in *Proc. of ESORICS 92, Lecture notes in Computer Science, Springer-Verlag, (Y. Deswarte, G. Eizenberg, J.J. Quisquater editors)* , November 1992, pp. 193-208.

An Object-Oriented View of Fragmented Data Processing for Fault and Intrusion Tolerance in Distributed Systems

Jean-Charles Fabre

LAAS-CNRS & INRIA
7, avenue du Colonel Roche
31077 Toulouse cedex
(France)

Jean-Charles.Fabre@laas.fr

Brian Randell

Computing Laboratory
University of Newcastle upon Tyne,
Newcastle upon Tyne, NE1 7RU
(United Kingdom)

Brian.Randell@newcastle.ac.uk

Abstract. This paper describes a technique, called Object-Oriented Fragmented Data Processing, for jointly improving the reliability and security with which distributed computing systems process sensitive information. The technique protects the information contained in, and the processing performed by, a given object by first fragmenting the object into the subsidiary objects of which it is composed. It then relies on the (i) the correct execution of a majority of a set of copies of these subsidiary objects, and (ii) the reliable storage of a majority of a set of copies of each of these subsidiary objects, having distributed the subsidiary objects widely across a number of computers in a distributed computing system. The intent is to impede intruders and to tolerate faults, and involves ensuring that an isolated subsidiary object is not significant, due to the lack of information it would provide to a potential intruder. This technique can be applied to application objects and/or to the objects used in the implementation of the basic object-oriented system. The paper illustrates the technique using a detailed example, of an "electronic diary", that has been designed using Eiffel, and experimented with using the DELTA-4 Support Environment.

1 Introduction

The notions of reliability/availability and security though attributes of the generic concept of dependability [1], are often considered separately because the techniques used to achieve them are usually perceived as being mutually antagonistic. Firstly, *reliability* and *availability* are generally achieved by incorporating mechanisms for tolerating any faults (especially accidental faults) that occur, or that remain despite attempts at fault prevention during the system design process. These techniques will of necessity involve space and/or time redundancy; they can easily take advantage of a distributed computing architecture by means of replicated computation using sets of untrusted (or fallible) processors. Secondly, *security features* are generally achieved by means of fault prevention mechanisms (w.r.t. intentional faults, such as intrusions)

This work was supported in part by the CEC-sponsored ESPRIT Basic Research Action n° 3092 PDCS (Predictably Dependable Computing Systems).

in *Proc. of ESORICS 92, Lecture notes in Computer Science, Springer-Verlag, (Y. Deswarte, G. Eizenberg, J.J. Quisquater editors)* , November 1992, pp. 193-208.

whereby critical applications are implemented using physically and/or logically protected computers; such protection is usually based on the *TCB* (Trusted Computing Base) or *NTCB* (Network Trusted Computing Base) concepts. Here we provide instead an overview of recent work to a *combined* approach to the provision of *both* reliability/availability and security, as applied to object-oriented systems.

2 Fragmented Data Processing

The technique termed "Fragmented Data Processing" (FDP) [2,3,4] is a new approach to the combined provision of overall system security (in the sense of data and processing confidentiality) and reliability in distributed systems. It can provide each of the users of a distributed system with an individual set of processing and storage resources which are to a great extent protected not only from the effects of hardware and software faults but also of so-called "intrusions". By this term we mean (presumably) deliberate attempts by other (possibly unauthorized) users of the system to gain information from, or modify, or deny access to, the user's resources. For example, such attempts could even involve tampering physically with the hardware, or inserting "Trojan Horse" software.

The FDP approach, and the original Fragmentation-Redundancy-Scattering (FRS) scheme [5] on which it is based, are strongly related to conventional fault tolerance techniques. FDP achieves high reliability/availability and security for critical applications by arranging that their execution depends merely on (i) the correct execution of a majority of a set of copies of each of a number of program fragments, and (ii) the reliable storage of a majority of a set of copies of each of a number of data fragments; such fragments are widely distributed across a number of computers in a distributed computing system so as to impede intruders and to tolerate faults, and are defined so as to ensure that an isolated fragment is not significant, due to the lack of information it would provide to a potential intruder. (The problems of ensuring the security and reliability of the underlying network will not be considered here, though FDP-like approaches, such as in [6] for meshed networks interconnection, as well as more conventional solutions, to these problems are quite feasible.)

In effect, fragmentation and scattering is just a form of encryption, though one whose overheads are quite modest, and whose use fits well with general fault tolerance provisions (replication and voting) that are aimed at providing high reliability and availability despite the presence of hardware and software faults. Indeed, the crucial point about FDP is that the services it provides depend not on the integrity of any individual software or hardware components (which would imply the existence of "single points of failure"), but rather on majority voting by members of various sets of components. It simply presumes that such majorities exist (thus assuming a limit on the number of simultaneous faults) and in particular that voting is not being invalidated by either accidental or deliberate collusion between voters.

More specifically, systems employing FDP are, from the point of view of each user, divided into two sets of resources, namely a "trusted" (and it is hoped trustworthy) set and an "untrusted" set. Typically, the untrusted resources form a shared set of processing and storage servers, which users access from their individually trusted personal workstations, and it is in these terms that the technique will be described here.

