BR113305

UNLIMITED

2

Report No. 89024

Report No. 89024

ROYAL SIGNALS AND RADAR ESTABLISHMENT,
MALVERN

AD-A221 940

BASIC MECHANISMS FOR COMPUTER SECURITY

Author: S Wiseman

PROCUREMENT EXECUTIVE, MINISTRY OF DEFENCE
RSRE
Malvern, Worcestershire.

January 1990

UNLIMITED

ROYAL SIGNALS AND RADAR ESTABLISHMENT

REPORT 89024

Title:	**Basic Mechanisms for Computer Security**
Author:	**Simon Wiseman**
Date:	**January 1990**

Abstract

The protections facilities required for computer security are expressed as four basic mechanisms. It is shown how a security model maps to these mechanisms and how they can be implemented on conventional architectures, capability architectures and in high level languages.

Introduction

Computer architectures provide various kinds of protection mechanism and these can be used in numerous ways to implement secure systems. Rather than attempt to categorise these possibilities, this paper describes four basic mechanisms which it claims are their essential essence and are sufficient for implementing secure systems.

It is proposed that these four basic mechanisms are used as an intermediate step in the refinement of design to implementation. Consequently, the designers of a secure application do not need to concern themselves with the minute detail of the protection facilities of the target machine while working at the higher levels of abstraction. The realisation of the four basic mechanisms on any particular computer hardware is of course concerned with fine details, but this need only be considered once for each target computer.

Splitting the implementation process into two steps in this way, not only simplifies it but should also make secure applications more portable. This is because conventional techniques lead a design to depend on the detail of the target hardware's protection facilities, which locks a design to a particular configuration.

The presentation is in terms of the security policy model of [Terry&Wiseman89]. This not only considers the confidentiality aspect of security, but covers integrity as well. The model is general purpose and can be applied to a wide variety of secure systems, as described by [Harrold90].

This paper shows how the elements of the Terry-Wiseman model can be mapped onto the proposed mechanisms, thus showing that the mechanisms are sufficient to implement any secure computer system. The proof obligations that arise from the refinement of model to mechanisms are considered. Implementation techniques are then explored for each of the mechanisms in turn, covering the use of conventional, capability and high level language architectures.

Overview of the Model

The Terry-Wiseman security model considers that systems comprise a number of entities, each with some control attributes and some functionality attributes. The control attributes of entities can be observed at any time with no constraint, but functionality attributes can only be observed via an entity, and such observations are subject to the entity's control attributes. One control attribute is the classification which controls confidentiality, others control the integrity of the controls.

A potential covert channel exists through the control attributes of entities. However the model acknowledges this and requires that requests which create or destroy entities, or which modify an existing entity's controls, can only be made by entities which are trusted not to exploit the signalling channel.

Both entities and attributes will be implemented as machine level objects, but it is vital to preserve the abstraction that functionality attributes can only be observed via entities. This is because if changes to an entity's functionality attributes can be detected without observing an entity, a covert channel is introduced, as the implementation has effectively introduced a new means of communication which has not been considered by the security analysis.

A system is modelled as a state machine, with axioms ensuring that all transitions preserve confidentiality and integrity. Transitions occur at the request of a set of entities, called the requestors, thus allowing separation of duty[1] [Clark&Wilson87] to be specified for the modification of important information.

Essential Mechanisms

The four essential mechanisms are called: 1.unforgeable opaque addressing, 2.data hiding, 3.pedigree and 4.context sensitive addressing. This section introduces the mechanisms in turn and briefly justifies why each is necessary. Discussion about how they may be implemented is deferred until later.

Unforgeable, Opaque Addressing

The abstract view of an entity and its attributes will inevitably be implemented at a lower level of abstraction as one or more machine level objects, for example memory segments. If these objects are created and destroyed as the attributes of the entity change this must have no effect on any other entity, otherwise the implementation has introduced a covert channel. There are a variety of ways in which this rule could be broken and these must be avoided.

1. by guessing an address and trying to access an object, its existence is detected if it is possible to distinguish between "object does not exist" and either "access denied" or "success".

2. if the addresses are specified by the caller, the creation of an object will be detected by an attempt to create another object with the same address.

3. if addresses are automatically generated and if the address of an object can be examined, it may be possible to determine whether some other entity was created between two other consecutive creates. For example, if addresses are simply increasing integers, examining them reveals if other objects have been created.

4. it may be possible to detect the usage of the shared resource which is consumed when the object is created. For example a command may allow the amount of free space to be determined. Alternatively, since the resource is inevitably finite it may be exhausted. Unfortunately exhaustion of a shared resource can never be hidden and so creation of an object can always be detected in this way. However, it can be argued that this channel is small and, since exhaustion does not happen in normal usage, action can be taken to prevent it being exploited. The alternative, that is not sharing resources, perhaps by imposing a strict quota system, has a serious impact on resource utilisation and so should only be used in cases of extreme need.

5. in some cases the effect of an address may be modified in a way which can be detected by other users of the address. A well known example of this is forcing segments in a paged memory system to be in either primary or secondary memory. The difference in access time to the page may then be detected. Similarly, if the free list of a shared pool can be ordered by selectively deleting other objects, observation of addresses used to allocate new objects can detects the activity of others. Other examples arise when objects may be explicitly deleted or access rights can be revoked.

