Access Control to Reflection with Object Ownership

Camille Teruel
Inria Lille – Nord Europe, France
camille.teruel@inria.fr

Stéphane Ducasse
Inria Lille – Nord Europe, France
stephane.ducasse@inria.fr

Damien Cassou
Lille 1 University, France
damien.cassou@inria.fr

Marcus Denker
Inria Lille – Nord Europe, France
marcus.denker@inria.fr

Abstract

Reflection is a powerful programming language feature that enables language extensions, generic code, dynamic analyses, development tools, etc. However, uncontrolled reflection breaks object encapsulation and considerably increases the attack surface of programs *e.g.,* malicious libraries can use reflection to attack their client applications. To bring reflection and object encapsulation back together, we use dynamic object ownership to design an access control policy to reflective operations. This policy grants objects full reflective power over the objects they own but limited reflective power over other objects. Code is still able to use advanced reflective operations but reflection cannot be used as an attack vector anymore.

Categories and Subject Descriptors D.3.3 [*Programming languages*]: Language Constructs and Features

Keywords reflection, encapsulation, object ownership

1. Introduction

Reflection is a powerful ability that allows programs to examine and modify their own structure and behavior (Smith 1984). Reflection helps writing highly generic code and frameworks. It can alter program interpretation to create language extensions, to perform dynamic analyses or to factor non-functional concerns. It is a basis for the implementation of development tools, dynamic software updates and self-adaptive programs. Early, reflection has been considered to fit the object paradigm well (Maes 1987; Ferber 1989), making object-oriented languages the vehicles of choice to implement reflective architectures.

However, most reflective operations break object encapsulation. For example with *object state introspection* objects can break into the encapsulation boundaries of other objects to obtain new references. This reflective operation is available in many languages *e.g.,* `instance_variable_get()` in Ruby, `getattr()` in Python, `Field.get()` in *Java*, `instVarNamed:` in Smalltalk. Consider an example consisting of two person objects, *Alice* and *Bob*, each holding a reference to its own private wallet object. In this context, if *Alice* has a reference to *Bob*, she can use object state introspection to access *Bob*'s wallet without his consent. For example in *Smalltalk*:

```
bobWallet := bob instVarNamed: #wallet.
```

Alice can now access the content of *Bob*'s wallet, like is credit card. Such leaked object reference can in turn be introspected: from a reference to one object, all indirectly-connected objects become reachable. So once *Alice* has obtained *Bob*'s credit card she can introspect it to get its PIN:

```
creditCard := bobWallet creditCard.
pin := creditCard instVarNamed: #pin.
```

From a security point of view the tension between reflection and object encapsulation is problematic. An unrestricted access to advanced reflective features allows any code to inspect and corrupt any code loaded in the runtime or any computation running therein. The deployment of an application requiring reflection support in a shared runtime relies on trusting this application to not misuse reflection. This situation is not satisfactory.

This tension is also problematic because reflection is used to implement security mechanisms (Riechmann and Hauck 1997; Ancona et al. 1999). If still available, reflection can easily bypass these mechanisms. Consequently these mechanisms must implement policies that forbid reflection

except for their own implementation. Rather than forbidding reflection or most of its applications, we want to control it.

In this article, we reconcile reflection and object encapsulation via an access control policy to reflective operations. Such policy has to determine when breaking into the encapsulation boundary of an object is legitimate. To this end, we explore the notion of dynamic object ownership (Noble et al. 1999; Gordon and Noble 2007) that organizes object graphs around a notion of ownership. We use the object ownership relation to determine access rights to reflective operations on a per-object basis. These access rights are thus based on the dynamic arrangement of objects rather than on static relations between structural entities (*e.g.,* classes and packages) as it is the case for visibility modifiers in most languages. Ownership information becomes our basis to decide when it is legitimate for an object to break into the encapsulation boundary of another one using reflection.

The contributions of this article are:

- a description of the problem of object encapsulation violations caused by reflection, a presentation of the existing solutions and their shortcommings (Section 2);
- a presentation of an access control policy to reflective operations based on dynamic object ownership (Section 3);
- an evaluation of this access control policy through a prototype *Metaobject protocol* (Section 4).

2. Problem

Most reflective operations break object encapsulation. To reconcile reflection and encapsulation we want an access control policy to reflective operations. The purpose of such policy is to decide when an object can legitimately use a reflective operation on another object and thus potentially break the encapsulation of the latter.

In this section we set up the context of the discussion to explain the tension between encapsulation and reflection. To stress this tension we analyse it through the point of view of a security model where object encapsulation is a central requirement: the *Object-Capability Model* (*OCap* model). This brief introduction to the *OCap* model also introduces some vocabulary.

2.1 Object-Capability Model

The Object-Capability Model (Miller 2006; Miller and Shapiro 2003) is a capability-based security model that builds upon the object paradigm. In capability-based security, a capability is an unforgeable reference to a *resource* together with a set of access rights to this resource. Capabilities are *unforgeable i.e.,* it is impossible to counterfeit a capability. A capability grants *subjects* holding it the *permission* to invoke operations of the associated resource according to the associated access rights. In a capability system it is impossible to designate a resource without having the permission to access this resource: a capability is at the same time a designation and a permission.

