U.S. DEPARTMENT OF COMMERCE / National Bureau of Standards

Software Engineering Structures for Support of Secure and Reliable Software

Operating System Structures to Support Security and Reliable Software

Theodore A. Linden

Institute for Computer Sciences and Technology
National Bureau of Standards
Washington, D. C. 20234

U.S. DEPARTMENT OF COMMERCE, Elliot L. Richardson, *Secretary*

Edward O. Vetter, *Under Secretary*

Dr. Betsy Ancker-Johnson, *Assistant Secretary for Science and Technology*

NATIONAL BUREAU OF STANDARDS, Ernest Ambler, *Acting Director*

Issued August 1976

National Bureau of Standards Technical Note 919

Nat. Bur. Stand. (U.S.), Tech. Note 919, 51 pages (Aug. 1976)

CODEN: NBTNAE

U.S. GOVERNMENT PRINTING OFFICE
WASHINGTON: 1976

For sale by the Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402
(Order by SD Catalog No. C13.46:919). Stock No. 003-003-01658-6 Price $1.25
(Add 25 percent additional for other than U.S. mailing).

TABLE OF CONTENTS

OPERATING SYSTEM STRUCTURES TO SUPPORT SECURITY AND RELIABLE SOFTWARE

Theodore A. Linden

Security has become an important and challenging goal in the design of computer systems. This survey focuses on two system structuring concepts that support security; namely, small protection domains and extended-type objects. These two concepts are especially promising because they also support reliable software by encouraging and enforcing highly modular software structures--in both systems software and in applications programs. Small protection domains allow each subunit or module of a program to be executed in a restricted environment that can prevent unanticipated or undesirable actions by that module. Extended-type objects provide a vehicle for data abstraction by allowing objects of new types to be manipulated in terms of operations that are natural for these objects. This provides a way to extend system protection features so that protection can be enforced in terms of applications-oriented operations on objects. This survey also explains one approach toward implementing these concepts thoroughly and efficiently--an approach based on the concept of capabilities incorporated into the addressing structure of the computer. Capability-based addressing is seen as a practical way to support future requirements for security and reliable software without sacrificing requirements for performance, flexibility, and sharing.

Key Words and Phrases: Capability, capability-based addressing, computer security, extended-type objects, operating system structures, protection, reliable software, reliability, security, small protection domains, types.

1. INTRODUCTION

For the year 1974, one source has identified 339 cases of computer-related crime. 1/ The average loss in the 339 incidents was $544,000. This average is not distorted by a few exceptional cases--the median loss was very close to the average. Most of the incidents involved simple fraud by an employee who had access to computerized financial records. In 85% of the cases, management did not report the incident to the police--often because publicity about it would have been embarrassing.

The fraud is usually possible because of some oversight in an applications system. A simple oversight, for example, may allow a clerk to feed data to an accounts payable system in such a way that no one notices when checks are diverted to a dummy corporation.

If the amount of computer-related fraud is to be controlled, then it is necessary to automate the concepts of segregated duties, independent checking, and accountability for actions that are typical in manual accounting systems. These concepts are often much less rigorously applied when financial records are computerized. While the structure of current computer systems may not be to blame for this neglect of sound accounting practices, current operating systems do little to encourage the segregation and independence that is desirable when processing financial records. New operating system structures could make it much easier and less expensive to enforce these basic principles of sound accounting practice. Furthermore, while current instances of computer-related fraud have not exploited security weaknesses in the underlying operating system, it is well-known that such weaknesses exist and that a programmer could exploit them to bypass the controls in applications programs. Thus improvements to security that do not consider the security of the underlying operating system may only deter the small-time

1/ This information is based on conversations with Robert Courtney. Courtney reports that details on these cases are in his possession, but that they cannot be made public. The work of [Parker 75], based on public reports, supports a similar conclusion about the average loss in computer-related crime.

criminals. The increasing amount of valuable and private information processed by computers implies a long-term need for much more rigorous security controls in the operating system. Those responsible for protecting information affecting the National Defense have been facing this problem for some time.

1.1 Security and Reliability

In the attempt to design computer systems that support more rigorous security, a narrow focus on the security problem alone is not advisable. While the cost of inadequate security controls may be several hundred million dollars a year 2/, these costs are only a small fraction of the total costs attributable to faulty and unreliable software. Furthermore, from the viewpoint of computer design, a technical breakthrough on both the security and the software reliability problems appears to be as feasible as a breakthrough on the security problem alone. While we are striving for secure computers, we should also strive for more reliable computers and for computers that make it easier to implement reliable programs--including but not limited to, the programs that do accounting and auditing for security.

Many security controls might not be cost-effective if similar controls were not also needed to improve the reliability and the overall performance of the system. In particular:

- The complexity and disorganization of most existing operating systems make it very difficult to achieve security. To guarantee security--and especially to maintain security over the lifetime of the system--operating systems must be structured so that interactions between system modules are more clearly defined and more closely controlled. This same control over the interaction of modules is also needed for reliability. Furthermore, a well-structured system is easier to maintain and modify; and in a well-structured system it is likely that overall performance can be improved.

- The protection mechanisms needed for security can also be used to enforce software modularity. Such modularity would improve the reliability and correctness of the software. In particular, debugging and testing would be easier to the extent that the effects of an error can be confined within the module where the error occurs. Since debugging and testing often account for half of a program's cost, these protection mechanisms might help reduce programming costs.

- In some applications a system crash is a security problem. In any case, an operating system that is built to provide security must eliminate most of the sources of software-induced system crashes. Furthermore, hardware malfunction and inadequate fault recovery strategies are potential sources of many forms of security violations. Thus, there is enough overlap between the requirements for security and the requirements for high system availability so that it is reasonable to attempt to solve both problems at the same time.

1.2 Overview

It is an ambitious goal to design a computer system that satisfies rigorous security requirements, supports reliable software and at the same time meets the performance, flexibility, sharing, and compatibility requirements that are needed to make a computer competitive in the marketplace. Decreasing hardware costs are making these goals much more feasible. This survey focuses on two system structuring concepts that promise to help solve some of the remaining software problems. These two concepts are identified as:

2/ The cost of the frauds identified by Courtney was almost $200 million for the one year. The total cost of all computer-related fraud may be far higher. Furthermore, the increased computer processing costs needed to protect classified defense information has been estimated at $100 million a year [Anderson 72].

(1) small protection domains, and
(2) extended-type objects.

The survey also covers capability-based addressing as a way of implementing these two concepts.

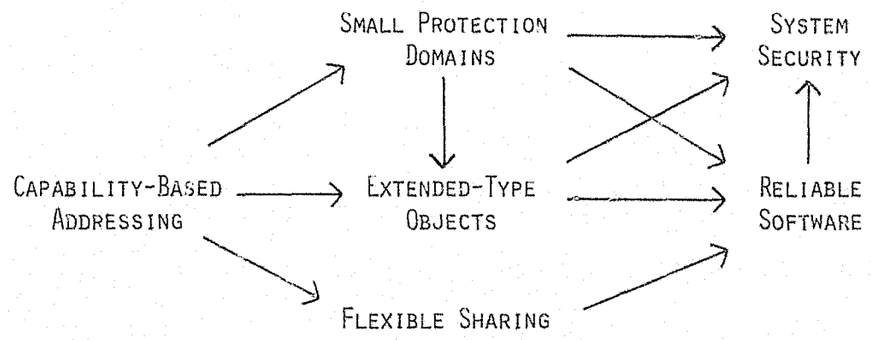

Figure 1 - Overview

Figure 1 shows the interactions between the principal ideas covered in this survey. Arrows between terms in the figure are to be read as meaning "supports" or "facilitates." (Definitions of the terms are given in Section 1.3.) Figure 1 will be repeated throughout the survey with boldface terms and arrows indicating topics for current discussion, and terms entirely in lower case and dashed arrows indicating topics covered previously.

Figure 1 is not meant to indicate that capability-based addressing is the only way to support small protection domains and extended-type objects. It is the most frequently advocated way for a system to support these concepts thoroughly and efficiently; and it is the one covered in this survey. Research on compile-time support for these concepts is also in progress, but it is not covered in this survey.

The arrows in Figure 1 must not be interpreted to mean "guarantees." Security and reliable software are both dependent on many other ideas that fall outside the scope of this survey and are not listed in this figure. Nevertheless, the ideas discussed here would go a long way toward building an environment where it would be realistic to expect that both security and very reliable software could be achieved.

Sections 3 to 5 of this survey cover small protection domains and their usefulness for reliable software and security. Sections 9 to 11 cover the uses of extended-type objects. In the middle, Sections 6 and 7 deal with capabilities and capability-based addressing. The two sections on capability-based addressing give a brief survey of a very complex subject. The reader who is interested in more details on capability-based addressing is referred to [Fabry 74] and [Saltzer 75]. Section 8 indicates that flexible sharing is not only compatible with the other ideas listed in Figure 1 but even interacts favorably with some of them. Readers who are interested in reliable software but not in security may skip Sections 5 and 11. Readers who are only concerned about security in a narrow sense may omit Sections 4, 8, and 10.

Readers should be aware that the ideas discussed in this survey are quite controversial. Many of my colleagues would disagree with one or more aspects of Figure 1. While I feel that the interaction of all of these ideas is crucial in order to attain the broad goals being addressed, many other approaches have been proposed which omit parts of Figure 1 or give different interpretations to some of its terms. In particular,

system security is often pursued in a way which is much less closely linked with reliable software. Less ambitious approaches to system security may be adequate if data sharing is restricted and security requirements are narrowly defined.

1.3 Introduction to Basic Terms

This section provides introductory definitions for the terms appearing in Figure 1 and for some related terms. These definitions may be skipped by readers who are generally familiar with the subjects being covered. Other readers can use these definitions to obtain an initial understanding of the relations depicted in Figure 1. This section may also be used as a glossary while reading the remainder of the survey.

SECURITY - The protection of resources (including data and programs) from accidental or malicious modification, destruction, or disclosure.

SYSTEM SECURITY - The state of a computer system (hardware and software) that makes it possible to provide reasonable assurance of security. System security presupposes that appropriate steps are taken for physical protection of the computer, for operating and maintaining the system, and for identification and authentication of users of the system.

RELIABLE SOFTWARE - Software that provides services which are (1) usable, (2) correct, (3) trustworthy, and (4) available on demand.

PROTECTION MECHANISMS - System features that are designed to protect against unauthorized or undesirable access to data.

SUBJECTS - Users of a computer system together with any other active entities that act on behalf of users or on behalf of the system; for example, processes, jobs, and procedures may be subjects. Subjects are also objects of the system.

OBJECTS - Identifiable resources or entities in the system. Software-created entities such as files, programs, semaphores, and directories are objects as well as hardware resources such as memory blocks, disk tracks, terminals, controllers, I/O ports, and tapes.

MODES OF ACCESS - The set of distinct operations that the protection mechanisms recognize as possible operations on an object. "Read," "write," and "append," are possible means of access to a procedure, and "debit account" is a possible mode of access to an object of type "bank account record."

ACCESS RIGHT - The right to use an object according to one of its recognized modes of access.

PROTECTION DOMAIN - An environment or context that defines the set of access rights that a subject has to objects of the system.

SMALL PROTECTION DOMAINS - Protection domains that typically restrict a subject to access rights for only those objects that are needed to accomplish the current task.

TYPE - Objects are classified by type. The examples under "objects" illustrate different types of objects. A type is defined by defining the set of operations applicable to objects of that type. Two objects are of different type if the allowable operations on the objects are different.

EXTENDED-TYPE OBJECTS - If the system allows applications programs to define new types and to create objects of these newly defined types, then such objects are called extended-type objects. The protection mechanisms should control access to objects of extended type in terms of the operations defined by the extended type.

LEVELS OF ABSTRACTION - Computers can solve human problems even though their electronic circuits only manipulate bits. The gap between the human problems and the bits is bridged by many concepts starting from concepts such as data base models and query languages that are implemented in terms of other concepts such as stacks, segments, and sequencing operations that are ultimately implemented as machine words and then as bits. One concept is said to be at a higher level of abstraction than other concepts if the concept organizes

instances of the lower-level concepts so that they can be manipulated effectively without having to understand the details of how the lower-level concepts interact. Levels of abstraction can be realized as types. The isolation of different levels of abstraction is a current goal of much work on programming methods and on system design.

CAPABILITY - A token used as an identifier for an object such that possession of the token confers access rights for the object. A capability can be thought of as a ticket. Modification of a capability (except to reduce its access rights) is not allowable; however, unlike the case for tickets, reproduction of a capability is legal.

CAPABILITY-BASED ADDRESSING - The use of capabilities to address and control access to objects even when the objects are stored in the primary memory of a computer system.

USER - An individual who interfaces with the computer system and can be held accountable for his actions. The term covers all uses of the system whether to submit data, queries, or other transactions; to execute programs; or to operate or maintain the system.

USER JOB - Used here as a general term for a unit of processing services performed on behalf of an identifiable user.

2. SYSTEM SECURITY AND RELIABLE SOFTWARE

Figure 2 indicates that this section introduces the terms system security and reliable software, and it covers the relation between them.

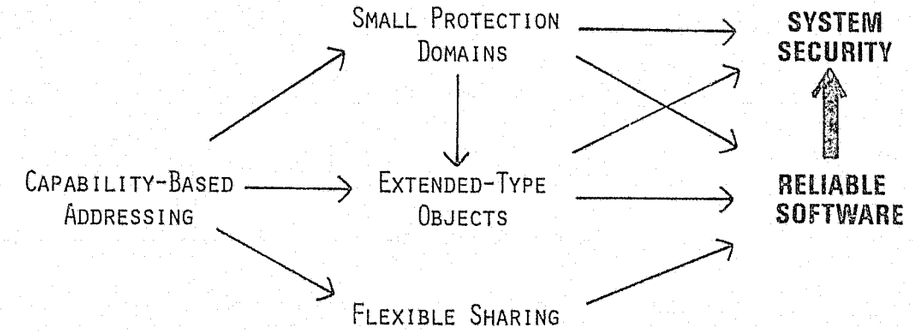

Figure 2 - System Security and Reliable Software

2.1 System Security Requirements

Corporate financial records, personal information as defined by privacy legislation, and classified military information are examples of information which must be protected during computer processing. It is hard to give a precise definition of computer security because specific security requirements depend so strongly on the larger human, social, and financial systems that are served by the computer processing. In general, security is concerned with any unauthorized or undesirable modification, disclosure, or destruction of information. In some situations (e.g., air traffic control), it is even concerned with a potential loss of service that would make critical information unavailable. 3/ For many installations, the unauthorized modification of information is the most serious security threat.

3/ Typically, computer security is also concerned with protecting the investment in the computer itself; however, this is mostly a matter of physical protection and is not discussed in this survey.

Security must be concerned with any path by which information could be modified, disclosed, or lost. For example, security requires that the system's operator interface be designed so that users cannot easily spoof the operator by sending him a counterfeit message that appears to be a system message. Security must also be concerned with the correctness of procedures for system initialization and for fault recovery and restart. For example, on some current systems the checkpoint/restart facility is a security weakness because the checkpoint data is not adequately protected from modification by users.

