JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2008**

Vol. 7, No. 4, May-June 2008

Security and Protection in Timor Programs

J. Leslie Keedy, Klaus Espenlaub, Christian Heinlein and Gisela Menger, University of Ulm, Germany

Abstract

Timor offers a wide variety of security and protection features which are not available in other programming languages. A basic capability mechanism allows access to the methods associated with a Timor persistent file or a local internal object to be selectively controlled. A general qualifier mechanism allows arbitrary checks to be programmed both before methods are invoked and when they attempt to invoke other methods. This enables mechanisms such as access control lists, capability revocation lists and password checking to be applied to some or all the method invocations on an object, and can also be used for example to encrypt parameters. Since such qualifiers can be arbitrarily programmed they can also easily provide rule-based access controls. Mechanisms are also provided to allow objects with which users entrust their information to be confined. Finally, it is possible to program authentication objects which can provide arbitrary checks to establish the identities of users as they log in.

1 INTRODUCTION

A key issue in any persistent system is the protection from misuse of information stored in persistent objects. This is especially true for Timor, as one purpose of the language is to provide an environment which allows database systems to be supported in an object oriented and component oriented manner. Hence Timor provides a number of protection mechanisms which are not present in other programming language designs.

Timor programs store information in *persistent file objects*. These are persistent objects which in principle correspond to files in conventional systems, except that they are defined in accordance with the information hiding principle to be accessible only via methods associated with the type of the file. Internally such a file can contain *local objects*, which correspond loosely to objects in a conventional object oriented program, and are also defined in accordance with the information hiding principle.

Because a file object is accessed via its methods it can serve not only as a repository of information, but also as a program or as a subroutine library. Hence Timor has no special concepts of program, subroutine library or similar. The Timor concept of persistent objects is described in detail in [7].

J. Leslie Keedy, Klaus Espenlaub, Christian Heinlein and Gisela Menger: “Security and Protection in Timor Programs”, in Journal of Object Technology, Vol. 7, No. 4, May-June 2008, pp 123-138 http://www.jot.fm/issues/issue_2008_05/article3/

Activity in a Timor environment occurs via threads, which are stored in *persistent process objects* [8]. A persistent process object, which may have multiple threads that can be dynamically created and deleted, can persist between logouts and logins of its associated user, as will be described in more detail below. Threads are organised according to the in-process (procedure oriented) model [14, 16]. In the Timor environment this means that an active thread of a process can invoke methods of different file objects as procedures, with parameters passed and values returned on the thread's stack. Inter-file linkage is stored in the process object with which the thread is associated. There is no process or thread implicitly associated with a file object. File objects located at remote computers can be accessed in the same way as file objects on the local node, via invocations of their methods. The semantics of such calls are identical and the programmer need not be aware of the fact that the file object which he is accessing happens to be on a remote computer. The *SPEEDOS* emulator, which forms part of a Timor run-time environment, is responsible for hiding the differences. This emulates certain aspects of a *SPEEDOS* operating system environment [1] for Timor programs which are executed on systems with conventional operating systems.

This paper describes the protection mechanisms available for Timor programs. These mechanisms are uniformly available for protecting file objects and the local objects which can exist within them. Section 2 provides a general discussion of protection principles, in particular the concepts of access control, capabilities, access control lists, confinement of information; it concludes by emphasizing the importance of being able to identify subjects and objects uniquely. In section 3 the kinds of unique identifiers available to Timor programmers are listed and discussed. Section 4 presents the Timor capability concept. Section 5 briefly outlines the idea of qualifiers, showing how they can be used to implement a variety of protection mechanisms, including access control lists and the revocation of capability access rights. Section 6 describes various mechanisms for solving the confinement problem. In section 7 some issues related to the control of untrusted qualifiers are discussed. Section 8 describes how users of Timor programs can safely authenticate themselves to their persistent processes. The paper concludes with indications of related work in section 9 and a short conclusion in section 10.

2 SOME PROTECTION PRINCIPLES

Control over access to information can be viewed technically as falling into two basic categories:

- control via access rights (classically defined in Lampson's access control matrix [13] but possibly also defined in terms of access rules, cf. [2]), and
- the confinement of information.

Although the first case is relatively well understood, existing operating systems are frequently inadequate in their provision of such access controls. The second case is not so well understood and creates very serious problems in existing systems. Programming languages frequently provide no mechanisms for controlling access in either form.

