Password capabilities revisited

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 present a new formulation of the password capability paradigm that takes advantage of techniques of symmetric-key cryptography to represent password capabilities in memory. We assign a cryptographic key to each application; the password capabilities held by a process of a given application are encrypted by using the key of this application. Passwords are associated with object types; two or more objects of the same type, which are allocated to the same node, share the same set of passwords.

Our password capability paradigm preserves all the advantages concerning simplicity in access right representation and administration (distribution, verification, review and revocation) that characterize the classical paradigm, while keeping the memory requirements for password storage low, and solving the problems connected with password capability stealing and forging.

Keywords: access right, distributed system, password capability, protection, revocation, symmetric-key cryptography.

1. INTRODUCTION

1.1. Capability-based addressing

Let us refer to a distributed system consisting of nodes connected by a local area network in an arbitrary network topology. In a system of this type, an important problem is the representation of access rights in memory. A classic solution is based on the concept of a *capability* [1]. This is a pair (*G*, *AR*), where *G* is the identifier of a protected object, and *AR* is a set of access rights on this object. The capability makes it possible to access the object identified by *G* to perform the operations corresponding to the access rights specified by *AR*. Object identifiers are unique system-wide; it is never the case that two objects exhibit the same identifier, even if these objects are hosted in different nodes.

The segregation problem

A salient problem of capability-based addressing techniques is capability segregation in memory. This is necessary to prevent a process from tampering with an existing capability to alter its access right field to amplify access rights illegitimately, for instance, or even to cause the capability to reference a different object. Segregation will also preclude processes from forging new capabilities from scratch.

Several solutions have been proposed for the segregation problem [2], [3]. Special objects, which we shall call the *capability objects*, can be reserved for capability storage (in contrast, *data objects* will contain ordinary data items) [4], [5]. This approach is subject to object proliferation. Processes must adhere to a complicated memory structure complying with a hierarchical object organization in which one or more capability objects are reserved to contain the capabilities for other capability and data objects, and the data objects form the lowest level in the hierarchy. Even the simplest data structure should include at least one capability object and one data object. In a distributed system, this implies that the transfer of the data structure to a different node must be preceded by the marshalling of its component objects to linear form. In the recipient node, the data structure will be unmarshalled to reconstruct its hierarchical composition.

In a different approach, capability segregation takes advantage of a tagged memory system. In this approach, a one-bit *tag* is associated with each memory cell to identify the cells reserved to contain a capability [6], [7], [8]. The usual machine instructions for data processing are destined to fail if they are executed on a memory cell tagged to contain a capability; instead, a cell of this type can only be accessed by using a set of special instructions, the *capability instructions*, aimed at capability processing. Tag-based capability segregation needs an *ad-hoc* memory system aimed at supporting cell tags, e.g., in a 64-bit primary memory, the size of a memory cell is 65 bits. This is contrary to hardware standardization [9]. Complications ensue in secondary memory management from the necessity to transfer the tags as part of each memory swapping activity. In a distributed memory system, when an information item is moved between two network nodes, the corresponding tags should be transferred, too.

The revocation problem

Capabilities can be freely copied, possibly with reduced access rights; an action of this type grants an access permission to the capability recipient. Indeed, ease of access right transfers is a salient aspect of capability-based addressing environments. In turn, a process that receives a capability is free to transmit this capability further. In a distributed system, capability proliferation in memory is exacerbated by the possibility that copies of the same capability spread on different nodes.

A related problem is that of capability revocation [10]. The original owner of a given capability should be in a position to revoke the existing copies of that capability from the respective recipients. Revocation should extend to all subsequent copies, recursively.

Stolen capabilities

The validity of a given capability is independent of the process that holds this capability. It follows that a process that steals a capability can take advantage of this capability, to access the object it references illegitimately. This is a serious security problem of capability based addressing systems.

It should be clear that the extent of a stolen capability might extend well beyond the object referenced by that capability. In a system that segregates capabilities into capability objects, let us consider a capability that references a capability object, for instance. A process that steals this capability will be in a position to access all the objects referenced by the capabilities contained in that capability object. This is true even if the stolen capability grants a read-only access permission for the capability object [11].

1.2. Password capabilities

Password capabilities are an important improvement to the classical capability concept [12], [13], [14]. In a protection system based on password capabilities, one or more *passwords* are associated with each protected object. In a possible approach, each password corresponds to a subset of all the access rights defined for that object by its type. A password capability is a pair (G, w), where G is an object identifier and w is a password. If a match exists between w and one of the passwords of object G, then the password capability grants the access rights associated with the matching password on G.