While security must be concerned with all paths which might provide unauthorized access to information, some aspects of the overall security problem are clearly beyond the control of a central computer system and can be regarded as separate problems. Generally speaking, communication security, identification of users, and physical protection of the computer site are distinct problems. Other problems fall on the borderline. For example, the security of most systems can easily be broken if an operator can be bribed. This might seem to be outside the control of the hardware/software system. However, if a system is designed for security, it is reasonable to expect that it should be designed so that the operator's command language provides a protection environment that carefully limits his privileges. Clearly this implies a major rethinking of the role of the operator with respect to the system. Nevertheless, it will be necessary to have some control over the damage that can be done by a corrupt operator--or an incompetent one. Similar comments apply to system programmers and system administrators.

This survey does not describe specific solutions to the above security problems; rather it describes operating system structures that support effective and efficient solutions to a wide variety of security problems. Other surveys and tutorials contain more details on specific security problems [Saltzer 74, 75, Popek 74b].

2.2 Reliable Software

Reliable software plays a dual role in Figure 2. It is a means to security, and it is an end in itself. Security depends in part on the reliability of software; however, the general problem of unreliable software is much broader than the security problem.

Reliable software provides services that are adequate for the intended application with respect to being:

(1) usable,
(2) correct,
(3) trustworthy, and
(4) available on demand.

Recent research on reliable hardware has been able to focus on the final aspect of reliability; namely, the constant availability of services. With respect to software services, a broader meaning for "reliable" is needed because it is still not realistic to presuppose that software services are usable, correct, and trustworthy. Usable means that the user receives services that are effective for his application. Correct means that the software meets its functional specifications. If the specifications are incomplete, then correct software may not be usable. Trustworthy means that there is a minimum level of services that is provided correctly, and there is an effective way to evaluate or measure the performance of the software with respect to this minimum level of service. Software may be correct even if there is no effective way to demonstrate its correctness; however, trustworthy software must be structured so that testing, auditing, and/or proofs of correctness can be used to achieve a reasonable level of confidence in the software.

There is much current research aimed at relieving the problem of unreliable software. This survey concentrates on protection mechanisms and other operating system structures that enhance the reliability of software--both systems software and applications software. Nevertheless, work on operating system structures to support reliable software is almost inseparable from recent work on designing modular, well-structured

programs. Furthermore, appropriate operating system structures can improve the results obtainable from many other software development techniques--including techniques for program management, testing, validation, proof, and maintenance.

2.3 Reliable Software for System Security

Reliable software is not only an end in itself, it is also a means to support system security. Typically, security depends on the reliability of much of the system software, and that reliability must be preserved through many versions and modifications of the software. Faulty system software is the system security problem that has been most difficult to deal with.

Security's dependence on the reliability of software can be reduced if the hardware and software are structured so as to reduce the size and complexity of the software needed to guarantee security. Security kernels that concentrate all the security-relevant code into a small, well-identified part of the system have been proposed [Schiller 73, Popek 74a, Lipner 74]. Yet, even with ideal hardware and software, many security concerns are dependent on a substantial amount of software. This survey describes operating system structures that support security directly--and also indirectly by improving the reliability of the security-relevant software.

3. SYSTEM PROTECTION MECHANISMS

While security and reliability requirements vary greatly from one application to another, the protection mechanisms that are built into the hardware and basic software of the computer system cannot be redesigned to meet the needs of each application. Thus it is desirable to have a basic set of protection mechanisms that are versatile enough to meet the requirements of many diverse computer applications. Even a single installation usually has a wide variety of security and reliability requirements.

The protection mechanisms of most third-generation computers were designed to confine user programming errors in order to prevent such errors from damaging either the system or other users. These protection mechanisms are based on a distinction between a privileged supervisor state and a non-privileged problem state (instructions that halt the machine or modify certain registers cannot be executed from the non-privileged problem state). This basic protection mechanism improves the reliability of system software by protecting it from the most obvious source of unreliability; namely, user programming errors. However, it does nothing to help the system protect itself against its own errors. Furthermore, while this protection mechanism could theoretically provide a basis for security against deliberate subversion of the system, in practice the problems of securing a computer system are so complex that many researchers have concluded that more sophisticated protection mechanisms are needed before rigorous security can be expected at a reasonable cost. 4/

3.1 Protection Models and Protection Domains

The versatility of a system's protection mechanisms can be characterized abstractly in terms of a protection model. A protection model views the computer as a set of active entities called subjects and a set of passive entities called objects. The protection model defines the access rights of each subject to each object. This protection model

4/ A variety of other protection features such as passwords and activity logging have been included in most computer systems. A combination of such protection features can be used to provide deterence against some security threats; however, these other protection features can be bypassed if the basic protection mechanisms are subverted. Despite many serious efforts to correct flaws in the protection mechanisms of current computer systems, it is still true that no computer system has withstood determined efforts to bypass its internal security controls by someone who is given user programming access to the system. Such penetration efforts have been successful against virtually all commercially-available operating systems.

can be represented in the form of a protection matrix [Lampson 71, Graham 72] as exemplified in Figure 3. In this protection matrix, subjects are associated with rows of the matrix and objects are associated with columns. For each subject-object pair, the corresponding entry in the matrix defines the set of access rights that the subject has to the object. Figure 3 shows that subject C may read or execute object X.

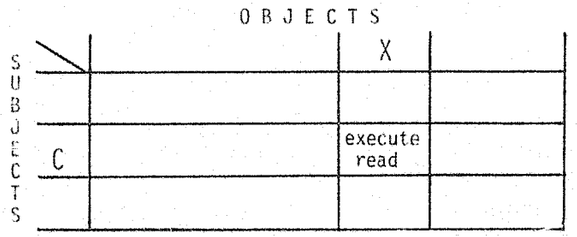

Figure 3 - A Protection Matrix

Access rights represented in the protection matrix also control changes to the protection matrix itself; for example, a subject with "delete" access to an object can eliminate that object from the protection matrix. Subjects also appear as objects in the protection model so that one subject can have access rights to another subject. For example, one subject may be allowed to transfer control to another subject by using an "enter" access right to the other subject.

A protection domain defines the set of access rights that one subject has to the objects of the system. A protection domain is represented as a row of the protection matrix. The term "protection environment" is used as a more general word that is similar to a protection domain except that a protection environment also includes everything that a subject might cause to be done on its behalf by another subject. A protection domain is a more restricted concept and includes only access rights to objects that are accessible by the subject.

Most third-generation computer systems support a protection model in which the subjects are basically the authorized users of the system. The supervisor or operating system is another subject that typically has total access to all objects in the system. In these systems every subunit of a user's program executes in the same protection domain, and that protection domain has access rights to all objects that the user ever needs. With this protection model, there is no easy way to limit the access rights of specific subprograms executed on behalf of a user. While the access rights of a protection domain can be increased or decreased, any such change is relatively permanent; and if access rights are deleted before calling a subprogram, they cannot easily be retrieved when the subprogram terminates.

Multics introduced the concept of protection rings which allow each user to execute in a linearly ordered set of protection domains. In Multics a protection subject is the combination of the user ID and a ring number. Each user can execute in as many protection domains as there are ring numbers. The different protection domains of a single user are linearly ordered in that the protection domain of a lower ring contains all the access rights of any higher ring. Hardware modifications for Multics that would eliminate this ordering of a user's protection domains are described by [Schroeder 72a, 72b]. This modified hardware should support the concept of small protection domains described in the next subsection.

3.2 Small Protection Domains

The phrase "small protection domains" is used as a qualitative description of a certain class of protection models. The word small is not intended in a rigid quantitative sense. The basic idea is that the protection domains should be as small as possible while still allowing programs to access what they need to access. This idea has been called the "principle of least privilege."

A small subunit of a program typically only needs access to a small number of objects. If small subunits of a program execute in their own protection domains, then the protection domains can be kept small. A large program usually needs access to many objects. Thus, protection domains can be kept small only if a large program executes in many different protection domains and constantly switches between these protection domains during its execution.

The flexibility, ease, and efficiency of domain switching is the primary factor in determining whether protection domains can be kept small and closely tailored to actual needs. However, other factors are also important; namely:

(1) The size of the protectable objects in the system.
(2) The different ways in which the protection matrix is allowed to change with time, and the ease of setting up new protection domains.
(3) The flexibility for defining different modes of access to objects.

Small protection domains characterize protection models that are very flexible. The protection matrix is large and sparse. The protection matrix is large because the protection system recognizes many distinct subjects (protection domains) and many distinct objects. The protection matrix is sparse because subjects have access to relatively few objects and with relatively limited modes of access.

Figure 4 indicates the role of small protection domains with respect to the other concepts covered in this survey.

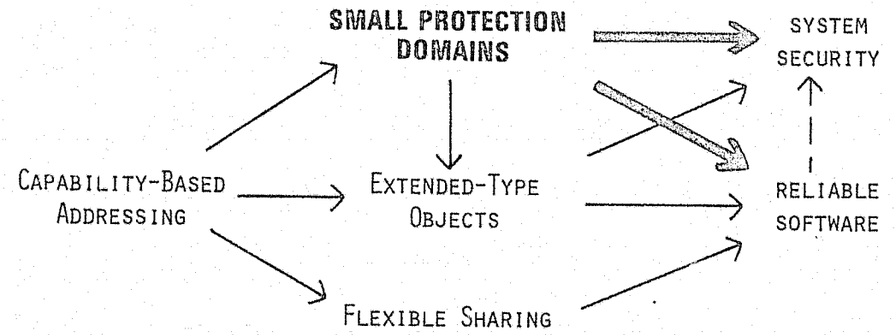

Figure 4 - The Role of Small Protection Domains

The usefulness of small protection domains for reliable software is discussed in Section 9. A way of implementing small protection domains with efficient switching between them is sketched in Section 7. The remainder of this section discusses some of the complexities that are involved in the concept of protection domain switching.

3.3 Protection Domain Switching

It is natural to integrate protection domain switching with the calling of a procedure. This means that each procedure could have its own protection domain, although every procedure call does not necessarily involve a domain switch. The phrase "protected procedure" is used when it is necessary to emphasize that the procedure call does involve a domain switch.

A protected procedure has its own protection domain associated with it. Thus, the right to access certain objects may be available during the execution of that procedure--and possibly only during executions of that procedure. Furthermore, each execution of that procedure possesses these access rights independent of the calling environment.

This is analogous to the concept of an own variable from ALGOL. It also means that a protected procedure can have a state which is preserved between calls to the procedure--and that state is independent of the calling environments. In this sense a protected procedure has a characteristic which has commonly been associated with the word process. Nevertheless, in this survey the word process is being used for a thread of sequential execution. A single process is allowed to execute in many different protection domains, and multiple processes are necessary only when there is the possibility of parallel execution.

A protected procedure appears as both a subject and an object in a protection matrix. It is an object because other subjects may have the right to call it. The right to call the procedure requires a special access right such as an "enter" right to the procedure. The protected procedure is also a subject in the protection matrix because it executes in its own protection domain.

A switch to a different protection domain involves a call to a protected procedure. If there are no access rights passed as parameters in the call, then everything is quite simple. If the caller has the right to call this protected procedure, then the call takes place and execution begins in the protection domain of the called procedure. A return instruction triggers a return to the previous protection domain.

The protection matrix in Figure 5 illustrates this situation. User A, executing in his basic domain, can call the editor. A dictionary (which may be a proprietary file) can only be read while executing in the editor's domain. The user can read or write files X and Y either from his basic domain or after calling the editor; however, he can use the dictionary to check the files for apparent spelling mistakes only when he has transferred control to the editor.

Objects / Subjects	Editor	File X	File Y	Dictionary	
⋮					
User A	Enter	Read Write	Read Write		
Editor		Read Write	Read Write	Read	
⋮					

Figure 5 - Simple Domain Switch

The domain switch is more complex if access rights to objects are to be passed as parameters and if the protected procedure is to be reentrant. In this case the call of the protected procedure results in the creation of a new protection domain--conceptually this means that a new row is created in the protection matrix. The new protection domain contains both the permanent access rights of the protected procedure (these are defined by a template domain associated with the procedure) and the access rights that are passed as parameters in the call. The new protection domain is destroyed by the return from the protected procedure. This situation is illustrated by Figures 6 and 7. In Figure 6 the user is executing in his basic domain, and the editor's template domain only has the right to read the dictionary. If the user then calls the editor in order to edit file X, he passes access rights for file X to the editor. This creates a new domain labeled "instance of editor" in Figure 7. Note that other users may be editing other files using other instances of the same editor.

Objects / Subjects	Editor	File X	File Y	Dictionary	
⋮					
User A	Enter	Read Write	Read Write		
Editor Template				Read	
⋮					

Figure 6 - Protection Matrix Before Call to Editor

Objects / Subjects	Editor	File X	File Y	Dictionary	
⋮					
User A	Enter	Read Write	Read Write		
Editor Template				Read	
Instance of Editor		Read Write		Read	
⋮					

Figure 7 - Protection Matrix During Call to Editor

The example in Figures 6 and 7 illustrates a situation involving mutual suspicion. If file Y is sensitive, the user does not have to allow the editor access to it, and the editor can protect the dictionary from direct access by the user. Implementation of domain changing is simpler when the domain changes involve only an increase in access rights (e.g., a system call) or a decrease in access rights (e.g., a testing program that calls the programs to be tested). The more general form of domain changing, where some new access rights are obtained while others are lost, is needed if the principle of least privilege is to be enforced.

The domain change for a re-entrant protected procedure sounds cumbersome when it is explained in terms of an abstract protection model. Section 7 suggests an implementation that allows protection domains to be created and destroyed easily and efficiently.

If a procedure has permanent access rights to an object, and if access to that procedure is shared by asychronous processes, then distinct activations of the procedure could lead to synchronization problems. In this case it would be the responsibility of the procedure code to handle the synchronization problem. The concept of a monitor [Hoare 74] can be viewed as a multiple-entry procedure of this sort which is invoked precisely to handle synchronization problems.

4. PROTECTION FOR RELIABLE SOFTWARE

It is far more difficult to build a 50,000 line program than it is to write 1,000 programs that are each 50 lines long. This phenomenon leads to rapidly escalating costs for the development and maintenance of large software systems, and it leads to serious reliability problems due to the difficulty of adequately debugging and testing a large program. Both the reliability and the cost of software could be greatly improved if the complexity of large programs could be kept more in line with the size of the program. Small protection domains are one of the most promising ways to achieve a breakthrough in reducing the complexity of large software systems.

The emphasis of this section is on the reliability of large software systems. Small protection domains will not greatly improve the realiability of a single small program. (It is an unfortunate reflection on the state of programming that programs of a few hundred lines can take on the characteristics of large complex systems.)

4.1 The Decomposition of Complex Systems

The complexity of any large system is more manageable when it is decomposed into relatively stable subsystems. These subsystems interact with each other, and each subsystem is itself made up of parts which interact; however, to avoid excessive complexity, a part of one sybsystem must have negligible interactions with parts of distinct subsystems. This decomposition of the system can then be iterated on each of the subsystems to result in a hierarchically structured system. Simon [Simon 69] has suggested that this is a common organizing principle of all complex systems, and that it can be observed throughout the physical, biological, and social sciences.

