1.0 INTRODUCTION

The notion of the "capability" as an object name and as a ticket for the authorization of access to objects has been known for more than a decade. Several capability-based systems have been designed, but never built. Still, several other commercially-available or experimental operating systems have been, and continue to be, built based on different implementations of capabilities. The many and varied implementations, each appearing to have perceptible differences in their properties of protection, have resulted in confusion, or lack of understanding, as to the role of capabilities in supporting different security policies.

This paper uses a model referred to as a "traditional" capability-based system, to clarify the role of capabilities in supporting different security policies. In particular, these "traditional" systems are analyzed for their ability to meet [TCSEC83]. The "traditional" systems model used is derived from common properties encompassed in these capability-based systems; PSOS, BURROUGHS B5500, CAL-TSS, HYDRA, PLESSEY System 250, IBM System 38, INTEL iAPX 432 and CAP.

It is intended through this analysis of the "traditional" systems model that NCSC product evaluation team members and designers of operating systems will see why and where extensions to, or departures from, these "traditional" properties are needed or why and where extensions have been used in more recent capability-based products. Further, this paper also provides a comparison of traditional capability-based systems to descriptor-based systems so that a better understanding of the fundamental and technological advantages and disadvantages of capabilities can be developed in practice.

This paper does not cover in any detail three classes of capability-based systems. First, distributed systems and the extensions they require, are not thoroughly analyzed. Second, newer, non-traditional capability-based systems that have been implemented such as NL TSS and KEYKOS are not included; future evaluations will undoubtedly clarify the relative advantages of these systems. Third, systems whose implementations are not complete, e.g., SAT at Honeywell and FLEX at RSRE in the UK, are not discussed unless extensive documentation exists in the public domain, e.g., PSOS.

This paper is organized in the following manner. First, a review of the key security-related properties of traditional capability-based systems is presented. Then, they are viewed in terms of the NCSC TCSEC. Next, a comparison with descriptor-based operating systems is included to emphasize the relative merits of capability-based systems. Finally, conclusions are presented. A bibliography has been included to guide the reader to additional reference material discussing various aspects in more detail.
2.0 Properties of Traditional Capability-Based Systems

In order to analyze protection issues in operating systems, the subjects and objects in the system must be identified. Subjects are usually the active entities in the system. Objects are usually the passive entities that are created, used, and deleted by subjects. The typical example of a subject is a user or a process executing instructions in some domain on behalf of a user or group of users. Typical examples of an object are files and memory segments. Note that, in some cases, subjects are acted upon by other subjects. Thus, subjects can also be viewed as objects [Graham72, Lampson71].

In non-capability systems, access to an object can be restricted based on the identity of the subject. For example, access in UNIX [Grampp84] is dependent on whether the subject is the owner of the object, a member of the owner's group, or someone else. Systems using access control lists, such as MULTICS [Saltzer74], also provide this type of protection model. Capability systems, however, provide an entirely different model of protection.

In capability systems, access to an object is restricted based on the subject having possession of a particular string of bits, called a capability. When an object is created in these systems, the operating system kernel returns to the subject a capability or, in some cases, an index for a capability, that uniquely identifies the object just created. When the subject wants to access the object at a later time, merely possessing the capability (with the privileges enabled as explained later) is sufficient proof that access should be granted. Note that, if a capability is ever stolen or given away, this protection mechanism can result in other subjects accessing the object without incurring a protection violation. Thus, special care must be taken to prevent unintended access, and, special (encryption-based) mechanisms must be provided to prevent the use of capabilities that are stolen while migrating on off-line storage or among different sites of a network [Nessett82, Donnelly80].

A capability contains three important fields: an object name field, a type field, and a privilege field. The discussion of capabilities in this paper will be based on these three fields. Each field is described below. Note that only the object name and access privileges are fundamentally necessary for the definition of a capability; practical considerations require that the type field be included.

The object name field is used to locate the object that the capability is referencing in a system-wide data structure. No two objects can have the same name, and no name should be reused. More information on this name field will be discussed in Section 2.1.

The type field indicates what kind of abstraction the capability references. The kernel and parts of the operating system above the kernel usually implement several basic object types (e.g., files, directories, pipes) that user domains (or subsystems) use to build other abstractions. Each type abstraction is associated with a domain that functions as the manager for that object type. The type manager, which is discussed in Section 2.2.4, implements the operations (e.g., create, read, write, destroy) on objects of that type.

