FDP techniques in Object-Oriented Systems

Brian Randell* and Jean-Charles Fabre**

*** Computing Laboratory,**

University of Newcastle upon Tyne, NE1 7RU, UK

**** LAAS/CNRS and INRIA,**

7 Avenue du Colonel Roche, 31077 Toulouse, France

1. Introduction

The technique termed "Fragmented Data Processing" (FDP) [Fray and Fabre 1989], [Deswarte, Blain et al. 1991], [Trouessin, Fabre et al. 1991] 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 [Fray, Deswarte et al. 1986] 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.

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.

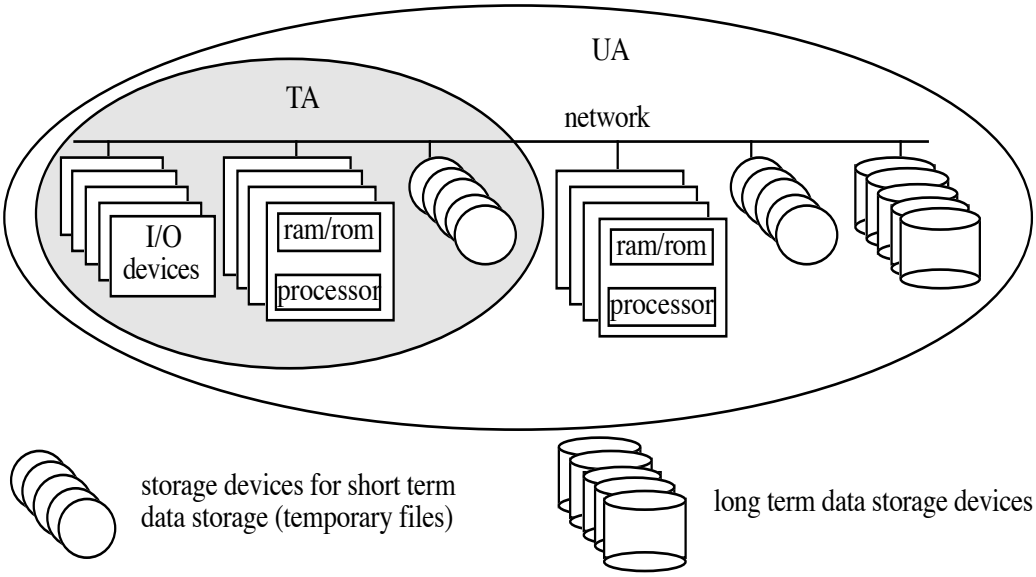


Figure 1: A User View of the System

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, as illustrated in Figure 1, which along with Figure 2 comes from [Fray and Fabre 1989]; users can designate particular applications running on the "untrusted" set of resources, as being ones which are to be protected (using FDP and the "trusted" resources) against the effects of accidental and intentional hardware and software faults. This protection involves the use of the trusted hardware and software resources, whose own integrity (including freedom from intrusions) is thus relied upon by the FDP mechanisms. The trusted resources, with the user and his/her means of accessing these resources, constitute a "Trusted Area". (The issue of what precautions are needed for these resources to be worthy of the given user's trust, and hence such reliance, are dependent upon the risks posed by the environment. This issue is not addressed here - however the topic of 'trusted area reduction' is returned to briefly in Section 8.)

Typically, as in Figure 1, 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. Figure 2 shows how an application launched by a user in one Trusted Area will in fact be fragmented and replicated so as to use computers in a number of separate, and perhaps individually guarded, areas. The problems of ensuring the security and reliability of the network that interconnects these areas will not be

considered here, though FDP-like approaches, such as in [Koga, Fukushima et al. 1982] for meshed networks interconnection, as well as more conventional solutions, to these problems are quite feasible.

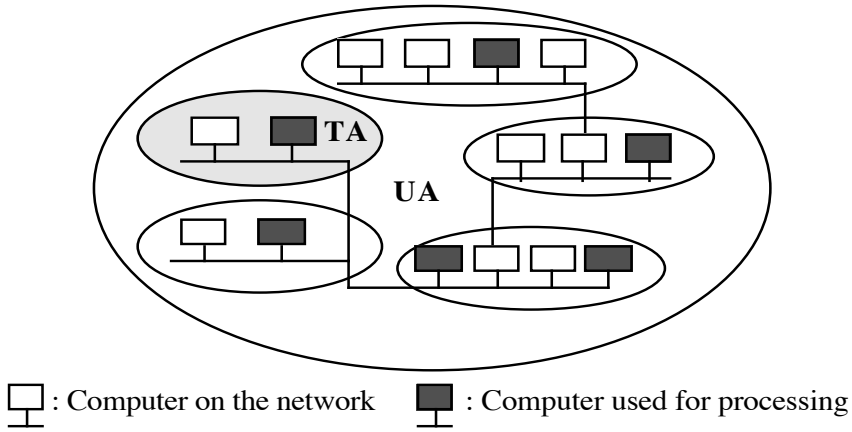


Figure 2: Computers Involved in an Application

To date, the FRS and FDP schemes have been described, and implemented, using conventional system structuring techniques. The principal purpose of this paper is to discuss how FDP can be used in, and can benefit from, an object-oriented model of system structuring. In particular, a first aim is to see whether the model can be used to (re)interpret the classification of fragmentation and scattering mechanisms described by [Fray and Fabre 1989]. The paper then goes on to speculate on possible new ways of using fragmentation and scattering, and to discuss implementation issues briefly. Finally, a brief attempt is made to discuss object-based security in more general terms.

2. Object-Based Structuring

The model used here is one in which we assume, at least initially, that at two (or more) levels of abstraction we have a distributed *complete interpreter*. However, in practice, the upper level is more likely to be just an *interpreter extension* (in the sense of the term as defined by [Anderson and Lee 1981]) - discussion of the ramifications of this point is deferred mainly to Section 2.1 below; and the possibility of there being more than two levels will be disregarded from now on.

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, in fact a directed acyclic graph, (based on the "is-part-of" relation), with objects being composed in part of smaller objects, down to some set of elementary objects. Each object will belong to a class (or type) whose declaration defines the operations (or "methods") available for using objects belonging to that class, these operations providing the only means by which objects can interact directly. (The declaration also defines any other operations, available for internal use, and any internal variables (i.e. objects), by means of which such objects are implemented.)

The example in Figure 3 below, which uses C++ syntax, shows the class declaration for objects whose external interface consists of just two operations (Method1 and Method2), and

which are each composed of three smaller objects, each of class SubObject. There is one operation declared for private use inside NewObjects, and two special operations (NewObject and ~NewObject) which are implicitly invoked, respectively, when a NewObject is created or destroyed.

```
class NewObject  {

      SubObject  InternalObject1;

      SubObject  InternalObject2;

      SubObject  InternalObject3;

//  private:

      InternalOp();

public:

            NewObject();     //  constructor

            ~NewObject();     //  destructor

      Method1();

      Method2();

}
```

Figure 3: A Class Declaration

It is assumed that such class declarations can, when appropriate, inherit definitions from one (or, when multiple inheritance is permitted, more) parent classes. By such means the characteristics, both functional and "non-functional", of the objects of a given class can be declared to possess a modified, typically augmented, set of characteristics, based on those that the parent class defines (or inherits). Class declarations can be thought of as providing information which is used at compile time, for example to provide each object with a reference to, or when appropriate its own copy of, the code of its operations. Alternatively, as in Smalltalk, class declarations can themselves be regarded, and may be implemented in the actual system, as objects (of a special system-defined type called "class"), each containing the code that is used by all the objects of a given class .

On the other hand, the relationship between the object hierarchy at the upper system level and the hierarchy that exist 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, for example having 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.

Method (i) above is exactly what the Arjuna distributed programming system [Shrivastava, Dixon et al. 1991], developed at Newcastle by a team led by Santosh Shrivastava, uses to provide 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 of the use of method (ii) in combination with method (i) would be the use of replication of, and voting by, the processors which implement the Arjuna system, so augmenting at the lower level (by masking arbitrary processor faults) the upper level fault tolerance facilities provided via C++ declarations. This in fact is something that the Arjuna group is investigating using multiple transputers[Ezhilchelvan and Shrivastava 1991].

2.1. Object-based Protection

The strength of any security provisions is limited by the extent to which potentially dangerous actions can be intercepted, checked, and perhaps prevented, by some form of trusted "reference monitor" [Ames, Gasser et al. 1983]. A complete interpreter, by its very nature, can incorporate facilities for monitoring every action that occurs in the system that it is supporting. In contrast, an interpreter extension can of course directly monitor only those actions which it itself implements, and must otherwise rely (even for the protection of its own code and data) on whatever monitoring functions are provided by or via the interpreter on which it is implemented.

Some implementations of object-based programs do much of their checking only at compile time (C++ for instance) - and to this extent provide very limited built-in protection against any hardware faults or software faults which occur (e.g. are surreptitiously introduced) after compilation, perhaps just prior to execution. They thus suffer from a dangerously long TOCTOU (Time of Check to Time of Use). This situation could however be remedied by implementing the programs on some appropriate sort of capability-based computer architecture, so in effect providing a strong guarantee of the validity at run time of the valuable compile-time abstraction provided by the use of an object-based language.