Direct Access Controls: Lampson's Matrix

Lampson's matrix provides a neutral representation of access rights control defined in terms of a set of subjects (e.g. users, processes, programs), a set of objects (e.g. files, programs, memory segments) and for each subject-object pair a set of access rights. Because in a typical multi-user operating system or database system

- there may be very many subjects and objects,
- most users have no access to most objects, and
- subjects and objects are dynamically created and deleted quite frequently

such a matrix is typically very sparse and frequently changing, so that it does not make sense to implement an abstract access matrix as a two dimensional array. Two fundamental implementation possibilities arise, representing two lower level abstractions of the matrix.

A *capability list* (CL) is an abstract implementation whereby a set of <object, access rights> pairs is associated with each *subject*. If a subject has no access according to the access matrix, his CL does not need to contain an entry for the object in question. Each entry in a CL is known as a *capability*.

Alternatively an *access control list* (ACL) is an abstract implementation whereby for each *object* there is a set of <subject, access rights> pairs. Where a subject has no access to a particular object in the matrix, the ACL does not need to contain an entry for the subject in question.

Each of these abstractions has an analogy in real life. A capability list has similar properties to a bunch of keys on a key ring, where each key gives access to a room in a building. The deciding factor for access to a room is whether the subject has a key. The corresponding analogy for an ACL is a list showing which subjects are permitted to enter particular rooms in the building, where the building's doorkeeper takes individuals to rooms which they are permitted to enter.

Just as a key does not need to be on a key ring to be useful, so a capability can be useful without being held in a list. Hence a capability can in OS design be regarded as a separate entity which may but need not be held in a CL. On the other hand the concept of an entry in an ACL as an independent entity is not so useful, and there is no corresponding independent concept in ACL-based operating systems.

Both abstractions can be efficiently implemented, because they do not require entries for the cases where no access is permitted. However, both potentially have disadvantages. A capability list (or an individual capability), because it is associated with a subject, has a revocation problem: how can access rights, once granted, be withdrawn? (How do I get a key back from someone to whom I have distributed it, if he is unwilling to give it up?) In an ACL based system revocation is not a problem: the owner can simply remove or change the entry in the ACL, because this is associated with his object, not with the subjects.

ACL based systems can have a different problem: how can a subject express his right to access an object when he cannot name/find the object? (This leads in a system such as Unix to the situation where users typically browse through directories, and may thereby obtain information about objects to which they have no access.)

In practice some systems combine the two techniques. For example in a hotel the reception clerk might consult a list (ACL) to determine whether a potential guest should have access to a room and then hand him a key to the room (i.e. an individual capability).

Rule Based Access: the Access Rule Model

Implementations of Lampson's matrix are often insufficient to represent the access controls required in a real system. Some implementations (e.g. Unix) place restrictions on the naming of subjects, forcing them to be grouped into inappropriate categories. But even more significant, the control of access rights often needs to be coupled with arbitrary rules for governing access. For example an employer might want to implement the access rule: "Employees [the subjects] can only access information in file X [the object] between 8 am and 6 pm [access rule]" (possibly with a list of exceptions to the rule and maybe also with a control rule that accesses should be monitored and recorded in a logging file).

Such rules can be expressed using the "access rule model". According to this model [2] an access rule takes the form:

```
condition: s → o.ar
```

with the meaning that a subject `s` has the access rights `ar` for the object `o` if the boolean `condition` evaluates to `true`. Lampson's matrix is a special case thereof:

```
true: s → o.ar
```

By using the quantifier ∀ in conjunction with subjects, access rights and/or objects, a single access rule can neatly express many fields of Lampson's matrix. Thus

```
∀S: S → Spooler.print
```

gives all subjects the right to print using the Spooler object. Similarly the access rule

```
∀O: Superuser → O.read
```

gives the Superuser read access to all objects in the system. The access rule

```
∀AR: Smith → MyFile.AR
```

gives `Smith` all access rights to `MyFile`.

With sets and the operation ∈ on sets, the rules can be used to discriminate more finely:

```
∀x ∈ BankTellers: x → Account.deposit
```

More complex conditions can be expressed by using boolean operators in the conditions:

```
∀x ∈ BankTellers ∧ ¬ Account.overdrawn: x → Account.withdraw
```

More powerful rules can be formulated by introducing predicates, e.g. `confined` and `confined_to` in order to express the need for confinement (see next section).