The *OCap* model applies capability-based security to object-oriented programming by treating objects both as subjects and resources. An object is a resource for objects holding a reference to it and a subject for objects it holds a reference to. In a memory-safe language a capability is encoded as an object reference: the absence of pointer arithmetic ensures that object references are unforgeable. A capability grants a subject the right to send messages to the resources.

In the context of our wallet example, if *Alice* holds a capability on *Bob*, she can send him messages. On his side, *Bob* encapsulates its capability to its wallet: he is the only one to decide if he gives its wallet away to strangers. That is to say, it is up to the code of *Bob* to take care of exercising its wallet capability himself and not to return or introduce this capability to collaborating objects. For example, if *Alice* were to ask *Bob* to pay her for some item, *Bob* would have to take some coins out of his wallet and give them to *Alice*, but he would not give away his wallet for *Alice* to take these coins herself.

The *OCap* model relies crucially on object encapsulation: a capability only permits a subject object to send messages to the associated resource object, but not to access the capabilities of the resource without its consent. In the introduction we saw that the global availability of object state introspection allows any object to access all the objects indirectly connected to it. This is unacceptable because any capability would bring as much authority than the sum of capabilities in its connected object graph.

The integration of reflection and the *OCap* model in a same language is thus challenging. Consequently *OCap* languages provide limited reflective abilities. For example, *Joe-E* (Mettler et al. 2010), an object-capability subset of *Java*, limits reflection to introspection of public members. Another example is the *E* language (Miller 2006) that limits reflection to the execution and interception of message sends.

An *OCap* language could allow an object to perform many reflective operations on itself. An object inspecting its own state and behavior, instrumenting its own code or altering its own interpretation does not contradict the principles of the *OCap* model as soon as these reflective operations affect only the object itself. Only the reflective operations that an object performs on another object needs to be controlled. More precisely, an object should not be able to perform a reflective operation that produces an effect that could not have been carried out without reflection. This corresponds to the principle of *reflection protection* formulated by De Meuter *et al.* in the context of the *ChitChat* language (De Meuter et al. 2005). The reflection protection principle states that, by default, an object does not expose more of itself at the meta-level than it does at the base-level. An object can still expose more at the meta-level than it does at the base-level, but only

if it chooses to do so. The set of reflective operations that respect the reflection protection principle depends on the host language. For example in *Java*, the accessibility of a field or a method has to be taken into account. Reflection protection allows method calls, field readings and field writings but only if the corresponding member is accessible from the subject object according to the visibility restrictions. Another example is Smalltalk, where instance variables are private to each object and all methods are public. In this case reflection protection allows every message send but no access to instance variables.

While the reflection protection principle reconciles reflection with the *OCap* model, it also constrains it and limits its power. We are looking for a more permissive approach.

2.2 Reflection as Separate Capabilities

A first step toward the reconciliation of reflection and the *OCap* model is to encapsulate reflective operations in separate objects. Likewise the capability to send messages to an object and the capability to reflect on it are kept distinct. Additionally this separation prevent name clashes between the base-level operations and the reflective operations. The objects that expose the reflective operations are called *metaobjects*.

To allow fine-grained access control, a reflective architecture should provide *metaobjects* that grant a subject the right to perform reflection on a single object. In the rest of this paper we consider reflective architectures that fulfill this requirement. For example, the reflective API of Java does not fulfills this requirement: in Java, the method getClass() returns a reification of the receiver's class (instance of the class Class), that allows subjects to inspect and modify any instance they reference. Reflective architectures that expose reflective operations via metaobjects can be divided in two categories: those where metaobjects are accessed directly from objects and those where metaobjects are acceded indirectly via an external provider.

2.2.1 Direct Access to Metaobjects

Many reflective architectures grant base-level objects the right to access the metaobject of any other object directly. When the access to a metaobject is negotiated via message passing, and when the corresponding accessing methods can be redefined, objects may restrict the set of reflective operations they provide (De Meuter et al. 2005). Typically, this kind of solution has two drawbacks.

The first drawback is a lack of principal information: an object has no easy mean to know which subject object is asking for its metaobject. One solution can rely on the different method visibilities provided by the language to grant different subjects different levels of authority. But in most languages method visibilities are based on static criteria, like the package or the hierarchy of the class defining the method. Such solution prevents fine-grained access control by making the assumption that all instances of a class have necessary the same rights. And more often than not, different instances need different access rights to properly encapsulate their respective private state from each other. In the context of our wallet example, restricting the reflective access of someone's wallet to instances of the Person class (like *Java*'s private modifier) would imply that *Alice* has the permission to access *Bob*'s wallet reflectively. Likewise, restricting the reflective access of linked-list nodes to instances of the LinkedList class would imply that any linked list has the permission to access any other linked-list nodes. Also, if a malicious program manages to take over one instance, it can then take over all other instances it can access. Such solution also makes the assumption that principal boundaries matches the visibility scopes provided by the language. A more satisfying solution is to take the identity of subjects into account for more fine-grained control. This is what the access controlled policy presented in this article does. Of course this is only possible if the language offers a way to know the sender of a message.