Two major implemented examples of the application of the original FDP scheme have been completed, both using the DELTA-4 distributed system [7]. These are respectively an archiving system [8] and a user authorization service [9,3].

With respect to sensitive information processing, several types of FDP techniques which have been described in [4,10] can be used to produce scattered application fragments at different granularity levels, including:

(i) Structured fragmentation, which treats program and related data structures together in producing sets of fragments for replication and scattering. Each fragment consists of one of the programmer-defined code modules (this can be recursively performed on sub-modules within modules) and its local data on an instruction-by-instruction (or block-by-block) basis, with the global data being shared (transmitted) somehow between such fragments.

(ii) Bit-slice fragmentation, which consists in defining fragments of basic data items without regard to the way in which they are formed into larger data structures or used by the program. This technique does not try to make the program code secure - it just requires the necessary multiple variants of the program needed to deal with the set of different data slices.

Although FDP was not originally based on the use of object-oriented programming, we show in this paper how it can take advantage of an object-oriented design, and how such a design enables the technique to be generalized. The resulting OOFDP technique is illustrated using a detailed example, of an “electronic diary”, that has been designed using Eiffel, and experimented with using the DELTA-4 Support Environment.

3 Object-Oriented Structured Fragmentation

The object-oriented model gives a reasonably straightforward method of implementing *structured fragmentation*, which involves arranging that objects are split into fragments consisting of the subsidiary object of which they were originally composed, and doing this simply by defining and providing an implementation of the appropriate class characteristic, and then choosing which classes of object should inherit this characteristic. Thus just as Arjuna [11] allows all objects of a class to be declared as recoverable, so the objects of a given class could be declared to be "Secured" by being fragmented and scattered.

Using a simple, albeit rather unrealistic example, one might have either the classes complexvector or vector (or indeed both) inherit the characteristic "secured". These three possibilities are symbolized in Figure 1, in which the various objects are labelled with numbers indicating the different (sets of) computers they have been allocated to. These numbers have been chosen based on the simplistic rule that the number of computers is minimized, subject to ensuring that the immediate sub-objects of any fragmented object, and the object itself, are allocated to different (sets of) computers, depending on which class(es) of objects have been defined to inherit the characteristic "Secured". Inheritance of the "secured" characteristic is denoted by "Secured: ObjectClass" in the following figures.

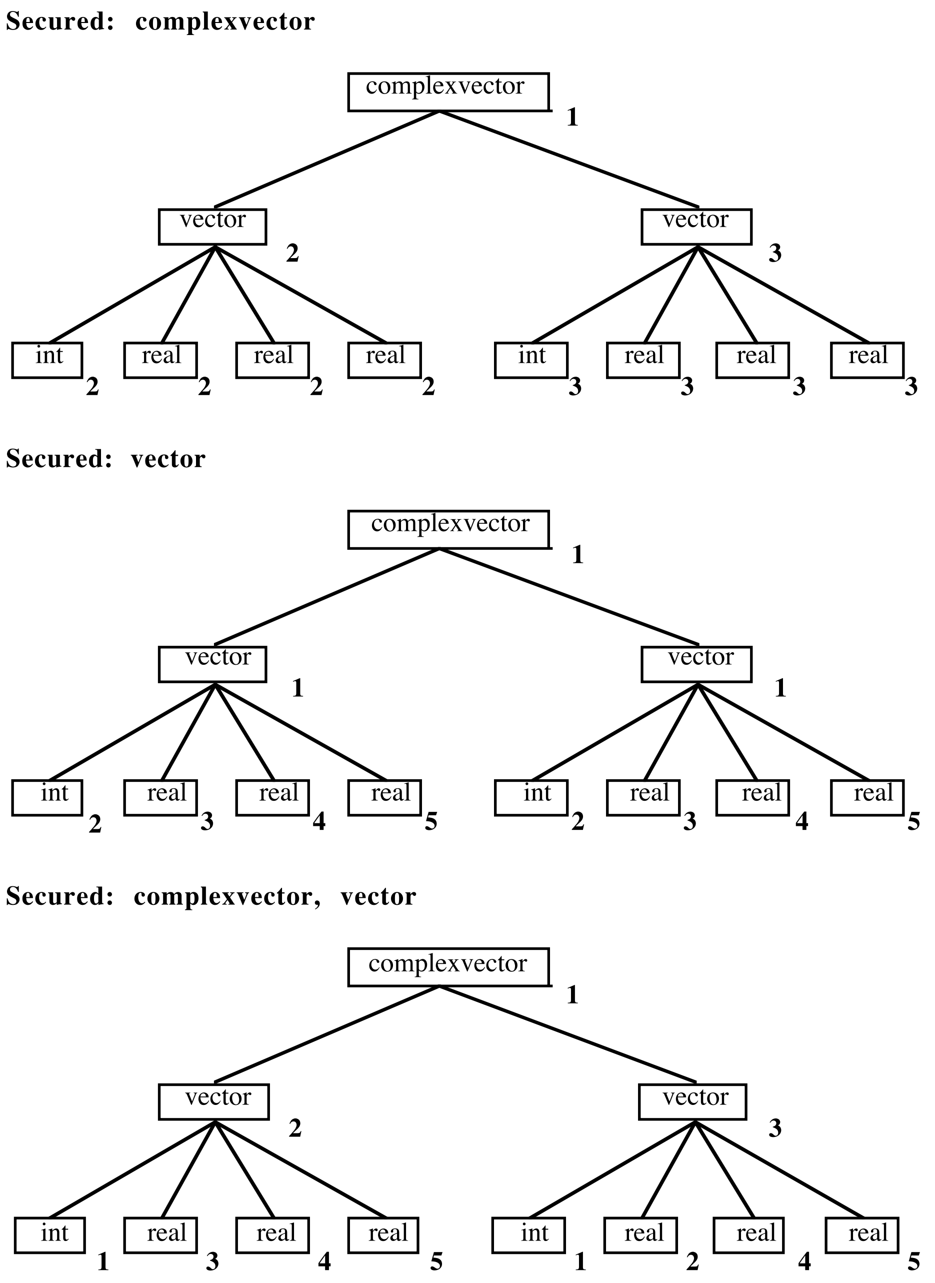
Secured: complexvector
complexvector 1
vector 2
vector 3
int 2, real 2, real 2, real 2
int 3, real 3, real 3, real 3
Secured: vector
complexvector 1
vector 1
vector 1
int 2, real 3, real 4, real 5
int 2, real 3, real 4, real 5
Secured: complexvector, vector
complexvector 1
vector 2
vector 3
int 1, real 3, real 4, real 5
int 1, real 2, real 4, real 5

