Reflections on MOPs, Components, and Java Security

Denis Caromel and Julien Vayssière

INRIA - CNRS - I3S
Université de Nice Sophia-Antipolis
{First.Last}@sophia.inria.fr

Abstract. This article investigates the security issues raised by the use of meta-programming systems with Java. For each possible type of MOP (compile-time, load-time, etc.), we study the permissions required for both the base and the meta-level protection domains, taking into account the flow of control between the different parts of the application.
We show that the choice of a particular MOP architecture has a strong impact on security issues. Assuming a component-based architecture with code from various origins having different levels of trust, we establish a set of rules for combining the permissions associated with each protection domain (integration, base-level, meta-level, etc.).

1 Introduction

In this article we investigate how Meta-Object Protocols (MOPs) [15] may be combined with Java's security architecture, especially in the context of component-based applications.

Java is the first mainstream programming language to take security into account from scratch, however it has also given birth to quite a large number of meta-programming extensions. Since its initial public release in 1995, Java has arguably become one of the most popular implementation platforms for researchers in the field of meta-programming with object-oriented languages. There now exist Java implementations of many different types of MOP.

Specifying a security policy for a Java application means determining the different protection domains involved and granting permissions to each of them, the goal being to run each piece of code using the smallest set of permissions necessary. As monolithic programs are now being gradually replaced with programs that are made up of a number of components, the above *principle of least privilege* becomes even more relevant. Furthermore, within the framework of meta-programming, one can use standard (base-level), meta-level, and MOP components, together with application-specific code, each of them having a specific origin and a different level of trust. The question of what happens to the specific security permissions of base-level and meta-level components when used together presents itself. This article tackles the more general problem of combining reflection and security in component-based Java applications.

The organization of the paper is as follows. In section 2 we successively give an overview of the security architecture of Java, introduce a classification of MOPs in four distinct categories, and identify a typical reflective component architecture. For that application, section 3 investigates the consequences of the MOP architecture on the original permission sets. Analyzing the different types of MOPs results in the development of a set of rules that govern the combination of the permission sets. We demonstrate that the different MOPs raise different security issues. This section also takes into account the `doPrivileged` construct. Section 4 presents related but orthogonal research: assuming that a MOP is secure, how can one use it to implement security policies? Finally, Section 5 summarizes and further generalizes the results.

2 Security, MOPs, and Components

In this section provide some background information on the security architecture of Java and meta-programming in Java, we then present a typical component-based application which will be used as a reference for the remainder of the paper.

2.1 The Security Architecture of Java

What follows is a short introduction to the security architecture of Java. A more detailed presentation can be found in [13].

The development of a security architecture for Java was motivated by the need to protect local resources from Java applets downloaded from untrusted sites. This explains why the current version of Java is heavily focused on *access control*, i.e. protecting access to critical local resources such as files, sockets, or the windowing system. The security architecture of Java is not concerned however with issues such as controlling the flow of information between the different pieces of code running inside a virtual machine [21], a problem of great importance, for example, in the JavaCard environment [4]. The Java language also has built-in security features, such as strong typing, enforcement of access modifiers, and no pointer arithmetic, which we will not address in this paper because they do not raise major problems when used with MOPs.

Central to the security architecture of Java is the notion of *protection domain.* A protection domain corresponds to either a URL from which classes can be downloaded, or a set of certificates that can be used for signing classes, or to any combination of both. This enables the mapping of protection domains to *principals* (persons or organizations on behalf of whom some code is distributed, and who can be held responsible if the code misbehaves). Each protection domain has an associated set of *permissions.* Each permission consists of a resource (for example, a file) that we want to protect access to, and an access mode (such as read, write, or execute).

Specifying a security policy for a Java application means determining the different protection domains involved and granting permissions to each of them.

This enables an application to run in accordance with the *principle of least privilege* [25], which states that a piece of code "should operate using the least set of privileges necessary to complete the job". This principle is of equal importance for both computer security and software engineering since it limits the damages that can result from a security attack and also protects a program from the consequences of a bug unwantingly introduced by a programmer.

At runtime, each of the classes loaded inside the virtual machine is associated with a specific protection domain. Whenever a thread performs a call that requires a specific permission, the security manager computes the *intersection* of the permission sets of all the protection domains on the execution stack of the thread. If this intersection contains the permission needed for accessing the resource, then access is granted, otherwise it is denied. This means that it is not sufficient for a given class to belong to a protection domain that has the right permissions in order to be granted access, all the other classes on the stack need to have this set of permissions as well.

In the context of a MOP-based application, with calls on the stack going successively back and forth between the base-level and the meta-level, the fact that it is the intersection of the set of permissions on the stack that determines whether or not access is granted is of great importance for our discussion.

However, the security architecture of Java provides a way to reduce this constraint. The `doPrivileged` construct enables us to limit the computation of the intersection to the protection domain from which the `doPrivileged` call is made and the protection domains subsequently called from it. The signature of the method is as follows:

```
public static Object doPrivileged (PrivilegedAction action)
```

The parameter of type `PrivilegedAction` encapsulates the code that is executed with the permissions of the new intersection of the protection domains. In other words, it means that the class that uses this `doPrivileged` construct takes responsibility for the classes that called it.

2.2 Meta-programming and Java

Since its initial public release in 1995, Java[1] has arguably become the implementation platform of choice for researchers in the field of meta-programming with object-oriented languages. There now exist Java implementations of all the major types of MOPs, from compile-time and load-time MOPs to run-time MOPs. We identify three main reasons why Java has become such an appealing platform for implementing meta-programming systems.