Password capabilities are protected from forging by the password size; for large passwords, the probability of guessing a valid password is vanishingly low [15]. It follows that password capabilities can be freely mixed in memory with ordinary data items, and consequently, they represent a valid solution to the segregation problem.

In a password capability environment, high memory costs may follow from the necessity to maintain a set of passwords for each object. This is especially true if objects are small-sized [16], and if several passwords are associated with each given object. Passwords are a viable solution to the revocation problem. If we modify one or more passwords of a given object, we invalidate all the password capabilities expressed in terms of these passwords (it will be no longer possible to use these password capabilities to access the object they reference).

The validity of a password capability extends system-wide. A process that steals a valid password capability from the legitimate owner will be in a position to exercise all the access permissions granted by that capability on the referenced object. With respect to classical capabilities, this problem is exacerbated by the lack of capability segregation in memory. Storage of password capabilities in the stack and heap memory areas may result in occasions for application of well-known techniques for data stealing [17], [18], for instance.[1]

[1] In fact, the opportunity to steal a capability is strictly dependent on the environment. For instance, in the Walnut

In this paper, we refer to a distributed system featuring an operating system unable to guarantee the origin and the integrity of capabilities. We present a new formulation of the password capability paradigm that improves the classical paradigm in many respects. From now on, we shall use the term *p-capability* to refer to our new paradigm, while reserving the term *password capability* to denote the classical paradigm.

We take advantage of techniques of symmetric-key cryptography in the representation of p-capabilities in memory. To this aim, we assign a cryptographic key, called the *application key*, to each application. An *application* is the result of the execution of one or more closely related, cooperating processes that are possibly allocated on different network nodes. All the processes of the same application are considered mutually trustworthy. In sharp contrast with the classical password capability paradigm, which associates passwords with objects, we associate passwords with object *types*, and we reserve different sets of passwords for the same type in different nodes.

Our protection system does not rely on *ad-hoc* hardware inside the processor or the memory management system; instead, it is designed for a distributed system whose nodes exhibit a conventional architecture, with functionalities to handle p-capabilities retrofitted at software level. We hypothesize that each node supports the two usual execution modes, a kernel (privileged) mode and a user (non-privileged) mode with memory access limitations. A memory management system is deputed to virtual to physical address translation, forcing separation between the kernel space and the user spaces, as is necessary to support storage of cryptographic keys and passwords at kernel level, in reserved memory areas of the protection system.

The rest of this paper is organized as follows. Section 2 introduces our protection model with special reference to the encrypted form of p-capabilities in memory and the transformation of p-capabilities between plaintext and ciphertext. Section 3 presents a small set of primitives, the *protection primitives*, which form the interface of the protection system to user processes. Section 4 discusses our p-capability paradigm from a number of salient viewpoints, which include p-capability stealing and forging, the review and revocation of access permissions, and the memory requirements for password storage. Section 5 gives concluding remarks. The Appendix illustrates the actions involved in the execution of each protection primitive.

kernel [15] the stack and the heap of a given process are mapped into the address space of that process, which is made invisible to other processes by the virtual space separation enforced by the underlying virtual memory system.

2. THE PROTECTION MODEL

Let us refer to a local area network consisting of up to 2^d nodes. The memory system distributed over the network nodes gives physical support to a protected environment based on typed objects and access rights. Up to 2^g objects can be supported. The g-bit *global* identifier $G = (M, G_L)$ of a given object consists of the d-bit name M of the node where that object is allocated, and a $(g - d)$-bit *local* identifier G_L of the object in that node. An object created in a given node is never moved to a different node. It follows that the node name portion of the global identifier of a given object allows us to determine the present network location of that object. (On the other hand, as will be shown shortly, it is possible to create a copy of an existing object, and the copy may well be allocated to a different node.)

An object type T is defined as a set of values that can be assumed by the objects of that type, a set of operations R_0, R_1, … that operate on these values, and a set of access rights AR_0, AR_1,…. The type definition associates the operations with the access rights, so that each given operation is made possible by possession of an access permission expressed in terms of one or more access rights. In all object types, access right AR_0 is the OWN access right that includes all access rights, and access right AR_1 is the COPY access right that, if applied to a given object, makes it possible to create copies of that object.

2.1. P-capabilities

A set of passwords is associated in each node with each object type. Each password corresponds to one or more access rights. In the following, for the given object type, we shall denote the i-th password associated with this type in node M by $w_{M,i}$. The password corresponding to access right OWN will be called the *owner password* and will be denoted by $w_{M,\text{OWN}}$. We wish to remark that passwords are specific to the node; if objects of a given type are allocated to different nodes, a set of passwords is reserved for that type in each of these nodes.[2]

A p-capability is a pair (G, w) where G is a global object identifier and w is a password. Let T be the type of object G, and suppose that this object is allocated to node M. If a match exists between w and the one of the passwords associated with T in M, say password $w_{M,i}$, then p-capability (G, w) grants the access rights corresponding to $w_{M,i}$ on the object identified by G.

