Bracket Capabilities for Distributed Systems Security

Mark Evered
School of Mathematical and Computer Sciences
University of New England
Armidale, 2351, NSW, Australia
markev@mcs.une.edu.au

Abstract

The per-method access control lists of standard middleware technologies allow only simple forms of access control to be expressed and enforced. Research systems based on capabilities provide a more secure mechanism but also fail to support more flexible security constraints such as parameter restrictions, logging and state-dependent access. They also fail to enforce a strict need-to-know view of a persistent object for each user. In this paper we present the concept of bracket capabilities as a new, simple security mechanism which fulfils these requirements. We discuss the reasons for integrating bracketing and view types at a fundamental level of the security mechanism. We demonstrate the use of the mechanism in a simple E-commerce environment to provide secure electronic cheques and describe a prototype implementation of the mechanism in middleware for secure, distributed Java applications.

Keywords: security, objects, distributed systems

1 Introduction

With the development of middleware technology, it has become standard practice to construct software systems as collections of heterogeneous distributed components. Such systems are increasingly used for database integration, decision support systems, electronic commerce and many other applications. In general, the information stored within the components of these systems is sensitive and requires some form of access control. This is particularly important as the internet is increasingly used as the basis for distributed systems and as the threat from hackers and malicious software continues to grow. As well as protecting a system from these external threats, the access control mechanism also must ensure that each user with access to the system only uses the information exactly as required for their role within the organization.

Despite the sensitivity of the data and the growing threat, relatively little attention has been paid to security constraints in middleware development. OMG's Corba (Blakley 2000), Microsoft's COM (Eddon 1999) and Sun's Java RMI (Sun 1999) all include a form of access control list (ACL) but these are add-on features which remain limited and inflexible. Much attention has been given to encryption techniques but, while encryption is certainly very important, it protects only the communication and authentication in the system. It provides only the *basis* for a secure access control mechanism.

In this paper we introduce the concept of bracket capabilities, a new, simple security mechanism which can be integrated into middleware technology at a fundamental and therefore maximally secure level. The mechanism provides for a finer-grained access control than previously proposed mechanisms as well as supporting other security factors such as the logging of accesses and access attempts.

We begin in the next section by discussing semantic protection and role-based access as supported in standard commercial systems and research systems. In the following section we demonstrate the need to extend the framework of role-based access control. In section 4 we describe the concept of bracket capabilities and in section 5 we show how this mechanism can be used in an example E-commerce application to provide secure electronic transfer of funds. In section 6 we briefly describe a prototype system using bracket capabilities as the basis for access to distributed components. Finally, in section 7, we give an overview of and comparison with other related research.

2 Semantic Role-based Access Control

The components of a distributed system can be viewed as objects according to the object-oriented paradigm with each object containing persistent data hidden by encapsulation and accessible via interface methods[1]. One advantage of an object-oriented approach is that the security can be based on the interface methods of an object. This provides a fine-grained *semantic protection* (Evered 2000) in contrast to the course-grained read/write protection of traditional files and databases. This means that the access rights can be based on meaningful, high-level operations associated with the object in the real world. So, for example, we may define a persistent object which implements a bank accounts file with

[1] This may, of course, simply be a façade around a legacy component such as a relational database.

methods for creating a new account, depositing some amount in one of the accounts, checking the balance in one of the accounts etc. (Fig. 1).

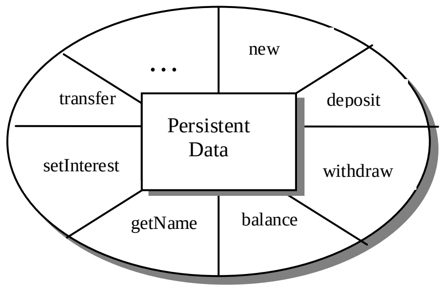

Fig 1: A bank accounts object

This corresponds to the Java interface definition:

```
interface Accounts {
  void new(Key newKey, String name);
  void deposit(Key key, Currency amount);
  void withdraw(Key key, Currency amount)
         throws insufficientFunds;
  Currency balance(Key key);
  String getName(Key key);
  void setInterest(Percent rate);
  void transfer(Key fromKey,
                Key toKey,
                Currency amount)
         throws insufficientFunds;
}
```