Note that these problems only arise if resources are dynamically allocated when attributes are modified. However this will be the case for most complex secure systems so it is necessary to have a mechanism for creating objects which avoids these problems. This will be called the unforgeable, opaque addressing requirement. This is because, while addresses can be freely copied, it must not be possible either to construct

[1]This is not a new idea, automation of segregation of duties to control fraud was proposed by [Linden76].

them from scalar data, that is they cannot be guessed or forged, or to detect anything about their inner workings.

Data Hiding

The abstract notion of observing or modifying an entity's attributes corresponds at the lower level of abstraction to accessing the contents of the machine level objects which implement the entity and its attributes. Obviously, in order to enforce security this access must be controlled.

One possibility is to limit the ability to address an object to those who are authorised to access the contents. However, this is very limiting because the access control check would have to be made when the ability to access was granted. This would prevent access control checks from being based upon dynamic parameters, for example it may be desirable to allow access only at certain times of the day. A more fundamental problem is that if access is controlled by the propagation of access permissions, it is generally not possible to decide whether an undesirable state can be reached [Harrison*et.al.*76].

Thus the implementation of entities and their attributes requires a mechanism that allows a machine level object to be referenced without necessarily giving access to its contents. The requirements for the control of access, however, are very varied and the mechanism must be general purpose. Its main function is to hide the data in the object from direct manipulation, thus a data hiding facility is called for. This allows an object to be freely referenced, but only permits access to its data by code which is written by the object's creator. This code is responsible for ensuring that the data in the object is only manipulated in an approved way.

Another use of data hiding is, where several entities have the same attribute, space and time can be saved by using the same machine level object for each. However, if such an optimisation is made, it is necessary to control access to the object to ensure that it can never be modified once it is created. If this were not done, the abstraction that attributes can only be observed or modified via an entity would be broken, effectively introducing covert channels.

Also for flexible operation it is often desirable to divorce the ability to address an object and the ability to access its contents. This allows entities to be referenced by those who are not cleared to access their contents. For example an electronic mail service needs to reference the text of messages so it can deliver them, but it does not require access to the text, only the header information.

So for entities, and their unshared attributes, the data in the objects which implement them must be hidden by code which ensures that confidentiality and integrity are upheld. For shared attribute objects the code must ensure they can never be modified.

Pedigree

Separation of duty can be employed to ensure that any modifications to important information leaves it in an appropriate state. For example, changing the classification of an entity may only be possible when two people, with sufficiently conflicting roles, agree that the change is reasonable. The software acting on behalf of these people must, of course, faithfully represent their views.

If some people agree to make an important modification to an entity, they must be assured that the change which occurs is the one they sanctioned. For example, if they agree to downgrade document x, the system must not downgrade document y, which could occur if some malicious software creates an entity which it claims is a document.

To do this, the malicious software would construct a low level object whose contents are protected by data hiding, so it is not possible to look inside it and discover that it is a forgery. The interface offered would appear to be that of document **x**, except that the downgrade operation would be that of document **y**. Thus the reviewer of the fake document would see the text of **x**, and having agreed that the downgrade is reasonable, would authorise the downgrade, with the unfortunate effect that downgrade of document **y** would be approved instead.

The problem is avoided by using a mechanism that establishes the origin, or pedigree, of an object. The ability to attach a pedigree mark to an object is limited to the authority responsible for creating a particular kind of entity. By inspecting the pedigree, before deciding to authorise modification of important information, the authenticity of the entity can be established and such spoofs are avoided.

The pedigree mechanism could simply determine whether or not a machine object represents an entity. Alternatively it may distinguish between different types of entity, for example documents and messages. This depends on whether the decision to modify an entity is dependent on its type.

Context Sensitive Addressing

In an abstract specification of a transition, a set of requestors will access other entities. The specification will call for some access control decisions and will probably ensure that a journal of accesses is maintained. These actions are based upon details of the requesting entities, such as their classification and identity. Thus in practice the requestors will be observed.

At the concrete implementation level, requestors will be implemented as processes and entities as machine level objects protected by data hiding. The operations which hide the data are responsible for making the access control checks and updating journals. Therefore they must have access to information about the requestors.

However, the code which invokes the access control code may not attempt to voilate the system's security constraints and so cannot be relied upon to give correct information. Therefore a mechanism is required which allows the access control code to gain the information without recourse to the invoking code. The mechanism must allow the access control code to ask for this information without relying on the invoker in any way. That is context sensitive addressing is needed.

A context sensitive address is one which references a different object depending upon the context in which the address is used. For example, "tt0", "tt1", etc. may be addresses of terminal ports in an operating system. Programs, however, use the context sensitive address "tty" to access the current terminal port of the user who runs them. The context which maps "tty" to the appropriate terminal port is set up when the user logs on.

The objects which can be accessed through the context may use data hiding to protect themselves from unauthorised modification. However, the ability to set up a context sensitive address, or replace which object is referenced by one, must be controlled. This can be done by using data hiding to protect the addresses. For example, software is allowed to call an operation which returns the "current clearance". This hides the context sensitive address for the "current clearance" and so untrusted software cannot directly ask for, or alter, the current clearance.

Mapping the Model to the Mechanisms

The previous section developed the four mechanisms and partly justified them by showing how they were necessary to implement the model. In this section the mapping between abstract model and implementation mechanisms is made more explicit.

