Meta Objects for Access Control: A Formal Model for Role-Based Principals

Thomas Riechmann
Martensstr. 1, D-91058 Erlangen
University of Erlangen, Germany
Dept. of Computer Science IV
+49-9131-85-2-7269
riechmann@cs.fau.de
http://www4.cs.fau.de/~riechmann/

Franz J. Hauck
Martensstr. 1, D-91058 Erlangen
University of Erlangen, Germany
Dept. of Computer Science IV
+49-9131-85-2-7906
hauck@cs.fau.de
http://www4.cs.fau.de/~hauck/

*This work is supported by the *Deutsche Forschungsgemeinschaft DFG* Grant *Sonderforschungsbereich SFB 182*, Project *B2*.

Abstract

Object-based programming is becoming more and more popular and is currently conquering the world of distributed programming models. In object-based systems access control is often based on capabilities, despite the difficulty to keep track of their distribution. Access control lists are used only rarely, as information about the principal on whose behalf an operation is to be executed is needed and it is difficult to determine which principal information to use for a specific method invocation. Current object-based systems use domain-based or thread-based principals. Domains or threads are associated with principals. If a specific object or a specific thread invokes a method, the invocation is always executed on that principal's behalf. Both policies suffer from the reference proxy problem: A low privileged object can pass references to a highly privileged object and may animate it to call methods with its high privileges via these obtained references (Unix S-bit problem). As there are no formal models for such systems, we cannot decide if such a situation actually occurs. On the other hand, most mandatory access control policies (where we have formal models) are too restrictive for many applications. In this paper, we introduce role-based principals. An object domain may act in different roles to different other parties. Each object reference to objects of other domains is associated with a specific role, which determines trust, authentication (i.e., which principal information to use) and allowed data flow via the reference. Exchanged references automatically inherit the role. By initially defining such roles, we can establish a security policy on a very high abstraction level. We provide a formal model and present two examples, where we show that we can prove that accidental propagation of references and unintentional use of privileges are prevented.

1 Introduction

The object-based programming paradigm is becoming more and more popular. Currently, it is conquering the world of distributed programming models. There are two basic paradigms for access control in such systems. Capabilities [2], [18], [6], [16], [8] is the most obvious one: An object reference is per se a capability. Capabilities suffer from the disadvantage that it is hard to keep track of their distribution [5]. It is hard to determine, who is able to access a specific object. It is also difficult to do auditing and accounting in pure capability-based systems.

So most object-based systems additionally support access control lists (ACLs). Examples are CORBA [10], DSOM [1], and Legion [19]. With ACLs it is easy to determine who has access to specific objects; auditing and accounting are easily implemented. For each operation (that is, each method invocation) information about the principal on whose behalf an operation is to be executed is needed. For each method invocation, such principal information may be provided by the caller. Then the invocation is executed on behalf of that principal. The principal is held responsible for the call and access checks can be done.

In object-based environments, there are three common policies for the decision on how to provide principal information for method invocation [17]:

- The principal information can be provided explicitly for each call.
- The principal information can be bound to objects or groups of objects (e.g., domains). Examples are DSOM, Java [13], Legion, and Unix. An object uses the same principal information for every call it executes (object-group–based principals).
- The principal information can be bound to threads (e.g., Java-1.2 [15] uses a mixture of a domain-based and a thread-based model). All invocations executed by a thread use the thread's principal information.

These three policies suffer from disadvantages: Call-based decision leads to an unstructured and distributed security policy. An object-group–based decision suffers from the reference proxy problem: a low-privileged application part can pass object references to a high-privileged application part to make that part accidently call methods via that references with its high privileges. Thread-based decision suffers from the callback problem: If a privileged thread invokes methods of an untrusted object, the untrusted object gets all privileges of the thread.

Additionally, all of these policies have another disadvantage in common: there are no formal models to check if the above mentioned problems actually occur in a system.

On the other hand for most systems and applications mandatory policies like multilevel security or role-based access control (RBAC, [3]) are too restrictive and even RBAC can suffer from the above mentioned problems[1].

In this paper, we introduce a new security paradigm which eliminates these problems. Our mechanism allows us to configure roles: An application part can act in different roles to different other parties. It can use different principal information for authentication for different roles (*role-based principals*). In Fig. 1, the system administration application is a highly privileged application: it has privileges to alter the system's password database. On the other hand it does not need these privileges for printing. With role-based principals we can define two roles which are assigned to two different principals and are used in these two different situations.

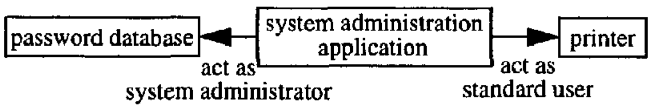
password database
system administration application
printer
act as system administrator
act as standard user

Fig. 1 Role-based principals

As we consider object-oriented systems, we assign roles to object references: The system administration application has, for example, an object reference to the printer object which is associated with the role "standard user". Our system administration application might even have a second reference to the printer object for administrating the printer (e.g., resetting it) with administration privileges (role "system administrator"). We have the principle of least privilege on a per-reference basis: The application only uses the minimal privileges it needs for interaction with other parties. The configuration of these roles is orthogonal to the application implementation and is automatically applied to object references that are exchanged between application parts.

We achieve that with so-called *Security Meta Objects* (SMOs) [11]. Such an SMO can be attached to an object reference. It can provide principal information for method invocation, keeps track of references that are passed due to method calls and can do access checks. In this paper we will concentrate on SMOs that provide principal information and keep track of references.

Each SMO is associated with a specific role and implements the role-specific behavior, for example, which principal information to use for method invocations and what to do with passed parameters and return values (exchanged references). When such an SMO is attached to a reference, calls via this reference are automatically processed by the SMO and use the role-specific behavior of the SMO.