Fig. 1 Different Inheritances of the Characteristic "Secured"

Such a method of fragmentation and scattering would leave the original object largely empty, apart from the information necessary for accessing its now remote subsidiary objects, and the code (or a reference to the code) for the various operations (methods).

The fact that the subsidiary objects were allocated, in many cases, to separate machines would provide significant potential parallelism for achieving a speeding-up of the original object's methods. (There already exist a number of techniques (under various different names) which are somewhat akin to fragmentation and scattering, aimed at exploiting parallelism for performance purposes rather than at providing security. These include, at the hardware level, so-called "disk striping" and, in object-based programming, the object fragmentation provisions of the SOS system at INRIA [12,13].)

The actual means by which such forms of fragmentation and scattering can be achieved, e.g. the methods for placing, and later accessing remote subsidiary objects, will depend on the strategy that is being provided to users for handling distribution problems. For example, a user who is programming the class "secured" might be provided with the simple, but rather inflexible, facility of a single virtual name space (e.g. [14]), whose implementation embodies and hides the distribution policies which are in use. Another alternative is the facility provided in SOS for allowing users to declare and implement shared distributed objects (termed "fragmented objects" in SOS) out of elementary objects which are located on different computers. (Clearly, with such an approach the distribution policies remain under user program control.) However in either case, a trusted means will be needed for identifying the set of fragments - normally some sort of "key"; methods for providing and, most importantly, protecting such keys are discussed in Section 7**.**

Fragmentation decisions might be largely static, based on information in the program or generated by the compiler, or might be highly dynamic. The latter would necessarily be the case if one wished to fragment and scatter the elements of a dynamic array of objects - and could be defined in a special class "scattered array", so that a single definition of fragmentation and scattering could be used for arrays of different classes of objects, provided class-based object structuring [15] and multiple inheritance are supported. The allocation of objects to computers involved in fragmentation and scattering could in principle also be, in some cases at least, partly static. Dynamic allocation, however, would allow one to make better use of a set of operational computers whose membership changes as computers fail and are repaired, and also to attempt to obtain performance improvements via load balancing.

In summary, the advantages of object-based methods of controlling fragmentation and scattering are that:

(i) they avoid the complications of global variables,

(ii) they readily support repeated fragmentation and scattering, that is of objects within objects,

(iii) fragmentation and scattering can be applied selectively, at least on a per class basis, and different object classes can use different methods of performing it,

(iv) a standard implementation scheme can be provided as a built-in class declaration, but this can be overwritten under programmer-control, so as to provide a scheme more attuned to a particular class or classes of objects.

(v) being in terms of the structuring already introduced by the programmer, their performance (including their ability to exploit the existence of multiple processors) will benefit from the fact that this structuring (presumably) reflects patterns of access and interaction, and so provides good locality of reference. (Alternatively, it perhaps could be argued that a fragmentation and scattering technique which cuts across programmer-defined structuring would be more effective in obscuring the semantics of the program from intruders than one which respects such structuring; this issue is pursued further in the next section.)

4 A Two-Level Object-Oriented Model

We assume, that at two (or more) levels of abstraction we have a distributed *complete interpreter*. In practice, the upper level is more likely to be just an *interpreter extension* (in the sense of the term as defined in [16]). Each of the two levels is viewed as supporting a set of objects. In general within each level (and certainly at the upper level) these objects will form a hierarchy (based on the "is-part-of" relation), with objects being composed in part of smaller objects, down to some set of elementary objects.

The relationship between the object hierarchy at the upper system level and the hierarchy that exists at the lower level could be quite distant, especially if the upper level is a complete interpreter (e.g. for Smalltalk, but written in C++). Even with a partial interpreter one might have quite complex inter-relationships, much as one might have objects such as segments provided by an operating system at the upper level, and pages provided by the hardware at the lower level, using an intricate scheme whereby small segments were packed into pages, and large segments split across pages.