Entities must be implemented so that observation and modification of their data attributes and modification of their control attributes are controlled, though their control attributes may be freely observed. To protect an entity in this way requires its underlying representation to be hidden behind operations which preserve its abstract specification. However, access to these operations need not be controlled, because determining the state of an entity's access controls, including its existence, can never yield classified information.

So data hiding is used to implement an entity and its attributes, with the operations protecting the entity's attributes from arbitrary observation or modification. If the operations dynamically allocate machine level objects for any reason, unforgeable opaque addressing must be used to reference them, because otherwise their existence could be determined leading to a covert channel. Further, if any machine level object is shared between entities, to save resources used for common attributes, data hiding must be employed to guarantee that the attribute's representation can never be altered.

Note that the model ensures that an entity's controls, including knowledge of its existence, are always unclassified. This is achieved by insisting that controls are only modified by software which is trusted to not encode classified information in them. Therefore it is not necessary to use unforgeable, opaque addressing to reference an entity.

The integrity of an entity's controls is preserved by imposing n-man rules on their modification. In general this will be implemented as two phases. First, the representation of the entity is examined to enable a decision to be made about whether to agree to the modification. Second, the modification is authorised. Once sufficient authorisations have been collected to satisfy the separation of duty constraints, the modification is made.

It is necessary to assure the person authorising the change that the data they examined and authorise operation they invoked are both part of the implementation of the same entity. To this end it must be possible to determine the pedigree of the object which allegedly implements the entity.

The pedigree can be determined either statically or dynamically depending on circumstances. For example, if files are addressed using a separate address space, such as names in a directory, pedigree is established statically by virtue of their location. However, if file objects are referenced as part of a uniform address space, a dynamic pedigree mechanism must be employed.

Entities that request transitions are implemented as processes, whose data attributes are protected using data hiding. The control attributes of such active entities are placed in the process context, which allows their classification and other control attributes to be determined by the access control code of entities they access.

When transitions require more than one requestor, to preserve integrity, the processes are made to cooperate in some application specific way. For example, extra variables may be introduced to record the wishes of each requestor individually. Only when enough agreement has been reached is the abstract state of the entity altered. For example, a two-man rule to downgrade a document would be implemented by a request to downgrade followed by an authorise downgrade request. Auxilliary variables would record the downgrade request as pending until the authorisation was given.

Thus it has been shown informally that an application whose security specification is expressed in terms of the Terry-Wiseman model can be mapped to the four basic mechanisms, independently of how these are to be implemented. It remains to be shown that these mechanisms are sufficient and it is as yet unclear whether a better set of mechanisms exists, however these questions can only be answered by trying to use the approach.

Refinement

A security policy model is a specification of all possible systems that are secure. It is a very general statement which says very little about what a system should do, as it is really concerned with what it should not do. A subset of these possible secure systems will have the properties required for a particular application. This subset is specified by the application's functionality specification. Using the Terry-Wiseman approach both the security model and functionality specification are expressed in the same way, so an application is simply specified as functionality AND security.

A system specified in this way is always, by definition, secure. However it may not have the expected functionality, because some operations may not satisfy the constraints imposed by security. Whether this lost functionality is important to the customer can only be determined by validating the specification [Sennett89].

For example, the system's functionality specification may define an update transition on entities **a** and **b** that will cause **a**'s attributes to be moved to **b**, without reference to their classification. However, the security policy model only allows this if **b**'s classification dominates **a**'s. So some transitions specified by the functionality cannot occur. It may well be that the customers are happy with this, but more obscure examples might cause problems and mean the specification has to be reworked.

It can be argued that it is undesirable for an application's requirements to be specified using the Terry-Wiseman approach as this gives the specification an implementation oriented bias. So the requirements may well be specified at an abstract level, followed by a refinement to a more concrete specification in terms of entities and attributes. The validity of this refinement does not affect the security of the system, but it does of course impact upon its correct operation.

The previous section showed that having specified the security and functionality requirements in terms of entities and attributes, this could be mapped to the four implementation mechanisms. This is an example of a refinement from a specification involving abstract operations applied to abstract objects to an equivalent one involving more concrete operations and objects.

In general a refinement preserves the properties of the more abstract level, but introduces more detail. In safety critical systems this extra detail is unimportant, but in secure systems it could be used for illegal information flow. It is therefore necessary to either apply the security model again at the lower level to confirm that no security flaws are introduced or to ensure that the refinement preserves the abstraction completely.

For example, a specification of a fly by wire system would state that when the stick is pushed left, the aircraft must bank left, and when it is pushed right it banks right. The refinement would introduce extra detail about the movement of flaps and state of cockpit indicators. However, a valid and safe refinement could also affect the status of the cabin signs, illuminating the no smoking lights when banking left and the fasten seat belt lights when banking right. Such a refinement would be unacceptable in a secure system, because information now flows from the cockpit to the cabin in the implementation whereas it did not in the abstract specification.

Refinement takes a specification of abstract operations acting upon abstract objects and produces an equivalent specification of concrete operations acting upon concrete objects. If each abstract object is implemented using data hiding to hide the concrete objects which represent it, effectively no extra detail is introduced. Of course, this is a property that needs to be proven about the data hiding mechanism, but this only has to be done once for each target machine. The implementation of the abstract object's specification can then use the relatively well understood techniques for safety preserving refinement [Morgan*et.al.*88].