A second drawback is that this kind of access to metaobjects forces the repetitive redefinition of these access methods and may lead to duplicated code. Also, developers may either implement over-restricting access policies and prevent the usage of development tools and dynamic analyses or implement over-permissive policies and introduce security breaches. To avoid these repetitive redefinitions, the default accessing methods must provide a sensible default access control policy. This is one of the contributions of this article.

2.2.2 Indirect Access to Metaobjects

Access to metaobjects can also be done indirectly via an external provider. This is the case in mirror-based reflective architectures (Bracha and Ungar 2004), where all reflective operations are performed via metaobjects called *mirrors*. A mirror-based reflective architecture follows three design principles:

- Encapsulation: The implementation of reflective operations is encapsulated. It is then possible the substitute one implementation with another, *e.g.,* for adapting existing development tools to a different runtime or to provide reflection on remote objects.
- Stratification: The meta-level is totally separated from the base-level. Mirrors are not accessed directly via base-level objects but instead via a *mirror factory*.
- Ontological correspondence: The reflective API describes the reflected language in its entirety and distinguishes between static and dynamic aspects of the language.

In our context, stratification is the most important principle. To perform reflection upon a resource, a subject needs a capability over the resource and a capability over a *mirror factory*. Without access to a mirror factory, a subject cannot use reflection at all. Typically, a default mirror factory creates mirrors that expose all available reflective operations. A capability over the default mirror factory thus grants a subject

the right to reflect upon any object it references directly or indirectly.

Thanks to the adherence to the Abstract Factory design pattern, mirror-based reflective architectures make it possible to design custom mirror factories that produce mirrors with less authority. But such mirror factories still need to keep track of access permissions to determine which rights it grants to different subjects. Our policy uses ownership information to keep track of these permissions.

3. Access Control Policy to Reflective Operations

An access control policy to reflective operations has to decide when a subject object can legitimately perform a reflective operation on a resource object. Ideally such policy should be permissive enough to retain most of the power of reflection and restrictive enough to prevent abuses. A policy should also be generic enough to be a sensible default for most situations.

Consider the case where the subject and the resource are the same object. In this case we consider all reflective operations legitimate, *i.e.,* an object can always access its own metaobject. An object should have the right to decide for itself how it behaves: it should have the right to inspect and alter its own state as it intends. The problematic case is when the subject and the resource are different objects. Specifying access rights on a per-class basis is not a satisfactory solution because all instances of a class would have the same rights. Rather our policy takes the dynamic relations between objects into account. We can observe that the relation between *Alice* and *Bob* is not the same than the relation between *Alice* and her wallet. While *Alice* should not be able to get access to *Bob*'s wallet, she should have a privileged access to her own wallet and then to the credit card it contains. She should be able to use reflection to alter the behavior of her credit card. Likewise she can monitor and limit withdrawals when she lands it to an untrustworthy friend. The relationship between a person and his wallet is stronger than the relationship between persons: while *Alice owns* her wallet, she simply *refers* to *Bob*. The distinction between this *owning* relationship and the standard *referencing* relationship is embodied by the concept of *Object Ownership*.

3.1 Object Ownership

Object ownership was originally introduced to control the effects of object aliasing in the context of *Flexible Alias Protection* (Noble et al. 1998). It was first embodied as a type system with ownership types (Clarke et al. 1998) and was then adapted in the context of dynamically-typed languages with dynamic ownership (Noble et al. 1999; Gordon and Noble 2007). In this paper we consider the latter. Note that we do not use object ownership to control object aliasing but to keep track of reflection permissions on a per-object basis.

The notion of object ownership comes from the observation that objects are rarely autonomous but instead form aggregates. An aggregate object is composed of objects that constitute its *representation* and it refers to other objects that constitute its *arguments*. An aggregate owns its representation but simply refers to its arguments. A typical example is a linked-list: the list owns its nodes (its representation) but simply refers to the elements it contains (its arguments).

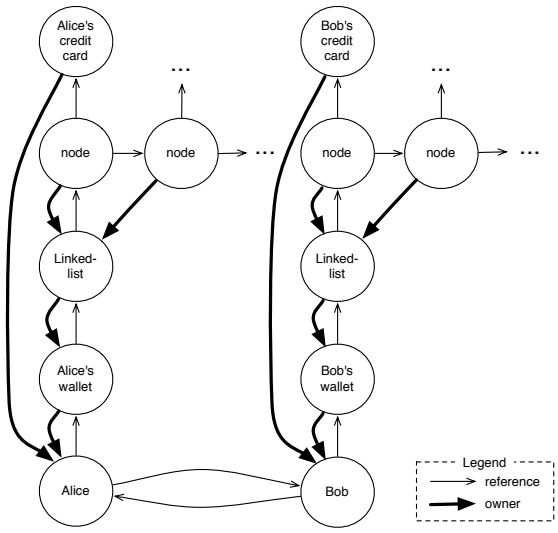

Figure 1. An example showing the ownership relation. *Bob* and *Alice* each own their own wallet and credit card. Each wallet owns its respective linked-list and each linked-list own its respective nodes. *Bob* and *Alice* only refer to each other just like each node refers to its contents.