P-capabilities are never stored in memory in plaintext. Instead, they are protected from tampering by a form of symmetric-key cipher. As seen in Section 1, the protection system associates a cryptographic key, the application key, with each application. The key of a given

[2] Thus, passwords are never passed between different nodes. No network overhead is connected with password management, and no security problem follows from password transmission across the network.

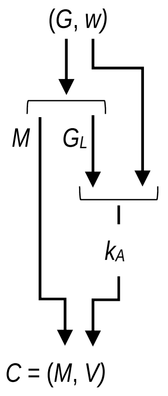

Figure 1. Transformation of p-capability (G, w) from plaintext to the ciphertext $C = (M, V)$. k_A is the key of the application A of the process executing the transformation. Validation field V is obtained by encrypting pair (G_L, w) using a symmetric-key cipher and key k_A.

application is shared by all the processes that form this application, and is stored in each node that hosts one of these processes, in the memory area of the protection system.[3]

Let $G = (M, G_L)$ be the global identifier of a given object, where M is the name of the node storing G, and G_L is the local identifier of G in M. Furthermore, let A be an application, let k_A be the key of this application, and let w be a password. Figure 1 shows the transformation of p-capability (G, w) from plaintext to the ciphertext $C = (M, V)$. Quantity V is called the *validation field* and is obtained by encrypting pair (G_L, w) using a symmetric-key cipher and key k_A. The cipher should guarantee a careful mixing of the bits of G_L and w, so that in V it will be impossible to separate the part corresponding to G_L from the part corresponding to w. It should be noted that, in the transformation of (G, w) to $C = (M, V)$, node name M is not encrypted.

Figure 2 shows the reverse transformation of p-capability $C = (M, V)$ to plaintext. Application key k_A is used to convert validation field V to plaintext pair (G_L, $\underline{w}$). Let T be the type of object $G = (M, G_L)$. Quantity $\underline{w}$ is compared with the passwords in the set S_T of passwords associated with type T in node M. If a match is found and w is the matching password, then p-capability C is valid and grants the access rights associated with w on object G.

As seen previously, different sets of passwords are associated with the given type T in different nodes. On the other hand, an object that was allocated to a given node is never moved to a different node. This means that the passwords granting access permission to a given object

[3] When a new application is created, a cryptographic key is generated for that application. This key is distributed to all the nodes that contain a process of the new application. If a process is added to an application, the corresponding application key will be distributed to the node where the new process is allocated. We may conclude that the actions of key distribution are comparatively rare, and the costs in terms of network traffic connected with these actions are negligible. The usual security measures will be used in the protocols for inter-node communications (e.g. message encryption, and prevention of forms of replay attacks) [19], [20]. We shall take advantage of the separation between the kernel address space and the user address spaces for secure storage of the application keys in a kernel space area.

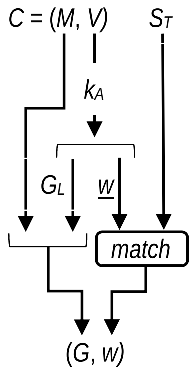

Figure 2. Transformation of p-capability $C = (M, V)$ from ciphertext to the plaintext (G, w), and validation of the result. Key k_A is used to convert validation field V into the plaintext pair $(G_L, \underline{w})$. Quantity $\underline{w}$ is compared with the passwords in the set S_T of passwords associated with the type T of object G in node M. If a match is found, C is valid, and it grants the access rights associated with the matching password on G.

never change. When process p_A of application A creates object G of type T in node M, a p-capability $C = (M, V)$ is generated for the new object. This p-capability contains node name M. The validation field is obtained by encrypting the local object name G_L and the owner password $w_{M,\mathrm{OWN}}$ associated with T in M; the encryption key is the key k_A of application A.

We wish to remark that the key used to encrypt p-capabilities is application-specific. This is an important security measure against the stealing of p-capabilities; this issue will be discussed in detail in subsequent Section 4.1.

3. THE PROTECTION PRIMITIVES

The protection system defines a set of primitives, the *protection primitives*, aimed at p-capability processing (Table I). Execution of a protection primitive is completely accomplished within the boundaries the node where this primitive is issued (the *current* node), or, for a few primitives, it can imply cooperation with a different node. This will be the case for a primitive involving a given object, which is issued in a node that does not store this object. All primitives can be fully implemented at software level by system routines; they will be executed in the privileged state, to have access to the memory areas reserved by the kernel for storage of cryptographic keys and passwords.

