Method-Specific Java Access Control via RMI Proxy Objects using Annotations (Short Paper)

Jeff Zarnett, Patrick Lam, and Mahesh Tripunitara

University of Waterloo
Waterloo, Ontario, Canada

Abstract. We propose a novel approach for granting partial access on arbitrary objects at the granularity of methods to remote clients. The applications that we target use Remote Method Invocation (RMI). We automatically build custom proxy objects, and give them to untrusted clients in place of the originals. Proxy objects expose a subset of methods to prevent potentially dangerous calls from clients. We present semantics of our system, an implementation, and its evaluation. The creation of a proxy object takes an order of magnitude less time than the corresponding RMI lookup.

1 Introduction

Access control is a key security feature that protects sensitive information. Strongly-typed languages such as Java help prevent arbitrary accesses to memory and contribute to the development of secure systems. Java's built-in security features are fairly coarse-grained. Existing approaches, such as stack inspection [1, 2], provide the ability to grant access to only some of an object's methods. However, they must consider all remote accesses to be untrusted as the client's call stack is unavailable to the server.

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 expose only a permitted subset of methods, proxies are safe by construction and may be passed to untrusted clients. Our system works with Remote Method Invocation (RMI).

We have implemented our system and analyzed its performance. Our system imposes minimal overhead (see Section 5); creation of a proxy object takes an order of magnitude less time than the corresponding RMI lookup. Our contributions include an approach to automatically-generated RMI Proxy Objects for Security, an algorithm for deriving proxy interfaces, and an experimental evaluation of the feasibility and performance of our system.

2 Motivating Example

Our technique handles cases where a heterogeneous software system needs to share objects with subsystems that are not fully trusted. We are able to ex-

pose only parts of an object's functionality to a client. This is useful when it is undesirable or dangerous to grant the client unrestricted access.

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 creates a customizable application, ShipItems, for online commerce. The clients (store owners) purchase an account and 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. Figure 1 shows relations between these entities for three stores, "Scuba 3000," "Ed's Electronics," and "Shoe Palace."

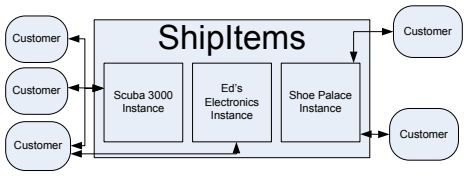

Fig. 1. *ShipItems* with Multiple Customized Instances

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 and provide rule templates for those cases, or give out full access to the Java objects. 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. This is costly and error-prone. Alternately, the owner can access the Java objects directly.

Suppose ShipItems validates that the Postal/ZIP Code field is not empty. The store owner will 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 that represents Canada. A modification to the Canada object affects every user. Likewise, by navigating the object graph, the store owner could access other stores' confidential data.

We could write a restrictive interface and provide this 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.

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, while limiting her access to the Java objects.

3 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 methods. We require a pre-processing step between the Java Compiler and the RMI Compiler (see section 4).

There are three levels at which a developer may specify a policy: global, class and method. The policy that takes effect is the one that is most specific to the method. If a method is annotated with a policy, then that policy takes effect. Otherwise, if the class in which the method is defined is annotated with a policy, then the method is associated with that policy. Otherwise the global policy takes effect for the method. The developer must specify a global policy. The global policy can be seen as the default for the system.

We support two kinds of policies: permit and deny. A developer specifies that his policy is permit by assocating an annotation called safe, and deny with an annotation called unsafe. If we infer that a method has an unsafe annotation, then it is unavailable to untrusted clients.

An object can have one or more proxies. Every proxy is associated with an original object that resides at the server. Proxies forward method execution to the original.

The developer annotates a method or class by adding `@Safe` or `@UnSafe` above it. These annotations impose no requirements upon the methods or classes they accompany; they do not affect 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 invoke methods that are annotated unsafe as well, but only via other methods. Clients then do not have direct control over the arguments with which these unsafe methods are invoked. We assume that it is safe to invoke these methods from other methods that are safe.