First of all, Java is an interpreted language and interpreters have proven to be an appropriate model for thinking about and implementing reflective features into programming languages [10]. Interpreters provide a natural separation between how an application is written in a given language and the description of

[1] by *Java*, we mean the whole *Java Platform*, which encompasses the *Java Virtual Machine* [17] (JVM), the *Java Language* [14] itself, and all the *Java Core APIs.*

how this language is executed. Interpreters allow us to alter the execution of a program simply by modifying the interpreter, without having to modify the program itself. This is the approach taken by sych runtime-MOPs as MetaXa[2][16], Guarana [22] and Iguana/J [23].

Seconfly, Java comes with a set of built-in reflective features, both structural and behavioral, which can be used as basic building blocks by designers of metaprograming systems. The best-known reflective feature of Java is the *Reflection API* [20]. This API was introduced with JDK 1.1 and provides structural reflection: Java programs can discover at runtime what types[3] exist inside a JVM and can inquire about all the constructors, methods, and variables for a given type. The Reflection API also provides a limited possibility to dynamically invoke those reflected members; however this does not come close to full-fledged *behavioral reflection* [10].

A third reason why Java is an interesting platform for implementing metaprogramming systems is that Java classes are loaded and linked on demand at runtime. This class-loading mechanism is itself reflected through objects of type `ClassLoader`, which provides the indispensable hook for implementing load-time MOPs. MOPs in this category usually modify the bytecode for a class at the moment it is loaded into the JVM.

For these three reasons, a significantly large number of MOPs have been written for Java. Depending on when meta-level code is executed, MOPs can be broadly sorted into four categories:

– *Compile-time MOPs* reflect language constructs available at compile-time by creating metaobjects to represent things such as classes, methods, loops, statements, etc. The meta-level code is executed at compile-time in order to perform a translation on the source code of a program.
– *Load-time MOPs* reflect on the bytecode and make use of a modified class loader in order to modify the bytecode at the moment it is loaded into the JVM. Their behavior if somehow similar to compile-time MOPs, except that they operate on bytecode rather than on source code.
– *VM-based runtime MOPs* rely on a modified version of the JVM in order to intercept things that only exist at runtime such as method invocations and read or write operations on fields. When such events occur, control is transferred to meta-level objects that are standard Java objects.
– *Proxy-based runtime MOPs* introduce hooks into the program, either at compile-time or load-time, in order to reify runtime events, mostly method invocation. They do not require any modification to the VM, which explains why some low-level events cannot be reified.

2.3 Component-Based Architecture and MOPs

As observed in [27], monolithic programs are now being gradually replaced with programs that are made up of a number of components, originating from various

[2] formerly known as MetaJava
[3] By *type* here we mean primitive types, arrays, classes, and interfaces.

sources and with various levels of trust, plus some application-specific code for gluing pieces together. This is especially true for Java applications, thanks to the JavaBeans component model [19] which provides a standardized way to describe and compose reusable Java components. However, it may happen that a third-party component lacks a specific non-functional property, such as support for transactions, persistence, or security. Meta-level protocols have proven to be a solution of choice for adding non-functional properties to third-party components [31], and so we will focus on component-based applications for the remainder of the article.

We will now illustrate the notion of component-based architecture with an example of a typical application (see Fig.1). This application is composed of five different components. The application-specific code, which is itself treated as a component named A, glues together two third-party components, $B1$ and $B2$. As $B2$ lacks some non-functional property, it is extended with a meta-level behavior described in component $MetaB2$ by using a metaobject protocol encapsulated inside component MOP. It is important to note that both the MOP and the meta-level classes are considered as components of the application, and as such are granted some permissions but not necessarily all permissions.

We would like to stress that this example, however typical, does not pretend to describe or model all possible applications that can be built using MOPs and meta-level classes. Rather, it should be seen as a first step in the direction of a general model for talking about the security properties of such applications. Building such a model is a subject for future research and is beyond the scope of this paper.

Nevertheless, we think this example is typical enough to describe most interesting cases. First of all, we have three base-level components with all direct and transitive chains of calls and callbacks between them. Moreover, one of the base-level components is extended with a meta-level component while the other is not. One of the components is implicitely using a reified component, and the example also includes some integration code, which is also packaged as a component.

Arrows on the figure represent possible method calls from one component to another component. We assume a one-to-one mapping between components and protection domains, since each component was written by a possibly different principal. Hence, performing method calls from a component to another component means crossing the boundaries of protection domains. For example, $MetaB2$ calls the methods of $B2$ for executing reified calls originally sent to $B2$, as represented by the arrow from $MetaB2$ to $B2$. It is worthwhile to mention here that the absence of an arrow from $B2$ to $MetaB2$ is due to the fact that classes inside $B2$ are not MOP-aware. We now list all the potential calls between protection domains for this typical application. This is essential for determining the protection domains that may appear together on the stack.

