Object protection in distributed systems

Lanfranco Lopriore

Dipartimento di Ingegneria dell'Informazione, Università di Pisa,
via G. Caruso 16, 56126 Pisa, Italy. E-mail: l.lopriore@iet.unipi.it

Abstract — With reference to a distributed system consisting of nodes connected by a local area network, we consider a salient aspect of the protection problem, the representation of access permissions and protection domains. We present a model of a protection system supporting typed objects. Possession of an access permission for a given object is certified by possession of an object pointer including the specification of a set of access rights. We associate an encryption key with each object and a password with each domain. Object pointers are stored in memory in a ciphertext form obtained by using the object key and including the value of the domain password. Each process is executed in a domain and can take advantage of a given object pointer only if this object pointer was encrypted by including the password of this domain. A set of protection primitives makes it possible to use object pointers for object reference and to control the movements of the objects across the network. The resulting protection environment is evaluated from a number of salient viewpoints including ease of access right distribution and revocation, interprocess interaction and cooperation, protection against fraudulent actions of access right manipulation and stealing, storage overhead and network traffic.

Index Terms — access right; distributed system; domain; object; protection; symmetric-key cryptography.

1. INTRODUCTION

We shall refer to a distributed architecture consisting of *nodes* connected by a local area network. We make no hypothesis concerning the network topology. Collectively, the network nodes support a common pool of *typed objects*, which are the elementary unit of information movement across the network. Beside a processor, each node features memory resources for object storage.

In an environment of this type, we shall present a model of a protection system that assigns a *protection domain* to every given process; this is a collection of access permissions for the existing objects [15], [16]. The salient aspects of the protection problem are related to the representation in memory of access permissions and protection domains. The protection system defines mechanisms that allow a process being executed in a given domain to certify the access permissions it holds within the context of that domain. Furthermore, the process is prevented from manipulating the composition of the domain, to add new access permissions, for instance. Forms of interprocess cooperation are supported, so that a process may grant an access permission in its own domain to a different process.

In our system, the representation of access permissions and protection domains is based on the application of a form of symmetric-key cryptography [7], [23]. Possession of an access permission for a given object is certified by possession of an *object pointer* (*o-pointer* from now on, for short) referencing that object and including the specification of a set of *access rights*. We associate an encryption key with each object and a password with each domain. O-pointers are maintained in memory in a ciphertext form obtained by using the object key and including the value of the domain password. Each process is executed in a domain and can take advantage of a given o-pointer only if this o-pointer was encrypted by including the password of this domain. A set of *protection primitives* forms the process interface of the protection system. These primitives make it possible to use o-pointers for object reference and to control the movements of the objects in the network. In particular, a process being executed in a given node and holding access permissions on a given object may cause migration of the object to that node. The protection system makes it possible to determine the network position of every given object, independently of the previous migrations of this object.

The rest of the paper is organized as follows. Section 2 introduces our protection model with special reference to the definition of domains and o-pointers, and to the ciphertext form of o-pointers in memory. Section 3 presents a conceptual scheme for a fully-distributed implementation of our object protection strategies. The actions involved in the execution of each protection primitive are described with special reference to interactions between the network nodes. Section 4 outlines the relation of our work with previous work, and discusses the proposed protection environment from a number of salient viewpoints including ease of access right distribution and revocation, interprocess interaction and cooperation, protection against fraudulent actions of access right manipulation and stealing, storage overhead, cryptographic costs and costs in terms of network traffic. Section 5 gives concluding remarks.

2. THE PROTECTION MODEL

Our local area network consists of up to 2^t nodes that collectively give physical support to up to 2^v objects. The v-bit identifier B of a given object consists of two components, i.e. $B = \langle B_{\text{node}}, B_{\text{local}} \rangle$. Quantity B_{node} is codified in the t most significant bits of B, and is equal to the name of the node where the object was created. This node is called the *principal* of B, and is denoted by PR(B). Thus, $B_{\text{node}} = \text{PR}(B)$. Quantity B_{local}, codified in the $(v - t)$ least significant bits of B, is called the *local object identifier*. A simple strategy for the generation of local object identifiers is a sequential allocation. This strategy hypothesizes that object identifiers are so wide (e.g. 64 bits) that identifier reuse is never necessary. In a situation of this type, we shall use a *local object counter* in each node, containing the local identifier of the object to be allocated next in that node. This counter is initialized to 0 when the system is generated, and is incremented by 1 after creation of a new object in that node.

The *repository* RP(B) of object B is the node reserving physical storage resources for B. This is the only node where the contents of object B can be accessed and operations can be executed on that object. Initially, when object B is created in a given node N, it is allocated in the physical memory of that node. Thus, node N assumes both the functionalities of the principal and the repository of the new object, i.e. $B_{\text{node}} = \text{PR}(B) = \text{RP}(B) = N$. The repository may well change later, as a consequence of movements of the object across the network, whereas the principal never changes. The principal keeps track of the name of the present repository of B. It follows that it is always possible to identify the physical position of B in the distributed storage, as this information is maintained in the principal PR(B), and the name of the principal is part of the object identifier.

2.1. Domains, and object pointers

A domain identifier D consists of two components, i.e. $D = <D_{\text{node}}, D_{\text{local}}>$. Quantity D_{node} is codified in the t most significant bits of D and is equal to the name of the node where the domain has been created. Quantity D_{local}, codified in the (v - t) least significant bits, is called the *local domain identifier*. Each node maintains a *local domain counter* containing the local identifier of the domain to be created next in that node. This counter is initialized to 0 when the system is generated, and is incremented by 1 after creation of a new domain in that node.

When the operating system kernel generates a new process (i.e. the process has no parent), a new domain D is created. A *domain password* w_D is generated and is associated with that domain; this password is written into the descriptor of the new process. When a process generates a new child process, the child process is assigned the same domain as the parent process, and the password of this domain is written into the descriptor of the child process. Thus, the tree structure originated by subsequent actions of child process generation is entirely confined within the boundaries of the same domain. We say that all these processes are *tightly connected*, i.e. they share the same domain (and consequently, the same domain password). At any given time, in node N, a register of the protection system, the *domain password register* DPR$_N$, contains the password of the *current domain*, i.e. the domain of the process being executed at that time by the processor of that node. When a process switch takes place, the current process relinquishes the processor and a new process is assigned the processor, the password of the domain of the new process is copied from the descriptor of the new process into DPR$_N$.

Let B be an object, let T be the type of B, and let R_0, R_1,... be the operation defined by T. Execution of operation R_m is made possible by possession of an access privilege for B, which is expressed in term of a subset of the set AR_0, AR_1,... of the access rights defined by T. As will be clarified shortly, a few operations, and the corresponding access rights, are part of the definition of every object type. These operations are called the *protection primitives*, and form the application program interface of the protection system.

Possession of an access privilege for object B is certified by possession of an o-pointer referencing B. This is a pair $<B, AR>$, where AR is a bit configuration that specifies a collection of access rights for B: if the i-th bit of AR is asserted, then the o-pointer grants access right AR_i on B. An o-pointer is always part of a domain and can be profitably used only by the processes in that domain. This means that a process Q holding o-pointer $P = <B, AR>$ which is part of domain D can access object B and perform the actions permitted by the access rights in AR only if Q is part of the same domain, D. This essential requirement is enforced by taking advantage of cryptography, as is illustrated below.