Hierarchical decomposition of a complex system has frequently been advocated by programmers--both under the name of program modularity and of structured programming [Dijkstra 68, 72, Parnas 72b]. While much progress has been made in recent years, the programming profession has had a difficult time decomposing large programs in such a way that the interaction of distinct subprograms can be defined and anticipated. This problem has two aspects:

(1) It has been very difficult to structure large programs in such a way that the decomposition does not result in longer and substantially less efficient programs.

(2) There is no way of knowing that distinct subsystems are interacting only as planned. The usefulness of program decomposition is greatest when there are errors in the system, and it is precisely these errors that are likely to cause distinct subsystems to interact in unanticipated ways.

Some recent ideas with respect to the first point will be discussed in Section 10. The second point is the primary reason why a protection system will make a major contribution to more reliable and less costly software.

4.2 Protection Should Be Distinct From Functionality

When a large system is decomposed into interacting subsystems, it is important to have limits on the interaction of the subsystems. These limits should not be dependent on the proper functioning of all of the subsystems. Otherwise, the subsystem interactions may change precisely when one of the subsystems fails, thus causing the whole system to degenerate into chaos. Simon [Simon 69] notes that in physical and biological systems, extraneous interactions among subsystems are often limited by physical distance. Physical distance also provides reasonable isolation of computer hardware modules. In the case of large software systems, there has been no equivalent of physical distance to help control extraneous interactions between subsystems. The result is that malfunctions in a software module more easily propagate throughout the whole system.

It is not feasible to eliminate all malfunctions from software subsystems. On a case-by-case basis, careful defensive programming can limit the effects of potential malfunctions. A more general solution is possible by introducing a protection mechanism which is distinct from the proper design and functionality of the subsystems. The role of the protection mechanism is precisely to prevent malfunctions from spreading beyond the subsystem where they occurred. To achieve the desired protection, almost every procedure should be run in a protection domain that gives it access to exactly what it needs to accomplish its function and nothing more. This is called the principle of least privilege. Furthermore, protection domain switching must be easy and efficient because the protection must not inhibit the desirable interactions between subsystems.

A protection mechanism will not prevent every error from propagating outside of the erroneous module. Many erroneous results of a module will appear to be normal results, and the protection mechanism will have no way of distinguishing these from correct results. However, with good system design, erroneous results that look like expected results should not cause other modules to behave in unpredictable ways. As long as other modules continue to behave in predictable ways, there is a much better chance of finding the origin of the error. The protection mechanism will guard mostly against the errors that result from unexpected interactions of the modules. These are the errors that are usually the hardest to trace.

Much recent literature on programming suggests various means of preventing program modules from interacting in unanticipated ways. These generally fall into three categories.

(1) Defensive programming practices - Programmers can include extra code that is designed to detect errors and to check whether modules are interacting as planned. For example, parameters and global data structures can sometimes be checked for consistency and reasonableness before they are used. The value of defensive programming is now well recognized; however, it must be used with discretion for it can increase the complexity of a program as well as its execution time.

(2) Language-enforced protection - The procedure, as it exists in many current programming languages, is a unit of modularity and it can prevent some unwanted interactions between modules. Other compile-time protection features have been advocated in [Morris 73, Palme 74, Liskov 74, Wulf 74b, 76a] and elsewhere. Much protection against unanticipated interactions between modules can be enforced at the time of compilation and linking. Other protection features--especially those dealing with access to shared data structures--are very difficult to implement at compile time.

(3) Protection mechanisms supported by the operating system - Small protection domains with the system enforcing protection at run time have been advocated in [Lampson 69, Needham 72, Price 73, Wulf 74a, England 74, Spier 73] and elsewhere. Sections 6 and 7 sketch the argument that protection checking for small protection domains can be enforced efficiently at run time. Efficient system enforcement of small protection domains requires

redesign of very fundamental parts of the computer system including the addressing mechanism.

4.3 Protection Information in System Design and Documentation

Most current programming practices do not require that the access rights of each module of a system be explicitly defined. While the definition of this access control information would be an additional programming requirement, this redundant information would be very useful as part of a formal system design document since it defines all the allowable interactions between modules. The definition of the access rights should be regarded as an important step of the system design, it would constitute an important system design document, and it would be executable in the sense that the protection mechanisms would enforce these controls at run time.

4.4 Value of Small Protection Domains

Small protection domains will be of the most value for the following aspects of the programming process:

- o Debugging Programming errors will be easier to find because errors in one module are less likely to manifest themselves by anomalous behavior of a different module. The correction of one error is also less likely to cause other modules to begin to malfunction.

- o Testing Testing one module at a time will be easier since the execution environment of the module is more rigorously defined. Furthermore, since that environment is enforced at run time, module interactions that were not anticipated in the tests should be prevented.

- o Fault detection, recovery, and retry It will be easier to contain the effects of either hardware or software errors within the execution environment where the error occurred. Since the execution environment is rigorously defined it will be easier to incorporate additional redundancy or run-time tests to protect against the remaining potential sources of error. Recovery and retry procedures are critically dependent on discovery of the error before things have gotten out of control.

- o Maintenance and modification Protection information defines the set of modules which could be affected by a modification to the system. This identifies the modules which have to be examined to guarantee that any modification will not have unexpected side-effects.

- o Proving properties of programs The origin of the author's interest in protection was partially to make it feasible to prove properties of moderate size programs. The length and complexity of a proof typically grows much faster than the length of the program. This is because each subunit of the program makes many assumptions about its execution environment. Typically, much of the rest of the program has to be used in order to prove that these assumptions are always valid every time the subunit is entered. If proofs are to be simplified, it is clear that ways must be found to prove the validity of these assumptions without using large amounts of otherwise irrelevant code. Protection mechanisms can provide a simple basis for this.

Small protection domains cannot be used effectively without some substantial modifications to most existing programming languages. Programs written in existing languages could still be run on such a system, but they would generally be compiled to execute in a few large protection domains. To take advantage of the small protection domains, programming languages would have to incorporate additional features that make it possible to define and control the protection domains.

5. SMALL PROTECTION DOMAINS FOR SECURITY

The economics of building large computer systems is such that the basic protection mechanism incorporated in the system must be able to satisfy many diverse security requirements. Small protection domains provide a flexible basis for implementing many different security requirements.

5.1 Flexibility vs. Security

Flexibility is not necessarily desirable for security. In general, security provisions must be as simple and rigid as possible in order to minimize the danger of oversights and of human error. Nevertheless, for security in a computer system, the flexibility of small protection domains is desirable for the following reasons:

(1) System security will be attacked at its weakest point. It makes little sense to build extremely rigorous security barriers if there is a back door into the system that is left open. Some common security problems are very difficult to solve without a flexible protection mechanism; for example, small protection domains are useful if there are to be any software controls to protect against a fraudulent system programmer or operator, and they are often needed to handle the Trojan Horse problem described later in this section.

(2) A serious danger to security arises whenever the need for flexible protection is underestimated. If protection mechanisms are so rigid that they prevent efficient processing of information, then the protection is usually circumvented. A single general protection mechanism that is used without exception is better than a rigid one that has many exceptions.

(3) Sound accounting and auditing principles require a system of independent checking where each individual is accountable for his actions and no individual is able to modify information in such a way that the modifications are not detected. Small protection domains would provide a good base for restoring the segregation of duties and the independent checking that is often bypassed during computerized record handling.

(4) Flexible and efficient switching between protection domains makes it more feasible to build redundant security controls. As long as the basic protection mechanism itself is extremely reliable, redundant security checks incorporated in software can provide very rigorous security control. The use of redundant security controls is discussed in subsection 5.3 on intermediaries. Extended types provide a natural way of implementing this redundancy and are discussed in Section 11.

(5) Even though the generality of small protection domains may be hard to understand, specific security controls can still be simple and easy to understand. In particular, a flexible protection system should make it easier to build user interfaces that are tailored to the specific needs of the user. Thus, users of a specific security system should see a simple security system.

(6) In the overview (Figure 1 or 4) there are arrows leading indirectly from small protection domains to system security via reliable software and extended-type objects. These indirect paths require very flexible protection mechanisms, and they are as significant for overall security as the direct path. For example, the protection mechanisms that support reliable software make it easier to build reliable software to monitor security.

5.2 The Trojan Horse Problem

Most access controls only guarantee that one user's information is protected from access by other users. Unfortunately, it is often not realistic for a user to trust all the programs that execute as part of his own processing. Most users make calls to a large number of service routines and other programs that the user has not written himself. On most systems, all these routines and programs execute with the full access privileges of the user. It is possible for these programs to perform actions totally unrelated to the caller's intent; for example, they may access any file accessible by the user, and on many systems they can even give away access rights to these files. Daniel Edwards has given this general class of problems the very descriptive name "Trojan Horse" because it involves a foreign or gift program that is brought within the walls of a protection domain [Branstad 73]. The gift program can then subvert the security of everything accessible from that protection domain.

Many discussions of computer security have paid as much attention to the Trojan Horse problem as the Trojans did. When building thick security walls, it is convenient to forget about this problem; however, it will do little good to build a new generation of "secure" computers if their security can easily be bypassed by a Trojan Horse attack.

The Trojan Horse problem is an extremely general and difficult problem. Programs that could have subversive routines in them are used constantly. Programmers and systems personnel routinely try out new programs that play games, print pictures, or aid in the development of better programs. The most acute danger from the Trojan Horse problem occurs when someone executing with system privileges runs a program given to him by "a friend"; however, the Trojan Horse problem arises for all programs that are executed on the system. This includes support programs such as editors, compilers, and library routines. A user may choose to believe that programs supplied with the system are unlikely to act like a Trojan Horse--but this should be recognized as a calculated risk.

It might seem that the Trojan Horse problem should be solved by administrative controls. Systems personnel and anyone who has very sensitive data should never run a program in their protection environment unless they trust it. Unfortunately, this administrative solution is often not practical unless the system makes it easy to run untrusted programs in a restricted protection environment where they can do little harm. Finding a reasonable solution to the Trojan Horse problem is probably the most challenging aspect of developing an adequate set of system security controls.

Three distinct aspects of the Trojan Horse problem must be distinguished when a foreign or untrusted program is to be run on a system:

(1) The foreign program is expected to modify sensitive data. In this case the foreign program must be thoroughly examined so that it can be trusted. If the program is to alter data, then it must be trusted with respect to that data--at least with respect to the particular types of modifications it is expected to make.

(2) The foreign program is expected to read sensitive data but not disclose its contents except to the calling program. This is called the confinement problem [Lampson 73]. It is difficult enough to prevent a program from hiding the information in a file or other form of storage; however, it is even more difficult to prevent it from communicating the information via a covert channel. Covert communications channels can be created by encoding the information in the program's resource utilization. For example, a program might communicate one bit to another program by using 10 minutes of CPU time if the bit is 1, and only using a fraction of a second if the bit is 0. The other program has to be able to detect or estimate the execution time of the first program--possibly by simply observing the performance of the system. Much higher data rates can be achieved by encoding the information in paging rates, disk utilization, or in the locking and unlocking of files. A formal way of approaching this problem is proposed by [Lipner 75], and partial solutions appear to be feasible. The partial solutions would reduce the data rate of the communications channels that a program can use to disclose the information, and they would increase the probability that various forms of monitoring (either of the system or of the program) could detect the communication.

(3) The foreign program is run on behalf of a user who has access to sensitive data, but the untrusted program is not expected to access any sensitive data. This problem should be easy to solve; however, the solution is difficult to enforce with the protection mechanisms available on most existing computer systems.

Security always involves trusting or believing something. A "solution" to the Trojan Horse problem means that the amount of trusted software is minimized. For a secure system, solutions to the third aspect of the Trojan Horse problem should be natural and routine. A solution to the second aspect--the confinement problem--should be possible and a matter of system tradeoffs. Some help can be provided with the first aspect of the problem by making it possible to distinguish different modes of write access to the data. The amount of software that still has to be trusted depends on the processing and security requirements; however, when the amount of trusted software is minimized, it may be feasible to audit, certify, or prove the integrity of that software which is to be trusted.

There are two approaches that have been taken to the Trojan Horse problem. The first approach is applicable when the primary security requirement is to prevent unauthorized disclosure outside of fixed, relatively broad security classifications. In this case the first aspect of the Trojan Horse problem is not relevant, and the third can be eliminated by running each user process in a fixed but limited environment. Efforts can thus be concentrated on solving the confinement problem for the process as a whole.

The second approach toward solving the Trojan Horse problem is more general, but it requires frequent changing between protection domains. Whenever a partially untrusted procedure is called, that procedure should be executed in an environment that gives it a minimum number of access privileges-- while still allowing it to carry out its assigned tasks. This approach to solving the Trojan Horse problem is based on the principle of least privilege, and is attributable to Daniel Edwards [Branstad 73].

Note that the Trojan Horse problem differs from the general software reliability problem only over the question of whether the called program may be malicious or whether it may be incorrect. Thus it should not be surprising that solutions to the two problems involve the same feature--frequent switching between protection domains to enforce the "principle of least privilege."

5.3 Intermediaries

A large class of security problems can be solved by putting a level of indirection between a subject and the object it is seeking to access. Protected procedures that act as intermediaries can be programmed to control access to an object by checking the calling process's identification, by checking for special capabilities which indicate authorization, or by performing any other programmable operation [Hoffman 70, Conway et al. 72]. For example, an intermediary can implement any of the following security controls:

(1) Redundant controls. Assuming that access to the intermediary is already controlled, the intermediary can implement a second and redundant check to guarantee that all access to the object is authorized. Redundant controls are especially useful to contain the effect of errors made by those administering, maintaining, or using the system. Of course, redundant controls are useful only if no single act can bypass both controls [Fabry 73].

(2) Restricted access. The intermediary can restrict access to parts of the object. Field and record level security controls could be handled in this way.

(3) Data dependent controls. The intermediary can check the contents of the object before deciding what information to return to the caller.

(4) Auditing and monitoring. The intermediary can create an audit trail or log of all accesses to the object, or it can try to identify suspicious or undesirable patterns of access to the object.

All these forms of indirect or mediated access are easy to implement as long as the intermediary can execute in its own protection domain and as long as there is no way to bypass the intermediary. Of course, the intermediary does result in some additional overhead. Extended types as discussed in Sections 9 and 11 provide a convenient and natural way of implementing intermediaries.

6. CAPABILITY-BASED ADDRESSING

System support for limited forms of protection domain switching has been implemented by the ring structure of Multics and by a protection feature in UNIX that allows the effective user identification to be changed to that of the owner of a program file when that program file is called [Ritchie 74]. Other approaches to implement domain switching have been proposed in [Schroeder 72a, 72b, Price 73, and Spier 73]; however, capability-based addressing appears to be the simplest, most thorough, and most frequently proposed way to enforce small protection domains while a program is executing.