The security model specifies how the flow of attributes between entities should be controlled. In order to do this accurately it is necessary to show which entities are 'observed' and which are 'modified'. These are properties given in the abstract specification which must be upheld by the refinement.

Showing that an operation does not modify an entity is relatively straightforward, it is simply the property that the entity's state after equals the state before.

Showing that an entity is not observed is more difficult. In the first instance those entities which are not addressed are obviously not observed, though this is of course a property of unforgeable, opaque addressing and data hiding that must be proven for the target hardware.

Cases where an entity is addressed but not observed, effectively pure modification without observation, are rare. The most obvious example is updating an entity containing audit information. This has to be classified top to allow any user to modify it and so it is necessary to show that nothing about the original state of the audit trail is observed. This entails proving that the result of the operation does not depend upon the original contents of the entity, which is particularly easy if a nil result is always returned. Note that the exhaustion of shared resources would constitute observation of the entity's contents, though a special case might be made for this as it should not happen in the normal course of events.

Thus the refinement from security specification to implementation is greatly simplified by directly mapping the entities of the security model to the protection mechanisms of the machine. Proof that the mechanisms are implemented correctly are required, but this need only be performed once per machine, rather than for each application.

Implementing the Mechanisms

Previous sections have described the four mechanisms and shown how they can be used to implement applications modelled using the Terry-Wiseman approach. Clearly for this to be of practical use it must be possible to implement the mechanisms. The following four sections discuss in turn how each could be implemented using conventional architectures, capability computers and high level languages.

Conventional architectures are those which control access to memory segments using either a two-state memory mapping unit, with supervisor calls switching from user to kernel state, or a ring protection scheme [Frosini&Lazzerini85], with gates allowing transfer to inner rings.

Capability computers are those where access to an object is permitted if and only if the right capability is possessed. A capability is a protected value comprising the address of an object and some information about the kinds of access which are permitted using that capability. They were first described by [Dennis&vanHorn66], but as indicated by

[Fabry74] they need to be first class data values for maximum benefit. Various capability computers have been built each providing capabilities in different ways [Wiseman82].

High level languages control access to data using strong typing [Cardelli&Wegner85]. Usually the inner workings of a programming language, eg. the stack and heap, can be accessed by writing programs in another language, in particular an assembler language. However, if an entire application is written in one language this amounts to using an abstract machine which provides strong typing. Two such abstract machines are considered, Ada and Ten15 [Foster89]. This approach is not new, and has been used successfully in some Burroughs systems for a number of years.

Implementing Unforgeable, Opaque Addressing

Conventional Architectures

In a conventional architecture, access to volatile memory is controlled by a memory protection unit which is used to provide a number of virtual address spaces. The unit may perform a translation from virtual to physical address and addresses may refer to memory segments residing in primary or secondary memory.

If no mapping of addresses is performed and memory resources are allocated dynamically, allocating a segment in one address space has an effect on others. This is because the addresses can always be examined, leading to the problems described earlier. Thus simple memory protection does not offer unforgeable, opaque addressing and cannot be used to implement dynamic high assurance secure systems.

In systems that map virtual to physical addresses, the virtual addresses hide the existence of the physical addresses. The memory management sub-system is responsible for choosing the physical addresses of new segments, even though the virtual address may be chosen by the caller. It is therefore not possible for the caller to use physical addresses directly. Thus the physical addresses are opaque and, because access to the mapping tables is limited, they are unforgeable.

If it were possible to make an object appear in another user's virtual address space, this could be detected, leading to a covert channel. However, this problem is not due to the sharing of physical addresses, because they still remain hidden. It arises because in this case the virtual addresses are themselves being shared to a limited extent.

Therefore, with careful use, it is possible to provide unforgeable, opaque addresses for volatile memory objects using conventional memory mapping techniques. However, the physical segment addresses are the unforgeable, opaque address, not the virtual addresses used by programs.

If the underlying memory is a paged system, where pages of virtual memory are automatically moved between main RAM and secondary disc store, the difference in access time to an object can be detected. If separate virtual address spaces contend for the same physical resources, their opaqueness is lost because pages brought into main store for one address space may displace pages belonging to another. This covert channel can be avoided by ensuring that the total working set of all programs does not exceed physical memory. A residual channel through the usage of disc bandwidth does remain, but it is unlikely to be of any consequence.

Non-volatile memory is usually organised into files which are objects whose addresses are textual names. These names form a single address space, though it is often structured hierarchically into directories. When a new object is created its name, unlike the physical addresses for volatile objects, is specified by the caller. Therefore the names are neither opaque nor unforgeable and introduce covert channels. The only way to make the use of file names secure is to operate them in disjoint address spaces. Effectively this means having separate file stores for each classification, which means that software with different clearances cannot communicate easily because of confusion over duplicate names.

A file is opened by specifying its file name, however once open it is referred to by some kind of channel number. These numbers are effectively addresses of open file objects, but since they are forgeable the address space cannot be shared securely. The same applies to other kinds of object, such as offspring processes.

Capability Architectures

In a capability computer, operations which create new objects generate and return a capability as a result. This is a data value and can be freely copied. However scalar values cannot be treated as capabilities, because this would allow unauthorised access to be obtained by guessing addresses.