With such a two-level system model, various different dependability-related characteristics could:

(i) be defined in the upper (application) level of abstraction, and associated with particular classes of objects, via the class declarations of their operations, and then could when so desired be inherited or redefined in further class declarations, and/or

(ii) be supplied by application-independent mechanisms in the underlying complete hardware (and therefore probably not very object-oriented) or software interpreter.

As seen from the application level, method (i) above corresponds to what was termed structured fragmentation in Section 2, and method (ii) corresponds to bit-slice fragmentation. The model makes it clear that conceptually the difference between the two methods is simply one of viewpoint, though in practice the effects achieved are very different, since the former involves use of the object structuring defined by the application programmer, and the latter can be quite independent of this structure.

The two methods could be used in combination. Indeed, lower-level fragmentation and scattering could in fact be implemented and used without explicitly considering the fact that the upper-level interpreter was also performing fragmentation and scattering, in much the same way that a conventional paging system might be unaware of dynamic storage allocation of arrays being performed by the program that is being paged. Alternatively, the interpreter could be specially designed to respect the object structuring defined at the application-level. That is, it might for example deal separately with the information relating to individual upper-level objects, provided that it has knowledge of their boundaries.

The use of an application-level scheme, as in method (i), in fact corresponds to the way in which the Arjuna distributed programming system [11] provides various reliability (but not as yet security) characteristics, such as recoverability, atomicity and stability for distributed application programs in the face of possible processor crashes. (Arjuna is based on C++ and UNIX, and provides its facilities completely by means of C++ declarations, without introducing any changes into the language, its run time support system, or the UNIX system.) An example which is equivalent to employing methods (i) and (ii) in combination is provided by the plan to augment Arjuna by means for the replication of, and voting by, the processors which implement the Arjuna system, something that the Arjuna group is investigating using multiple transputers [17].

5 An Example

We have investigated the object-oriented fragmented data processing using several detailed examples. The first was in fact based on part of the specification of the user authorization service [9,3] provided in the DELTA-4 distributed system [7]. A fuller account of this first example to be found in [18].

The example given here is a distributed Electronic-Diary[1] which has been designed using Eiffel [15] tools and implemented on top of the DELTA-4 Support Environment (DELTASE). A first prototype is currently running on a set of Unix workstations. We describe here this service by a small series of classes. The latter only address the definition of management operations on meetings day-per-day; the information related to a meeting is composed of a given topic, a group of people attending, a venue and time/date information. Any two or more of such items is considered as constituting confidential information. Otherwise, any person attending is defined by several identification items and can be considered as being public information.

For the moment, we omit from our example any mention of the mechanisms involved in providing or using fragmented data processing, deferring such matters until Section 5.2. The information used for the management of meetings is stored in each of a set of meeting descriptors and can be summarized as follows:

• *topic:*	topic to be discussed during the meeting - this is considered to be confidential information;
• *venue/time/date:*	place where the meeting is held and time/date information;

[1] The objective is not here to design a complete E-Diary with every functionnality a user can expect from such tool. Only a subset is provided to illustrate the object-oriented approach of FDP.

- *dynamic person list:* list of persons attending the meeting, all together constituting confidential information even though the identity of any of them is public information.

These descriptors are the main leaves of a tree (a subtree) of the E-Diary which is considered as being an object which is private to a given user (the E-Diary is not shared by multiple users).

5.1 Object-Oriented Management

This section presents an object-oriented approach to the management of *meeting* information; the basic items forming a meeting descriptor and further the person list are also taken into account but not shown in Figure 2. Some of the object classes (and their component objects) forming the E-Diary application object are shown in Figure 2, where an asterisk indicates the possibility of there being several components of a given object class.

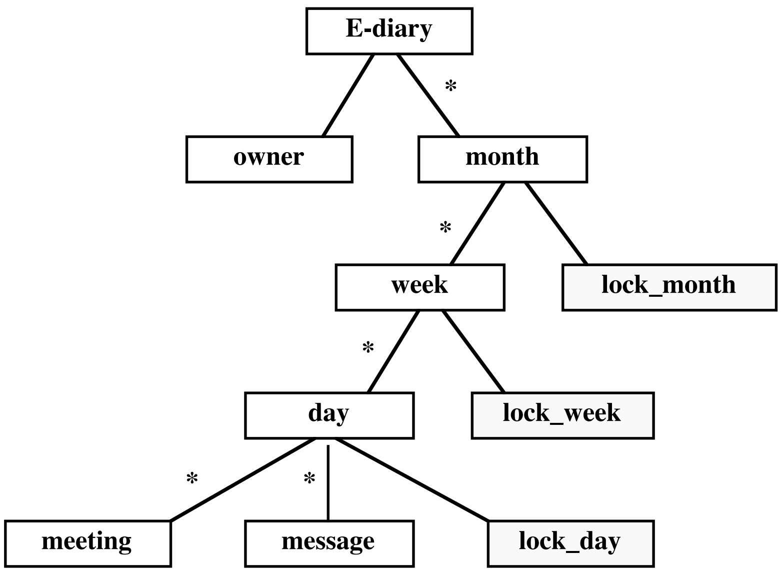

Fig. 2 The E-Diary object composition hierarchy

The object hierarchy represented in Figure 2 for the E-diary service is as follows: the *E-Diary* is composed of several *month* objects and is owned by a given user (*owner*). Each month is composed of a number of *weeks* and can be locked (*lock_month*) for a given reason (holidays, for instance). Each week is composed of a number of *days* and can also be locked (*lock_week*) for a given reason (travel abroad, for instance). Any day is composed of a list of *meetings*, a list of *messages* (note pad) and can be also locked for a given reason (*lock_day*). Any lock set to true implies that no meeting can be allocated in the month, week or day, respectively. The E-Diary is considered as a persistent object and can thus be activated (from persistent storage) after being created. It offers several services to the owner: create, modify, move, delete a meeting, put, release a message in the note pad of a given day, and lock a month, a week, or a day for a given reason.