Much can also be done at compile time to enforce the concept of small protection domains--in particular, much of the modularity needed for reliable software can be enforced at compile time. The limitations on compiler-enforced protection appear to be the following:

(1) Compilers cannot handle many of the problems involved in real-time sharing of data between independent programs.

(2) Protection enforced at compile time would not help to detect and recover from failures in the hardware or in the system.

(3) The compiler could only handle part of the protection needed for security. Isolation of users and some control over resource sharing would still have to be handled by the system. 5/

Most of the limitations on compiler-enforced protection can be avoided in a network of small computers if there is relatively little resource sharing and if most data sharing is handled by making copies of the data. In such a network, compilers can enforce protection between program modules, and the reduced amount of resource sharing avoids many (not all) security problems. Capability-based addressing should be most effective for large, closely-coupled systems--especially for systems designed to support centralized data management services or large software development activities.

5/ In addition, if security depends in part on the compilers, then the compilers would also have to be validated for security. While it may be easier to validate a compiler than to validate an operating system, the validation of several compilers in addition to the validation of parts of the operating system would make security validation more difficult. Note, however, that compiler correctness cannot be completely eliminated as a security concern. If the operating system is written in a high level language, then the correctness of the compiler for that language is a security concern. Furthermore, the Trojan Horse problem applies to any compiler that is used by anyone with sensitive information.

This section introduces the concept of capability-based addressing, and the next section covers its use for an efficient implementation of small protection domains. Figure 8 indicates the relation of capability-based addressing to other terms yet to be covered.

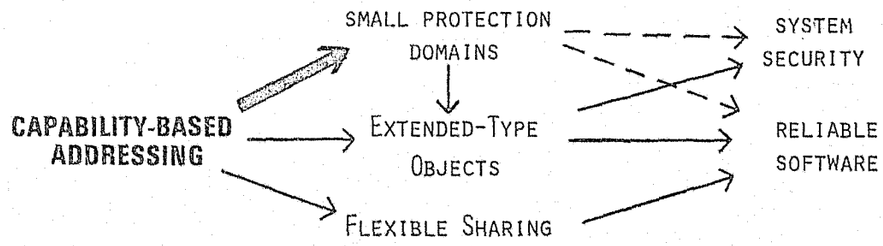

Figure 8 - Capability-Based Addressing

6.1 The General Concept of Capabilities

A capability may be thought of as a protected name for an object. While different systems use capabilities in quite different ways, capabilities generally have the following properties:

(1) Capabilities are system-wide names for an object. A subject has access to an object only if it possesses a capability for that object. 6/

(2) A part of the capability determines the access rights that the capability allows to the object that it names.

(3) Capabilities can be created only by a special low-level part of the system, and modification of a capability (except to reduce its access rights) is not allowable. Nevertheless, any subject in possession of a capability has some freedom to move it, to copy it, or to pass it as a parameter.

When an object is created, a capability for that object is also created. This initial capability includes all access rights to the newly-created object. The creator of the object may give a copy of the capability to other subjects. Recipients of a copy of a capability may use it to access the object, or they may make other copies of it to give to other subjects. When a capability is given to another subject, the access rights of the capability may be restricted. Thus each copy of a capability may allow differing access rights to the object. Except for the idea of amplification as discussed in Section 9.3, a capability that is passed to another subject cannot have more access rights than the capability from which it was copied.

6.2 The Use of Capabilities and Capability-Based Addressing

Capabilities as a general addressing and protection mechanism were first proposed by Dennis and Van Horn [Dennis 66]. Since then some version of capabilities has been used in the CAL-TSS system [Gray 72, Lampson 76], the BCC 5000 of the Berkeley Computer Corporation [Lampson 69], the SUE system for the 360 at the University of Toronto [Sevick 72], the HYDRA system [Wulf 74a, Cohen 75], the Cambridge Capability System [Needham 72, 74], and the Plessey System 250 [Cosserat 72, 74, England 74]. The reader should note that most of these systems are experimental in nature, several of them are no

6/ In the Cambridge Capability System [Needham 74], capabilities are interpreted relative to the capabilities in superior processes; and hence, they are not system-wide names for an object.

longer in use, and none has yet developed into a successful commercial product. Nevertheless, the idea of a capability has enough appeal so that many different experimenters continue to develop and use it. Furthermore, capabilities are similar to descriptors as implemented in systems such as Multics [Organick 72] and the larger Burroughs systems [Organick 73].

Several systems have used capabilities to facilitate sharing and protection of objects that are not loaded in primary memory. In these systems, interpretation of capabilities is done by software, and the primary memory is addressed and controlled by whatever means is available. Calls to the system software are needed in order to use a capability or switch to a different protection domain. Typically these calls require a millisecond or more. 7/

Other systems have integrated capabilities into the memory addressing mechanisms of the hardware. In this case a capability is interpreted on each reference to primary memory. This is called capability-based addressing.

The following explanation of capability-based addressing assumes that memory is organized into segments where a segment is a variable length sequence of memory words. A word in a segment is addressed by supplying an identifier for the segment and an offset that specifies the particular word of the segment. (For simplicity, fixed-size paging is being omitted from the present discussion since it is easy to add into any of the addressing schemes discussed.) A descriptor, as implemented in Multics and the Burroughs systems, is a protected identifier that points to a segment (or possibly to another object such as an I/O device). The descriptor also specifies the access rights that are allowed to the segment. An instruction references a memory word by pointing to a descriptor for the segment and by providing an offset to specify the desired word of the segment. The access rights of the descriptor are used to prevent any undesired access to the segment.

Capabilities used for the purpose of addressing segments of memory are almost indistinguishable from descriptors. They serve the same functions of identifying the segment and specifying the access rights to the segment. The primary difference between capabilities and descriptors arises because descriptor-based systems usually provide little freedom to manipulate the descriptors, and the hardware and low levels of the system software control all movements of the descriptors. Capability-based systems allow the capabilities to be moved and copied. This freedom to manipulate capabilities greatly simplifies the implementation of parameter passing during a domain switch; however, it also creates some security problems that must be handled by approaches discussed in Sections 7 and 11.

The Plessey System 250 [England 72, 74] and the Cambridge Capability System [Needham 72, 74] have implemented capability-based addressing, and system designs using capability-based addressing are reported in [Fabry 66] and [Neumann 74, 75].

6.3 Implementations for Capability-Based Addressing

Implementations of capabilities differ considerably; however, a capability usually consists of an identifier that can be used to find the object, a field defining the type of the object, and a field defining the access rights. A capability that allows only read access to a segment is illustrated in Figure 9. The access rights field is probably a set of bits--one bit for each mode of access. The interpretation of these bits depends on the type of the object. In some implementations the type field and/or the access rights field can be determined during interpretation of the capability, and they are not stored as part of the capability itself [Redell 74a, Neumann 75].

7/ This statement is supported in [Spier 75] and through verbal comments by B. Lampson about CAL-TSS, by K. Sevick about the University of Toronto SUE system, and by W. Wulf about HYDRA.

IDENTIFIER	TYPE OF THE OBJECT	ACCESS RIGHTS
POINTER TO THE SEGMENT	SEGMENT	READ

Figure 9 - Internal Structure of a Capability

Control over capabilities is necessary to prevent a user from creating a capability that he then could use to gain unauthorized access to an object. There are two approaches to achieve this control:

(1) Always have the capabilities stored in special locations such as capability segments and capability registers.

(2) Include an extra tag bit with each memory word. The tag bit must be inaccessible to the user. It identifies whether the word contains (part of) a capability, and the hardware then controls the modification of words that are identified as capabilities.

The advantages of each approach are discussed in [Fabry 74]. The second implementation avoids any rigid restrictions on how capabilities can be stored, moved, or copied.

(1) The identifier may be a pointer to the object--it may contain the address and a bounds for the object, or it may point to the object indirectly through an indirection table or a page table.

(2) The identifier may be a unique code that is permanently associated with the object. This is called a unique identifier.

The pointer approach makes it simpler to use the capability to reach the object; however, it means that the capabilities have to be updated periodically. If the identifier points directly to the object, then it must be updated whenever the object is moved; if it points indirectly then some of this overhead is reduced, but the capability still must be updated when the entry in the indirection table is changed. If the capabilities are not updated properly, then a capability for one object may end up pointing to a different object.

The second approach, based on unique identifiers, makes it unnecessary to keep track of capabilities and to update them. A unique identifier cannot be reused unless all capabilities for the previous object have been destroyed. It is usually best not to reuse identifiers. This means that the unique identifiers must be about 50 bits long. (Fifty bits allows the system to generate a new identifier every microsecond for about 35 years.)

The unique identifier approach requires that the current address of the object must be determined from the unique identifier each time the capability is used to address the object. This would be implemented by maintaining a large hash table to associate the current address of objects with the unique identifiers of the capabilities. Associative registers would be used to bypass the hash table search for subsequent accesses to the same object.

The disadvantages of the unique identifiers are the obvious space and time inefficiencies that are inherent in the searching and maintenance of the hash table. With proper hardware to optimize this, it appears that these disadvantages can be minimized. In exchange, the system is relieved of any need to modify the contents of capabilities (except to reduce its access rights), and shared access to objects is simplified.

Unique identifiers have been used in most software-based implementations of capabilities. The capability-based addressing used in the Plessey System 250 and the Cambridge

Capability System do not use unique identifiers. Appropriate hardware to support the unique identifier approach to capability-based addressing has not yet been built. 8/ For further discussion on the efficiency of capability-based addressing and on the use of unique identifiers in particular, the reader is referred to [Fabry 74, Neumann 75].

7. IMPLEMENTING SMALL PROTECTION DOMAINS

Capabilities provide one reasonable way to implement very flexible protection models. A capability corresponds to a set of access rights for a single object in the protection model. A protection domain, which is a row of the protection matrix, is realized as the set of capabilities that are accessible to the subject. This is illustrated in Figure 10 where part of a protection matrix is given on the left and its realization in terms of capabilities is depicted on the right. Note that User A can call the editor and pass access rights for File X by passing a copy of the capability for File X.

OBJECTS / SUBJECTS	Editor	File X	File Y	Dictionary
User A	enter	read write	read write	
Editor Template				read

CAPABILITIES OF USER A

ID for Editor	Proc.	enter
ID for File X	File	read,write
ID for File Y	File	read,write

CAPABILITIES OF EDITOR TEMPLATE

ID for Dictionary	File	read

Figure 10 - Protection Matrix Stored as Capabilities

If capabilities are used to address all objects in the system, then the concept of a protection domain corresponds to an address space or a name space. Any object that is not accessible to a subject cannot even be addressed by the subject.

This section describes the implementation of two aspects of small protection domains; namely:

(1) Efficient switching between protection domains.

(2) The storage and maintenance of protection domains in a way that allows them to be established and changed easily--yet under strict controls.

The potential reliability and correctness of a capability-based implementation of protection is discussed in subsection 7.3, and possible restrictions on the movement of capabilities are given in subsection 7.4.

8/ In the BCC 5000 computer, unique identifiers were used in capabilities for pages.

7.1 Capability-Based Implementation of Efficient Domain Switching

With capability-based addressing it is reasonably straightforward to implement domain switching as part of the hardware implementation of the call and return operations. With appropriate hardware support, the overhead to switch protection domains could be comparable to that of a simple procedure call in existing computer systems. Furthermore, call-by-reference parameters can be included in these cross-domain calls by including capabilities as parameters. The called domain does not need any additional addressing information or access authorization in order to use the passed capability. Since the capability is a system-wide address for the object, there is no danger that the called domain can misinterpret the capability. The capability also automatically provides access authorization to the object and enforces limitations on the authorized access.

The most efficient implementation for domain switching is probably achieved by using stacks [Neumann 75]. The process stack is divided into frames. At any point in its execution, the process only has access to the stack frame associated with the most recent protected procedure activation. In calling another protected procedure, parameters for the call are pushed onto the stack, and the call instruction delimits the new stack frame to be used by the called procedure. Figure 11 illustrates this by using the example of a call to an editor. For this illustration, stack frame markers are used to delimit the stack frames. After the call to the editor, only that part of the stack above the highest stack frame marker would be accessible. Note that parameters may be either capabilities or data.

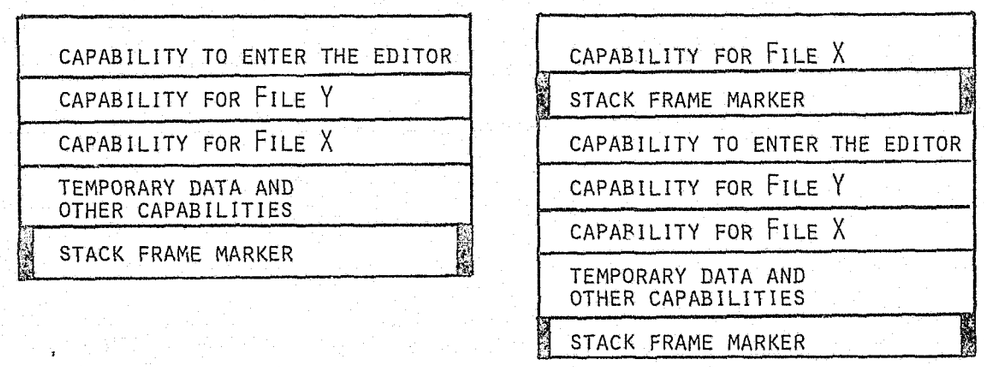

Figure 11 - State of the Stack Before and After a Protected Procedure Call

When the editor issues a return instruction, the editor's stack frame is deleted--except for any return data or capabilities. The return data is left on the top of the stack (see Figure 12). If the editor has copied a capability for the dictionary onto the stack, then this copy of that capability is automatically deleted by the return instruction.

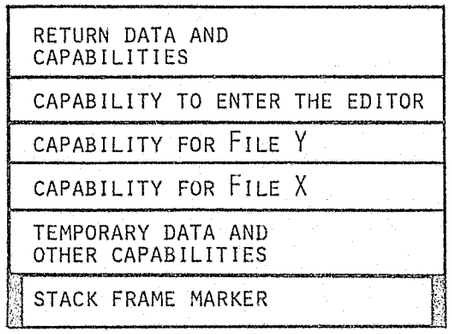

Figure 12 - Return From the Protected Procedure Call

The protection domain of the called procedure is defined by the capabilities that are:

(1) passed to it on the stack;
(2) embedded in the procedure code;
(3) available to the procedure from a directory system (see next subsection);
(4) otherwise accessible to the procedure; e.g., if they are stored in a segment that is accessible to the procedure.

Thus, when viewed in terms of its capability-based implementation, the creation of a new protection domain for each activation of a re-entrant protected procedure is quite simple.

7.2 Directories for the Storage and Sharing of Capabilities

In a protection system which allows a large number of independent protection domains, the protection domains must be stored and maintained efficiently. If each protection subject had to store large lists of capabilities--one for each object it is allowed to access--then the maintenance of all this information could be a serious problem.

There must also be provisions for controlled sharing of capabilities between distinct users of the system. If capabilities are stored in data segments, then any segment can be used to store and share capabilities. To maintain control over capabilities, most long-term storage and sharing should be handled by a system of directories that are specifically designed for these purposes.

