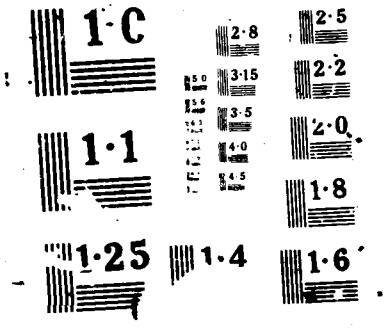
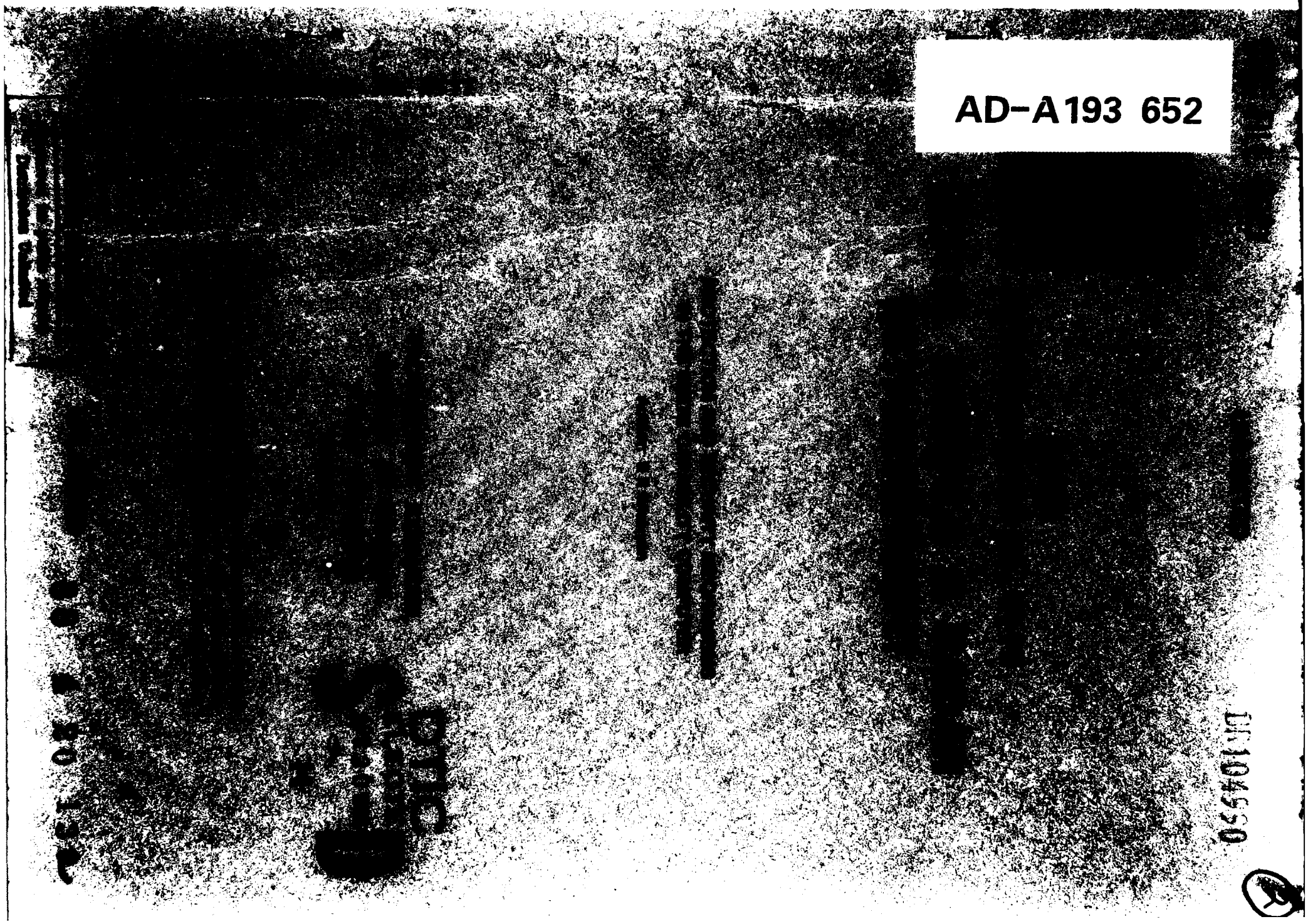
AD-A193 652 PROTECTION AND SECURITY MECHANISMS IN THE SMITE CAPABILITY COMPUTER(U) ROYAL SIGNALS AND RADAR ESTABLISHMENT MALVERN (ENGLAND) S R WISEMAN JAN 88 RSRE-MEMO-4117 DRIC-BR-104990 1/1

UNCLASSIFIED F/G 12/6 NL

END
DATE FILMED
8 88


1·C
1·1
1·25
2·8
3·15
3·5
4·0
4·5
1·4
2·5
2·2
2·0
1·8
1·6



ROYAL SIGNALS AND RADAR ESTABLISHMENT

Memorandum 4117

Title : Protection & Security Mechanisms in the SMITE Capability Computer
Author : Simon R. Wiseman
Date : January 1988

Summary

The SMITE system will support high assurance, yet flexible multi-level secure applications. The SMITE multi-processor capability computer is being developed, based on RSRE's Flex computer architecture, to provide a suitable environment. This paper describes the protection mechanisms provided by the microcoded hardware and introduces the security mechanisms built in software on top of these.