Note the above makes no mention of inheritance, and hence of whether a public method's implementation is given in the class definition of the particular object, or is inherited from some other class definition.

The object which is of interest in the above hierarchy is the meeting object which contains confidential information; the composition hierarchy of this object is presented in Figure 3. A *meeting* is composed of a persons list (*P-list*), a *venue*, *time* and *topic*. The P-list can be implemented in various different ways, possibly using the Eiffel pre-defined class list (of *persons*). Person is composed of three sub-objects in our example: *name*, *address* and *position*.

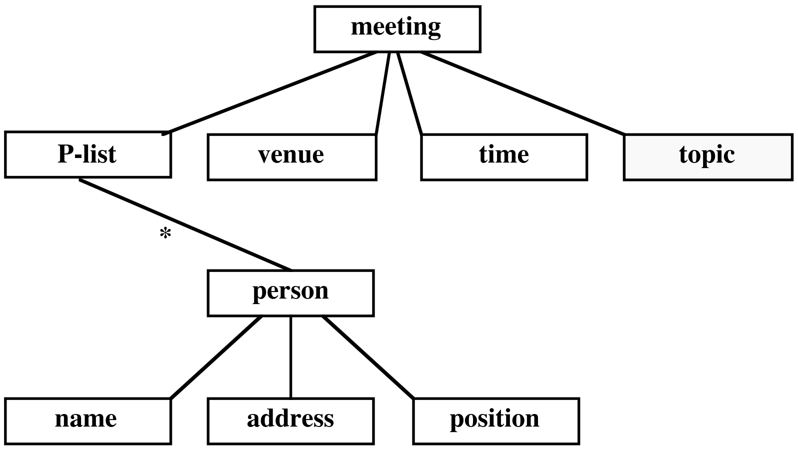

Fig. 3 The Meeting object composition hierarchy

The object hierarchies presented in Figure 2 and 3 (in a form similar to Eiffel browser output) illustrate the various components in the design of the E-Diary object down to elementary objects (i.e. a combination of Eiffel elementary objects such as integers, booleans, strings...). Some of the elementary objects represented by grey boxes are confidential leaves of the tree that it is assumed for our purposes cannot be usefully decomposed into smaller objects; for instance the topic is a string that is ciphered to ensure confidentiality as soon as it is entered by the user in the system. The same is true for lock objects which correspond to a boolean value and a string that indicates the locking reason.

In order to illustrate the usage of such object classes, we briefly present a few steps of some management operations on one example in which the creation of one meeting object is detailed. The E-Diary is created by invoking the E-Diary constructor, that further invokes the Month constructor twelve times, then the Week constructor and finally the Day constructor. From the user interface object (not presented in the above figures) located in the "trusted" area of the system, a new meeting is created by invoking the meeting constructor that defines and initializes data structures; then

in *Proc. of ESORICS 92, Lecture notes in Computer Science, Springer-Verlag, (Y. Deswarte, G. Eizenberg, J.J. Quisquater editors)* , November 1992, pp. 193-208.

input information provided by the user leads to the creation and filling of meeting subobjects such as persons' list, venue, time, topic. At each level, several checking operations are performed with respect to the various locks and to meeting overlaps.

5.2 Inheritance of the "Secured" Characteristic

In this section we give one example of the effects that can be achieved by arranging that the characteristic "secured" be inherited by a given object class, say the meeting class.

The effects of using such inheritance are discussed using the above hierarchy. For clarity in our examples we attach the characteristic "secured" directly to a chosen class to demonstrate what effect this will have. (In an object-oriented design of the E-Diary that used inheritance among its class definitions, one could arrange that the "secured" characteristic was inheritable from higher in the class hierarchy. The implementation of the characteristic "secured" and its effects on objects that inherit this characteristic are discussed in Section 6.)

In each of our examples, site number 1 represents the user site where the owner is able to execute management operations (in particular input/output operations) and where all the meeting descriptors are located when FDP is not used. In this first case the characteristic "secured" is attached to the *meeting* class. This solution leads to processing (and perhaps storing) *Person list* , *venue*, *time* and *topic* at distinct sites as shown in Figure 4. (The fragmentation and scattering of the P-list objects is discussed later.)