O-pointers are never stored in memory in plaintext. Instead, the protection system associates each given object B with an encryption key k_B called the *object key*. Let $P = <B, AR>$ be an o-pointer in domain *D*. *P* is stored in memory in the ciphertext form that results from application of a transformation involving both the key k_B of object *B* and the password w_D of domain *D*. From now on, we shall use an underline to denote a ciphertext. Figure 1 shows the transformation of *P* into ciphertext $\underline{P}$. Let $\underline{AR}$ be the result of encrypting pair <*AR*, w_D> by using a symmetric key cipher with key k_B. Quantity $\underline{P}$ is given by relation $\underline{P}$ = <*B*, $\underline{AR}$>.

Figure 2 shows the reverse transformation of ciphertext quantity $\underline{P}$ = <*B*, $\underline{AR}$> into the corresponding plaintext o-pointer *P*. Object key k_B associated with object *B* is used to convert quantity $\underline{AR}$ into plaintext. Let <*AR*, *w**> be the result of this conversion. Quantity *w** is compared with domain password w_D to validate *AR*; if a match is found, validation is successful and o-pointer *P* is given by pair <*B*, *AR*>.

Of course, after translation into plaintext, o-pointer *P* is a sensitive information item that must be stored in a protected memory region. To this aim, the protection system reserves a *pointer table* for each given process: each entry of this table is aimed at containing an o-pointer in plaintext. Let PT_i denote the *i*-th entry of the pointer table PT of a given process, and let <*B*, *AR*> be the o-pointer it contains. We say that PT_i *references* object *B* with access rights *AR*. As will be made clear shortly, the pointer table is mainly aimed at object access; at any given time, the entries of the pointer table of the running process contain o-pointers for the objects being used by that process at that time.

3. THE PROTECTION SYSTEM

3.1. Protection tables

A conceptual scheme for the implementation of the object protection strategies outlined so far takes advantage of three tables in each given node *N*. These tables collectively implement a form of distribution of the protection information, as follows (Figure 3):

- the *principal table* PRT_N of node *N* features one entry for each object *B* for which B_{node} = PR(*B*) = *N* (that is, node *N*, as identified by B_{node}, is the principal of *B*). The entry reserved for *B* contains the local object identifier B_{local} and the name RP(*B*) of the repository of *B*.
- the *repository table* RPT_N features one entry for each object *B* for which RP(*B*) = *N* (that is, node *N* is the repository of *B*). The entry reserved for *B* contains the object key k_B. When an object is created in node *N* or is moved across the network to node *N*, a new entry is reserved for this object in RPT_N, and the object key is inserted into this entry. When an object is deallocated from *N* (either as a consequence of a movement of the object to a different node, or when the object is finally deleted) the corresponding entry of RPT_N is made free.
- the *domain table* DMT_N features one entry for each domain *D* for which D_{node} = *N* (that is, the domain has been created in node *N*). The entry reserved for *D* contains the local domain identifier D_{local} and the password w_D of that domain.

3.2. Protection primitives

The process interface of the protection system consists of a number of primitives, the *protection primitives*. Table I summarises the effects of the execution of each of these primitives and specifies the corresponding access right. Execution of each primitive is completely accomplished within the boundaries of the node where the primitive has been issued, or, for a few primitives, it causes interactions with the other network nodes (e.g. if the primitive involves object *B*, interactions with the object principal PR(*B*) and the object repository RP(*B*)). Interactions take the form of *control messages* and *object messages*. A control message can be a *request message* specifying actions to be carried out by the recipient node, or a *reply message* containing the results of the actions performed as a consequence of receipt of a request message. An object message contains the value of an object, and is sent when an object is moved or copied to a different node.

In the rest of this section we shall describe the activities involved in the execution of each protection primitive in some detail, with special reference to interactions between the network nodes. We shall suppose that the given protection primitive has been issued in node *N*. We make no hypothesis on the hardware configuration of each node. Instead, our system is designed to be implemented at software level within the framework of a conventional processor architecture, the only requirement being the support of two usual control modes, a privileged mode and a user mode with memory access limitations. The protection primitives will be executed in the privileged mode, and the protection tables will be stored in reserved memory regions of the protection system.

loadPtr(*addr*, *i*) is a first example of a protection primitive. Execution of this primitive in node *N* converts the ciphertext quantity <*B*, $\underline{AR}$> contained in memory location *addr* of this node into plaintext o-pointer <*B*, *AR*>, and loads this plaintext into the *i*-th entry PT_i of the pointer table of the process issuing the primitive. In detail, execution is as follows:

1. A search is made in repository table RPT_N to find the entry reserved for object *B* (if this search fails, node *N* is not the repository of *B*; an addressing exception is raised and *loadPtr*() fails). Object key k_B is extracted from this entry and is used to convert quantity $\underline{AR}$ into plaintext. Let pair <*AR*, *w**> be the result of this conversion.
2. Password w_D of the domain of the running process is read from the domain password register DPR_N and is compared with quantity *w**. If $w_D = w^*$, then validation of *AR* is successful and pair <*B*, *AR*> is loaded into PT_i; otherwise, an exception of violated protection is raised and *loadPtr*() fails.

Let <*B*, *AR*> be the plaintext o-pointer contained in the *i*-th entry PT_i of the pointer table of the running process, let *mask* be a bit configuration of the same size as the *AR* field, and let *AR** denote the result of the logical AND of quantities *AR* and *mask*. Execution in node *N* of protection primitive *storePtr*(*i*, *mask*, *addr*) converts plaintext o-pointer <*B*, *AR**> into a ciphertext, and stores this ciphertext into memory location *addr* of *N*. In detail, execution produces the actions that follow:

1. Password w_D of the domain of the running process is read from the domain password register DPR_N.
2. A search is made in repository table RPT_N to find the entry reserved for object B (if this search fails, node N is not the repository of B; an addressing exception is raised and *storePtr*() fails). Object key k_B is extracted from this entry and is used convert pair $<AR^*, w_D>$ into ciphertext quantity $\underline{AR}$. Finally, quantity $<B, \underline{AR}>$ is assembled and is stored into memory location *addr* of node N.

We wish to point out that execution of both the *loadPtr*() and the *storePtr*() protection primitives is completely accomplished within the boundaries of the node N where these primitives are issued; execution generates no message exchange with the other network nodes. This important result has been obtained by taking advantage of the distribution of the object keys across the network, as is supported by the repository tables.

Let T be the type of object B, and let R_0, R_1,... be the operations defined by T. Furthermore, let $P = <B, AR>$ be a plaintext o-pointer, and suppose that P is contained in the i-th entry PT_i of the pointer table of the running process. Execution of operation R_m on B is made possible by the *operation*(i, m) protection primitive. Execution of this primitive transfers control to the code of R_m; the contents of the AR field of PT_i is transmitted to R_m as an input parameter. The actions involved in the execution of R_m will include the access right check necessary to verify whether AR includes the access rights permitting successful execution of R_m; if this check fails, an exception of violated protection is raised. Thus, the set of the access rights for a given object is defined by the type of that object, according to the set of the type operations. As is specified in Table I, a few access rights are part of all object types, and are required to execute the protection primitives successfully. These include the OWN, COPY and MOVE access rights, which are necessary to delete an object, move the object to a different node or creating an object copy.

