not be instantiated, which prevents the trivial bypass scenario of simply making a subclass with zero or irrelevant changes in order to instantiate the initial class.

The major advantage of using access control rules is that we may declare security components in a fairly intuitive way. Unfortunately, the expressiveness of this system is limited, as acknowledged by its creators. It requires that the programmer specify either all permitted or all forbidden instantiations and invocations in advance, depending on the default.

Type Qualifier Inference We can examine Java code to infer *type qualifiers* – properties that augment the expressiveness of a standard type. This is also a system for fine-grained access control. An example of this approach is a `readonly` qualifier. A reference so marked states that the holder has no ability to change the object to which it refers. The `readonly` marking is “sticky" - if it applies to a field `x`, then it also applies to `x.f`. Since JQual, the tool used for this task, is not sensitive to context, Greenfieldboyce and Foster [12] extend it to have field sensitivity in place of “stickiness” – each instance of an object has its own field types.

A use case of a type qualifier is to use the `opaque` qualifier to represent integers which hold C pointer values, and the `transparent` qualifier for regular integers, so that transparent integers are never sent to opaque locations. This enhances type checking for the Java Native Interface (JNI), a tool with which Java programs may invoke C code. Using JNI, C pointers may need to pass through Java, such as from the windowing toolkit, through Java, back to the library to manipulate the windows in question. Since a pointer is really just an integer, it could be inadvertently manipulated, and passed back to the C library, leading to a crash or memory corruption. The fundamental idea is that untrusted (`transparent`) data may not be put into trusted (`opaque`) positions.

Type qualifiers bear some resemblance to a proposed extension to the Java language is called *Javari* (Java + *Reference Immutability*) and the `readonly` qualifier is much the same. It is possible to determine the `readonly` qualifiers at the same time as the `final` qualifier, although `final` is intended for variables and references, and not classes or methods. However, we note some differences from Javari: type qualifier inference does apply the read-only behaviour for every instance of a class marked `readonly`, and Javari allows generic immutability annotations on classes. Type qualifier inference also ignores qualifiers when examining overriding and overloading, since the inference process becomes very difficult [12].

Qualifiers in JQual are effective for tracking the flow of data throughout a program, from source (producer) to sink (consumer), because they are “sticky". Once applied, they remain attached to that piece of data throughout its operational lifetime. The programmer need only add qualifier annotations to the sources and sinks, and the inference strategy will work out the data flow and place the intermediate qualifiers. This can also work with program fragments, as long as source/sink information is given at entry and exit points to the fragment.

Security Passing Style *Security Passing Style* proposes a model in which designers use an English-like language structure to model security rules, and is used to generate specific rules using Abadi, Burrows, Lampson, and Plotkin (ABLP) logic. In addition to the concepts of principals and targets mentioned earlier, we introduce some new concepts from the ABLP subset used in [10]:

A *Statement* is something a principal “says", which can be explicit or implicit (e.g., caused by something the principal does). A principal *saying* a statement means it is possible to continue as if the principal supports that statement. Principals can, however, make untrue statements; we can only believe a statement if there are valid grounds to do so, and the speaker has the authority to make it. An example statement would be a principal *P* authorizing access to a target *T*: *P* **says** *Ok(T)*.

It is impossible to have a logical contradiction, because ABLP does not allow negation. Since there are no contradictions, the analysis avoids the problem of deciding which source is most credible.

Granting Authority is an operation that allows one principal to speak for another. If *A* **speaks for** *B*, then any statement that A makes is as if both A and B state it. This relationship is transitive.

The process of using these rules is relatively simple. A piece of code (at run-time, a frame) speaks for all parties that have “signed" the code, whether cryptographically, based on location, or otherwise. Frame credentials are simply the credentials of a principal. For targets, we create a dummy principal with the same name as the target; this principal makes no statements, but may be spoken for.

In [10], the virtual machine will check if a particular operation is allowed. This check will return true if so, and false otherwise. A permission check operation cannot be always decided for statements using the full ABLP grammar; however, the subset used in Wallach, Appel, and Felten's implementation is restricted such that we will certainly be able to decide and find the correct answer. The algorithm's strategy is a case analysis: divide the statements into three categories, and evaluate each category sequentially.

Class one is a direct statement of authorization – a principal explicitly allows access to a target. These statements are the most trustworthy and take priority. If we find such a statement, access is granted. A class two statement says one principal speaks for another, and we use these statements to create a directed graph with an edge between A and B if there is a statement that indicates that A speaks for B. A class 2 statement is insufficient on its own; we use class two statements in conjunction with class three statements. Class three statements have the form of quotations - A says that B says access is allowed. The evaluation procedure allows access if a quotation statement exists, and that statement is backed by the existence of a path in the speaks-for graph.

Those who advocate the security passing style implementation acknowledge that they do not handle dynamic code loading and generation well. Since less information is available at the time of analysis, security passing gives strictly worse (at best, nearly equal) performance in a world with classloading and other dynamic code than without it.

The security passing system for evaluating the security rules is equivalent to stack inspection, with an inductive proof provided in [10]. Stack inspection is a significant area in Java, and we will explore it next.

2.2 Stack Inspection

Stack Inspection is an important component of access control, because the stack provides information about the requester. A stack frame is considered a principal. As explained earlier, the rules described in section 2.1 use information about principals to make permit and deny decisions.

We cannot look solely at the direct caller of a given method; access can only be granted if all entities in the call stack should be granted access [13], as we next explain: indirection would allow attackers to bypass the security requirement. This is called the *Confused Deputy Problem*, and this problem warrants a short explanation:

A *Confused Deputy* is a program that runs with more than one authority; for example, a pay-for-usage compiler has both the invoker's authority to compile the program, and a higher level of authority to record billing information [14]. Compiler output could be used to destroy the billing information.

In the context of stack inspection, untrusted or malicious code could manipulate a trusted component into performing the desired operations (which would be denied if untrusted code asked directly).

Inspection Algorithms and Trade-offs The following pseudocode represents a basic stack-inspection algorithm [10]:

```
BEGIN METHOD CheckPrivilege
     FOR EACH stack frame
          IF local policy forbids access to target by stack frame
               THEN deny access
          END IF
          IF local policy permits access to target by stack frame
               THEN allow access
          END IF
     END FOR EACH
     IF default permit
          THEN allow access
     END IF
     IF default deny
          THEN deny access
     END IF
END METHOD CheckPrivilege
```

The worst-case run-time behaviour of this algorithm is obviously $O(n)$; that is, linear with the depth of the stack. Algorithms implemented in practice are not quite so simple: they iterate over each stack frame to find its protection domains (groups of principals), and then look at all the permissions in those domains [9]. Thus, the algorithmic behaviour is a function of the number of protection domains, the number of permissions, and the stack depth, but remains $O(n)$, as each element is examined once.

Checking privileges is rather expensive; implementers minimize checks for the sake of performance. In fact, the standard I/O library checks permissions only when a file is opened, and not for every access, presumably due to the expense [10]. To illustrate,

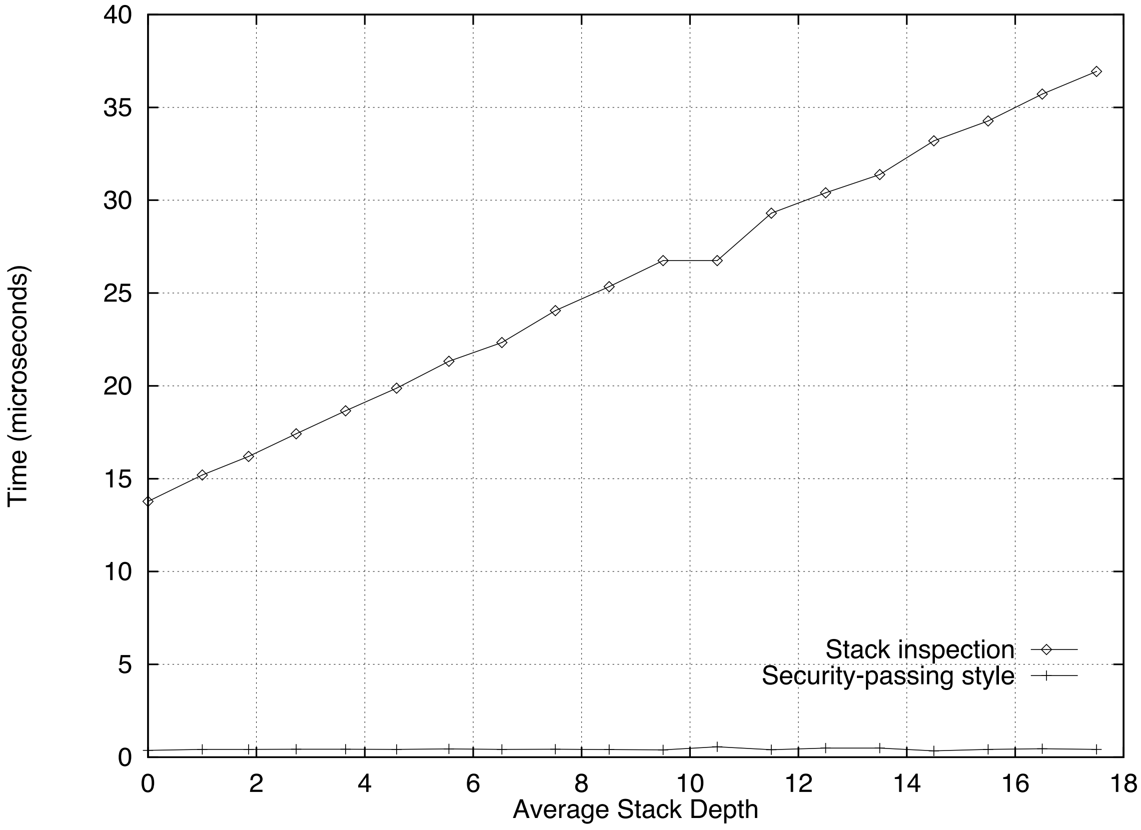


Figure 2.2: Stack Inspection Overhead vs Security Passing Overhead: Stack Inspection Overhead is Linear with Stack Depth; Security Passing Overhead is Unaffected by Stack Depth [10]

Figure 2.2 shows just how large the penalty for stack inspection is compared to the security-passing style.

Bartolleti, Degano, and Ferrari seek to lessen the burden of run-time stack inspection by performing a *Control Flow Analysis* - “a static technique for predicting safe and computable approximations to the set of values that the objects of a program may assume during its execution" [13]. We compose the stack information based on a series of contexts. Without running the program, we assemble a simulated stack “backwards" by looking at a call graph. Figure 2.3 presents a sample call graph.

The sample call graph enables an analysis on the callers of the `read()` method by following the call arrows backwards. Thus, one possible stack trace for `read()` in Figure 2.3 is:

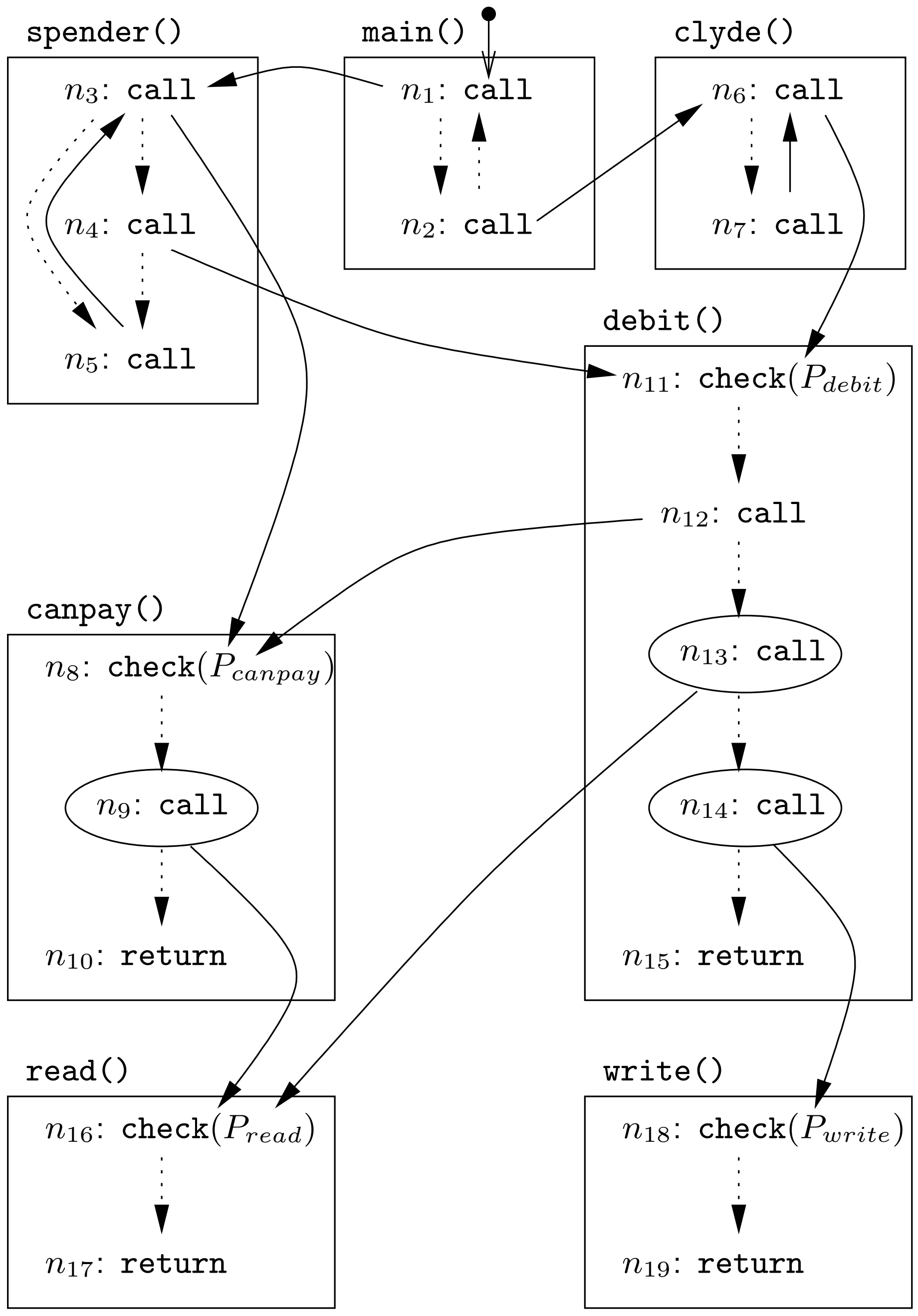


Figure 2.3: Sample Call Graph [13]

```
read()
canpay()
spender()
main()
```

There are more possible stacks leading to `read()` in the figure, but we omit them for brevity. We can make the approximations shown immediately above from the full set of these stacks, and then use these approximations to analyze the program. A test that holds true in static analysis will also hold true at run-time, but the reverse is not necessarily so. On this account, the analysis is likely to deny certain invocations which it should allow [13]. If we are interested in highest security, this may be preferable to allowing things that should be denied. We compute two subsets of the permissions - the allow and deny sets - for use at run-time. When checking permissions dynamically, it is only necessary to reach some point at which the access is explicitly allowed or denied, and that allow or deny statement is our final answer [13]. Cutting off subsequent checks is a significant savings (given a large stack and/or a large number of checks) over checking every level for every invocation. However, precomputing answers provides limited benefit when dynamic class loading is involved, because no preparatory work is done on the loaded classes.