Secured: meeting

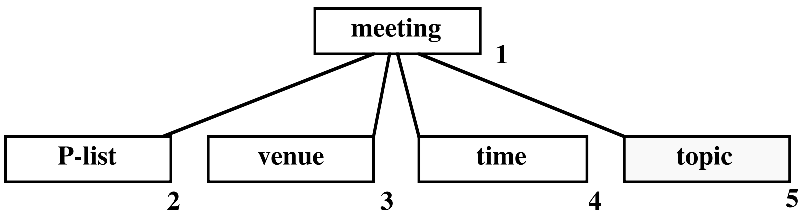

Fig. 4 "Secured" meeting descriptors

In this case, site 1 is responsible for the management of *meeting* objects. Considering just *P-list* objects, all the *person* objects that appear in the meeting will be managed (and perhaps stored) at the same site (say site 2). An intruder located at site 2 is unable to find out about the *topic* of the meeting (even in its enciphered form) ; the confidentiality of the relation (*person list*, *topic*) is thus preserved by sites 2 and 5. Similarly, an intruder located at site 5 is able to obtain the (enciphered) topic of the meeting but is unable to find out the list of persons attending.

The characteristic "secured" could be attached to the meetings but also inherited by the *P-list* object as in Figure 5. This solution leads to two phenomena:

1) *P-list*, *venue*, *time* and *topic* objects in the meeting are fragmented and scattered onto different sites (see figure 4);

2) Use of the characteristic "secured" at the *P-list* level, leads to the *persons* objects belonging to the list being fragmented and scattered.

Secured: meeting, P-list

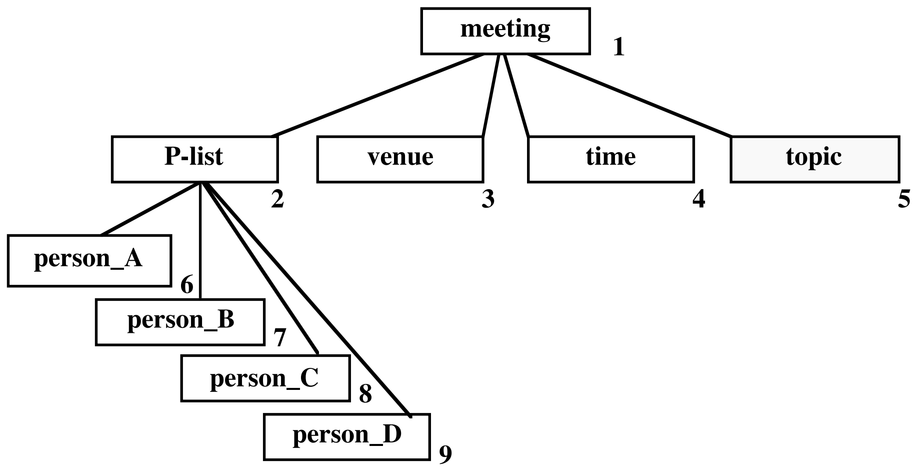

Fig. 5 Secured meetings and P-list by inheritance

This solution provides a complete fragmentation and scattering of *persons* objects belonging to the *P-list*. It would provide to an intruder a partial view on the meeting object, similar to the notion of views used in multilevel security in object-oriented database systems [19]. An intruder located at one of the sites 6 to 9 is unable to get the P-list information (the list of persons attending the same meeting). At site 2, the P-list object is a collection of references to person objects: these references are produced by a naming facility based on one-way functions similar to those used in the archiving system described in [3].

The result of this last solution leads to the definition of the current implementation of E-Diary on the DELTA-4 platform. DELTA-4 [7] does not provide an object-oriented layer but provides a runtime support for objects as a collection of servers responsible for object management; a server is defined by an interface composed of a set of operations described using an Interface Definition Language and operation activation from the clients is transparent. In our implementation a server is associated with an Eiffel class and is responsible for the management of object instances of this class. For a given class several servers can be created on several sites, such as *person* servers in the above example. The Eiffel design presented in this section has been mapped onto DELTA-4 by hand. At configuration time, any server can be created replicated for fault tolerance and several error processing strategies are available. The complete application including more objects and more confidentiality constraints (including operator interface, ciphering and naming functions) is currently running using scattering and replication (with majority voting) on a set Unix workstations, using the DELTASE layer and the DELTA-4 error processing protocols, based on DELTA-4's Multicast Communication System.

5.3 Commentary on the Example

The example presented in Section 5 reflects the use of an object-oriented design as a basis for implementing a variety of fragmentation-redundancy-scattering techniques. The example shows that different grades of confidentiality can be obtained using appropriately chosen fragmented and scattered objects. The aim of our example was not to provide a new implementation of an actual E-Diary Service. Nevertheless, it gives the flavour of several different strategies for ensuring the confidentiality and reliability of meetings, attendance lists and topics; for instance, meetings may be stored by meeting servers on a day-by-day basis locally on one given site, persons may be stored on a collection of persistent servers. More importantly, the actual implementation differences between these strategies are quite minor, since they only involve differing decisions regarding inheritance.

One interesting aspect of this simple example is that it shows that using the OOFDP approach the confidentiality of information such as the relation between *topic* and *person list* or the relation between people attending the same meeting together (i.e. present within the same *person list*) can be preserved with respect to intrusions performed by other unauthorized users (even site/workstation administrators) or by external intruders tampering with some sites in the network.