Object allocation and deletion

Protection primitive *newObject*(T, i) allocates a new object B of type T in node N where the primitive is issued. This means that node N assumes both the functionalities of the principal and the repository of the new object, and quantity N will be codified in the B_{node} component of the identifier B of the new object, i.e. $B_{\mathrm{node}} = \mathrm{PR}(B) = \mathrm{RP}(B) = N$. Execution deposits an o-pointer with full access rights into the i-th entry PT_i of the pointer table of the process issuing the primitive. Execution is completely accomplished within the boundaries of node N, and is as follows:

1. A storage area is reserved for the new object in node N. The specification of the size of this area is part of the definition of type T.
2. The local identifier B_{local} of the new object is generated, and identifier B is assembled by using relation $B = <B_{\mathrm{node}}, B_{\mathrm{local}}>$ where $B_{\mathrm{node}} = N$. An entry is reserved for B in the principal table PRT_N. This entry is filled with quantity B_{local} and node name N to indicate that N is the repository of the new object.

3. A new object key k_B is generated and an entry is reserved for B in the repository table RPT$_N$. This entry is filled with quantities B and k_B to associate k_B with B.
4. A plaintext o-pointer is assembled that references object B and specifies full access rights. This o-pointer is stored into PT$_i$.

Of course, after allocation, the new object will need to be initialized. This result will be obtained by taking advantage of the *operation*() primitive to execute the initialization operation, as is defined by the object type.

Let <*B*, *AR*> be the plaintext o-pointer contained in the *i*-th entry PT$_i$ of the pointer table of the running process. Protection primitive *deleteObject*(*i*) destroys object *B*. This primitive can be successfully accomplished in node *N* only if *B* is stored in memory in this node, i.e. RP(*B*) = *N*. Execution of this instruction requires access right OWN in *AR*. The actions produced by execution of *deleteObject*() can be easily imagined, and will not be described in detail.

Object migration

Let *P* = <*B*, *AR*> be the o-pointer contained in the *i*-th entry PT$_i$ of the pointer table of the running process. If executed in node *N*, the *moveObject*(*i*) protection primitive causes the migration of object *B* referenced by PT$_i$ from its present position in the network to node *N*. On termination, RP(*B*) = *N*, that is, node *N* becomes the repository of object *B*. Execution produces the actions that follow:

1. Node *N* inspects the o-pointer *P* = <*B*, *AR*> contained in PT$_i$ to ascertain whether the access right field *AR* of this o-pointer specifies access right MOVE (if this is not the case, an exception of violated protection is raised and *moveObject*() fails).
2. Node *N* extracts quantity B_{node} from object name *B* to identify the principal PR(*B*) of *B*. Then, a request message is sent to PR(*B*). On receipt of this message, PR(*B*) performs a search in its own principal table PRT$_{\text{PR}(B)}$ for an entry reserved for *B* (if this search fails, *B* has been deleted; a negative reply message is returned to *N* and *moveObject*() fails). The name RP(*B*) of the repository of *B* is extracted from this entry and is returned to *N* in a reply message.
3. Node *N* sends a request message to RP(*B*). On receipt of this message, RP(*B*) accesses the entry reserved for *B* in its own repository table RPT$_{\text{RP}(B)}$ and extracts the key k_B of object *B* from this entry. A reply message is assembled containing key k_B and the value of object *B*; this message is returned to *N*. The entry reserved for *B* is deleted from RPT$_{\text{RP}(B)}$ and the storage area reserved for *B* is made free.
4. On receipt of the reply message from RP(*B*), node *N* reserves a free storage area for object *B* and copies the value of this object from the reply message into this area. Then, an entry is reserved for *B* in the repository table RPT$_N$; key k_B is copied from the reply message into this entry.
5. Node *N* sends a request message to the principal PR(*B*) of object B. On receipt of this message, PR(*B*) accesses its own principal table PRT$_{\text{PR}(B)}$ and inserts node name *N* into the

table entry reserved for object *B*, to indicate that *N* is now the repository of *B*. Finally, PR(*B*) sends a positive reply message to *N*.

Creating an object copy

Let *i* and *j* be the indexes of two entries, PT_i and PT_j, of the pointer table of the running process, and let *P* = <*B*, *AR*> be the o-pointer contained in PT_i. If executed in node *N*, protection primitive *copyObject*(*i*, *j*) creates a new object in this node and leaves an o-pointer for this object, with full access rights, into PT_j. The new object has the value of object *B* referenced by *P*. Successful execution of this instruction is permitted by access right COPY in the *AR* field of *P*. The actions caused by execution of *copyObject*() involve message exchanges with the principal PR(*B*) and the repository RP(*B*) of object *B*. These actions can be easily imagined by considering those described previously, and involved in the execution of the *newObject*() and *moveObject*() protection primitives; we shall not discuss these actions any further.

Limiting access privileges

O-pointers are stored in memory in ciphertext as ordinary data items; consequently, they can be freely moved and duplicated. Let us consider a process Q_1 that is part of domain *D* and holds ciphertext o-pointer $\underline{P}$ = <*B*, $\underline{AR}$>. Suppose that Q_1 transmits a copy of $\underline{P}$ to a process *Q* that is part of the same domain *D*. *Q* and Q_1 share the same domain password. Consequently, *Q* will be in the position to issue the *loadPtr*() protection primitive and convert $\underline{P}$ into plaintext. So doing, *Q* will gain access to object *B*.

Let us now suppose that process Q_1 wishes to transmit only a subset of the access rights it possesses on *B* to process *Q*. To this aim, Q_1 will take advantage of the protection primitive *storePtr*(*i*, *mask*, *addr*). By using a suitable value for the *mask* argument, process Q_1 will be in the position to produce a copy of $\underline{P}$ with reduced access rights. This copy will be transmitted to *Q*.

Inter-domain o-pointer conversion

Let *Q* and Q_1 be processes belonging to two different domains *D* and D_1, and let *w* and w_1 be the passwords of these domains. Let $\underline{P}$ = <*B*, $\underline{AR}$> be a ciphertext pointer referencing an object *B*, and suppose that Q_1 transmits a copy of $\underline{P}$ to *Q*. On receipt, if *Q* issues *loadPtr*() to convert $\underline{P}$ into plaintext, *loadPtr*() will use the password of domain *D* instead of the password of domain D_1 that was originally used to encrypt $\underline{P}$. Consequently, the validation process illustrated in Figure 2 and involved in the execution of *loadPtr*() will fail.