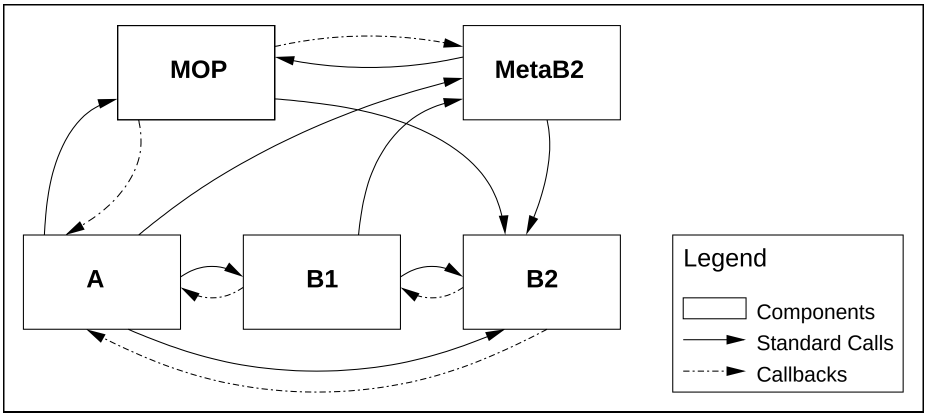

Fig. 1. Calls and callbacks between components

$A \to B1$, $A \to B2$, $A \to MOP$, $B1 \to B2$, $MetaB2 \to MOP$	A client component uses the service provided by another component.
$B1 \to A$, $B2 \to A$, $MOP \to A$, $B2 \to B1$, $MOP \to MetaB2$	Callbacks from a component to one of its client components.
$A \to MetaB2$, $B1 \to MetaB2$	Calls initially sent to $B2$ intercepted by the MOP for meta-level processing by $MetaB2$.
$MetaB2 \to B2$	$MetaB2$ executes a reified call originally sent to $B2$.
$MOP \to B2$	The MOP instantiates a reified instance of $B2$.

For completeness, we also list the calls that will not happen and explain why.

$B1 \nrightarrow MOP$, $MOP \nrightarrow B1$, $MetaB2 \nrightarrow B1$	$B1$ is not MOP-aware. It does not know that the calls it sends to $B2$ are reified.
$B2 \nrightarrow MOP$, $B2 \nrightarrow MetaB2$	$B2$ does not have to be MOP-aware either, which is one of the advantages of using MOPs for adapting third-party components.
$MetaB2 \nrightarrow A$	No callback from meta-level code to client code.

In the figure and the two tables above, all method calls could be considered equally important, because we assumed that any chain of calls might trigger an access check. However, it is not quite true. There are a couple of assumptions that we can make which help reduce the number of possible chains of calls that we must consider.

First, let us notice that calls such as $A \rightarrow MOP$ and $MOP \rightarrow B2$ are only concerned with creating reified instances of types in $B2$. We can expect object creation and initialization not to perform security-sensitive actions, as these usually happen later on in the life cycle of the object.

Rule 1 *The instantiation of reified objects should not perform actions that require access checks. If such actions must be performed, they should be delayed until after the construction of the object, for example until the first method call to the reified object is made (lazy initialization).*

The second assumption we would like to make is concerned with callbacks. Callbacks are method calls sent from a component to one of its client components through a specific interface in order to notify the client of the occurrence of an event. Event-based components originated in the realm of graphical user interface components and are becoming increasingly popular, thanks to the event model of JavaBeans. Although callbacks are standard method calls, their function is simply to deliver an event and then return, as opposed to calls from a client to a component in which a request for a service is made[4]. As it is not possible to guarantee that a method, called as part of the delivering of an event, does not perform an action that requires an access check, the specifications of the event model strongly recommend that one should perform such actions in a separate dedicated thread, and thus in a different security context. As callbacks are normal method calls, the security architecture of Java treats them just like any other method call. We believe that callbacks, when used according to the following rule, do not have any consequence security-wise.

Rule 2 *Callbacks should not perform actions that may trigger an access check. If such an action must be performed as the consequence of the callback, it should rather be done in a separate thread that was created and launched by the component that received the callback.*

Let us present the immediate benefits of applying this rule in a simple case with no MOP and no meta-level code. The only components we consider here are A, $B1$, and $B2$. They are connected together as in figure 1. If we call P, the function that maps components to the set of permissions of their protection domain, a call from A to $B1$ and then to $B2$ results in the following constraint on the permission sets of the protection domains:

$$P(A) \supseteq P(B1) \supseteq P(B2)$$

If callbacks need to perform actions that require access checks, callbacks from $B2$ to $B1$, and from $B1$ to A, respectively, add the two following equations:

$$P(B2) \supseteq P(B1)$$

[4] Although there is no way to actually enforce this property, it is considered in the Java community to be sound programming practice to do so.

and

$$P(B1) \supseteq P(A)$$

and hence, together with the first equation above, this leads to:

$$P(A) \equiv P(B1) \equiv P(B2)$$

If the rule above is enforced, we are back to the first constraint, which is far better as it allows us to maintain the specific permission sets, and hence abide by the least privilege principle. In the remainder of this article we will use this example architecture together with the two rules above in order to investigate the permission issues raised by the use of different kinds of MOPs.

3 Combining Permission Sets with Reflection

We now detail how the different categories of MOPs work and investigate their impact on security. We will present compile-time MOPs, load-time MOPs, VM-based runtime MOPs, and proxy-based runtime MOPs. In general, we will consider the most static solution for each MOP. For instance, a compile-time MOP could be used for implementing a proxy-based runtime MOP, however we will only consider the case of translations that do not introduce metaobjects at runtime.