The access rights can then be granted on the basis of the roles of the users within the organization. A bank teller may have access to the deposit and withdraw methods whereas the bank manager may also have access to the method for setting the interest rate. This idea goes back as far as Jones' and Liskov's (1978) suggestion of a static type-based constraint mechanism and has been adopted in contemporary middleware mechanisms. Java's RMI can be used in conjunction with a standard package for access control lists but it is up to the programmer to explicitly establish the relationship between the 'rights' and the allowed method calls. The COM security mechanism does this automatically, offering 'per-method access control lists' which record, for each method, a list of users allowed to invoke that method (Eddon 1999).

The alternative to a protection mechanism based on access control lists is an object-based capability protection mechanism. A capability for an object is simultaneously an identifier for the object and a list of allowed methods. The possession of the capability represents the right to call those methods on that object. This has the added security advantage that the naming of objects is unified with the protection mechanism so that someone who has no access to an object does not even know of the object's existence (Wilkes and Needham 1979). Among the disadvantages of capabilities are that they are held by the user and not with the object and so must themselves be protected in some way from forgery and tampering. The Monads system (Rosenberg and Abramson 1985) supports object-based capabilities for distributed systems but assumes a homogeneous network and a special operating system kernel. The author has proposed a middleware technology based on a form of sparse capabilities (Evered 2000). Brose (1999) has proposed a language-based extension to the Corba security model in which the allowed 'views' for each user are defined in terms of the methods of an object type.

3 Extending Role-based Security

The access control mechanisms described in the previous section all limit the access to an object by returning an error message if certain methods are invoked. However, this is not the only kind of access restriction which is possible or useful in controlling access to a persistent object.

In fact, even in terms of per-method access control, the above mechanisms are not ideal. With each of these mechanisms, all the methods of the object are still known to all the users even if they cannot be called. Ideally, in a need-to-know security environment, someone who is not allowed to invoke a method should not know of the existence of that method, just as someone without a capability does not even know of the existence of the object[2]. So, for example, if an ATM machine only requires access to the `withdraw` and `balance` methods of an `Accounts` object, then the ATM software should ideally see the object as if it had the type:

```
interface ATMAccounts {
  void withdraw(Key key, Currency amount)
         throws insufficientFunds;
  Currency balance(Key key);
}
```

instead of the type `Accounts`. This type effectively defines the *view* that the software has of the underlying object.

As an example of a form of access control not supported by a simple per-method approach, we now consider a role which is not within the organization (the bank) in which the `Accounts` object is held, but may be part of a broader distributed system of which that object is a part. What access to an `Accounts`

[2] As well as increasing security, it also simplifies the use of the object by the user if only the relevant methods are visible.

object should be given to the owner of an individual account? As well as restricting access to the methods `balance`, `getName` and `transfer`, we must also ensure that only the right account is being accessed. This means that the `Key` parameter of `balance` and `getName` and the `fromKey` parameter of `transfer` must be restricted to a particular value.

This requires a mechanism which grants access rights to the user in such a way that an error is returned if the wrong account number is specified in the call. Or, better still, in terms of views, we would like the account owner to view the object as if it had the type:

```
interface MyAccount {
  Currency balance();
  String getName();
  void transfer(Key toKey,
                Currency amount)
         throws insufficientFunds;
}
```

where it is implicit that the appropriate account is to be accessed.

A number of further kinds of access control are also useful for flexibly specifying the security constraints associated with different roles in a system. In (Evered 2001), the author has described a formalism in which five basic type operators are used in combination to specify security constraints in terms of a *view type*. The operators modify the methods, parameters and semantics of the type through which a user accesses the underlying persistent object. The operators provide for:

- specifying that some methods should return an access violation error
- specifying that an access violation error should be returned for parameter values other than a specified value
- restricting the view type to exactly the allowed methods and parameters
- enhancing the semantics of the object type with logging of accesses and access attempts
- specifying a state-dependent rule such as 'access only at specified times' or 'access allowed only once'

The view type effectively gives the user the illusion of accessing an object of that type when in fact it is just a representation of a restricted access to an object of the original type. So, for example, the type `MyAccount` would appear to be the type of an individual bank account object but a call to this object would actually be a call to an `Accounts` object with the account number automatically inserted. The `MyAccount` object can be seen as a *virtual object*[3].