6 Implementing the "Secured" characteristic

What we have termed "attaching" the characteristic "secured" to a given class means arranging for this class to inherit a set of facilities defined in an appropriate class declaration. These characteristics will, for example (i) ensure that when object of this given class are created, their constituent sub-objects will be scattered, and (ii) provide each object with any necessary information, such as a "key", needed to control the fragmentation and scattering and the means by which the object's operations access its scattered sub-objects, and operations such as SetKey and GetKey. These operations may well be defined in, and inherited by the class declaration from, a class "key".

Some such keys might be used only at compile and generation time, and then deliberately discarded (i.e. for what can be termed "static" fragmentation and scattering). Others are likely to be retained within, or associated with, objects at run time (e.g. for "dynamic" fragmentation and scattering, in which sub-object names are computed when the sub-objects are invoked). Key objects might of course also be used for other purposes by other classes, as is the case in DELTA-4, where keys are also used in the implementation of the archive server.

However the characteristic "secured" should involve not just fragmenting an object when it is generated, into its sub-objects, but also replication as well as the scattering of the resulting fragments. According to the example given in the previous sections, replication of the *meeting* sub-objects needs a definition of an appropriate set of workstations to store, for instance, replicates of *P-list* objects.

The approach we suggest is that used in Arjuna, in which the replication characteristic is implemented using inheritance with a ReplicationBase class. A replicated object is declared as:

class ReplicatedObject: ReplicationBase;

The class ReplicatedObject inherits the replication methods from ReplicationBase class. An instance of a replicated object in five copies is declared as:

class ReplicatedObject(5)

where the number of copies 5 is passed to ReplicationBase.

The "secured" characteristic could be declared in the same way, using the class SecurityBase:

class SecuredObject: SecurityBase;

Thus, using the above declarations Fragmention-Replication-Scattering is implemented at the object level by multiple inheritance of the classes SecurityBase and ReplicationBase.

Different ways of implementing the stub generator of the SR-bases (where SR stands for Scattering or Replication) classes may be defined:

(i) First, a generic stub generator for any class of object would need to be able to determine the class of objects that is to be fragmented and scattered and/or replicated, and thus conduct the appropriate operation with respect to the object type including multiple initiations, handling multiple return values (and voting operations), etc.

(ii) A second solution would be to have one SR-base class per object class. The appropriate SR-operation to generate a given object instance would be performed by the associated stub generator; using inheritance and SR-operator overloading, the appropriate stub generator being invoked for each "secured" object (sub-object).

7 "Guarding the Guards"

Clearly, the techniques we have described so far depend on making sure that an intruder cannot easily reconstruct an object that has been fragmented and scattered. If a key (e.g. that expresses the relationships between its fragments) is needed to access a fragmented and scattered object then the security of this key of course becomes critical. Such keys could be kept and used only in appropriate trusted areas, or one could instead apply a further level of fragmentation and scattering to them. However, such a potentially indefinite recursion must eventually be broken either by the use of trusted areas, or by avoiding the use of keys.

This latter is the approach taken in the actual DELTA-4 system, which uses the notion of a threshold scheme [20] to make the management of the keys it uses for its scheme of FDP intrusion-tolerance (and, as mentioned earlier, also for other purposes). Such a scheme represents the value of an object which is to be kept secret

in *Proc. of ESORICS 92, Lecture notes in Computer Science, Springer-Verlag, (Y. Deswarte, G. Eizenberg, J.J. Quisquater editors)* , November 1992, pp. 193-208.

by a set of N shadows s1, s2, s3... (Thus the replicates of a given object would contain different shadows of the object's key, the key itself not being needed once the shadows had been created.)

This technique, which is appropriate only for relatively small objects such as keys, has the following properties:

(i) a number of shadows greater or equal to the threshold T is required to create the secret information,

(ii) less shadows than the threshold T do not give any information about the secret.

Implementing a key object by means of a threshold scheme thus can ensure the continued availability of the key, despite the occurrence of less that N-T faults, N being the number of shadows generated, as well as its confidentiality.

8 Concluding Remarks

The Object-Oriented Fragmented Data Processing approach that we have described generalizes the technique of FDP, as previously described and implemented. The multi-level view of system structuring which we have used shows how fragmentation-scattering can be implemented independently by each of a hierarchy of interpreters (and interpreter extensions), thus providing protection at various granularity levels.

We would therefore argue that FDP, perhaps in common with certain other approaches to security, so far explored mainly in the database world, can derive significant benefits from being viewed and used in conjunction with a suitable object-oriented structuring scheme. Indeed, its provision as an independently inheritable characteristic alongside the use of several forms of inheritable reliability characteristics (such as "stable" and "atomic") that have already been devised elsewhere, such as in the Arjuna Project, seems perfectly feasible. However, one particularly attractive feature of the FDP technique is that it would seem to have the potential of being simultaneously beneficial not just to security and reliability but also, because of its exploitation of parallelism, to performance - characteristics which are normally mutually antagonistic!

Detailed experimental investigations are however now needed to substantiate the hopes expressed in this paper, and to determine the likely actual cost/effectiveness of OOFDP, as compared to the FDP that has already been implemented in DELTA-4, and any other approaches to the joint provision of reliability and security.