3.1 Compile-Time MOPs

The typical MOP in this category is OpenJava[5] [7][28], which inherits most of the design philosophy of its direct ancestor Open C++ Version 2 [6]. OpenJava can be seen as an "advanced macro processor" that performs a source-to-source translation of a set of classes written in a possibly extended version of Java into a set of classes written in standard Java.

The translation to be applied to a base class is described in a metaclass associated with the base class. The metaclass is written in standard Java using a class library that extends the Java Reflection API with new classes for reflecting language constructs such as assignments, conditional expressions, field accesses, method calls, variables, type casts, etc. As a result, the object-oriented design of the library makes writing translations easier and more natural than with Open C++ where the sole abstraction available to the meta-level programmer is bare abstract syntax trees.

At first sight, the use of OpenJava does not break the security model of Java in any way: OpenJava outputs standard Java classes that compile and run within the standard Java environment and are subject to the same security restrictions as any Java class. Nevertheless, there is still a little security concern

[5] We should also mention *Reflective Java* [32], a compile-time MOP for intercepting method invocations.

with OpenJava. Even though it does not introduce any breach of security as such, it weakens the protection one might expect because it goes against the principle of least privilege. OpenJava allows a translation associated with a given base class to affect classes that may belong to different protection domains than the protection domain of the base class, and this may blur the fine-grained protection policy of the security architecture.

OpenJava defines the scope of the translation, expressed in the metaclass associated with a base class, according to the following rule: a translation can only affect the base class itself (*callee-side* translation) or the classes that perform method calls to the base class (*caller-side* translation)[6].

As a consequence, a caller-side translation may introduce, into all the client classes of a base class, code that may require additional permissions in order to run (figure 2). In the typical component-based application presented in figure 1, this means that the meta-level component $MetaB2$ will be incorporated into both the application code A and the MOP-unaware component $B1$ because the base-level component $B2$ is reified.

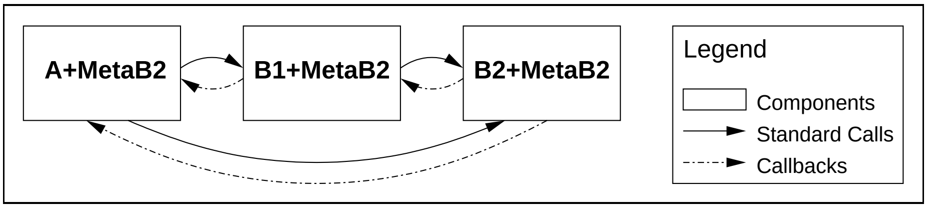

Fig. 2. Pure compile-time MOP

As OpenJava does not provide a way to tell what are the permissions required by the bits of code inserted by $MetaB2$ into the other components, we have to take the most conservative approach. In order to ensure that the resulting program does not raise security-related exceptions, the permission sets to be assigned to the resulting components are:

$$P(A + MetaB2) = P(A) \cup P(MetaB2)$$
$$P(B1 + MetaB2) = P(B1) \cup P(MetaB2)$$
$$P(B2 + MetaB2) = P(B2) \cup P(MetaB2)$$

Which means that the permission set of $MetaB2$ has to spread to the whole program. This goes against the principle of least privilege and defeats the purpose of a fine-grained security architecture.

There is also a problem with the requirement that as soon as a piece of code is modified by the meta-level (either caller- or callee-side) its set of permissions is

[6] Performing caller-side translation implies that all the client classes of the base class on which the translation is performed are known at the time of the translation.

expanded to include the permissions of the meta-level code. That is because there is no way to know what are the permissions required by the code inserted by the meta-level into the base-level code. Hence the most conservative approach has to be taken and all the permissions of the meta-level have to be added to the base-level classes. In the security literature, this problem is known as the *composite principal problem* and is of great importance to access control in distributed systems [1]. We will see that it also appears in run-time MOPs.

As a conclusion to this section, compile-time MOPs present inherent incompatibility issues with security but do not raise major security problems. What might be needed is a companion tool that would help the user understand how the translations expressed with the MOP affect the permission sets of the caller and callee protection domains, and also help the user dispatch the permissions needed by all the different components.

3.2 Load-Time MOPs

In load-time MOPs, meta-level computations take place either only at load-time or at both load-time and run-time. This leads to two different kinds of load-time MOPs: pure load-time MOPs and load-time MOPs that are used for implementing run-time MOPs.

Pure load-time MOPs [8] are close to compile-time MOPs, except that they operate on the bytecode representation of a class instead of on its source code. We call these kind of MOPs *pure* because metaobjects only exist at load-time, as opposed to load-time MOPs used for implementing run-time MOPs, like Kava [29], where meta-objects also exist at runtime.

As we can expect, the consequences in terms of permissions are similar to what we obtained with compile-time MOPs. However, there is one important difference. A transformation applied to a class in a compile-time MOP may also affect the client classes for the class, this is what we called caller-side translation in section 3.1. This is not possible for load-time MOPs since translations have to be performed at load-time on a class-by-class basis: client classes may have been loaded before the class the translation is performed on, and it is not allowed to modify classes after they are loaded inside the JVM [12].