1. RBAC is a sort of mandatory access control. A principal can choose a specific role from a set of roles associated with it. In that role it can execute transactions which are assigned to that role. The roles are independent of each other, so security problems between different roles can be prevented. But, when used in object-oriented environments, for each role the allowed transactions have to be selected carefully. Otherwise the thread-based–principal problem occurs.

In many object-oriented systems object references, which are in fact capabilities, can be passed without restriction. As an object reference with attached SMO is a capability for method invocations on behalf of a specific principal, we have to restrict propagation of such references. We define so-called *virtual object domains*, which contain a set of objects and object references (e.g., the above mentioned "system administration application" may be such a domain). If an object reference leaves such a domain (that is, the capability is passed to objects outside of the domain), our SMOs are invoked and they can deny passing or remove the principal-providing SMO from the reference before it is actually passed.

We define a formal model for SMOs that allows to analyze systems built with SMOs. Our formal model defines the semantics of SMOs exactly. We will show that we are able to prove specific properties of such systems.

The paper is structured as follows: In Section 2 we introduce SMOs and virtual object domains and present the basic idea of our security mechanism. In Section 3 we define the formal model for SMOs. In Section 4 we show two applications of our formal model.

2 Security Meta Objects and Virtual Domains

In this section we explain SMOs and what we call a virtual object domain. Although SMOs can also be used to restrict access via a reference ([11], [12]), we will concentrate on the possibility to provide principal information and restrict propagation of object references.

As in most object-based models, we see object references as capabilities. If a client has an object reference at hand, it can access the corresponding object. If a client cannot get an object reference to an object, it is not able to access it. We extend this simple model by adding the possibility to attach one or more special objects to an object reference. These special objects are invoked for each security-relevant operation on the object reference. The special objects are not visible to the application; that is, protected and unprotected object references look the same to the application. In general, such special objects can be considered as meta objects [7]; we call them *Security Meta Objects* or SMOs for short.

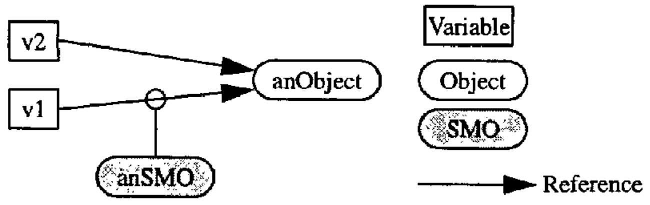

Fig. 2 A reference with an attached security meta object

SMOs are attached on a per-reference basis. There may be many references to an object, each with a different set of attached SMOs. Fig. 2 shows an object reference stored in variable `v1`, which has a meta object `anSMO` attached to it. The SMO is invoked each time a method is called via this reference. There may be other references to the same object in the system, with no, another, or even multiple SMOs attached to it (e.g., the reference stored in variable `v2`).

We will only look at SMOs that affect distribution of references and provide principal information. For example, in Fig. 2 the object `anObject` may be only accessible by a specific principal (this restriction could be implemented by an access control list in object `anObject`[2]). In our model, principal information (information about on whose behalf a call is to be executed) is provided by SMOs. Thus, `anObject` may be accessible via the reference stored in `v1`, if `anSMO` provides the required principal information. In our example `v1` is a capability for method invocations to object `anObject` on behalf of a specific principal.

SMOs are also able to keep track of object references which are to be propagated due to method invocations: If a method is to be invoked via a reference with SMO attached, each parameter of the invocation is passed to the SMO. The SMO might cancel the method invocation by disallowing the parameter to be passed or it might attach or detach SMOs to or from the parameter. We use this mechanism to implement virtual domains.

2.1 Virtual Domains

Our model is just based on SMOs. They implement the functionality of providing principal information and restricting propagation of references. SMOs seem to suffer from the disadvantage of distributed policies (each SMO implements a part of the policy). If we look at a system with SMOs, it is very difficult to determine the global security policy. That is why we introduce so-called virtual domains as a global property of the system. If initially SMOs are attached to cross-domain references in a certain way the SMOs will take care of this property and establish the virtual domain.

A virtual domain consists of objects and object references. Only objects in a domain are able to use the object references of the domain. An object domain may have different roles. For example, objects of the domain may act on behalf of different principals when interacting with other domains: Object references to other domains may have different SMOs with different principal information attached to them. As such object references are capabilities for method invocation on behalf of a specific principal, we have to be careful: if we pass such a reference to another domain, that domain is able to act on behalf of that principal. In Fig. 3 we have one domain *d1* with a reference *or* with no SMO attached to it. If we call a method via this reference with parameter (e.g., *or*.method(*p*)), we may pass a parameter with principal SMO attached (*p* is passed as *pn*). The target domain is able to act on that principal's behalf.

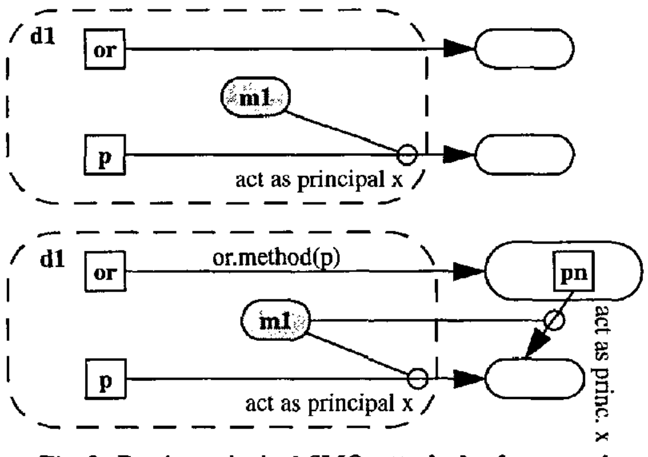

Fig. 3 Passing principal-SMO–attached references via unprotected references is dangerous

2. We can also implement such access control lists with SMOs, but in this paper we will not examines such SMOs.

In some special cases we might want to delegate our rights, but certainly not in general. Usually, when we have a reference with a principal SMO attached, we only want our object domain to be able to act on the principal's behalf. As mentioned above, SMOs are able to keep track of passed references. We use that feature to remove principal SMOs, when references leave a domain.