Consider the following example. A system has a global default permit policy (configured at compile-time); all methods may be invoked unless marked as unsafe. A class O is marked as default deny, and its `a()` 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 O because O has a policy of default deny. The class-level policy of O is overridden by the annotation of `a()`, which is the only method in O that is visible to untrusted clients.

```
// Global Policy: Default Permit
@UnSafe // Class Policy: Default Deny
public class O {
    @Safe // Method is safe
    public int a() { ... } // Permitted method
    // Unannotated method receives the annotation of its class
    public void b(String s) { ... } // Denied method
}
```

This system has trusted clients (T) and untrusted clients (U). Figure 2 shows the placement of the proxy object P in the system.

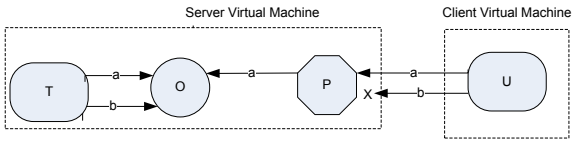

Fig. 2. Proxy Object P Guards O from Untrusted Client U

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 proxy object because it does not exist. Conversely, because trusted client T can access O directly, it is free to invoke b. However, even if b is unsafe in general, it may be 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.

We can create a proxy for any object. 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 (see Section 4) creates an interface I based on C's derived interface. Furthermore, a method appears in I only if it is allowed by the policy.

Our solution is impervious to attacks that use Java reflection. Consider a proxy object that is 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 [3]. RMI hides all fields of the implementation class from remote clients; fields do not appear on the client-side stub. We require that arbitrary access to memory on the server side is not permitted. We do not make assumptions about the client side virtual machine, as it is untrusted.

3.1 Semantics of Annotation

To express the semantics of our approach precisely, we use First Order Logic [4]. To express that a method m has annotation a in its definition in class c, we adopt the predicate $\mathsf{annotatedMethod}\,(m, c, a)$. Annotations on methods can be explicit or inferred. The counterpart to $\mathsf{annotatedMethod}$ 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 inherits c_1, and $\mathsf{definedIn}\,(c, m)$ to express that method m is defined in the class c. We use $\neq$ in the customary manner. We use $\neg$ for negation, $\vee$ for disjunction, $\wedge$ for conjunction, and $\longleftarrow$ for implication. Our inference rules are shown in Figure 3.

For a semantics, we specify a model, $\mathcal{M}$ and an environment or look-up table, l [4]. 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

$$\mathsf{definedIn}\,(c, m) \longleftarrow \mathsf{annotatedMethod}\,(m, c, a) \tag{1}$$

$$\neg\mathsf{annotatedMethod}\,(m, c, a') \longleftarrow \mathsf{annotatedMethod}\,(m, c, a) \wedge (a \neq a') \tag{2}$$

$$\neg\mathsf{annotatedClass}\,(c, a') \longleftarrow \mathsf{annotatedClass}\,(c, a) \wedge (a \neq a') \tag{3}$$

$$\mathsf{inherits}\,(c_1, c_3) \longleftarrow \mathsf{inherits}\,(c_1, c_2) \wedge \mathsf{inherits}\,(c_2, c_3) \tag{4}$$

$$\mathsf{definedIn}\,(c_2, m) \longleftarrow \mathsf{definedIn}\,(c_1, m) \wedge \mathsf{inherits}\,(c_2, c_1) \tag{5}$$

$$\mathsf{annotatedClass}\,(c, a) \longleftarrow (\mathbf{globalAnnotation} = a) \wedge (a \neq a') \wedge \neg\mathsf{annotatedClass}\,(c, a') \tag{6}$$

$$\mathsf{annotatedMethod}\,(m, c, a) \longleftarrow \mathsf{definedIn}\,(c, m) \wedge \mathsf{annotatedClass}\,(c, a) \wedge (a \neq a') \wedge \neg\mathsf{annotatedMethod}\,(m, c, a') \tag{7}$$

Fig. 3. Inference Rules for Determining Safe and Unsafe for Each Method

have the following mappings for our five predicates $\mathsf{annotatedMethod}\,(m, c, a)$, $\mathsf{annotatedClass}\,(c, a)$, $\mathsf{definedIn}\,(c, m)$, $\mathsf{inherits}\,(c_2, c_1)$, $a \neq a'$. The variables c, c_1 and c_2 map to elements of A_c, m to an element of A_m, and a and a' to elements of A_a.