It is this extended concept of security constraints for which bracket capabilities have been developed.

4 Bracket Capabilities

Our mechanism is based on object capabilities rather than ACLs because, as mentioned above, security is enhanced and simplified by the unification of object naming with the protection mechanism. Capability systems have traditionally had three major problems. Firstly, the capabilities themselves must be protected, secondly, revocation of access rights is more difficult than with ACLs and, thirdly, garbage collection can be more difficult. In the context of persistent objects in a distributed system, we neglect the third problem since we want objects to be explicitly deleted.

A number of possible alternatives have been suggested for protecting capabilities[4]. These include special architectures (Rosenberg and Abramson 1985), encryption (Mullender and Tanenbaum 1986) and sparse (or password) capabilities (Anderson, Pose and Wallace 1986). We base our mechanism on sparse capabilities since these require no special architecture or costly encryption algorithms and also because they alleviate the revocation problem. A sparse capability generally consists of an object identifier (for locating the object) together with a large (un-guessable) random number. The access rights associated with the capability (that is, with that particular random number) are not stored in the capability itself but with the object being accessed, so can easily be modified or revoked without access to the capability.

Our capabilities differ from traditional sparse capabilities in that they contain not an object identifier, but an identifier for a (capability) server which knows the location of the object. This indirection allows for object migration. When a persistent object is created, a capability for the object is created and registered with the capability server. The capability server also contains other information about accessing the object, including the type of the object as seen by the possessor of that capability.

To gain access to an object, the object is 'opened' using a capability. For example:

```
Accounts acc= c.open();
```

[3] Such an object may actually exist within the `Accounts` object or it may not exist if, for example, the `Accounts` object is just an object-oriented façade around a relational database.

[4] We mean here protection from tampering by the possessor of the capability. It is also necessary to protect the capability from being copied or modified by third parties when being sent across the network. We assume that this done via message encryption.

where **`c`** is a variable of type **`Capability`**.

So far, our mechanism is not much different from other mechanisms based on sparse capabilities. The main difference is seen when the possessor of a capability wishes to grant a more restricted view of the object to other users in the system. This is done by a call to the **`refine`** method. Each persistent object, as well as implementing an interface such as **`Accounts`** also implements the standard interface **`Persistent`** which includes methods such as **`deleteObject`**, **`deleteCapability`** and **`refine`**. The **`refine`** method is called as:

```
x= c.open();
Capability cref=x.refine(interface, class);
```

where **`interface`** denotes the type with which the persistent object is to be viewed using the capability **`cref`** and **`class`** denotes the class of an object *through which calls to the persistent object will pass* when invoked via **`cref`**. This class must implement the type **`interface`** and must have a constructor with a single parameter of the view type associated with the capability **`c`**. The result of the **`refine`** call is depicted in Fig. 2.

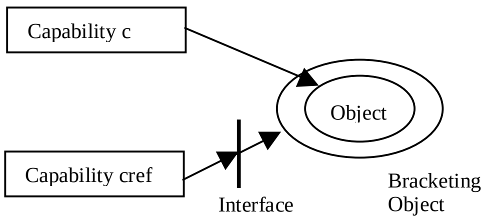

Fig 2: The result of the 'refine' operation

It can be seen that calls using the capability **`cref`** are directed through a kind of proxy or *bracketing* object of class **`class`**. This bracketing object is stored together with the persistent object in the same way that access rights are stored with the object for traditional sparse capabilities.

A copy of **`cref`** can be given to the users who are to have this kind of access. Of course a possessor of the capability **`cref`** may wish to create an even more restricted view of the object. This would result in the situation shown in Fig 3., in which a second bracketing object brackets the first.