However, to provide the unforgeable, opaque address property required for secure systems, it must be impossible to "examine" the capability itself, since this would introduce a covert channel. Effectively this means that software must be prevented from treating a capability as a scalar value. Such a feature is however usefully provided by many capability computers, in particular in Flex [Foster*et.al.*82] it can be used to display object addresses when debugging.

Some architectures provide weak capabilities [Lieberman&Hewitt83], which behave in the same way as the ordinary strong capabilities except that they do not protect the object they reference from garbage collection. Thus an object is recovered if it is only accessible via weak capabilities and all capabilities referring to it are replaced by nil. This is useful for aliasing different representations of the same object [Currie*et.al.*81], but it does mean that weak capabilities do not satisfy the requirement for unforgeable, opaque addresses, because they may change detectably.

High Level Languages

In high level programming languages objects are addressed either statically, with identifiers referring to variables on the stack, or dynamically using values which refer to variables in the heap. The most important abstraction presented by high level languages is that which hides details of the addressing mechanism used. This is achieved by preventing scalar values being interpreted as addresses and vice versa.

So it can be seen that high level languages potentially provide unforgeable, opaque addresses. However, Ada provides a means of discovering the physical address of a variable in order to allow hardware interfaces to be driven. This allows the abstraction to be penetrated and thus admits covert channels.

In contrast to Ada, Ten15 provides a perfect abstraction of addressing using pointers and references[1]. However pointers can exist in a weak form, so, because they may change detectably, they do not satisfy the requirement for unforgeable opaque addresses. However, Ten15 references do not have this property and so satisfy the needs of security.

[1] Pointers refer to entire variables while references can refer to individual parts of a variable.

Implementing Data Hiding

Conventional Architectures

Open files are an example of an object which can be addressed, using a channel number, but not accessed directly. Read and write access is made by invoking the operating system code, using some form of supervisor call. The code which implements the access ensures that the object remains hidden from the caller and that any necessary access control and auditing takes place.

The data hiding mechanism is therefore provided by the supervisor state of a two state machine or by inner rings of a ring protection system. The disadvantage of this method of data hiding is that the access control code of all objects resides in the same place. Giving access to the underlying representation of one object gives access to all other objects, including those of other types.

Using memory mapping units to give a finer degree of protection is very costly in terms of context switching time. However, it is sometimes done with larger objects such as files. For example a file control process may be used to manage the backing store, with access requests being made by message passing. This puts all the access control code for files in one place, but without giving access to other kinds of object. However, protecting small objects individually in this way in not practical because of the excessive overhead in the storage and switching of page tables.

Capability Architectures

When an object is accessed, the access rights of the capability used to address it limits the operations that may be carried out. However this mechanism does not provide data hiding. This is provided by entry objects [Dennis&vanHorn66] and the enter operation, which is all that can be applied to them.

An entry object consists of some data and some code which has well defined entry points. The enter operation is effectively a protected procedure call. It passes control to one of the entry points and supplies capabilities for accessing the data, along with any user parameters. Another operation allows a return to be made.

In order to control access to data, code which performs permitted operations is bound with the data it protects into an entry object. It is however necessary to ensure that the data is properly hidden. In particular after creation of an object, capabilities for its internal data structures must be found only within the entry object. Also, each interface routine must preserve this requirement. This ensures the access control checks made by the interface cannot be bypassed.

Some capability computers implement entry objects by having an enter access right, rather than a separate class of object, which indicates that the object referred to is to be treated as an entry object. When the object is entered, the code is given a copy of the capability with full access rights, allowing it to manipulate the data. Apart from this, however, it is important to note that access rights have little role in providing access control in capability systems.

It is well known that the use of a read only access right does not mean that an object cannot be modified [Boebert84]. This is because the object may be tree structured, with internal links made of capabilities having write access. It is then possible to modify the structured object by reading one of the capabilities with write access and performing some modification using that.

The only role for read only access rights in a secure system is in allowing simple objects, especially code, to be shared without copying. An object can be safely shared in this way if it is a simple storage object containing only scalar data or read only capabilities for like objects.

The use of capabilities can be extended to reference persistent objects in backing store [Currie&Foster87]. In these cases the contents of an object need to be accessible to the disc manager so that they may be copied to disc. This is achieved by the type manager providing a 'flatten' operation which provides the contents of an object in a flattened form, suitable for writing to disc. Unfortunately mutual authentication between the type manager and the disc manager is required. This is to ensure that the type manager does not divulge the contents of an object to software other than the trusted disc manager, and to ensure that the disc manager is not fooled into creating a spoof persistent object by untrusted software.

Mutual authentication can be achieved by establishing the pedigree of the flatten operation, which the disc manager first checks before handing over the operation which puts data to disc. The type manager must check the pedigree of this operation before giving it the flattened data.

High Level Languages

Modular compilation systems for high level languages generally allow for a subset of the variables declared in a module to be exported. Variables not exported are only in scope within their module and so cannot be accessed directly by other software. Procedures which are exported from the module can give access to these hidden variables, and so the requirement for data hiding can be met. This simple scheme can, however, only hide variables that are declared statically when the hiding mechanism is written.

A more flexible scheme which allows new hidden variables to be introduced as and when required is provided by abstract types. Here a module can introduce a new type, providing a concrete representation and operations for creating and manipulating objects of the type. Outside of the module only the abstract operations can be used. This allows any number of new objects to be declared or generated without any need to alter the defining module.