1. Introduction

The SMITE system will support flexible multi-level secure applications that offer a high degree of assurance that the security is upheld. To provide this environment the SMITE multi-processor capability computer is being developed. It is based on the Flex computer architecture [Foster et.al.82] which was the product of RSRE's research into software engineering.

Capabilities are used as the basic means of controlling access to objects. However this is augmented with first class procedures for information hiding and a typing mechanism for authentication purposes. These are combined with a modular compilation system and structured backing store to give a powerful set of primitive mechanisms with which secure systems can be built.

This paper describes the protection mechanisms provided by the microcode hardware and system software, and introduces the security mechanisms built in software on top of these.

2. Pointers

The main memory of the SMITE computer is organised as a heap store. That is, it is divided into discrete blocks, of various sizes, each capable of containing a mixture of scalar data and pointers. A pointer is a primitive capability made up of the address of a block, and one access right. There are various types of block, and certain instructions only apply to particular block types.

SMITE has sixteen types of block. These are used for holding data, constants or instructions, and for representing procedures, abstract typed objects, types, hash tables, processes, semaphores, peripherals and storage resources.

Various instructions exist that create new blocks. The result of these is a pointer to the new block. Apart from copying existing pointers, this is the only way new pointers can be created. The computer maintains a distinction between scalar data and pointers, using hidden tag bits in the memory, thus preventing all software, however 'privileged', from treating scalar data as pointers or from otherwise forgeing pointers.

The SMITE instruction set offers no facility for deleting a block. Instead inaccessible storage is recovered by a garbage collector. This is performed by microprogram although it is invoked by software. The advantage of placing the garbage collector in the firmware, apart from improved performance, is that no software needs to be able to break the rules of capability protection. If it were performed by software, the garbage collector would have to be highly privileged and great care would be needed in controlling this privilege.

A block can therefore only be accessed if the program can obtain a pointer to it. This forms the basis of all protection mechanisms in the computer.

Unlike some other capability architectures, such as the Cambridge CAP [Needham&Walker77] and Intel iAPX-432 [Tyner81], pointers may be freely copied and used in any context without losing their meaning. Thus the computer has one uniform address space, even though the multi processor hardware has separate physical memories, shared by all software. This plays a most important part in the provision of object oriented programming.

Pointers to some types of blocks have an access right associated with them. For example, pointers to blocks used for holding arbitrary data may have the right to write into the block. This right is initially granted, but an instruction

exists to remove it from a particular pointer. A pointer without this right cannot be used to write data into the block to which it refers. Thus it is possible for 'read only' pointers to be used to give away limited access to data.

Such 'read only' access rights do not form a major part of SMITE's protection system, but often form a useful optimization when creating higher level protection mechanisms.

The pointer mechanism ensures that access to an object cannot be gained illicitly, however it does not provide a means of information hiding. The use of the Read Only access right is of limited use in this respect because it does not prevent the user from reading further pointers from the block and writing into the blocks they refer to. Mechanisms to prevent such problems occuring, such as the 'sense keys' of the KeyKos system [Rajunas86], have not been included. This is because the problems can be solved at higher levels of abstraction, as described in section 4.

3. Procedures

Information hiding is achieved using blocks of type **procedure**. Procedure blocks are a closure [Landin64] of the code to be executed and the environment in which it must execute. Procedures are therefore First Class data objects, in that they can be freely passed around and remain valid in any context. Procedures are called by supplying suitable parameters and appropriate results are returned.

Procedures are blocks, two words long, which hold a pointer to the code and constants of the procedure and a pointer to a data block which holds the non-local data that comprises its environment. Neither of these words may be altered and the code is always read-only, though the contents of the environment block could be changed.

All procedures in SMITE are implemented as closures, there is no subroutine mechanism provided to speed up calls to procedures declared in the same module, as is found in all other capability computers. This is because the calling mechanism is quite fast, largely due to the way procedure calling is incorporated into the high level language oriented instruction set.

Procedure blocks may only be called, no instruction exists which will read or write their contents (except for a highly privileged part of the backing store software). The information contained within them is therefore hidden, unless it is made available before the procedure was created or by the code of the procedure.

Abstract data objects may be implemented using procedures to provide the required abstract functions while hiding the underlying data from the user. Some care must be taken to ensure the implementation does not inadvertently pass the user a capability to access the hidden data, especially if exceptions occur, but this is relatively straightforward to verify.

Note that unlike earlier capability computers which have many general purpose registers, such as the Plessey PP250 [England75], each SMITE computer has only one, though this is capable of containing an object of arbitrary size. This register is used to pass parameters to procedures and return results or exceptions. Therefore it is not possible to mistakenly return sensitive capabilities in 'unused' registers.