The access rule model has the advantage that it matches the OO paradigm, by interpreting the general notion of object as an OO object, and the access rights as the right to invoke methods. Arbitrary conditions can be programmed, provided that the language offers a mechanism for executing such code to check whether a method can be invoked. We shall see in section 6 that this proviso is fulfilled in Timor in the form of qualifying types with bracket methods.

Confinement of Information

The confinement problem is concerned with ensuring that an object or process, which itself has legitimate access to information (e.g. a spooler with access to information to be printed), cannot misuse this access by providing unauthorised third parties *indirectly* with access to this information. In many, but not all, cases the issue behind confinement is that the code of programs or objects to which information is passed to perform a service is untrustworthy, either because the programmer of the implementation has introduced deliberate "errors" which are to his advantage or to the advantage of a third party, or because the code has been "infected", e.g. by a virus.

Identifying Subjects and Objects

In order to achieve adequate protection it is important to be able uniquely to identify the subjects who exercise access rights and the objects on which they are exercised. This is not as trivial as might first appear. There are several difficult issues.

The first concerns the concept of uniqueness as such. In many operating systems and programming languages identifiers are provided for subjects and/or objects which are not unique in the sense that they may be re-used over time. This can of course lead to confusion and errors, and to misuse by hackers. One (but by no means the only) root of this problem in OO environments used with conventional operating systems is that the systems are not normally persistent. For example each time a user logs in, the operating system creates a new process for him which has a non-unique identifier. In a persistent system, this issue can be resolved by having persistent processes/threads which are re-used by a user whenever he logs in (see section 8). In such a system these processes/threads can retain the same identifier over many user sessions. The important issue then becomes: how can such processes be associated with the right users? This issue will be discussed in section 8.

A second significant issue is the ability to make distinctions with respect to who/what is intended as the subject of an operation. There are at least three possibilities:

1. The *user* may be the subject who needs to demonstrate that he has an access right. For example User A might give User B the right to read a file F.
2. Individual objects might need to demonstrate a right to invoke another object. For example only the code of my text file objects might have the right to read my preferences file object.
3. Objects of a particular type or objects with a particular implementation might (or might not) have the right to access a particular object. For example, I might decide that my file objects should never be accessed by other objects which have code produced by a particular software house.

In the following sections we describe the facilities available in Timor to handle the protection issues discussed in this section.

3 UNIQUE IDENTIFIERS

As indicated in section 2, systems which aim to provide a satisfactory level of protection must be in a position uniquely to identify subjects and objects. All global identifiers in *SPEEDOS* are unique world-wide and over time.

There are two standard types `GlobalID` and `LocalID`. These allow unique identifiers to be stored and compared.

The type `GlobalID` supports identifiers which can be used globally. Identifiers of this type define users, their processes and their persistent file objects. They can be used globally to identify particular users, objects and processes. The global identifier of a user is the identifier of the first persistent process which is created for him. If he deletes this process he ceases to be a user.

Not all global identifiers can be obtained from within a Timor program. Some of the identifiers which are available are accessible in Timor programs via the following pseudo variables:

Pseudo variable	Returns the global identifier of the
`processID`	current process
`processOwnerID`	owner of the current process
`fileID`	current file object
`fileOwnerID`	owner of the current file
`implID`	code implementation of the currently executing code
`implOwnerID`	owner of the current code implementation
`callingFileID`	file having directly called the current file
`callingFileOwnerID`	owner of the calling file object
`callingImplID`	code of the implementation which called the current file
`callingImplOwnerID`	owner of the calling code

Each global object is marked with the identifier of the user which created it, and is subsequently known as its owner. (Ownership can be changed by mechanisms provided in SPEEDOS, but the details of this are not relevant to this paper.) Notice that a single `owner` operator with an operand which defines a `GlobalID` would be a risk to security in that it would allow users to establish the owner of any arbitrary object world-wide! Hence owners of objects can only be established via pseudo variables which are relevant to the current protection environment of a program.

An `implID` is the global identifier of the actual program which has been used to implement the currently executing code or that of the file or local object which invoked it.