We have to make sure, that our SMOs are invoked, when references leave the domain. The only way to do so is to have SMOs attached to all references which refer to other domains. So, references like `or` in Fig. 3 must not exist in our system; Fig. 3 shows an invalid state of our system.

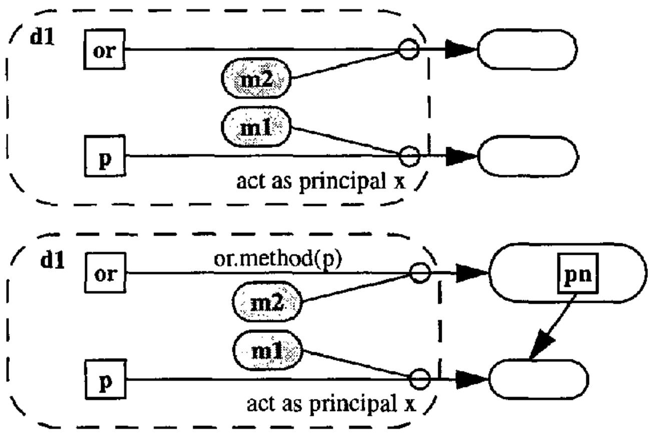

Fig. 4 SMOs are removed automatically when leaving a domain

In Fig. 4 shows a secure state. The references *or* and *p* are protected by SMOs. When the parameter *p* is to be passed, it is first given to the SMO *m2*, which controls the invocation. It notices the attached SMO *m1* and removes it.

Let us now examine, what happens, if we have references into a domain. In Fig. 5 we have a reference *or*, which points into domain *d1*. If we call a method which returns a reference (e.g., *rn*=*or*.method()), we again pass a principal SMO reference out of our domain.

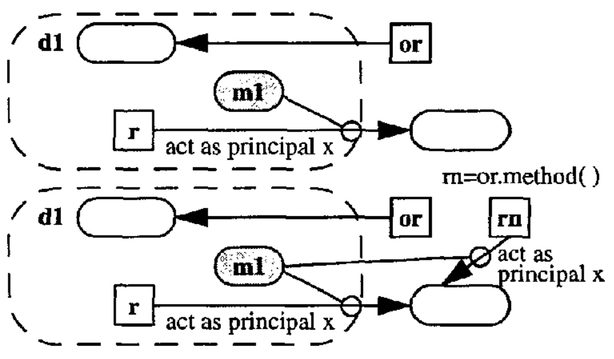

Fig. 5 References into a domain without SMO attached are dangerous, too.

So, also references into a domain have to be protected with an SMO (Fig. 6). The SMO for such references has to be different: our domain *d1* does not want, for example, to provide principal information for method invocations via object reference *or*. Instead of using completely different SMOs, we define two sorts of attachments: source attachment (src) and destination attachment (dst). If an SMO is attached in source mode, it provides principal information. If it is attached in destination mode, it does not provide principal information. Source-attached and destination-attached SMOs keep track of exchanged references: If we pass parameters via references with source-attached SMO or pass return values via references with destination-attached SMO, these references leave our domain (in our formal model, which we will describe later, the function *out* of the SMO is called). Parameters passed via references with destination-attached SMO or return values passed via references with source-attached SMO enter our domain (the function *in* of the SMO is called).

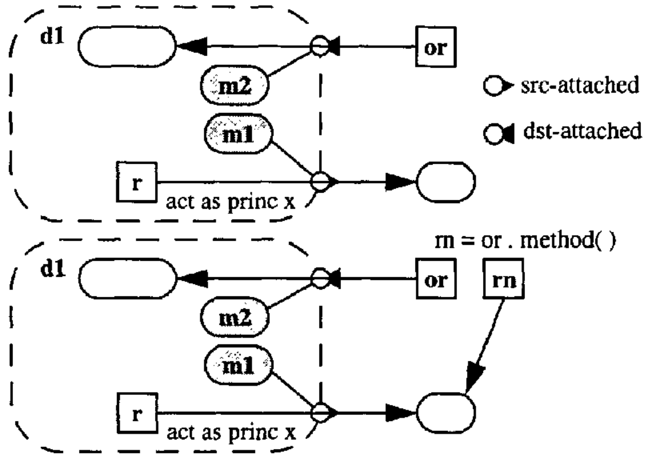

Fig. 6 SMOs are removed automatically when leaving a domain

Let us now present an example with two domains. In Fig. 7 (upper part) we have two object domains: `d1` and `d2`. Initially `d1` has the object reference `or` to an object in `d2`. Because the boundaries of `d1` and `d2` are crossed, it has two SMOs `m1` and `m2` attached to it. The SMO `m1` is attached in source mode, `m2` is attached in destination mode. Let us assume, in `d1` the method call `rn=or.someMethod(p)` is executed. This method call passes *p* as parameter. The parameter is stored in the target domain `d2` and it returns `r` as result.

When calling the method, principal information may be provided by `m1` to authenticate the call. The SMO `m2` may not provide principal information, as only source-attached SMOs may provide principal information in our model.

The parameter p is passed via the reference `or` to domain `d2`. The SMOs have to keep the virtual domain property. In our example they simply attach themselves to the passed object references (`p`, `r`). Note, that after the method call invocations via `rn` are authenticated by the principal information `m1` provides; calls via `pn` are authenticated by the principal information provided by `m2` (if it provides principal information).