The practical need for having the object type in capabilities (as opposed to having it in the object representation) is apparent when one considers the semantics of access privileges. Since object privilege semantics are defined relative to the type of object, then the interpretation of the access privileges of a capability needs type information. Should the type field be represented in the object representation, all access checks would require object access.
The privilege field contains a one-to-one mapping of bits to allowed access to the object. Normally, there is a bit for read access, a bit for write access, and bits for object dependent operations (e.g., execute for process objects). In general, the semantics of the access privilege depend on the object type. This implies that the interpretation of the access privilege sets depends on the contents of this type field. For example, bit i of the privilege field may identify a READ operation on a segment or a "search" operation on a directory. Through the capability protection mechanism, the possessor of a capability can always delete privileges without incurring a protection violation, but cannot add them. Details of the capability protection mechanism will be discussed in Section 2.2.

These three fields together define a capability. Systems that use capability-based object addressing and protection at the processor or kernel interface are called capability-based systems. The important security-related properties of capability-based systems can be divided into two major areas: (1) capability-based addressing and (2) capability-based protection. These two areas are examined in Sections 2.1 and 2.2 respectively, and a set of properties which are common to many capability-based systems is presented. These common properties define a "traditional", or typical, capability-based system. Section 2.3 then discusses the general security and integrity policies that traditional capability systems can and cannot support. Although there exist capability based systems that do not fit the traditional system model, identification of such a model and the policies that it supports is necessary to enable the study of the impact of the NCSC criteria (Section 3.0).

2.1 Capability-Based Addressing

In this section, the mechanisms and techniques used in capability-based addressing are discussed. In Section 2.1.1, the mechanisms used to address operands of instructions are examined. In Section 2.1.2, the technique used to address programs for transfer of control is discussed. Other mechanisms, such as those used in domain invocation that also rely on capability-based addressing, are discussed in the context of domain representation and protection in Section 2.2.3.

2.1.1 Operand Addressing

There are three steps involved in addressing an operand (an element of an object) in a capability-based system. First, a capability for the object (or segment) is identified using the address field of the instruction. Second, the unique object name of the capability is mapped to the memory address of the object. Third, the operand is obtained from a specified offset within this newly identified object. The first and third steps are implemented by the addressing mechanism of the system. Several addressing techniques are discussed in Section 2.1.1.1. The second step is implemented by a mapping mechanism, which is discussed in Section 2.1.1.2.

2.1.1.1 Addressing Mechanisms

The addressing mechanism of any system defines the correspondence between an address field of an instruction and an operand in memory or in processor registers. In capability systems, the addressing mechanism is similar to that of any segmented system, and consists of two steps. First, starting with an instruction address, the mechanism identifies the capability for the object (or segment) containing the operand. Then, it uses the identified capability with a specified offset to locate the operand within the object.

Capability-based addressing mechanisms can be implemented either with or without processor support. In systems that provide processor support (e.g., CAP, Plessey) the address field of an instruction includes: (1) an identifier for the processor register
containing the capability used for addressing an object, and (2) an offset used for addressing the operand within that object. Two examples of this type of instruction are shown in Figures 1a and 1b and are described below. Figure 1a shows a partitioned memory implementation while Figure 1b shows a tagged memory implementation. (These implementations are discussed further in Section 2.2.)

Example 1a

In a partitioned memory system, memory segments and registers are divided into those that contain data and those that contain capabilities. Similarly, the instructions are divided into those that operate on data and those that operate on capabilities. As shown in Figure 1a, two instructions may be required to load an operand from memory. The first instruction is a capability-load instruction and is used only once for all references to the segment named by the loaded capability. The capability source register and an index register specify the capability segment (C-List) and offset (j) to the needed capability. This capability is loaded into the capability destination register. The second instruction is a data instruction, and it uses the capability that was stored in the capability destination register to address the data segment (or object), which contains the operand. The offset into the data segment is specified by an index register. The specified operand is then loaded into the data destination register.

Example 1b

In a tagged memory architecture, each element of memory is tagged as either data or a capability, depending on its current contents. In Figure 1b, the capability for the appropriate segment has already been loaded into the source register. This capability is used along with the offset in the index register to address the operand. The operand is then loaded into the destination register. Note that the same segment (or object) that contains the operand may also contain capabilities.

(Note that for some operands, such as the next instruction to be executed, the register identifiers are implicitly specified by each instruction and, therefore, are not represented in the address field of the instruction. For example, the program counter registers, which contain the capabilities for the program segment and the offset of the next instruction within the segment, are never explicitly identified in an instruction address. This is true in any system -- capability-based or otherwise.)