Figure 3 illustrates the situation at runtime. The components MOP and $MetaB2$ have disappeared, since they only exist at load-time. For pure load-time MOPs we have

$$P(B2 + MetaB2) = P(B2) \cup P(MetaB2)$$

and hence

$$P(A) \supseteq P(B1) \supseteq P(B2 + MetaB2)$$

Again, the problem of the composite principal ($B2 + MetaB2$ in the above two statements) appears. It is actually a harder problem than one might first imagine because the meta-level transformation performed on $B2$ may add some code that requires permissions that are not needed for the execution of either $B2$

or $MetaB2$. For instance, $MetaB2$ might include, within a string, the instruction to be added into $B2$, for example `new File().write(...)`. It seems impossible to correctly determine the set of permissions required by $B2 + MetaB2$ without the cooperation of the meta-level code.

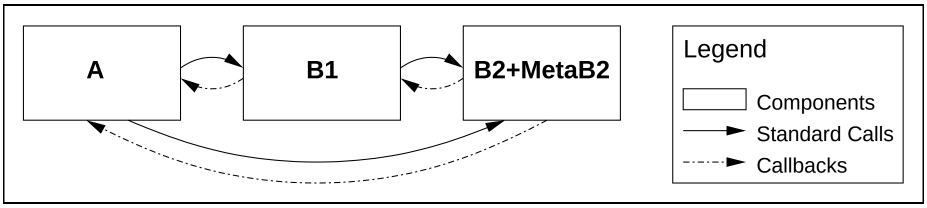

Fig. 3. Pure load-time MOP

The second category of load-time MOPs contains MOPs, such as Kava [29], which modify the bytecode of a class at load-time in order to introduce hooks that implement a shift from the base-level to the meta-level at some specific reification points in the code, usually on entering or leaving a method, or on reading from or writing to a field. This is why we say Kava is a run-time MOP implemented through load-time transformations of the code.

We can assume that the code inserted into the base-class does not trigger any security check, since it is simply responsible for sending calls to the meta-level. In our example, this means that the permission set required by $B2$ remains the same, and the scenario at runtime is close to the one presented in figure 1. However there is one assumption made by the authors of Kava that greatly simplifies the problem: both the MOP classes (MOP) and the meta-level classes ($MetaB2$) are assumed to be part of the "trusted computing base", which means that these classes are granted the same permissions as the classes in packages `java.*`, namely all permissions.

As a result, computing the intersection of the permission sets on the stack at any moment no longer depends on the permissions of components MOP and $MetaB2$, and so we are back to a scenario without any meta-level components, as is illustrated in figure 4. This means that base-level access checks remain the same as in a non MOP-enabled version of the application. Meta-level access checks will not be a problem, as long as they are performed inside `doPrivileged` blocks, so as to exclude base-level classes when computing the intersection of the permission sets.

To conclude this section, load-time MOPs either suffer almost from the same problem as compile-time MOPs or manage to solve the problem with permissions at the expense of granting meta-level classes with all permissions.

3.3 VM-Based Runtime MOPs

MetaXa [16] and Guaraná [22] are two examples of VM-based run-time MOPs. They both rely on a modified version of the JVM. Guaraná is implemented using

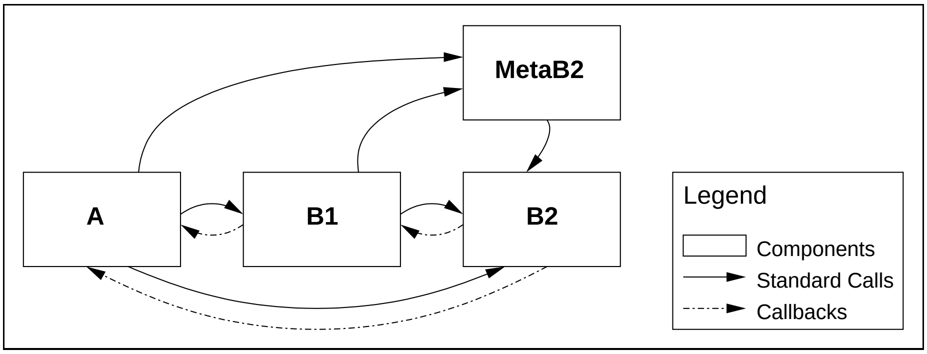

Fig. 4. Load-time MOP for implementing a run-time MOP

a modified version of the freely-available Kaffe virtual machine and MetaXa extends the virtual machine with a collection of native methods put together in a dynamic C library.

If our example application were run using such a MOP, there would be no modification to the set of permissions required by $B2$ because the interception mechanism is no longer visible as a standard call on meta-objects, but instead is buried deep inside the JVM.

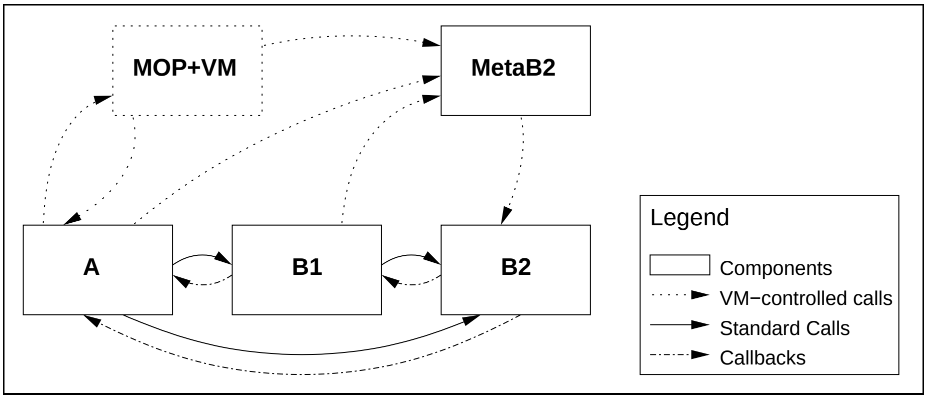

Fig. 5. VM-based Runtime MOPs

It is difficult to express relations between the different permission sets as we did for the previous kinds of MOPs since run-time MOPs have the power to alter the security architecture because they have access to the inner workings of the JVM. The very fact that these MOPs rely on a modified version of the JVM is not necessarily a security problem, instead it is actually a software diffusion problem.