The type `LocalID` supports identifiers which can only be used within the context of a particular global object or process to identify constituent objects or threads. The significance of this two tier identifier system is discussed in detail in [7]. Some local identifiers are accessible in Timor programs, via the following pseudo variables:

Pseudo variable	Returns the local identifier of the
`threadID`	current thread
`objectID`	current local object
`callingObjectID`	local object which directly called the current object

Since threads and local objects are constituent parts of processes and files they are owned by the owner of the process or file of which they are a part.

The uniqueness of `GlobalID` identifiers over time and space is organised by *SPEEDOS* (or the *SPEEDOS* emulator), while the uniqueness of `LocalID` identifiers (within a global object) is guaranteed by the local Timor runtime system. Some further global and local identifiers will be introduced after we have discussed qualifiers in section 5.

Because Timor rigorously adheres to the in-process model, there is no pseudo variable which returns an identifier for a "calling process" or "calling thread".

4 THE TIMOR CAPABILITY CONCEPT

At the programming language level a capability based approach to access control is an appropriate form of protection, because it reflects the idea that an object can be referenced if a pointer (variable) for it is within the current addressing scope. In this sense Timor references for local objects and capabilities for file objects (and thread references/process capabilities) can be considered as examples of the more general capability concept.

Timor references and capabilities can be *restricted*. A restricted reference or capability provides access to only a subset of the methods of the object assigned to the variable (see [7, 9]). In this sense each reference or capability has an associated set of access rights.

This provides the basis for a very powerful capability system in the sense that objects can be defined *and protected* in terms of the semantics of individual user applications. For example a bank account object might be defined to have semantic methods for making deposits and withdrawals, for authorising overdrafts, for accumulating interest, etc. and the right to invoke such methods (on an individual basis) can be protected using restricted references or capabilities as appropriate.

At the level of file objects, which in Timor are only accessible by invoking their methods, this means that a much finer grain of semantic protection is available than is found in conventional operating systems, which support access controls for files primarily in terms of basic operations such as read, write and/or execute.

In Timor the same kind of semantic access controls can be applied to local objects within a file (by restricting references). Such a facility can provide a high degree of protection e.g. for database objects.

5 THE TIMOR QUALIFIER CONCEPT

It is sometimes argued that a capability should be the necessary and *sufficient* condition to guarantee access to an object (e.g. [15]). This view fits nicely with the fact that in most systems it is difficult to revoke access rights provided in capabilities. However, it is not the Timor view. Timor provides a quite different concept (known as qualifying types), which inter alia allows such revocation to be achieved. Qualifying types are described in more detail in [4-6, 9, 10]. The basic idea is that a qualifier (an instance of a qualifying type) can be associated with another object (the target). When a client object invokes a method of the target object a call-in bracket method of the qualifier object "catches" the invocation (see Figure 1).

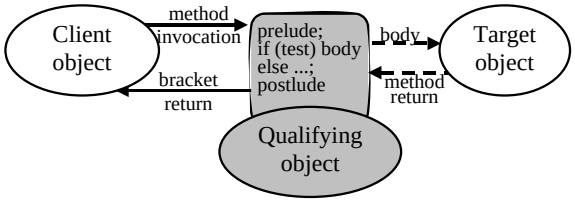

Figure 1: A Testing Bracket Method

It executes prelude code which can, for example, carry out any arbitrary test to decide whether to use a special `body` statement and so allow the call to proceed (or not). Since a qualifier is a separate object it can have its own state (and normal instance methods for reading and changing this). A qualifier's state can be used to store information needed, for example to check whether the capability access rights used to invoke the target object have been revoked by its owner[1].

The information held in the state of a qualifier might for example take the form of a "revocation list", i.e. a negative ACL. With this scenario it is a simple matter for the owner of a target object (either at the local or object level) to revoke access which he has granted via a reference or a capability by associating with the target a qualifier containing a list of (unique identifiers for) forbidden subjects which the bracket method can examine to determine whether the caller's access has been revoked.

Alternatively the list might be organised as a "positive" ACL, i.e. containing a list of subjects who are permitted to call the target. Other subjects, even if they have a reference or a capability, are not allowed to proceed.

Which objects are qualified can be determined on an individual basis (e.g. some objects of a type may be qualified while other objects of the same type may not be). It is possible to add qualifiers to an already existing object. Different target objects (of the same or different types) might be qualified by a single qualifier or by different qualifiers of the same or different types. Hence several objects might be protected by the same ACL or each might be individually protected by a separate ACL.

[1] or of course by some other subject with access to the instance methods of the qualifier.