Just like it is difficult or impossible to debug an optimized program, stack inspection constrains transformations on the programs it is used upon: it precludes inlining [1, 10]. Furthermore, we cannot be entirely sure that stack inspection is providing reasonable guarantees of carrying out its objectives. For instance, stack inspection does not do much to account for interactions outside of the call stack, such as results returned by untrusted code, mutable state, inheritance, side effects, concurrency, and dynamic loading [1], which we cannot simply ignore.

Stack inspection is not the only thing we can do with bytecode. We can also generate bytecode and load it into the application at run-time.

2.3 Dynamic Code Generation and Class Loading

Results from static analysis may be invalidated by dynamic code generation. Java programs can generate classes at run time, and then load them via dynamic class loading. Generated classes may supplement or replace parts of the program. Before continuing, we will examine how classes are generated and loaded. This is important because our proxy object system relies on code generation and loading.

Class Loading A class loader simply loads class files and makes them available to the program. A class file may be thought of as a “blueprint" for an instance of such a class. A program may have multiple class loaders and always has at least the default virtual machine-supplied class loader [15]. Classes belong to a class loader, and the removal of an owner results in the removal of its owned classes. Every class loader has the system types included automatically.

Different classloaders may be thought of as different domains for different classes. A potential source of issues in dynamic class loading is that two types may have the same name, as long as they do not share the same class loader [15]. Therefore, a class may be uniquely identified dynamically at any point in time by specifying a name and a classloader. Java programs often load classes behind-the-scenes – to instantiate any object, the virtual machine must first locate its class file, and then instantiation may proceed.

Dynamic class loading may be used to bypass security policies. A run-time class is determined by both name and classloader (at a given point in time), and the Java type-safety system relies on name-based static typing. Knowing this, an attacker could replace a class with his own version. Therefore, we must ensure that the class in question is the one we meant to load, and not simply another sharing its name. Liang and Bracha’s proposed solution to the security problem is simply to prevent the redefinition by never allowing a classloader to load the same class name twice.

Likewise, dynamic class loading may invalidate the results of a static analysis. If we replace a class with a new version, as in the previous example, the analysis conducted on the old version of the class will be out of date. If a function’s new version has side effects, but the old did not, then a check referencing the old version will indicate the method has no side effect, even though the new method has some. When new classes are loaded, the analysis has no information about them; a default-permit policy will let all methods run, even the impure ones, while a default-deny policy will forbid all methods, even the pure ones.

Class loading does not necessarily mean the run-time loading of pre-prepared class files; programs can generate code at run-time, and then classload these new or modified classes.

Run-Time Code Generation The standard approach to using run-time code generation for security is modification of the original objects to include security resources, as in [5]. Their implementation of access rules (as discussed in Section 2.1) involves a substitution within the bytecode. The abstract example of `deny access to R when B` in pseudocode is: `IF B THEN error ELSE access R END IF`. A bytecode editor rewrites

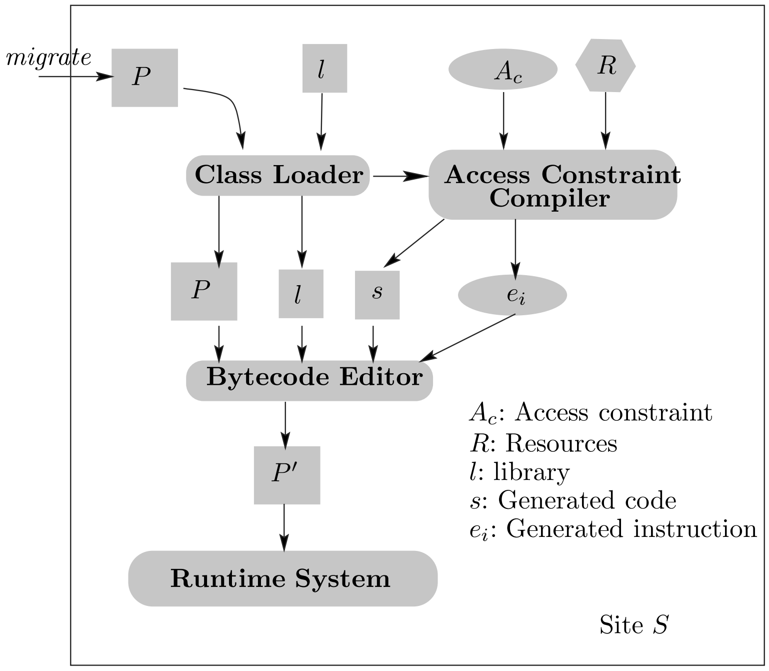


Figure 2.4: Bytecode Rewriting Data Flow Diagram [5]

the program bytecode as needed. Figure 2.4 shows a data flow diagram depicting assembly of the run-time objects.

Modification does not have a large performance penalty. Pandey and Hashii find that they spend minimal time editing code. The overall speed of execution is better than using the security manager, because inlining the security check is faster than retrieving the security manager and then using it. Figure 2.5 shows clearly that in the long run (above 100 000 method invocations), modification proves more efficient than the security manager. The one-time penalty of modification is clear at the far left of the graph, when binary editing is more time-consuming. As the number of invocations increases, at approximately 100 000 invocations, the time taken is equal. As the number of invocations increases, the cost of loading the security manager dominates the modification penalty.

One uncommon application of dynamic loading is the run-time insertion of intermediary objects for access control. In such a system, the term *guarded object* refers to an access-restricted object. A *sentinel object* controls access to the guarded object. Sentinel objects may be dynamically generated and inserted between objects. In Bryce and Razafimahefa's approach [16], *Bridge Classes* guard targets; they are intermediate objects generated as needed that span protection domains (spaces), a subclass of the original. Bridge classes (denoted B1, B2...) are necessary for the interaction of Objects

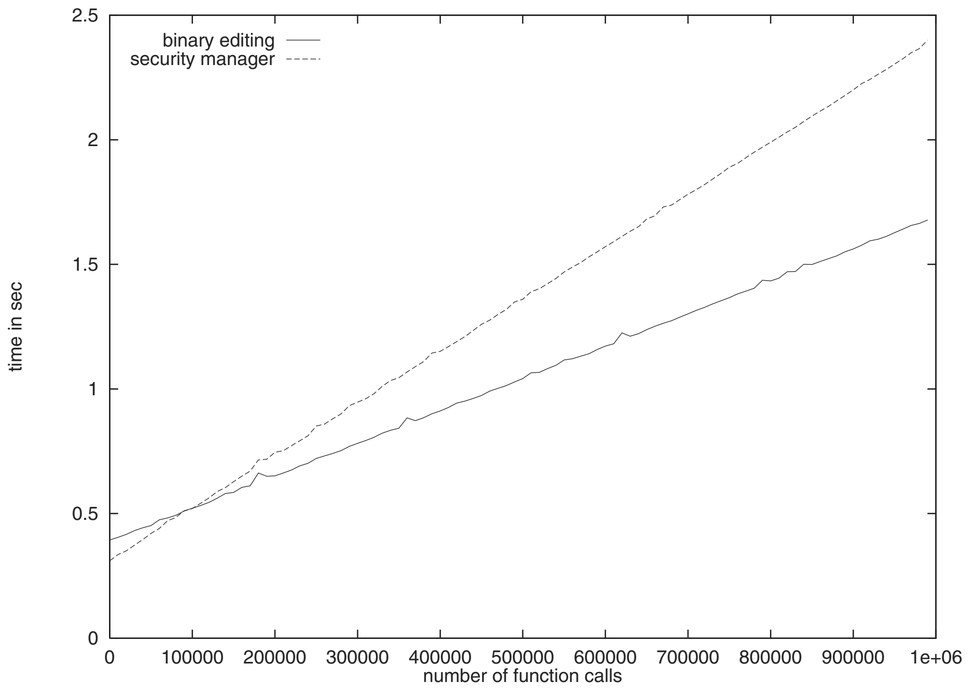


Figure 2.5: Access Control Execution Times [5]

(denoted O1, O2...) across space boundaries, as illustrated in Figure 2.6.

The simpler of the two cases is on the left of Figure 2.6. If object 2 (O2) wishes to communicate with object 3 (O3), it may do so directly, because they share the same space. As soon as object 2 attempts to communicate with object 1 (O1) across a boundary, then a bridge object (B2) is dynamically created and inserted between O2 and O1. B2 mediates all accesses between these objects. If O1 attempts to communicate with O2, a different bridge object (B1) is created and inserted between O1 and O2. The right side of Figure 2.6 expands on this example, introducing object 4 (O4) and a third space .

The main advantage of this approach is the dynamic insertion of intermediary objects. This approach requires no foreknowledge of the objects in various spaces. This is a serious advantage in light of dynamic class loading. The authors further find that the performance impact of their method is a net positive; it is more efficient than the copy-by-value (object duplication) that normal object sharing takes.

There are limitations to this approach. Classes marked `final` cannot be subclassed, preventing bridge object generation for such classes; Bryce and Razafimahefa suggest the solution to dealing with this problem is to use the classloader to ignore the `final` modifier and allow subclassing. Furthermore, the bridges do not control ac-

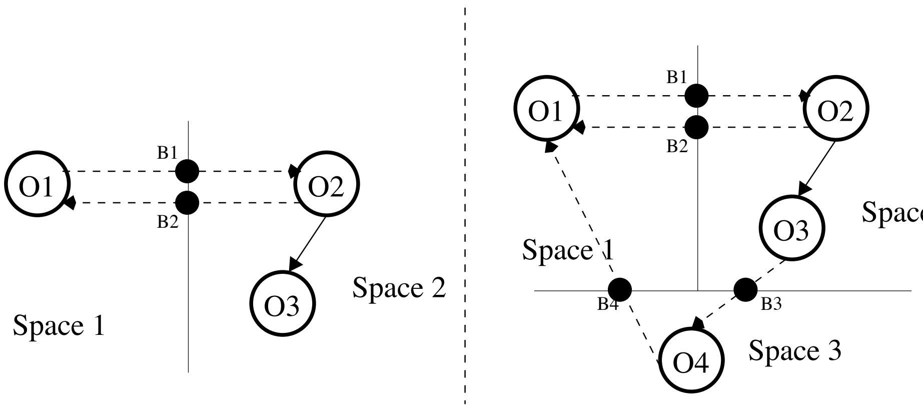


Figure 2.6: Spaces with Bridges [16]

cess to fields. Lastly, the authors discuss the possible handling of the Java Language (`java.lang.*`) objects, as those are shared between all spaces by default. They create special wrapper objects (e.g., `IOString` for `String`) for system classes.

Chapter 3

Safe and Unsafe Access Control

We present a method for providing security through proxy objects. In our technique, developers specify which methods to allow and deny; we use this information to automatically construct proxy objects. As they only expose a permitted subset of functions, proxies are safe by construction and may be passed to untrusted clients. We partition methods into two categories: safe (may be invoked by anyone) and unsafe (may only be invoked by trusted clients); we consider a more expressive system in Chapter 4. Our system is designed to work with Java RMI or in an EJB context.

3.1 Motivating Examples

Our technique is designed to handle cases where a heterogeneous software system needs to share objects with subsystems that are not fully trusted. In particular, our technique enables a system to expose parts of an object's functionality to a client; it may be undesirable or even dangerous to grant the client unrestricted access to the object. Consider the following domain object `D`, which stores data, and an untrusted client `C`:

```
class D {
     String getID() { ... };
     void setID(int i) { ... };
}
class C { // untrusted client
     boolean modify (D d) () { ... // C wants to manipulate d }
     }
}
```

One could avoid security concerns by ensuring that `D` objects are never made available to clients. Such a solution is unacceptable when clients require partial access to `D` objects. Allowing full access to `D` objects might be problematic: a simple programming error (or malicious intent) could result in data loss or corruption; we wish to limit the potential for error. Furthermore, full access might result in the client gaining unintended privileges, particularly by navigating the object graph and accessing objects that he or she is unauthorized to access.

In our example, `D` resides on the server, while the client code runs remotely, in a separate JVM. As `C` is untrusted, we do not wish to allow `C` access to all methods in `D`. The `setID` method will modify the `id` field—the index of this object. Note that the `setID` method is necessary so that the object can be restored from storage. We would like to deny access to that method, yet allow the accessor method `getID`. In many applications, we might consider only methods that do not make any writes safe to invoke, and allow access.

We could also write a restrictive interface and provide this interface to the client, with the data objects implementing the interface. Deriving this interface manually is time-consuming and error-prone, and it must be kept up to date when the original object changes. Our system supports the automatic generation of such interfaces, based on light-weight annotations.

3.1.1 Software-as-a-Service Scenario

As a second example, consider a software-as-a-service case with three parties: a software developer (the service provider), a store owner (service client), and store customers (who buy from the online store). The developer has created a customizable application, *ShipItems*, for online commerce. The clients (store owners) purchase an account and then customize their instance of ShipItems according to their needs. They use the application to organize and track product shipments to their customers. Customers place orders. In this example, the store owner owns a scuba diving shop, and she fills electronic orders from customers. Figure 3.1 shows relations between these entities.

As the software developer cannot foresee all the business rules of store owners (service clients), he can either guess at the rules the store owner wants (possibly providing rule templates for typical cases), or give out (full access to) the Java objects themselves. If the store owner cannot implement her business logic using the given rules or templates, then a human must verify every order before it goes out (costly and error-prone), or the owner must access the Java objects directly.

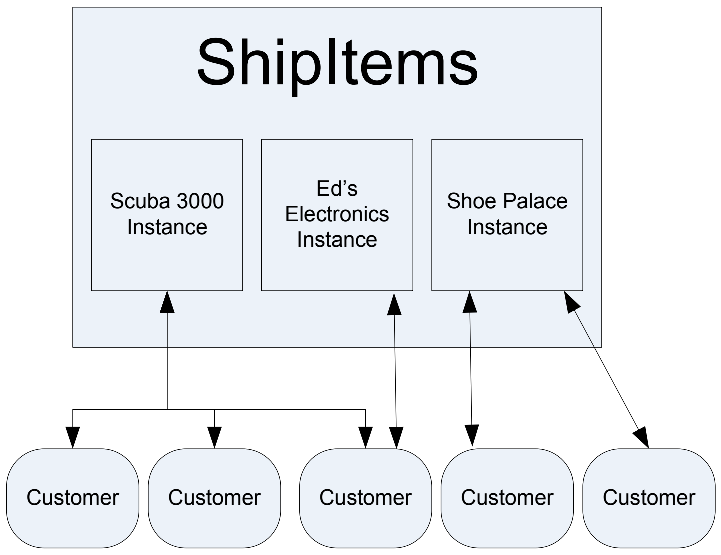


Figure 3.1: *ShipItems* with Multiple Customized Instances

Suppose ShipItems validates that the Postal/ZIP Code field is not empty. The store owner is willing to ship only within Canada. She tries to add a rule that the postal code must conform to the Canadian format (e.g., A1B 2C3). If the developer did not grant the owner the ability to define the formatting for this box, then the owner must verify her rule manually.