For example, suppose make_buffer is a procedure which creates a simple buffer. Its environment contains a pointer to the system supplied procedure for

making n-valued semaphores (the instruction set only supplies binary semaphores). The code of make_buffer call this *twice* to create two new semaphores, one to control filling the buffer and the other emptying it. A suitable buffer is then created, along with appropriate index variables for the buffer.

In figure 3a, make_buffer is shown to be a procedure that takes no parameters and delivers a structure of two procedures. The first of these takes no parameter and delivers an integer, while the second takes an integer and delivers nothing.

```
make_buffer : Void -> ( Void -> Int x Int -> Void )
```

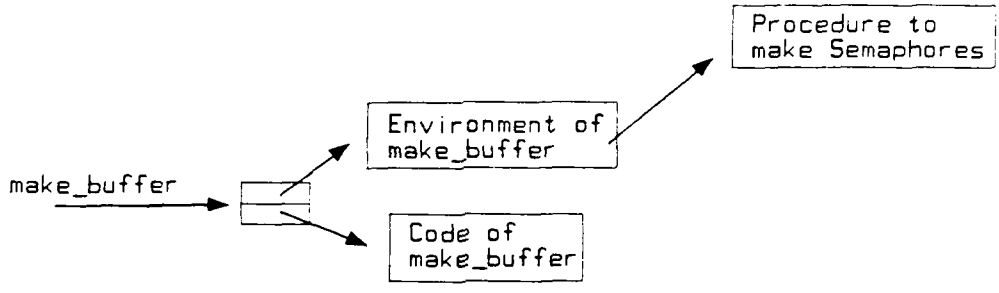


Fig3a: Procedures bind together the code and non-local environment, and hide them from the user.

Next two procedures are declared, these are for moving data into and out of the buffer. The declarations actually consist of some executable code. This creates the environment for the procedures and creates two closures by binding this with the code for put and get. The environment is simply a data block containing pointers to the buffer, buffer index variables and controlling semaphores.

The result of the call of make_buffer is the two procedures, put and get. These may be called to put data into the buffer, or take it out. However they do not give arbitrary access to the underlying data structure of the buffer, because they can only be called.

Note that with buffers implemented in this way, the operations put and get do not take a parameter specifying which buffer to operate on. Instead this is bound into them and for each buffer that has been created, different procedures for put and get are created. This is not inefficient because all versions, including those used by other processes, are able to share the code, since all software operates in one uniform address space.

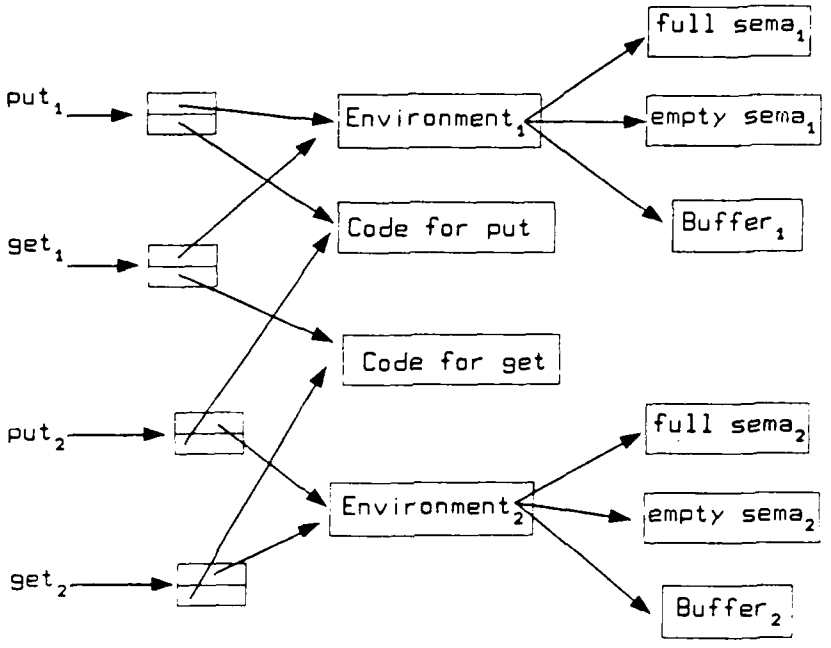


Fig3b: The buffer to be manipulated is bound to the procedures.

4. Typed Objects

While information hiding is provided by procedures, users cannot, in general, determine what kind of procedure they possess. This is because procedures carry no distinguishing type information that can be interrogated by the caller. Without such a mechanism, a procedure received as a parameter cannot be passed sensitive data, in case it is not of the correct type and misuses that data. This severely restricts the functionality of the abstract types that can be implemented securely.