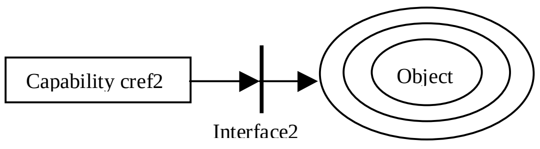

Fig 3: The result of a further 'refine' operation ('c' and 'cref' not shown)

The power of this mechanism stems from the ability to specify arbitrary classes as brackets (subject to the rules just described). In a given security environment, however, the allowed classes may be restricted to a particular approved set. Traditional object-based capabilities or ACLs in which access is restricted to particular methods can easily be simulated with our mechanism. The ATM access described in section 3 could be achieved as:

```
acc= objc.open();
Capability atmCap=
   acc.refine(ATMAccounts, ATMBracket);
```

where ATMAccounts is defined as in section 3 and ATMBracket is defined as:

```
class ATMBracket
            implements ATMAccounts {
  private Accounts underlying;

  public ATMBracket(Accounts acc) {
    underlying=acc;
  }

  public void withdraw(Key key,
                       Currency amount)
             throws insufficientFunds {
    underlying.withdraw(key, amount);
  }

  public Currency balance(Key key) {
    return underlying.balance(key);
  }
}
```

Clearly, other bracket classes can be used to implement the full range of security constraints described in section 3, including restrictions on parameters, logging of method calls and constraints based on time, number of accesses etc. It should also be noted that the bracketing class may have more methods than those available in the view given by the interface type. These extra methods can be used by the creator of the capability for monitoring or altering the bracketing such as to inspect logging information or to revoke or alter access constraints.

Before we turn to a more extensive example, one possible criticism of the mechanism must be addressed. It is possible to simulate this mechanism by using a traditional object-based capability or ACL mechanism. The bracketing object could be created as a persistent object in the same way as the object being protected. The user with the restricted view would be given a capability for this new object which would contain a reference to the original object and pass on the calls in the same way as in our mechanism. What is the advantage in making the bracketing integral to the protection mechanism? Apart from being easier to use, there are three important reasons: information, efficiency and control.

1. A possible problem with discretionary access control is that the administrator of a system can lose track of what different kinds of access exist and how they are related. If the bracketing is a part of the security mechanism, then information about all the capabilities to the object and the nesting of brackets associated with each capability can be stored centrally with the object and inspected by the administrator or owner[5].
2. A persistent object in a distributed system is not an ordinary object[6]. There is generally a considerable overhead involved in invoking a persistent object, for example in inter-process communication. If the bracketing objects are stored as persistent objects in the same way as the protected object then this overhead will be necessary for each step in the call sequence. If, however, the bracket objects are stored centrally with the protected object, they can be handled together by the persistence mechanism and the calls between bracketing objects and from the bracketing objects to the protected object can be ordinary method calls.
3. As indicated above, although the mechanism itself allows arbitrary bracketing classes, in a particular security environment, we may want to limit the set of classes that can be used. While still allowing users to create more restricted views, we may want to specify what kind of restriction they can impose. This is only possible if the bracketing is part of the security mechanism. In fact, since the **`refine`** call is itself just a method call to the persistent object, we can use the mechanism itself to specify that, for some user, it can only be invoked with certain values for the **`class`** parameter.

[5] While capabilities are, in the first instance, a form of discretionary access control it is nevertheless possible to implement mandatory access control policies such as (Bell and LaPadula 1973) by using them as a basis. This goes beyond the scope of the current paper but has been discussed in (Keedy and Vosseberg 1992).

[6] We mean here persistence in the sense of concurrent sharing of a persistent object by different programs and processes, not the kind of *light-weight persistence* supported in Java.

5 Example: Secure Electronic Cheques for E-Commerce

In this section we demonstrate the potential of bracket capabilities by developing a simple system for electronic funds management. At the centre of the system is an object of the type **`Accounts`** as described above. After creating this object, we have a capability **`objc`** for unrestricted access.

The first level of security is the logging of all accesses and access attempts to the object. We can achieve this by creating a new capability as:

```
acc = objc.open();
logc =
  acc.refine(Accounts, LoggedAccounts);
```

where **`LoggedAccounts`** is a class for an object which records the parameters, time and state of the relevant account and passes the call on to the **`Accounts`** object. Note that, in this case, the interface to the object remains unchanged. Only copies of the **`logc`** capability and not the original **`objc`** capability will be further distributed in the system.