A directory is basically a table of entries that associate user-chosen names with capabilities. Directories can have three distinct roles in a capability-based system:

(1) They simplify the storage and maintenance of the information required to implement a protection matrix, and they preserve the capabilities of inactive users.

(2) They allow objects to be addressed by user-chosen names rather than by the system-generated capabilities. They also make it possible to alter the association between a name and an object.

(3) They can be used to solve the lost object problem. If it were possible to erase the last capability for an existing object, then that object could never be accessed or deleted. The directory system could guarantee the existence of at least one capability for every existing object [Neumann 75].

A subject with access to a given directory is allowed to request the capability associated with a given name. To facilitate controlled sharing, it is desirable to have a means of allowing subjects access to some of the capabilities stored in a directory without necessarily allowing them access to all the capabilities in the directory. In Multics, this was accomplished by using access control lists [Saltzer 74]. For a system where each program activation of each user may be a distinct subject, a generalization of this approach has been suggested based on the idea of locks and keys [Lampson 71]. A request to the directory system requires both a capability for that directory and a key. The request is fulfilled only if the key matches a lock that has been associated with the named entry in the directory. The key can be implemented as a capability. In this case, the capability is simply a non-forgeable identifier which is not meaningful to the addressing mechanism. It would be meaningful only to the programs that implement directories.

Directories themselves are protected objects of the system, and a specific directory can be accessed only by a subject possessing a capability for that directory. Capabilities for directories can be stored in other directories, thus creating a network structure among the directories. The network structure is usually restricted to be partially ordered.

Directories are the primary repository for long-term storage of capabilities. Thus directories play a key role in storing and maintaining protection domains. (Each subject's protection domain includes all the capabilities that the subject can retrieve from the directory system.) Directories are also useful as a way of modifying protection domains when users share access to objects. The stack handles the relatively short-term modifications to protection domains that occur when capabilities are passed as parameters during domain switches.

7.3 Correct Implementation of Protection

Much of the computer security problem is due to our inability to design and implement large computer systems that are correct. Correct implementation of the basic protection mechanism is clearly critical to all security. While different objects may be given different degrees of protection according to their relative sensitivity, no object in the system can be more secure than the basic protection mechanism. Even objects that are protected by redundant security controls are not safe if the basic protection and addressing mechanisms can be broken or bypassed. Thus the correctness of the protection mechanisms must be guaranteed with a very high degree of confidence.

The implementation of a very flexible protection system is more complex and more difficult than the implementation of a more rigid and limited protection system. In a capability-based system the amount of hardware and software that supports the protection mechanism is greater than that needed to implement a security kernel. However, capability-based addressing simplifies some of the system software, and the small protection domains make it easier to control the interactions between different system modules. Furthermore, capability-based addressing automatically avoids many of the common integrity flaws that have been found in existing computer systems. For example, a common integrity flaw occurs when an address that is passed to a system routine is changed between the time the system routine checks it for validity and the time it is used. Similarly, the integrity of several systems has been broken because the system gives special privileges to anything with a certain name, such as FORTRAN COMPILER. Capabilities prevent these types of integrity compromises from occurring.

The implementation of capabilities using unique identifiers can also handle the danger that a hardware error might alter a few bits in an address so that the address can now be used to access a different object. Such a change would usually be detected by the error-detecting codes that can be expected on any larger future systems. However, in a system using unique identifiers for capability-based addressing, even if the hardware does not detect an error, the probability that a capability would be transformed into capability for another existing object could easily be made exceedingly small--probably less than 2^{-30} if the unique identifier is 50 bits long. 9/

It is still a difficult task to implement a capability-based system with the degree of reliability and integrity that is desirable for security. Nevertheless, if the modularization and reliability techniques discussed in Sections 4 and 10 are used in the design of the system itself, then a very high level of confidence in the integrity and correctness of the protection systems should be possible. This confidence might be based in part on proofs of properties of the system. A system design that uses capability-based addressing and is structured so that proofs about it are feasible, is reported in Neumann [75].

7.4 Controls Over the Movement and Storage of Capabilities

If the addressing of all objects in the system is based on capabilities, and if the protection mechanisms associated with this addressing are correct and reliable, then the restrictions of a protection matrix can be guaranteed if each subject has access only to those capabilities that correspond to entries in the protection matrix. This represents a major step toward being able to handle security problems. It means that one only has to control the movements of capabilities. This is much better than having a variety of poorly defined concerns about almost everything that happens within the computer system. Nevertheless, the problem of controlling the movement and copying of capabilities is far from trivial--especially since capabilities are designed to be moved and copied easily in order to support small protection domains.

With a tagged architecture where extra "tag" bits on each memory word are used to distinguish capabilities from data, it would be possible to intermix capabilities and data freely. This has some advantages for implementing multi-segment data structures with the capabilities used for cross-segment pointers. Nevertheless, to maintain control over capabilities, it may be necessary to prevent capabilities from being stored in most user data segments. If enough specific facilities are provided for indirectly manipulating capabilities, then direct manipulation or storage of them may not be necessary by most users of the system. The stack that handles capabilities passed as parameters, the directory system, a linkage manager, and an extended-type manager (see Section 9) may make direct manipulation of capabilities unnecessary for most users.

For security it would be useful if the system could guarantee that the directory system is the only means to share capabilities between distinct users or to store them for relatively long periods of time. This would make it much easier to monitor the security status of the system. It would be useful even if it only applied to capabilities for especially sensitive objects. Additional protection features that might be used for this purpose are:

9/ This assumes that there are less than 2^{-20} extant objects at any one time. It also assumes that unique identifiers are scattered throughout the space of possible bit patterns; for example, they could be generated by using an appropriate linear congruential sequence (see [Knuth 69], pages 9-19.) Note also that if the unique identifiers are generated using a linear congruential sequence, then the hash address for a unique identifier may be taken as some subset of the bits in the identifier. Furthermore, the regular patterns that occur in the final bits of words generated by a linear congruential relation might allow some optimization in distributing the unique identifiers among hash buckets.

- Capabilities restricted to the stack - Situations arise fairly frequently where access rights must be passed to an activation of some service routine, but the service routine should not be able to preserve those access rights for later use. An option probably should be available so that specific capabilities passed as parameters on the stack can be restricted from being copied off the stack. This would not be so restrictive as to prevent the capability from being passed as a parameter in a further procedure call; however, it would guarantee that no copy of the passed capability could exist after the service routine returns. The right to move the capability off the stack could be controlled as an access right of the capability.

- Restriction on loading or storing capabilities - If direct manipulation of capabilities by users is allowed, then the right to load a capability from a segment or store one into a segment should be distinct from the right to read or write data in the segment. This distinction is useful to implement provisions of the security classification models proposed in [Bell 73] and [Walter 75].

- Interprocess communication channels - It must be possible to impose restrictions on the direct passage of capabilities between processes via interprocess communications channels.

- Revocation of capabilities - For some security problems it is necessary to revoke access rights which have previously been given to another subject. If all relatively long-term storage of capabilities is handled by the directory system, then the directories might be able to handle this problem. If not, then selective revocation of an access right requires special features because the capabilities that represent the access rights may have been copied many times. Access rights to an object can always be revoked by deleting the object (after making a copy of it), but this may destroy the access rights of other subjects. It may be desirable to revoke the access rights of a single subject--and of any other subject that received the access rights from that subject. Selective revocation of capabilities can be implemented by creating revocable capabilities that point to an object indirectly through the main capability for the object. The revocable capability can thus be distributed to other subjects who can use it to access the shared object. Access via the revocable capability can be efficient since associative registers can bypass the indirection on all but the first reference to the object. The revocable capability can later be made ineffective without disturbing the access rights of those subjects who possess either the main capability or an independent revocable capability. For a full discussion on the revocation of capabilities, see [Redell 74a, 74b, Neumann 75].

8. FLEXIBLE SHARING

In this survey, the discussion of capabilities has focused on their usefulness to promote security and reliable software. Nevertheless, one of the primary motivations for capability-based addressing is to facilitate sharing. This other motivation for capability-based addressing is not covered here. (It is covered in detail in [Fabry 74].) This brief section is intended to indicate that the approaches to security and reliable software discussed in the rest of this survey are not only compatible with flexible sharing but also enhance it.

Figure 13 indicates that capability-based addressing supports flexible sharing and that flexible sharing supports reliable software. The arrow from flexible sharing to reliable software is based on the argument that software would be more reliable if programmers could more easily build on the work of other programmers rather than constantly reinventing the wheel. (Reinvented wheels often turn cut to be not quite round.)

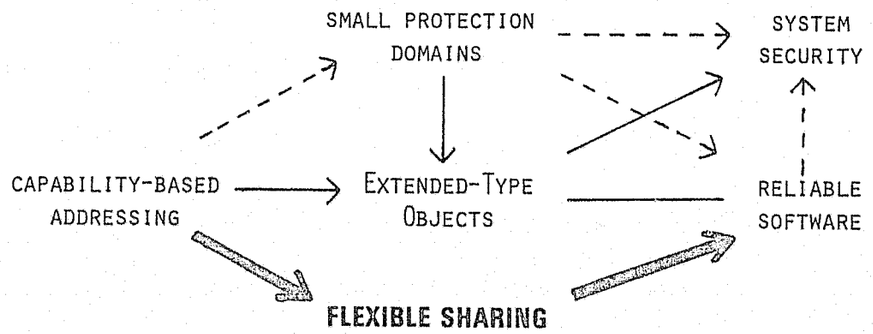

Figure 13 - Flexible Sharing

In general, sharing is opposed to both security and reliability--it is especially opposed to security. The simplest way to improve system security is to reduce the amount of sharing. For example, an especially sensitive applications program can be run on its own dedicated computer. Unfortunately, sharing and security are often concurrent requirements. Indeed, if there is no requirement for any form of sharing--not even resource sharing--then there is no need for internal system security. In many situations--especially situations involving privacy concerns--security is needed in the presence of very flexible sharing of both resources and data.

Reliable software is also more difficult when there is extensive sharing; in particular, time-critical sharing of data can result in deadlocks or inconsistent data. On the other hand, the sharing of program modules could lead to more reliable software. The idea of building programs by piecing together modules from a program library is not new; however, it has always been difficult to make this idea work unless the modules perform isolated and easily definable functions. The difficulty occurs when one tries to integrate the different modules. In particular, it has been difficult to develop useful library modules that deal with complex data structures.

Despite the above difficulties, a building block approach to reliable software may soon become feasible. Recent evolution of the concept of extended types is leading toward a unit of modularity that is more general and easier to integrate as part of a larger program. Furthermore, these modules can be used to specify and implement common data structures such as stacks, queues, trees, and symbol tables. Such a module can be quite general; for example, a single module that implements trees may be used to obtain a tree of integers, a tree of stacks, or a tree of any other data structure. Furthermore, the concept of a generator, as is proposed in [Wulf 76b], allows another module to iterate over the elements of any of these trees without making the other module dependent on the internal workings of the tree module. Two other concepts interact with this approach to program modularity and re-enforce it. First, very flexible protection is useful to keep one module from becoming dependent on the internal workings of other modules. Second, specification and proving techniques are more effective in conjunction with this new approach to modularity [Wulf 76a]. To achieve reliable software, it is clearly important that modules obtained from a library be fully specified and verified.

The description of extended-type objects in the following two sections may give the reader some additional insight into why the building block approach to reliable software could become a reality. For a more thorough treatment of some of the supporting ideas, the reader is referred especially to [Wulf 76a, 76b, 76c]. It should be noted that this approach to reliable software is arising mostly from research on programming languages; and the required support for these new concepts can be handled largely by a compiler; however, capability-based addressing would extend the usefulness of these concepts and facilitate their implementation.

9. EXTENDED-TYPE OBJECTS

As indicated by Figure 14, this section introduces the final concept covered by this survey. It also explains how small protection domains and capability-based addressing support the implementation of extended-type objects.

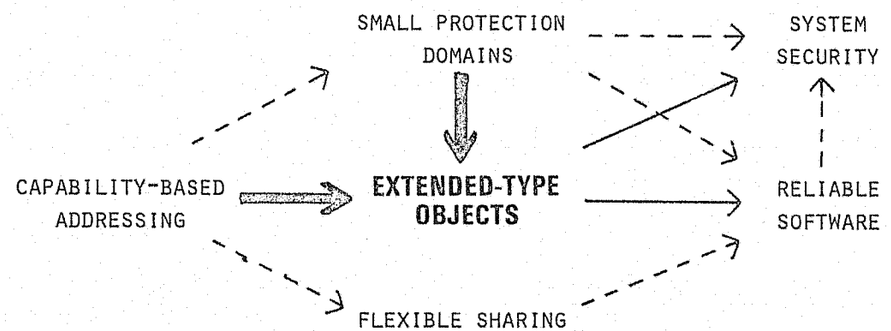

Figure 14 - Overview on Role of Extended-Type Objects

9.1 Background on Typed Objects

The previous discussion of capabilities focused on the addressing and protection of information in memory. A protection system is simplified if I/O devices are addressed and protected in the same way as memory. For example, in the PDP-11, I/O devices are addressed as if they were words of memory, and they can be protected by memory protection registers. This simplifies and unifies the protection mechanisms; however, in the PDP-11 the flexibility of this protection is limited since the protection of an I/O device is not independent of the devices with neighboring addresses.

A capability can easily be used to address and protect individual I/O devices. When a capability is used to address I/O devices, the access rights of the capability are interpreted differently for each different type of I/O device. For example, a capability for an object of type "tape drive" might have "rewind" as one of its modes of access, while a capability for a card reader would not recognize rewind as a possible access right. 10/

Capabilities can be used in an even more general way to address and protect all objects in the system--not only memory and I/O devices but also software-created virtual objects. For example, procedures and directories may both be implemented as segments of memory; however, they are different from ordinary segments because the protectable modes of access to them are different. Procedure objects have an additional mode of access not applicable to data segments; namely, "enter" access. It is critical to security that this operation be separately protected. Similarly, directories must be recognized by the protection mechanisms as a different type of object even though

10/ Many systems will do strange things in response to requests for undefined actions such as "rewind the card reader!" Such requests can frequently be used to break the security of a system [Edwards 73]. In a system based on capabilities and typed objects, it is not even possible to formulate such a request in terms of a capability.

the hardware does not distinguish directory segments from data segments. Operations such as add an entry, delete an entry, and change the protection of an entry should all be separately protectable operations on a directory. These operations on a directory are not reducible to the usual operations of read, write, append and delete for a data segment, and the protection systems must be prepared to handle these different operations on directories.

In most systems the operations on directories are protected by implementing them as part of the operating system. In this case the operations on directories are system calls, and the directories themselves are implemented as system data. Thus, the implementation and protection of directories is accomplished entirely by the system software. Directories are a good example of the way additions to the system software are often used to extend the protection system and make it more flexible; however, directories could also be implemented as one instance of extended-type objects.