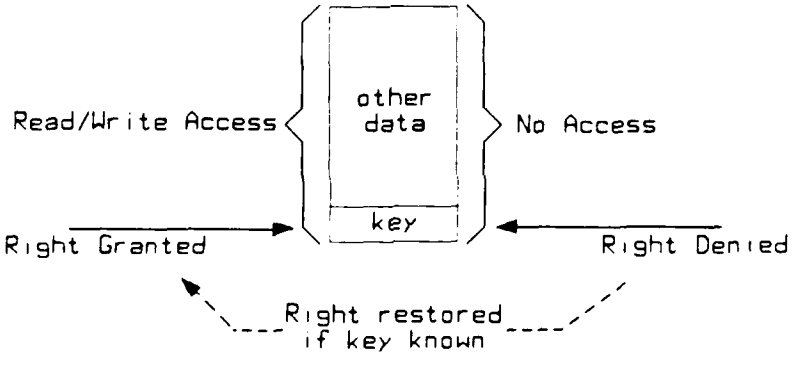


Fig4a: The right to access data in a keyed block can be denied. However this right can be restored if the key, which is generally a pointer, is known.

Typed objects can be provided by using Keyed Blocks. Pointers to a keyed block have an access right associated with them. This is the right to access the data in the block. Without this right no access is possible, not even read, though there is an instruction which will grant the right and allow full read/write access. However, this Open operation must be provided with the key to the block, otherwise it will fail. The key is a single word value, which is stored in the first word of the block.

If the key were a scalar value, the user could quickly try all 2^{32} values and gain access to the protected data. However the key may be a pointer and these cannot be forged or guessed. The pointer will be kept hidden by the Type Manager software, which will use it to create new data objects of the type and to gain access to the underlying data of objects passed to it.

The keyed block mechanism allows type information to be attached to an object and hides the underlying data from its users. The type is represented by a pointer to some block. This pointer is used as the key for all keyed blocks which represent objects of that type. Procedures which create the typed objects and access their representation keep the key in their non local environment and do not disclose it.

For example, consider an alternative form of the implementation for the simple buffer described in section 3, in which a buffer is made a typed object. Four procedures would be supplied, for creating new buffers, moving data to and from a buffer and for deciding whether an object is a buffer. Note that the latter procedure will be needed only if the programming language does not provide type abstraction or cannot be trusted to enforce it properly.

```
make-buffer : Void -> Buffer
put : Buffer * Int -> Void
get : Buffer -> Int
is_buffer : Buffer -> Bool
```

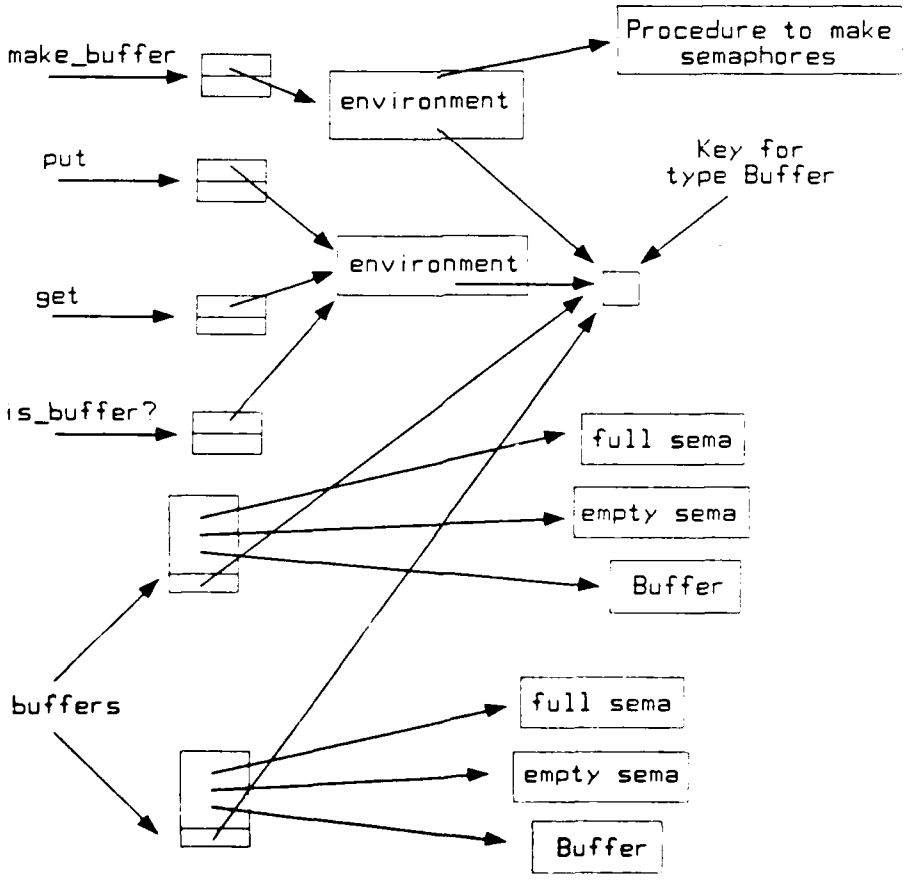


Fig4b: Keyed blocks hide the representation of a buffer and attach type information. Procedures hide the key to the type.