3. Types of FDP

In the paper [Fray and Fabre 1989], which introduced the idea of FDP applied to programs without considering object-oriented structuring, there is a discussion of types of fragmentation, the classification being based on data granularity, four levels being identified: bit-slices, basic variable types, structured variable types and modules/procedures. In effect, these four levels of granularity encompass two rather different types of fragmentation. The two larger

granularity schemes respect and use the structuring of the original program and its code; the other two define and use an alternative to this original structuring.

Users are in general expected to indicate what data and/or code is "confidential", the level of fragmentation to be applied, and which resulting fragments are to be "scattered" to computers in different security areas. These fragments will in general involve one or more data items and its corresponding code segment, and will either be specified explicitly or perhaps with the aid of a preprocessor.

3.1. Classification of FDP techniques

The more detailed, and somewhat different, classification of fragmentation and scattering techniques given by [Trouessin, Fabre et al. 1991], is summarized in the four classes below, in each case first giving the original term for a class, and then the term used in this paper. The techniques are used to produce scattered fragments at different granularity levels:

(i) **Vertical, or Bit-slice fragmentation (B-FDP)**: The bit-slicing technique is applied to 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. This approach is very convenient for arithmetic operations on confidential integer values for instance: the definition of elementary fragments in this way can be seen as a *data-driven* approach.

(ii) **Horizontal, or Structured fragmentation (S-FDP)**: This technique 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. Thus though this technique makes the overall program obscure to an intruder, it does not really attempt to hide what each individual program module does, or what data it does it to. This approach is very convenient for basic building blocks within elementary programs: the definition of elementary fragments in this way can be seen, a priori, as a *code-driven* approach.

(iii) **Transverse, or Parallel fragmentation (P-FDP):** The idea here is to rely on the analogy between "parallelism" and "fragmentation"; ideally, parallel versions of specific algorithms are used to define fragments. Nevertheless, this approach can be seen as derived from the above two techniques, and involves first applying bit-slice fragmentation to the basic data items, and then further fragmenting the resulting program fragments (code and data) horizontally using the programmer-defined module boundaries. (The description given by [Trouessin, Fabre et al. 1991] implies the possibility of using a succession of such bit-slice and structured-oriented fragmentations, essentially recursively, so presumably each successive use of the bit-slice fragmentation involves using narrower "slices". The notion of repeated use of structured-oriented fragmentation presumably involves taking advantage of successively smaller program modules.) This approach relates well to applications which are or could easily be written so as to make explicit use

of parallelism and seems very convenient for matrix computations for instance; the definition of elementary fragments in this way can be seen as both *code-* and *data-driven* .

(iv) **Client-server, or Trusted fragmentation (T-FDP)**: The original term is confusing since the basic execution model for FDP is the client-server model for all types of fragmentation. The T-FDP approach is in fact orthogonal to the above fragmentation techniques since it aimed at satisfying security objectives not only with respect to data items but also to the main body of the application code, whatever the granularity of the fragments. This approach can thus either be applied to elementary operations or to user-defined programs. The approach relies on the "trusted/untrusted" division of the resource space. This technique is described as an extension of either the structured or the parallel fragmentation techniques - so it in fact provides two further classes of technique. Further security is given to the program by arranging that in addition to having its code fragmented and scattered it is also arranged that execution of the main program module is performed by a trusted computer, with only subsidiary modules being spread amongst other (untrusted) computers, and invoked using remote procedure calls from the main module.

In the case of bit-slice fragmentation, sets of corresponding bits from the various data items are held together, along with an appropriately modified version of the complete program. The modifications are to arrange (i) that the right bits are worked on by a given process, and (ii) that each process cooperates appropriately with the processes executing the programs (in other computers) relating to neighbouring bit slices (i.e. those from adjacent bit positions).

In the other cases discussed in [Fray and Fabre 1989] and [Trouessin, Fabre et al. 1991], only the relevant parts of the program, augmented by the necessary code for ensuring cooperation, are held with the set of data fragments allocated to a particular set of computers. Thus performance (and program code security) is enhanced, though data security is diminished, as one moves to higher levels of data granularity.

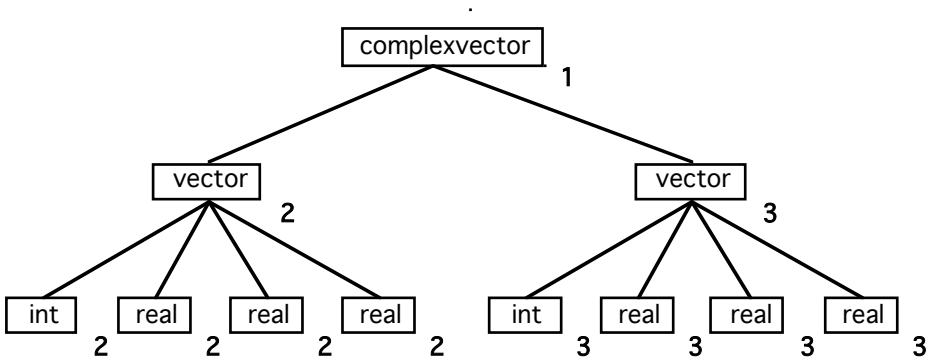
4. Object-Oriented FDP

The object-oriented model would appear to provide a reasonably straightforward method of generalizing and implementing structured fragmentation (S-FDP) and its trusted variant (T-FDP), 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 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. (One could alternatively provide fragmentation and scattering on a per object rather than a per class basis - this point will be returned to briefly in Section 8 below.)

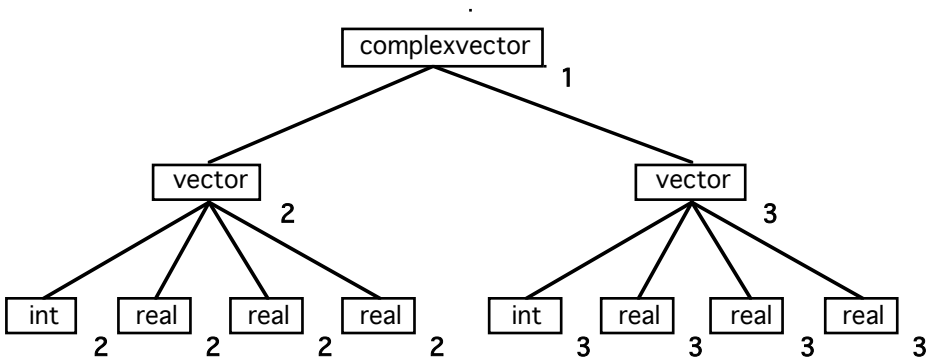
For example, one might have either complexvector or vector (or indeed both) inherit the characteristic "secured". These three possibilities are symbolized in Figure 4, in which the various objects are labelled with numbers indicating the different (sets of) computers they have been allocated to. These 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 figure.

Secured → complexvector



Secured → vector



Secured → complexvector, vector

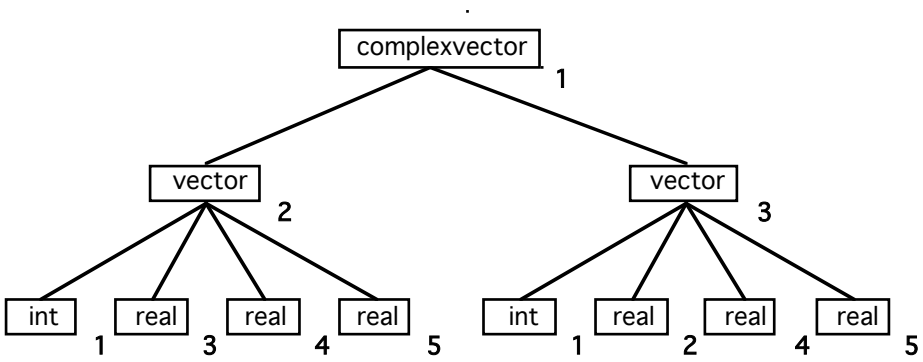


Figure 4: Different Inheritances of the Characteristic "Secured"

Considering implementation issues in slightly more detail, one possible implementation of the characteristic "secured" would involve arranging that for each object of the class all the immediately subsidiary objects are extracted from within the object, and then separately replicated and scattered across the various computers. With the example shown in Figure 3 above, this could mean that InternalObject1, InternalObject2, and InternalObject3 were held on computers separate both from each other and from those holding the object's operation

definitions (see Figure 5). (This leaves open the question of which computer or computers are used to execute any particular operation - the simplest possibility being the one(s) holding the operation definitions.)

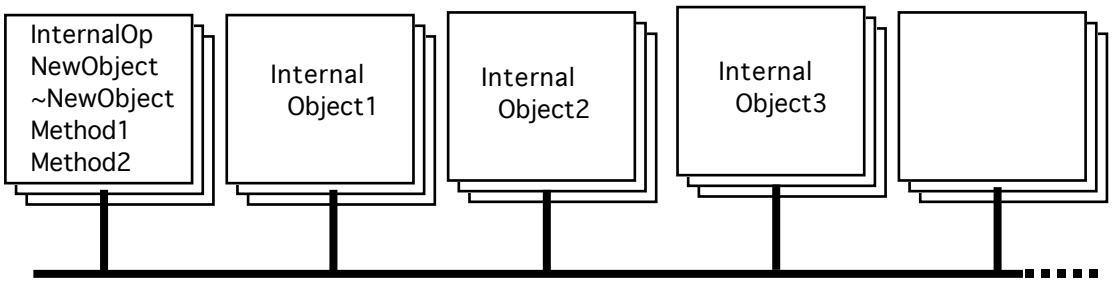


Figure 5: Sub-Object Fragmentation/Redundancy/Scattering

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. (A more extreme form of implementation of such a declaration might involve also somehow fragmenting the code corresponding to the set of operations, perhaps to the extent that InternalOp, NewObject, ~NewObject, Method1 and Method2 were represented (and executed) on several different computers.)