In the rest of this section, we shall illustrate the effects of the execution of each protection primitive from the point of view of the process that issues the primitive. A more detailed presentation of the actions involved in the execution of each primitive can be found in the Appendix.

Table I. Protection primitives.[1]

$C \leftarrow$ *newObject*(T) Allocates a new object of type T in the primary memory of the current node. Returns a p-capability that references this object and contains access right OWN.
deleteObject(C) In the current node, deletes the object referenced by p-capability C. Requires access right OWN in C.
$S \leftarrow$ *operation*(C, i) Accesses the object referenced by p-capability C, and executes operation R_i of the type T of this object. Returns the result of this operation. Requires the access rights corresponding to R_i in C.
$C' \leftarrow$ *copyObject*(C) Allocates a new object in the primary memory of the current node, and returns a p-capability referencing this object, with access right OWN. The new object has the value of the object referenced by p-capability C. Requires access right COPY in C.
$C' \leftarrow$ *reduce*(C, i) Returns a p-capability for the object referenced by C, defined in terms of the i-th password of the type of this object. Requires access right OWN in C.
$C' \leftarrow$ *convert*(C, *app*) Returns a p-capability that references the same object, and includes the same access rights, as p-capability C. The resulting p-capability is encrypted by using the key of application *app*.

[1] The *current node* is the node where the given protection primitive is issued.

3.1. Object allocation and deletion

A first example of a protection primitive is the $C \leftarrow$ *newObject*(T) primitive. Its execution in node M allocates an object of type T in the primary memory of this node and returns a p-capability for the new object. This p-capability specifies access right OWN, that is, it includes the password $w_{M,\text{OWN}}$ of type T that corresponds to this access right in node M.

We wish to remark that *newObject*() can only be used to create an object in the primary memory of the node where this primitive is issued; it is impossible to take advantage of *newObject*() to allocate an object in a remote node.

Object deletion is supported by the *deleteObject*(C) primitive whose execution in node M deletes the object G referenced by p-capability C. This p-capability should specify access right OWN, that is, it should include the password $w_{M,\text{OWN}}$ of type T in node M. Execution is successful only if object G is contained in the primary memory of node M, that is, the node name field of p-capability C should contain quantity M. This means that a process running in a given node cannot delete an object stored in the primary memory of a different node.

3.2. Accessing an object, and creating an object copy

Let $G = (N, G_L)$ be an object of type T allocated to node N, where G_L is the local object identifier, and let R_i be the generic operation defined by T. Furthermore, let $C = (N, V)$ be a p-capability that references G. Execution in node M of operation R_i on object G is made possible by protection primitive $S \leftarrow$ *operation*(C, i). Execution returns the result S of R_i. Execution terminates successfully only if the password in C includes the access rights corresponding to R_i.

Creation of a copy of a given object is supported by the $C' \leftarrow$ *copyObject*(C) protection primitive. Let $G = (N, G_L)$ be an object of type T allocated to node N, where G_L is the local object identifier, and let $C = (N, V)$ be a p-capability that references G and includes the COPY access right. Execution in node M of this primitive allocates an object of type T in the primary memory of this node and returns a p-capability C' for the new object. This p-capability specifies access right OWN in terms of the password $w_{M,\text{OWN}}$ of type T in node M. The new object has the value of object G.

3.3. P-capability reduction

Suppose that a process p_A of application A holds a p-capability including the OWN access right for object G of type T. If p_A transfers this p-capability to another process p'_A of the same application, this process will be in a position to access G and perform all the operations defined by type T, including object deletion. Capability *reduction* is the action of transforming a p-capability with the OWN access right into a new p-capability with a reduced privilege. An action of this type will be possibly performed before transferring the p-capability to limit the access privilege of the recipient.

Let $G = (N, G_L)$ be an object of type T allocated to node N, where G_L is the local object identifier. Let $C = (N, V)$ be a p-capability that references G and specifies the OWN access right. Execution of protection primitive $C' \leftarrow$ *reduce*(C, i) returns a p-capability C' referencing object G, defined in terms of the i-th password $w_{N,i}$ of type T in node N.

3.4. Inter-application p-capability conversion

Let p_A and p_B be processes of two different applications A and B, and let k_A and k_B be the keys of these applications. Suppose that p_A holds p-capability $C = (N, V)$; this p-capability is encrypted by using k_A. Suppose also that p_A grants C to p_B, and p_B tries to take advantage of C to access the object it references, for instance, by executing the *operation*() primitive. In the execution of this primitive on behalf of p_B, C is converted to plaintext (see the Appendix); this action will use key k_B instead of key k_A that was used to generate this p-capability. Consequently, p-capability validation, as illustrated in Figure 2 and involved in the execution of *operation*(), is destined to fail.