Figure 1 shows the ownership relation in the context of our wallet example. *Alice* and *Bob* refer to each other, and both own their respective wallet and credit card. Such object graph could be the result of the following code.

```
Person>>initialize
   wallet := Wallet new.
   wallet add: CreditCard new

alice := Person new.
bob := Person new.
alice friend: bob.
bob friend: alice
```

3.2 Ownership-Based Access Control to Reflective Operations

We now present our ownership-based access control to reflective operations. Each object refers to its *direct owner*, which by default is the object that instantiated it.[1] This forms the *direct ownership* tree. Depending on the execution model of the host language, the root of this tree is either the first object instantiated or a special object. For example, for *Java* there would need to be a special object to act as the owner of objects instantiated within the main method. The *ownership*

[1] Other methods are possible to specify the direct owner of an object *e.g.,* with annotations.

relation we consider is the transitive reflexive closure of the *direct ownership* relation. In other word, we say that an object *A owns* an object *B* if:

- *A* directly owns *B* (*i.e., A* is the direct owner of *B*), or
- there exists an object *C* such that *A* owns *C* and *C* directly owns *B*, or
- *A* and *B* are the same object.

The access control policy is to grant an object the permission to perform any reflective operation on the objects it owns. For example, in the context of Figure 1, *Alice* owns her credit card and can alter its behavior to monitor or limit withdrawals. If the subject object does not own an object the policy falls back to the reflection protection principle *i.e.,* only the reflective operations that perform an action that could have been carried out without reflection are available. Reflection is not entirely forbidden: the subject object can still perform some reflective operations. In the context of Figure 1, *Alice* does not own *Bob* but can still reflectively send messages to *Bob*. But she cannot introspect *Bob* to obtain a capability to his wallet.

Root Object as Superuser Because the direct ownership relation forms a tree, the object at the root indirectly owns all objects. Consequently, this root object has the permission to perform any reflective operation on any object. This root object can thus use development tools and perform dynamic analyses on any object. For example, an application in production should be able to monitor all hash messages sent from each hash-table to determine which hash methods are slow or produce too many collisions. To perform such analysis this application should be able to reflect unrestrictedly on all its objects. At the same time, that same application may use a library that it does not trust completely. This library can use reflection but only on its own objects.

About Ownership Transitivity. Unlike other notions of object ownership our ownership relation is transitive. This transitivity stems from the power of reflective abilities. Consider three objects *A*, *B* and *C* such that *A* directly owns *B* and *B* directly owns *C*. Even without this transitivity, *A* could exercise its full reflective power over *B* to change its behavior at will. Doing so, *A* could easily obtain *B*'s authority to have full reflective power over *C*.

4. Implementation

To demonstrate our access control policy we consider its application in the context of a simple *Metaobject Protocol*[2] (MOP) (Kiczales et al. 1991) for *Pharo*, a programming language and environment of the Smalltalk family. A MOP is an object-oriented model of the interpreter of the host language. A MOP provides behavioral intercession by allowing the substitution of default metaobjects by custom ones. This alters the interpretation of the program and thus the default semantics of the language. Here, each object behavior is described by a unique metaobject. The base-level object that a metaobject controls is called its *referent*. This kind of MOP has been described in detail in the literature (Maes 1987; Ferber 1989; Mostinckx et al. 2009) and we refer to them as *object-centric MOPs* (sometime referred in literature as *the metaobject model*). In a class-based language with an object-centric MOP, the metaobject of an object is distinct from its class: the metaobject is responsible for the interpretation of the structure the class defines.

Our MOP consists of the following reflective operations:

Object state introspection: via the method
read: aVariableName.

Object state modification: via the method
write: anObject in: aVariableName.

Message reception: via the method
receive: aSelector[3] withArguments: argumentArray.

4.1 Ownership Encoding

Conceptually, encoding ownership information consists in adding a directOwner instance variable to the root class Object. In practice, *Pharo*'s virtual machine imposes some limits on the memory layout of some special classes so we use an external table to map objects to their direct owner instead.

To initialize the direct owner of a newly-created object, this object receives the message initializeDirectOwner after instantiation. The default implementation of initializeDirectOwner determines the direct owner of the newly-created object by traversing the call-stack. The receiver associated with each stack frame receives the message wantsOwnership: with the new object as argument: the first receiver that answers true become the owner of the new object. The default implementation of wantsOwnership: in Object unconditionally returns true but a class can redefine this behavior if needed.