Acknowledgements

The preparation of the work briefly presented in this paper has been greatly assisted by discussions in particular with Dan McCue, but also with numerous other colleagues, including Yves Deswarte, Ron Kerr, Jean-Claude Laprie, Mark Little, Peter Neumann, Robert Stroud and Gilles Trouessin. The implementation of the complete example has been done by Vincent Nicomette and Tanguy Perennou. This research was in part supported by the CEC-sponsored ESPRIT Basic Research Action on Predictably Dependable Computing

Systems, and was aided by a CNRS fellowship that enabled Brian Randell to spend a period at LAAS.

References

1. J.C. Laprie, Ed., *Dependability: Basic Concepts and Terminology* (in English, French, German, Italian and Japanese), series Dependable Computing and Fault-Tolerant Systems, (A. Avizienis, H. Kopetz, J.C. Laprie Eds.), Vol.5, Springer-Verlag, 265 p., ISBN 3-211-82296-8 and 0-387-82296-8, 1992.

2. J.-M. Fray and J.-C. Fabre, "Fragmented Data Processing: an Approach to Secure and Reliable Processing in Distributed Computing Systems", in *Proc. 1st IFIP Int. Working Conf. on Dependable Computing for Critical Applications*, Santa Barbara, California (USA), 1989, pp. 131-137.

3. Y. Deswarte, L. Blain and J.-C. Fabre, "Intrusion Tolerance in Distributed Computing Systems", in *Proc. IEEE Symp. on Security and Privacy*, Oakland California (USA), 1991, pp. 110-121.

4. G. Trouessin, J.C. Fabre and Y. Deswarte, "Reliable Processing of Confidential Information", *Proceedings of the 7th Internaltional Conference on Information Security, IFIP/Sec'91*, Brighton (United Kingdom), 1991, pp. 210-221.

5. J.-M. Fray, Y. Deswarte and D. Powell, "Intrusion Tolerance Using Fine-Grain Fragmentation-Scattering", in *Proc. IEEE Symp. on Security and Privacy*, Oakland, California (USA), 1986, pp. 194-200.

6. Y. Koga, E. Fukushima and K. Yoshihara. “Error recoverable and securable data communication for computer network,” in *Proc. 12th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-12),* pp. 183-186, Santa Monica, California (USA), 1982, pp. 183-186.

7. D. Powell, Ed., *Delta-4: A Generic Architecture for Dependable Distributed Computing*, series Research Reports ESPRIT, Project 818/2252, Delta-4, Vol. 1 of 1, 484 p., ISBN 3-540-54985-4 and 0-387-54985-4, Springer-Verlag, 1991.

8. P.G. Ranéa, Y. Deswarte, J.M. Fray, D. Powell, "The Security approach in DELTA-4", in *Proc. of the European Telematics Conference (EUTECO-88) on Research into Networks and Distributed Applications,* Viena, Austria, pp.455-466 (Ed. North-Holland, April 1988).

9. L. Blain and Y. Deswarte, "An intrusion-tolerant security server for an open distributed system", in *Proc. of the European Symposium in Computer Security (ESORICS 90)*, AFCET, Toulouse, France, pp. 97-104, 1990, ISBN 2-9036778-9.

10. G. Trouessin, Y. Deswarte, J.C. Fabre and B. Randell, "Improvement of Data Processing Security by means of Fault Tolerance", *Proceedings of the 14th National Computer Security Conference*, NCSC, Washington DC (USA), 1991, pp. 295-304

11. S.K. Shrivastava, G.N. Dixon and G.D.Parrington, "An Overview of the Arjuna Distributed Programming System", *IEEE Software*, vol. 8, #1, 1991, pp.66-73.

12. M. Makpangou, Y. Gourhant, J.-P.L. Narzul and M. Shapiro, "Structuring Distributed Applications as Fragmented Objects", INRIA Research Report 1404, INRIA, Rocquencourt, France, 1991.

13. M. Shapiro, Y. Gourhant, S. Halbert, L. Mosseri, M. Ruffin and C. Valot, "SOS: An Object-Oriented Operating System - Assessment and perspective", *Computing Systems*, vol. 2, #4, December 1989, pp. 287-338.

14. E.H. Bal and A.S. Tanenbaum, "Distributed programming with shared data", in *Proc. of the ICCL*, Miami, Florida (USA), IEEE, Computer Society Press, 1988, pp. 82-91.

15. B. Meyer, "Eiffel: Programming for Reusability and Extendibility", *ACM SIGPLAN*, vol. 22, #2, pp.85-94, 1987.

16. T. Anderson and P.A. Lee, *Fault Tolerance: Principles and Practice*, Prentice Hall, 1981.

17. P.D. Ezhilchelvan and S.K. Shrivastava, "A Distributed System Architecture Supporting High Availability and Reliability", in *Preprints, 2nd Int. Working Conference on Dependable Computing for Critical Applications*, Tucson, Arizona (USA), 1991, pp. 36-48.

18. B. Randell and J.C. Fabre, "FDP techniques in Object-Oriented Systems", *LAAS Research Report n°91114, Computing Laboratory of the University of Newcastle-upon-Tyne Research Report n°337,* 35p., May 1991.

19. T.F. Lunt, "Multilevel Security for Object-Oriented Database Systems", in *Proc. 3rd IFIP Workshop on Database Security,* Monterey CA, USA, 1989.

20. A. Shamir, "How to Share a Secret", *Comm. ACM*, vol. 22, #11, pp.612-613, 1979.