The procedures for manipulating buffers are created when the type manager is

brought into existence. At this time a key is made which will represent the type 'buffer'. This key is simply a capability, which is created by generating a new block of store, which is not distributed to any but the four buffer procedures. In this example the block is not used to hold data and so will have zero size.

The procedures put, get and is_buffer all have the same environment, which is simply a block containing the key (a pointer) that represents the type Buffer. Procedure make_buffer also requires a pointer to the procedure for making n-valued semaphores.

A call of make_buffer generates space for the buffer and creates the controlling semaphores. Then a keyed block is generated and pointers to these are stored in it. The buffer type key is stored in the first word and a pointer to the new keyed block, with the access right denied, is delivered as a result of the call. No access to the underlying data that implements the buffer is permitted using this pointer. The access right may be restored if the key can be presented, but this is not given to the users of the type. Therefore users cannot gain access to the underlying data structure of the buffer.

The procedure is_buffer takes a pointer as a parameter. It attempts to grant the access right on the pointer, supplying the buffer key as a further operand to the instruction. If this succeeds the procedure simply returns true. If the instruction fails, either because the pointer did not refer to a keyed block or the key was wrong, false is returned. Note that the pointer which has the access right granted is not accessible to the caller.

The procedures put and get act similarly, first granting the access right on the pointer to the buffer and then performing the appropriate operation on the buffer. Note that, unlike the example given in section three which used procedures which had a buffer bound into them, the buffer to be manipulated must explicitly be passed as a parameter.

If a subsystem expects to receive a pointer to a buffer as a parameter, it can check that the pointer really does refer to a buffer using is_buffer. It can then use the buffer, safe in the knowlege that it will obey the rules about buffers. If the buffer was implemented using procedures, as in section 3, it would not be possible to determine whether the parameter (a procedure) gives access to a buffer.

The protection provided by keyed blocks is equivalent to seals [Redell74]. However the implementation in SMITE is quite straightforward, since the type information is held in the object rather than in the capabilities (pointers). Therefore the pointers are all of fixed length, occupying just one 32 bit word.

5. Module Loading

Programs on SMITE are constructed using a modular compilation system which can allow mixed language working. However, programs are not 'linked' to form executable images which are then copied into store and run. Instead a form of linking loader is used which is much better suited to the use of capabilities. It also has the advantage that out of date images cannot be run.

A module is essentially a procedure which creates some data structure and delivers a capability to access it. A module may require the values created by other modules, but each module in a program must only be loaded once. This is achieved by passing a Loader to a module. The Loader is a procedure which, given a module, returns the value it keeps. The Loader records the value kept by each module it loads. If it is requested to load a module again, it simply returns the value associated with it.

A module must ensure that the parameter it is given is a proper Loader, and not just some arbitrary procedure even if it appears to act in the same way. This is because a program consisting of several modules may perform checks in one module and perform sensistive operations in another, based on the result of the check. By providing a special spoof loader, a user could construct a program which incorporated a fake checking routine, which would allow illegitimate use of the sensitive operation.

This could be prevented by typing the Loader using a keyed block. However, since modules are invariably brought in from backing store, so must the procedure for checking the type. This would seriously slow up the system, so special provision is made for the type Loader in the instruction set.

The ability to create a Loader is carefully controlled, whilst changing a Loader into a procedure of type Module $\underline{X} \rightarrow \underline{X}$ is an instruction that is universally available. In practice only the trusted type manager for loaders may produce new loaders. When a module needs to access the data structure kept by another, it first checks that the procedure it was handed as a parameter is a true Loader. The instruction to do this fails if it is not, but otherwise delivers the underlying procedure. This is called, passing the module to be 'loaded' as a parameter. The result is a read-only pointer to the object kept by the module.

A more detailed description of the SMITE modular compilation system, including details of language specific modules, in given in [Harrold88].

6. Modes

A frequently used form of abstract data type is where a copy of part of the underlying object is freely accessible. The implementation of Mode objects follows this pattern, in that the mainstore representation of a mode may be read at any time, while the alias on backing store must be kept hidden to prevent forgery.

A mode, which is the Algol68 term for type, is treated as an object in the Ten15 algebraic abstract machine [Core&Foster86], which is a generalisation of the ideas developed during the Flex project. Ten15 is effectively a strongly typed language system whose types include the universal union of all types. Mode objects are provided to implement this in such a way that the users cannot fabricate their own illegal modes, yet modes may be examined.

Mode objects could be implemented using a procedure which delivers the accessible part of the object while keeping the rest hidden, however they are made a special case because they are often used. A special kind of block, similar to the keyed block is provided. Here the access right gives full read/write access, but if a pointer is denied the right it can still be used to read all but the first two words of the block. The first word is the key for the block and the second word is the protected data. This may be a pointer to further protected data.