Data hiding can also be achieved with first class procedures [Currie82], if they are provided by the high level language. Here the procedure provides an interface to the data contained in its environment and, since several procedures can share parts of an environment, it is possible to arrange for the data to be hidden by the operations which act upon it. This method of data hiding is sometimes more convenient than abstract data types, especially when the object is transient, such as an open file.

Both Ada and Ten15 provide simple modules for hiding statically created data and abstract types for hiding data created more dynamically. Procedures in Ada are only treated as control structures, but in Ten15 they are treated as values and elevated to first class status, providing an alternative technique for data hiding.

An architecture based on strong typing could provide persistent storage without the problems found in capability computers. The compiler knows the type of values being made persistent and therefore knows how to flatten them. Thus there is no explicit 'flatten' operation which must be carefully protected. This approach is taken in Ten15, which has type constructors that describe persistent objects on backing store and remote objects on other machines.

The disadvantage of this approach is that objects will be flattened in a standard way, while it might be more appropriate to have different representations on disc and in store. If this functionality is required, a mutual authentication mechanism must still be provided.

Implementing Pedigree

Conventional Architectures

In conventional systems, processes have different name spaces for different kinds of object. For example, volatile memory would be addressed using virtual addresses, persistent files by file name and open files by channel number. In such cases the pedigree of an object is evident from which name space is used to address it. This is because the only software able to include a new object in a particular name space is that which is responsible for hiding the object behind access control checks.

A sophisticated system could use a ring protection scheme to control access to a number of different types of object, all referenced using the same name space of virtual addresses. In this case the calling software may need to distinguish one type of object from another, but unfortunately one gate looks exactly the same as another. In such cases the pedigree of an object cannot be determined from its interface.

This means that even if references to objects can be freely passed around, faithful software cannot agree to modify an object using a reference obtained dynamically. This is because the object may be of the wrong type and calling its interface may cause an inappropriate modification to occur.

The interface of all objects could be extended to include a standard means of determining their pedigree. For example the first gate of any object could be a routine that returned the type of the object. It would of course be necessary for the caller to determine that the address is that of a real gate and the system must ensure that ordinary users cannot create new gates.

Capability Architectures

The pedigree of an entry object cannot be determined by software because the only operation that can be applied to it is enter, which immediately gives control to the object. An extra mechanism is required to allow pedigree to be ascertained. Sealed objects, first described by [Redell74], provide this mechanism, but it should be noted that not all capability computers provide them.

A sealed object is created by attaching a seal to an existing object using the seal operation. The only operation which can be applied to a sealed object is unseal. This requires that a copy of the seal be presented and returns a capability for the original object as a result, but only if the seals are equal.

For each kind of object, for which pedigree needs to be determined dynamically, a unique seal must be created. This is simple to arrange in a capability computer because the capability for a new object is always guaranteed to be unique. The objects must be sealed with the appropriate seal before being distributed.

To establish the pedigree of an object it is simply unsealed using the appropriate seal. This operation will fail if the object is not a sealed object or it was made using a different seal to the one supplied. Obviously to prevent forgery the seals must be protected from disclosure. Thus the data hiding mechanism is used to hide the seal with code that performs the unseal operation.

The pedigree of the entry object which is entered to establish an object's pedigree is itself established statically.

In order that a class of objects can be made persistent with pedigree, the unique seal must itself be persistent. Furthermore, when the type manager for that class executes it must be able to obtain the seal, but it must be hidden from all other software. This can be achieved using persistent data hiding mechanisms to bind the seal to the type manager code. The seal would be generated and bound to the code at installation time, but provision must be made for maintenance of the code without revealing or changing the seal [Harrold89].

High Level Language Architectures

Dynamically establishing the pedigree of a hidden object is achieved with a high level language's abstract data type mechanism. The type of an object cannot be forged and so provides the pedigree mark necessary for security. Note that abstract data types provide pedigree in addition to providing data hiding and so are convenient to use. First class procedures, however, only provide data hiding.

Implementing Context Sensitive Addressing

Conventional Architectures

Context information is invariably attached to processes. It is observed by making supervisor calls which either return a copy or map read only access into the process' address space. This ensures the context cannot be modified, except by suitable software executing in some privileged state.

Capability Architectures

Context information is invariably attached to processes, just like in conventional systems. The minimum requirement is for the hardware to enable a process to be able to establish a unique identifier for itself. This allows a general purpose context to be constructed in software, protected by the data hiding mechanisms.

A powerful extensible context mechanism would use unique identifiers to 'name' elements of the context. New elements can be introduced simply by inventing new element identifiers. Unique identifiers are easily generated in a capability computer because capabilities for new objects are guaranteed to be unique.

The operation to add something to the current process' context would take two parameters: the element identifier, which dictates which element of the context is being added or replaced, and a capability for an arbitrary data structure which is the new value of that element. A process can find out the current value of an element in its context by calling an operation which takes an element identifier as a parameter. The result is a capability for the arbitrary data structure currently associated with the element name.

High Level Language Architectures

Context is usually provided by operating systems and not by programming languages. This is the case for Ada, where it is unfortunately not even possible to uniquely identify the current task. Ten15, however, is intended to be a complete description of an abstract machine, rather than just another programming language, and so does provide this facility. As explained earlier this is all that is necessary to build an extensible context mechanism in software.