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, when class c_2 extends (in Java) c_1, we instantiate $\mathsf{inherits}^{\mathcal{M}_0}$ to the pairs $\langle c_2, c_1\rangle$ Similarly, we instantiate $\mathsf{annotatedMethod}^{\mathcal{M}_0}$ to those $\langle m, c, a\rangle$ tuples such that the method m has the explicit annotation a in its definition in class c.

We define $\mathcal{M}$ to be the least fixed point from applying the rules from Figure 3. Our algorithm α for computing $\mathcal{M}$ from $\mathcal{M}_0$ is as follows. We first apply Rule 4 repeatedly until no more entries are added to $\mathsf{inherits}^{\mathcal{M}_0}$. We then repeatedly apply Rule 5 (which grows $\mathsf{definedIn}^{\mathcal{M}_0}$), and then Rule 6 (which grows $\mathsf{annotatedClass}^{\mathcal{M}_0}$), and finally Rule 7 (which grows $\mathsf{annotatedMethod}^{\mathcal{M}_0}$). The result is $\mathcal{M}$. Note that α indeed computes the least fixed point, because it respects the topological ordering of the inference rules. It runs in worst-case time that is quadratic in the number of classes, $|A_c|$.

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 Rules 2 and 3 in Figure 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 $\mathcal{M} \models_l (c, m) \in \mathsf{definedIn}^{\mathcal{M}} \longrightarrow (m, c, a) \in \mathsf{annotatedMethod}^{\mathcal{M}}$.*

We point out that to construct a proxy object for a class c, we do not need to use algorithm α. We can 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. Then, we can identify the annotation of each method in c in constant-time.

3.2 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. Our inference rules are shown in Figure 4. Our semantics are specified as in the previous section. In our model $\mathcal{M}$, $\mathbf{safe}^{\mathcal{M}} = \text{safe}$.

$$\mathsf{invokes}\,(m_1, m_3) \longleftarrow \mathsf{invokes}\,(m_1, m_2) \wedge \mathsf{invokes}\,(m_1, m_2) \tag{8}$$

$$\mathsf{canSafelyInvoke}\,(m, c) \longleftarrow \mathsf{annotatedMethod}\,(m, c, \mathbf{safe}) \tag{9}$$

$$\mathsf{canSafelyInvoke}\,(m_2, c) \longleftarrow \mathsf{canSafelyInvoke}\,(m_1, c) \wedge \mathsf{invokes}\,(m_1, m_2) \tag{10}$$

Fig. 4. Inference rules associated with Invocation

4 Implementation

Bytecode generation and modification lie at the heart of our implementation. We modify RMI-enabled classes and generate interface class files. We employ an interface builder, which examines classes, and creates a modified derived interface I. As the interface builder does not produce executable code, we can make the interfaces without resorting to a full-featured code generation library. Our interface building routine is based on the code by McManus [5].

I contains only safe methods, but we alter some of those methods. We leave unmodified all methods that return a simple type (e.g., `int`) or a `String`. For all other methods, we replace the return type with a proxy.

At run-time, we also employ a `ProxyObject` class, which we provide. This class is registered as the invocation handler for all proxy objects. If the method exists in the modified derived interface I, then the proxy passes the invocation on to O; O executes the method as requested. If proxy objects are found in the parameters, the invocation handler replaces those proxies with their corresponding originals before forwarding the execution. The `ProxyObject` class is also responsible for intercepting the return of a non-proxy object and performing the appropriate substitution. If a method is 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.

The interface builder runs during a compile-time preparation step that takes place between compiling the source files and running the RMI Compiler (`rmic`). No modifications to the Java compiler or `rmic` are necessary. Figure 5 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`. The application is now ready to run.