Alternatively, if given the address object, the store owner might change the name of the object representing Canada. A modification to the Canada object could negatively affect every user. Worse yet, by navigating the object graph, the store owner could see or alter confidential data of other stores.

This usage scenario is an instance of applying specific knowledge to a general system. The developer writes the general system (ShipItems) and sells it to the store owner, who applies her specific business rules to the system. We propose a way to allow the store owner to programmatically apply her business rules in general, while limiting her access to the Java objects.

3.1.2 Enterprise Java Bean Container Scenario

Our technique also applies in conjunction with the JBoss Application Server and the Hibernate persistence layer. Figure 3.2 presents an overview of the life cycle of a persistent domain object. Persistent objects are stored in the database and continue to exist through application restarts. A domain object `d` exists within a persistence

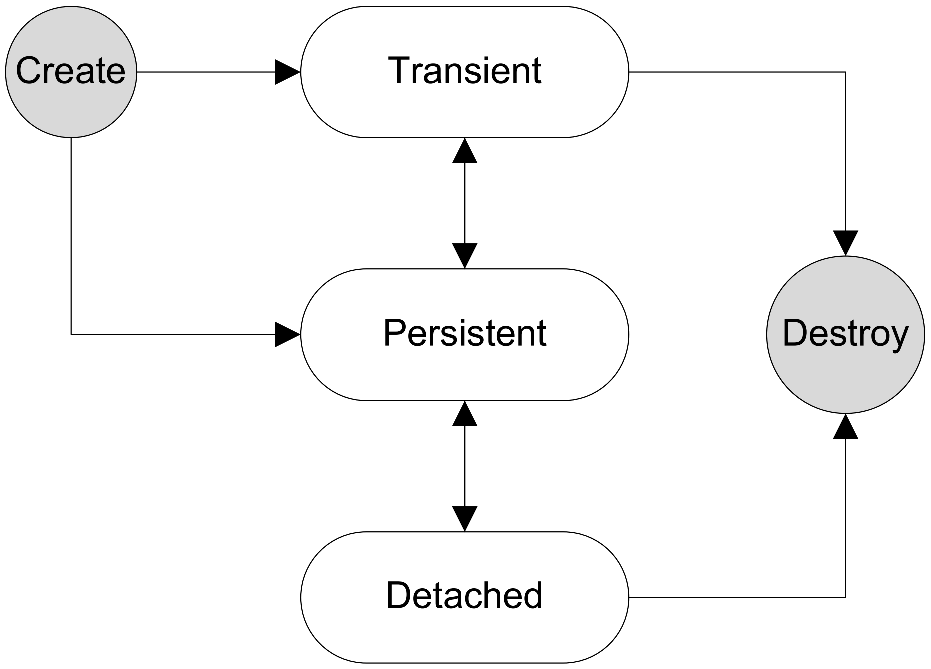


Figure 3.2: Domain Object Life Cycle [17]

context (in this scenario, the JBoss environment). If `d` is emitted to a remote caller, it is "detached". If the remote client modifies and returns `d`, then we must commit these changes to the database; if the caller never returns the object, then there are no commits.

Suppose an Enterprise Java Bean emits a persistent domain object. When the callers returns the detached object for storage, the server must verify its state. As the client cannot be trusted, the server is responsible for checking the validity of the object. Verification might be computationally expensive and the server's checks will fall out of date when the domain object's code is changed.

We propose a solution in which we can hand out instances of objects, like `D` from the first example, while preventing changes to key attributes. Since we are targetting production environments, we want to minimize overhead. Therefore, the security must minimize the ability of the users to do damage to the running system, but must allow the system to operate normally.

3.2 Proxy Objects

Our solution enables servers to give out Java objects such that recipients cannot adversely change the state of the system. We build custom proxy objects from the original Java objects, and give those to clients in place of the originals. Proxy objects are stand-ins generated from the originals that expose a subset of the original's functions. Normally, we require a pre-processing step between the Java Compiler and the RMI Compiler, except when used in conjunction with an EJB context, as described in section 3.3.

We support both default-permit and default-deny configurations. In a default-deny configuration, each callable function is specifically annotated as safe, and only annotated methods are available on the proxy objects. In a default-permit configuration, functions that should be unavailable to invoke are specifically annotated as unsafe, all remaining functions are available on the proxy objects. An object can have one or more proxies. Proxies contain no executable code; they delegate method execution to the original. Our solution enables object integrity preservation and provides fine grained access control.

There are three levels where we apply policy: global, class, and method. The global policy applies everywhere, but may be overridden. We support two global policies: default permit and default deny. Each class can be marked as default permit (safe) or default deny (unsafe), and the class annotation overrides the global policy for all instances of that class. Method-level annotations take precedence over class and global policy.

The developer sets global policy at compile time using a configuration flag. Most of the time, the developer will use a global default-permit policy, because it allows access to various methods on Java built-in classes, such as `Integer` or `DateTime`. Applications needing high security should allow only a limited subset of methods, so the developer will choose a default-deny global policy.

The developer annotates a method or class by adding `@Safe` or `@UnSafe` above it. These annotations impose no requirements upon the functions or classes they accompany; the annotation only restricts who may call the method, and has no implications on the method's behaviour. Below is the `@Safe` annotation; `@UnSafe` is identical except "Safe" is replaced with "UnSafe".

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Safe { }
```

If a method is annotated safe, then untrusted clients can invoke that method with their choice of arguments. Clients can (indirectly) invoke methods that are annotated unsafe as well, but only via other methods. Our rationale for this design choice is that clients then do not have direct control over the arguments with which these unsafe methods are invoked.

Consider a system with a global default permit policy; all methods may be invoked unless marked as unsafe. A class S is marked as default deny, and its `getID()` method is marked as safe to invoke. Because precedence goes from most specific to least specific, all methods are permitted globally, except for those in S (because S has a policy of default deny). The class-level policy of S is overridden by the annotation of `getID()`, which is the only callable method in S.

```
// Global Policy: Default Permit
@UnSafe // Class Policy: Default Deny
public class S {

    @Safe
    public int getID() { ... } // Permitted method

    public void setName(String newName) { ... } // Denied method
}
```

Any class C induces a derived interface, defined by its declared and inherited methods. So that proxy objects may be used in place of their corresponding real objects, a proxy object P implements the derived interface of the original class C. Our interface builder creates an interface I based on C's derived interface. Furthermore, a method appears in I only if allowed under the local policy.

Our solution is impervious to attacks using Java reflection. Consider a proxy object accessed remotely using RMI. Although the proxy object keeps a reference to the original, the original remains inaccessible, because reflection cannot be used on a remote object [18]. RMI semantics hide all fields of the implementation class from remote clients; fields do not appear on the client-side stub. We also require that arbitrary access to memory on the server side is not permitted; we require no assumptions about the client side virtual machine, as it is untrusted.

Example Consider object O with methods a and b. We have declared a safe and b unsafe. This system has trusted clients (T) and untrusted clients (U). Figure 3.3 shows the placement of the proxy object P in the system.

As a is safe, U can invoke a on the proxy, and the proxy invokes the corresponding method on the original object O. Method b is unsafe, so b cannot be invoked on the

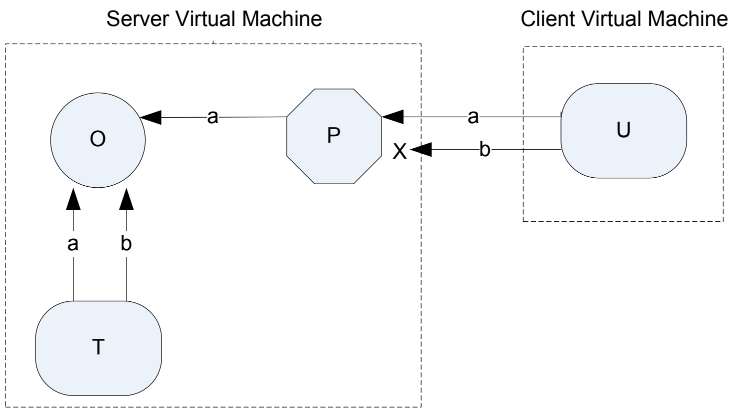


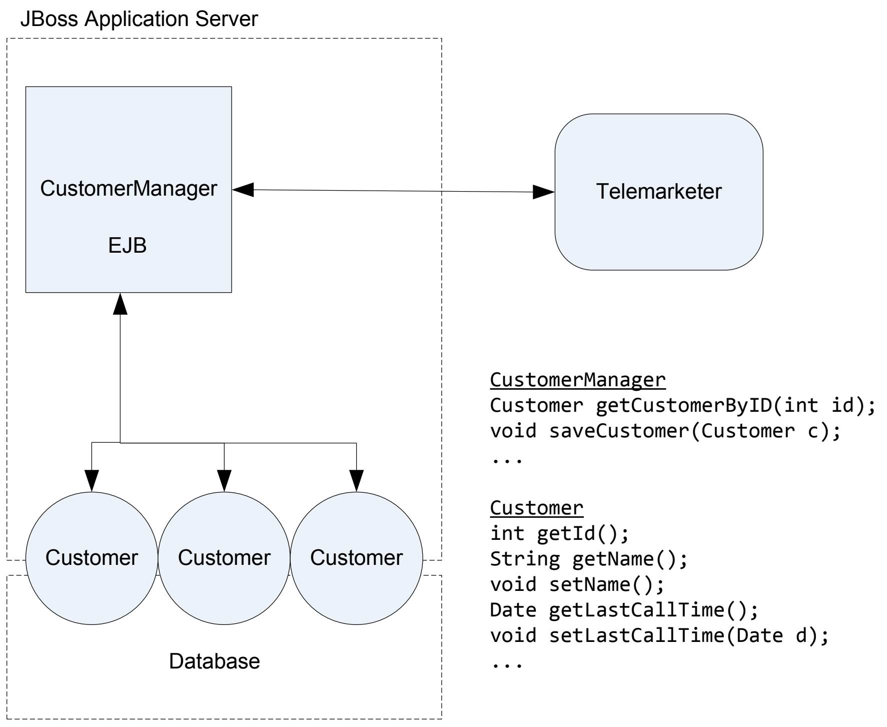
Figure 3.3: Proxy Object *P* Guards *O* from Untrusted Client *U*

proxy object because it does not exist. Conversely, because trusted client *T* can access *O* directly, it is free to invoke *b*.

If we were to use an automatic analysis to infer whether a method is safe, then we might conclude that any method that calls an unsafe method is itself unsafe. Suppose, however, even though *b* is unsafe in general, it is safe when invoked with specific parameters. In this scenario, it is meaningful and appropriate for a safe method to invoke an unsafe one, under controlled circumstances. The developer applies domain knowledge to enable more nuanced behaviour than an automatic analysis would provide.

3.2.1 Enterprise Java Bean Container Integration

Our security model also complements the EJB specification model. In an EJB scenario, a bean is a class encapsulating some functionality, that obeys certain properties (e.g., no-arguments constructor and getter/setter methods) such that it can be deployed using an application server. Where EJB allows for method-specific access controls on the beans themselves, our solution protects the domain objects returned through the interfaces. An example is a shopping cart (an EJB), which can be protected using EJB specification security model. The cart contains a number of books and proxy objects protect the books. Although we could make each book a bean, that is unreasonable; making a class a bean imposes certain requirements, such as the no-arguments constructor and getter and setter methods, which might not make sense for all objects. When proxy objects are employed in conjunction with an EJB container, there is no need for a pre-processing step between the Java compiler and the RMI compiler, because of the dynamic deployment system of the application server. As compared to making every single object a bean, our solution is light-weight, and does not impose



```
CustomerManager
Customer getCustomerByID(int id);
void saveCustomer(Customer c);
...