Even if we trust the implementers of the MOPs, how can we control the enormous power given to the meta-level developer by letting him access and modify the internal data structures of the virtual machine, for example the state of the execution stack of a thread which is central to the decision-making algorithm of the security architecture of Java ?

If we assume that these potentially dangerous mechanisms are only used for intercepting the events that we want to see reified, then we are back to the security constraints of load-time MOPs used for implementing run-time MOPs. One hidden assumption about Java's security architecture is that all actions are performed through method calls. If a meta-level method is called as the consequence of, for example, reading a field or releasing a lock on an object, the behavior of the security architecture is not clear. To the best of our knowledge, all VM-based runtime MOPs were designed for versions of Java prior to Java 2, which explains why no attention at all has been paid to this problem.

3.4 Proxy-Based Runtime MOPs

Just like VM-based runtime-MOPs, *Proxy-based runtime MOPs* such as *RJava* [9] or ProActive[5], reify things that only exist at runtime, like object creation or method calls. The difference is that proxy-based runtime-MOPs are targeted at the standard Java runtime environment and do not rely on a specialized VM. As a consequence, there are things that VM-based MOPs can reify that proxy-based MOPs cannot, such as field access or operations on object locks.

The interception mechanism usually follows the *Proxy* design pattern [11]: a surrogate object with the same interface as the reified object acts as the reified object for its clients, intercepts method calls, and sends reified method calls to the meta-level.

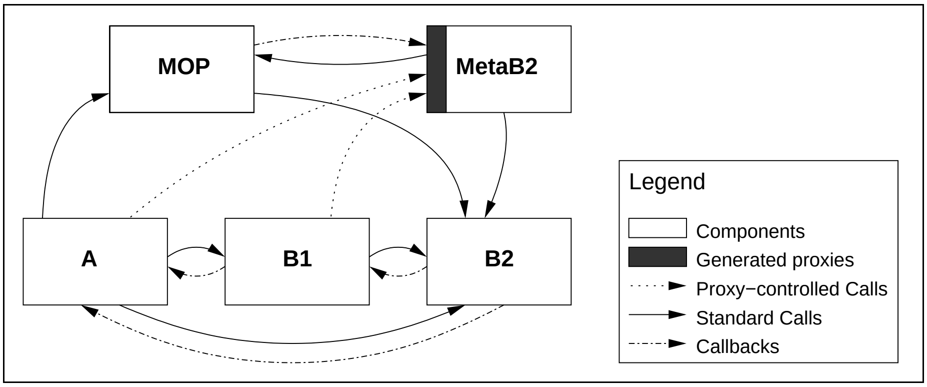

Fig. 6. Proxy-based Runtime MOPs

One can further differentiate *polymorphic* run-time MOPs when the reference returned by the MOP on the instantiation of a reified object is compatible with the original class of the reified object. For instance, a standard class C can be instantiated with a MOP together with a meta-object of type M, and assigned to a variable of type A. The Java pseudo-code for this would be:

```
C c = (C) MOP.newReified (‘‘C’’, ‘‘M’’);
```

This feature is quite important if one wants to be able to use reified instances of C in a component that was originally designed to be a client of objects of type C[7]. In *ProActive*, for example, the objects that intercept method calls are called *stubs*. They are instances of classes that are subtypes of the class of the object that is reified. Stub classes are generated and compiled on demand at runtime. As all classes involved in a reified call are normal Java classes and the execution environment is absolutely standard, all the protection domains involved appear on the stack.

As a result, in the general case of proxy-based runtime MOPs, the meta-level component has to include the permission sets of the component it reflects on:

$$P(MetaB2) \supseteq P(B2)$$

Note that, by contrast with the VM-based case, there seems to exist no means to avoid it. However, such a restriction is quite fair and moderate and should not be an obstacle to the use of reflection in secure component-based architectures.

Also, as can be seen from the chain of calls in the case of reified object creation, we call the MOP component to trigger the creation, then the meta-level code $MetaB2$, and finally the base code $B2$. Hence, the permissions of the MOP have to be at least equal to the permissions of $MetaB2$, which should be enough. So we have:

$$P(MOP) \supseteq P(MetaB2)$$

To generalize, the permissions of the MOP in that case have to be at least the union of the permission sets of all the meta-level components.

3.5 `doPrivileged` and Summary

As explained in section 2.1, the `doPrivileged` construct allows us to limit the computation of the intersection of the permissions to the current protection domains after `doPrivileged` is invoked. We now examine how this feature may influence the constraints on the permission sets of our example application.

Usually, `doPrivileged` calls are found in standard libraries in order to execute security-sensitive method calls which are well-understood (such as reading font files from a fonts directory) and are not a security threat, whatever the permission set of the class that calls the method is.

For compile-time and pure load-time MOPs, one might be tempted to add `doPrivileged` calls into $MetaB2$ in order to alleviate the need to take the

[7] Note that the new *dynamic proxy* feature of the Reflection API only allows dynamic stub generation for interfaces but not for classes.

union of $P(B2)$ and $P(MetaB2)$ for the set of permissions of the resulting class. However, this does not work because at runtime the resulting class belongs to a single composite principal.