In systems that provide no special processor support for capability-based addressing (e.g., CAL-TSS, HYDRA), the capability needed for addressing is identified differently. In these systems, a process must execute a "system call" to the capability management layer of the system (e.g., to the kernel) with a parameter that identifies the location of the desired capability in the current C-list of that process. The operand offset is specified as an additional parameter of the system call. The capability identified in the system call is used to find the base address and the length of the memory segment (or object), and the offset is used to locate an operand within the segment boundaries. Thus, in capability systems that do not provide processor support, the address field of an instruction has the same form, and is interpreted in the same way, as an address field in a non-capability system.

Whenever indirect addressing is specified within an instruction or system call, the operand field identifies the beginning of a chain of capabilities. During instruction execution, each capability of the chain is loaded, either in a processor register or in a system-protected memory area, and used for further capability loading until the target operand is reached. An additional offset must be specified for each level of indirection of
Figure 1a. Address Mapping in Partitioned Memory with Processor Support

LEGEND:

- $R_i^D$ ($R_j^D$) = index (data) registers
- $R_S^C$ ($R_D^C$) = source (destination) register for capabilities
- $R_D^D$ = destination register for data
- C-list = capability list
- ACC$_1$·n = access privileges
- IND/INX = indirection/indexing fields

ADDRESS

CAPABILITY INSTRUCTION

DATA INSTRUCTION

SOURCE OFFSET DESTINATION

data registers

C-list

Data Segment
Figure 1b. Address Mapping in Tagged Memory with Processor Support
the chain, either next to each capability or within the capability mapping mechanism (discussed in Section 2.1.1.2).

Figure 2 shows three levels of indirect addressing. In this tagged memory implementation, the "target" operand, which is loaded into the destination register, could be either a capability or data. Note that indirect addressing differs from the programmed use of multiple capabilities for the traversal of multiple objects. In the former case, a single instruction is executed; in the latter case, multiple instructions are executed. Consequently, in processor-supported capability systems, the processor must ensure that the indirection chains have finite length. Otherwise, a single instruction may place the processor in an infinite address-translation cycle. (Such indirection problems are not restricted to capability systems. They exist in any system that allows arbitrary levels of indirect addressing, such as the TI-ASC [Galie75].)

The authentication mechanism of indirect addressing has two important aspects. First, each capability must authorize the reading of the next capability in the indirection chain. Second, the action required by the instruction operation code, or by the specific kernel call, must be authorized by the set of privileges identified by the chain-end capability. This is discussed in more detail in Section 2.2.2.

2.1.1.2 Capability Mapping

The capability mapping mechanism establishes the correspondence between the capability and the object it names. Traditional proposals and implementations for capabilities stress that a capability consists of an object name in addition to a type and access privileges. The unique object name is constructed, in the most general case, from a unique identifier (UID) and an index. The identifier is unique for the lifetime of the system, and is used for the selection of an object map in a global system table, called the master object table (MOT). An object map contains the description of the object representation in file storage. As shown in Figure 3, the MOT resides on the file device and has an entry for the map of every object in the file system. A MOT entry contains the object's UID, its base address on the file storage, its length, etc. In general, any capability mapping mechanism needs to ensure (1) that a capability accurately identifies an object, and (2) that no capability can be used to reference a destroyed object. Traditional capability systems may differ in specific details of the capability mapping implementation. The model described below, however, can be used to explain the mapping mechanisms of most traditional capability-based systems.

The capability index provides the first step in the model mapping mechanism. It is a hint given to the processor (or kernel) to aid it in the identification of an object in the virtual memory. This index is used as an offset in a table that has fixed length and that is resident in the main memory. Because of the fixed size of this table, its entries must be multiplexed among those of the MOT in a way similar to the entries of a cache. Two outcomes are possible in the use of the capability index as a hint. First, the hint succeeds whenever the UID matches that of the entry in the fixed length table; otherwise, the hint fails. Whenever the hint succeeds, the object map is retrieved from the table entry and is used for operand addressing. Second, whenever the hint fails (a local search of the UID among the resident table entries does not find the correct map) then the object is not in virtual memory. If the object still exists on file memory, the MOT has an object map entry for the object; this entry is mapped in the resident table using the multiplexing mechanism mentioned above. If the MOT does not include an entry with the object's identifier, the object does not exist, and a violation is signalled. The above mechanism ensures that objects can be destroyed without having to garbage collect all their outstanding capabilities. These capabilities become unusable automatically.
Figure 2. Indirect Addressing [3 levels] in Tagged Memory with Processor Support
Figure 3. A Traditional Approach to Capability Mapping