Which invocations of an object are caught by bracket methods depends on how the latter are specified. A bracket method can be defined to qualify individual methods of a type, or of a view (a set of methods designed to be integrated into many types) or it can qualify methods which fall into a particular category (e.g. all the `op` or all the `enq` methods[2]). Since `op` methods (operations) are instance methods which potentially change the state of the object and `enq` methods (enquiries) are instance methods which cannot change the state of the object, the designer of a qualifying type can distinguish between reader methods and writer methods. This allows him to define protection in terms of reading and/or writing, as is commonly needed in protection environments.

Much finer access controls can be programmed in a qualifier, e.g. using bracket methods which with the aid of a further pseudo variable can check whether a particular method can be invoked (i.e. selective revocation of individual access rights) or which demand a password before access is granted, etc. Any arbitrary check can be programmed into a qualifying type and its bracket methods, allowing most access rules which might be defined using the Access Rule Model (see section 2) to be implemented.

Qualifiers (if they are created as file objects) can be used to control access to file objects or (if they are created as local objects) to control access to the local objects within a file.

Because a qualifier which executes a `body` statement is not the "real" invoker of its destination, programmers both in the qualifiers and target objects need to be able to distinguish between qualifiers and normal objects when obtaining unique identifiers. Hence some further pseudo variables are provided. Because several qualifiers can be placed between a caller and its target, it must be possible to cycle through a list of qualifiers. Because the aim of this paper is to enunciate principles rather than provide details, we leave the latter to the reader's imagination.

6 CONFINING OBJECTS

Unfortunately, not all access control rules can easily be implemented using the mechanisms described so far. In particular, the problem of confining information is in some cases easy to express informally (or more formally with the access rule model [2]), but is by no means easy to implement in conventional systems.

For example, the users of an operating system frequently need to invoke a spooler module to print their files. But how can they guarantee that the spooler, which has legitimate access to the information in the file, will not make a secret copy which can be accessed by an unauthorised third party? With conventional means it is almost impossible in a straightforward manner to confine a spooler (or editor or other service program) such that it is guaranteed not to release information entrusted to it in good faith.

[2] All the instance methods of a type must be explicitly defined as `op` or `enq`.

Confinement Control by Call-out Bracket Methods

In Timor programs some simple forms of confinement can be implemented using bracket methods. So far only call-in bracket methods have been mentioned, but Timor also supports call-out bracket methods, i.e. bracket methods which are activated when the target with which their qualifier is associated invokes a method of some other object. Figure 2 shows the relationship between call-in and call-out brackets. A qualifier may have one or both kinds of bracket methods (cf. [10]).

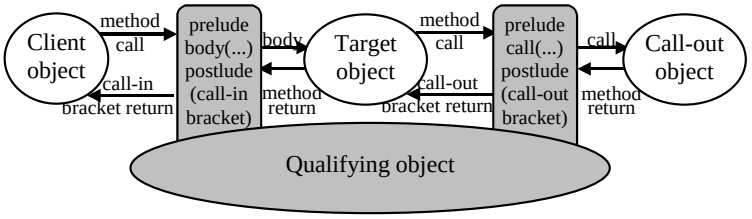

Figure 1: A Qualifier with Call-in and Call-out Bracket Methods

Just as a call-in bracket can determine whether to invoke a target method, so a call-out method can determine whether an executing object can invoke other objects. Whereas for a call-in bracket it is generally useful to determine which subject is calling, in a call-out bracket method it can be useful for confinement purposes to determine what object is being invoked by the object which it is qualifying. This information (which might be formulated in terms of the called object as such, or for example the code implementing it or the owners of either of these) can be obtained in a call-out bracket by using further pseudo variables.

It is thus possible to program a call-out bracket method to determine whether a target may invoke methods of some specified object(s), or pass particular information or capabilities to another object or list of objects. This might be achieved for example by checking a list of forbidden (or permitted) destination modules – or their owners, or modules with specific implementations or implementations from specific software houses, etc. – stored in the state of the qualifier.

Confinement Control by Unsetting Confinement Permissions

Call-out bracket methods allow information to be confined in terms of arbitrary rules, but only on the basis of two relatively naive assumptions, first that the subject wishing to confine the information has control of the object to be confined (i.e. has the right to associate a qualifier with it), and second that it is adequate to block a method invocation entirely. Unfortunately neither of these assumptions need be true.