On the other hand, using the `doPrivileged` construct proves interesting for run-time MOPs as it alleviates the following constraint:

$$P(A) \supseteq P(B1) \supseteq P(MetaB2)$$

An example of this is when $MetaB2$ requires permissions that we do not want to give to the principal of $B1$. This means that $MetaB2$ no longer has to worry about the permission sets of the classes that perform reified calls, and the constraint above is no longer required to hold.

Let us now summarize the constraints on permission sets that we obtained for all the different kinds of MOPs:

Compile-time

$$(P(A) \cup P(MetaB2)) \supseteq (P(B1) \cup P(MetaB2)) \supseteq (P(B2) \cup P(MetaB2))$$

Pure load-time

$$P(A) \supseteq P(B1) \supseteq (P(B2) \cup P(MetaB2))$$

Run-time VM-based

$$P(A) \supseteq P(B1) \supseteq P(MetaB2) \supseteq P(B2)$$

Run-time proxy-based

$$P(A) \supseteq P(B1) \supseteq P(MOP) \supseteq P(MetaB2) \supseteq P(B2)$$

Fig. 7. Summary of the constraints on the permission sets

4 Related Work: MOP-Based Security Enforcement

The idea of using MOPs for expressing and implementing security policies is not a new one. The security aspect of an application has long been recognized as fairly orthogonal to functional code, although this point has never, to the best of our knowledge, been thoroughly investigated.

A metaobject that intercepts method invocations for an object that represents a resource to be secured is the ideal place for implementing access control checks without having to mix functional code with security-related code. In a model based on capabilities, a metaobject attached to a reference can control the propagation of the capability across protection domains. Riechmann [24], for example, proposes a model in which metaobjects, attached to the boundary of a component, control how references to objects that live inside the component are transmitted to other components. It dynamically attaches security metaobjects to these references according to the level of trust of the component the reference is transmitted to.

A similar idea was developed with the concept of *Channel Reification* [18, 2]. This model enhances the message reification model with the notion of history (or state). The model was implemented in Java as part of a history-dependent access control mechanism [3] that goes beyond the well-known access matrix model [25], which is essentially a stateless access control mechanism. The channel reification model is also claimed to be superior to the meta-object model where a single metaobject monitors all access to a resource because it works with method-level granularity and can be used for implementing *role-based access models* [26] which are particularly well-suited to distributed object-oriented computing.

Another instance of using MOPs for implementing security policy is presented in [30]. The idea here is to use the load-time MOP Kava (see 3.2) in order to adapt third-party components to meet real-world security requirements. The authors contrast their approach with the wrapper-based approach adopted by the Enterprise Java Beans framework and argue that load-time MOPs provide a cleaner implementation of meta-level security policies. In addition, having a separate meta-level for the security policy attached to a component eases the expression of any kind of high-level security mechanisms, while the wrapper-based approach seems less expressive and is in fact only appropriate for enforcing access control on resources.

These experiments proved the feasibility of using MOPs for implementing security policies. Another issue is to know if this approach is worthwhile, i.e. if the expression of a security policy at the metalevel is orthogonal enough to functional code for this approach to be used in real-world applications.

In the context of Java, the very fact that the declaration of which permissions are granted to which piece of code (the *policy* file) is separated from the source code might be interpreted as a proof that functional code and the declaration of security policies are orthogonal. However, there is at least one hint that functional code and security are not that orthogonal. In practice, the security policy as described in the *policy* file is unworkable if the code does not make use of the `doPrivileged` call for bypassing part of the security mechanism.

5 Conclusion

We first defined a classification of MOPs based on the time of reflection shift (compile-time, load-time, run-time VM-based, run-time proxy-based). From that basis, and within a typical component-based reflective application, we have demonstrated that the type of MOP being used greatly impacts the constraints on permission sets.

A compile-time MOP globally imposes the constraint that the permissions of a meta component have to be added to the corresponding base-level component, and moreover, to all its client components. More generally, if a meta component reflects on several base components, the former permissions will have to spread to all the base components and to all their clients. This fact might be an important security concern as meta-level code can potentially require high permissions (for instance in order to write persistent data on disk) that would be dangerous to associate to untrusted reflected-on components.

A similar problem, but with less consequences, occurs with load-time MOPs as the meta-level permissions only need to spread to the base-level class, and not to all the clients. That is still an important issue to take into account.

In the case of run-time proxy-based MOPs, the only constraint seems to be the inclusion of the meta-level permission set within its base-level permission set. Indeed, this is a rather reasonable constraint. When the technique of generated proxies is used, the proxy code must also have at least the base-code permissions. Furthermore, if several meta-level components are used, the permissions of the MOP have to be at least the union of all their permissions.

Finally, a run-time VM-based MOP in theory implies similar constraints as a run-time proxy-based MOP. However, as the MOP is actually within the modified VM, the constraints can be alleviated by the MOP implementer if needed. This MOP architecture, besides its specific software diffusion limitation, raises the problem of letting the meta-level control the inner working of the VM. A solution might be to define specific MOP permissions to provide the integrator with the ability to control and enforce, in a declarative manner, what can be reflected on within the VM.

References

[1] Martin Abadi, Michael Burrows, Butler Lampson, and Gordon Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, September 1993.