Customer
int getId();
String getName();
void setName();
Date getLastCallTime();
void setLastCallTime(Date d);
...
```

Figure 3.4: Proxy Object Application to JBoss Application Server

requirements on the classes to be protected.

We further explain the EJB scenario using a second example, shown in Figure 3.4. We are interested in the transitions between Persistent and Detached (as depicted in Figure 3.2). In this scenario, the client is the `Telemarketer`, who interacts with the `CustomerManager` EJB. A telemarketer is one of many untrusted clients.

The EJB Security model permits specification of method level restrictions on the beans. If a telemarketer is not permitted to make any changes to the object, then we may simply choose to deny telemarketers access to the `saveCustomer` method of `CustomerManager`. The telemarketer can modify the `Customer` object emitted through the EJB interface, and because it is never returned to the server to save, only his local copy is affected. Note that the telemarketer has arbitrary access to `Customer`, and that EJB does not protect `Customer` objects.

Suppose, however, that telemarketers are permitted to update the date and time that the customer was last contacted regarding purchases. If we allow the telemarketer access to the `Customer` object, he can make arbitrary changes and then return it to be saved via the `saveCustomer` method. We could instead make the returned objects of methods of the remote and local interfaces proxy objects, and therefore substitute a proxy when the object is about to become detached. Standing in for the originals, the proxy objects are detached and modified by the client. When returned for storage and converted back to the original, we can be assured that the original's state is valid without needing to run validity checks, because we know that the only changes that took place were the ones we explicitly allowed. In a normal EJB context, we would have to verify all the rules upon return of the object to the server, or disallow changes entirely.

3.2.2 Semantics of Annotation

We use First Order Logic [19] to express the semantics of our approach precisely.

To express that a method m has annotation a in its definition in class c, we adopt the predicate $\mathsf{annotatedMethod}\,(m, c, a)$. The counterpart corresponding to annotatedMethod for a class is $\mathsf{annotatedClass}\,(c, a)$. To express the global annotation, we adopt the constant **globalAnnotation**. To model the inheritance and method-definition aspects of Java, we adopt the predicates $\mathsf{inherits}\,(c_2, c_1)$ to express that class c_2 directly inherits c_1, and $\mathsf{definedIn}\,(c, m)$ to express that method m is defined in the class c.

$$\begin{aligned}
&\mathsf{annotatedClass}\,(c,a) \longleftarrow \\
&\qquad (\mathbf{globalAnnotation} = a) \wedge (a \neq a') \wedge \neg\mathsf{annotatedClass}\,(c,a') && (3.1)\\
&\mathsf{annotatedMethod}\,(m,c,a) \longleftarrow \mathsf{definedIn}\,(c,m) \\
&\qquad \wedge\, \mathsf{annotatedClass}\,(c,a) \wedge (a \neq a') \wedge \neg\mathsf{annotatedMethod}\,(m,c,a') && (3.2)\\
&\mathsf{annotatedMethod}\,(m,c_2,a) \longleftarrow \\
&\qquad \neg\mathsf{definedIn}\,(c_2,m) \wedge \mathsf{annotatedMethod}\,(m,c_1,a) \wedge \mathsf{inherits}\,(c_2,c_1) && (3.3)
\end{aligned}$$

Rule 3.1 says that a class that has no explicit annotation of its own receives the global annotation. Rule 3.2 indicates that a method without an explicit annotation receives the annotation of the class, as long as it is explicitly defined in that class. Finally, Rule 3.3 says that if a method is not defined in a class c, but rather in a superclass of c, the annotation the method receives in c is exactly the annotation it receives in its defining class.

For a semantics, we specify a model $\mathcal{M}$ and an environment l [19]. The set of concrete values, A, that we associate with $\mathcal{M}$ is $A = A_c \cup A_m \cup A_a$, where A_c is the set of classes, A_m is the set of methods and $A_a = \{\text{safe}, \text{unsafe}\}$. We associate one of the values from A_a with the constant **globalAnnotation**. We consider only environments in which our variables have the following mappings for our five predicatesannotatedMethod (m, c, a), annotatedClass (c, a), definedIn (c, m), inherits (c_2, c_1), and $a \neq a'$: the variables c, c_1 and c_2 must map to elements of A_c, m to an element of A_m, and a and a' to elements of A_a.

To compute $\mathcal{M}$, we begin with a model $\mathcal{M}_0$, with A as its universe of concrete values. In $\mathcal{M}_0$, we populate the relations that make our predicates concrete with those values that we glean from the Java code. For example, $\mathsf{inherits}^{\mathcal{M}_0}$ contains every pair $\langle c_2, c_1 \rangle$ for which class c_2 extends c_1. Similarly, we instantiate $\mathsf{annotatedMethod}^{\mathcal{M}_0}$ to those $\langle m, c, a \rangle$ tuples such that the method m has the annotation a in its definition in class c. We point out that an annotation can exist for a method in a class in the code only if the method is defined in that class. Also, there is at most one annotation in the code for each method in a class. There is also at most one annotation for a class.

We define $\mathcal{M}$ to be the least fixed point from applying the inference rules. Our algorithm α for computing $\mathcal{M}$ from $\mathcal{M}_0$ is as follows. We first apply Rule 3.1, which grows $\mathsf{annotatedClass}^{\mathcal{M}_0}$, repeatedly until no new entries are added. We then apply Rule 3.2 to grow $\mathsf{annotatedMethod}^{\mathcal{M}_0}$ until no new entries are added. Finally, we apply Rule 3.3, once again growing $\mathsf{annotatedClass}^{\mathcal{M}_0}$ with the same stopping condition. The result is $\mathcal{M}$. Note that α indeed computes the least fixed point, because it respects the topological ordering of the inference rules. The algorithm α runs in worst-case time that is quadratic in the number of classes, $|A_c|$.

To construct a proxy object for a particular class c, we can instead use a "bottom-up" algorithm that is linear in $|A_c| + |A_m|$. We first identify all methods that are defined in c by a breadth- or depth-first search of the inheritance graph in reverse, starting at c. We can then identify the annotation of each method in c in constant time.

Correctness $\mathcal{M}$ is sound and complete. What we mean by sound is that a method has at most one annotation in $\mathcal{M}$; we never infer any contradictions. What we mean by complete is that a method that is defined in a class has at least one annotation in $\mathcal{M}$. Soundness follows directly from inference Rules 3.2 and 3.3. We make the following assertion with regards to completeness.

Proposition 1 *For every method $m \in A_m$ and class $c \in A_c$, there exists $a \in A_a$ such that $\mathcal{M} \models_l (c, m) \in \mathsf{definedIn}^{\mathcal{M}} \longrightarrow (m, c, a) \in \mathsf{annotatedMethod}^{\mathcal{M}}$.*

Proof By contradiction. Consider the algorithm α that we discuss above that we use to construct $\mathcal{M}$ and the inference rules. We assume that $\mathcal{M} \models_l (c, m) \in \mathsf{defined}^{\mathcal{M}}$ and $\mathcal{M} \models_l (m, c, \text{safe}) \notin \mathsf{annotatedMethod}^{\mathcal{M}} \wedge (m, c, \text{unsafe}) \notin \mathsf{annotatedMethod}^{\mathcal{M}}$. By Rule 3.2, $\mathcal{M} \models_l (c, \text{safe}) \notin \mathsf{annotatedClass}^{\mathcal{M}} \wedge (c, \text{unsafe}) \notin \mathsf{annotatedClass}^{\mathcal{M}}$. Then, by Rule 3.1, $\nexists\, a$ such that $\mathsf{globalAnnotation}^{\mathcal{M}} = a$, a contradiction. $\square$

3.2.3 Semantics of Invocation

To express that a method m in class c may be safely invoked, we adopt the predicate $\mathsf{canSafelyInvoke}\,(m, c)$. We use the predicate $\mathsf{invokes}\,(m_1, m_2)$ to indicate that method m_1 invokes method m_2. We introduce a constant, **safe**, to indicate the safe annotation (that is, a method may be invoked by untrusted clients). We now present our inference rules. Our semantics are specified as in the previous section. In our model $\mathcal{M}$, $\mathbf{safe}^{\mathcal{M}} = \text{safe}$.

$$\mathsf{invokes}\,(m_1, m_3) \longleftarrow \mathsf{invokes}\,(m_1, m_2) \wedge \mathsf{invokes}\,(m_2, m_3) \tag{3.4}$$

$$\mathsf{canSafelyInvoke}\,(m, c) \longleftarrow \mathsf{annotatedMethod}\,(m, c, \mathbf{safe}) \tag{3.5}$$

$$\mathsf{canSafelyInvoke}\,(m_2, c) \longleftarrow \mathsf{canSafelyInvoke}\,(m_1, c) \wedge \mathsf{invokes}\,(m_1, m_2) \tag{3.6}$$

Rule 3.4 merely indicates the transitive nature of the invokes relation. Rule 3.5 indicates that a safe method may be safely invoked. Rule 3.6 expresses that safe methods can invoke unsafe methods.

3.3 Implementation

We have also created an implementation of our system consistent with the formal semantics. Bytecode generation and modification lie at the heart of our implementation. We modify RMI-enabled classes and generate interface class files. As we are not producing executable code, we can make the changes without resorting to a full-featured code generation library. Our code generation routine is based on interface generating code published by Eamonn McManus [20].

The crux of our implementation is modifying the interface's set of methods. We leave unmodified all methods that return a simple type (e.g., `int`) or a `String`. Otherwise, we replace the return type with a proxy object.

Our Interface Builder examines any class C and returns its modified derived interface. The Interface Builder is provided only the set of methods which are allowed.

A proxy object is an instance of the `ProxyObject` class. This class handles all method invocations on the proxy object. If the method exists in the modified derived interface I, then the proxy will pass the invocation on to O; O executes the method as requested. If proxy objects are found in the parameters, they are replaced with their corresponding originals. If a method was not permitted, it did not appear in I and hence is not available for invocation on the proxy P. Therefore, only the methods that we consider safe may be invoked.

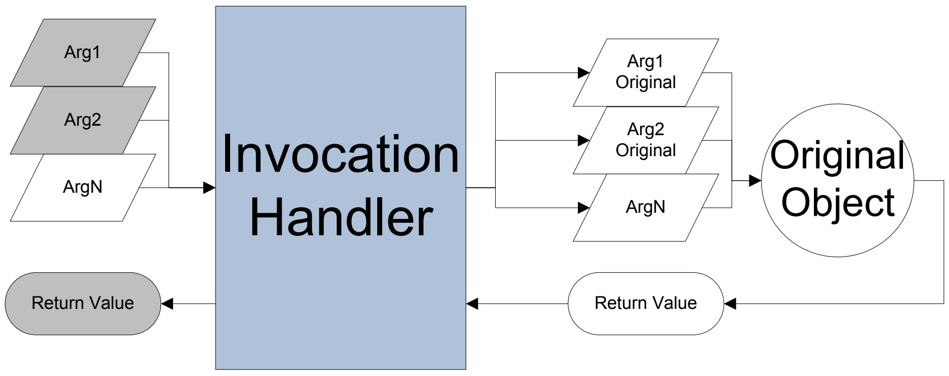


Figure 3.5: The Invocation Handler substitutes Proxies for Originals, and Vice-Versa

The `ProxyObject` class is also responsible for intercepting the return of a non-proxy object and performing the appropriate substitution. This class is registered as the invocation handler for all methods on the proxy class. Invocation handlers execute before the actual method is called and after the method returns some value.

Consider Figure 3.5; an example of the invocation handler's functioning. Arguments 1 and 2, shaded in the figure, are proxy objects, so the invocation handler replaces them in the arguments set with their corresponding original objects. Argument N is not a proxy, so it is unaltered. Arguments 1 through N are passed to the original object on which the call was to take place, and execution there proceeds as normal. There is a return value, and the invocation handler examines it, and it is not a simple type. Thus, the invocation handler returns a proxy object in place of that return value. This proxy object is dynamically generated, and the interface for it may need to be dynamically generated if an object of this type has not had a proxy before.

When using our technique in an EJB context, the system runs without a compile-time component, because the application server does dynamic deployment of our code. Interface generation, as described above, takes place at run-time, and the interfaces are dynamically class loaded. We make no modifications to the original object's source code. Otherwise, our technique also uses a compile-time preparation step that takes place between compiling the source files and running the RMI Compiler (`rmic`). See Figure 3.6 for a step-by-step overview of the process.

Our pre-processor searches the code base and finds RMI-accessible objects, examines each of these objects and derives its remote interface. This derived interface enforces the `@Safe` and `@UnSafe` rules. The pre-processor saves the interface as a class file, and modifies the original object's class file to indicate that it implements the derived interface. At run-time, the server simply enters a dynamically-created proxy object of the correct type into the RMI server. No modifications to the Java compiler or `rmic` are

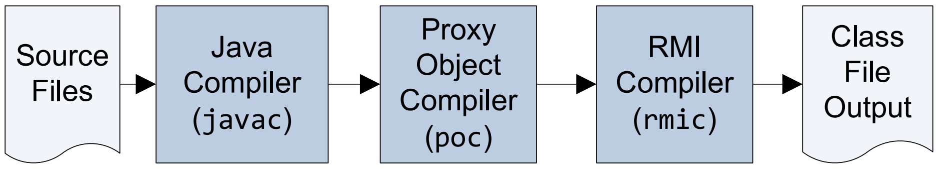


Figure 3.6: The Proxy Object Compiler Runs Between the Java and RMI Compilers

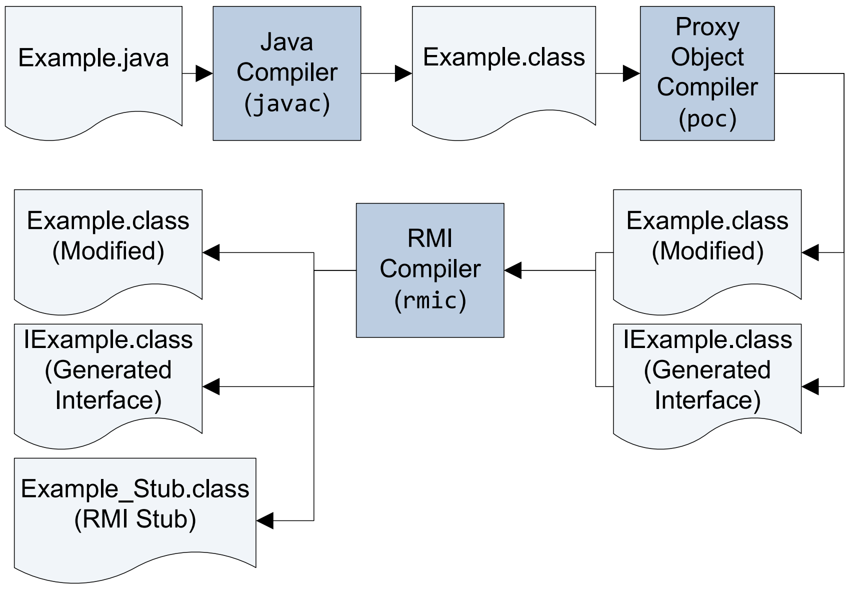

Example.java
Java Compiler (javac)
Example.class
Proxy Object Compiler (poc)
Example.class (Modified)
IExample.class (Generated Interface)
RMI Compiler (rmic)
Example.class (Modified)
IExample.class (Generated Interface)
Example_Stub.class (RMI Stub)

Figure 3.7: Proxy Object Compiler Processing `Example.java`

necessary. Figure 3.7 depicts this process on class `Example`.

A developer has written the `Example.java` file, and compiled it into a class file using `javac`. We identify `Example` as being a remote-accessible object (it extends `UnicastRemoteObject`), and give it to the proxy object compiler (`poc`). The `poc` examines `Example` and derives interface `IExample`, which contains only safe methods. The `poc` modifies `Example.class` so it implements `IExample`. Once our modifications are complete, we invoke `rmic` on the resultant class files, and the RMI compiler produces `Example_Stub.class`.

Chapter 4

Role-Based Access Control via Proxy Objects

We generalize our system from the binary access trusted and untrusted roles of the previous chapter to consider different levels of permission and role-based access controls (RBAC). The previous chapter supported only two roles: trusted and untrusted. With a multiple-role system, we could annotate methods with the roles allowed to access them; for example, we could annotate `createSalaryReport` with `@Accounting` and `@HumanResources`. In that respect, the methods in the salary report would be accessible to different roles. Moving to role-based access controls greatly increases the flexibility of the security rules.

We bind clients to roles, and clients have access to only certain methods, based on the roles to which a client is authorized. This is a significant improvement over our work in chapter 3, which only supported coarse-grained "accessible" or "not accessible" annotations for methods, interfaces and classes.

4.1 Example & Approach

We next present an example, and discuss our approach to realizing RBAC in Java. We discuss the specification of RBAC policies in our approach and how annotations denote roles. We discuss also the security benefits from using our approach. In Section 4.1.2, we present precise semantics of our approach in first-order logic.

In our approach, we build proxy objects from the real objects, and allow partially-trusted users to directly access the proxies only; fully trusted system components may access the original objects. For each role, we derive a proxy object. A client may

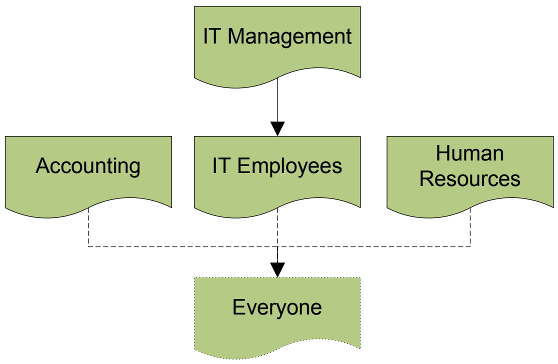


Figure 4.1: Roles in the Separation of Privilege Scenario. Arrows indicate a subsumption relationship.

be bound to more than one role, and may successfully request the proxies that are associated with every role to which the client is bound. A proxy implements only those methods to which the role with which it is associated is authorized. Whether a role is authorized to a method or not is inferred from annotations in the source code.

Annotations in Java are metadata regarding the code they accompany. In the context of this paper, an annotation indicates a set of roles. Annotations may be associated with an interface, a class, or a method (within an interface or a class). We first discuss how we derive the effective annotation on a method informally, and more precisely in Section 4.1.2. The only clients to which a method is authorized are those that are members of roles in the method's effective annotation.

We also provide a mechanism for specifying the roles to be used in annotations. Developers may associate roles with one another in a role hierarchy [2]. A role hierarchy, denoted as RH, is a partial ordering of the roles. If $\langle r_1, r_2 \rangle \in RH$, then we say that r_1 subsumes r_2. The consequence is that if r_2 is in the effective annotation of a method m, and a client is authorized to r_1, then the client is authorized to m.

Example We present an example of an RBAC policy, and a discussion of its use in a snippet of Java code. In this scenario, we wish to assign some rights to one set of users, and different rights to another set of users. Figure 4.1 presents a role hierarchy, for use in the example.

We have five roles, **IT Management**, **Accounting**, **IT Employees**, **Human Resources** and **Everyone**. The arrows indicate role *subsumption*. That is, everyone in **IT Management** is an IT Employee (and therefore in **IT Employees**), and every member of **Accounting**, **IT Employees** and **Human Resources** is a member of **Everyone**. (We use the term “subsumption” rather than “inheritance” as is customary in RBAC. We reserve the term “inheritance” for Java’s notion of inheritance.)

The Java code we intend to protect follows.