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. [Bal and Tanenbaum 1988]), whose implementation embodies and hides the distribution policies which are in use. Another alternative is the facility provided in SOS at INRIA [Makpangou, Gourhant et al. 1991; Shapiro, Gourhant et al. 1989]), 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 7.4.

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 [Meyer 1987] 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 Section 5 below.)

However such schemes do not directly address one of the techniques discussed in [Fray and Fabre 1989], namely the use of programmer-supplied application-specific "constraints" - these being, in effect, a method of specifying particular objects which it is desired to prevent an individual intruder from correlating, and which must therefore be allocated to different computers. In fact, it appears that the task of re-organizing a program so as to meet such a fragmentation and scattering specification would be very application-specific; it seems to have a lot in common with the (in general difficult) task of detecting potential parallelism in a sequential program and transforming the program (either by hand or automatically) so as to make use of this parallelism.

The basic FDP techniques are of course also related to parallelism issues, though in this case the parallelism which is already may be more or less explicit in the program structure, and hence much more easily identified and exploited. Indeed, 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 that were referred to earlier.

5. Multi-Level FDP

It can be seen that the (two-level) object-based model described in Section 2 above also provides a way of discussing, and generalizing bit-slice fragmentation, and hence the related aspects of B-FDP fragmentation and its T-FDP variant (as discussed in Section 3). This involves viewing them as techniques which are carried out by the underlying interpreter, at whatever granularity is required, from bits to pages.

Bit-slice fragmentation, as originally envisaged, presumes that the required sets of modified programs needed to accompany the data fragments, and to cooperate with each other so as to perform the task of the overall application, can be generated readily, though not automatically, from the original program. However, at least in principle, one could instead provide bit-slice fragmentation simply by implementing the underlying interpreter in a truly object-based language, and employing in it exactly equivalent mechanisms to those we described in Section 4. In other words, one would arrange that those interpreter object types used to contain representations of the code and data of upper-level objects would be constructed from appropriate sub-objects and would inherit the characteristic "secured". (This would obviate any need to modify application programs.) Moreover it makes it clear that the difference between bit-slice (B-FDP) and structured (S-FDP) fragmentation is merely a matter of the level of abstraction at which one is viewing the overall system - and that parallel fragmentation (P-FDP) just involves their joint utilization.

The lower-level (B-FDP) fragmentation and scattering could be implemented and used without explicitly considering the fact that the upper-level interpreter might also be performing fragmentation and scattering (i.e. S-FDP), 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).

In principle, therefore, the provision of sophisticated combined schemes performing what is in effect a generalization of parallel fragmentation (P-FDP), is quite straightforward, though their engineering would need careful consideration. In fact it would appear that the problems of having co-existing application level and interpreter-level fragmentation and scattering, and of arranging that their combination achieves the desired overall effect without excessive performance penalty, are probably analogous to those relating to multi-level recoverability. An analysis of the latter issues is to be found in [Anderson and Lee 1979] and [Anderson and Lee 1981]. (The potential problems, particularly those concerning performance, also resonate faintly with those of recursive virtual machines [Lauer and Wyeth 1976].)

6. Different Views of Object-Oriented FDP

Starting from an object-oriented design of a secure application, the implementation using FDP can be considered from at least two different viewpoints:

The *designer's view* (see Figure 6) consists in defining (i) the various class hierarchies, (ii) identifying the classes of objects which contain/process "confidential" information, (iii) among such classes of objects those that must be executed on the "trusted" resources and those that can take advantage of FDP solutions and therefore be executed on "untrusted" resources.

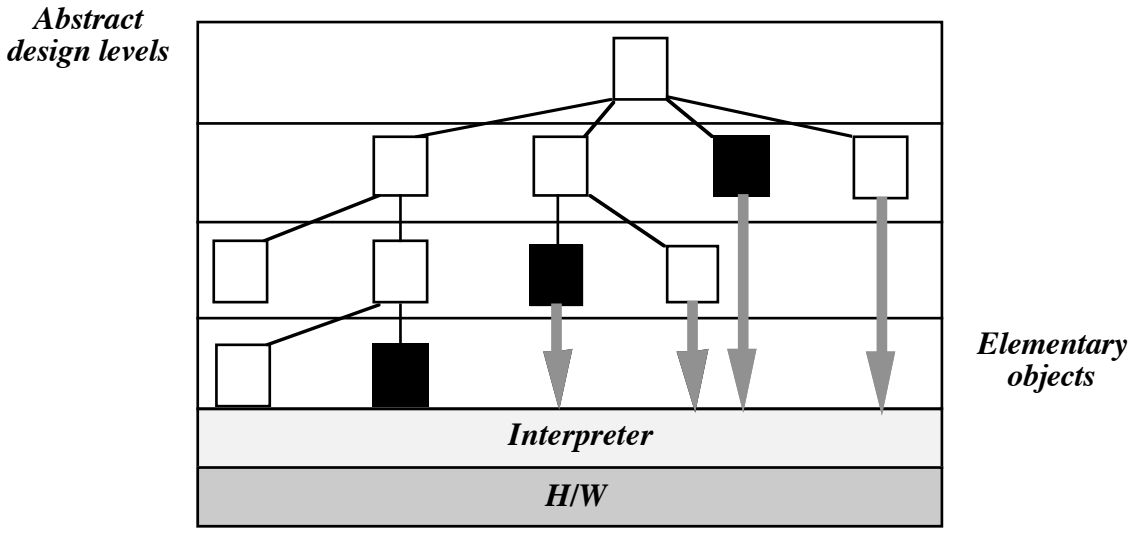


Figure 6: Designer view of FDP

The hierarchy presented in Figure 6 shows one top-level object decomposed into several sub-objects at different abstraction levels and identifies the objects which contain "confidential" information (black boxes) from others that do not have to process sensitive information (white boxes). All the leaves of this hierarchy are "elementary" objects whose processing is supported by the object-oriented runtime and the underlying operating system. This non-balanced tree actually defines different interpreter (some complete, some just interpreter extension) layers of elementary objects.

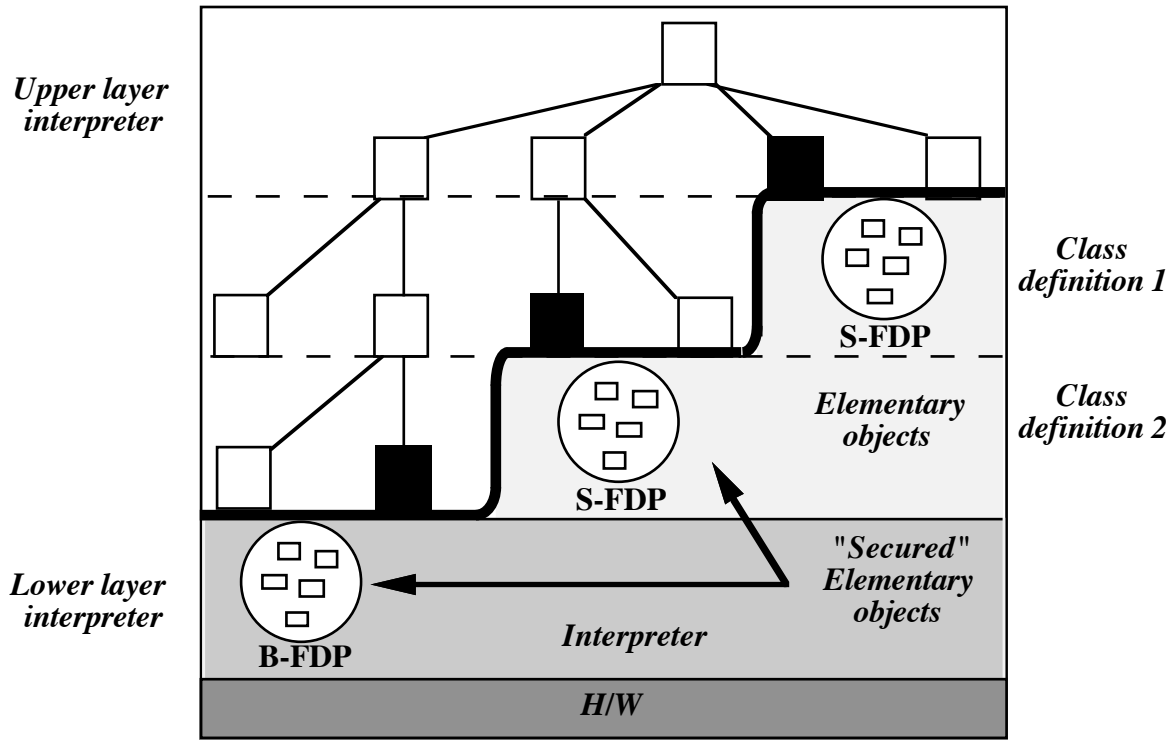


Figure 7: Engineering view - Interpreter layers for "secured" objects

The *engineering view* of the FDP execution (Figure 7) consists in deciding which type of fragmentation has to be used in order to satisfy the security objectives with respect to sensitive objects (black boxes), and also the associated replication strategy

The object-oriented runtime support and the appropriate class definitions provide various FDP solutions - represented here by fragments within circles. The figure re-emphasizes the points made in Section 5 above by showing the possibility of providing what the application programmer will view as S-FDP by defining various appropriate classes, or B-FDP by the underlying run-time system. If both S-FDP and B-FDP are provided simultaneously then one will have what the user will view as P-FDP.