In fact, transmission of an o-pointer between processes of different domains must be preceded by an inter-domain conversion of the o-pointer, from the domain of the granting process to the domain of the recipient process. In our example, Q_1 must preventively convert $\underline{P}$ to domain *D*. To this aim, Q_1 translates $\underline{P}$ into plaintext and loads the result *P* = <*B*, *AR*> into an entry, say the *i*-th entry PT_i, of its own pointer table. This action will be carried out by executing *loadPtr*() and will terminate successfully, as $\underline{P}$ is codified using the password w_1 of domain D_1. Then, Q_1 issues protection primitive *convertPtr*(*i*, *D*, *addr*). Execution of this primitive in node

N encrypts the plaintext o-pointer contained in PT$_i$ by using the domain password w_D of domain *D*, and stores the resulting ciphertext o-pointer into memory location *addr* of node *N*. This ciphertext o-pointer will be transmitted to *Q*. Execution of *convertPtr*() produces the actions that follow:

1. Node *N* extracts quantity D_{node} from domain identifier *D*; as seen in Subsection 2.1, this quantity is equal to the name of the node, say node *M*, where domain *D* was created. Then, node *N* sends a request message to node *M*. On receipt of this message, *M* performs a search in its own domain table DMT$_M$ for the entry reserved for *D* (if this search fails, *D* does not correspond to an existing domain, a negative reply message is returned to *N* and *convertPtr*() fails). The password w_D of domain *D* is extracted from DMT$_M$ and is returned to *N*.
2. Node *N* accesses its own repository table RPT$_N$ to find the entry reserved for object *B*. Object key k_B is extracted from this entry and is used convert pair <*AR*, w_D> into ciphertext quantity $\underline{AR}$ (see Figure 1). Finally, quantity <*B*, $\underline{AR}$> is assembled and is stored into memory location *addr* of node *N*.

4. DISCUSSION, AND RELATION TO PREVIOUS WORK

4.1. Capabilities and password capabilities

A classical approach to the representation of access privileges and protection domains in memory is based on the concept of a *capability* [14]. This is a pair <*B*, *AR*> where *B* identifies an object and *AR* codifies a set of access rights on this object. Capabilities can be freely copied. A process holding a capability for a given object and wishing to grant an access privilege included in this capability to another process will simply transfer a capability copy to the other process, possibly with reduced access rights.

Of course, we must prevent a process that holds a given capability from modifying this capability, for instance, by adding new access rights and extending the access privileges it grants. Similarly, we must prevent processes from forging new capabilities for existing objects from scratch, thereby gaining access to these objects. Several solutions to this *segregation problem* [4], [12] have been devised and actually implemented in existing systems. In a tagged memory environment, a one-bit *tag* can be associated with each memory cell specifies whether this cell contains a capability or an ordinary data item [8], [11], [21]. In an alternative approach, a specific object type, that we shall call the *capability object*, is reserved for storage of capabilities (in contrast, *data objects* will be reserved for storage of ordinary data items) [6], [13], [26]. This approach is prone to object proliferation. The internal representation of an object consisting of several data objects assumes a hierarchical structure where one or more capability objects store the capabilities for the data objects. In a distributed system, migration of a compound object of this kind across the network implies the marshalling of the compound object into linear form for transmission to the recipient node where the hierarchical structure of the object will be reconstructed.

Password capabilities are an important improvement to the original capability concept [1], [3], [10], [20]. The password-capability paradigm associates one or more passwords with each given object. Each password corresponds to a subset of the set of all the access rights defined by the type of that object. A password capability is a pair <*B*, *w*>, where *w* is a password. If a match is found between *w* and one of the passwords associated with object *B*, then the password capability grants the access rights corresponding to that password on *B*. If the password size is so large as to discourage fraudulent attempts to forge passwords, then password capabilities can be freely mixed in memory with ordinary data items and as such, they are an effective solution to the segregation problem.

In our approach, o-pointers are segregated in memory by taking advantage of cryptography [4]. We associate an encryption key with each object and a password with each protection domain. An o-pointer referencing a given object is always part of a domain, and is stored in memory in ciphertext form. The encryption scheme involves both the object key and the domain password. The identifier *B* of the referenced object is not altered by the transformation of the o-pointer between plaintext and ciphertext. In fact, knowledge of quantity *B* is only necessary to determine the object key. It follows that the position of *B* in memory is irrelevant. In particular, it is not necessary that the object name and the access right specification be stored in contiguous memory cells. A process holding two or more o-pointers referencing the same given object may well maintain a single copy of the object identifier, for instance; the process will reconstruct the association of the object identifier with the access rights just before issuing the *loadPtr*() primitive, in view of a subsequent object access.

The duality of plaintext and ciphertext o-pointers allows us to preserve the simplicity of access privilege representation and access right distribution that characterizes capability and password-capability systems, without incurring the hardware design problems and storage management complications connected with memory tagging, e.g. support by specialized memory banks. We avoid the complex software structures that ensue when capabilities are segregated into capability objects. With respect to both the original capabilities and the password capabilities, further advantages follow from encryption in terms of protection against o-pointer stealing.

4.2. O-pointer manipulation and stealing

O-pointers are stored in memory together with ordinary data item, in indistinct form. It follows that a process may well alter an o-pointer by using the machine instructions for ordinary data manipulation. An action of this type will be aimed at access privilege amplification, for instance, by modifying the access right field of the o-pointer. In fact, any such attempt to o-pointer forging is destined to fail.

Let us consider ciphertext o-pointer $\underline{P}$ = <*B*, $\underline{AR}$>, let k_B be the key of the object *B* referenced by this o-pointer, let *D* be the domain of $\underline{P}$ and let w_D be the password of this domain. The $\underline{AR}$ field is the result of a symmetric-key encryption using k_B and involving plaintext quantities *AR* and w_D (see Figure 1). We shall hypothesize that the encryption algorithm producing $\underline{AR}$

guarantees a careful mixing of the bits of AR and w_D. In this hypothesis, suppose that process Q holding $\underline{P}$ modifies $\underline{AR}$ to amplify the access permissions it contains, and let $\underline{P}^* = <B, \underline{AR}^*>$ be the ciphertext o-pointer resulting from the modification. In order to take advantage of $\underline{P}^*$ and access object B, Q will issue the *loadPtr*() protection primitive. Execution of *loadPtr*() uses key k_B to transform quantity $\underline{AR}^*$ into pair $<AR^*, w^*>$ (see Figure 2). Of course, quantity w^* will not match the password w_D of domain D that was used to codify the access right field in the original form, $\underline{AR}$. This means that the validation of AR^*, performed as part of execution of *loadPtr*(), is destined to fail, and execution of *loadPtr*() will terminate with an exception of violated protection.

Let us now suppose that process Q generates a ciphertext o-pointer $\underline{P} = <B, \underline{AR}>$ for an existing object B from scratch. In order to take advantage of $\underline{P}$ and access B, Q will issue the *loadPtr*() protection primitive. Execution of *loadPtr*() uses key k_B to transform quantity $\underline{AR}$ into pair $<AR, w^*>$. Of course, w^* will be an arbitrary quantity. The subsequent validation involves a comparison of w^* with the password w_D of the domain D of process Q. The probability of a match is determined by the size of w_D. For large passwords, e.g. 64 bits, this probability is vanishingly low. This means that *loadPtr*() will terminate with an exception of violated protection. The consequences of an exception of this type are part of the protection system design. If a delay is forced on the process generating the exception, the time cost of a force brute attack tends to become prohibitive.

A serious security hole of capability-based protection is that the access permission granted by a given capability (as specified by the access right field or, in a password capability, by the password) is independent of the capability origin. This means that a process that steals a given capability may well exercise all the access rights included in that capability. The consequences of an action of this type may well extend to a large fraction of the protected resources. For instance, consider a system that segregates capabilities into capability objects. In a system of this type, a process that steals a capability granting even read-only access permission to a capability object will be in the position of taking advantage of all the capabilities contained in that capability object, and recursively, in all the capability objects referenced by these capabilities [22].