The SMOs (e.g., *m1*) implement a specific policy (e.g., provide principal information) for a specific role: The initial object reference *or* and all resulting object references (*rn*, *pn*) inherit that role. The SMOs use the same policy for such references. There may be other references from *d1* to *d2* or to other domains with different roles (that is, with different SMOs attached to them, which can provide different principal information).

In the next section we will provide a formal model for our SMOs and define the semantics of the SMOs exactly.

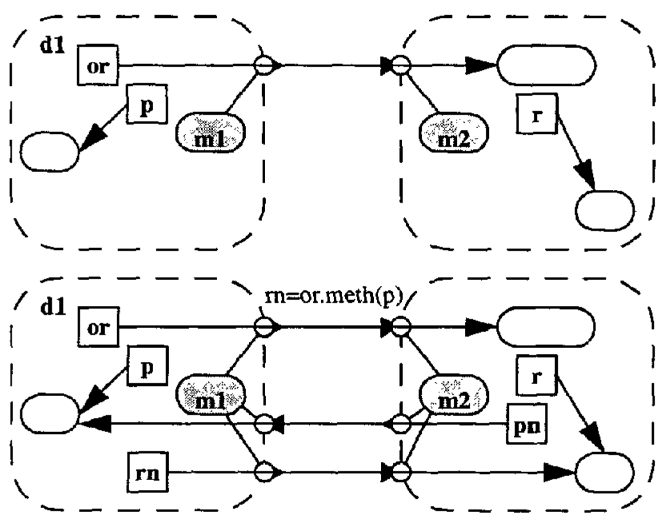

Fig. 7 Method invocation to another domain

3 The formal model

Let us now present a formal model for Security Meta Objects. As the general model for SMOs is quite complicated, we will just formalize one aspect of SMOs: the implementation of virtual object domains with SMOs.

Thus, our model only contains virtual object domains and the accessibility of object references from an object domain. Our model does not consider the state of objects. In fact, the formal model does not even consider objects, but only object references. Such an object reference has a source domain, that is the domain, from where the reference is accessible, and a target, that is the domain containing the target object of the reference. As the target object of the reference is not important, we do not include objects in our model. Only the domain, which contains the target object, is important. In this section we will only examine propagation of references. In the next section we will assign principal information to references and implement role-based principals.

3.1 Basic definitions

We define the following sets:

D = set of object domains

M = set of meta objects

A = set of meta attachments = $\{src, dst\} \times M$

So a meta attachment consists of the type of attachment (we have two types of attachments: *src* and *dst*) and a meta object. An example for such an attachment is: (*src*, *m1*).

An object reference consists of a list of attachments and a target object. As the target object is not important, we only look at the list of attachments. We define the set of object references as follows:

OR = set of object references = A^*

An example of an object reference is:

$((src, m1), (src, m2), (dst, m3))$.

The object domains are organized hierarchically (Fig.8).

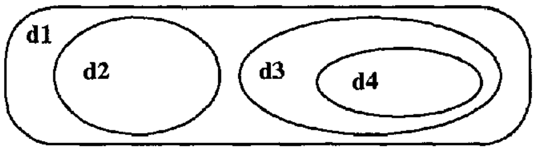

Fig. 8 Object domains

The function *parent* defines the encapsulating domain. For example, *parent(d2)* is *d1*.

$parent : D \cup \{\perp\} \rightarrow D \cup \{\perp\}$

The symbol $\perp$ means "undefined". The function *parent* is hierarchical: it has no cycles. So if we apply *parent* several times to a domain, the result is a top-level domain, which has no parent (that is, *parent* of the top-level domain is $\perp$).

$parent(\perp) = \perp$

$\forall d \in D: \exists n \in N: parent^{(n)}(d) = \perp$

There may be more than one top-level domain. The set of top-level domains can be computed as follows:

$D_{Top} = \{d \in D | parent(d) = \perp\}$

3.2 Domains

Now we have to define the relation of meta objects and object references to domains. Each meta object is assigned to a domain. It can only be used by that domain. We use the function *metadom* to define this assignment. Note, that top-level domains cannot contain SMOs.

$metadom : M \rightarrow D \backslash D_{Top}$ (defines the location of an SMO)

We need a similar function for object references. It assigns a domain to an object reference. The reference is valid in that domain, that is, only objects located in that domain may be able to use the object reference for method invocations. We call that function *dom*. Note, that *dom* is a function, which means that an object reference is only valid in one domain. For example, an object reference cannot be passed to another domain without being changed, as we will see later. The function *dom* either assigns a domain to an object reference, which means that the object reference may exist in the assigned domain, or it is undefined, which means that the object reference is invalid and cannot exist in our system. If *dom* assigns a domain to a reference, the reference may or may not exist, but if it exists, it is in that domain. The function *dom* is recursively defined by the following:

$dom : OR \cup \{\perp\} \rightarrow D \cup \{\perp\}$

$dom(\perp) = \perp,\ dom(\ (\ )\ ) = \perp$

$$dom((\mathrm{src},m), or) = \begin{cases} metadom(m) \text{ if } or = (\ ) \vee \\ \quad dom(or) = parent(metadom(m)) \\ \perp \text{ else} \end{cases}$$

$$dom((\mathrm{dst},m), or) = \begin{cases} parent(metadom(m)) \text{ if } or = (\ ) \vee \\ \quad dom(or) = metadom(m) \\ \perp \text{ else} \end{cases}$$

Additionally, we define the function "target" which defines the target domain, that is the domain, where the target object of an object reference is located.