For example a capability for "my" file can legitimately be needed by an editor that provides a text editing service which I would like to use. But I am not authorised to add a qualifier to the editor to ensure that it does not pass on the contents of my file to a third party (or secretly print it on a printer at some remote location of which I am not aware, etc.).

The other problem is that even if I could prevent a method invocation, how can the editor function properly if it is not permitted to invoke subsidiary objects to assist it, e.g. a dictionary object or an object in which my editing preferences (e.g. my preferred font, my styles, etc.) are stored.

These examples show that confinement is a non-trivial problem, and most systems have no adequate solutions. Following ideas formulated for confinement control in the *SPEEDOS* operating system (which are described in greater detail in section 6.8.3 of [1]), the Timor run-time environment provides a relatively simple set of "confinement permissions" which in effect allow the caller of a method selectively to switch off particular permissions and so restrict the operations which the called object can carry out. These include the possibility of switching off the called method's right to modify

- its own state (and hence for example preventing it from persistently storing information which its caller wishes to pass as confined information) and/or
- the state of its input parameters (thus preventing it from passing confined information back to its caller)

and/or to

- return values.

More drastically it is also possible to

- forbid calls entirely or
- forbid calls to objects which have state variables.

All these measures can be applied separately in terms of a (primary) call to a target object and/or in terms of secondary calls (i.e. calls from a target to further objects – which are then automatically applied to further object calls). It is also possible to apply different secondary confinements to calls which are based on parameters passed to the primary module and calls based on references or capabilities not known to the user applying the confinements.

These mechanisms can easily be used for example to permit an editor to access the file/local object to be edited, a dictionary module and a preferences file (all passed as parameters) while permitting calls to other objects only on condition that the state of these objects may not be modified. This condition in effect prevents information from the file which is being edited from being stored in secret places, where it might otherwise later be accessed by a third party cooperating with a Trojan horse in the editor code.

7 CONTROLLING UNTRUSTED QUALIFIERS

More subtle protection problems can arise, especially in relation to the use of qualifiers, e.g. what is to prevent a bracket method itself from secretly accessing methods of a target object or their parameters? We briefly outline some of the mechanisms available in Timor which can be used to prevent leakage of information from untrusted bracket methods.

First, a qualifier is like any other object in that it cannot directly access the methods of an object (including its target object) unless it has an appropriate reference or capability for the object. Without such a reference a qualifier can access its target using only the `body` statement from within authorised bracket methods.

Second, even in cases where a bracket method can legitimately catch a method invocation, it can only access parameters being passed to and from its target if (a) the qualification is for a specific method and (b) the type definition indicates that parameters can be accessed in the bracket method (i.e. the parameters are explicitly listed in the bracket method definition). Thus any bracket method defined to have parameters syntactically expressed as `(...)` has no access to parameter and return values.

Third, even if a bracket method formally has access to parameters it is possible to ensure that the latter are rendered unintelligible. For example [10] describes how the same qualifier can be associated with a client object and with the target object which it invokes, such that a call-out bracket associated with the client object encrypts a parameter (e.g. defined as a `String`) and a call-in bracket of the same qualifier decrypts the same parameter, on the basis of an encryption key stored in the qualifier's state. One effect of this is that all further bracket methods activated between the encryption and the decryption can only view encrypted data.

Fourth, there is a run-time check which ensures that a `body` statement can be executed only once during the activation of a bracket method, thus ensuring that the programmer of a bracket method cannot cause a problem akin to replay attacks by repeatedly invoking the `body` statement.

8 USER AUTHENTICATION

The most fundamental security issue in a system is that its active users are who they claim to be. Here the main problem is to ensure that a hacker cannot impersonate a genuine user. Unfortunately most current systems are relatively unsafe in this respect because generally speaking all users of a system have to authenticate themselves in the same way (e.g. by providing a password) and there is generally a central repository of information in the system (e.g. a password file). Consequently a hacker has the advantages first of knowing how he has to pass the authentication test (e.g. by providing the right password) and often he knows where to look (e.g. the system password file) to obtain the authentication information. That the information might be encrypted may make his task more difficult, but he nevertheless has the advantage of knowing how he can go about cracking the system.