[2] M. Ancona, W. Cazzola, and E. B. Fernandez. Reflective authorization systems: Possibilities, benefits, and drawbacks. *Lecture Notes in Computer Science*, 1603:35–50, 1999.

[3] Massimo Ancona, Walter Cazzola, and Eduardo B. Fernandez. A history-dependent access control mechanism using reflection. In *Proceedings of the 5th ECOOP Workshop on Mobile Object Systems (MOS'99)*, Lisbon, Portugal, June 1999.

[4] Pierre Bieber, Jacques Cazin, Virginie Wiels, Guy Zanon, Pierre Girard, and Jean-Louis Lanet. Electronic purse applet certification: extended abstract. In Steve Schneider and Peter Ryan, editors, *Electronic Notes in Theoretical Computer Science*, volume 32. Elsevier Science Publishers, 2000.

[5] D. Caromel, W. Klauser, and J. Vayssiere. Towards Seamless Computing and Metacomputing in Java. *Concurrency Practice and Experience*, 10(11–13):1043–1061, November 1998.

[6] Shigeru Chiba. A metaobject protocol for C++. In *OOPSLA '95 Conference Proceedings: Object-Oriented Programming Systems, Languages, and Applications*, pages 285–299. ACM Press, 1995.

[7] Shigeru Chiba and Michiaki Tatsubori. Yet another java.lang.class. In *ECOOP'98 Workshop on Reflective Object-Oriented Programming and Systems*, Brussels, Belgium, July 1998.

[8] Geoff A. Cohen, Jeffrey S. Chase, and David L. Kaminsky. Automatic program transformation with JOIE. In *Proceedings of the USENIX 1998 Annual Technical Conference*, pages 167–178, Berkeley, USA, June 15–19 1998. USENIX Association.

[9] José de Oliveira Guimarães. Reflection for statically typed languages. In Eric Jul, editor, *ECOOP '98—Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 440–461. Springer, 1998.

[10] J. Ferber. Computational reflection in class based object-oriented languages. *ACM SIGPLAN Notices*, 24(10):317–326, October 1989.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software.* Addison Wesley, Reading, 1996.

[12] Li Gong. Secure Java class loading. *IEEE Internet Computing*, 2(5):56–61, 1998.

[13] Li Gong. *Inside Java 2 platform security: architecture, API design, and implementation.* Addison-Wesley, Reading, MA, USA, june 1999.

[14] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, Reading, USA, 1997.

[15] Gregor Kiczales and Jim des Rivieres. *The art of the metaobject protocol.* MIT Press, Cambridge, MA, USA, 1991.

[16] Juergen Kleinoeder and Michael Golm. Metajava: An efficient run-time meta architecture for java. Techn. Report TR-I4-96-03, Univ. of Erlangen-Nuernberg, IMMD IV, 1996.

[17] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, Reading, USA, 1997.

[18] Gabriella Dodero Massimo Ancona, Walter Cazzola and Vittoria Gianuzzi. Channel reification: A reflective model for distributed computation. In *Proceedings of IEEE International Performance Computing, and Communication Conference (IPCCC'98)*, pages 32–36, Phoenix, Arizona, USA, Feb 1998.

[19] Sun Microsystems. The JavaBeans API Specification, July 1997.

[20] Sun Microsystems. The Java Core Reflection API, 1998.

[21] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages (POPL)*, pages 228–241, San Antonio, Texas, Jan 1999.

[22] Alexandre Oliva and Luiz Eduardo Buzato. The design and implementation of Guaraná. In *Proceedings of the Fifth USENIX Conference on Object-Oriented Technologies and Systems*, pages 203–216. The USENIX Association, 1999.

[23] Barry Redmond and Vinny Cahill. Iguana/J: Towards a dynamic and efficient reflective architecture for java. In *ECOOP 2000 Workshop on Reflection and Metalevel Architectures*, June 2000.

[24] T. Riechmann and J. Kleinoeder. Meta objects for access control: Role-based principals. In C. Boyd and E.Dawson, editors, *Proceeding of the Third Australasian Conference on Information Security and Privacy*, number 1438 in Lecture Notes in Computer Science, pages 296–307. Springer, July 1998.

[25] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9), September 1975.

[26] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *Computer*, 29(2):38–47, February 1996.

[27] Peter Sewell and Jan Vitek. Secure composition of insecure components. In *Proceedings of the Computer Security Foundations Workshop, CSFW-12*, 1999.

[28] Michiaki Tatsubori. An extension mechanism for the Java language. Master's thesis, Graduate School of Engineering, University of Tsukuba, 1999.

[29] I. Welch and R. Stroud. From Dalang to Kava — the evolution of a reflective Java extension. In Pierre Cointe, editor, *Proceedings of the second international conference Reflection'99*, number 1616 in Lecture Notes in Computer Science, pages 2 – 21. Springer, July 1999.

[30] I. Welch and R. J. Stroud. Using reflection as a mechanism for enforcing security policies in mobile code. In *Proceedings of ESORICS'2000*, number 1895 in Lecture Notes in Computer Science, pages 309–323, October 2000.

[31] Ian Welch and Robert Stroud. Using metaobject protocols to adapt third-party components. Work-in-Progress paper presented at Middleware'98, Lake District England, September 1998.

[32] Zhixue Wu and Scarlet Schwiderski. Reflective Java: Making Java even more flexible. Technical report, ANSA, 1997.