```
struct Item {
    String name;
    double quantity;
    String unit;
    double price;
}
class Order {
// Constructor
    Order(List<Items> items) { ... }

    void approve() { ... }
}
```

The code includes a class `Order`, which has two methods, `Order()` and `approve()`. The method `Order()` is called a constructor, and executes upon creation of objects of class `Order`. Clients call `approve()` to approve an order that exists.

We require that only members of **ITEmployees** may create orders, and only members of the role **Accounting** may approve orders. We assume that an object of type `Order` on which all clients operate resides on a server, and all clients invoke methods on it via RMI.

The earlier system, with only the trusted and untrusted roles, is unable to express this policy. Suppose IT Employees were considered untrusted, and Accounting users were trusted, and the `approve()` method were annotated unsafe. In that case, although IT Employees would have the appropriate rights, users of Accounting would be able to invoke both the constructor and the `approve()` method.

We next discuss how our system supports and enforces this policy.

Effective annotation As we mention above, we allow developers to annotate methods, interfaces, and classes. We infer the effective annotation on a method as follows. We have adopted the approach of “most specific annotation applies” in how we infer the effective annotation on a method. We present these rules precisely in Section 4.1.2.

If a method has an explicit annotation when it is defined in a class, then that explicit annotation is the effective annotation of the method. If a method is inherited from a superclass, but is not explicitly defined in the subclass, then the annotation in the superclass is its effective annotation. The reason is that in Java, when a method that is defined only in the superclass is invoked on an object of the subclass, it is the method from the superclass that is invoked.

If a method is defined in a class but has no annotation on it, and the class has an annotation, then the effective annotation on the method is the annotation on the class. We point out that a class may have no annotation. In this case, we do not infer its annotation from a superclass, if that class indeed inherits another class. The reason is that, as a design choice, we have chosen to disassociate the semantics of security from the intended semantics of class inheritance.

We allow interfaces to be annotated as well. An interface as a whole may be annotated, as may individual methods that are declared within it. If the interface is annotated, then that annotation affects methods declared within the interface only if a method does not have an annotation of its own within the interface. Annotations on an interface do not carry over to subinterfaces. If a method is re-declared in a subinterface, then its annotation in the subinterface affects classes that implement the subinterface.

Annotations on interfaces have different semantics than annotations on classes and methods within classes. Because Java interfaces constrain the contents of classes which choose to implement them, we decided to interpret annotations on interfaces as constraints on the classes implementing those interfaces.

Suppose we infer an effective annotation of the set of roles R on a method m that is defined in class c. Suppose also that we infer an annotation R' on all interfaces that c implements that declare m. Then, we require that every role in R' subsumes some role in R. Consequently, a client that is authorized to a role in R' has access to the method. R may include other roles, and consequently authorize other clients as well; R' is a lower bound on the clients that may access the method. We impose this constraint at compile-time—if some role in R' does not subsume any role in R, then we declare a compile-time error.

Security As we mention in this chapter's introduction, we target our access control approach to RMI-accessible objects only. There exist some design approaches that allow granularity at the level of instances of the class, but in our design, all instances of a class are treated equally. Our threat agents are remote clients. We assume that the potential attacker is a client in a different Java Virtual Machine (JVM) [18] whose only way of accessing objects is via RMI. It is possible to adapt our approach to local clients to provide more limited access control. However, other approaches such as stack inspection [1] combined with resistance to code-modification may be more effective in thwarting local attackers.

By default, our policy is "deny all." Annotations can be seen as selective "allow" rules. As only the methods to which a client is authorized are implemented in the proxy's interface (see the next section for details), our approach hides even the existence of other methods in an object. We see this as a performance benefit. From the standpoint of security, we have some measure of confidentiality in addition to access control. From the standpoint of performance, the server needs no run-time check to verify whether a client's method invocation is authorized.

We point out also that our solution remains impervious to attacks that use Java reflection. Consider a proxy object that is accessed remotely using RMI. As we stated earlier, reflection cannot be used on remote objects, and RMI hides all fields of the object at the server from remote clients; fields do not appear on the client-side stub. Thus, reflection cannot be used to access methods to which a user is not authorized.

4.1.1 Annotations

In this section, we discuss how the developer defines roles and the role hierarchy, and how developers insert annotations into the program's code.

Roles and role hierarchy We declare a role using what we call a meta-annotation. The following code illustrates how to declare the role **ITEmployees** from Figure 4.1.

```
@Role
@Retention(RetentionPolicy.RUNTIME)
public @interface ITEmployees { }
```

This code segment is an example of a basic role annotation. `@Role` is a standard Java annotation, used as metadata. Annotations are syntactically denoted by the `@` symbol, and thus the first line applies the `Role` annotation to the current class. The second line is an annotation we add for the compiler that specifies that the `ITEmployees` annotation's presence should be observable at run-time. The compiler will therefore propagate that annotation to the output class file, and the virtual machine will load the annotation with the class. Thus, when executing, the program can check if this annotation is present on a method or class. Finally, the syntax for declaring an empty annotation appears on line three. Empty annotations serve solely as markers attached to a method, interface, or class.

We choose to annotate a role declaration with the roles it subsumes as our method of declaring the subsumption relationships. This method is simple and concise, and at compile-time, we can automatically compute the role hierarchy based on the subsumption annotations. The following snippet of code expresses how we declare the role subsumption relationship; **ITManagement** subsumes **ITEmployees**. This means a user in IT management has all of the rights of a user in IT employees. Note that the following annotation declares a new role and re-uses an existing role in a subsumption relationship.

```
@Role
@ITEmployees
public @interface ITManagement { }
```

To declare that two roles r_1 and r_2 subsume r_3, we annotate the declarations of r_1 and r_2 with r_3—we write "@r_3" above "public @interface r_1 { }" and above "public @interface r_2 { }". To declare that r_4 subsumes r_5 and r_6, we write @r_5 @r_6" above "public @interface r_4 { }". Subsumption is transitive, so in a system where r_7 subsumes r_8 and r_9, and a role r_{10} is introduced which should subsume roles r_7, r_8, and r_9, it suffices to declare that r_{10} subsumes r_7.

Annotations on classes, methods and interfaces The following snippet of code illustrates how we annotate the class `Order` and its method `approve()`. `Order` is annotated with the role **ITEmployees** and `approve()` is annotated with the role **Accounting**.

```
@ITEmployees
public class Order {
// Constructor
    Order(List<Items> items) { ... }

    @Accounting
    void approve() { ... }
}
```

We allow the developer to annotate a class or method with more than one role, by simply applying multiple role annotations to that class or method.

As we mention above under "Effective annotation," we may annotate interfaces and method declarations within interfaces as well. The following snippets show an example of a method, `getSalary()`, that is declared within an interface `IHiringRequest`, annotated with the role **HumanResources**. We show also a class, `HiringRequest`, that implements `IHiringRequest`. As we clarify in our earlier discussions under "Effective annotation," and more precisely in the following section, we require, at compile-time, the effective annotation on `getSalary()` to include **HumanResources**. In this example, we meet this requirement by explicitly annotating `getSalary()` in its definition in the class `HiringRequest` with **HumanResources**.

```
public interface IHiringRequest {
   @HumanResources
   public Money getSalary();
   ...
}