In fact, transmission of a p-capability between processes of different applications must be preceded by an inter-application conversion of the p-capability, from the key of the granting process to the key of the recipient process. In the foregoing example, process p_A should convert p-capability C from key k_A to key k_B before granting this capability to process p_B. To this aim, p_A executes the $C' \leftarrow$ *convert*(C, *app*) protection primitive, where *app* is the name of the application of the recipient process (B, in our example). Execution of this primitive uses key k_A to convert p-capability C to plaintext, first, and then uses the key of application *app* to convert the plaintext to ciphertext p-capability C', which is returned to the caller. Execution of this primitive is possible in node M only if the key of application *app* is stored in this node. This means that a process of *app* should be executed in M.

4. DISCUSSION

4.1. Stealing p-capabilities

As seen in Section 1, a serious security problem of classical password capability environments is that of stolen password capabilities. This problem follows from the fact that the validity of a given password capability extends system-wide and is independent of the process that generated that password capability. In a distributed system, this problem is exacerbated by the fact that the validity of a password capability is independent of the node where that password capability is stored.

The opportunity to steal a capability depends strictly on the environment and the implementation. For instance, the Walnut kernel precludes password capability stealing by a strict separation between user address spaces enforced by the underlying virtual memory system [15]. In our system, the validity of a p-capability is confined within the boundaries of an application. For instance, let us consider two processes p_A and p_B that are part of different applications A and B, respectively, and let k_A and k_B be the keys of these applications. Suppose that process p_A holds p-capability $C = (M, V)$; this p-capability is encrypted by using key k_A. Suppose also that process p_B steals C, and then tries to take advantage of this p-capability to access the object it references, for instance, by executing protection primitive *operation*(). Execution of this primitive uses the key k_B of the application B of the issuing process p_B to transform the validation field V of p-capability C from ciphertext to plaintext. Pair (G_L, $\underline{w}$) resulting from the transformation is sent to node M for validation. In fact, C was encrypted by using key k_A and is decrypted by using key k_B. Thus, quantity $\underline{w}$ is meaningless, and validation is destined to fail.

We wish to point out that the security mechanism illustrated above does not apply to processes of the same application, which are always considered mutually trustworthy. Furthermore, as seen in Section 3.4, protection primitive *convert*() allows the issuing process to convert a p-capability from the key of its own application to the key of a different application; however, this primitive cannot be used for a conversion in the opposite direction, from an arbitrary key to the key of the issuing process. In the previous example, process p_B that stole p-capability *C*, encrypted by using key k_A, cannot take advantage of *convert*() to translate this p-capability from key k_A to its own key k_B.

4.2. Forging p-capabilities

Let *C* and *C*' be p-capabilities that reference two objects of the same type, and suppose that *C*' includes the owner password. Let p_A be a process that holds both *C* and *C*', and suppose that p_A tries to extract the owner password from the validation field of *C*' and insert it into the validation field of *C*. In fact, this illegitimate attempt to amplify the access rights is destined to fail. As stated in Section 2.1, the cipher used to transform a p-capability from plaintext to ciphertext guarantees a careful mixing of the bits of local object identifier and the password, so that in the validation field it is impossible to separate the two components.

Similarly, suppose that process p_A tries to modify p-capability *C* to obtain a p-capability referencing a different object of the same type. To this aim, p_A should replace the validation field of *C* to insert the local identifier of the other object. This a partial modification of the validation field, which is virtually impossible.

Now suppose that process p_A modifies p-capability $C = (M, V)$ by replacing node name *M* with a different node name, say *N*. Let $C' = (N, V)$ be the p-capability resulting from the modification. Process p_A will try to take advantage of *C*' to access the object it references, by executing protection primitive *operation*(), for instance. In the first phase of the execution of this primitive, key k_A of the application *A* of process p_A is used to convert the validation field *V*' of p-capability *C*' to plaintext. Let $(G_L, \underline{w})$ be the result of this conversion, where G_L is the local name of an object supposedly allocated to node *N*. Pair $(G_L, \underline{w})$ is sent to node *N* for validation. However, $\underline{w}$ is a password of type *T* in node *M*, and the passwords depend on the node; consequently, no match will be found, and validation will fail.

Finally, suppose that process p_A tries to forge a p-capability for a given object *G* from scratch. The validation field of this p-capability should be obtained by using key k_A of application *A* to encrypt the local identifier of *G* and one of the passwords of the object type. In fact, the application key and the passwords are stored in reserved memory areas of the protection system, and process p_A cannot access them. Suppose that p_A resorts to using an arbitrary value for the validation field. Decryption of a p-capability forged in this way will produce a casual local object identifier and a casual password. Of course, validation of this p-capability is destined to fail.