Summary and Conclusions

The Mechanisms

Secure systems place certain demands on the protection facilities offered by computer architectures. The four basic mechanisms proposed in this paper are mechanistic, yet are still independent of the target architecture. This contrasts with an earlier attempt to assess -hardware requirements by [Landwehr&Carroll84], which concludes that support for domains is essential. These domains are at the same level of abstraction as the entities of the Terry-Wiseman security model and no framework for assessing different hardware architectures is given. The basic mechanisms proposed here provide that framework as they are at an intermediate level of abstraction between the hardware and entities or domains of the model.

Unforgeable opaque addressing provides a perfect abstraction of the addressing mechanism used to reference objects in the computer. In the abstract specification, transitions change the attributes of entity's, and for this to be implemented using dynamically created machine level objects, without introducing any covert channels, unforgeable opaque addressing is required.

Data hiding prevents data from being accessed in an arbitrary way by restricting access to code that is bound to it by its creator. This code can perform any access control checks that are required, so can be made responsible for preserving confidentiality and integrity.

Pedigree allows the authenticity of an object to be established, even when they are obtained dynamically. This is required to prevent users being fooled into applying operations which affect integrity to the wrong objects.

Context sensitive addressing is necessary to allow access control code to obtain the security critical parameters it needs. Control attributes, such as clearance, will be stored in the context of a requestor and can be freely observed, but their modification is subject to the usual integrity controls.

Conventional Architectures

A conventional architecture can only provide unforgeable, opaque addressing by mapping virtual to physical addresses, however this results in many separate address spaces. This means modern software engineering techniques, which require a uniform address space [Stanley85], cannot be utilised.

From an assurance point of view, the correct implementation of unforgeable, opaque addressing on a conventional architecture relies on the memory mapping tables being managed properly. However, these are accessible to all kernel software, so it is not a straightforward task to show this.

Data hiding can be provided by utilising the privileged state or inner rings of a conventional architecture. However this gives access to all the data stored there, thus it is effectively one object hidden behind supervisor calls or gates, and this goes against the principle of least privilege. It is possible to use separate processes for different objects or classes of object, but the lengthy context switching time makes this impractical except for collections of large objects, such as files.

So, for performance reasons, more than one entity has to be mapped to a machine level object. Thus, entities which are observed will inevitably be stored in the same machine level object as entities which are not observed and this makes the proof of secure refinement much more difficult. It is necessary to prove, for each abstract operation, that the data representing those entities which are not observed does not affect the modified entities. Hence it is difficult, though obviously not impossible, to achieve high assurance systems on conventional architectures.

Conventional architectures generally have separate address spaces for different classes of object. Thus an object's pedigree is established statically by virtue of which address space is used, and so a dynamic mechanism is normally unnecessary.

The problem with a context mechanism provided by an operating system kernel is that it is not user extensible. Incorporating some application specific information to the context involves modifying the kernel's code. For example, adding an electronic mail package to an existing system may require the addition of "mailbox name" to the process context, in order that the send function can guarantee this is placed in the 'from' field correctly. Having to modify the kernel, however, will require its assurance to be assessed again.

Capability Architectures

Capabilities are essentially unforgeable, opaque addresses which do provide a uniform address space. However, some implementations offer facilities, such as weak capabilities or revocation [Redell&Fabry74], which destroy this property.

Most capability computers have been implemented so that memory management is performed by kernel software, however capabilities allow 'least privilege' to be applied so that the kernel is not monolithic, making the assurance task easier. Memory management can, however, be implemented in microcode [Currie*et.al.*81] or even hardware [Wiseman89], which is the extreme of 'least privilege'.

In capability computers it is the entry objects that provide a means for data hiding, not capability access rights. They allow small amounts of data to be hidden and so can be efficiently used to individually protect each entity, satisfying the principle of least privilege.

Entry objects were included in the very first definition of capabilities [Dennis&vanHorn66], unfortunately they have been ignored by many. In particular the definition in [Boebert84] explicitly excludes entry objects. This has lead to several proposals, for example [Karger&Herbert84] and [Gong89], which are complex yet are not as general purpose as the original concept.

An object's pedigree can be established dynamically if the capability computer supports sealed objects. These allow an object's creator to mark it in a way which uniquely identifies the creator and cannot be forged by anyone else.

The context mechanism in capability computers is very similar to that in conventional computers, but it is generally extensible, allowing application specific elements to be added to the context without requiring modification of any other software.

Many proposals have been made for using capability computers in secure systems, but none have as yet come to fruition. However, the main problem is not technological, but is "the inertia which makes it easier to continue doing things as as they have been done in the past" [Linden76].

High Level Languages

High level languages should in theory provide unforgeable, opaque addresses as part of their addressing abstraction, however this is often compromised for practical reasons, as has happened in Ada. Ten15 is an example of a language which does not need to compromise the abstraction, because it includes a view of the operating system world in its type system.

Simple data hiding comes from a modular compilation system, though dynamically created objects require the sophistication of first class procedures or abstract data types to hide their contents. Abstract data types also provide the means for establishing an object's pedigree.

Programming languages do not generally supply a context mechanism as standard, because they usually rely on services provided by underlying operating systems. Ten15, however, offers a context mechanism as standard without operating system support.