As described in [8] Timor persistent threads can use a powerful authentication technique for checking the identities of users as they log in. This technique was first implemented for the persistent processes in the MONADS system [3] and is also included in the *SPEEDOS* operating system [1]. The basic idea is that when a user logs out, the thread via which he has been controlling his work persists in a suspended state. When he logs in again the thread is activated at the point where it was suspended. The user has the freedom to arrange that this re-activation point is in a user-supplied authentication module that can be programmed to authenticate the person attempting to login in any way which he chooses.

In this way each user can easily define his own authentication procedure rather than simply relying on a standard mechanism such as password checking. Hence a hacker does not know a priori what he has to do to login correctly, nor is there a central repository of authentication information which he can attempt to crack. Authentication modules can be written as normal Timor objects, and in non-*SPEEDOS* envi-

ronments are supported at run-time by a *SPEEDOS* emulator which is part of the Timor run-time environment.

9 RELATED WORK

The basic protection mechanisms of capabilities and access control lists, as described in section 2, were formulated many years ago and have been largely ignored in modern programming languages, which in general scarcely provide any significant programmable protection features. Similarly conventional operating systems are inadequate in terms of their support for protection and security concepts, with the result that software systems are currently suffering from a severe security crisis.

For this and other reasons the designers of Timor decided to extend the conventional run-time environment by providing Timor programs with the basic features of a secure operating system environment. This takes the form of an emulator for relevant parts of the *SPEEDOS* operating system (assuming that programs are running on a system other than *SPEEDOS* itself).

The *SPEEDOS* system [1] is being developed as a parallel project to Timor, and has as one of its primary aims the provision of an environment in which its users can enjoy (and themselves provide components which support) a high degree of protection. It includes all the features (at the OS level) which are described in this paper. It is a capability based system which supports the concept of qualifying types (enabling it to support the revocation of access rights provided in capabilities, ACLs, and any arbitrary form of protection which the user wishes to provide for checking access to his file objects). Like Timor file objects, these can only be accessed via methods, and only then when a suitable capability is supplied. It also has the features described earlier which allow confinement strategies to be implemented by turning off permissions (see section 6).

Qualifying types have a superficial resemblance to aspect oriented programming (AOP) [11] in that both allow the behaviour of objects to be modified using separately programmed "aspects". However both the style of integration into the base programming language and the implementation technique used, for example in AspectJ [12] have the effect that AOP is far less suitable than Timor for providing protection aspects. The main reason for this is that the code and state data of AspectJ aspects are integrated into objects by modifying the target class. The result is that the aspects do not have a separate state from the state of their target. Hence even if for example an ACL were programmed as an aspect, this could not be used to selectively control access to some objects of a class or to objects of different classes, and such an ACL could not be added later to an object.

Finally, because other programming languages do not support a concept of identifiers which are unique over time, they are not in a position to allow subjects and objects to be uniquely identified for protection purposes.

10 CONCLUSION

As is appropriate in modern globally networked computing environments, Timor provides a number of powerful protection mechanisms, including capability based protection for file objects (which are loosely equivalent to Java remote objects), a qualifier mechanism which can be used to implement access control lists, revocation lists, password checking, etc. and can enforce any access control rule which can be formulated as an executable algorithm. In addition it can be used to implement some confinement policies, while more advanced confinement is also supported through the use of certain permissions which the user can turn off to enforce particular advanced confinement strategies.

Because Timor objects and processes are automatically persistent [7, 8], and since the protection mechanisms described can function not only at the level of file objects but also at the level of the local objects which they contain, Timor can be used not only as an OO programming language but also as an object oriented database language, allowing highly protected databases to be developed and used with fine grained control at the level of the individual objects within the database.

REFERENCES

[1] K. Espenlaub, "Design of the *SPEEDOS* Operating System Kernel," *Department of Computer Structures*: University of Ulm, 2005, 234 pp., http://vts.uni-ulm.de/doc.asp?id=5333.

[2] M. Evered and J. L. Keedy, "A Model for Protection in Persistent Object-Oriented Systems," Security and Persistence, Proceedings of the International Workshop on Computer Architectures to Support Security and Persistence of Information, 1990, Springer Verlag, pp. 67-82.

[3] J. L. Keedy and K. Vosseberg, "Persistent Protected Modules and Persistent Processes as the Basis for a More Secure Operating System," Proceedings of the 25th Hawaii International Conference on System Sciences, 1992, vol. 1, pp. 747-756.