4.3. Memory requirements

Our system associates passwords with object types, and reserves different sets of passwords for the same type in different nodes. In a given node, two or more objects of the same type share a single set of passwords. With respect to the classical password capability paradigm, which associates passwords with objects, significant savings follow in our approach in terms of memory space for password storage. This is especially true if the system should support a large number of small-sized objects [16], as will be the case if we are aimed at exercising protection at a high level of granularity [21], [22].[4]

We have obtained this result by relying on the cryptographic form of p-capabilities in memory. In a classical password capability environment, passwords are stored in plaintext; of course, if objects could share the passwords, it would be possible to use the password of a given object to forge a password capability for a different object. Instead, in our environment, the validation field of a p-capability contains a local object identifier that is indissolubly linked to the password, cryptographically; as seen in the preceding Section 4.2, it is impossible to extract the password and use it to forge a p-capability for a different object.

It is worth remarking that, in a p-capability, the node name is not encrypted; the conversion process from plaintext pair (G, w) to the ciphertext $C = (M, V)$ does not modify quantity M (see Figure 2). It follows that the two components, the node name and the validation field, do not need to be stored in contiguous memory locations. Furthermore, a process that holds two or more p-capabilities for objects allocated to the same node may well maintain a single copy of the node name. The process will reconstruct the association of the node name with the validation field of a given p-capability before using this p-capability, to transmit the p-capability to another process or to execute a protection primitive, for instance.

4.4. P-capability revocation

In Section 1 we introduced the problem of the revocation of access permissions. Several solutions have been proposed to this problem. A reference monitor can be associated with a protected object to manage all the access rights for that object [11]. Capabilities can be short-

[4] For each object, at least two passwords are necessary, the owner password granting the OWN access right and the copy password granting the COPY access right that makes it possible to generate object copies. Of course, more passwords are necessary for several access rights, as will be the case if fine-grained protection should be supported. In a classical password capability environment, for 64-bit passwords and small objects, the memory cost for password storage can be significant in percentage. In our protection environment, this cost is paid only once for each object type, and is negligible. Of course, if memory is large, memory cost is a less critical factor. This may well be the case, especially given Moore's law [23].

lived, so that the access permission granted by a given capability needs to be periodically renewed [24]. A propagation graph can be associated with a given capability to keep track of all copies of that capability [10]. These solutions tend to subvert the main characteristic of capability-based addressing systems, i.e. simplicity in access right distribution. In a distributed system, the propagation graph extends beyond the boundaries of the node that contains the original capability, for instance. Complex message exchanges across the network will be necessary for the periodical renewal of capabilities, or, in the presence of a reference monitor, to consult the monitor at each object access.

In our system, for the given object type, the revocation of access permissions can be obtained at the node level by changing the passwords. If we change the *i*-th password $w_{M,i}$ associated with type *T* in node *M*, we revoke all the p-capabilities expressed in terms of this password, irrespectively of the nodes where these p-capabilities are stored (it will be no longer possible to use them for object access). Revocation extends to the p-capabilities for all the objects of type *T* allocated to node *M*, but it does not affect the p-capabilities for the objects of type *T* allocated to the other nodes. Password substitution is a local action that affects a single node (node *M*, in our example). Thus, no network traffic is generated by an action of p-capability revocation. This feature is especially important if p-capabilities are being frequently distributed and revoked across the network.

Despite its simplicity, our revocation mechanism results to possess a number of interesting properties. Revocation is [10]:

- *Transitive*, that is, if a process transmits a p-capability to other processes, and these in turn grant the p-capability to subsequent recipients, in the same node or in different nodes, the effects of the revocation propagate across the network to all the copies of the original p-capability, recursively, at any transition depth. Indeed, if a password is changed, all the p-capabilities expressed in terms of that password are invalidated, independently of the present network location of these p-capabilities.
- *Temporal*, that is, the effects of the revocation can be reversed through the same mechanism as for revocation. By returning a given password to its original value, we restore the validity of all the p-capabilities expressed in terms of that password, which were invalidated by the password change.
- *Immediate*, that is, a process that holds a p-capability expressed in terms of a given password cannot take advantage of this p-capability to access the object it references, past the time when the password is changed.

Let us suppose that, in node *M*, process p_A of application *A* holds p-capability *C* expressed in terms of password $w_{M,i}$ of type *T*. *C* is encrypted by using key k_A of application *A*. Let us now suppose that p_A takes advantage of protection primitive *convert*() to transform *C* into a p-capability *C*' encrypted by using the key of a different application, say key k_B. The conversion does not change the password (see Section 3.4). It follows that, if we replace password $w_{M,i}$, *C*' is invalidated, too. Indeed, the effects of the revocation are independent of the application.