$target : OR \cup \{\perp\} \rightarrow D \cup \{\perp\}$

$target(\perp) = \perp,\ target(\ (\ )\ ) = \perp$

$$target(or, (\mathrm{dst},m)) = \begin{cases} metadom(m) \\ \quad \text{if } dom(or, (\mathrm{dst},m)) \neq \perp \\ \perp \text{ else} \end{cases}$$

$$target(or, (\mathrm{src},m)) = \begin{cases} parent(metadom(m)) \\ \quad \text{if } dom(or, (\mathrm{src},m)) \neq \perp \\ \perp \text{ else} \end{cases}$$

Let us have an example for a valid object reference. In Fig. 9 we have four domains. The relations between the domains and meta objects are as shown:

for example, $parent(d2) = d1$ and $metadom(m3) = d4$.

The object reference $or = ((src,m1), (dst,m2), (dst, m3))$ is a valid object reference with $dom(or) = d2$ and $target(or) = d4$. That means, that or might exist in our system and if it exists, it is in domain *d2*.

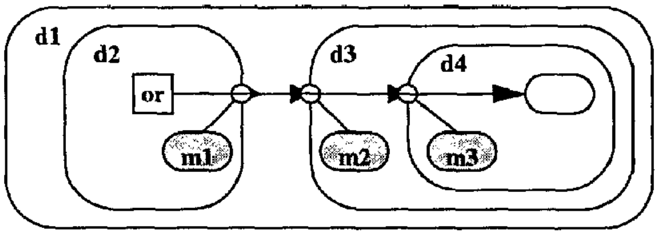

Fig. 9 Object reference example

3.3 Method invocations

The definitions in the previous section are static and do not define, which object references actually exist in our system. So let us now define a "current state" of our system and define how the state of the system can change.

The system itself is defined by the above-mentioned sets D and M and the functions *metadom* and *parent*. All other sets and functions (e.g., *dom* and *target*) result from these definitions.

The current state of our system is completely defined by the set of object references that currently exist. The current state can only contain valid object references:

$OR_{cur} \subseteq OR,\ \forall or \in OR_{cur}: dom(or) \neq \perp \vee or = (\ )$

If an object reference is in that set, the statically defined functions *dom* and *target* compute, where the target object is and in which domain the reference exists.

The system starts with a set of initial object references OR_{init}. The only way to change the state of the system is via method invocations. Thus, if we invoke a method, new object references may be created and added to the set of current object references.

The function *meth* executes a method invocation. We start with a current state OR_1 and apply a method invocation to that state. Afterwards we have a state OR_2. For the method invocation we need a target object reference (*or*), a parameter (*p*), and a return value (*r*). In our model we only look at method invocations with one parameter and one return value. Method invocations take no time, that is, during an invocation no other invocation can take place. That is no limitation, because all other cases can be implemented with our model. Passing several parameters can be implemented by calling a method several times. Invocations without parameter can be implemented by passing a dummy object. Methods that take a long time to execute, can be implemented by two method invocations: one at the beginning of the "real" method invocation and one at the end.

In this section we sometimes need the empty reference ( ). This is a reference inside one domain without SMOs attached ("local object reference"). We do not examine method invocations via such a reference, as they cannot propagate object references to other domains. But we examine method invocations with local object references as parameter or return value. The empty reference ( ) is the only valid reference with $dom(())=\perp$.

As our model only covers distribution of object references, we do not need information about the method to be invoked. We assume that the target object domain stores the object reference and that the calling object domain stores the return value.

$$OR_2 = meth(OR_1, or, p, r)$$
$$\text{with } OR_1, OR_2 \subseteq OR;\quad or \in OR_1;\quad p, r \in OR_1 \cup \{(\ )\}$$

Such a method invocation passes two object references: the parameter and the return value. The function *meth* just adds these two object references, which are computed by the functions *param* and *ret*, to the set of current object references in the system.

$$meth(OR_1, or, p, r) =$$
$$\begin{cases} OR_1 \cup \{param(or, p)\} \cup \{ret(or, r)\} \\ \quad \text{if } (dom(or) = dom(p) \vee p = (\ )) \wedge ret(or, r) \neq \perp \wedge \\ \quad (target(or) = dom(r) \vee r = (\ )) \wedge param(or, p) \neq \perp \\ OR_1 \ \text{else} \end{cases}$$

Now we have to define the functions *param* and *ret*.

$$param((\ ), p) = p$$
$$param(((src, m), or), p) = param(or, out_m(p))$$
$$param(((dst, m), or), p) = param(or, in_m(p))$$

$$ret((\ ), r) = r$$
$$ret((or, (src, m)), r) = ret(or, in_m(r))$$
$$ret((or, (dst, m)), r) = ret(or, out_m(r))$$

The functions *in* and *out* are provided by each meta object. They define how passed references are protected. These two functions must have specific properties, because the passed references have to fit into our domain model, that is, they have to be valid object references. We do not define these properties. Instead we present an implementation of *in* and *out* and prove that only valid object references are created by them.

$$in_m(p) = ((src, m), p)$$

$$out_m((\ )) = \begin{cases} \perp & \text{if } \neg allow_m((\ )) \\ (dst, m) & \text{else} \end{cases}$$

$$out_m((src, m_2), or) = \begin{cases} \perp & \text{if } \neg allow_m((src, m_2), or) \\ or & \text{else} \end{cases}$$

$$out_m((dst, m_2), or) = \begin{cases} \perp & \text{if } \neg allow_m((dst, m_2), or) \\ ((dst, m), (dst, m_2), or) & \text{else} \end{cases}$$

There is just the boolean function *allow* left, which is defined by each meta object. We will now prove, that with these functions *in* and *out* only valid object references can be created.