Next we can create a capability for an individual bank account holder. For account number 12345, this can be achieved with:

```
acc = logc.open();
accountc =
  acc.refine(MyAccount, Account12345);
```

where MyAccount is defined as in section 3 and the class Account12345 is defined as:

```
class Account12345 implements MyAccount {
  private Accounts underlying;

  public Accounts12345(Accounts acc) {
    underlying=acc;
  }

  public Currency balance() {
    return underlying.balance(12345);
  }

  public String getName() {
    return underlying.getName(12345);
  }
```

```
    public void transfer(Key toKey,
                         Currency amount)
             throws insufficientFunds {
    underlying.transfer(12345, toKey,
                        amount);
    }
  }
```

The possessor of a copy of the **accountc** capability appears to have access to an object of type **MyAccount** but is actually accessing the **Accounts** object (through the logging object).

Finally, the account owner may wish to provide a restricted access to his/her account so that another account owner can transfer a certain amount out of the account as a payment. The capability for such an access is in fact a secure electronic cheque. As well as fixing the amount, we must ensure that this capability can only be used once. We can achieve this as:

```
  MyAccount a= accountc.open();
  chequec = a.refine(Cheque, ChequeXyz);
```

where **Cheque** is defined as:

```
  interface Cheque {
    void transfer(Key toKey)
           throws insufficientFunds;
  }
```

and **ChequeXyz** is defined as:

```
  class ChequeXyz implements Cheque {
    private MyAccount underlying;

    public ChequeXyz(MyAccount acc) {
      underlying=acc;
    }

    public void transfer(Key toKey)
        throws insufficientFunds {
      underlying.transfer(toKey, 100);
      underlying.deleteCapability();
  }
```

where, in this case, the cheque is for $100. The call **deleteCapability** removes all access for the capability with which the object was called and so prevents the cheque from being used again.

Clearly, the destination account could also be fixed if desired, or, by using a class such as **OncePerMonth** instead of **ChequeXyz**, the same mechanism could be used to create a capability for regular transfers rather than a once-off payment.

Finally, it should be noted that, in a particular implementation such as described in the next section, the classes **Account12345** and **ChequeXyz** need not written by hand but are generated automatically from templates using standard utilities.

6 Implementation

In this section we briefly describe a system for constructing distributed Java applications based on the mechanism of bracket capabilities. The system consists of a middleware mechanism for connecting distributed persistent objects and a set of utilities for constructing the components of an application and specifying the security constraints.

The two main features of the system are that:

- identifiers for persistent objects are 128-bit bracket capabilities
- it offers flexibility and transparency in the mechanisms used for both distribution and persistence

As described above, the capability essentially specifies through which sequence of bracketing code the object will be accessed and as what type of object the object is viewed by a possessor of that capability. Each capability consists of a 36-bit capability-server identifier (CSID) and a 92-bit password. The CSID identifies a server object which is guaranteed to know the location of the persistent object[7]. The first 4 bits of the CSID specify the protocol to be used to contact the server. Currently, only one protocol is supported, with the remaining 32-bits of the CSID specifying the IP number of the server. The **open** operation on a capability leads to a look-up operation on the server, using the lower 46-bits of the password in the capability as an index. Only these 46-bits are stored in the capability server rather than the whole 92-bits since otherwise the server would essentially own a copy of the capability and therefore have all the associated access rights itself.

Two mechanisms for transparent distribution and two mechanisms for transparent persistence are currently supported. Remote method calls can be implemented either via Java's RMI or via a web-based CGI mechanism. The mechanism to be used in communicating with a particular object is provided to the middleware by the capability server when the object is opened and remains unknown to the user of the capability. Persistence is implemented either via Java's built-in light-weight persistence (using inter-process communication to achieve sharing) or, for limited object types, as a wrapper around a Postgres database.

As described in section 4, the parameters to the **refine** call are an interface and a class. In most

[7] For robustness and efficiency, the location may be cached elsewhere as well.

object-oriented languages these are not first-class objects so a language-dependent realisation of these must be used. In the Java implementation, the Java reflection mechanism is used for this purpose as well as for the dynamic binding of communication mechanisms and view types.