A system that directly supports many different object types would be baroque and complex. Before asking how many different object types a system should support, one should first ask whether there has to be a fixed set of object types and whether different types have to be supported and protected directly by the system. Possibly the system should just provide a mechanism for creating, defining, and protecting new types. Such a mechanism has two principal advantages:

(1) It eliminates the need to incorporate into the system the code and data which support different types of objects. Even the code and data which implement the directory systems could then be independent of the rest of the system. In fact, there would no longer be a clear distinction between "the system" and applications.

(2) The protection system can be extended to support applications programs directly. If applications programmers can create new object types, then they can extend the protection system and protect objects in ways that are tailored to a specific application. This would greatly increase the flexibility of the protection system, and it would provide a very natural solution to a wide range of protection problems.

Objects of a type that are not directly implemented by the system are called extended-type objects [Gray 72, Jones 73, Wulf 74a, Ferrie 74, Neumann 75].

Wulf et al. [Wulf 74a] give an example of a system for creating, maintaining and accessing special bibliographic files. They describe a set of reliability and security concerns that arise naturally, and they argue that these concerns can most easily be solved by creating a new extended type and then having bibliographic files be objects of that extended type. As another example, in a payroll system it might be desirable to provide distinct access controls for operations such as: modifying salary, reading salary, changing an address, and totaling all salaries. These access controls can be provided easily if the payroll files are declared to be objects of a new extended type.

9.2 Nature of Extended Type Objects

Much current research on operating systems structures and on programming languages is focusing on generalizations of the concept of a data type. The term "extended-type objects" is taken from work on operating system structures. The word "extended" is added to emphasize that new types are definable and that the protection system can be extended to handle these newly defined types. When it is not needed for emphasis, the word "extended" can be dropped with no change in meaning. As discussed in Section 10, recent research on programming languages has led to a similar concept, but quite different terminology is often used.

From the viewpoint of this survey, a type is defined by the set of operations that are allowed on objects of that type. This view is consistent with most research on generalized data types. Thus, segments and directories are different types of objects because different operations are possible on them. (These operations often correspond to

the modes of access to the object; although many operations could be associated with a single protectable mode of access.)

An extended type is defined by specifying and implementing a set of operations applicable to objects of that type. These operations should include operations for creating and deleting objects of the type. All these operations could be implemented as software procedures. These operations normally have a parameter that indicates the object on which the operation is to be performed.

Objects of a new type can be created once the type has been defined. These objects are distinct from the type itself. 11/ The objects may be viewed abstractly simply as primitive entities that can be manipulated only by the operations of the type. For example, a directory is defined as an entity that allows the operations of adding and deleting entries, changing the accessibility of an entry, etc.

Objects must also have an implementation or an underlying representation which is defined in terms of other objects. The representation of a directory may be a linked list in a segment. The implementations of the operations on a directory manipulate the linked list in this segment. Ideally, the subject that initiates this operation does not need to know how the directory is represented and can take an abstract view of it. Furthermore, if the operations on the typed object are to be individually protected, then the subject that initiates the operation to add an entry to a directory must not be allowed write access to the segment that implements the directory. Write access to the segment must be available only during execution of the procedure that implements the add operation.

Extended-type objects are implemented or represented in terms of more primitive objects--segments, I/O devices, or objects of other previously-defined types. The extended-type object should be thought of as distinct from the objects used to represent it--the representation exists at a different "level of abstraction." In particular, subjects that initiate operations on an extended-type object should normally not have direct access to the representation. The value of protecting the representations is not just for security; as discussed in Section 10, the protection also separates distinct levels of abstraction and protects the representation from undesirable modifications by the code of other modules.

9.3 The Implementation and Protection of Extended-Type Objects

Only the operations of an extended-type object are to have access to the objects that are used to implement or represent the extended-type object. Thus, each call to one of the operations requires a domain switch. This domain switch is straightforward when there is only one underlying object that contains the representations of the extended-type objects. In this case, the operations of the extended-type may have the access rights for the underlying representation, and all that is required is a simple domain switch. In the more general case there may be many objects of the extended type (e.g., many directories) and each may be represented by its own underlying object(s) (e.g., a segment that represents a single directory). An instance of an operation does not need access to the underlying representation for all the objects of the type, it only needs access to the representation of the object that it is to operate on. If each instance of the operations had access to the representation of all the objects, then the entire burden of selecting the right representation would be placed on the code of the operation. This might be dangerous for security. It is preferable if the access rights for the object that contains the representation object are passed to the operation as a parameter. The problem with this is that the caller does not have the right to access the representation either; the caller only has the right to call the operations on the extended-type object. Three methods to handle this problem have been proposed in the literature:

11/ The new type may be treated as a distinct object of the protection system. This allows the same protection matrix to control access to the type itself; in particular it can control modifications to the operations of the type. The type itself then has a type which may be taken as a special system-supported primitive type called TYPE [Wulf 74a].

(1) Amplification - The HYDRA system [Jones 73, Wulf 74a] allows a called procedure to amplify the access rights of certain capabilities. Amplification allows extended-type objects such as directories to be handled along the lines of the following example. A subject that has a capability with "add" access to a specific directory, calls the "add" operation and passes the capability for the directory. The add operation has a special "template" capability that allows it to amplify the access rights of capabilities for objects of type "directory." In this case the template capability would allow the add operation to obtain read and write access to the directory. (Amplification of access rights is actually more general than just what is needed just to implement extended-type objects.) In HYDRA, domain switching and amplification are done entirely in software; ideas for a hardware supported implementation of amplification are discussed in [Ferrie 74].

(2) Indirection - The Plessey System 250 [Cosserat 74] allows a procedure to be called indirectly through another object. This provides most of the features of extended types. To perform an operation on an extended type object, the caller would use an indirect "enter" right for the object. This transfers control to a procedure that implements the operations on the object. Using this indirection facility, directories could be implemented as follows. Stored at special locations in each directory are pointers to the code that implements operations such as the operation of adding an entry to a directory. To add an entry to a directory, a subject can use a capability with indirect "enter" access for the directory, and control is transferred to the procedure indicated by the pointer in the directory. This approach does not provide separate controls over the rights to add, delete, or read an entry.

(3) Extended-type manager - In the system designed by Neumann et al. [Neumann 75], there are different capabilities for an extended-type object and for its representation. The mappings from capabilities for objects of extended type to capabilities for their representation are maintained by a special module in the system called the Extended-Type Manager. This module returns a capability for the representation object if it is passed a capability for an object of extended type and if the request is made by an operation defined for that type.

An efficient implementation of extended-type objects clearly requires small protection domains with very efficient switching between protection domains. A domain switch is required with each call to an operation on an extended-type object. Capability-based addressing is useful for implementing extended-types because it provides a uniform and general way of naming and addressing all objects in the system. With capability-based addressing, extended-type objects can be addressed and protected as if they were primitive objects.

10. TYPED OBJECTS AND PROGRAM MODULARITY

The general concept of a typed object can be used as a primary means to decompose and modularize software. Section 4 discussed the value of protection in a well-structured and modularized program. This section discusses the use of typed objects to obtain that structure and modularity. Many recent approaches to structured programming involve a generalization of the concept of typed objects. An operating system with an efficient extended-type mechanism would facilitate these approaches to structured programming. Indeed, the assembly language of such a system would have many of the data structuring features that are desirable in a high level language [Cosserat 74].

The role of this section with respect to the general terms of the overview is indicated by Figure 15. Since small protection domains are such an integral part of extended-types, many of the discussions in this section also apply to the arrow from small protection domains to reliable software.

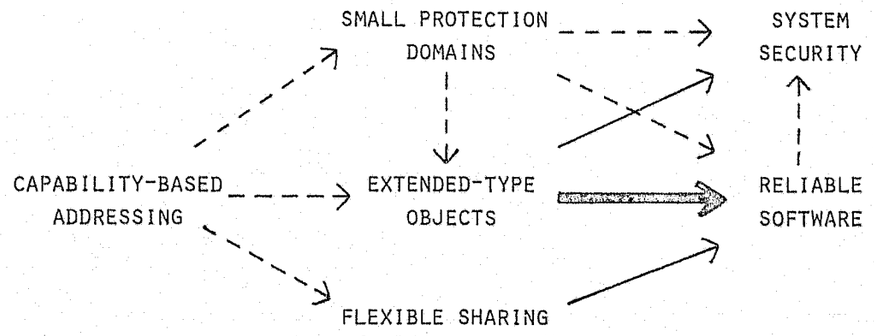

Figure 15 - Extended-Type Objects to Support Reliable Software

10.1 Background--Horizontal and Vertical Modularity

A careful statement of a programming problem usually leads to a decomposition of the problem into a number of separate tasks. The decomposition of a program into distinct problem-level tasks is called horizontal modularization. Unfortunately, horizontal modularity alone does not lead to an adequately modularized program for the following reasons:

- The decomposition into problem-level tasks usually does not divide the program into small modules. When the problem is described in user terminology, the smallest units or tasks which are meaningful, often turn into quite large programs. An input module, an input validation module, or an update module are meaningful in user terms, but they are only a first step toward dividing the program into small, independent modules.

- Different problem-level tasks or modules often need access to common information. If the data structures for this information are declared as global to the entire system, then there is little hope that modules can be independent of each other or that the interactions between modules can be clearly defined [Wulf 73].

- Ideally, a module should only deal with one level of abstraction. A module may implement operations that are meaningful at the user level, or it may deal with the idiosyncracies of the machine, or it may handle some intermediate concepts or data structures; but it should not implement concepts from different levels of abstraction. A module is very difficult to comprehend if one program statement implements a problem requirement and the next statement handles some subtle efficiency concern arising from pecularities of the hardware.

The division of a program into modules according to different levels of abstraction is called vertical modularity. Both horizontal and vertical modularity are needed if modules are to be small and clearly defined, and if significant benefits are to be obtained from protection between modules. Vertical modularity is closely related to the concept of hierarchical structure; however, the latter term has been used with many different specific connotations [Parnas 74]. The term vertical modularity is used here as a general term that does not connote a specific approach.

10.2 Programming Language Support for Modularity

Vertical modularity of programs is hard to achieve because current computer systems and programming languages do not support an appropriate unit of program modularity. The procedure is the most common unit of program modularity and is supported by most computer systems and programming languages; however, as a unit for vertical modularity it is often inadequate because:

(1) The Algol scope of variables rule is wrong when procedures are used as the unit of vertical modularity. Variables generally do not need to be global across different levels of abstraction. A unit of vertical modularity should not automatically have access to variables declared at higher levels of abstraction.

(2) A procedure cannot easily preserve information between successive calls, and thus it cannot be used as a unit of modularity to encapsulate a data base. Furthermore, it is, at best, difficult for a procedure to gather statistics about its use, or to incorporate redundancy checks based on the consistency of successive calls to the procedure.

(3) A unit of modularity often needs many entry points. It is awkward to use a parameter to obtain the effect of many entry points.

Recent research on programming languages has addressed the problem of providing a more generally useful unit of program modularity; for example:

- The class concept was introduced into Simula 67 [Dahl 68] as an aid to modularity. The main features of a class are: (1) it can define data objects that are normally preserved between calls, and (2) a class consists of several procedures or entry points. Thus, a class defines a set of operations each of which may operate on data objects. The original version of a class did not protect its data objects from direct access by other parts of the program; however, more recent versions do include protection [Palme 74].

- Liskov and Zilles have developed the language CLU which implements a concept called function clusters [Liskov 74, 75]. These clusters are similar to classes except that the representation of the cluster's data objects is not accessible from outside the cluster. Clusters implement the same idea of typed objects that was discussed in Section 9. Enforcement of the access restrictions is done by the compiler.

- The language Alphard has incorporated a concept called a form [Wulf 74, 76a, 76c]. It provides the features of a cluster described above, but is more general. For example, a form can accept parameters that allow only limited access rights to the object passed. Alphard also introduced the concept of abstract sequencing operations that allow a program to iterate through a data object without making the calling program dependent on the length or structure of data object [Wulf 76b].

- Parnas has proposed a method for decomposing programs into rigorously specified modules [Parnas 72a, 72b, 72c]. He suggests that a module can be defined in terms of a set of operations and a set of value functions supported by the module. (Value functions return a value but do not change the state of the module.) Parnas makes sure that the representation of the module's state (or its data) is hidden from other modules. At the level of specifications he accomplishes this by not defining the representation at all. The module is defined abstractly by defining the effect of all the operations on the value-returning functions and by defining all error conditions.

The search for the most effective unit for program modularity is still going on; however, based on recent research trends, it seems safe to conclude that a unit of modularity should have the three properties listed below. These three properties correspond (in reverse order) to the three reasons why a procedure is not adequate as a unit of vertical modularity.

(1) A module can have many operations or entry points which can be called from other modules.

(2) A module can have a set of data objects (or a state) which is preserved between successive operations.

(3) Interactions between modules can be explicitly defined and rigorously controlled. In particular, the representation of the data objects maintained by the module is not directly or automatically accessible by other modules.

Large units of modularity have always had the first two of these properties. The class concept from SIMULA extended these two properties to small units of modularity by providing programming language support for them. The third property adds protection to these units of modularity so that interactions between modules can be explicitly defined and controlled.

Terminology--and the underlying concepts--in this subject area is still in a state of flux. The term "extended type" arose from work on operating systems. The term "abstract data type" is a fairly general term that is now widely but not universally used by the programming language community. Furthermore, the following terms have all been used with an equivalent or similar meaning: class, cluster, form, opaque type, type, space, mode, module, abstract machine, virtual machine, and procedure.

10.3 Extended Types as Modules for Reliability

Modularity achieved by extended types is useful for software reliability in several ways that have not yet been mentioned:

- Proofs - The implementation of a program as a hierarchy of typed objects simplifies a proof of correctness [Wulf 74, 76a, Robinson 75]. Much of the simplification comes because assumptions about the content and structure of the representation of a typed object depend only on the correctness of the operations of that type--they are not dependent on the actions of any other modules.

- Redundancy - The type is a natural unit in which to incorporate redundancy checks. The ability to preserve information between successive operations is important to implement this redundancy.

- Error detection and recovery - The definition of error conditions is relatively easy when it is done as part of the specifications for a type. Recovery techniques can be structured by defining appropriate error calls and error returns as part of the external interface of the modules [Parnas 72a].

- o Modifications and maintenance - Since the representation of typed objects is hidden from other modules, the representation can be changed without affecting other modules. Programs can be implemented quickly as a hierarchy of types, and then the more critical types can be tuned for efficiency [Linden 76]. The fact that the data structures can be modified without affecting any module except the one that maintains that data structure is critical to making this optimization work.

- o Security - The extended type is a useful unit for security as discussed in the next section.

11. CONTROLLING AND MONITORING ACCESS TO OBJECTS

In order to encourage good programming practices and to support the concept of small protection domains, protection mechanisms should not prevent a program from delegating subtasks to other protected procedures. Thus, a program should be able to pass any of its own access rights to another protected procedure. On the other hand, there are situations where one user of the system must be prevented from making access rights available to another user.