Customisation of Ownership Tree Construction. The construction of the ownership tree can be customized for different situations. In object-oriented design, some engineering practices, like *Dependency Injection* frameworks and design patterns such as *Factory Method* and *Abstract Factory*, strive for reducing coupling between software components. These practices have in common that instantiation is performed by a third party. If these practices are followed systematically, no object instantiates objects that are part of their representation. Without special care, the resulting ownership tree would be very large and very shallow. In this context, the object that instantiates another object has to be able to let its clients become owner of that newly-created object or to transfer ownership.

For example, a factory can let its clients become the owners of the object it creates by redefining wantsOwnership:

[2] http://rmod.lille.inria.fr/archives/demos/OwnershipMOP/mop.zip

[3] In *Smalltalk*'s terminology, a *selector* is the name of a method or a message.

to answer false for the classes it instantiates. For example, a widget factory would implement wantsOwnership: as follows.

```
WidgetFactory>>wantsOwnership: anObject
    ^ (anObject class inheritsFrom: Widget) not
```

Another situation where customisation of object ownership is desirable is when an *inversion of control* scheme takes place. This is for exemple the case of dependency injection frameworks where the framework has to be able to transfer ownership of the object it creates. Our implementation allows changing the direct owner of an object. This operation is only available from an object's metaobject, so only an owner of an object can change its direct owner.

Another situation where the ownership tree construction needs to be customized is when a class wants all its instances to be owned by the same object. This is the case of immutable objects denoting values, like numbers, characters, points, etc, that are unconditionally owned by the object nil, the root of the ownership tree. This saves some space by not using the ownership map for these objects.

4.2 Implementation of Ownership-based Access Control

Our ownership-based access control policy grants an object the permission to access the metaobjects of the objects it owns. When a subject requests the metaobject of a resource it does not own, the policy is to restrict the available reflective operations. Only the reflective operations that perform an action that could have been carried out without reflection are allowed, following the reflection protection principle. In the context of Pharo, where all instance variables are object-private and all methods are public, these restrictions only allow message reception. In our implementation, wrapper objects encapsulate metaobjects to enforce these restrictions. The method meta implements the access control to metaobjects. This method is context-sensitive: it checks whether the sender of the message is an owner of the receiver or not. If the sender is not an owner the metaobject is wrapped.

These access rules extends to metaobjects returned by reflective operations. For example, if the metaobject of an object *A* is about to return the metaobject of another object *B* that *A* does not own, the metaobject of *A* wraps the metaobject of *B* before returning it. Since a wrapped metaobject is used by clients that do not own the referent, a wrapped metaobject always return wrapped metaobjects.

Our access control policy does not need to be implemented with metaobject wrappers. The restrictions could be implemented in the metaobject directly. We choose the wrapper approach because it has two advantages. First, the implementation is simpler because only the method meta is context sensitive. If the restrictions were implemented in the metaobjects directly, each methods would have to check whether the corresponding reflective operation is permitted or not. This would imply code duplication and slower execution. Second, an object can choose to pass the metaobject of an object it owns to an external object it trusts. If the restrictions were implemented in the metaobjects, they would apply to the external object and prevent delegation of reflective rights as capabilities.

Also it should not be possible to use reflection to bypass the access control mechanism implementing our policy. In our context, this mechanism is the metaobject wrapper. So it should not be possible to reflect on a wrapper to leak its wrapped metaobject. To ensure this, the owner of a wrapper is its wrapped metaobject. If an object asking for a metaobject gets a wrapper, it means that it does not own the referent. Consequently it does not own the referent's metaobject, nor the wrapper. So if this same object asks for the metaobject of the wrapper, it will get another wrapper for the metaobject of the first wrapper.

4.3 The Policy at Work

We now show our policy at work, first with proxies, a reflective mechanism that can be used as an access control mechanism, then in the context of a reflective tower. Proxies shows that our policy enables the reflective implementation of a security mechanism and prevents reflection to be used to bypass this mechanism. Reflective towers shows that our policy can control reflection at any meta-level.

Proxy-based Behavioral Intercession. Performing behavioral intercession on an object consists in replacing its metaobject. Proxies provide another way to perform behavioral intercession at a finer-granularity (Pascoe 1986; Büchi and Weck 2000; Eugster 2006; Van Cutsem and Miller 2010, 2013; Peck et al. 2015). A proxy is an object whose behavior is defined in terms of the behavior of another object called its *target*. Instead of enabling some behavioral variation for every access to an object, a proxy object enables this behavioral variation for itself but not for its target. The target continues to behave regularly but the proxy can be passed around to scope the behavioral variation to certain client objects only. A proxy is like an alter-ego of its target with a different meta-level behavior.

Proxies can be provided as an independent language feature but in the context of an object-centric MOP a proxy is a pattern: a proxy is an object whose metaobject forwards some of its operation to the metaobject of its target. Figure 2 shows an example of proxy-based behavioral intercession via message interception. *Bob* has created a proxy for its credit card and passed it to *Alice*. When *Alice* sends a message to the proxy, the proxy can decide if it forwards the message to the credit card or not depending on the policy *Bob* chose to enforce: withdrawal limit, mere logging of the operations, revocation, etc. The policy applies to *Alice* who references the proxy but not to *Bob* who references the credit card directly.