So we have to prove, that parameters and return values are valid in the domain where they are passed to:

(I) $dom(or) \neq \perp \wedge (dom(p) = dom(or) \vee p = (\ ))$
$\Rightarrow param(or, p) \in \{(\ ), \perp\} \vee$
$dom(param(or, p)) = target(or)$

(II) $target(or) \neq \perp \wedge (dom(r) = target(or) \vee r = (\ ))$
$\Rightarrow ret(or, r) \in \{(\ ), \perp\} \vee dom(ret(or, r)) = dom(or)$

We will now present the outline for the proof for the function *param* (I). If one of the called *out* functions returns $\perp$, the *param* function returns undefined and (I) is true. So we will not examine that case in our proof. In our proof we do not look at some special cases, for example, that the parameter (p) may become empty. As we just want to present an outline, we do not show the proof for this special case.

First we present the proof for object references with size 1. Afterwards we assume, that we already have the proof for object references with size n and proof (I) for object references of size n+1. So we assume,

$dom(or) \neq \perp \wedge (dom(p) = dom(or) \vee p = (\ ))$

For object references of size 1 there are two possibilities.

(1) $or = (src, m)$

(1a) $p = ((dst, m_2), p_2) \vee p = (\ )$

$dom(param(or, p)) = dom(out_m(p)) = dom((dst, m), p) =$
$parent(metadom(m)) = target(or)$

q.e.d.

(1b) $p = ((src, m_2), p_2)$

$dom(param(or, p)) = dom(out_m(p)) =$
$dom(p_2) = parent(metadom(m_2))$
$= parent(metadom(m)) = target(or)$

q.e.d.

(2) $or = (dst, m)$

$dom(param(or, p)) = dom(in_m(p)) = dom((src, m), p) =$
$metadom(m) = target(or)$

q.e.d.

Now we assume that we already have proved (I) for object references of size n.

So *or* has size n+1. There are again two possibilities:

(1) $or = ((src, m), or_2)$

(1a) $p = ((dst, m_2), p_2) \vee p = (\ )$

$dom(param(or, p)) = dom(param(or_2, ((dst, m), p)))$

As

$dom(or_2) = parent(dom(or)) =$
$parent(metadom(m)) = dom((dst, m), p)$

we use our assumption:

$dom(param(or_2, ((dst, m), p))) = target(or_2) = target(or)$

q.e.d.

(1b) $p = ((src, m_2), p_2)$

$dom(param(or, p)) = dom(param(or_2, p_2))$

As $dom(or_2) = dom(p_2)$

we use our assumption:

$dom(param(or_2, p_2)) = target(or_2) = target(or)$

q.e.d.

(2) $or = ((dst, m), or_2)$

$dom(param(or, p)) = dom(param(or_2, ((src, m), p)))$

As *or* is valid, we have:

$dom(or_2) = metadom(m) = dom((src, m), p)$

and from our assumption follows:

$dom(param(or_2, ((src, m), p))) = target(or_2) = target(or)$

q.e.d.

As we just proved, *param* only creates valid object references. The proof for *ret* is nearly the same, so we do not present it here.

3.4 Conclusion

Our model covers object references and their distribution. We showed that our meta objects keep the domains intact. Method invocations only propagate valid object references, that are references that consist of one SMO per domain boundary. In the next section we will show two example configurations.

4 Application of virtual domains

In the previous section we provided definitions for our model. Let us now examine two example configurations. To define a configuration, we have to determine the basic sets (*D*, *M*), the *allow* function of our meta objects and the initial state OR_{init}. We define this configuration and afterwards we prove specific properties of the system.

4.1 The hierarchical domain example

In many systems we have hierarchical system parts. For example, we have a printing system containing printer spoolers and printers. User applications have to interact with the printer spooler and the printer spooler has to interact with user applications (e.g., with the documents, that are to be printed) and with the printers. We now assume, that the printer spooler needs special principal information to authenticate to the printers. The problem with the authentication policies mentioned in the introduction is, that malicious user applications may trick the spooler to use its principal information. Fig. 10 shows an example. The user application (left side) may obtain a reference to a temporary spool file. It cannot access that file, as it has neither spooler nor printer principal information to authenticate. But the application may simply try to print that file by passing it to the spooler, which in turn accesses it or passes it to the printer. With domain-based principals that would succeed, as the spooler or the printer would accidently use its privileges (spooler principal information) to access that file. This is the object-oriented version of the Unix S-Bit problem.

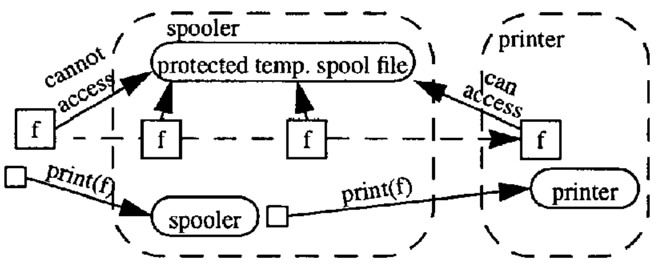

Fig. 10 Printer spooler with domain-based principals

As we will see, with hierarchical domains that kind of attack does not succeed.

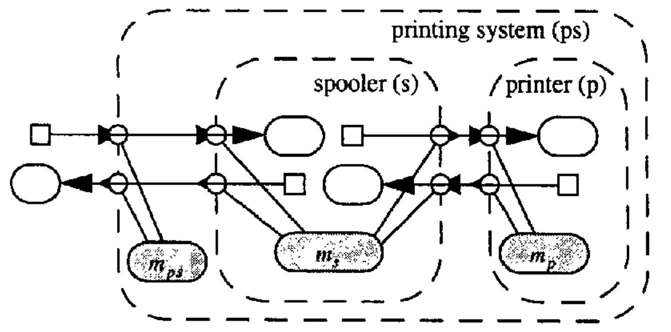

Fig. 11 The printer spooler with hierarchical domains

Our printing system consists of three domains (Fig. 11): the printing system d_{ps}, the spooler d_s and the printer d_p. Of course, there may be other domains, for example, there must be a parent domain of d_{ps} in the system. The domains d_s and d_p do not have any son domains (that are domains, that have d_s or d_p as parent) and the domain d_{ps} only has the two sons d_s and d_p. These domains only contain the SMOs m_s, m_p, m_{ps}, as shown in Fig. 11. Of course, there may be other SMOs in other domains of our system. For this configuration we do not need to restrict the distribution of object references, so *allow(or)=true* for all object references and all SMOs.