Our encryption scheme is able to contrast any fraudulent action of o-pointer copy. This is a consequence of the fact that the validity of a given o-pointer is limited to the domain of that o-pointer. Let Q_1 and Q_2 be two processes, let D_1 and D_2 be their domains, and let w_1 and w_2 be the passwords of these domains. Suppose that process Q_2 steals o-pointer $\underline{P} = <B, \underline{AR}>$ from process Q_1. Being part of domain D_1, this o-pointer was generated by using password w_1. In order to take advantage of $\underline{P}$ and access B, Q_2 will issue the *loadPtr*() protection primitive. Execution of *loadPtr*() will use key k_B to transform quantity $\underline{AR}$ into pair $<AR, w_1>$. Validation of the access right field, as involved in the execution of *loadPtr*(), will compare w_1 with the password w_2 of Q_2, and is destined to fail.

4.3. Storage requirements

Let us consider a medium-scale network consisting of up to 2^{16} nodes, and suppose that the size of an object name is 64 bits. In a configuration of this type, the size of the B_{node} component of an object name B is 16 bits, and the size of the B_{local} component is 48 bits. Thus, the memory requirement of a principal table entry is 8 bytes (6 bytes for the B_{local} field and 2 bytes for the RP(B) field). Furthermore, in the hypothesis of 64-bit object keys, the memory requirement of a repository table entry is 16 bytes (8 bytes for the B field and 8 bytes for the k_B field). This means that the total memory requirement of the protection table entries for the generic object is 24 bytes (here, we do not consider the storage requirement for the domain table, as each domain is intended to include several objects and the resulting per-object cost is negligible). Let us now define the memory overhead for storage of the protection information for a given object as the ratio between the size of this information and the object size. For small-sized, 1-KByte objects the memory overhead is 2.3%.

We shall now consider two alternative, significant configurations, that is, a distributed system concentrating all the information for the management of object protection in a single node that we shall call the *master node*, and a single processor machine. In the master node configuration, a single table, which we shall call the *master table*, features one entry for each existing object. The entry for a given object contains the name of the object repository and the object key, for a total memory requirement of 18 bytes (8 bytes for the object identifier, 2 bytes for the name of the object repository and 8 bytes for the object key). For 1-KByte objects the memory overhead is 1.8%. As will be shown in the next subsection, if compared with the fully distributed case, the resulting small memory savings correspond to significantly higher costs in terms of the network traffic generated by the execution of the protection primitives. In the case of a single processor machine, for each object the protection system will have to maintain the association between the object name and the object key, with a total memory requirement of 16 bytes. For 1-KByte objects, the memory overhead decreases to 1.6%.

We may conclude that in our system the global memory requirements of the protection information are a negligible fraction of the overall memory requirements for object storage. Distribution of the protection information produces a marginal increase in memory cost with respect to both a configuration with a master node and a single-processor machine.

4.4. Network costs

Table II shows the network cost of each protection primitive, expressed in terms of the number of the control and object messages transmitted during execution of that primitive. The costs can be easily derived from the analysis of the actions involved in the execution of each primitive, presented in Subsection 3.2. Let N be the node where a given protection primitive is issued, and let PR(B) and RP(B) be the principal and the repository of the object B involved in the execution of this primitive. The figures in the table are relevant to the case of N, PR(B) and RP(B) being distinct nodes. This is a worst-case analysis, corresponding to the highest network traffic. For instance, the *moveObject*() and *copyObject*() primitives need to know the name of the repository of object *B*. To this aim, two control messages are necessary, a request message from *N* to PR(*B*) and a reply message from PR(*B*) to *N*. If *N* = PR(*B*) (i.e. node *N* is the principal of *B*), these two messages are saved. If *N* = PR(*B*) = RP(*B*) (by far the commonest situation, corresponding to the case of an object that is used in the node where it was created), no network traffic is generated at all. This important aspect of the network behaviour of our memory protection system follows from the full distribution of the memory protection information among the network nodes.

In terms of the network costs, the protection system is highly scalable. The number of messages that are required to retrieve the protection information for a given object in the distributed protection tables is fixed and does not increase if new nodes are added to the network. These costs are independent of the past movements of the object in the network. The network position of an object is determined without resorting to a message broadcast.

Master node

Let us now consider the network costs connected with execution of the protection primitives in the aforementioned configuration featuring a master node where all the information concerning object protection is kept. We shall hypothesize that the protection primitives are issued in a node *N* which is not the master node. Of course, less messages are required with respect to the full distributed case when execution of the given primitive needs to know the names the object principal and the object repository. This is the case for *moveObject*() and *copyObject*(); for these primitives, the master node configuration leads to a saving of 2 messages with respect to the fully-distributed case (see Table II).

On the other hand, in the presence of a master node, memory traffic follows even for those protection primitives whose execution in the fully distributed configuration is entirely confined within the boundaries of the node where the primitives are issued. This is the case for *loadPtr*(), *storePtr*(), *newObject*() and *deleteObject*(). For all of them, the presence of a master node implies the transmission of two messages to access the object key (in the fully distributed configuration, the object key is stored locally, in the repository table). We may conclude that the configuration with a master node increases the network costs of the protection primitives that are executed more frequently. For all protection primitives, the network costs can be kept to a minimum even in the fully distributed case by introducing forms of caching of the contents of the principal tables.

Repository cache

Let us refer, for instance, to execution in node *N* of the *moveObject*() protection primitive. In a first execution phase (at step 2 in the description given in Subsection 3.2), node *N* sends a request message to the principal PR(*B*) of object *B* to identify the object repository. On receipt of this message, PR(*B*) extracts repository name RP(*B*) from its own principal table $\mathrm{PRT}_{\mathrm{PR}(B)}$ and returns this quantity to *N*. A similar situation takes place in the execution phase of the *copyObject*() primitive. The total network cost of these actions is 2 messages. This network cost can be eliminated, as follows.

In each given node N we maintain a table that we call the *repository cache* K_N. Each entry of the cache contains the name of the repository of an object whose principal is *not* node N (this is in contrast with principal table PRT_N). In the execution of a protection primitive, if node N needs to know the name of the repository of an object whose principal is a different node, cache K_N is inspected first, and, if a hit is found, the repository name is taken from the cache. Of course, if the cache access produces a miss, interaction with the object principal PR(B) becomes mandatory; in this case, the repository name returned by PR(B) is added to the cache, thereby becoming available for any subsequent protection primitive involving B.

It may well be the case that the information contained in K_N be outdated, for instance, for a given cache entry, if the object corresponding to this entry has been moved to a different node. The protection primitives should be able to comply with situations of this type. Let us refer again to *moveObject*(), and let us suppose that the name of the repository of object B contained in K_N is outdated. Consequently, at execution step 3, the message asking for the object value is sent to the wrong node, say $RP(B)_{old}$. On receipt of this message, node $RP(B)_{old}$ inspects its own repository table in search for object name B. Of course, this search will fail, and a negative reply message will be returned to N. In this way, node N ascertains lack of coherency between the contents of cache K_N and the real situation of object allocation. Consequently, N interrogates principal PR(B) for the correct repository name and uses this name to update the cache.