In each case non-sensitive objects (white boxes) are not fragmented, and can be executed as a whole either on "trusted" or "untrusted" resources, taking into account that input-output, at least, operations have to be performed in the "trusted" area. The engineering view of the FDP execution, consists finally in setting up the configuration of the application objects on distributed resources. FDP solutions provided by the lower interpreter layer are transparent to the user and will have been implemented and installed previously. For T-FDP the execution of the corresponding fragments must take into account the user-defined partition "trusted"/"untrusted", since some of the fragments may be required to be executed in the "trusted" area of the current user. (T-FDP is not represented in Figure 7 since it may be used for the implementation of any FDP solution.)

7. An Example

Our example is in fact based on part of the specification of the user authorization service [Blain and Deswarte 1990] [Deswarte, Blain et al. 1991] provided in the DELTA-4 distributed system [Powell, Bonn et al. 1988]. (For the purposes of this example it is assumed that the service merely (i) controls the forms of access rights that already-authenticated users can have wrt a hierarchically organized shared (in fact transparently distributed) archive system, and (ii) enables system administrators to modify these access rights.) The service is provided by an (abstract) single "authorization server". In fact this abstract server is implemented as a distributed set of local servers, each of which is located at a separate security site, where it is managed by a local administrator.

Our example, like the actual DELTA-4 authorization server, is designed to provide confidentiality and freedom from denial of service (wrt archive access control) in the face of possible hardware faults affecting the local authorization servers, and intrusions into the security sites, including accidental or deliberate faults committed by the security administrators. In line with the actual provisions of the DELTA-4 authorization server, two "grades" of confidentiality are provided, one based on protecting the access rights information, and another yet higher grade of confidentiality for the "keys" that this first level of protection depends upon. (These keys are in fact used in DELTA-4 not just for the fragmentation operation on the archives and for naming fragments [Fray, Deswarte et al. 1986], but also for controlling access to other servers - see Figure 8, which is taken from a DELTA-4 presentation.)

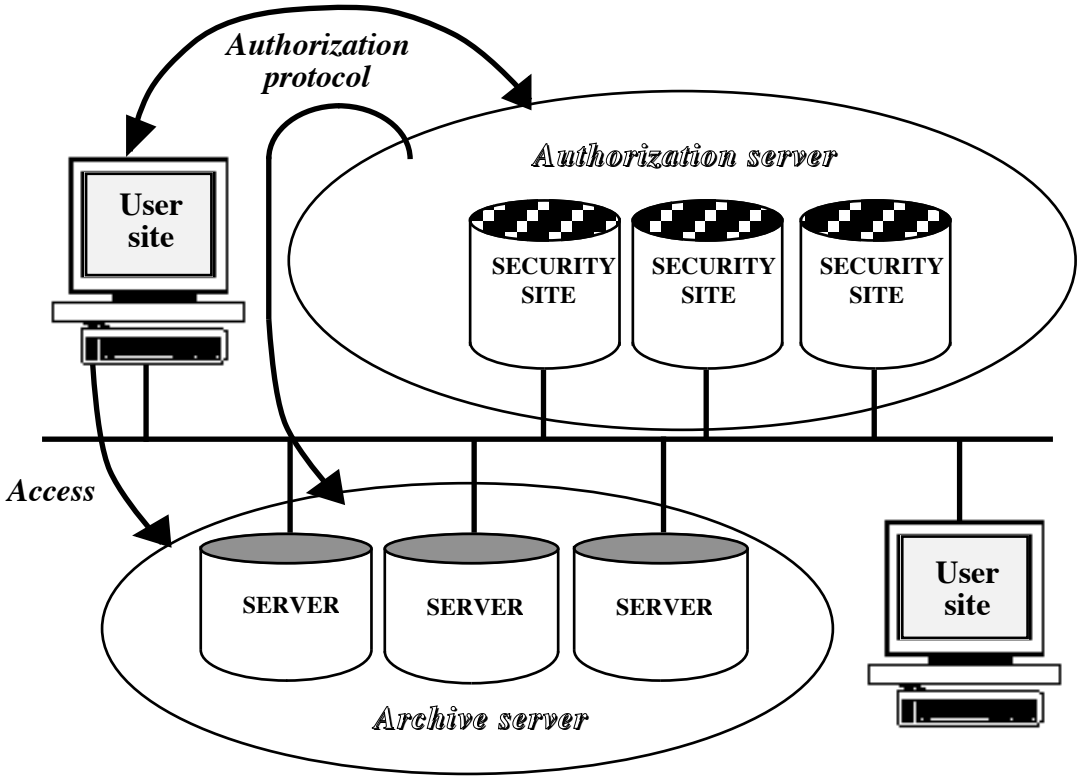


Figure 8: Authorization and Archive servers in DELTA-4

The actual DELTA-4 authorization service was not originally designed or implemented using an explicit object-oriented style. Here, however, we present a small series of classes, which enable us to describe (a simplified version of) the original service. (A new design of the DELTA-4 authorization service using a pure object oriented style and making full use of inheritance is beyond of the scope of this paper.) These classes only address the definition of the management operations of archive descriptor directories which are performed by an abstract directory server implemented by a local directory server on each security site. (In the DELTA-4 implementation, archive descriptor directories are replicated on security sites but not fragmented.) For the moment, we omit from our example any mention of the mechanisms involved in providing or using FDP, deferring such matters until Section 7.2.

The information used for the management of archives is stored in each of a set of archive descriptors and can be summarized as follows:

- *identification:* archive name;
- *fixed access list:* access rights similar to UNIX permissions (r,w,x) and categories (owner, group and others);
- *dynamic access list:* list of entries related to one specific user or group of users and the corresponding permissions.

These descriptors are the leaves of a tree (a subtree) of the Archive Directory Server. Any user in the system can create several archives and thus hold several descriptors; descriptor information is stored within files in his "personal archives directory": */archives/fred/document.a*

is the archive descriptor of the archived document belonging to user fred. A user is also able to store the archive descriptors hierarchically, and thus have several "archive working directories": */archives/fred/nuclear/document.a* is the descriptor of an archived document, presumably of nuclear information, and thus in the sub-directory *nuclear*.

One interesting aspect of this example is that it shows that using the FDP approach, the confidentiality of information such as the relation between directories and descriptors (i.e. subjects and documents) or the relation between descriptors and access rights (i.e. who is able to access what) can be preserved even with respect to administrators of security sites.

In order to make the presentation more clear, access-right management at the directory level will not be considered in detail here; only access-rights within archive descriptors are covered. Nevertheless, the object-oriented access-rights management scheme which is defined in the next section could be used at the directory level.

7.1. Object-Oriented Management

This section presents an object-oriented approach to the management of Directory and Descriptor information; the basic rights checking operations are also taken into account. Some of the object classes (and their component objects) forming the Archive Directory Server object are shown in Figure 9, where an asterisk indicates the possibility of there being several components of a given object class.

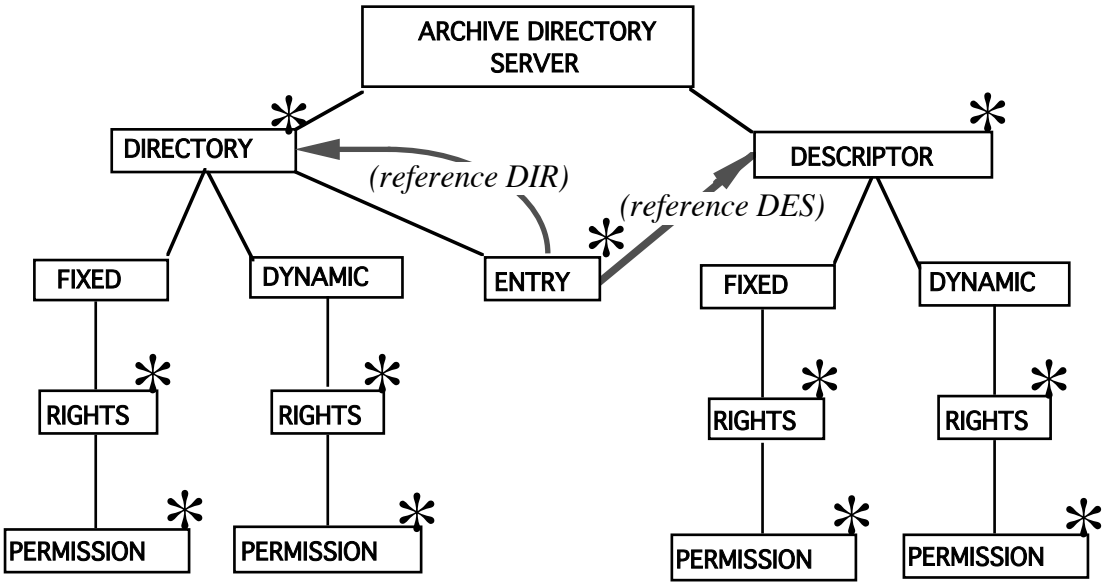


Figure 9: The object composition hierarchy

The object hierarchy represented in Figure 9 for the archive directory service is as follows (the abbreviated names used in the figure are given below in bold caps):

ArchiveDirectoryServer: Implements the operations of the archive directory service; it is partially composed of a set of AuthorizationDirectories (**DIRECTORY**), the whole organized as a tree, and a set of Descriptors which are in effect the leaves of this tree;

The set of available operations on this object are: **Create, Destroy, VerifyRights, NewRights, ModifyRights.**

AuthorizationDirectory: Implements the operations which manage directories; it is composed of set of references (**ENTRY**) to AuthorizationDirectories and ArchiveDescriptors and a set of access rights related to the AuthorizationDirectory; these are represented in one FixedAccessRights list (**FIXED**) and one DynamicAccessRights list (**DYNAMIC**);

The set of available operations on this object are: **Create, Destroy, AddEntry, RemoveEntry, SetEntry, SearchEntry, SetRights, UnsetRights, AddRights, CheckRights.**

ReferenceEntry: Implements the operations which manage references (**ENTRY**) to directories or to descriptors but also to search a given reference;

The set of available operations on this object are: **Create, Destroy, SetEntry, SearchEntry.**

ArchiveDescriptor: Implements the operations performed on the information related to a given archive. It is composed of a set of access rights represented in one FixedAccessRights list (**FIXED**) and one DynamicAccessRights list (**DYNAMIC**).