As well as tools for the generation of communication stubs, persistence wrappers etc., the system also contains utilities which allow the generation of the most common kinds of bracketing classes. So, for example, given the interfaces **`Accounts`** and **`MyAccount`**, the class **`Account12345`** can be generated by using the utility **`bracket`** as:

```
bracket Account12345 MyAccount Accounts
        key=12345 fromKey=12345
```

7 Related Work

As mentioned above, Corba (Mowbray and Zahavi 1995) and COM (Eddon 1999) both include the possibility of a per-method, role-based access control list for limiting the access of users to objects. In some cases, fixed forms of rule-based access, such as access at certain times of day, are supported. These correspond only to simple, special cases of access control. No direct equivalent of the logging and parameter restrictions as required for the above E-commerce example are supported. No direct equivalent of a restricted view type is supported for hiding the existence of unallowed methods and parameters from the users. In both of these middleware technologies, the use of ACLs instead of capabilities makes the security mechanism an add-on feature rather than fundamental and detracts from the security.

Object capabilities have been used in a number of research systems, most notably the Monads system (Rosenberg and Abramson 1985) but these capabilities require architectural support (or at least a special operating system kernel) and so are not appropriate for heterogeneous networks. In a previous project, the author has developed a capability-based mechanism for heterogeneous distributed applications (Evered 2000). Like the Monads system and the ACL approaches of Corba and COM, however, this supported only simple per-method access control.

The concept of 'bracketing' for applying access constraints has been suggested both as a programming language construct (Keedy et al. 2000) and as a form of 'design pattern' (Gamma et al. 1995). The suggested programming language approach is interesting in supporting the *reuse* of the bracketing code but it does not allow modification of the interface to the underlying object and, being integrated into the type system of the language, it is a static mechanism.

One use of the *proxy* design pattern is as a protection (or access) proxy. In this case, the interface is identical to the underlying object. The proxy decides whether the access can proceed and returns an error if it should not. Simple per-method access control can be realised by this kind of protection proxy. A proxy object which maintains a log of access attempts could be seen as a kind of *decorator* pattern (though this is most often seen as a graphical decoration) since it maintains the original functionality while enhancing it with a logging and reporting functionality. Bracketing objects which modify the interface offered to a client cannot be seen as strict proxies. They can be seen as special cases of the adapter pattern but whereas an adapter is usually used to provide the view the client would *like* to have of the underlying object, in these cases the adapter is providing the view the client is *allowed* to have.

The concept of providing a user with a restricted view of persistent data is reminiscent of database systems. Database views are attribute-oriented and not method-oriented, however, and do not support the flexible kinds of access control demonstrated in our example. This is true even for object-oriented databases (Mishra and Eich 1994). Brose (1999) describes a 'view-based' mechanism for Corba but this is again simply a kind of language-based per-method access control. It does not hide the unallowed methods and does not support views involving parameter restrictions.

8 Conclusion

In applications based on distributed objects, the access control can be expressed in terms of the abstract interface operations of an object rather than simple read or write access to the data. Ideally, this access will be limited to exactly the access required for a user to fulfil their role within the system. The per-method access control lists of standard middleware technologies allows only simple forms of such access control to be expressed and enforced. Research systems based on capabilities provide a more secure mechanism but also fail to support more flexible security constraints such as parameter restrictions, logging and state-dependent access. They also fail to enforce a strict need-to-know view of a persistent object for each user.

We have presented the concept of bracket capabilities as a new, simple security mechanism which fulfils these requirements. A bracket capability is a form of sparse capability that determines, for a user possessing the capability, through what kind of bracketing object and through what interface (ie. as what type of object) the protected object can be accessed. This allows a very high degree of security and flexibility. We have discussed three important reasons for integrating the bracketing into the security mechanism. The concept of a capability server allows for object migration and flexible realisation of remote method calls.

The mechanism can be used with arbitrary bracketing classes or, in a particular system or security environment, with a fixed set of allowed bracket classes and view types. This can be achieved by using the mechanism itself to restrict the allowed values of the **`class`** and **`interface`** parameters of the **`refine`** call.

Finally, we have demonstrated the use of the mechanism in a simple E-commerce environment to provide secure electronic cheques and have described a prototype implementation of the mechanism in middleware for secure, distributed Java applications.