Table II gives the number of control messages involved in the execution of the protection primitives in a system configuration featuring the repository caches in the hypothesis of cache hits (of course, the caches have no influence on the number of object messages). It is worth noting that, for all protection primitives, the network costs become equal to or smaller than the costs in the master node configuration. Positive effects follow from the presence of the caches in terms of node throughput, as a consequence of the fewer context switches that are necessary in the execution of the protection primitives to suspend process execution after sending a request message and resume execution later, on receipt of the reply.

4.5. Cryptographic costs

As seen in Subsection 3.2, a single protection primitive, the *loadPtr*() primitive, produces a cryptographic transformation of a ciphertext o-pointer into plaintext form. This primitive then loads the plaintext into an entry of the pointer table of the issuing process. Afterwards, the contents of this entry will be used in all subsequent accesses by that process to the object referenced by that o-pointer, and these accesses will be performed by taking advantage of the *operation*() protection primitive. Thus, the cost in terms of execution times of this cryptographic o-pointer transformation corresponds to a potentially unlimited sequence of object accesses; comparatively, this cost is low. The pointer table acts as a form of software-controlled cache of the cryptographic transformations. This table eliminates the need to translate an o-pointer into plaintext at each object access, in the execution of the *operation*() primitive.

The inverse transformation of a plaintext o-pointer into ciphertext form is performed as part of the actions involved in the execution of two protection primitives, namely *storePtr*() and *convertPtr*(). *storePtr*() is mainly used after creation of a new object to deposit a ciphertext o-pointer for this object into memory. It is also used to generate a copy of a given o-pointer with reduced access rights, in view of the distribution to a different process in the same domain. Of course, these actions are relatively infrequent. *convertPtr*() is the only protection primitive that crosses the domain boundaries. Utilization of this primitive is restricted to the conversion of an o-pointer for transmission to a process of a different domain. The cryptographic cost is that of a single o-pointer transformation, from plaintext into ciphertext.

We may conclude that in our system the cryptographic cost of o-pointer transformations is not a critical factor (on the other hand, hardware support to these transformations may be easily devised [5]).

The implementation of the protection system should guarantee the necessary degree of security in the exchange of messages between the nodes. This aspect of overall system security can be approached by *ad-hoc* solutions including forms of cryptography in internode communication. The consequent costs in terms of execution times are inherent in distributed system organizations, and are not connected with the cryptographic form of o-pointers in memory. Indeed, if a protection primitive is issued in a given node, any cryptograph transformation involved in the execution of that primitive occurs within the boundaries of that node, and no network cost is connected with the transformation.

4.6. Cryptographic pointers

The utilization of cryptographic techniques in the implementation of forms of protected memory pointers is certainly not a new idea. These techniques have been used, for instance, in the design of a protection system for a single-processor computer architecture in which the protected entities are the memory segments [17]. In this system, a segment pointer contains a segment identifier and the specification of a set of access rights, and is stored in memory in a form resulting from application of a cryptographic transformation. The protection problem is approached by taking advantage of *ad-hoc* hardware. The processor features a number of registers, the *protection registers*, which are reserved to contain information concerning memory addressing and protection. Examples of protection registers are the *segment registers*, each of which is aimed at containing a segment pointer in plaintext. The segment registers are mainly used to reference segments in memory. A memory address has the form <<*sr*>> *f*, where *sr* denotes a segment register and *f* is an offset in the segment referenced by this register. The instruction set of the processor is designed to comply with this address format. The protection primitives operate on internal processor resources; as such, they are intended to be implemented as machine instructions (in certain cases, software-level support is necessary, e.g. when a new segment is allocated in memory). A protection domain is a collection of segment pointers. A cryptographic key is associated with each domain, and all segment pointers encrypted by using a given domain key are part of the corresponding domain. At any given time, a register of the processor, the *current domain register*, contains a pointer to the current domain. In the execution of a given process, the current domain may well change, and a result of this type is obtained by simply loading a pointer for the new domain into the current domain register.

In [18] the cryptographic approach to protected pointer implementation has been applied to a protection system for a single-processor environment in which the protected entities are the typed objects. In this system, the type of each given object defines the operations that can be applied on this object and, for each operation, the access rights that are necessary to accomplish this operation successfully. A *protected reference* contains the name of an object and the specification of a set of access rights on this object. An encryption key, the *object key*, is associated with each object and a further encryption key, the *domain key*, is associated with each domain. A complex cryptographic transformation is used to convert object references to a protected form. This transformation takes advantage of a double encryption using both the object key and the domain key.

In contrast, the system presented in this paper takes advantage of cryptographic techniques for pointer protection in the context of a fully distributed environment. In an environment of this type, a peculiar problem is to identify the node reserving storage for every given object; the concepts of the object principal and the object repository have been introduced to solve this problem. The system does not rely on an *ad-hoc* processor, and no special requirement exists for the hardware configuration of each node. Instead, the system can be implemented within the framework of a conventional processor architecture, at software level. This is especially true for the protection primitives. The concept of a protection domain is tightly connected with that of a process. When a process is generated, it is assigned a new domain or, if it was generated by an existing process, it is assigned the same domain as its father process. O-pointer transmission between processes in the same domain corresponds to a simple action of o-pointer copy in memory, whereas for processes in different domains a form of cryptographic o-pointer conversion is necessary.

We may conclude that the previous works [17], [18] demonstrate the possibility to take advantage of cryptographic techniques for pointer protection in single-processor systems. In this paper we show that these techniques can be successfully used in a fully-distributed environment, but a thorough redesign of the protection system is necessary to solve the new problems connected with the management of the distributed information concerning memory addressing and protection.

In the following, we shall consider three important examples of protected systems incorporating forms of cryptographic pointers, namely Amoeba, ICAP and MSSA. For each of them, a few aspects of the protection environment embedded in the system design will be discussed, with special reference to the cryptographic pointer transformations.

Amoeba

Amoeba [24], [25] is an object-oriented, distributed operating system that uses capabilities for object naming and protection. The system takes advantage of cryptography to prevent processes from tampering with existing capabilities or forging new capabilities. A capability referencing a given object consists of: (i) a *server port* that identifies the server managing this object; (ii) an *object field* whose value is an index in an internal table of the server, the *object table*, and identifies the table entry reserved for the object; (iii) a *rights field* specifying the access rights granted by the capability on the object; and (iv) a *check field* aimed at protecting the object.

When a client asks a server for creation of a new object, the server reserves an entry for the new object in its own object table and inserts a random number into this entry. Then, the server assembles an *owner capability* for the new object by inserting the random number into the check field. When an owner capability is presented to the server, the server compares the value of the check field with the random number. If a match is found, the capability is considered genuine, and all operations on the named object are allowed.

A client can create a capability for a given object with restricted access rights by passing the owner capability for that object back to the server, together with a bit mask for the new access rights. The new capability is constructed by evaluating the logical XOR of the original random number and the new access rights, and running the result through a *one-way function* (a function that is easy to apply, but hard to invert [23]). The server then creates a new capability featuring the same value as the original capability in the object field, the new access rights in the rights field and the result of the one-way function in the check field.

When a restricted capability is returned to the server for validation, the server ascertains that it is not an owner capability as at least one bit in the rights field is cleared. The server evaluates the logical XOR of the original random number and the contents of the rights field and runs the result through the one-way function. The capability is valid if the result matches the contents of the check field.