Conclusions

This paper has shown how the architectural requirements for secure systems can be expressed as four basic mechanisms and that these can be provided in a number of ways. Conventional two-state machines and ring protection architectures can be used, but gaining high assurance is quite difficult because they are too inefficient when mapped directly to the security model. Capability architectures offer fine grain protection and so can directly implement the abstraction given by the security model. This greatly simplifies the assurance task, as well as offering a better software engineering environment. Even more precise protection is available in the abstract machine provided by a high level language. However assurance is lower here because of the relative complexity of the compiler which is relied upon to maintain the strong typing that underpins security.

The best means of building high assurance secure systems appears to be to use a high level language implemented on a simple capability machine. Here compile time type checking is complemented by orthogonal run time checks. This reduces the trust in the compiler to acceptable levels, and makes the applications portable by hiding the details of the underlying capability machine. An important advantage of this approach is that the system will execute without modification on conventional hardware, though with lower assurance.

References

W.E.Boebert, "On the Inability of an Unmodified Capability Machine to Enforce the *-Property", 7th DoD/NBS Computer Security Conference, September 1984, pp291-293.

L.Cardelli & P.Wegner, "On Understanding Types, Data Abstraction and Polymorphism", Computing Surveys, Vol 17, Num 4, December 1985, pp471-522.

D.D.Clark & D.R.Wilson, "A Comparison of Commercial and Military Security Policies", IEEE Symposium on Security and Privacy, April 1987, Oakland, CA., pp184-194.

I.F.Currie, "In Praise of Procedures", RSRE Memo 3499, July 1982.

I.F.Currie & J.M.Foster, "The Varieties of Capabilities in Flex", RSRE Memo 4042, April 1987.

I.F.Currie, P.W.Edwards & J.M.Foster, "Flex Firmware", RSRE Report 81009, September 1981.

J.B.Dennis & E.C.van Horn, "Programming Semantics for Multiprogrammed Computations", Communications of the ACM, Vol 9, March 1966, pp143-155.

R.S.Fabry, "Capability Based Addressing", Communications of the ACM, Vol 19, July 1974, pp403-412.

J.M.Foster, "The Algebraic Specification of a Target Machine: Ten15", in "High Integrity Software", C.T.Sennett (Ed.), Pitman Press 1989.

J.M.Foster, I.F.Currie & P.W.Edwards, "Flex: A Working Computer Based on Procedure Values", Workshop on High Level Language Computer Architecture, Fort Lauderdale, Florida, December 1982.

G.Frosini & B.Lazzerini, "Ring Protection Mechanisms: General Properties and Significant Implementations", IEE Proceedings, Vol 132, Part E, Num 4, July 1985, pp203-210.

L.Gong, "A Secure Identity-Based Capability System", IEEE Symposium on Security and Privacy, May 1989, Oakland, CA., pp56-63.

C.L.Harrold, "A Security Policy Model and its Use", to appear 1990.

C.L.Harrold, "Secure System Initialization in SMITE", RSRE internal report 1989.

M.A.Harrison & W.L.Ruzzo, "Protection in Operating Systems", Communications of the ACM, Vol 19, Num 8, August 1976, pp461-471.

C.E.Landwehr & J.M.Carroll, "Hardware Requirements for Secure Computer Systems: A Framework", IEEE Symp. on Security and Privacy, April 1984, pp34-40.

H.Lieberman & C.Hewitt, "A Real Time Garbage Collector Based on the Lifetimes of Objects", Communications of the ACM, Vol 26, Num 6, June 1983, pp419-429.

T.A.Linden, "Operating System Structures to Support Security and Reliable Software", NBS Technical Note 919, August 1976.

C.Morgan, K.Robinson & P.Gardiner, "On the Refinement Calculus", Oxford University Programming Research Group, Technical Monograph PRG-70, October 1988.

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

D.D.Redell & R.S.Fabry, "Selective Revocation of Capabilities", Workshop on Protection in Operating Systems, Rocquencourt, France, August 1974, pp197-210.

C.T.Sennett, private communication, October 1989.

M.Stanley, "The Use of Values Without Names in a Programming Support Environment", RSRE Memo 3901, November 1985.

P.Terry & S.Wiseman, "A 'New' Security Policy Model", IEEE Symposium on Security and Privacy, Oakland, CA., May 1989, pp215-228.

S.R.Wiseman, "Two Advanced Computer Architectures: A Study of their Support for Languages and Operating Systems", RSRE Report 82013, July 1982.

S.R.Wiseman, "A Capability RISC Processor" RSRE internal report 1989.

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 Report 89024	3. Agency Reference	4. Report Security Classification Unclassified

5. Originator's Code (if known) 7784000	6. Originator (Corporate Author) Name and Location ROYAL SIGNALS & RADAR ESTABLISHMENT ST ANDREWS ROAD, GREAT MALVERN WORCS WR14 3PS
5a. Sponsoring Agency's Code (if known)	6a. Sponsoring Agency (Contract Authority) Name and Location

7. Title

BASIC MECHANISMS FOR COMPUTER SECURITY

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 WISEMAN S	9a. Author 2	9b. Authors 3, 4 ...	10. Date 1990.01	pp. ref. 18

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

15. Distribution Statement

Unlimited

Descriptors (or Keywords)

Continue on separate piece of paper

Abstract

The protections facilities required for computer security are expressed as four basic mechanisms. It is shown how a security model maps to these mechanisms and how they can be implemented on conventional architectures, capability architectures and in high level languages. [illegible]