We define the authentication policy as follows:

- Method invocations via references containing (src, m_{ps}) are never authenticated.
- Method invocations via references not containing (src, m_{ps}) but containing (src, m_p) are authenticated with the printer principal information.
- Method invocations via references not containing (src, m_{ps}) but containing (src, m_s) are authenticated with the spooler principal information.
- All other method invocations are not authenticated or authenticated by different principal information.[3]

For this configuration we do not need any restriction for the set of initial references OR_{init}. We can now prove the following:

(1) Method invocations via object references to objects in other domains than d_s, d_p, d_{ps} are never authenticated with the printer or spooler principal information.

(2) Object references obtained via references from or to other domains than d_s, d_p, d_{ps} are never authenticated with the printer or spooler principal information.

(1) $target(or) \notin \{d_s, d_p, d_{ps}, \perp\}$

$\Rightarrow \exists or_1, or_2 : or = (or_1, (src, m_{ps}), or_2)$

$\vee \neg\exists or_3, or_4, m \in \{m_p, m_s\} : or = (or_3, (src, m), or_4)$

3. Note that we do not examine local calls. A method invocation inside the printer domain via a reference without SMOs attached to it might have to be authenticated, too. Otherwise the printer cannot call methods of its own objects.

Proof:
We assume the opposite:

$$target(or) \notin \{d_s, d_p, d_{ps}, \bot\}$$
$$\wedge \neg\exists or_1, or_2\colon or = (or_1, (src, m_{ps}), or_2)$$
$$\wedge \exists or_3, or_4\colon or = (or_3, (src, m), or_4)$$

If we look at the definition of *target* we find that:

$$target((src, m), or_4) = target(or) \notin \{d_s, d_p, d_{ps}, \bot\}$$

But as $target((src, m)) = ps \in \{d_s, d_p, d_{ps}, \bot\}$

there must be a position in $(src, m), or_4$ where the target changes:

$$\exists or_5, or_6, or_7\colon (or_5, or_6, or_7) = ((src, m), or_4) \wedge |or_6| = 1$$
$$\wedge\, target(or_5) \in \{d_s, d_p, d_{ps}, \bot\}$$
$$\wedge\, target((or_5, or_6)) \notin \{d_s, d_p, d_{ps}, \bot\}$$

From the definition of *target* we conclude $or_6 = (src, m_{ps})$, which contradicts our assumptions.

q.e.d.

For (2) we do not want to present the proof here. We will now just present the first half of (2) as formal expression:

$$dom(or) \notin \{d_s, d_p, d_{ps}, \bot\} \wedge target(or) \in \{d_s, d_p, d_{ps}\}$$
$$\Rightarrow \forall or_1, or_2\colon param(or, p) = (or_1, (src, m_{ps}), or_2)$$
$$\vee\, param(or, p) = \bot$$

So we know (1) that method invocations to objects outside of our domains are never authenticated, so we cannot accidently use our principal information for outside references.

And we know (2) that an outside application cannot trick our objects to use our principal information for references that are obtained from other domains. In Fig. 12 we have two references from the printer domain to the spooler. One reference was obtained directly from the spooler, the other one was obtained from a domain outside of the printing system. Calls via the internal reference are authenticated, calls via the other reference are not authenticated, as it contains the reference part (src, m_{ps}).

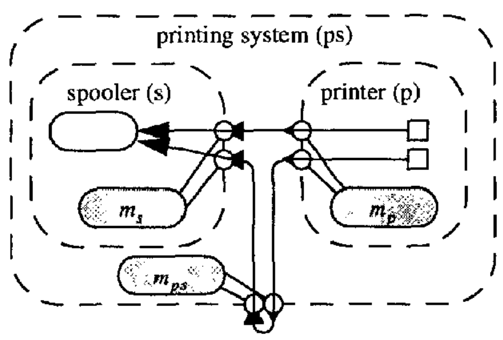

Fig. 12 A reference obtained from another domain

4.2 The disjunct interaction example

In many applications we have disjunct interaction with other application parts. That means, that our application part interacts with other application parts, but it does not initiate interaction between these other application parts (although these other parts themselves may initiate such interaction). Let us examine our printer spooler example: The printer spooler interacts with the printers and with user applications that might want to print, but the user applications do not have to interact with the printer directly. For example, if a user application prints a text object, the printer spooler must not pass the reference to the printer, otherwise the printer and the user application interact directly. We assume, that the printer only interacts with the spooler and that external applications only interact with the spooler. (Fig. 13).