ICAP

ICAP [9] is a capability-based protection system aimed at a distributed environment. In ICAP, much attention is paid to the propagation problem, to separate the right to exercise access permissions from the right to grant access permissions. To this aim, capabilities incorporate the identity of the owners.

In ICAP, the data segments are the protected objects to which access is controlled. When a server creates a new object B on behalf of a given client C_1, an *internal capability* is created consisting of pair <B, r_0>, where r_0 is a random number that is kept secret. This capability is not sent to the client. Instead, C_1 is assigned an *external capability* in the form of a triplet <B, AR, r_1>, where AR is the specification of a set of access rights, r_1 is the result of running $f(C_1, B, AR, r_0)$ and f is a one-way function. When the server receives an access request incorporating a capability, it executes the one-way function again and compares the result with the r_1 field; if a match is found, then the capability is validated and the access is granted.

If C_1 wants to pass external capability $<B, AR, r_1>$ to a different client C_2, the request must be explicitly presented to the server that decides whether to allow the propagation according to the intended security policy. If this is the case, the server uses the secret random number r_0 to create external capability $<B, AR, r_2>$, where r_2 is the result of running $f(C_2, B, AR, r_0)$. This capability is then transmitted to C_2. The rationale is that the number of actions of capability propagation is much less than the number of capability uses, and consequently it is much more efficient to control propagation than checking the security policy at access time.

MSSA

In the *Multi-Service Storage Architecture* (*MSSA*) [2], [19], the files are the protected objects, and the clients capable of performing access attempts to the files are called *principals*. A capability list is associated with each principal and specifies the access permissions held by that principal in the form of a collection of capabilities.

An MSSA capability for a given file can only be used by a specific principal and is only valid for a limited period of time. It consists of the name of the principal, the specification of a set of access rights, the expiry time, a signature and an optional comment. The signature is generated by applying a one-way function to the rest of the capability, to the file identifier and to a secret number that is associated by the storage server to the file. The file identifier is not part of the capability; it must be presented as a separate argument by the principal to the storage server. The storage server evaluates the signature again by using the secret number; if a match is found, the capability is validated. Of course, any forged capability will have an incorrect signature. This protects capabilities from tampering.

A principal holding a given capability is free to ask the storage server for a copy of that capability on behalf of a different principal. The expiry time guarantees that the new capability will not remain valid indefinitely. Furthermore, the storage server can revoke all the capabilities associated with a given file immediately by changing the secret number, but this mechanism cannot be used for a selective revocation.

5. CONCLUDING REMARKS

With reference to a distributed system consisting of nodes connected by a local area network, we have considered a salient aspect of the protection problem, the representation of access permissions and protection domains. We have presented a model of a protection system that is based on the application of techniques of symmetric-key cryptography. We have obtained the following results:

- O-pointers can be freely mixed in memory with ordinary data items. It is practically impossible to forge a valid o-pointer for an existing object. These results have been obtained by taking advantage of cryptography without incurring the hardware costs connected with memory tagging, while maintaining simplicity and effectiveness of access right distribution.

- The protection system supports two different models of process interaction, corresponding to different mechanisms of transmission of access permissions. Between processes of the same domain, a simple action of an o-pointer copy does not need to be mediated by the protection system, whereas between processes of different domains, the o-pointer copy must be preceded by a conversion of the o-pointer from the domain of the granting process to the domain of the process that receives the o-pointer. The two models correspond to different degrees of cooperation between processes. For mutually trustworthy processes, a common protection domain improves efficiency in access right transfers. For mutually suspicious processes, domain separation and the cryptographic mechanism connected with domain passwords guarantee a high degree of protection against o-pointer stealing: a process of a different domain will not be able to decrypt a stolen o-pointer and use it for object access. This is in sharp contrast with capability and password capability systems, which cannot prevent fraudulent actions of this type. A salient feature of our domain paradigm is that two or more processes can be part of the same domain even if they are hosted on different network nodes. Rather than being based on the physical allocation of the processes in the network, the paradigm defines process grouping at the logical level of process interactions.
- The memory requirements for storage of the information for object protection are a negligible fraction of the total memory requirements for object storage. These memory costs are fixed and independent of the network size. Distribution of the protection information produces a marginal increase in memory cost with respect to a configuration concentrating all this information in a single master node. Memory overhead is low even if compared with a single-processor machine with a small memory capacity equal to that of a single node.
- The memory traffic originated by the execution of the protection primitives is low and is independent of the number of nodes that form the network. No network cost is connected with protection in the common case of an object that is used in the same node where it was created and allocated in memory. This important result has been obtained by taking advantage of a full distribution of the information concerning memory protection. The number of messages exchanged in the execution of the protection primitives can be reduced further by introducing forms of caching of the distributed information concerning object allocation.

The interesting properties of the protection system presented in the foregoing sections suggest that the utilization of cryptography in the segregation of the information concerning protection can be a valid solution to the problem of domain and access right representation in a fully distributed environment.

REFERENCES

[1] M. Anderson, R. D. Pose, C. S. Wallace, “A password-capability system,” *The Computer Journal*, vol. 29, no. 1 (February 1986), pp. 1–8.

[2] J. Bacon, R. Hayton, S. Lai Lo, K. Moody, “Extensible access control for a hierarchy of servers,” *Operating Systems Review*, vol. 28, no. 3 (July 1994), pp. 4–15.

[3] J. S. Chase, H. M. Levy, E. D. Lazowska, M. Raker-Harvey, “Lightweight shared objects in a 64-bit operating system,” *Proceeding of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, Vancoover, October 1992; in *SIGPLAN Notices*, vol. 27, no. 10 (1992), pp. 397–413.

[4] M. de Vivo, G. O. de Vivo, L. Gonzalez, “A brief essay on capabilities,” *SIGPLAN Notices*, vol. 30, no. 7 (July 1995), pp. 29–36.

[5] T. Eisenbarth, S. Kumar, “A survey of lightweight-cryptography implementations,” *IEEE Design & Test of Computers*, vol. 24, no. 6 (November – December 2007), pp. 522–533.

[6] D. M. England, “Capability concept mechanisms and structure in System 250,” *Proceedings of the International Workshop on Protection in Operating Systems*, IRIA, Paris, 1974, pp. 63–82.

[7] N. Ferguson, B. Schneier, *Practical Cryptography*. Indianapolis, Indiana: Wiley, 2003.

[8] K. Ghose, R. M. Stewart, “The capability mechanism of a VLSI processor,” *Proceedings of the 1988 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, Rye Brook, NY, USA, October 1988, pp.106–109.

[9] L. Gong, “A secure identity-based capability system,” *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, Oakland, CA, USA, May 1989, pp. 56–63.

[10] G. Heiser, K. Elphinstone, J. Vochteloo, S. Russell, J. Liedtke, “The Mungi single-address-space operating system,” *Software – Practice and Experience*, vol. 28, no. 9 (July 1998), pp. 901–928.

[11] M. E. Houdek, F. G. Soltis, R. L. Hoffman, “IBM System/38 support for capability-based addressing,” *Proceedings of the 8th Annual Symposium on Computer Architecture*, Minneapolis, Minnesota, USA, May 1981, pp. 341–348.