7. Revocation

Most capability systems provide a low level mechanism for revoking capabilities once they have been distributed. The use of such mechanisms can lead to complications, especially in parameter validation. This is because the mechanisms provided revoke access to data blocks, which are at the lowest level of abstraction. What is generally required is to revoke access to highly abstract objects, such as files and messages. By using a low level recovation mechanism, the high level object becomes inconsistent, as parts of it are removed, which causes obscure exceptions to be raised.

In SMITE revocation can be provided where it is needed, by building appropriate checks in the procedures which access the underlying data of an object. This is in contrast to the unwieldy low level mechanisms that have been proposed by others [Redell&Fabry74],[Gligor79],[Corsini et.al.84].

An example of revocation is found in classified files. Access to the underlying file is only permitted if the user has the necessary clearances. Whenever users attempt to gain a copy of the classified material in a file, a check is made to ensure that they have sufficient clearances. Revocation can be achieved by reducing a user's clearance or altering the file's classification or distribution list.

8. Proces Context

Processes running in a system generally need access to generic resources, such as "my terminal" and "my current directory", which are provided by the operating system. In SMITE capabilities are used to control access to such resources, which are constructed as abstract data types. However, it is still necessary for a program which is loaded from backing store to gain access to its version of these capabilities.

The capabilities that refer to these generic resources are stored in the context of each process. The context is an association between unique identifiers and values. Each unique identifier represents one of the generic resources. An instruction allows a program to find out the current value attached to a particular identifier, and a special form of procedure call allows the context to be changed and restored in a stack like manner.

The unique identifiers will normally be pointers, so that they cannot be forged, which are kept hidden in much the same way as type keys. The capability for a generic resource will be obtained by calling a procedure that hides the key. This may be formed into a module so that it can be readily incorporated into programs.

When a process is launched, its inital context is set to that of the process which launches it. The lifetimes of processes are independent, therefore care must be taken when designing generic resources to ensure that they can safely be accessed after the process leaves the scope that designated them.

For example a procedure call may establish a new window as the "current window". When it exits the previous window will be restored to "current". Suppose a process was launched during the call and this remains active after the window is restored. This process will have inherited the new window and can access it even after it is supposed to be destroyed. Such disasterous results can be avoided easily by incorporating a revocation switch in the window description.

9. Write Once Structured Backing Store

SMITE provides a capability based structured backing store, organised in a write once fashion. Thus the backing store is a heap, able to store primitive objects (blocks of various types) much like main store, except that they cannot be overwritten. Procedures and an equivalent to keyed blocks may be stored, providing abstract type extension facilities.

The backing store is implemented entirely in software, using the abstract type mechanisms for protection. Access to objects in the backing store is governed by backing store capabilities, which are data objects constructed by the software. These are implemented as keyed blocks which contain the disc address and type information of the object on disc.

When a program stores data in the backing store it does not provide a destination address. Instead the backing store software places the data at some convenient free place and returns a backing store capability which can be used to access the data in the future. A program can only bring an object into main store if it posseses a capability for it. The result of doing this is a main store capability to a copy of the object in main store. This capability has read/write access revoked, so the copy cannot be altered. This facilitates sharing of data brought in from disc.

The backing store software does not provide an operation for overwriting objects. However, special objects called references, which can be atomically updated, are provided to support alterable objects such as directory structures and updatable program modules.

The advantage of such a backing store is that maintaining consistency is relatively simple. Directory structures can be built which cannot be damaged, even if the event of power failure. Also, since a garbage collector is used to recover inaccessible variables, complex structures can be implemented and maintained without "dangling reference" problems arising.

The write once organisation has a particular advantage for secure systems. Objects stored in the backing store are guaranteed to be unalterable, as long as they do not contain a reference. It is possible, using the other security mechanisms, to ensure that certain programs cannot use references. Therefore backing store objects produced by such a program cannot be used to communicate with that program.

In other words, untrusted software can share access to objects in backing store, and it is impossible for one to modify the object and thus send information to the other. Therefore the system offers separation between untrusted programs while having unrestricted sharing of backing store objects. Conventional systems cannot offer such flexibility because they do not provide a non overwriting file store.

Details of the implementation, including the on-the-fly garbage collection are given in [Wiseman88].

10. Security Mechanisms

The provision of multi-level computer security presents three separate problems: controlling access to classified information, preventing users being fooled into underclassifying information and preventing software altering access controls against the user's wishes. SMITE offers three solutions to these three problems: reference monitors, high water marks and the trusted path. A more detailed explanation is given in [Wiseman86a&b].

10.1 Reference Monitors

Reference Monitors are the most visible security mechanism in SMITE as they are responsible for controlling access to classified information. A reference monitor is placed between each function that accesses classified information and the users of that function. Its task is to check that the user is cleared to access the information, record the details of the access for later auditing and finally to perform the access.