Proxies are thus good candidates to implement access control. A proxy interposes an arbitrary policy between a client and a resource. *OCap* patterns like caretakers, facets or membranes (Miller 2006; Van Cutsem and Miller 2013)

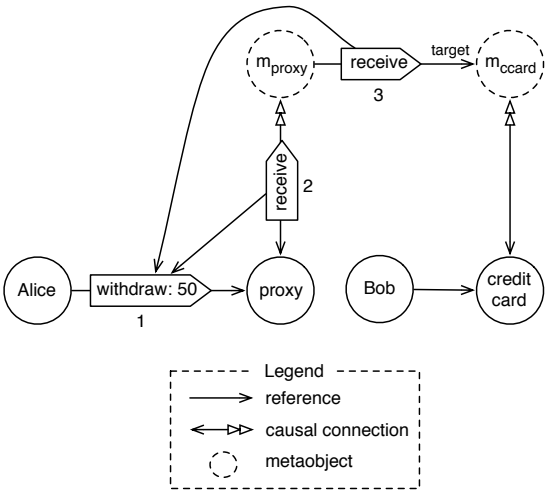

Figure 2. *Bob* has created a proxy for its credit card and passed it to *Alice*. When *Alice* sends a message to the proxy (1), the proxy sends a receive message to its metaobject with a reification of *Alice* message (2). The metaobject of the proxy can then decide whether to forward the message to the metaobject of the credit card (3). *Bob* can still use his credit card directly.

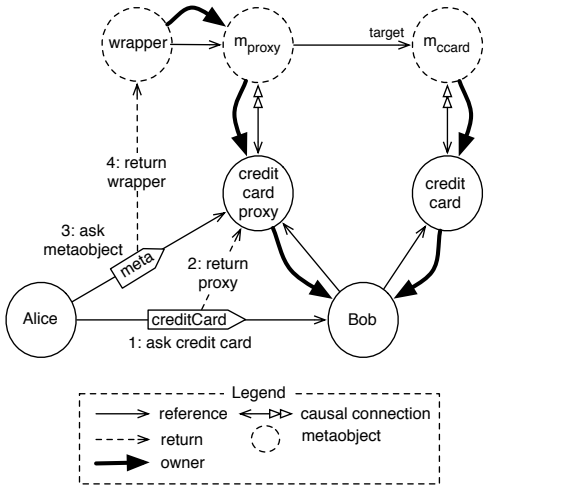

Figure 3. Proxy protection: (1) *Alice* asks *Bob* for his credit card, (2) *Bob* creates a proxy for his credit card to limit withdrawals and returns it, (3) *Alice* asks for the metaobject of the proxy to leak the credit card (target of the proxy), (4) but *Alice* does not own the proxy: she obtains a wrapper on the metaobject of the proxy and cannot leak the credit card.

are naturally expressed with proxies. When using proxies as a security mechanism, one has to face the problem of leaking target references (Eugster 2006). To avoid leaking its target object, a proxy should check if its target returns self-references and return itself instead. More generally, when a proxy implements a security policy, it is often desirable that the returned objects are also protected by the same policy. In this case a proxy has to ensure that the objects it returns are also wrapped with proxies implementing the same policy. Finally to avoid "Trojan Horse Attacks" (Riechmann and Hauck 1997), where the target pass itself or another supposedly protected object as argument to a potentially malicious object, a proxy can also wraps the arguments of the message it intercepts. These arguments are not wrapped to apply the security policy directly but to ensure that any object passed to them are. These wrapping rules are at the heart of membranes. A similar mechanism is described by Riechmann *et al.* and is discussed in Section 5.

The creator (*i.e.,* the direct owner) of a proxy has to be sure that no object apart its owners can bypass the policy the proxy enforces. In Figure 3 *Alice* asks *Bob* to lend her his credit card. Because *Bob* does not trust *Alice* entirely, he creates a proxy for his credit card to limit withdrawals to a certain amount. *Alice* hopes to bypass the withdrawal policy imposed by the proxy. To do so she asks the proxy for its metaobject to then access the metaobject of the proxy's target (*i.e.,* the credit card). But since *Alice* does not own the proxy, she obtains a wrapper on the proxy's metaobject instead. The wrapper allows her to send messages to the proxy (as she can do at the base-level) but do not permit her to access the state of the proxy's metaobject to obtain a reference to the metaobject of *Bob*'s credit card.

But the owner of a proxy does not have to own the target. In that case the proxy owner can only access a wrapper of the target's metaobject. Only message sends can be intercepted as it is the only operation that the proxy's metaobject can forward to the wrapper. This is enough to implement capability patterns such as caretakers, facets and membranes.

Reflective Tower. When performing reflection upon the meta-level, one attains the meta-meta-level (or meta-level 2). This is useful to record which meta-level operations are performed on an object (meta-tracing for optimization), to query the meta-level behavior for analysis, to debug the implementation of the meta-level, etc.