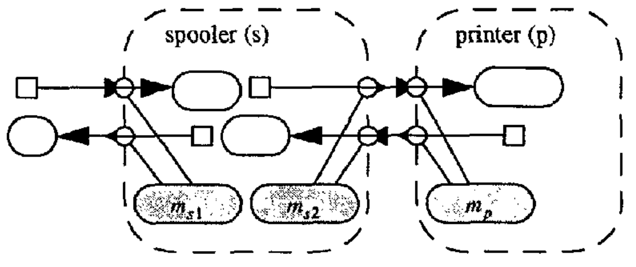

Fig. 13 The printer spooler with disjunct interaction

We need only two domains: the spooler domain d_s and the printer domain d_p. Initially the only references to the printer should be references from the spooler and these references (and only these references) should have m_p and m_{s2} attached to them:

$$(1)\ \forall m \in M\colon m = m_p \vee metadom(m) \neq d_p$$

$$(2)\ \forall or \in OR_{init}, or_{1..3}\colon or = (or_1, or_2, or_3)$$
$$\wedge\, or_2 \in \{(src, m_p), (dst, m_p), (src, m_{s2}), (dst, m_{s2})\}$$
$$\Rightarrow or = ((src, m_{s2}), (dst, m_p)) \vee or = ((src, m_p), (dst, m_{s2}))$$

For some situation condition (2) is too restrictive, as no other domains may have references to printer objects. Sometimes the printer references are obtained via the name server, so anybody can obtain such a reference, but the printer objects are protected with an access control list, so only the spooler may invoke methods. If we substitute (2) by this condition (which is a little more complicated), the following is also true.

Let us now define the function *allow*:

$$(m = m_{s1} \vee m = m_{s2}) \Rightarrow$$

$$allow_m(or) = \begin{cases} false \text{ if } \exists or_2\colon or = ((src, m2), or_2) \wedge m \neq m_2 \\ true \text{ else} \end{cases}$$

The *allow* functions of the other SMOs may be *true* for all object references.

With these definitions the printer spooler cannot accidently pass printer object references to applications and vice versa. From the definition of the method invocation follows, that the condition (2) is true for all possible states of the system. We can now define the principal policy as follows:

- If an object reference contains m_{s2}, the spooler principal information is used.
- In all other cases no principal information is used.

So only calls from the spooler to the printer via references obtained from the printer are authenticated. Applications cannot trick the spooler to use accidently its privileges to access, for example, a document passed to it.

Both configurations solve the object-oriented version of the Unix S-Bit problem.

5 Conclusion

We introduced a new security model based on Security Meta Objects, which allows fine-grained configuration of the principal information used for authentication on a per-reference basis. The projection to virtual object domains helps to define a global security policy based on these meta objects. We defined a formal model which describes the semantics of the Security Meta Objects. We are able to examine possible propagation and principal information usage with our formal model and we showed, that we are able to prove such properties for two very generic configurations. We proved, that these configurations do not suffer from the problem of unintentional use of privileges (Unix S-Bit problem) and we prevent accidental propagation of references. We built a Java prototype, to show, that our security model can be implemented.

6 References

[1] Benantar, M.; Blakley, B.; Nadalin, A.: Approach to object security in Distributed SOM, *IBM Systems Journal*, Vol. 35 No. 2, 1996, New York

[2] Dennis, J.B.; Van Horn, E.C.: "Programming Semantics for Multiprogrammed Computations", *Comm. of the ACM*, March 1966

[3] Ferraiolo, D.; Kuhn, R.: "Role-based access control", In: *15th NIST-NCSC National Computer Security Conference*, p. 554-563, Baltimore, Oct. 1992

[4] Flanagan, D.: *Java in a Nutshell*, O'Reilly & Associates, 1st edition, Feb 1996

[5] Lampson, B.: "A Note on the Confinement Problem", In: *Communications of the ACM 1973*, October, 1973

[6] Levy, H.: *Capability-Based Computer Systems*, Bedford, Mass.: Digital Press, 1984

[7] Maes, P.: *Computational Reflection*, Ph.D. Thesis, Technical Report 87-2, Artificial Intelligence Laboratory, Vrije Universiteit Brussel, 1987

[8] Mitchell, J. ; Gibbons, J.; Hamilton, G. et.al.: An Overview of the Spring System. *Proc. of the Compcon Spring 1994 (San Francisco)*, Los Alamitos: IEEE, 1994

[9] Neumann, C. B.: "Proxy-Based Authorization and accounting for Distributed Systems", In: *Proceedings of the 13th International Conference on Distributed Computing Systems*, Pittsburgh, May 1993

[10] OMG: *CORBA Security*, OMG Document Number 95-12-1, 1995Rashid, R.: "Threads of a New System". *UNIX Review*, 1986

[11] Riechmann, T.; Hauck, F. J.: "Meta objects for access control: extending capability-based security", In: *Proc. of the ACM New Security Paradigms Paradigms Workshop 1997*, Great Langdale, UK, Sept. 1997

[12] Riechmann, T.; Kleinöder, J.: "Meta objects for access control: Role-based Principals", In: *Proc. of the Third Australasian Conference on Information Security and Privacy*, Springer LNCS, Brisbane, Austalia, July 1998

[13] Sun Microsystems Comp. Corp.: *HotJava: The Security Story*, White Paper, 1995

[14] Sun Microsystems Comp. Corp.: The Java Language Environment, White Paper, 1995

[15] Sun Microsystems Comp. Corp.: *Java Security Architecture*, JDK 1.2 Beta Draft, 1997

[16] Tanenbaum, A. S.; Mullender, S. J.; van Renesse, R.: "Using sparse capabilities in a distributed operating system." *Proc. of the 6th Int. Conf. on Distr. Comp. Sys.*, pp. 558-563, Amsterdam, 1986

[17] Wallach, D. S.; Balfanz, D.; Dean, D.; Felten, E. W.: "Extensible Security Architecture for Java". *SOSP 1997*: p. 116-128, Oct. 1997, Saint-Malo, France

[18] Wulf, W.; Cohen, E.; Corwin, W.; Jones, A.; Levin, R.; Pierson, C.; Pollack, F.: "HYDRA: The Kernel of a Multiprocessor Operating System". *Communications of the ACM*, 1974

[19] Wang, C.; Wulf, W.; Kienzle, D.: A New Model of Security for Distributed Systems, In: *Proceedings of the 1996 ACM New Security Paradigms Workshop*, 1996