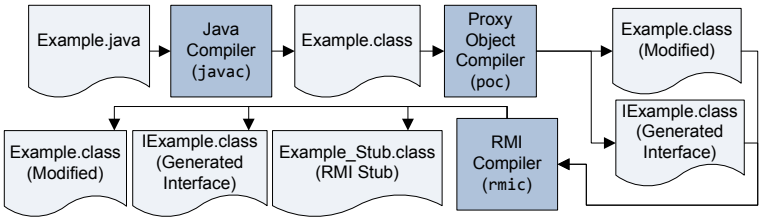

Fig. 5. Proxy Object Compiler Processing `Example.java`

5 Performance Analysis

To examine our system's performance, we use micro-benchmarks. Creating a proxy of an object with no defined methods, save those inherited from `Object`, takes 0.56 ms, on average. Thereafter, each method of the object adds a small penalty. The time to create a proxy object is linear with the number of methods.

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. After the 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 1 summarizes the data for both types of tests, including the mean and standard deviation. See also Figure 6 for a graph of the data.

Methods	0	25	50	75	100
Object Creation	0.563	1.856	3.207	4.487	5.834
Interface Build	21.51	30.43	34.67	38.85	42.15

Table 1. Mean Object Creation & Interface Build Times (in ms)

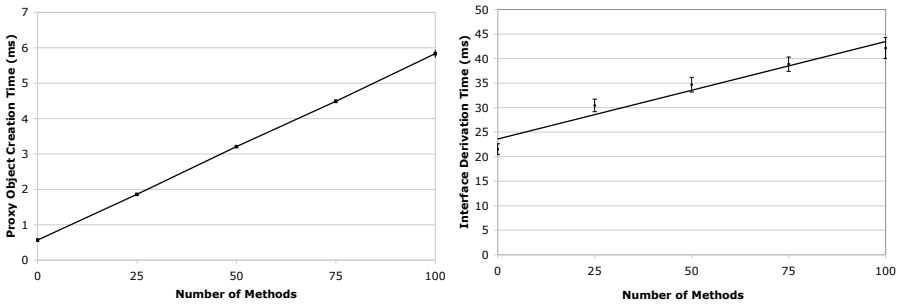

Fig. 6. Proxy Creation (L) and Interface Derivation (R): Linear Performance with the Number of Methods in the Object

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.

Finally, our tests reveal that locally invoking a proxy in place of the original has negligible overhead. Over 1 000 000 tests, invoking the proxy took, on average, 0.002 ms longer than invoking the original. This is below the noise threshold for our test. Therefore, the overhead of invoking a proxy is negligible.

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 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.

6 Related Work

While strong typing obviates many security concerns, access control remains a key issue. Pandey and Hashii [6] investigate bytecode editing to enforce access controls, but do not discuss RMI. Wallach et al [1] 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.

Stack inspection can provide effective access control, but the client call stack is unavailable to the server, and even if it were available, it would be untrustworthy. A stack inspection scheme would therefore have to consider all remote accesses untrusted, while 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 [1], while the proxy object overhead is constant. Stack inspection suffers from difficulties with results returned by untrusted code, inheritance, and side effects [2]. 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 [7] generate dynamic proxies to go between objects, and restrict access to methods. These bridges do not restrict access to fields; our solution allows only safe method invocations.

7 Conclusions

We presented a technique for method-level access control within the Java programming language. Our technique computes whether a method is safe or unsafe

based on program annotations. To capture the semantics of our system, we described them using First Order Logic. Our system is designed with Java RMI.

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.

References

1. Wallach, D., Appel, A., and Felten, E.: SAFKASI: A Security Mechanism for Language-based Systems. ACM Transactions on Software Engineering and Methodology (TOSEM) **9** (2000) 341–378
2. Fournet, C. and Gordon, A.: Stack Inspection: Theory and Variants. ACM Transactions on Programming Languages and Systems (TOPLAS) **25** (May 2003) 360–399
3. Richmond, M. and Noble, J.: Reflections on Remote Reflection. Proceedings of the 24th Australasian Conference on Computer Science **11** (2001) 163–170
4. Hugh, M. and Ryan, M.: Logic in Computer Science. 2nd edn. Cambridge University Press, Cambridge, UK (2004)
5. 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).
6. Pandey, R. and Hashii, B.: Providing Fine-Grained Access Control for Java Programs. Proceedings of the 13th European Conference on Object-Oriented Programming **LCNS 1628** (1999) 449–473
7. 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 (2000) 367–381