The set of available operations on this object are: **Create, Destroy, SetRights, UnsetRights, AddRights, CheckRights.**

FixedAccessRights: Implements the operations on access rights similar to Unix rights for owner, group and other users (like the chmod Unix command); it is composed of three Rights objects (**RIGHTS**) OwnerRights, GroupRights and OtherRights;

The set of available operations on this object are: **Create, Destroy, SetRights, UnsetRights, CheckRights.**

DynamicAccessRights: Implements an access control list (ACL) to particular users or groups of users not mentioned in the FixedAccessRights and the corresponding operations; it is composed of a list of Rights objects (**RIGHTS**) associated to users or groups;

The set of available operations on this object are: **Create, Destroy, AddRights, CheckRights.**

ElementaryAccessRights: Implements the access rights of a given owner, a given group or a given user but also access rights for other users, identified by a Name and is composed of a restricted set of two Permissions objects (**PERMISSION**), one for Read access and one for Write access (but could also include other permissions for Append, Delete...);

The set of available operations on this object are: **Create, Destroy, SetName, CheckName.**

ElementaryPermission: Implements a very elementary object, one permission (Read or Write), which is in fact a flag saying if the permission has been set or not; it could be represented on one bit!

The set of available operations on this object are: **Create, Destroy, Setflag, Unsetflag, Checkflag**.

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.

In order to illustrate the usage of such object classes, we briefly present few steps of some management operations on one example (rights checking is not presented here) in which the creation of one directory object and one descriptor object is detailed:

Directory creation:

- A new directory, say *fred*'s *nuclear* directory, is created by invoking the Directory constructor: **Directory (nuclear)**.
- This operation also creates a new entry in *fred*'s "*personal archives directory*", using the AddEntry method with type DIR: **AddEntry (nuclear, DIR)**.
- A new entry structure using the Entry constructor and set the name *nuclear* using the SetEntry method: **SetEntry (nuclear, DIR)**.
- The Fixed and Dynamic constructors are then invoked to create the Fixed and the Dynamic access right list; appropriate rights and permissions are also created.

Descriptor creation:

- A new descriptor, say *document.a*, can be inserted in a given directory (reference), say *nuclear*, using the method AddEntry: **AddEntry (document.a, DES)**.
- The AddEntry method invokes the SetEntry method to assign the EntryName: **SetEntry (document.a, DES)**.
- The Descriptor constructor is invoked to create the descriptor *document.a*: **Descriptor(document.a, fred, staff)**.
- The Fixed and Dynamic constructors are invoked to create the Fixed and the Dynamic access right lists; appropriate rights and permissions are also created.

This hierarchy of constructors and methods initializes the DirectoryName (*nuclear*), ArchiveName (*document.a*), the OwnerName (*fred*) and the GroupName (*staff*). In a real implementation, a descriptor should hold several other objects such as a semaphore and the size of the archive, which are omitted in this simplified example.

The Fixed constructor invokes the Rights constructor three times to create the OwnerRights, GroupRights and OtherRights objects. The Rights constructor invokes Permission to create the ReadPermission and WritePermission objects. The Permission constructor does not set any flags; a flag can be set using the SetFlag public method on the two permission objects ReadPermission and WritePermission.

Finally, the Dynamic constructor is invoked to create and initialize the dynamic access control list using the Rights and Permission constructors. The dynamic access right list is constructed and initialized to empty; new rights can be granted to named users using the AddRights method: **AddRights (user, brian, read)** using **SetFlag()**, for instance.

The above actions are used to create a new descriptor document.a in a new directory nuclear referenced in the */archives/fred* personal directory. Figure 10 shows the object hierarchy created for the *document.a* descriptor; this descriptor is associated with a descriptor entry of type DES in *fred*'s *nuclear* directory:

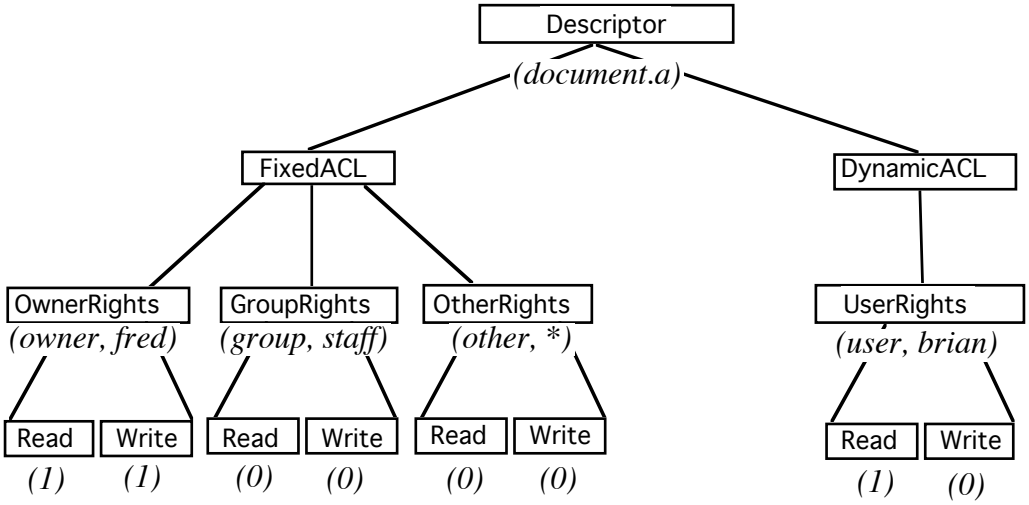


Figure 10: The Resulting DescriptorEntry object

The italic labels with the descriptor objects shown give example values of private variables at each abstract level:

- *document.a* is set by the Descriptor constructor;
- *fred* is set by the SetName method of the Rights object;
- *1* is set by the SetFlag method of the Permission object.

Rights checking is performed using CheckRights, CheckName and CheckFlag methods, but is not detailed in this section.

7.2. Object-Oriented Fragmented Processing

In this section we give some examples of the effects that can be achieved by arranging that the characteristic "secured" be inherited by various object classes:

(i) a directory, say *fred*'s "personal archive directory": */archives/fred*;

(ii) a descriptor, say *document.a*;

(iii) both a fixed and a dynamic access control list, say *FixedACL* and *DynamicACL*;

The effects of using such inheritance are discussed using the above three examples. For clarity in our examples we attach the characteristic "secured" directly to a chosen class or classes

to demonstrate what effect this will have. (In an object-oriented design of the DELTA-4 authorization server that used inheritance amongst its class definitions, one could arrange that the "secured" characteristic was inheritable from higher in the class hierarchy.)

In each of our examples, site number 1 represents the security site where the security administrator is able to execute management operations and where all the archive descriptor directories are located when FDP is not used. (The implementation of the class "secured" and its effects on objects that inherit this characteristic are discussed in Section 7.3.)

7.2.1. "Secured" directories

In this case the characteristic "secured" is attached to the */archives/fred* directory. This solution leads to processing (and perhaps storing) FixedACL, DynamicACL and Entry objects at distinct sites whatever the type of the Entry: DIR or DES. (The fragmentation and scattering of the FixedACL and the DynamicACL objects is discussed in Section 7.2.2.)

Considering just Entry objects, all the directory entries that appear in the */archives/fred* directory (*personal, nuclear, financial...*) will be managed (and perhaps stored) at different sites. All the descriptors entries at this level could also be managed (and perhaps stored) onto different sites, but we do not consider this case at first; we suppose that all the entries at this first level are directories as shown in Figure 11[1]:

Secured → DIRECTORY

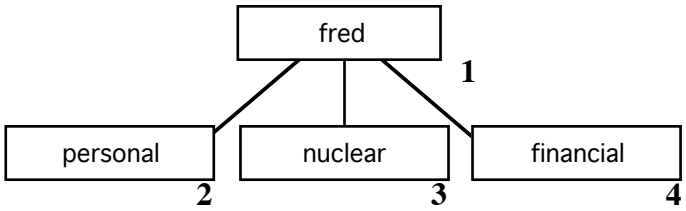


Figure 11: "Secured" directory entries

One possible implementation would thus be to manage and store all subdirectories at different sites. With such a solution a persistent local directory server at each of sites 2, 3, and 4 is responsible for the management of *fred*'s subdirectories. The Archive Directory Server at site 1 is responsible for the management of *fred*'s personal archives directory. An intruder located at site 2 is unable to find out about the various archive subdirectories belonging to user *fred*; the confidentiality of the relation (*fred*'s personal directory, subdirectories) is thus preserved by sites 2, 3 and 4.