public class HiringRequest implements IHiringRequest {
   @HumanResources
   public Money getSalary() { ... }
   ...
}
```

The expected annotation on `HiringRequest.getSalary()` is `@HumanResources`; that is, users who are members of the human resources group must be able to invoke `getSalary()`. The annotation's presence on the interface's method definition implies all implementations of `getSalary()` must also be annotated with `@HumanResources` or any role subsumed by `@HumanResources`; if any implementation is not so annotated, that is a compile-time error.

4.1.2 Semantics

We expand our First Order Logic semantics from the previous chapter to express the semantics of our Role-Based Acceess Control approach precisely.

Annotations in class and interface definitions Our annotation inference rules use the following predicates: (1) $\mathsf{effectiveAnnotation}(m, c, r)$ is used to express that the role r is in the effective annotation of method m in class c; (2) $\mathsf{subsumes}(r_1, r_2)$ is used to express that the role r_1 subsumes r_2. We note that the subsumption predicate is reflexive, so $\mathsf{subsumes}(r_1, r_1)$ is always true. (3) $\mathsf{annotatedCorI}(ci, r)$ is used to express that the class or interface ci is annotated with r—the annotation may be explicit or inferred; (4) $\mathsf{annotatedMethod}(m, ci, r)$ is used to express that the method m is explicitly annotated with the role r in the class or interface ci; (5) $\mathsf{definedIn}(ci, m)$ is used to express that the method m is defined or declared in the class or interface ci respectively; and, (6)

$\mathsf{extends}(ci_2, ci_1)$ is used to express that the class or interface ci_2 extends (or inherits) the class or interface ci_1.

Note that our predicate $\mathsf{definedIn} ci, m$ is true for at most one ci per m. For example, consider a scenario where c_3 extends c_2 and c_2 in turn extends c_1; both c_1 and c_2 define method m. We say that $\mathsf{definedIn} c_3, m$ is false because m is not explicitly defined in c_3. $\mathsf{definedIn} c_2, c_3.m$ is true, as there is an explicit definition for m in that class; however, $\mathsf{definedIn} c_1, c_3.m$ is false. This follows the Java language semantics; a call to $c_3.m()$ results in an effective invocation of the method $c_2.m()$ and not $c_1.m()$.

Having defined the predicates, we next use them to state the rules defining effective annotations.

$$\mathsf{subsumes}(r_3, r_1) \leftarrow \mathsf{subsumes}(r_3, r_2) \wedge \mathsf{subsumes}(r_2, r_1) \tag{4.1}$$

$$\mathsf{annotatedCorI}(ci, r) \leftarrow \mathsf{annotatedCorI}(ci, r') \wedge \mathsf{subsumes}(r, r') \tag{4.2}$$

$$\mathsf{effectiveAnnotation}(m, ci, r) \leftarrow \mathsf{annotatedMethod}(m, ci, r) \tag{4.3}$$

$$\mathsf{effectiveAnnotation}(m, ci, r) \leftarrow \mathsf{effectiveAnnotation}(m, ci, r') \wedge \mathsf{subsumes}(r, r') \tag{4.4}$$

$$\begin{aligned}&[\mathsf{effectiveAnnotation}(m, ci, r) \leftarrow \mathsf{annotatedCorI}(ci, r)] \leftarrow \\ &\qquad\qquad \mathsf{definedIn}(ci, m) \wedge \forall r \neg \mathsf{annotatedMethod}(m, ci, r)\end{aligned} \tag{4.5}$$

$$\begin{aligned}&\mathsf{effectiveAnnotation}(m, ci_2, r) \longleftarrow \\ &\qquad \neg\mathsf{definedIn}(ci_2, m) \wedge \mathsf{effectiveAnnotation}(m, ci_1, r) \wedge \mathsf{extends}(ci_2, ci_1)\end{aligned} \tag{4.6}$$

Rule (4.1) expresses that role subsumption is transitive. Rule (4.2) implements the role subsumption relation on class and interface annotations: if r subsumes r' and a class or interface ci is annotated with r', then we infer that ci is annotated with r as well. Rule (4.3) ensures that explicit annotations are also effective annotations: role r is in the effective annotation of method m in class c if it is an explicit annotation on m's definition in c. Rule (4.4) implements role subsumption for effective annotations: if r subsumes r' and r' is in the effective annotation of m in c, then so is r. Rule (4.5) causes class annotations to affect unannotated contained methods: if a method m, defined in class c, has no annotations, then the annotations on c are the effective annotation on m. Finally, Rule (4.6) computes effective annotations for inherited methods: if method m is not defined in class or interface ci_2, but has an effective annotation in a class or interface ci_1 that ci_2 extends (or inherits), then that effective annotation is also an effective annotation for m in ci_2.

Constraints from interfaces As we mentioned earlier, annotations on interfaces and method declarations in interfaces are "at least" constraints on the effective annotations on their implementing methods in classes. We specify the rule pertaining to annotations on interfaces below. We introduce the predicate $\mathsf{expectedAnnotation}(c,m,r)$ to express the constraint that r is required to be in the effective annotation of m in c. We adopt the predicate $\mathsf{implements}(c,i)$ to express that class c implements interface i.

$$\mathsf{expectedAnnotation}(c,m,r) \leftarrow \mathsf{implements}(c,i) \wedge \mathsf{effectiveAnnotation}(m,i,r) \tag{4.7}$$

Rule 4.7 expresses that if r is in the effective annotation of method m in interface i, and class c implements i, then r is required to be in the effective annotation of m in c. Note that m has an effective annotation in i only if (1) m is defined in i, or (2) i extends an interface that defines m. Also, because c implements i and m is defined in i, Java ensures that m must exist in c as a method, either by inheritance or by explicit definition.

Model Rules (4.1) – (4.7) have a well-founded semantics [21]. We use negation; however, the negation is stratified.

For a semantics, we specify a model, $\mathcal{M}$, and an environment or look-up table, l. The set of concrete values, A, that we associate with $\mathcal{M}$ is $A = A_i \cup A_c \cup A_m \cup A_r$, where A_i is the set of interfaces, A_c is the set of classes, A_m is the set of methods and A_r is the set of roles. We assume that the sets A_i, A_c, A_m and A_r are disjoint. We consider only those environments in which our variables have the following mappings: the variables r, r', r_1, r_2 and r_3 map to elements of A_r, ci, ci_1 and ci_2 map to elements of $A_c \cup A_i$, c maps to an element of A_c, i maps to an element of A_i and m maps to an element of A_m.

To compute $\mathcal{M}$, we begin with a bare model $\mathcal{M}_0$, with A as its universe of concrete values. We populate the bare model $\mathcal{M}_0$ with relations that make our predicates concrete, by extracting values directly from the Java code. For example, when the class or interface ci_2 extends (in the Java code) ci_1, we add $\langle ci_2, ci_1\rangle$ to $\mathsf{extends}^{\mathcal{M}_0}$. Similarly, we instantiate $\mathsf{annotatedMethod}^{\mathcal{M}_0}$ to those $\langle m, ci, r\rangle$ tuples such that the method m has the annotation r in its definition in the class or interface ci.

We define $\mathcal{M}$ to be the least fixed point from applying the Rules (4.1) - (4.7) starting at $\mathcal{M}_0$. The following algorithm α computes $\mathcal{M}$. First, populate the sets $\mathsf{definedIn}^{\mathcal{M}_0}$, $\mathsf{subsumes}^{\mathcal{M}_0}$, $\mathsf{annotatedMethod}^{\mathcal{M}_0}$, $\mathsf{annotatedCorI}^{\mathcal{M}_0}$ and $\mathsf{extends}^{\mathcal{M}_0}$ from the Java code. Observe that $\mathsf{definedIn}^{\mathcal{M}_0} = \mathsf{definedIn}^{\mathcal{M}}$ and $\mathsf{annotatedMethod}^{\mathcal{M}_0} = \mathsf{annotatedMethod}^{\mathcal{M}}$.

Apply Rule (4.1) repeatedly to compute the transitive closure of $\mathsf{subsumes}^{\mathcal{M}_0}$, which gives $\mathsf{subsumes}^{\mathcal{M}}$. Then, repeatedly apply Rule (4.2) to get $\mathsf{annotatedCorI}^{\mathcal{M}}$. Next, repeatedly apply Rules (4.3) and (4.5), followed by Rule (4.6). In the final step, compute

$\mathsf{effectiveAnnotation}^{\mathcal{M}}$ by repeatedly applying Rule (4.4) and $\mathsf{expectedAnnotation}^{\mathcal{M}}$ by repeatedly applying Rule (4.7).

We assert that α indeed computes the least fixed point. This is because it exploits a topological ordering of the inference rules. The algorithm α runs in worst-case time quadratic in the input size, which is the number of classes, interfaces, methods and roles.

Consistency and correctness The next definition states the consistency property that we enforce at compile-time.

Definition 1 *Our system is* consistent *if*

$$\mathsf{expectedAnnotation}^{\mathcal{M}} \subseteq \mathsf{effectiveAnnotation}^{\mathcal{M}}.$$

As we discuss in Section 4.1.1, annotations on interfaces impose constraints on the effective annotations on methods defined in classes. If the effective set of annotations on a method in an interface i is E_i, and the effective set of annotations on the corresponding method in a class c that implements i is E_c, the consistency property requires that E_i is a lower bound for E_c, that is, $E_i \subseteq E_c$.

We assert via the following proposition that $\mathcal{M}$ is correct. Correctness is characterized relative to $\mathcal{M}_0$; $\mathcal{M}_0$ expresses exactly what is specified in the Java code. A method in a class has in its effective annotation every role that it should (completeness) and no roles that it should not (soundness). We write our assertion somewhat informally for clarity.

Proposition 2 $(m, c, r) \in \mathsf{effectiveAnnotation}^{\mathcal{M}}$ *if and only if exactly one of the following cases is true. Furthermore, if (m, c, r_1) and (m, c, r_2) are two elements of* $\mathsf{effectiveAnnotation}^{\mathcal{M}}$, *then the same case from below is true for both elements.*

1. *If m is defined in c, then:*
   - *(a) m is not annotated with r in its definition in c, but is annotated with r' where r subsumes r', perhaps transitively.*
   - *(b) m has no annotations in its definition in c, and c is annotated with r', where r subsumes r', perhaps transitively.*
2. *Otherwise let $c_1, c_2, \ldots c_n$ be the inheritance hierarchy of c, and let c_i be the first class in the hierarchy (starting at c) that defines m. Then m is annotated with r in its definition in c_i via one the above cases.*

Proof ($\Leftarrow$): If exactly one of the three cases is true, then (m,c,r) appears in the set $\mathsf{effectiveAnnotation}^{\mathcal{M}}$:

Case 1(a): If method m is defined in class c, and has explicit annotations, then by Rule 4.3 (explicit annotation), along with Rules 4.1 and 4.4 (on subsumption), for each annotation r, we place (m,c,r) into $\mathsf{effectiveAnnotation}^{\mathcal{M}}$.

Case 1(b): If method m is defined in class c and has no explicit annotations, then by Rules 3.1 (class annotations) and 4.1 (subsumption), for each annotation r, we add (m,c,r) to $\mathsf{effectiveAnnotation}^{\mathcal{M}}$.

Case 2: If m is not defined in c, then by Rule 4.6 we search superclasses until we find the class c_i defining m. If m in c_i has explicit annotations, then by Rules 4.1, 4.3, and 4.4, for each annotation r, we place (m,c,r) into $\mathsf{effectiveAnnotation}^{\mathcal{M}}$. If m in c_i has no explicit annotations, then by Rules 4.1, 4.2, and 3.1, for each r, we add (m,c,r) to $\mathsf{effectiveAnnotation}^{\mathcal{M}}$.

Exclusivity, the fact that no two tuples (m,c,r_1) and (m,c,r_2) can be added to $\mathsf{effectiveAnnotation}^{\mathcal{M}}$ by different cases, holds since a method cannot be both defined in a class and not defined in that class, and since a method cannot have both zero and non-zero explicit annotations. Thus, no two annotations on m can come from different sources, so if (m,c,r_1) and (m,c,r_2) are elements of $\mathsf{effectiveAnnotation}^{\mathcal{M}}$, then both were added by the same case.

($\Rightarrow$) If (m,c,r) appears in $\mathsf{effectiveAnnotation}^{\mathcal{M}}$, exactly one of the three cases of the proposition is true:

The only valid combinations of rules, based on the predicates contained therein, are {1,3,4} (*Case 1(a)*), {1,2,5} (*Case 1(b)*) and [{6,1,3,4},{6,1,2,5}] (*Case 2*). The initial set $\mathsf{effectiveAnnotation}^{\mathcal{M}_0}$ is empty. Given case exclusivity and the fact that in our system, the only method for growing the size of $\mathsf{effectiveAnnotation}^{\mathcal{M}}$ is application of the set of rules, this implies that if (m,c,r) appears in $\mathsf{effectiveAnnotation}^{\mathcal{M}}$ then it must have been added by one of the three cases of the proposition, and therefore exactly one of them must be true. $\square$

Semantics of Invocation We next express formally the rules that govern who may invoke which methods, via the following rules. The predicate $\mathsf{invokes}(m_1,c_1,m_2,c_2)$ indicates that method m_1 in class c_1 invokes method m_2 in c_2. The predicate $\mathsf{member}(u, r)$ indicates that u is a member of the role r, and $\mathsf{canInvoke}(m, c, u)$ indicates that the user u is able to invoke method m in class c.

The predicate canInvoke is not restricted to the methods that a user is directly authorized to invoke; it includes methods invoked transitively. Thus, canInvoke includes all methods the user can effectively run. Consider a scenario where role r is not authorized to invoke `b()`, but may invoke `a()`, and `a()` invokes `b()`. Users who are members of r can invoke `a()`, but our semantics also reflect that members of r can, in effect, invoke `b()`.

$$\mathsf{invokes}(m_1, c_1, m_3, c_3) \leftarrow \mathsf{invokes}(m_1, c_1, m_2, c_2) \wedge \mathsf{invokes}(m_2, c_2, m_3, c_3) \tag{4.8}$$

$$\mathsf{canInvoke}(m, c, u) \leftarrow \mathsf{effectiveAnnotation}(m, c, r) \wedge \mathsf{member}(u, r) \tag{4.9}$$

$$\mathsf{canInvoke}(m_2, c_2, u) \leftarrow \mathsf{canInvoke}(m_1, c_1, u) \wedge \mathsf{invokes}(m_1, c_1, m_2, c_2) \tag{4.10}$$

Rule (4.8) indicates that invokes is transitive: if m_1 invokes m_2 and m_2 invokes m_3, then m_1 invokes m_3. Rule (4.9) indicates that a user is allowed to invoke a method to which he is authorized via his role memberships. When we say "user," we mean the customary RBAC meaning of a user [22]; in our context, an RMI client maps to a user.

Finally, Rule (4.10) indicates that if a user may invoke a method m_1, and m_1 invokes m_2, then the user can invoke m_2 as well. For a semantics, we specify a model as we do for the semantics of annotation.

As the rules above express, a client may of course invoke a method to which it is authorized. In addition, a client may invoke other methods, but only via methods to which he is authorized. That is, if a client is authorized to invoke m_1, and in its execution, m_1 invokes m_2, then this is allowed even if the client is not authorized to m_2 via annotations. The reason we allow this is that we assume that m_1's invocation of m_2 is controlled — the client cannot directly control the parameters and the manner in which m_2 is invoked.

4.2 Implementation

In this section, we discuss the implementation aspects of our approach. We separate our discussions into a compile-time (Section 4.2.1) and a run-time component (Section 4.2.2).

We modify RMI-enabled classes at compile-time and generate class files (compiled output) for interfaces we derive. Our routine for building interfaces is based on our work in Chapter 3. We use a compile-time preparation step that takes place between compiling the source files and running the RMI Compiler (`rmic`). In this step, we employ an *interface builder* (see Section 4.2.1), which examines classes, derives an interface, and modifies that interface according to the RBAC policy the developer has specified.

4.2.1 Compile-Time Component

Our program, the Proxy Object RBAC Compiler (`porc`) automatically performs all the compile-time work. It first builds the role hierarchy, based on the role annotations (see Section 4.1 under "Roles and role hierarchy"). It then expands annotations according to the inference rules (e.g., associates class annotations with contained methods). Next, it examines the classes that implement the interfaces. It reports an error if an implementation lacks an annotation that an interface implies it should have. Finally, it infers the annotations on method implementations from the annotations on the classes in which they are defined.

Once the information about roles and annotations is computed, `porc` searches the code base and finds RMI-accessible objects, examines each of these objects and derives its remote interface. This derived interface enforces the rules described by the annotations (by hiding inaccessible methods). `porc` outputs the bytecode version the generated interface to a new class file, and modifies the original object's class file to implement the derived interface. At run-time, the server simply enters a dynamically-created proxy object of the correct type into the RMI server. No modifications to the Java compiler or the RMI compiler (`rmic`) are necessary.

Figure 4.2 depicts this process on a sample class `Order`, in a version of our example from Section 4.1. We show selected roles from Figure 4.1 in Figure 4.2. In the figure, `javac`, `porc` and `rmic` are the three stages of compilation. The files `Accounting.java`, `Accounting.class`, `ITEmployees.java`, and `ITEmployees.class` are the roles, before and after compilation. All other boxes are files that contain source code or an intermediate representation. In the figure, we show only new output files and modified input files; unchanged input files do not appear as output of the processing steps.

We next describe how our system produces its various output files. In Figure 4.2, we assume that a developer has written the `Order.java` file, and compiled it and the roles into a class file using `javac`. The `porc` examines all the roles in our system and defines the role hierarchy. Next, `porc` identifies `Order` as being a remotely-accessible object, and processes it further by deriving interfaces corresponding to each role. In the figure, we show processing of the `Order` class for the roles of `ITEmployees` and `Accounting` only.

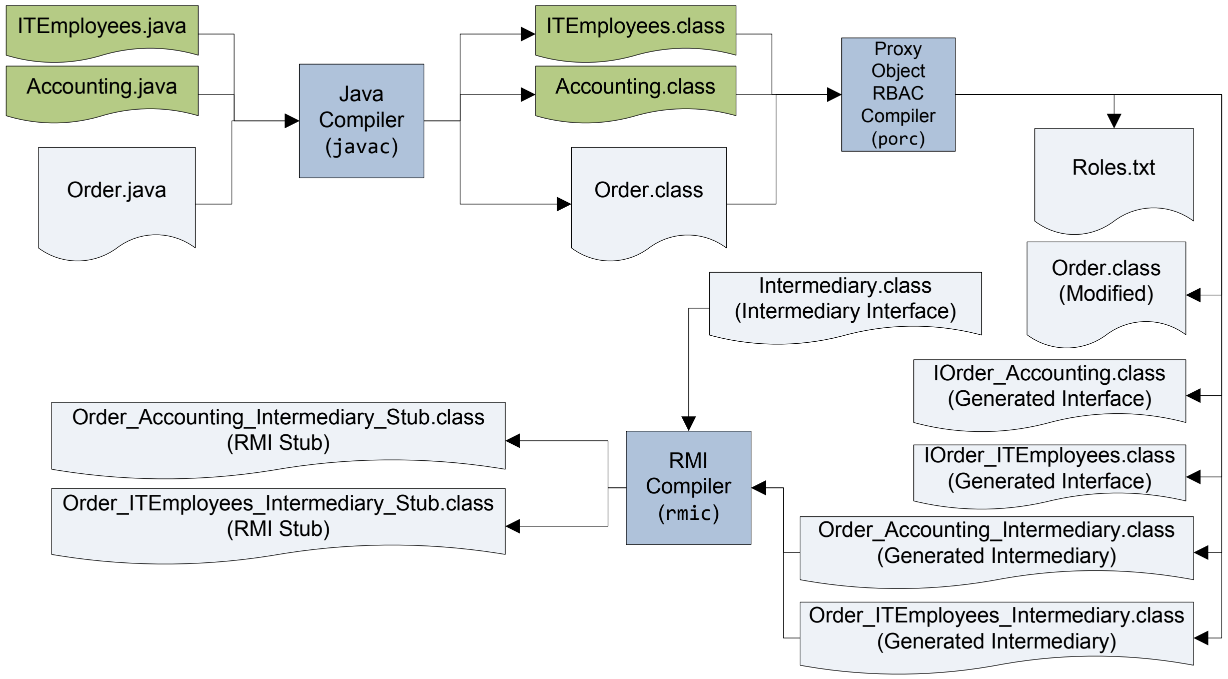

ITEmployees.java
Accounting.java
Order.java
Java Compiler (javac)
ITEmployees.class
Accounting.class
Order.class
Proxy Object RBAC Compiler (porc)
Roles.txt
Order.class (Modified)
Intermediary.class (Intermediary Interface)
IOrder_Accounting.class (Generated Interface)
IOrder_ITEmployees.class (Generated Interface)
Order_Accounting_Intermediary.class (Generated Intermediary)
Order_ITEmployees_Intermediary.class (Generated Intermediary)
RMI Compiler (rmic)
Order_Accounting_Intermediary_Stub.class (RMI Stub)
Order_ITEmployees_Intermediary_Stub.class (RMI Stub)

Figure 4.2: Proxy Object Compiler Processing `Order`

Each of the generated interfaces (e.g., `IOrder_Accounting`) contains only the methods authorized to its role. The `porc` modifies `Order.class` to implement each generated interface. It also outputs to `roles.txt` a summary of the role hierarchy from in the given application. In addition, if there are any RMI-accessible classes without any annotations, the `porc` produces a warning to help the developer ensure that he or she has applied the security policy to all classes.

Once our modifications are complete, we invoke `rmic` on the resultant class files using the our provided common intermediary interface `Intermediary.class`, and the RMI compiler produces the two intermediary stub classes appropriate to the IT Employees and Accounting roles: `Order_ITEmployees_Intermediary_Stub.class` and `Order_Accounting_Intermediary_Stub.class`.

4.2.2 Run-Time Component

Our mechanism uses the Java Authentication and Authorization Service. JAAS enables developers to specify their own methods for authentication, such as password-based authentication. Upon successful authentication, we issue credentials to the client that describe the client's role(s) in the system. These credentials will be used later when the

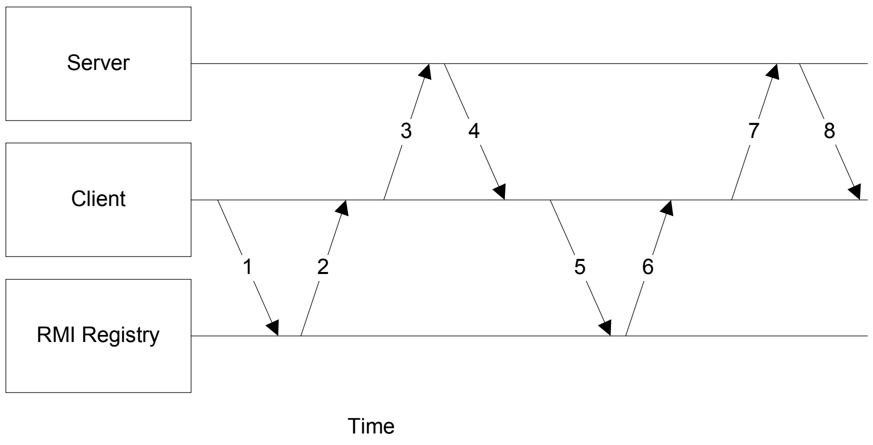


Figure 4.3: Communication Flow for a Client Accessing the First Proxy Object. Arrows correspond to messages and time proceeds from left to right.

client attempts to access a proxy. We note that we are not interested in the mapping of clients to roles, as there are many administrative models for doing so. The actual method of authentication is not relevant; we are interested only in the list of roles that the client may access.

Authorization relies on the credentials the server has issued to the client. To request a proxy object, the client presents the credentials to the server. Once the server verifies these credentials, it grants access to the proxy object.

To access the proxy, the client performs an RMI lookup of an intermediary object. Intermediary objects are accessible to everyone using RMI; they guard the proxy objects that correspond to roles, by only making proxies available upon receipt of credentials. Intermediaries are necessary because it is not possible to require authentication for access to RMI objects. RMI predates JAAS and has not been updated to enforce the JAAS capabilities [23].