Capabilities make it easy for one subject to pass access rights to any other subject. They directly implement a reasonably complete set of discretionary protection mechanisms. Discretionary protection mechanisms allow each subject, at its own discretion, to decide which of its own access rights are to be given to other subjects. There is also a need for non-discretionary controls so that certain security policies can be enforced without depending on the discretion of other users of the system. Extended types provide a convenient way to implement many non-discretionary controls. Figure 16 indicates that this section deals with the relation between extended-type objects and system security. More specifically, it discusses non-discretionary protection mechanisms--including classification systems. It discusses both extended types and other means to support non-discretionary controls.

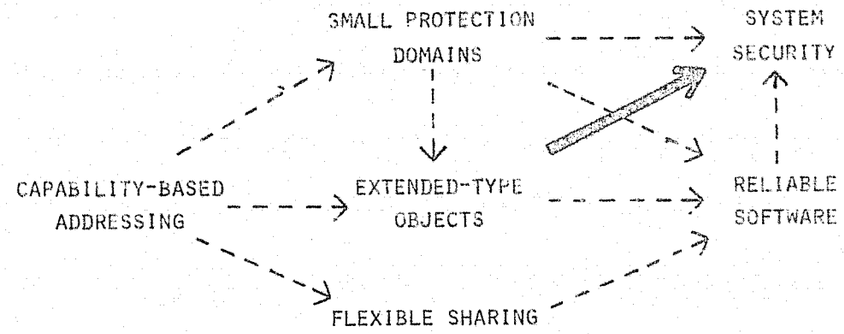

Figure 16 - Extended-Type Objects to Support Systems Security

11.1 Non-Discretionary Controls

The most basic way to enforce non-discretionary controls is to restrict the users' access rights for objects; however, simple restriction of access rights does not provide all the desired controls. For example, it is often desirable to monitor the way an object is being used. Similarly, it may be desirable to prevent one user from giving his access rights to another user. These examples can be handled easily by using extended types. To monitor and control access to certain objects, a new extended type is created for the objects and the operations of the type are programmed to enforce whatever controls are to

be maintained. If accesses to the objects are to be monitored and audited, the monitoring and auditing controls are built into the operations of the new type. If redundant control is to be maintained over the set of authorized users of the objects, then a check on the user's identity can be programmed into the operations of the type. Thus, as long as the users are only given access rights to the extended-type object, control can be maintained over which users can access the object even if some of the users give away copies of their capabilities for the extended-type objects. In general, extended-types provide a natural way to enforce the forms of mediated access discussed in Section 5.3.

With the extended type mechanism, users can be given very limited access rights that allow them to perform only preprogrammed operations on objects. Furthermore, additional controls and monitoring can be built into the programmed operations that are allowed by the access right. In most cases, a new type does not have to be created just to handle the controls and monitoring. They can be embedded in the operations of an existing type definition.

Non-discretionary controls implemented by using extended types involve some additional overhead to carry out the programmed checking. Unless the object to be protected is a primitive object with simple operations, the additional time required for the protection checking should not cause an unreasonable increase in the time required to carry out an operation on the object. If many relatively primitive objects, such as segments, have to be protected against misuse by authorized users, then it may be necessary to use a more centralized system of non-discretionary controls involving a security classification system as well as system monitoring and auditing.

11.2 Security Classification Systems

When non-discretionary controls are to be imposed in a centralized way, they usually involve a classification system. In a classification system, all users and all objects in the system are assigned a classification. A classification may be thought of as a security tag. The classifications may have a partial ordering relation between themselves so that they take on a lattice structure. Then any user is allowed access to an object only if the user's classification is either the same as the classification of the object or else is higher according to the ordering of the classifications.

The basic military classification system forms a very simple lattice with all points of the lattice being ordered along a single line (see Figure 17). In this system, a user who is authorized access to secret information is also authorized access to confidential and unclassified information, but not to top secret information. A corporation that wanted to separate information according to different departments might use a very broad classification lattice such as that in Figure 18. In practice, the classification lattice would have a more complex structure.

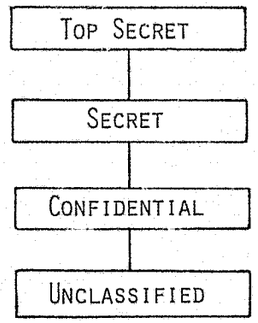

Figure 17 - Simple Military Classification Lattice

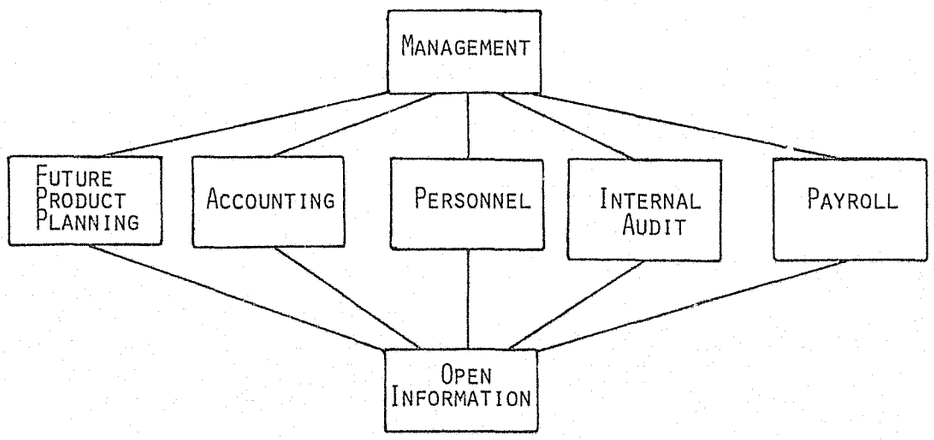

Figure 18 - A simple Classification Lattice for a Corporation

The problem of enforcing a non-discretionary classification system is more complex than it might seem at first. It is not enough to control the access rights that are passed from one user to another. If one user (A) has access rights for an object, and if the classification system is to prevent another user (B) from accessing the object, then it may do little good just to prevent A from giving the access rights to B. If A wants to subvert the classification system and allow B to access the object, then A can bypass restrictions on passing the access rights if he simply agrees to access the object on behalf of B; that is, A can set up a service whereby A carries out operations on the object whenever B requests them.

In general, if A has access to an object, and if B is to be prevented from accessing it, then either A must be trusted, or else all communication between A and B must be forbidden. If A, and all the programs executing on behalf of A, is trusted, then we are back to a discretionary system. If one is only concerned about inadvertant error on A's part, then it may be useful to prevent A from giving access rights to B; however, in order to implement rigorous non-discretionary controls, it is necessary to control all the potential communication channels between A and B. If the security concern is only that information from the object must not be disclosed to B, then only communications from A to B must be forbidden; and, conversely, if the object only needs to be protected from modifications originated by B, then only communication from B to A must be cut.

Classification systems are used primarily to protect information from being disclosed. It follows that, as long as users (or their programs) are not trusted, a user job should not be allowed to write or modify an object with a classification lower than the classification of any object previously read. The problem is that such objects could be used to disclose information of the higher classification. A non-discretionary classification system was incorporated in the ADEPT-50 system [Weissman 68], and classification systems have been formally defined in [Bell 73] and [Walter 75].

In a capability-based system that supports extended-type objects there are several possible approaches to implement a non-discretionary classification system.

In the first place, the protection matrix can enforce a classification system if the protection matrix is initialized so that users have access only to objects of appropriate classifications and have no way to obtain access to objects of another classification. (This does not handle the problem of covert communication channels as discussed in section 5.2.) In a capability-based system, this means the initial distribution of

capabilities to users would have to be done very carefully. If used by itself, this approach probably would not lead to very high confidence that the desired classification system was being enforced. A minor mistake in distributing capabilities could have unpredictable effects on the classification policy since seemingly unimportant access rights might enable users of different classifications to set up a communications channel between them. Furthermore, since the capabilities would be disbursed throughout the system, it would be hard to reevaluate whether the current dissemination of capabilities enforces the desired security policies.

A second approach is to maintain a much tighter control over the dispersion of capabilities. For example, all users might be forced to obtain their capabilities from the directory system, and they might be prevented from using any other means to preserve capabilities for more than short periods of time. With all permanent capabilities stored in the directory system, it would be relatively easy to determine who has access to any given object; however, small changes in the directory system might still have disasterous effects on the classification policy and would have to be very carefully controlled.

A third approach uses the extended type mechanism to enforce a classification system. All access to classified objects are explicitly controlled by a classified document manager that is implemented as an extended type [Neumann 75]. This approach need not be as inefficient as it might appear; however, it might work best when only a small fraction of the users are accessing classified objects.

The final approach is to build classification controls into the central part of the hardware and software as a second, independent protection mechanism. The need for some form of classification system seems to be sufficiently general so that it could legitimately be incorporated into the basic design of the system. This means that each user job and each object in the system would be tagged with a classification. These classifications could be checked each time a new object is made accessible to a user job. Note that this check on classifications would be in addition to the access controls built into the capability-based addressing.

12. CONCLUSION

Research has now progressed to the point where it is possible to discern the rough outlines of a potential breakthrough on both security and reliable software. No one idea will lead to such a breakthrough, but the proper combination of ideas that are now emerging could revolutionize both of these areas. The changes in computer systems that would help bring these ideas to fruition were outlined in this survey.

The reader should be aware that many of the ideas covered in this survey are still the subject of basic research, and before they can be put into practice they need a more rigorous examination than they have been given either here or elsewhere in the literature. However, further basic research is probably not the most important element on the critical path toward a breakthrough. The most important problem is to overcome the inertia which makes it easier to continue doing things as they have been done in the past.

The ideas discussed in this survey involve a substantial amount of discontinuity with the past. The basic addressing mechanisms of computer systems must be changed, and new structures for protection and modularity must be introduced into programming languages. These new ideas are not likely to be introduced into common practice unless there is a very strong economic incentive to do so and unless the ideas can be introduced in evolutionary stages:

(1) Economic incentive - Improved reliability and security usually involve higher costs. The new ideas promise to promote security and lead to substantially more reliable software while at the same time reducing costs--especially software development costs. Hard evidence to support this promise of decreased costs would go a long way toward overcoming inertia. Unfortunately, this evidence is very difficult to obtain without building a complete computer system that incorporates the new featues.

(2) Evolutionary stages - The current investment in computer systems and software precludes the development of large computer systems that are not compatible with older systems. Basic changes to a computer's addressing and protection mechanisms inevitably result in a substantially different computer. Nevertheless, the new addressing and protection mechanisms might make it easier to support multiple external interfaces. Compatibility with old systems could then be maintained by providing an external interface which simulates the interface of the old system.

A breakthrough on security and reliable software will not be easy to achieve. Several new ideas must be put into practice--and any one of the ideas may not succeed if it is not properly supported by other equally new ideas. It will be a major undertaking to achieve an effective combination of these ideas. Nevertheless, such a breakthrough must be sought. Ever more critical software applications, skyrocketing software costs, and the growing requirement for computer privacy all demand the development of computer systems which are at least as new and different as those discussed in this survey.

ACKNOWLEDGMENTS

The author became familiar with many of the ideas in this survey during the course of discussions with Peter Neumann, Robert Fabry, Lawrence Robinson, Karl Levitt and Daniel Edwards. When possible, their ideas have been referenced; however, many of their ideas are such an integral part of the overall approach that they can no longer be isolated from it. Suggestions that have helped to improve the accuracy and clarity of the document have also been received from Jerome Saltzer, Butler Lampson, Steven Lipner, Stuart Katzke, Thomas Lowe, and the editor and referees. Nevertheless, the author is solely responsible for any inaccuracies or lack of clarity that remain. My thanks to the above and to my wife, Betty, whose patience has been outstanding and whose editing was ruthless. Thanks also to Kathleen Durant and Anne Shreve for many long hours spent typing various drafts of the manuscript.

REFERENCES

[Anderson 72] Anderson, J., Computer security technology planning study. Air Force Elect. Systems Div., ESD-TR-73-51, (Oct. 1972).

[Bell 73] Bell, D., LaPadula, L., Secure computer systems. Air Force Elec. Systems Div., ESD-TR-73-278, (Nov. 1973).

[Branstad 73] Branstad, D. K., Privacy and protection in operating systems. Computer., Vol. 6, (Jan. 1973) pages 43-46.

[Cohen 75] Cohen, E., Jefferson, D., Protection in the Hydra Operating System. Proc. of the Fifth Symposium on Operating Systems Principles., ACM Operating System Review, Vol. 9, No. 5, (Nov. 1975) pages 141-160.

[Conway 72] Conway, R. W., Maxwell, W. L., Morgan, H. L., On the implementation of security measures in information systems. Comm. ACM, Vol. 15, No. 4, (Apr. 1972) pages 211-220.

[Cosserat 74] Cosserat, D. C., A data model based on the capability protection mechanism. IRIA Internat. Workshop on Protection in Operating Systems, Rocquencourt, France, (August 1974) pages 35-54.

[Dahl 68] Dahl, O.-J., Myhrhaug, B., Nygaard, K., The Simula 67 Common Base Language. Norwegian Computing Center, Oslo, (1968).

[Dennis 66] Dennis, J. B., Van Horn, E. C., Programming semantics for multiprogrammed computations. Comm. ACM, Vol. 9, No. 3, (March 1966) 143-155.

[Dijkstra 68] Dijkstra, E. W., The Structure of the THE Multiprogramming System. Comm. ACM, Vol. 11, No. 5, (May 1968) pages 341-346.

[Dijkstra 72] Dijkstra, E. W., Notes on structured programming. Structured Programming, Dahl, O.-J., Dijkstra, E. W., Hoare, C. A. R., Academic Press, (1972).

[Edwards 73] Edwards, D., private communication, (1973).

[England 72] England, D. M., Architectural features of System 250. International Switching Symposium, Cambridge, MA, (June 1972).

[England 74] England, D. M., Capability concept mechanism and structure in System 250. IRIA Internat. Workshop on Protection in Operating Systems, Rocquencourt, France, (August 1974) pages 63-82.

[Fabry 68] Fabry, R. S., Preliminary description of a supervisor for a machine oriented around capabilities. ICR Quart. Rpt. 18, Univ. of Chicago, (August 1968).

[Fabry 73] Fabry, R., Dynamic verification of operating system decisions. Comm. ACM, Vol. 16, No. 11, (Nov. 1973) pages 659-668.

[Fabry 74] Fabry, R. S., Capability-based addressing. Comm. ACM, Vol. 17, No. 7, (July 1974) pages 403-412.

[Ferrie 74] Ferrie, J., Kaiser, D., Lanciaux, D., Martin, B., An extensible structure for protected systems' design. IRIA Internat. Workshop on Protection in Operating Systems, Rocquencourt, France, (August 1974).

[Graham 72] Graham, G. S., Denning, P. J., Protection--principle and practice. AFIPS Conf. Proc. 1972, SJCC, AFIPS Press, Montvale, NJ, (1972) pages 417-424.

[Gray 72] Gray, J., Lampson, B. W., Lindsay, B., Sturgis, H., The control structure of an operating system. Research report, IBM Watson Research Center, (July 1972).

[Hoare 72] Hoare, C. A. R., Notes on data structuring. Structured Programming, Dahl, O.-J., Dijkstra, E. W., Hoare, C. A. R., Academic Press, (1972).