We now consider the case of entries being descriptors, when, for instance, the characteristic "secured" is attached to the */archives/fred/nuclear* directory. (For simplicity in our

[1] Figure 11 is in fact a simple representation of a hierarchy of directories. All the Entry objects in figure 11 are references to DIRECTORY objects named by their corresponding DirectoryName.

example all its entries are of type DES.) This solution leads to processing (and perhaps storing) descriptors at different sites (see Figure 12[2]):

Secured → DIRECTORY

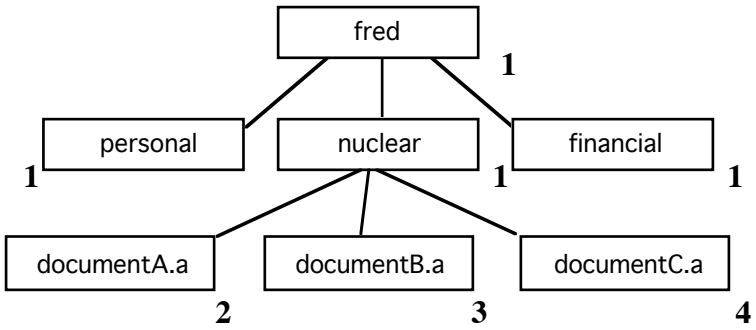


Figure 12: "Secured" descriptor entries

With such a solution, a persistent descriptor server responsible for descriptors management has to be defined by the class Descriptor, and installed at sites 2, 3 and 4. The fact that archive descriptors belong to a given directory (a given subject, in fact) is hidden from intruders at these sites; the confidentiality of the relation (subject, archives) is thus preserved. (Note that if the characteristic "secured" was also attached to */archives/fred* then all the objects in Figure 12 would be located at different sites.)

7.2.2. "Secured" descriptors

In this case the characteristic "secured" is attached to descriptor objects, *document.a* for instance. This solution leads to the fragmentation and scattering of sub-objects FixedACL, KeyFragment and DynamicACL to different sites; Rights methods for OwnerRights, GroupRights and OtherRights objects are executed at the same site, as shown in Figure 13.

In this case an intruder located at site 2 is not able to know who are the users referenced in the DynamicACL. With such object fragmentation and scattering, the confidentiality of the relation (FixedACL, DynamicACL) is also hidden from an intruder; for instance, it is not possible at site 2 to find that, despite no permissions having been given to *fred*'s group (*staff*), *brian* is nevertheless granted read permission for *document.a*.

---

[2] Figure 12 is also a simple representation of a hierarchy of directories and descriptors; the leaves of this tree are Entry objects which are references to DESCRIPTOR objects stamped by their corresponding document name.

Secured → DESCRIPTOR

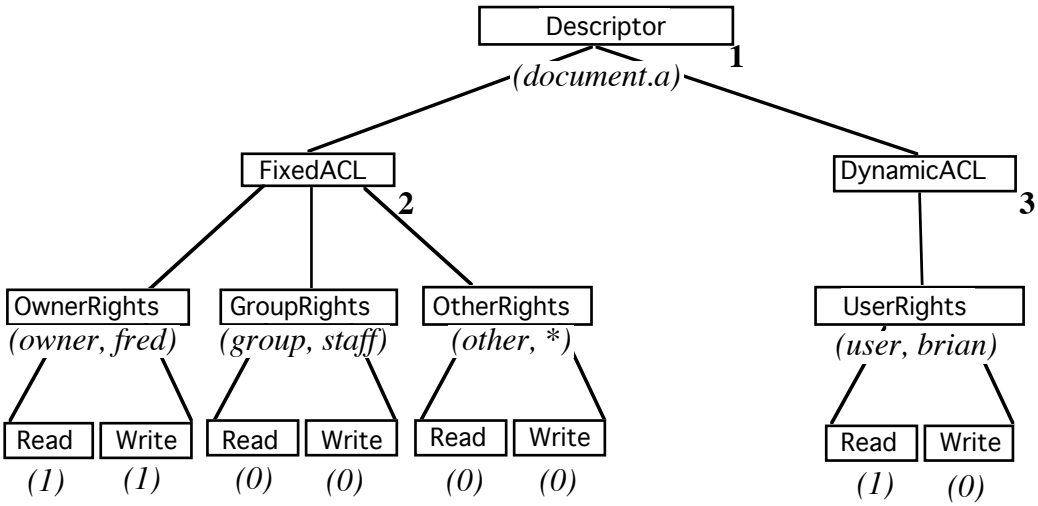


Figure 13: "Secured" descriptors

One possible implementation of descriptors derived from this solution would be to manage and store, for instance, the FixedACL objects of any of *fred*'s document at a given site, say 2.

7.2.3. "Secured" ACL

In this case the characteristic "secured" is attached to the access control list FixedACL, *document.a* for instance. This solution leads to fragmented and scattered Rights objects belonging to the FixedACL object being placed at different sites as shown in Figure 14:

Secured → FIXED

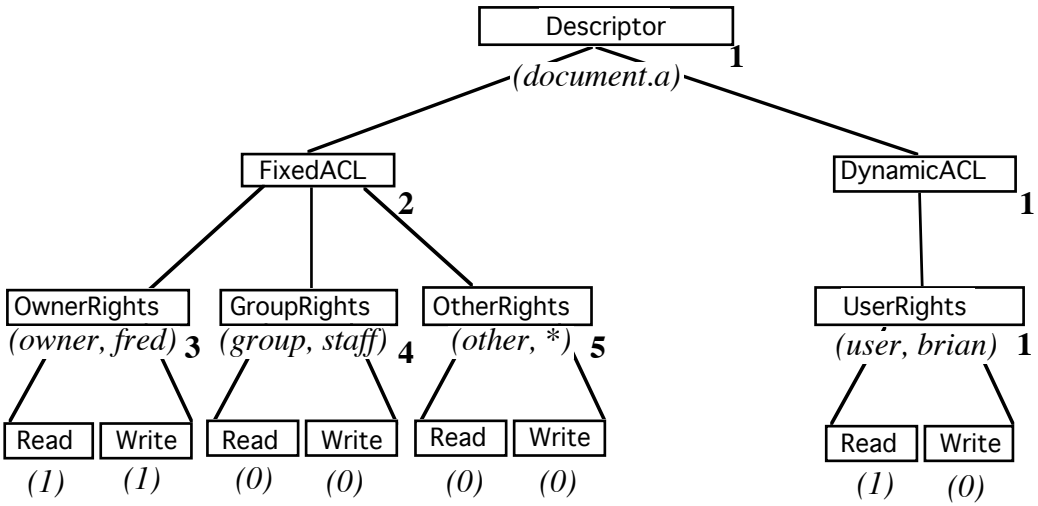


Figure 14: "Secured" FixedACL

Fragmentation and scattering could also be applied to the Rights objects, but these are very simple so this use would in most circumstances be rather unrealistic, but nevertheless is illustrative of what can be achieved. This could imply that, for instance, OwnerRights for

document.a and any other Rights object that appears in the FixedACL are all held at site 5, say. However another possible implementation of Rights objects could be such that the OwnerRights, GroupRights and OtherRights objects related to a given user's documents are processed and stored at different sites.

7.2.4. "Secured" ACL and Descriptors

For example, the characteristic "secured" could be attached to the access control lists FixedACL (of *document.a* for instance) but also inherited by the descriptor object as in Figure 15. This solution leads to two phenomena:

1) Rights objects in the FixedACL are fragmented and scattered onto different sites (see Section 7.2.3);

2) Use of the characteristic "secured" at the descriptor level, leads to the FixedACL and DynamicACL objects being fragmented and scattered in the same way as in Figure 13 (see Section 7.2.2).

Secured → DESCRIPTOR, FIXED

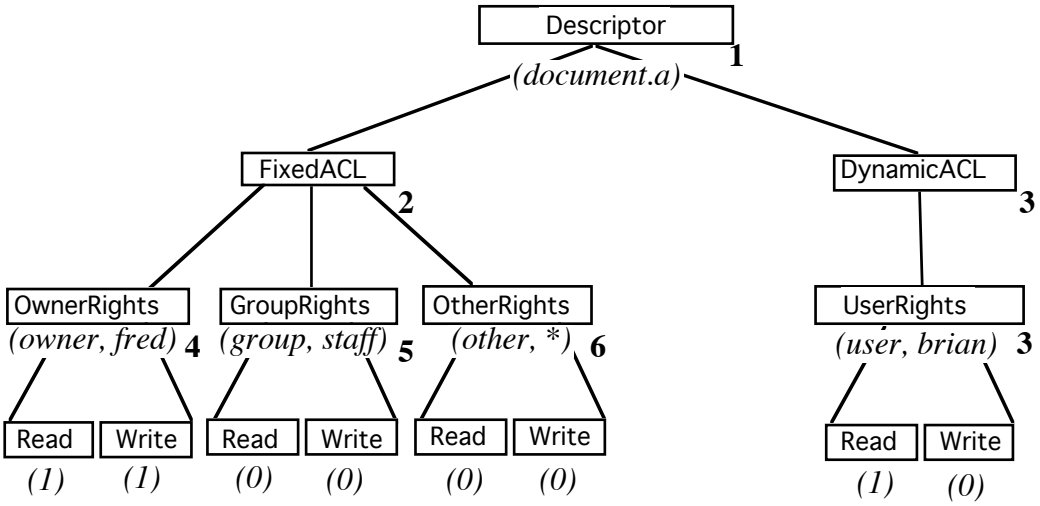


Figure 15: "Secured" ACL with inheritance

In contrast to the situation shown in Figure 15, another possibility would be to attach the characteristic "secured" to both the FixedACL and the Dynamic ACL object classes. This solution provides a complete fragmentation and scattering of Rights objects whether they belong to the Fixed or Dynamic ACL. The latter would provide to an intruder a partial view on the descriptor object, similar to the notion of views used in multilevel security in object-oriented database systems [Lunt 1989].