```
public interface Order_Accounting_Intermediary extends java.rmi.Remote {
      public Order_Accounting login  (Credentials credentials)
                          throws LoginException, java.rmi.RemoteException;
}
```

Consider a simple example of a client that connects to a server. The server has an object that can be used to verify its identity [24]. The server's signed identity object is available via RMI. Figure 4.3 shows the sequence of events starting from when a client wishes to use the first object.

A directed edge in Figure 4.3 depicts an interaction between two elements of the system, and these edges are numbered from left to right. Before client-server communication begins, the server instantiates the authentication object as well as any proxy objects, and registers these objects so that they are available via RMI (not shown). When a client wishes to access a proxy object, the steps are:

1. The client requests the server's authentication object from the RMI registry.
2. The RMI registry returns to the client a reference to the server's authentication object.
3. The client may verify the authentication object using the server's public key. If the verification of the server's identity succeeds, the client sends login information (using JAAS) to the server.
4. If the server accepts the client's attempt to log in, it returns to the client the set of credentials to which the client is entitled (according to the user-to-role mappings of the system).
5. When the client wants to access an object, it asks the RMI server for the location of the intermediary.
6. The RMI server returns that location to the client.
7. The client presents the server-issued credentials to the intermediary.
8. If the credentials are valid, the intermediary returns to the client a reference to the proxy object appropriate to the request.

A proxy object is an instance of the `Proxy` class. This class handles (in the sense of a Java Invocation Handler) all method invocations on the proxy object. If the method exists in the interface associated with a specific role, then the proxy passes the invocation on to the original object. The original object executes the method as requested. If a method is not permitted for that role, it does not appear in the interface, and hence is not available for invocation over RMI. Therefore, only the methods that are allowed for a given role may be invoked by users with that role.

4.3 Empirical Assessment

We have applied our technique to three sample programs that range from almost 9 000 to nearly 300 000 lines of code (LoC). Our approach is applicable to programs that

Program	Lines of Code	Roles	Relations	Annotations Inserted
JBoss-Messaging	294 388	5	6	144
jGnash2	69 555	2	1	149
HospitalRMS	8 965	6	5	20

Table 4.1: Case Study Results for Sample Programs

allow clients to use RMI to invoke methods on objects. Many software systems are suitable, possibly after some restructuring. Any Model-Value-Controller (MVC) system is a candidate, if the communication between the model and view/controller are made to use RMI. When the program is structured around persistent domain objects (objects representing program entities), the domain objects can be server-side objects and accessed remotely by the client. A more general scenario is remote administration. In remote administration, the management objects (objects exposing administrative functions) are published via RMI.

There is a performance penalty when developers convert a program from entirely local execution to involving RMI. However, the benchmarks in Chapter 5 indicate that our technique adds no significant additional cost over the base penalty for using Java RMI. If the intent is to convert the application to use RMI, then a performance penalty is expected. Our approach allows the application to operate via RMI where it may not have been feasible before, because we offer security.

Our three sample programs are: **JBoss-Messaging**, a Java Message Service (JMS) implementation that ships with the JBoss application server; **jGnash2**, a personal finance application; and **Hospital RMS**, a multi-role hospital record management system that was a course project in database systems. For each program, we developed a role hierarchy intended to represent a possible real-world security policy. Figure 4.4 shows the roles in the case studies and the subsumption relationships between these roles.

We applied these role hierarchies to their respective programs and examined the process of annotation to evaluate the applicability of our approach to real-world software. We summarize our results in Table 4.1.

JBoss-Messaging has many software components; we focus on the server management components. The server management objects are used for administrative functions in JMS, such as creation of topics and queues, setting the dead letter queue, and monitoring the system. The management interface is a relatively small subset of the program, but controlling access to it is vital to maintaining program security. Consider the following example of an annotation we apply in JBoss-Messaging.

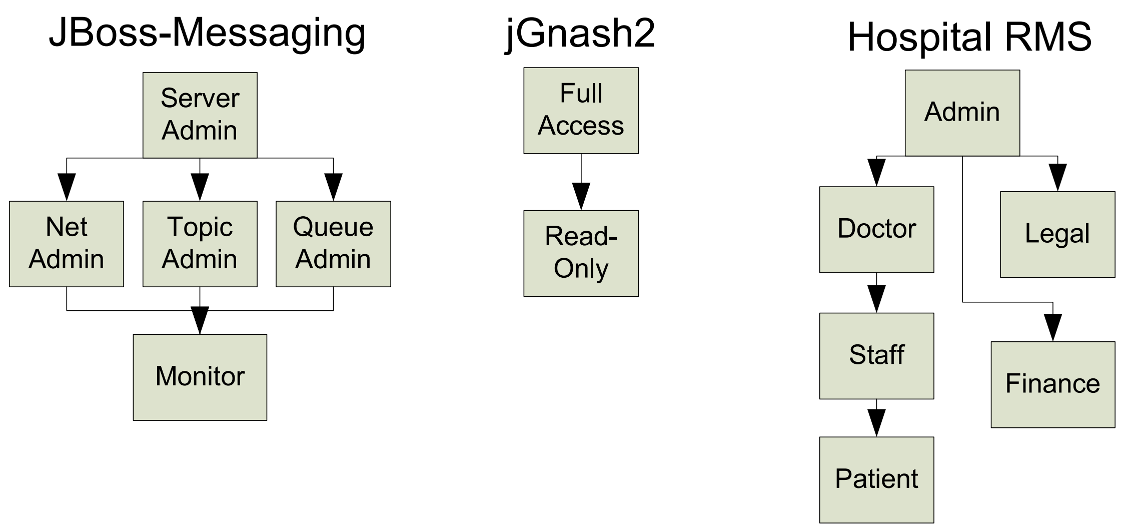


Figure 4.4: Role Hierarchies for Sample Applications. Boxes denote roles and arrows indicate subsumption relationships.