More generally one can reflect on the meta-level n at the meta-level $n + 1$. This virtually infinite stack of meta-levels is called a *reflective tower* (Smith 1984). Each level interprets operations performed at the level below. The tower is not actually infinite: the host language interpreter takes over level reification whenever operations at a level n are interpreted according to the default semantics.

Reflective towers appear naturally in the context of an object-centric MOP. Each base-level object is the first floor of its own reflective tower. The metaobject of a base-level object belongs to meta-level 1. But like other objects, the behavior of a metaobject is also defined by another metaobject, which

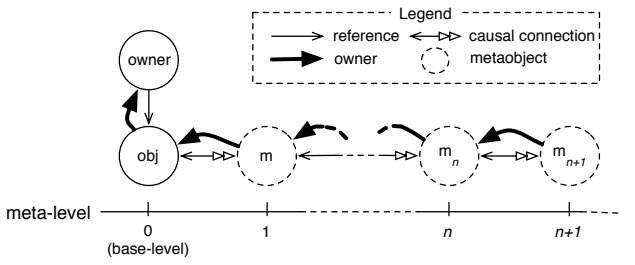

Figure 4. A chain of metaobjects forming a reflective tower with their respective owner. The owner of the object can reflect on this object at any level.

belongs to meta-level 2. The reflective tower is embodied by a virtually infinite chain of metaobjects as depicted in Figure 4.

The ability to jump from one meta-level to the meta-level above should not jeopardize the access control policy. That is to say that if the access control policy states that an object *A* has restricted access rights to the metaobject of another object *B*, *A* must not be able to circumvent this restriction by manipulating the metaobject of that metaobject. To ensure that, we customize ownership tree construction so that the direct owner of a metaobject is always its referent. Thereby, a metaobject at meta-level n owns its metaobject at meta-level $n + 1$. Consequently, an object owns all the metaobjects of its tower. It follows that only the owners of an object have the authority to reflect on this object at any level.

5. Related Work

Object Ownership. A lot of research has been done on object ownership through ownership types (Clarke et al. 1998, 2001) and dynamic object ownership (Gordon and Noble 2007; Noble et al. 1999). Originally, object ownership has been devised to control the effect of object aliasing. Later, many different ownership systems have been used for many other applications (Clarke et al. 2013): concurrency control, memory management, security, etc.

Since our access control policy relies on ownership information, it would be interesting to leverage this information with other applications. The first application that comes to mind is object alias control since it is the original application of object ownership. So an interesting question is to know if an effective alias control discipline can be compatible with our access control policy.

This seems difficult because one principle of the original ownership-based alias control (known as the *owner-as-dominator* discipline) is *"no representation exposure"*. This means that external objects can not reference the representation objects of an aggregate. The transitivity of our ownership relation contradicts with this principle. An alias control discipline has to rely on a transitive ownership relation to be compatible with our access control policy to metaobjects. Whether such alias control discipline can be devised or not remain an open question.

Security Mechanisms Implemented Reflectively Because of its ability to factor non-functional concerns, reflection has been used to implement security mechanisms. Ancona *et al.* provide an overview of these approach (Ancona et al. 1999).

For example, Riechmann *et al.* (Riechmann and Hauck 1997) propose to extend the *OCap* model with *Security Metaobjects* (SMO). They note that the frequent exchange of object references make hard to check which part of an application can access a given capability. A SMO can be attached to an object reference to control the messages that can be performed via this reference. Also an SMO can attach itself or other SMOs to incoming references (message arguments) and outgoing references (message returns). They show how this facility can implement SMOs that implicitly propagate an access control policy to all exchanged references, in a similar fashion than membranes (Miller 2006; Van Cutsem and Miller 2013). Riechmann *et al.* later proposed an extension of this model in the context of a role-based access control mechanism (Riechmann and Kleinöder 1998). In this extension, they use two kind SMOs: *principal SMOs* provides principal information that *access control SMOs* use to check access.

For any security mechanism that is implemented reflectively, it is important to ensure that reflection cannot be used to bypass them. Consequently these mechanisms must either rely on policies that forbid reflection except for their own implementation or rely on a reflective architecture that is already secured. We showed that our access control policy to reflective operations ensures that the access control policy of a proxy cannot be bypassed by subjects that do not own it. Since the functionality provided by SMOs can be emulated using a proxy-based approach, our access control policy can ensure that the resulting base-level access control policies cannot be bypassed reflectively.

Secure MOP. Caromel *et al.* (Caromel and Vayssière 2001; Caromel et al. 2001) presents concerns about MOP and security in the context of component-based applications in *Java*. These works do not concern the access control to reflective operations but the security implications of implementing a MOP within the security framework of *Java*. The security framework of *Java* is based on inspecting the call stack to determine whether the execution of a sensible operation is permitted or not. In this context, the type of MOP used greatly influences the necessary propagation of permissions from meta-level code to base-level code. This is problematic because the implementation of a MOP typically requires many permissions. The authors show that within a proxy-based run-time MOP, it is possible to capture the set of permissions before reflective calls and restore them after. MOP implementation gets more permission than base-level code even though the call stack contains stack frames from different levels.