5. CONCLUDING REMARKS

With reference to a distributed system consisting of nodes connected by a local area network, we have considered the security problems related to the representation and administration (distribution, verification, review and revocation) of the access rights to protected objects. We revisited the classical password capability paradigm to introduce a new paradigm relying on symmetric-key cryptography to represent p-capabilities in memory. In this paradigm:

- A set of passwords is associated with each object type. Each password corresponds to an access permission expressed in terms of one or more access rights. Passwords are specific to the node; different set of passwords are associated with the same given type in different nodes.
- P-capabilities are never stored in memory in plaintext. Instead, a cryptographic key is associated with each application, and is used to encrypt the p-capabilities for the objects allocated by the processes of that application.
- A small set of protection primitives forms the interface of the protection system to user processes. These primitives allow processes to allocate and delete objects in memory, to create copies of the existing objects, and to access the objects and execute the corresponding operations, as are defined by the object types. Two primitives make it possible to transform a p-capability, to reduce the access rights it contains, and to change the application key used to encrypt that p-capability in view of transferring the p-capability to a process of a different application.

We have obtained the following results:

- A process that steals a p-capability from another process of a different application cannot take advantage of this p-capability for object reference. Essentially, this is a consequence of the cryptographic form of p-capabilities in memory: the encryption key is application-specific, and a p-capability loses its validity outside the boundaries of its own application. In contrast, in classical password capability environments, the validity of password capabilities extends system-wide, and a process that steals a password capability is free to use this password capability to access the object it references.

- The cryptographic form of p-capabilities in memory guarantees that, in a given p-capability, it is impossible to separate the part corresponding to the local object identifier from the part corresponding to the password. This prevents attempts to modify an existing p-capability to insert the identifier of a different object, or to amplify the access rights it contains. Any such attempt is destined to produce an invalid p-capability whose utilization for object access is destined to fail.
- Two or more objects of the same type, allocated to the same node, share the same set of passwords. Significant savings follow in terms of memory space with respect to the classical password capability paradigm that associates passwords with objects. This is especially true if the average object size is small, as is the case if we are aimed at exercising protection at a high level of granularity.
- By taking advantage of the possibility to change the passwords, we can implement forms of review and revocation of access permissions. If we replace a given password, we revoke all p-capabilities defined in terms of that password, irrespectively of the nodes where these p-capabilities are stored. This revocation mechanism results to be transitive, temporal and immediate.

Application of cryptographic techniques in the implementation of forms of protected pointers is certainly not a new idea [2], [25]. In [26], protection is exerted at the level of the memory segments. In this proposal, a *segment pointer* specifies an access privilege in terms of the identifier of a segment and a set of access rights for this segment. Segment pointers are always stored in memory in ciphertext. Inside the processor, a set of *segment registers* are reserved for storage of segment pointers in plaintext. A segment pointer referencing a given segment can be effectively used to access an information item in that segment only after it has been converted to plaintext and it has been loaded into a segment register. The protection primitives are designed to be implemented at the hardware level as machine instructions, with partial support at the software level, e.g. for memory management. In a subsequent proposal [27], the basic unit of protection is the single memory page within the framework of a single-processor architecture supporting a form of segmentation with paging. The protection system makes it possible to define *protection contexts* defined in terms of collection of access rights for the pages that form a segment. A protected pointer, called a *handle*, specifies an access privilege in terms of one or more protection contexts for the same given segment. Handles are always stored in the primary memory in ciphertext, and are converted to plaintext for memory reference. The implementation of the protection system relies on *ad-hoc* hardware in the processor and the memory management system. Special *handle registers* are deputed to storage of handles in plaintext. The address translation circuitry includes hardware support for access right verification.

In contrast, in this paper we have considered a distributed system whose nodes include no special hardware for p-capability processing. Each node is required to support the two usual processor modes, kernel and user, and a separation between the kernel space and the user spaces. The protection primitives are designed to be implemented as kernel routines; cryptographic keys and passwords are stored in the kernel space.

Our proposal is aimed at demonstrating that a careful redesign of the password capability paradigm allows us to preserve all the advantages concerning simplicity in access right representation and administration that characterize the classical paradigm, in a distributed environment, while keeping the memory requirements for password storage low, and solving the problems connected with password capability stealing and forging.

ACKNOWLEDGEMENT

The author wishes to thank the anonymous reviewers for their constructive suggestions and insightful comments.

REFERENCES

[1] Levy, H. M. (1984) *Capability-Based Computer Systems*. Digital Press, Bedford, Mass., USA.