9 References

ANDERSON, M., POSE, R.D., WALLACE, C.S. (1986) A Password-Capability System, *The Computer Journal*, 29,1, pp.1-8.

ATKINSON, M.P., JORDAN, M.J., DAYNES, L. SPENCE, S. (1996) Design Issues for Persistent Java: a Type-Safe Object-Oriented, Orthogonally Persistent System, *Proc. 7th Intl. Workshop Persistent Object Systems*, Cape May.

BELL, D.E., LAPADULA, L.J. (1973) Secure Computer Systems: Mathematical Foundations, Mitre Corp., Bedford, Ma., Technical Report, ESD-TR-73-278.

BLAKLEY, B., BLAKLEY, R., SOLEY, R.M., (2000) *CORBA Security: An Introduction to Safe Computing with Objects*, Addison-Wesley.

BROSE, G., (1999) A View-Based Access Control Model for CORBA, in: Jan Vitek, Christian Jensen (eds.), *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, LNCS 1603, Springer.

EDDON, G., (1999) *The COM+ Security Model Gets You Out of the Security Programming Business*, Microsoft Systems Journal, Nov. 1999.

EVERED, M., (2000) *A Two-Level Architecture for Semantic Protection of Persistent Distributed Objects*, Proc. Intl. Conf. on Software Methods and Tools, Wollongong.

EVERED, M., (2001) *Type Operators for Role-based Object Security*, 3rd IFIP/ACM Intl. Conf. on Distributed Systems Platforms - Middleware (WiP), Heidelberg.

FAIRTHORNE, B. et al., (1994) eds. *Security White Paper*, OMG TC Document.

FONDÓN, M.A. et al. (1998) *Merging Capabilities with the Object Model of an Object-Oriented Abstract Machine*, 12th European Conference on Object-Oriented Programming, Brussels

GAMMA, E. et al., (1995) *Design Patterns*, Addison-Wesley.

GOSLING, J., JOY, B. AND STEELE, G., (1996) *The Java Language Specification*, Reading, MA: Addison-Wesley.

HARRISON, M.A., RUZZO, W.L., ULLMAN, J.D., (1976) *Protection in Operating Systems*, Communications of the ACM, 19, 8.

JOSHI, J.B.D. ET AL., (2001) *Security Models for Web-based Applications*, Communications of the ACM, 44, 2.

JONES, A. AND LISKOV, B. (1978) *A language extension for expressing constraints on data access*. Communications of the ACM, 21(5):358-367.

KEEDY, J.L. AND VOSSEBERG, K. (1992) *Persistent Protected Modules and Persistent Processes as a Base for a More Secure Operating System*, Proc. 25th Hawaii International Conference on System Sciences, IEEE Computer Society Press, S. 747-756.

KEEDY, J.L., ET AL., (2000) *Software Reuse in an Object Oriented Framework: Distinguishing Types from Implementations and Objects from Attributes*, Proc. Sixth International Conference on Software Reuse, Vienna.

MISHRA, P. AND EICH, M.H., (1994) *Taxonomy of views in OODBs*, Proc. ACM Computer Science Conference.

MORRISON, R., BROWN, A.L., CARRICK, C. ET AL. (1989) *The Napier Type System*, Proc. 3rd Intl. Workshop on Persistent Object Systems, Newcastle.

MULLENDER, S.J., TANENBAUM, A.S. (1986) *The Design of a Capability-Based Distributed Operating System*, Computer Journal, 29,4, pp.289-299.

MOWBRAY, T.J. & ZAHAVI, R. (1995) *The Essential Corba - Systems Integration Using Distributed Objects*, Wiley, New York.

ROSENBERG, J., ABRAMSON, D. A. (1985) *The MONADS Architecture: Motivation and Implementation*, Proc. First Pan Pacific Computer Conference, p. 4/10-4/23.

SUN MICROSYSTEMS INC. (1999) Java Remote Method Invocation Tutorial, http://java.sun.com/docs/books/tutorial/rmi/index.html

WILKES, M.V., NEEDHAM, R.M. (1979) *The Cambridge CAP Computer and its Operating System*, North Holland.