6. Conclusion

We explored the problem of encapsulation violation caused by reflective operations and its implications on the *OCap* model. The tension between the need for object encapsulation on the one hand and the need of reflection on the other hand led us to the conclusion that we need a way to track when breaking into an encapsulation boundary is legitimate. To this end we have explored the concept of dynamic object ownership that has been originally used to tame object aliasing. Instead of object aliasing, we showed how this notion of object ownership can be used to design an access policy to reflective operations. Thanks to this access control policy, owners of an object can perform any reflective operations on that object. An object that does not own another target object can only perform reflective operations whose effect could have been carried out without reflection. This simple policy reconcile reflection and security in the context of multiple interacting software components.

We implemented this policy in the context of a prototype MOP. In this context, we showed that this policy can ensure that domain-level access-control policies, implemented reflectively with proxies, cannot be bypassed using reflection at any meta-level.

Acknowledgments

We thank Walter Rudametkin for his valuable comments about this work and the French General Directorate for Armament (*Direction Générale de l'Armement*) for sponsoring.

References

M. Ancona, W. Cazzola, and E. B. Fernandez. Reflective authorization systems: Possibilities, benefits, and drawbacks. In *Secure Internet Programming*, pages 35–49. Springer, 1999.

G. Bracha and D. Ungar. Mirrors: Design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of OOPSLA'04*, pages 331–344, 2004.

M. Büchi and W. Weck. Generic wrappers. In *Proceedings of ECOOP'00*, pages 201–225, 2000.

D. Caromel and J. Vayssière. Reflections on MOPs, components, and Java security. In *Proceedings of ECOOP'01*, pages 256–274, 2001.

D. Caromel, F. Huet, and J. Vayssière. A simple security-aware MOP for Java. In *Proceedings of the International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (REFLECTION)*, pages 118–125, 2001.

D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Proceedings of OOPSLA'98*, pages 48–64, 1998.

D. G. Clarke, J. Noble, and J. M. Potter. Simple ownership types for object containment. In *Proceedings of ECOOP'01*, pages 53–76, 2001.

D. G. Clarke, J. Östlund, I. Sergey, and T. Wrigstad. Ownership types: A survey. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, pages 15–58. Springer, 2013.

W. De Meuter, E. Tanter, S. Mostinckx, T. Van Cutsem, and J. Dedecker. Flexible object encapsulation for ambient-oriented programming. In *Proceedings of DLS'05*, pages 11–21, 2005.

P. Eugster. Uniform proxies for Java. In *Proceedings of OOPSLA'06*, pages 139–152, 2006.

J. Ferber. Computational reflection in class-based object-oriented languages. In *Proceedings of OOPSLA'89*, pages 317–326, 1989.

D. Gordon and J. Noble. Dynamic ownership in a dynamic language. In *Proceedings of DLS'07*, pages 41–52, 2007.

G. Kiczales, J. des Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

P. Maes. Concepts and experiments in computational reflection. In *Proceedings of OOPSLA'87*, pages 147–155, 1987.

A. Mettler, D. Wagner, and T. Close. Joe-E: A security-oriented subset of java. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, pages 357–374, 2010.

M. S. Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Johns Hopkins University, 2006.

M. S. Miller and J. S. Shapiro. Paradigm regained: Abstraction mechanisms for access control. In *Advances in Computing Science–ASIAN 2003. Progamming Languages and Distributed Computation*, pages 224–242. Springer, 2003.

S. Mostinckx, T. Van Cutsem, S. Timbermont, E. Gonzalez Boix, E. Tanter, and W. De Meuter. Mirror-based reflection in ambienttalk. *Software: Practice and Experience*, 39(7):661–699, 2009.

J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In *Proceedings of ECOOP'98*, pages 158–185, 1998.

J. Noble, D. Clarke, and J. Potter. Object ownership for dynamic alias protection. In *Proceedings of TOOLS'99*, pages 176–187, 1999.

G. A. Pascoe. Encapsulators: A new software paradigm in smalltalk-80. In *Proceedings of OOPSLA'86*, pages 341–346, 1986.

M. M. Peck, N. Bouraqadi, L. Fabresse, M. Denker, and C. Teruel. Ghost: A uniform and general-purpose proxy implementation. *Science of Computer Programming*, 98:339–359, 2015.

T. Riechmann and F. J. Hauck. Meta objects for access control: Extending capability-based security. In *Proceedings of the Workshop on New security paradigms*, pages 17–22, 1997.

T. Riechmann and J. Kleinöder. Meta objects for access control: Role-based principals. In *Proceedings of the Australasian Conference on Information Security and Privacy (ACISP)*, pages 296–307, 1998.

B. C. Smith. Reflection and semantics in lisp. In *Proceedings of POPL'84*, pages 23–35, 1984.

T. Van Cutsem and M. S. Miller. Proxies: Design principles for robust object-oriented intercession APIs. In *Proceedings of DLS'10*, pages 59–72, 2010.

T. Van Cutsem and M. S. Miller. Trustworthy proxies. In *Proceedings of ECOOP'13*, pages 154–178, 2013.