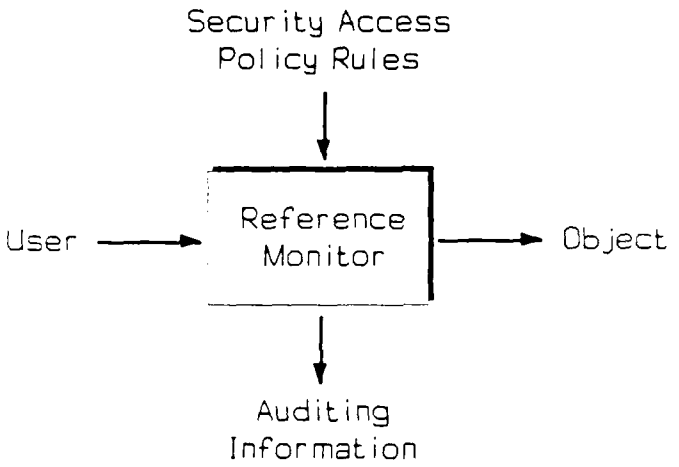


Fig10.1: A reference monitor encapsulates a classified object. It ensures that the users have the necessary clearances to access the information it contains and maintains auditing information.

The reference monitors are implemented as an abstract type whose underlying type is the original object. The interface provided is the same as that of the underlying object. That is the security checks are hidden from the user. This can be implemented using either procedures or keyed blocks as appropriate.

10.2 High Water Marks

While reference monitors control access to classified information, they do not prevent erroneous software from accessing data which the user did not wish to be accessed. For example consider a user who is cleared to access Secret information but is creating a document which is to be Unclassified. "Trojan Horse" software may access some Secret information and incorporate this into the document without the user realising. This would cause the user to distribute the Secret information in a document marked Unclassified. In effect the "Trojan Horse" has fooled the user into underclassifying information.

To prevent this from happening a system of High Water Marks is maintained for objects created by untrusted software. That is, the system maintains the classification of the most classified information that the object could conceivably contain. If the user gives these objects a permanent

classification, a check is made to ensure that this dominates the object's high water mark.

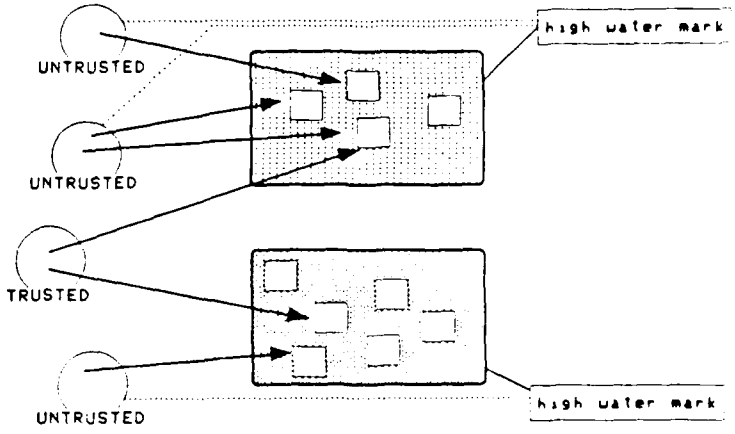


Fig10.2: Objects accessed by untrusted software all have the same high water mark.

Maintainance of the high water marks is the responsibility of the reference monitor functions. Whenever untrusted software accesses classified information, the reference monitor ensures that the caller's high water mark is increased accordingly.

10.3 Trusted Path

Some functions that the users of a system need to perform alter the access control information. That is they affect which users can access what information. Functions in this category include giving new information its classification, regrading existing information, altering access control lists or distribution lists and changing users' clearances.

While a reference monitor checks whether the user is allowed to invoke such a function, it cannot check that the software invoking it is doing so because the user requested it or because it contains a "Trojan Horse". To exclude the latter, these functions must only allow themselves to be invoked by the Trusted Path. This is interface software which is guaranteed not to contain a "Trojan Horse". It will at least comprise all software that controls the screen, keyboard and mouse, along with menu and window software.

The use of the type abstraction facilities offered by the SMITE architecture allows the trusted path to be implemented as a set of small independent modules. This will allow code level proofs of correctness to be tackled, giving the necessary high degree of assurance.

11. Summary

The microcoded hardware provides capability protection in the main memory of a SMITE computer. Possession of a capability for an object entitles the holder to access it, while it is impossible to access it without a capability. The objects in memory are typed, so each type of object is accessed in different ways. In particular procedures and keyed blocks are offered as mechanisms for information hiding and for creating user defined abstract types.

These primitive protection mechanisms allow software to be written to provide flexible security mechanisms. Moreover, the fine granularity of protection

allows the different concerns of security to be split up, allowing correctness to be established more easily, yet without loss of performance.

Acknowlegements

Thanks go to Michael Foster, Ian Currie and numerous others at RSRE who were involved in the Flex project, for the basic research which is being used as the foundation for SMITE. Also, special thanks are due to Phil Terry of TSL Communications, who has contributed a great deal towards understanding security issues within SMITE, and Peter Bottomley for commenting on earlier drafts of this paper.