[12] R. Y. Kain, C. E. Landwehr, “On access checking in capability-based systems,” *IEEE Transactions on Software Engineering*, vol. SE-13, no. 2 (February 1987), pp. 202– 207.

[13] G. Klein *et al.*, “seL4: formal verification of an OS kernel,” *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, Big Sky, MT, USA, October 2009, pp. 207–220.

[14] H. M. Levy, *Capability-Based Computer Systems*. Bedford, Mass.: Digital Press, 1984.

[15] T. A. Linden, “Operating system structures to support security and reliable software,” *ACM Computing Surveys*, vol. 8, no. 4 (December 1976), pp. 409–445.

[16] L. Lopriore, “Access privilege management in protection systems,” *Information and Software Technology*, vol. 44, n. 9 (June 2002), pp. 541–549.

[17] L. Lopriore, “Encrypted pointers in protection system design,” *The Computer Journal*, vol. 55, no. 4 (April 2012), pp. 497–507.

[18] L. Lopriore, “Reference encryption for access right segregation and domain representation,” *Journal of Information Security*, vol. 3, no. 2 (2012), pp. 86–90.

[19] K. Moody, J. Bacon, J. Bates, S. L. Lo, Z. Wu, “OPERA: storage, programming and display of multimedia objects,” *Proceedings of the Fourth Workshop on Future Trends of Distributed Computing Systems*, Lisbon, Portugal, September 1993, pp. 442–448.

[20] D. Mossop, R. Pose, “Information leakage and capability forgery in a capability-based operating system kernel,” *Proceedings of the Workshop On the Move to Meaningful Internet Systems*, Montpellier, France, October 2006; in *Lecture Notes in Computer Science*, vol. 4277 (2006), pp. 517–526.

[21] P. G. Neumann, R. J. Feiertag, “PSOS revisited,” *Proceedings of the 19th Annual Computer Security Applications Conference*, Las Vegas, NV, USA, December 2003, pp. 208–216.

[22] J. S. Shapiro, J. M. Smith, D. J. Farber, “EROS: a fast capability system,” *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, Kiawah Island Resort, SC, December 1999; in *Operating Systems Review*, vol. 34, no. 5 (December 1999), pp. 170–185.

[23] M. Stamp, *Information Security: Principles and Practice*, 2nd Edition, Wiley, 2011.

[24] A. S. Tanenbaum, S. J. Mullender, R. Van Renesse, “Using sparse capabilities in a distributed operating system,” *Proceedings of the Sixth International Conference on Distributed Computer Systems*, Cambridge, MA, USA, May 1986, pp. 558–563.

[25] A. S. Tanenbaum, M. F. Kaashoek, R. van Renesse, H. E. Bal, “The Amoeba distributed operating system – a status report,” *Computer Communications*, vol. 14, no. 6 (July-August 1991), pp. 324–335.

[26] M. V. Wilkes, R. M. Needham, *The Cambridge CAP Computer and Its Operating System*. New York: North Holland, 1979.

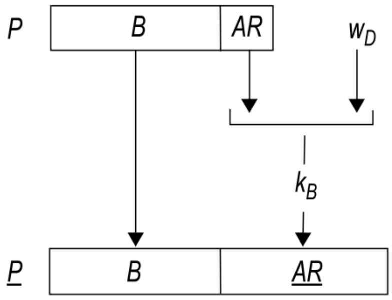

Figure 1. Transformation of o-pointer P = <B, AR> into ciphertext quantity $\underline{P}$ = <B, $\underline{AR}$> as part of domain D with password w_D. $\underline{AR}$ is the result of encrypting pair <AR, w_D> by using a symmetric key cipher and key k_B of object B.

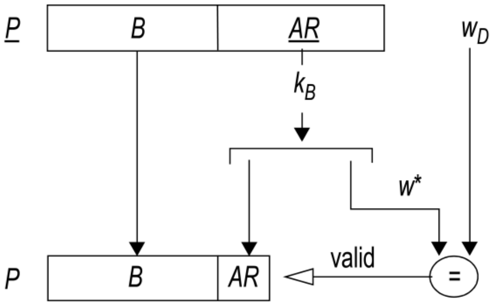

Figure 2. Transformation of ciphertext quantity $\underline{P}$ = <B, $\underline{AR}$> into the corresponding plaintext o-pointer P = <B, AR>, and validation of the result. Key k_B of object B is used to convert quantity $\underline{AR}$ into plaintext pair <AR, w*>. Quantity w* is compared with domain password w_D: if a match is found, P is valid.

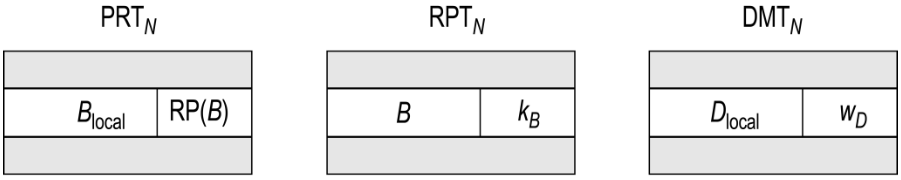

Figure 3. Protection tables in node N. For each object B whose principal is N, the principal table PRT_N contains its local identifier B_{local} and the name RP(B) of its repository. For each object B whose repository is N, the repository table RPT_N contains its object key k_B. For each domain D created in node N, the domain table DMT_N contains its local identifier D_{local} and its password w_D.

Table I. Protection primitives.[1]

loadPtr(*addr*, *i*)

Decrypts the ciphertext o-pointer stored in memory location *addr* of node *N*, and deposits the resulting plaintext into PT$_i$.

storePtr(*i*, *mask*, *addr*)

Eliminates access rights from the plaintext o-pointer contained in PT$_i$, as is specified by *mask,* encrypts the result and deposits the ciphertext into memory location *addr* of node *N*.

operation(*i, m*)

Executes the *m*-th operation on the object referenced by PT$_i$, as is defined by the type of this object.

newObject(*T*, *i*)

Allocates a new object of type *T* in node *N*, and deposits an o-pointer for this object, with full access rights, into PT$_i$.

deleteObject(*i*)

Deletes the object referenced by PT$_i$. Requires access right OWN in PT$_i$.

moveObject(*i*)

Moves the object referenced by PT$_i$ from its present network position to node *N*. Requires access right MOVE in PT$_i$.

copyObject(*i*, *j*)

Creates a new object in node *N* and deposits an o-pointer for this object, with full access rights, into PT$_j$. The new object has the value of the object referenced by PT$_i$. Requires access right COPY in PT$_i$.

convertPtr(*i*, *D*, *addr*)

Encrypts the plaintext o-pointer contained in PT$_i$ by using the password of domain *D*, and deposits the resulting ciphertext into memory location *addr* of node *N*.

[1] Each protection primitive is supposed to be issued in node *N*. PT$_i$ denotes the *i*-th entry of the pointer table of the process executing the primitive.

Table II. Network traffic generated by the execution of the protection primitives.

Primitive	Object message	Control messages		
		Fully distributed	Cache	Master node
loadPtr()	–	–	–	2
storePtr()	–	–	–	2
operation()	–	–	–	–
newObject()	–	–	–	2
deleteObject()	–	–	–	2
moveObject()	1	5	3	3
copyObject()	1	5	3	3
convertPtr()	–	2	2	2