[Hoare 74] Hoare, C. A. R., Monitors: an operating system structuring concept. Comm. ACM, Vol. 17, No. 10, (Oct. 1974) pages 549-557.

[Hoffman 71] Hoffman, L. J., The formulary model for access control. AFIPS Conf. Proc. 1971 FJCC., AFIPS Press, Montvale, NJ, (1971) 587-601.

[Jones 73] Jones, A. J., Protection in programmed systems. Ph.D. Dissertation, Carnegie-Mellon Univ., Pittsburgh, PA., (June 1973) 139 pages.

[Knuth 69] Knuth, D. E., The Art of Computer Programming, Vol. 2, Seminumerical Algorithms. Addison-Wesley Publ. Co., (1969).

[Lampson 69] Lampson, B. W., Dynamic protection structures. AFIPS Conf. Proc. 1969, FJCC, AFIPS Press, Montvale, NJ, (1969) pages 27-38.

[Lampson 71] Lampson, B. W., Protection. Proc. of the Fifth Annual Princeton Conf. on Information Sciences and Systems., Princeton Univ., (March 1971) pages 437-443, (Reprinted in ACM Operating Systems Review, Jan. 1974)..

[Lampson 73] Lampson, B. W., A note on the confinement problem. Comm. ACM, Vol. 16, No. 10, (Oct. 1973) pages 613-615.

[Lampson 76] Lampson, B. W., Sturgis, H. E., Reflections on an operating system design. Comm. ACM, Vol. 19, No. 5, (May 1976) pages 251-266.

[Linden 76] Linden, T. A., The use of abstract data types to simplify program modifications. Proc. of Conference on Data: Abstraction, Definition and Structure, SIGPLAN Notices, Vol. 8, No. 2, (March 1976) pages 12-23.

[Lipner 74] Lipner, S., Chm., A panel session--security kernels. AFIPS Conf. Proc. 1974 NCC, AFIPS Press, Montvale, NJ, Vol. 43, pages 993-999.

[Lipner 75] Lipner, S. B., A comment on the confinement problem. ACM Operating System Review, Vol. 9, No. 5, (Nov. 1975) pages 192-196.

[Liskov 74] Liskov, B., Zilles, S., An approach to abstraction. Proc. of a Symposium on Very High Level Languages, SIGPLAN Notices, Vol. 9, No. 4, (April 1974).

[Liskov 75] Liskov, B., Zilles, S., Specification techniques for data abstractions. IEEE Trans. on Software Engineering., Vol. 1, No. 1, (March 1975) pages 7-18.

[Morris 73a] Morris, J. H., Protection in programming languages. Comm. ACM, Vol. 16, No. 1, (Jan. 1973) pages 15-21.

[Morris 73b] Morris, J. H., Types are not sets. ACM Symposium on Principles of Programming Languages, Boston, MA, (1973) pages 120-124.

[Needham 72] Needham, R., Protection systems and protection implementations. AFIPS Conf. Proc. 1972 FJCC, AFIPS Press, Montvale, NJ, Vol. 41, pages 571-578.

[Needham 74] Needham, R. M., Walker, R. D. H., Protection and process management in the CAP computer. IRIA Internat. Workshop on Protection in Operating Systems, Rocquencourt, France, (August 1974) pages 155-160.

[Neumann 74] Neumann, P. G., Fabry, R. S., Levitt, K. N., Robinson, L., Wensley, J. H., On the design of a provably secure operating system. IRIA Internat. Workshop on Protection in Operating Systems, Rocquencourt, France, (August 1974) pages 161-176.

[Neumann 75] Neumann, P. G., Robinson, L., Levitt, K. N., Boyer, R. S., Saxena, A. R., A provably secure operating system. Stanford Research Institute Final Report, Menlo Park, CA, (June 1975).

[Organick 72] Organick, E. I., The Multics System: An Examination of its Structure. MIT Press, Cambridge, MA, (1972).

[Organick 73] Organick, E. I., Computer System Organization--the B5700/B6700 Series. Academic Press, New York, (1973).

[Palme 73] Palme, J., Protected program modules in Simula 67. Research Inst. of National Defense, Stockholm 80 Sweeden, (July 1973) 25 pages.

[Parker 75] Parker, D. B., Computer abuse assessment. Stanford Research Institute, Menlo Park, CA, (Dec. 1975) 33 pages.

[Parnas 72a] Parnas, D. L., A technique for software module specification with examples. Comm. ACM, Vol. 15, No. 5, (May 1972) 330-336.

[Parnas 72b] Parnas, D. L., On the criteria to be used in decomposing systems into modules. Vol. 15, No. 12, (Dec. 1972) 1053-1058.

[Parnas 72c] Parnas, D. L., Some conclusions from an experiment in software engineering techniques. AFIPS Conf. Proc. 1972 FJCC, AFIPS Press, Montvale, NJ, (1972) pages 325-329.

[Parnas 74] Parnas, D. L., On a "buzzword": hierarchical structure. Information Processing 74 - Software., IFIP Congress 74, North Holland Publ. Co., (1974) pages 336-339.

[Popek 74a] Popek, G. J., Cline, C. S., Verifiable secure operating system software. AFIPS Conf. Proc. 1974 NCC, AFIPS Press, Montvale, NJ, (1974) pages 145-151.

[Popek 74o] Popek, G. J., Protection structures. Computer., Vol. 7, No. 6, (June 1974) pages 22-31.

[Price 73] Price, R. W., Implications of a Virtual Memory Mechanism for Implementing Protection in a Family of Operating Systems. Ph.D. dissertation, Carnegie-Mellon Univ., (June 1973) 244 pages.

[Redell 74a] Redell, D. R., Fabry, R. S., Selective revocation of capabilities. IRIA Internat. Workshop on Protection in Operating Systems, Rocquencourt, France, (August 1974) pages 197-210.

[Redell 74b] Redell, D. D., Naming and Protection in Extendible Operating Systems. (Ph.D. Thesis Univ. of Calif. Berkeley) MAC TR-140, MIT, Cambridge, MA, (Nov. 1974).

[Ritchie 74] Ritchie, D. M., Thompson, K., The UNIX time-sharing system. Comm. ACM, Vol. 17, No. 7, (July 1974) pages 365-376.

[Robinson 75] Robinson, L., Levitt, K. N., Neumann, P. G., Saxena, A. R., On attaining reliable software for a secure operating system. Inter. Conf. on Reliable Software, SIGPLAN Notices, Vol. 10, No. 6, (June 1975).

[Saltzer 74] Saltzer, J. H., Protection and the control of information sharing in Multics. Comm. ACM, Vol. 17, No. 7, (July 1974) pages 388-402.

[Saltzer 75] Saltzer, J. H., Schroeder, M. D., The protection of information in computer systems. Proc. of the IEEE., Vol. 63, No. 9, (Sept. 1975) pages 1278-1308.

[Schiller 73] Schiller, W., Design of a security kernel for the PDP-11/45. Air Force Elect. Systems Div., ESD-TR-73-294, (Dec. 1973).

[Schroeder 72a] Schroeder, M., Saltzer, J., A hardware architecture for implementing protection rings. Comm. ACM, Vol. 15, No. 3, (March 1972) 143-147.

[Schroeder 72b] Schroeder, M., Cooperation of mutually suspicious subsystems in a computer utility. Ph.D. dissertation, MIT, Cambridge, MA, (1972).

[Sevick 72] Sevick, K. C., Project SUE as a learning experience. AFIPS Conf. Proc. 1972 FJCC, AFIPS Press, Montvale, NJ, (1972) pages 571-578.

[Simon 69] Simon, H. A., The Sciences of the Artificial. MIT Press, Cambridge MA, (1969).

[Spier 73] Spier, M. J., Hastings, T. N., Cutler, D. N., An experimental implementation of the kernel/domain architecture. ACM Operating Systems Review, Vol. 7, No. 4, (October 1973) pages 8-21.

[Walter 75] Walter, K. et al., Structured specification of a security kernel. Inter. Conf. on Reliable Software, SIGPLAN Notices, Vol. 10, No. 6, (Apr. 1975) pages 285-293.

[Weissman 69] Weissman, C. , Security controls in the ADEPT-50 time-sharing system. AFIPS Conf. Proc., 1969 FJCC, AFIPS Press, Montvale, NJ, (1969) Vol. 35, pages 119-133.

[Wulf 73] Wulf, W. A., Shaw, M., Global variables considered harmful. SIGPLAN Notices, Vol. 8, No. 2, (Feb. 1973) pages 28-34.

[Wulf 74a] Wulf, W. A., et al., HYDRA: the Kernel of a multiprocessor operating system. Comm. ACM, Vol. 17, No. 6, (June 1974) pages 337-345.

[Wulf 74b] Wulf, W. A., Toward a language to support structured programs. Computer Science Dept., Carnegie-Mellon Univ., Pittsburg, PA, (Apr. 1974).

[Wulf 76a] Wulf, W. A., London, R. L., Shaw, M., Abstraction and verification in Alphard: Intro. to language and methodology. Tech. Report, Carnegie-Mellon Univ., (June 1976).

[Wulf 76b] Wulf, W. A., London, R. L., Shaw, M., Abstraction and verification in Alphard: Iteration and generators. Tech. Report, Carnegie-Mellon Univ., (June 1976).

[Wulf 76c] Wulf, W. A., London, R. L., Shaw, M., Abstraction and verification in Alphard: A symbol table example. Tech. Report, Carnegie-Mellon Univ., (June 1976).

make color work for you

Color in our Daily Lives, a new consumer booklet from the National Bureau of Standards, takes the reader step by step through the fundamental principles of color and light, families of color, influence of colors upon other colors, and color harmony. This full-color, 32-page illustrated booklet highlights practical applications of color, including:

- Your personal color plan.
- Your color environment.
- Color plans for the home.
- Using color to dramatize or to hide.
- Color and illumination.
- Experimenting with color

This new basic guide can serve as your handbook in helping you make decisions about how to use color in your life and make it work for you. Order *Color in Our Daily Lives* prepaid for $1.70 from the Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402. Use SD Catalog No. C13.53:6.

NBS TECHNICAL PUBLICATIONS

PERIODICALS

JOURNAL OF RESEARCH reports National Bureau of Standards research and development in physics, mathematics, and chemistry. It is published in two sections, available separately:

- Physics and Chemistry (Section A)

Papers of interest primarily to scientists working in these fields. This section covers a broad range of physical and chemical research, with major emphasis on standards of physical measurement, fundamental constants, and properties of matter. Issued six times a year. Annual subscription: Domestic, $17.00; Foreign, $21.25.

- Mathematical Sciences (Section B)

Studies and compilations designed mainly for the mathematician and theoretical physicist. Topics in mathematical statistics, theory of experiment design, numerical analysis, theoretical physics and chemistry, logical design and programming of computers and computer systems. Short numerical tables. Issued quarterly. Annual subscription: Domestic, $9.00; Foreign, $11.25.

DIMENSIONS/NBS (formerly Technical News Bulletin)—This monthly magazine is published to inform scientists, engineers, businessmen, industry, teachers, students, and consumers of the latest advances in science and technology, with primary emphasis on the work at NBS. The magazine highlights and reviews such issues as energy research, fire protection, building technology, metric conversion, pollution abatement, health and safety, and consumer product performance. In addition, it reports the results of Bureau programs in measurement standards and techniques, properties of matter and materials, engineering standards and services, instrumentation, and automatic data processing.

Annual subscription: Domestic, $9.45; Foreign, $11.85.

NONPERIODICALS

Monographs—Major contributions to the technical literature on various subjects related to the Bureau's scientific and technical activities.

Handbooks—Recommended codes of engineering and industrial practice (including safety codes) developed in cooperation with interested industries, professional organizations, and regulatory bodies.

Special Publications—Include proceedings of conferences sponsored by NBS, NBS annual reports, and other special publications appropriate to this grouping such as wall charts, pocket cards, and bibliographies.

Applied Mathematics Series—Mathematical tables, manuals, and studies of special interest to physicists, engineers, chemists, biologists, mathematicians, computer programmers, and others engaged in scientific and technical work.

National Standard Reference Data Series—Provides quantitative data on the physical and chemical properties of materials, compiled from the world's literature and critically evaluated. Developed under a world-wide program coordinated by NBS. Program under authority of National Standard Data Act (Public Law 90-396).

NOTE: At present the principal publication outlet for these data is the Journal of Physical and Chemical Reference Data (JPCRD) published quarterly for NBS by the American Chemical Society (ACS) and the American Institute of Physics (AIP). Subscriptions, reprints, and supplements available from ACS, 1155 Sixteenth St. N. W., Wash. D. C. 20056.

Building Science Series—Disseminates technical information developed at the Bureau on building materials, components, systems, and whole structures. The series presents research results, test methods, and performance criteria related to the structural and environmental functions and the durability and safety characteristics of building elements and systems.

Technical Notes—Studies or reports which are complete in themselves but restrictive in their treatment of a subject. Analogous to monographs but not so comprehensive in scope or definitive in treatment of the subject area. Often serve as a vehicle for final reports of work performed at NBS under the sponsorship of other government agencies.

Voluntary Product Standards—Developed under procedures published by the Department of Commerce in Part 10, Title 15, of the Code of Federal Regulations. The purpose of the standards is to establish nationally recognized requirements for products, and to provide all concerned interests with a basis for common understanding of the characteristics of the products. NBS administers this program as a supplement to the activities of the private sector standardizing organizations.

Federal Information Processing Standards Publications (FIPS PUBS)—Publications in this series collectively constitute the Federal Information Processing Standards Register. Register serves as the official source of information in the Federal Government regarding standards issued by NBS pursuant to the Federal Property and Administrative Services Act of 1949 as amended, Public Law 89-306 (79 Stat. 1127), and as implemented by Executive Order 11717 (38 FR 12315, dated May 11, 1973) and Part 6 of Title 15 CFR (Code of Federal Regulations).

Consumer Information Series—Practical information, based on NBS research and experience, covering areas of interest to the consumer. Easily understandable language and illustrations provide useful background knowledge for shopping in today's technological marketplace.

NBS Interagency Reports (NBSIR)—A special series of interim or final reports on work performed by NBS for outside sponsors (both government and non-government). In general, initial distribution is handled by the sponsor; public distribution is by the National Technical Information Service (Springfield, Va. 22161) in paper copy or microfiche form.

Order NBS publications (except NBSIR's and Bibliographic Subscription Services) from: Superintendent of Documents, Government Printing Office, Washington, D.C. 20402.

BIBLIOGRAPHIC SUBSCRIPTION SERVICES

The following current-awareness and literature-survey bibliographies are issued periodically by the Bureau:

Cryogenic Data Center Current Awareness Service

A literature survey issued biweekly. Annual subscription: Domestic, $20.00; foreign, $25.00.

Liquefied Natural Gas. A literature survey issued quarterly. Annual subscription: $20.00.

Superconducting Devices and Materials. A literature survey issued quarterly. Annual subscription: $20.00. Send subscription orders and remittances for the preceding bibliographic services to National Bureau of Standards, Cryogenic Data Center (275.02) Boulder, Colorado 80302.

END