References

P.W.Core & J.M.Foster
Ten15: An Overview
RSRE Memo 3977, September 1986

P.Corsini, G.Frosini & L.Lopriore
Distributing and Revoking Access Authorizations on Abstract Objects: A Capbility Approach
Software - Practice and Experience, Vol 14, Num 10, pp931..943
October 1984

D.M.England
Capability Concept Mechanisms and Structure in System 250.
Rev. Fr. Autom. Inf. Rech. Oper. (France)
Vol 9, Sept 75, pp47..62

J.M.Foster, I.F.Currie & P.W.Edwards
Flex: A Working Computer Based on Procedure Values.
RSRE Memo 3500
Also in: Procs. Int. Workshop on High Level Language Computer Architecture
Fort Lauderdale, Florida, December 1982

V.D.Gligor
Review and Revocation of Access Privileges Distributed through Capabilities
IEEE Trans Software Engineering, Vol SE-5, Num 6, pp575..586, Nov 1979

C.L.Harrold
The SMITE Modular Compilation System
to appear
1988

P.J.Landin
The Mechanical Evaluation of Expressions
Computer Journal, Vol 6, Num 4, pp308..329, January 1964

R.M.Needham & R.D.H.Walker
The Cambridge CAP Computer and its Protection System.
Operating System Reviews
Vol 11, Num 5, Nov 77, pp1..10

S.A.Rajunas, N.Hardy, A.C.Bomberger, W.S.Frantz and C.R.Landau
Security in KeyKos
Procs. IEEE Symposium on Security and Privacy
Oakland, California, April 1986

D.D.Redell
Naming and Protection in Extendible Operating Systems
MIT Project MAC Technical Report MAC-TR-140
November 1974

D.D.Redell & R.S.Fabry
Selective Revocation of Capabilities
procs. IRIA Workshop, pp 197..209
1974

P.Tyner
iAPX-432 General Purpose Data Processor: Architecture Reference Manual
Intel Corp. Jan 1981

S.R.Wiseman
A Secure Capability Computer System
Proceedings IEEE Symposium on Security and Privacy
Oakland, California, April 1986

S.R.Wiseman
A Capability Approach to Multi-Level Security
Proceedings IFIP/Sec'86
4th International Conference on Computer Security
Monte Carlo, Monaco, December 1986

S.R.Wiseman
The SMITE Object Oriented Backing Store
to appear
1988

DOCUMENT CONTROL SHEET

Overall security classification of sheet ..... UNCLASSIFIED .....

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the box concerned must be marked to indicate the classification eg (R) (C) or (S))

1. DRIC Reference (if known)	2. Originator's Reference MEMO 4117	3. Agency Reference	4. Report Security Classification U/C
5. Originator's Code (if known) 778400	6. Originator (Corporate Author) Name and Location RSRE St Andrews Road, Malvern, Worcs. WR14 3PS		
5a. Sponsoring Agency's Code (if known)	6a. Sponsoring Agency (Contract Authority) Name and Location		

7. Title
PROTECTION AND SECURITY MECHANISMS IN THE **SMITE** CAPABILITY COMPUTER

7a. Title in Foreign Language (in the case of translations)

7b. Presented at (for conference papers) Title, place and date of conference

8. Author 1 Surname, initials	9(a) Author 2	9(b) Authors 3,4...	10. Date	pp. ref.
Wiseman S R			1988 01 ~~1987 10~~	14

11. Contract Number	12. Period	13. Project	14. Other Reference

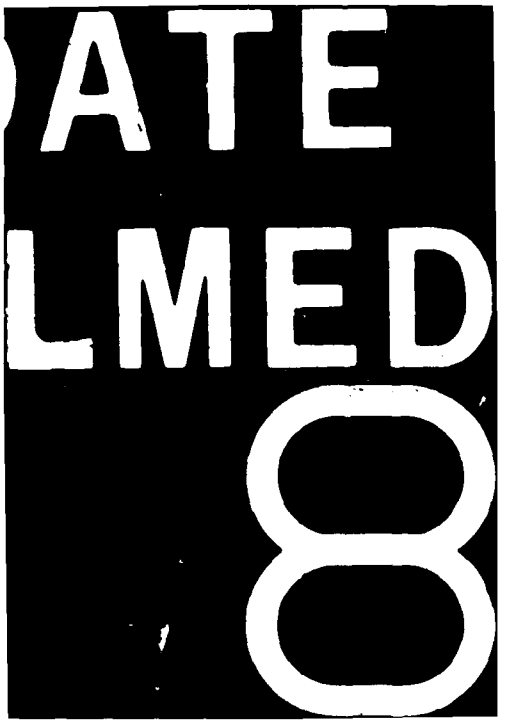
15. Distribution statement

Descriptors (or keywords)

continue on separate piece of paper

Abstract

The SMITE system will support high assurance, yet flexible multi-level secure applications. The SMITE multi-processor capability computer is being developed, based on RSRE's Flex computer architecture, to provide a suitable environment. This paper describes the protection mechanism provided by the microcoded hardware and introduces the security mechanisms built in software on top of these.


ATE
LMED
8