```
@QueueAdministrator
public void setDeadLetterAddress
     (final String deadLetterAddress)
     throws Exception { ... }
```

The Queue Administrator and the Server Administrator, by subsumption, now have rights to invoke this method when published via RMI. If the queue control object were published without protection, anyone in the system could set the dead letter address (the place where undeliverable messages end up). In that case, an attacker could then receive messages intended for other users.

In the case of jGnash2, we apply our policy to the domain objects (e.g., `Transaction`) and can therefore implement our policy with a relatively small number of annotations, when compared to the size of the code base. Controlling access to the domain objects prevents an unauthorized user from making changes to the data stored in the system. We may then publish the domain objects via RMI. There are only two roles in this system (read-only and full access) and only a few domain objects, so the number of annotations to implement the security policy is small.

The Hospital RMS application uses a strict model-view-controller (MVC) architecture; we apply our security policy by enforcing access control on the models. Accordingly, using role subsumption, we need only 20 annotations to enforce the basic security policy of the system. This is because each model (e.g., patient record model) is wholly accessible (all methods available) or inaccessible (no methods available) to various roles.

Overall, we can see that the number of lines of code in the program do not necessarily indicate the number of annotations required to implement the security policy, nor do more roles mean more annotations to insert. Instead, what matters most is the design pattern of the application (e.g., MVC) and the security policy to be implemented. The security policy can often be succinctly stated, so that few annotations are needed to implement the policy. Finally, a program typically has only a small number of objects that are intended to be accessible via RMI. Consequently, the total number of classes to annotate is small. As the proxy object compiler issues a warning if there are RMI objects without any annotation, it is easy to ensure all classes that should have a security policy have annotations.

Modifying the code to insert the annotations is easy and requires minimal time. We were able to annotate each of the sample applications in less than a day. As properly applying the security policy to all RMI-accessible objects ensures the enforcement of that policy, a developer need only locate all RMI-accessible objects, which is easy to do. He or she can then apply annotations to specify the desired security policy. Our system enables developers to succinctly and rapidly encode RBAC security policies without having to enforce them manually.

Chapter 5

Performance Analysis

We wanted to see if our system could have acceptable performance. Although there is always overhead in a security system, a system with excessive overhead is not very useful in practice. Some micro-benchmarks reveal the basic performance of our system.

Proxy Creation To create an proxy of an object with no defined methods, except those inherited from `Object`, takes 0.56 ms, on average. Thereafter each method to be processed adds a small penalty. The penalty is roughly constant, so we observe linear behaviour with the number of methods. Our sample data points are for sample objects with 0, 25, 50, 75, and 100 methods. Table 5.1 summarizes the data, including the mean and standard deviation. See Figures 5.1 for a graph of the data.

Interface Derivation In addition to the proxy object creation cost, the first time a proxy object of a particular class is created, there is an additional, one-time, server-side cost to derive the interface.

This step when using the RBAC proxy object system is all handled at compile-time; that is, all interfaces are generated by the Proxy RBAC compiler step. Accordingly, the cost at run-time for interface derivation is zero.

Methods	**0**	**25**	**50**	**75**	**100**
Mean (ms)	0.563	1.856	3.207	4.487	5.834
Standard Deviation	0.022	0.010	0.017	0.024	0.056

Table 5.1: Summary of Object Creation Performance

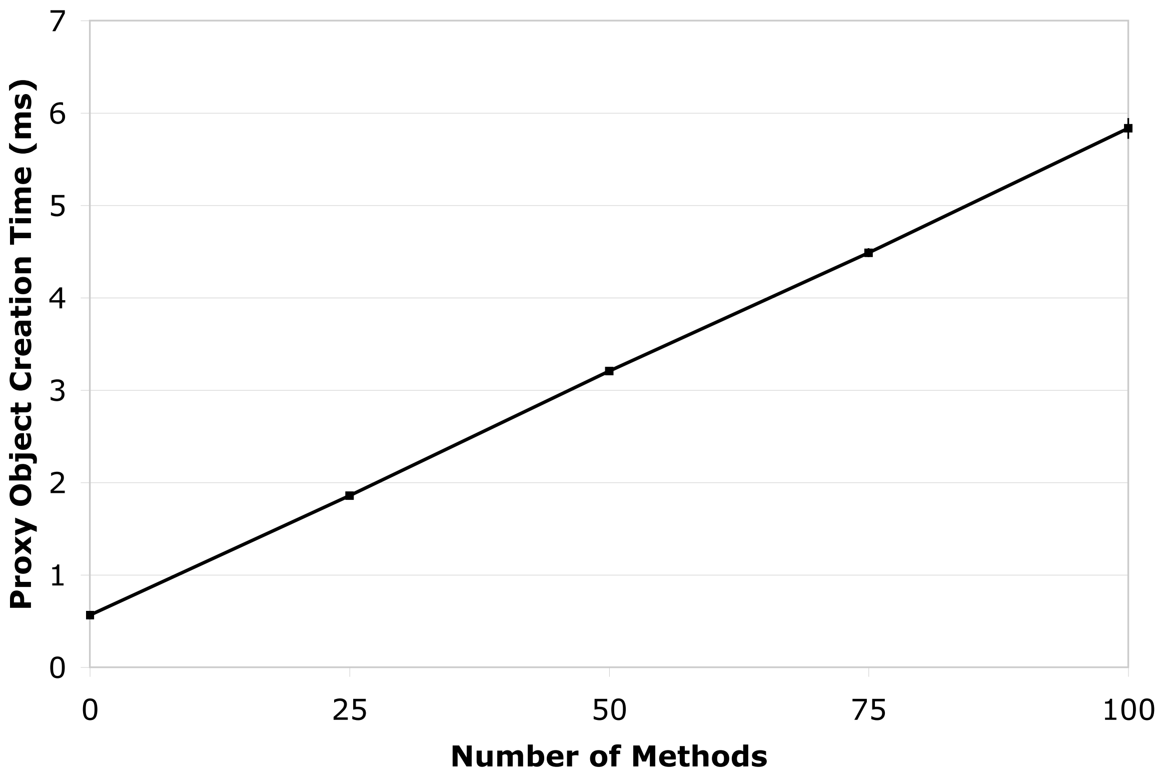


Figure 5.1: Proxy Creation: Linear Performance with the Number of Methods in the Object

Methods	0	25	50	75	100
Mean (ms)	21.51	30.43	34.67	38.85	42.15
Standard Deviation	0.560	0.640	0.753	0.730	1.07

Table 5.2: Summary of Interface Build Performance

In the safe and unsafe system, interfaces are generated both at compile-time and at run-time. After an interface is created and loaded, it is cached, so thereafter the cost of creating the interface is negligible. Furthermore, testing reveals that when the interface is created multiple times, the virtual machine optimizes or caches the derivation of the interface after the first time it is created. However, we conducted our analysis by restarting the Java virtual machine each time, to get consistent results. Table 5.2 summarizes the data, including the mean and standard deviation for each test set. See the right side of Figure 5.1 for a graph of the data.

Deriving the object's interface is linear in the number of methods in the object. Although the initial interface creation may be costly, we reiterate that it is a one-time cost; once derived, the interface is cached and is never re-created.

Testing Summary To provide some perspective, we conducted a comparison to how long it takes to do a Java RMI lookup of a simple example object (not a proxy). The server, RMI registry, and client ran on the same machine and used the same network interface. These tests reveal that over 1 000 tests, a lookup takes 89.2 ms on average (with a standard deviation of 8.52 ms). Creation of a proxy object, even one with 100 methods, takes an order of magnitude less time than the RMI lookup. Even deriving the interface is on the same order as the RMI lookup. Thus, our system's overhead is small in practice.

We conducted this performance analysis using a MacBook Pro (2.4 GHz Core 2 Duo and 4 GB RAM), using Java 1.6.0. We ran the creation test 1 000 000 times, and the interface building test 100 times. In both cases, we averaged over all executions to produce the final result.

Overall, the performance results are positive. Our tests reveal that locally invoking a proxy in place of the original takes almost no time. Over 1 000 000 tests, invoking the proxy took, on average, 0.002 ms longer than invoking the original, which is below the noise threshold for such a test. Therefore, the overhead of invoking a proxy is negligible.

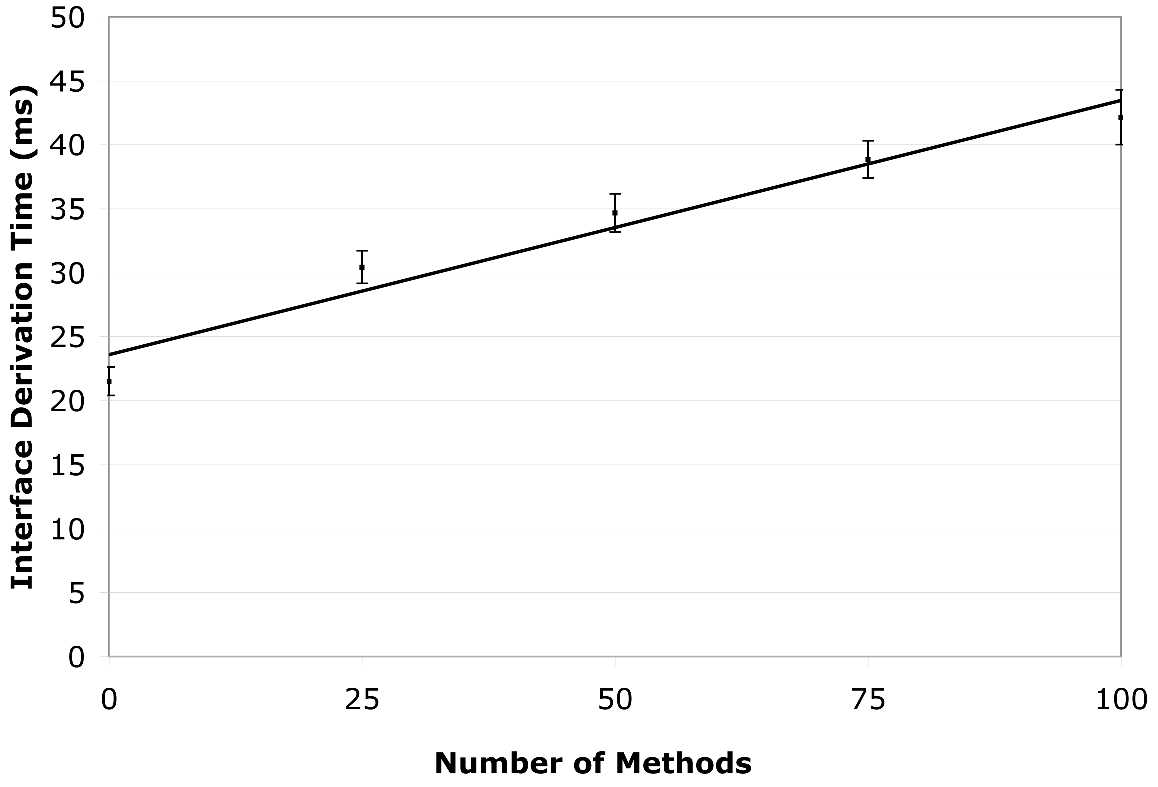


Figure 5.2: Interface Derivation: Linear Performance with the Number of Methods in the Object

Chapter 6

Future Enhancements

Both the basic system and the RBAC system present several areas of possible future expansion, alteration, or enhancement. Some of the enhancements listed below would be beneficial in a corporate situation, and others are suitable as research problems to be investigated in the future.

6.1 Enhancements to Safe and Unsafe

We discuss two enhancements to the Safe and Unsafe system: automatic inference annotation, and local client security.

Automatic Inference Annotation The Safe/Unsafe solution is effectively a white or blacklist. Given a large legacy code base, it will be difficult to identify all methods that should be allowed/denied, so we might deny some invocations that should be allowed, or vice-versa.

To determine which methods to mark as safe or unsafe, we could decide based upon whether or not methods write to the heap; this is a purity analysis [25]. If a method only reads from the heap, then it cannot change the system state, and it is presumably safe to use: methods without side effects would be automatically marked as safe to invoke, and will therefore appear on the proxy objects. The programmer may override these defaults by manually granting and revoking permissions at compile-time using the `@Safe` and `@UnSafe` annotations.

Local Client Security Java reflection could give untrusted clients unauthorized access, if they are running on the same virtual machine as the server (i.e., locally).

Because a proxy object references the original to pass along method invocations, an attacker could use reflection to access the private data of the proxy and retrieve the original object. Per Richmond and Noble [18], remote reflection does not exist, so this vulnerability is limited only to local clients. We consider it reasonable to require that any client running locally be trusted (like trusted client *T* in Figure 3.3).

6.2 RBAC Enhancements

We suggest two enhancements to the Role-Based Access Control version of our system: audit trails and improved role administration.

Audit Trails To complement the concept of role-based security, we may wish to introduce meaningful audit trails to our system, to track invocations. If implemented, we could output a log indicating the role that invoked a certain method at a particular time. Such an audit log could be used to catch instances of a System Administrator user performing an `Order.create()` and an `Order.approve()` on the same `Order`, in violation of separation of privilege principles. Such information could also be used for other purposes, such as gathering usage statistics.

Role Administration Role Administration is a related area that has received much attention in conjunction with role-based access control systems. Our administration procedure, described in the implementation, is configured only at compile time. Although configuration (addition and removal of annotations) is simple enough that a security administrator who is not a programmer could do it, code manipulation is still required. In the future, we could improve role administration to be outside of the code base of the system being protected. Specifically, we would support roles as data for the system (i.e., in a configuration file) instead of being tied directly into the code. This would also present opportunities to have better management of the role hierarchy in general; managing roles as a part of the code and without any visualization may suffice for the small examples, but may not work for a real-world system with a complex hierarchy.

Chapter 7

Related Work

Our solution is related to, but significantly different from, numerous areas in Java security. We discuss bytecode editing, stack inspection, interface derivation, proxy object generation, and, of course, role-based access controls.

While type safety obviates many security concerns, access control remains a key issue. Pandey and Hashii [5] investigate bytecode editing to enforce access controls, but do not discuss RMI. Wallach et al [10] enforce access controls using Abadi, Burrows, Lampson, and Plotkin (ABLP) Logic, where authorization is granted or denied on the basis of a statement from a principal allowing or denying access. However, their approach does not work with RMI, and, as acknowledged by the authors, does not handle a dynamic system with classloading well. Although Li, Mitchell, and Tong [24] provide a technique for securing a Java RMI application, their work does not use roles.

Giuri began with a basic overview of Role-Based Access Control [3] and extended this work to a Web context [26]. This work is foundational in Java Role-Based Access Control, and we build upon it by involving RMI and dynamic object generation from proxy objects. Ahn and Hu [27] implement a system for generating code from language-independent security models; their approach enables separate consistency verification for the model. Like many code generation approaches, however, their system does not support round-tripping and therefore cannot smoothly support concurrent changes to code and model, unlike our system, which includes the security policy directly in the code, helping ensure that the code and the policy both remain up-to-date.

Stack inspection can provide effective access control, but in an RMI context, the client call stack is unavailable to the server; even if it were available, it would be untrustworthy. A stack inspection scheme would therefore have to consider all remote accesses untrusted, whereas proxies can differentiate between trusted and untrusted

RMI calls. Furthermore, the time to perform a stack inspection increases linearly with the depth of the stack [10], while the proxy object overhead is constant. Stack inspection suffers from difficulties with results returned by untrusted code, inheritance, and side effects [1]. Proxy objects are more resistant to these difficulties, because they do not trust any results from untrusted code, are designed with inheritance in mind, and are intended as a tool to avoid harmful side effects. Proxy objects and stack inspection have different principles of trust. In proxies, a caller is trusted if it receives a reference to the original object. In stack inspection, the callee verifies its caller and all transitive callers.

Interface derivation is already in use in practice. For instance, Bryce and Razafimahefa [16] generate dynamic proxies to go between objects, and restrict access to methods. Guard objects, like proxies, are used as substitutes for the originals to control access. These bridges do not restrict access to fields; our solution allows only appropriate method invocations. Furthermore, we also permit role-based access controls rather than partitioning into trusted and untrusted clients.

Myers *et al* created JIF (Java Information Flow) [28] to restrict the leakage of information between objects within a Java program. We focus on method invocation by various roles, and do not assign ownership of data to particular principals. Finally, our system makes no changes to the Java language, meaning all semantics of the language remain consistent with the Java language specification.

Chapter 8

Conclusions

We have presented two techniques for method-level role-based access control for RMI-using Java programs.

Our first technique relies on computing whether a method is safe or unsafe based on program annotations. To clearly capture the semantics of our system, we have described them using First Order Logic. We implemented our system and obtained some basic performance metrics. Proxy objects can be employed statically, with Java Remote Method Invocation, or as a complement to the security model of the Enterprise Java Bean specification.

Our second technique computes the access rights to a given method based on program annotations. We have also described the semantics of our role-based system using First Order Logic. To investigate the behaviour of our system, we implemented a proxy object RBAC compiler. Using this compiler, we conducted a case study, which demonstrated the viability of our system in real software applications of various sizes.

Proxy objects have very little overhead in practice. We showed that creation of a proxy object takes an order of magnitude less time than the RMI lookup. Deriving the interface—a one-time cost—is on the same order as the RMI lookup.

Bibliography

[1] Fournet, C. and Gordon, A., “Stack Inspection: Theory and Variants,” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 25, pp. 360–399, May 2003. 2, 13, 34, 57

[2] Sandhu, R., Coyne, E., Feinstein H., and Youman, C., “Role-Based Access Control Models,” *Computer*, vol. 29, pp. 38–47, Feb 1996. 2, 6, 32

[3] Giuri, L., “Role-Based Access Control in Java,” *Proceedings of the third ACM workshop on Role-based access control*, pp. 91–100, 1998. 2, 56

[4] Gosling, J., Joy, B., Steele, G., and Bracha, G., *Java Language Specification, 3rd Edition*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2005. 2

[5] Pandey, R. and Hashii, B., “Providing Fine-Grained Access Control for Java Programs,” *Proceedings of the 13th European Conference on Object-Oriented Programming*, vol. LNCS 1628, pp. 449–473, 1999. 2, 5, 6, 14, 15, 16, 56

[6] Sun Microsystems, Inc., “Java Remote Method Invocation - Distributed Computing for Java,” 2010.
`http://java.sun.com/javase/technologies/core/basic/rmi/whitepaper/`
(Accessed on 2010-03-10). 3

[7] DeMichiel, L. and Keith, M., “JSR 220: Enterprise JavaBeans, Version 3.0: EJB Core Contracts and Requirements,” May 2006.
Available: `http://java.sun.com/products/ejb/docs.html`
(Accessed on 2010-03-10). 3

[8] Sun Microsystems Inc., “Controlling access to members of a class,” 1995-2008.
`http://java.sun.com/docs/books/tutorial/java/javaOO/accesscontrol.html`
(Accessed on 2008-11-14). 4

[9] Karjoth, G., “An Operational Semantics of Java 2 Access Control,” *Proceedings of the 13th IEEE workshop on Computer Security Foundations*, pp. 224–232, 2000. 4, 10

[10] Wallach, D., Appel, A., and Felten, E., "SAFKASI: A Security Mechanism for Language-based Systems," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 9, pp. 341–378, 2000. 4, 8, 9, 10, 11, 13, 56, 57

[11] Manocha, H., "Protection: Bell-Lapadula Model," 1999.
`http://courses.cs.vt.edu/~cs5204/fall99/protection/harsh/`
(Accessed on 2008-11-25). 5

[12] Greenfieldboyce, D. and Foster, J., "Type Qualifier Inference for Java," *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications*, pp. 321–336, 2007. 7, 8

[13] Bartolleti, M., Degano, P., and Ferrari, G., "Static Analysis for Stack Inspection," *Electronic Notes in Theoretical Computer Science*, vol. 54, pp. 69–80, August 2001. 9, 11, 12, 13

[14] Hardy, N., "The Confused Deputy: (or why capabilities might have been invented)," *ACM SIGOPS Operating Systems Review*, vol. 22, pp. 36–38, 1988. 10

[15] Liang, S. and Bracha, G., "Dynamic Class Loading in the Java Virtual Machine," *ACM SIGPLAN Notices*, vol. 33, pp. 36–44, October 1998. 14

[16] Bryce, C. and Razafimahefa, C., "An Approach to Safe Object Sharing," *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 367–381, 2000. 15, 17, 57

[17] Rajshekhar, A.P., "Hibernate: Understanding Associations," 12 2005.
`http://www.devarticles.com/c/a/Java/Hibernate-Understanding-Associations/1/`
(Accessed on 2009-05-14). 21

[18] Richmond, M. and Noble, J., "Reflections on Remote Reflection," *Proceedings of the 24th Australasian Conference on Computer Science*, vol. 11, pp. 163–170, 2001. 23, 34, 55

[19] Hugh, M. and Ryan, M., *Logic in Computer Science*. Cambridge, UK: Cambridge University Press, 2nd ed., 2004. 26

[20] McManus, E., "Build your own interface - dynamic code generation," 10 2006.
`http://weblogs.java.net/blog/emcmanus/archive/2006/10/build_your_own.html`
(Accessed on 2009-05-22). 28

[21] A. V. Gelder, K. A. Ross, and J. S. Schlipf, "The well-founded semantics for general logic programs," *Journal of the ACM*, vol. 38, pp. 620–650, July 1991. 39

[22] Ferraiolo, D.F., Kuhn, D.R., and Chandramouli, R., *Role-Based Access Control*. Norwood, MA, USA: Artech House, 2007. 42

[23] Kumar, P., *J2EE Security for Servlets, EJBs and Web Services: Applying Theory and Standards to Practice*. Upper Saddle River, NJ, USA: Prentice Hall, 2003. 45

[24] Li, N., Mitchell, J. C., and Tong, D., "Securing Java RMI-Based Distributed Applications," *Proceedings of the 20th Annual Computer Security Applications Conference*, pp. 262–271, 2004. 45, 56

[25] Salcianu, A. and Rinard, M., "Purity and Side-Effect Analysis for Java Programs," *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pp. 75–82, 2007. 54

[26] Giuri, L., "Role-based access control on the Web using Java," *Proceedings of the fourth ACM workshop on Role-based access control*, pp. 11–18, 1999. 56

[27] Ahn, G-J and Hu, H., "Towards realizing a formal RBAC model in real systems," *Proceedings of the 12th ACM Symposium on Access Control Models and Technologies (SACMAT)*, pp. 215–224, 2007. 56

[28] Myers, A. C., Nystrom, N., Zheng, L., and Zdancewic, S., "Jif: Java information flow," July 2001. Software release. `http://www.cs.cornell.edu/jif`. 57