The result of this last solution leads to the definition of one possible implementation of Descriptors:

1) OwnerRights for all of *fred*'s archives are stored at one site where the management of all of *fred*'s document ownership can take place - presumably a workstation that *fred* trusts; the GroupRights and OtherRights for all *fred*'s archives could also be managed this way.

2) UserRights in the DynamicACL could also be managed in this way and thus *brian*'s UserRights wrt *fred*'s *document.a* would be managed on a workstation trusted by *brian*.

Finally, we would like to recall that using fragmentation and scattering at the directory level may lead to user directories being organized in terms of users or at the level of subjects. Thus using all the fragmentation and scattering strategies presented in Section 7.2 leads to all information managed by the directory server being distributed among several sites according to whatever confidentiality requirements are given.

7.3. Implementing the "Secured" characteristics

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. Object-oriented replication techniques have not been considered in this paper since this topic has already been investigated by others, in the Arjuna project for instance. According to the example given in the previous section and the discussion in Section 7.2.3, replication of the Rights sub-objects needs definition of an appropriate set of workstations in *fred*'s trusted area to store, for instance, replicates of Rights objects.

In Arjuna 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 is invoked for each "secured" object (sub-object).

Finally, different replication and/or fragmentation and scattering strategies can also be implemented and selected from amongst the schemes which have been described in earlier sections.

7.4. "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 [Shamir 1979] to make the management of the keys it uses for its scheme of FDP intrusion-tolerant (and, as mentioned earlier, also for other purposes). Such a scheme represents the value of an object which is to be kept secret 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, 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 an object by means of a threshold scheme thus can ensure the continued availability of the object, despite the occurrence of less that N-T faults, N being the number of shadows generated, as well as confidentiality. This scheme is therefore analogous to the use of

both fragmentation/scattering and replication. Thus in an object-oriented system it could be provided, for inheritance by chosen object classes, via a class declaration which constituted an alternative to the pair of classes SecurityBase and ReplicationBase discussed in Section 7.3 above, and which could itself use a class "key" implementing the shadowing scheme.

7.5. Commentary on the Example

The example presented in Section 7 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 the Archive Directory Server. Nevertheless, it gives the flavour of several different strategies for ensuring the confidentiality and reliability of directories, archive descriptors and access rights; for instance, directories may be stored by directory objects on a user-by-user basis locally on one given site, access rights may be stored on a document-by-document basis with only the associated owner rights, group rights or other rights. More importantly, the actual implementation differences between these strategies are quite minor, since they only involve differing decisions regarding inheritance.

Our example has however only explicitly demonstrated what we have termed S-FDP techniques. However, one can readily imagine some such system as we have been discussing being extended so as to form a complete interpreter for some language (say of authorization). With respect to programs written in this language, the fragmentation and scattering techniques we have described would be classed as B-FDP. And if fragmentation and scattering was also provided within a new language the two mechanisms together would constitute P-FDP. Finally, the use of T-FDP on a given application would involve arranging to hold the main body of the application on the user's trusted workstation.

8. Other Forms of Object-Oriented Security

The principal, and in retrospect obvious, point made in this paper is that, with the use of appropriate class declarations, security (together with reliability and availability) could be provided as an inheritable characteristic in an object-oriented system - an idea that has recently also been explored in the database community. (Though we have concentrated on its provision on a per class basis, as being most transparent to application programmers, one could alternatively provide it on a per object basis; this would of course require that the user indicate, when requesting creation of an object, whether or not it is to be fragmented and scattered.) In the above we have concentrated on one particular means (FDP) of implementing one particular form of security (data confidentiality and integrity) together with facilities for hardware and software reliability. It is interesting to explore what other methods of implementation, and other forms, of security could beneficially be allied with object-oriented structuring

One very simple implementation of the inheritable characteristic "secure" could be merely that of ensuring that the data items stored in such objects are normally held in encrypted form. This would provide a modest but in some cases useful degree of security (in the limited sense of confidentiality) even within a single computer system, and even against intruders who managed to by-pass whatever storage protection mechanisms were provided - as long as they cannot obtain the decryption key, or manage to access the data at any times when it is in

unencrypted form. It thus would provide little protection against a serious attacker, but in certain circumstances could have merit.

Regarding the FDP implementation, an underlying capability-based computer architecture would almost certainly have to be used to enforce the object structuring if one wished to admit the possibility of the user workstations suffering intrusions, or the software in them not being entirely trusted, yet nevertheless still wished to rely on them to control the use of FDP (and, for example, contain fragmentation keys). However, in such circumstances, the use of a capability architecture, though necessary, would almost certainly be insufficient. Rather, one would also need (i) any security-critical software to be thoroughly validated, since the capability mechanism checks only the legality of programs, not their correctness, and (ii) administrative or mechanical means of physically protecting the workstation, to ensure that what is being executed is what was validated. In effect, this is going down the difficult and lengthy road of developing a general purpose operating system security kernel, whether entirely in software, or (perhaps preferably) based on a powerful hardware-based reference monitor [Sami Saydjari, Beckman et al. 1987; Sami Saydjari, Beckman et al. 1989]). However, if this were feasible and if the resulting kernel were deployed throughout the distributed system, there would be little need for FDP, at least with regard to the provision of confidentiality, though the problems of denial of service, and indeed of reliability in general would remain.

An alternative strategy, again avoiding having to trust the workstations, would be to interpose Trusted Network Interface Units (TNIUs) between the workstations and their means of communicating with the rest of the system, as is done in the Distributed Secure System (DSS) [Rushby and Randell 1983] [Barnes and MacDonald 1986] [Bates 1991]. These TNIUs could then be used to control FDP, rather than (as in DSS) to provide conventional encryption. This seems quite feasible, as such TNIUs would still be comparatively simple, though they would presumably require physical protection. (In terms of the obviously valuable goal of 'Trusted Area Reduction', the individual TNIUs are much 'smaller' than a typical workstation, but the set of TNIUs are likely to be spread over a considerable geographical area, outside the control of an individual user.)

Turning to issues of security policy, the FDP technique concerns the provision of just a simple policy, namely separation and the prevention of denial of service. However, as with the basic DSS concept, such separation provides a good foundation on which one could construct more sophisticated mandatory (e.g. multi-level) and/or discretionary security policies (as in fact have already been explored in the object-oriented database arena, e.g. [Fernandez, Gudes et al. 1989; Lunt 1989; Lunt 1990]). In the case of DSS further security policies have been implemented either within the TNIUs, or by means of specialized servers, but in either case involve trusted code. With FDP it could either be within the workstations (with all the security concerns that this implies) or in specialized servers (which could themselves make use of FDP, but nevertheless their code would have to be trustworthy). However, for both FDP and DSS the trustworthiness of the code could be based either on fault prevention techniques, such as validation, or fault masking techniques based on the use of design diversity.

9. 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, thus providing protection at various granularity levels. Moreover we have shown how this approach provides a unifying framework for various apparently rather distinct FDP techniques, herein termed S-FDP (Structured FDP), B-FDP (Bit-sliced FDP) and P-FDP (Parallel FDP).

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 object-oriented FDP, 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.

10. Acknowledgements

The preparation of this paper has been greatly assisted by discussions in particular with Dan McCue, but also with numerous other colleagues, including Yves Deswarte, Jean-Claude Laprie, Peter Neumann, Mark Little and Gilles Trouessin. 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 BR to spend a period at LAAS.

11. References

[Ames, Gasser et al. 1983] S.R. Ames Jr., M. Gasser and R.R. Schell, "Security Kernel Design and Implementation: An introduction," *Computer*, vol. 16, no. 7, pp.14-22, 1983.

[Anderson and Lee 1979] T. Anderson and P.A. Lee. "The Provision of Recoverable Interfaces," in *Proc. 9th Int. Symp. on Fault-Tolerant Computing (FTCS-9),* pp. 87-94, Madison, IEEE, 1979.

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

[Bal and Tanenbaum 1988] E.H. Bal and A.S. Tanenbaum. "Distributed programming with shared data," in *Proc. of the ICCL,* pp. 82-91, Miami, FL, IEEE, Computer Society Press, 1988.

[Barnes and MacDonald 1986] D. Barnes and R. MacDonald, "A Practical Distributed Secure System," *J. IERE*, vol. 56, no. 5, pp.192-196, 1986.