[4] J. L. Keedy, G. Menger, C. Heinlein, and F. Henskens, "Qualifying Types Illustrated by Synchronisation Examples," in *Objects, Components, Architectures, Services and Applications for a Networked World, International Conference NetObjectDays, NODe 2002, Erfurt, Germany*, vol. LNCS 2591, M. Aksit, M. Mezini, and R. Unland, Eds.: Springer, 2003, pp. 330-344, http://link.springer.de/link/service/series/0558/papers/2591/25910330.pdf.

[5] J. L. Keedy, K. Espenlaub, G. Menger, and C. Heinlein, "Qualifying Types with Bracket Methods in Timor," *Journal of Object Technology*, vol. 3, no. 1, pp. 101-121, http://www.jot.fm/issues/issue_2004_01/article1/, 2004.

[6] J. L. Keedy, K. Espenlaub, G. Menger, C. Heinlein, and M. Evered, "Statically Qualified Types in Timor," *Journal of Object Technology*, vol. 4, no. 7, http://www.jot.fm/issues/issue_2005_9/article5/ 2005, pp. 115-137.

[7] J. L. Keedy, K. Espenlaub, C. Heinlein, and G. Menger, "Persistent Objects and Capabilities in Timor," *Journal of Object Technology*, vol. 6, no. 4, May-June 2007, http://www.jot.fm/issues/issue_2007_05/article3/, 2006, pp. 103.-123

[8] J. L. Keedy, K. Espenlaub, C. Heinlein, and G. Menger, "Persistent Processes and Distribution in Timor," Journal of Object Technology, vol. 6, no. 6, July-August 2007, http://www.jot.fm/issues/issue_2007_07/article2/, pp. 91-108, 2007.

[9] J. L. Keedy, K. Espenlaub, C. Heinlein, G. Menger, F. Henskens, and M. Hannaford, "Support for Object Oriented Transactions in Timor," *Journal of Object Technology*, vol. 5, no. 2, March-April 2006, http://www.jot.fm/issues/issue_2006_03/article1/, pp. 103-124, 2006.

[10] J. L. Keedy, K. Espenlaub, G. Menger, and C. Heinlein, "Call-out Bracket Methods in Timor," *Journal of Object Technology*, vol. 5, no. 1, January-February 2006, http://www.jot.fm/issues/issue_2006_01/article1/, pp. 51-67, 2006.

[11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," ECOOP '97, 1997, pp. 220-242.

[12] G. Kiczales, E. Hilsdale, J. Hugonin, M. Kersten, J. Palm, and W. G. Griswold, "An Overview of AspectJ," ECOOP 2001 - Object-Oriented Programming, 2001, Springer Verlag, LNCS, vol. 2072, pp. 327-353.

[13] B. W. Lampson, "Protection," Proc. 5th Princeton Symposium on Information Sciences and Systems, 1971

[14] H. C. Lauer and R. M. Needham, "On the Duality of Operating System Structures," *ACM Operating Systems Review*, vol. 13, no. 2, pp. 3-19, 1979.

[15] R. M. Needham, "Capabilities and Security," Security and Persistence, Bremen, 1990, Springer-Verlag, pp. 3-8.

[16] K. Ramamohanarao, "A New Model for Job Management Systems," *Department of Computer Science*: Monash University, 1980.

About the authors

J. Leslie Keedy retired from the position of Professor and Head, Department of Computer Structures, University of Ulm, Germany in 2005, where he previously led the Timor language design and the Speedos operating system design groups. His email address is keedy@jlkeedy.net. His biography can be visited at http://www.jlkeedy.net/biography_short.php

Klaus Espenlaub completed his Ph.D. in Computer Science at the University of Ulm in 2005. He is currently employed by InnoTek Systemberatung GmbH. His research interests include secure operating systems, protection mechanisms and computer architecture. His email address is klaus@espenlaub.com.

Christian Heinlein is Professor for Fundamentals of Computer Science and Software Engineering at Aalen University, Germany. In his research, he has developed "Advanced Procedural Programming Languages", which are both conceptually simpler and more flexible than standard object-oriented languages. More information about him and his work can be found at www.htw-aalen.de/personal/christian.heinlein.

Gisela Menger received a Ph.D. in Computer Science from the University of Ulm in 2000. She recently retired from the Department of Computer Structures at the University of Ulm. Her research interests include programming language design and software engineering.