[Bates 1991] A.S. Bates. "Distributed Secure Systems," in *Proc. DECUS 91,* University of Warwick, UK, 1991.

[Blain and Deswarte 1990] 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),* pp. 97-104, Toulouse (F), AFCET, ISBN 2-9036778-9, 1990.

[Deswarte, Blain et al. 1991] Y. Deswarte, L. Blain and J.-C. Fabre. "Intrusion Tolerance in Distributed Computing Systems," in *Proc. IEEE Symp. on Security and Privacy,* Oakland CA, USA, 1991.

[Ezhilchelvan and Shrivastava 1991] 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,* pp. 36-48, Tucson, AZ, 1991.

[Fernandez, Gudes et al. 1989] E.B. Fernandez, E. Gudes and H. Song. "A Security Model for Object-Oriented Databases," in *Proc. IEEE Symp. on Security and Privacy,* pp. 110-115, Oakland CA, USA, 1989.

[Fray, Deswarte et al. 1986] J.-M. Fray, Y. Deswarte and D. Powell. "Intrusion Tolerance Using Fine-Grain Fragmentation-Scattering," in *Proc. IEEE Symp. on Security and Privacy,* pp. 194-201, Oakland CA, USA, IEEE, 1986.

[Fray and Fabre 1989] 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,* pp. 131-137, Santa Barbara, California, 1989.

[Fray, Deswarte et al. 1986] J.M. Fray, Y. Deswarte and D. Powell. "A Secure Distributed File System based on Intrusion Tolerance," in *Proc. IFIP SEC/86, 4th Int. Conf. Computer Security,* pp. 407-418, Monte-Carlo, France, 1986.

[Koga, Fukushima et al. 1982] 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, 1982.

[Lauer and Wyeth 1976] H.C. Lauer and D. Wyeth. "A Recursive Virtual Machine Architecture," in *Proceedings ACM Workshop on Virtual Computer Systems,* pp. 113-116, Cambridge, Massachusetts, 1976.

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

[Lunt 1990] T.F. Lunt. *Discretionary Security for Object-Oriented Database Systems,* Final Report, SRI Project 7543, SRI International, Menlo Park CA, USA, 1990.

[Makpangou, Gourhant et al. 1991] M. Makpangou, Y. Gourhant, J.-P.L. Narzul and M. Shapiro. *Structuring Distributed Applications as Fragmented Objects,* Research Report 1404, INRIA, Rocquencourt, France, 1991.

[Meyer 1987] B. Meyer, “Eiffel: Programming for Reusability and Extendibility,” *ACM SIGPLAN*, vol. 22, no. 2, pp.85-94, 1987.

[Powell, Bonn et al. 1988] D. Powell, G. Bonn, D. Seaton, P. Verissimo and F.Waeselynck. “The Delta4 approach to dependability in open distributed computing systems,” in *Proc. of the 18th International Symposium on Fault-Tolerant Computing (FTCS-18),* pp. 246-251, Tokyo, IEEE, 1988.

[Rushby and Randell 1983] J.M. Rushby and B. Randell, “A Distributed Secure System,” *Computer*, vol. 16, no. 7, pp.55-67, 1983.

[Sami Saydjari, Beckman et al. 1987] O. Sami Saydjari, J.M. Beckman and J.R. Leaman. “Locking Computers Securely,” in *Proc. 10th Nat. Computer Security Conf.,* pp. 129-141, Baltimore MD, USA, 1987.

[Sami Saydjari, Beckman et al. 1989] O. Sami Saydjari, J.M. Beckman and J.R. Leaman. “LOCK Trek: Navigating uncharted space,” in *Proc. IEEE Symp. on Security and Privacy,* pp. 167-175, Oakland CA, USA, 1989.

[Shamir 1979] A. Shamir, “How to Share a Secret,” *Comm. ACM*, vol. 22, no. 11, pp.612-613, 1979.

[Shapiro, Gourhant et al. 1989] M. Shapiro, Y. Gourhant, S. Halbert, L. Mosseri, M. Ruffin and C. Valot. *SOS: An Object-Oriented Operating System - Assessment and perspective,* INRIA, 1989.

[Shrivastava, Dixon et al. 1991] S.K. Shrivastava, G.N. Dixon and G.D.Parrington, "An Overview of the Arjuna Distributed Programming System" *IEEE Software*, vol. 8, n°. 1, pp.66-73, 1991.

[Trouessin, Fabre et al. 1991] G. Trouessin, J.-C. Fabre and Y. Deswarte. “Reliable Processing of Confidential Information,” *in proc. of IFIP/Sec,* Brighton, UK, pp. 210-221, May 1991.

APPENDIX

The various classes given below are presented in a C++ like formalism, but they are not completely defined and implementation details have not been investigated in detail. Nevertheless, some additional comments are necessary:

1) Most of the variable instances are defined statically (e.g. using fixed size arrays of pointers); implementation details on the use of files or more sophisticated data structures representing lists of elements are not of interest in this example.

2) DIR and DES are pre-defined constants for a directory entry type and a descriptor entry type, respectively (cf. Class Directory and Entry).

3) "owner", "group", "other" and "user" are pre-defined constants for the owner rights, group rights, other rights and user rights, respectively(cf. Class Fixed, Dynamic and Rights)

4) "read" or "write" are pre-defined constants for read and write pennissions, respectively (cf. Class Rights and Pennission).

4) "ok" or "null" are pre-defined constants for setting (unsetting) pennissions (cf. Class Pennission).

6) The C++ functions presented in Table 1 and given in appendix A are member functions; a member function F belonging to the object 0 is invoked O.F, providing the object oriented flavour of C++. Such functions always receive an implicit argument which identifies the object performing the functions.

Class SERVER definition:

```
Class       Server      {
        Directory*      DirectorySet [1000];
        Descriptor*     DescriptorSet [1000];
public:
        Server (char*);          // constructor
            // (directory name)
        ~Server (char*);         // destructor
void VerifyRights (char*, int, char*, int, int);
            // (reference, owner/group/other, name, read, write)
void NewRights (char*, int, int, char*, int, int);
            // (reference, type, owner/group/other, name, read, write)
void ModifyRights (char*, int, char*, int, int);
            // (reference, owner/group/other, name, read, write)

// Only few methods have been defined for this class and thus the list is
// not exhaustive; several other appropriate methods should be added here.
}
```

Class DIRECTORY definition:

```
class    Directory {
            char        DirectoryName [20];
            Entry*      EntryTable [100]        // table of 100 entries
            Fixed;      FixedACL;               // Fixed Access Control List
            Dynamic     DynamicACL;
public:
               Directory (char*);   // constructor
                     // (directory name)
               ~Directory (char*); // destructor
void AddEntry (char*,int);      // create a new entry in a directory
            // (entry name, type of entry DIR or DES)
void RemoveEntry ();
}
```

Class ENTRY definition:

```
Class    Entry {
         char     EntryName[20];         // reference name
         int      EntryType;             // type of entry DIR or DES
public:
         Entry () ;          // constructor
         ~Entry ();                // destructor
void SetEntry (char*,int);      // Initializes Entry structure
            // (entry name, type of entry)
void SearchEntry (char*, int) ;
            // (entry name, type of entry)
}
```

Class DESCRIPTOR definition

```
class  Descriptor {
        char            ArchiveName [20];
        Fixed           FixedACL;          // Fixed Access Control List
        Dynamic         DynamicACL;
public:
        Descriptor (char*, char*, char*);  // constructor
                // (entry name, owner name, group name)
        ~Descriptor ();                          // destructor
}
```

Class FIXED definition:

```
class   Fixed {
        Rights (owner)      RightsOwner;    // define OwnerRights object
        Rights (group)      RightsGroup;    // define GroupRights object
        Rights (other)      RightsOther;    // define OtherRights object
public:
        Fixed () ;          // constructor
        ~Fixed () ;         // destructor
void SetRights (int, int);
                // (owner/group/other, read/write)
void UnsetRights (int, int);
void CheckRights(int, int);
}
```

Class RIGHTS definition

```
class       Rights {
        int  Type;              // type reference for owner, group, other, user
        char    Name [ 20] ;  // owner name, group name or user name
        Permission PermissionRead;      // Read object
        Permission PermissionWrite;     // Write object

                Rights (int);    // constructor
                ~Rights ();      // destructor
void SetName (char*);          // set Name as owner, group or user
                // (owner name, group name, * or user name)
void CheckName (char*);     // check if it is owner, group or user
                // (owner name, group name, * or user name)
}
```

Class PERMISSION definition

```
class       Permission {
        int      Flag;

public:
        Permission ();          // constructor
        ~Permission ();         // destructor
        SetFlag (int) ;
                // ("ok" or "null" value)
        UnSetFlag () ;
        CheckFlag () ;
}
```

Class DYNAMIC definition

```
class  Dynamic {
        Rights* RightsTable [50]; // list of users or groups
public:
        Dynamic ();             // constructor
        ~Dynamic ();            // destructor
void AddRights (int, char*, int);       // creates new Rights
    // (type user or group, user name or group name, read or write)
void CheckRights (int, char*, int);     // check user or group rights
    // (type user or group, user name or group name, read or write)
}
```

Class KEY definition

```
class  key {
    int     Key;    // Fragmentation key
public:
    key ();         // constructor
    ~key ();        // destructor
void SetKey(int);
    // (key value K of the fragmentation key)
void GetKey (int);
}
```