[2] de Vivo, M., de Vivo, G. O. and Gonzalez, L. (1995) A brief essay on capabilities. *ACM SIGPLAN Notices*, **30**, 7, 29–36.

[3] Wilkes, M. V. (1982) Hardware support for memory protection: capability implementations. *ACM SIGARCH Computer Architecture News*, **10**, 2, 107–116.

[4] England, D. M. (1974) Capability Concept Mechanisms and Structure in System 250. In *Proceedings of the International Workshop on Protection in Operating Systems*, IRIA, Paris, France, pp. 63–82. IRIA, Paris, France.

[5] Klein, G. *et al.* (2009) seL4: Formal Verification of an OS Kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, Big Sky, MT, USA, October, pp. 207–220. ACM, New York, NY, USA.

[6] Brown, J. *et al.* (2000) A Capability Representation with Embedded Address and Nearly-Exact Object Bounds. Project Aries Technical Memo 5. Available at: *http://www.ai.mit.edu/projects/aries/Documents/Memos/ARIES-05.pdf*

[7] Carter, N. P., Keckler, S. W. and Dally, J. W. (1994) Hardware support for fast capability-based addressing. *ACM SIGPLAN Notices*, **29**, 11, 319–327.

[8] Houdek, M. E., Soltis, F. G. and Hoffman, R. L. (1981) IBM System/38 Support for Capability-Based Addressing. In *Proceedings of the 8th Annual Symposium on Computer Architecture*, Minneapolis, Minnesota, USA, May, pp. 341–348. IEEE Computer Society Press, Los Alamitos, CA, USA.

[9] Meyer, M. (2004) A novel processor architecture with exact tag-free pointers. *IEEE Micro*, **24**, 3, 46–55.

[10] Gligor, V. D. (1979) Review and revocation of access privileges distributed through capabilities. *IEEE Transactions on Software Engineering*, **SE-5**, 6, 575–586.

[11] Shapiro, J. S., Smith, J. M. and Farber, D. J. (1999) EROS: A Fast Capability System. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, Kiawah Island Resort, SC, USA, December, pp. 170–185. ACM, New York, NY, USA.

[12] Chase, J. S., Levy, H. M., Lazowska, E. D. and Raker-Harvey, M. (1992) Lightweight Shared Objects in a 64-Bit Operating System. In *Proceeding of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, Vancouver, October, pp. 397–413. ACM, New York, NY, USA.

[13] Heiser, G., Elphinstone, K., Vochteloo, J., Russell, S. and Liedtke, J. (1998) The Mungi single-address-space operating system. *Software – Practice and Experience*, **28**, 9, 901–928.

[14] Pose, R. (2001) Password-Capabilities: Their Evolution From the Password-Capability System into Walnut and Beyond. In *Proceedings of the Sixth Australasian Computer Systems Architecture Conference*, Gold Coast, Australia, January, pp. 105–113. IEEE.

[15] Castro, M. D., Pose, R. D. and Kopp, C. (2008) Password-capabilities and the Walnut kernel. *The Computer Journal*, **51**, 5, 595–607.

[16] Gehringer, E. F. (1979) Variable-Length Capabilities as a Solution to the Small-Object Problem. In *Proceedings of the Seventh Symposium on Operating Systems Principles*, Asilomar, California, USA, December, pp. 131–142. ACM, New York, NY, USA.

[17] Shahriar, H. and Zulkernine, M. (2010) Classification of Buffer Overflow Vulnerability Monitors. In *Proceedings of the Fifth International Conference on Availability, Reliability, and Security*, Kraków, Poland, February, pp. 519–524. IEEE.

[18] Younan, Y., Piessens, F. and Joosen, W. (2009) Protecting Global and Static Variables from Buffer Overflow Attacks. In *Proceedings of the Fourth International Conference on Availability, Reliability and Security*, Fukuoka, Japan, March, pp. 798–803. IEEE.

[19] Ferguson, N. and Schneier, B. (2003) *Practical Cryptography*. Wiley, Indianapolis, Indiana, USA.

[20] Stamp, M. (2011) *Information Security: Principles and Practice*, 2nd Edition. John Wiley & Sons, Hoboken, New Jersey, USA.

[21] Leontie, E., Bloom, G., Narahari, B. and Simha, R. (2012) No Principal Too Small: Memory Access Control for Fine-Grained Protection Domains. In *Proceedings of the 15th Euromicro Conference on Digital System Design*, Izmir, Turkey, September, pp. 163–170. IEEE.

[22] Witchel, E., Cates, J. and Asanović, K. (2002) Mondrian Memory Protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, California, October, pp. 304–316. ACM, New York, NY, USA.

[23] Mack, C. A. (2011) Fifty years of Moore's law. *IEEE Transactions on Semiconductor Manufacturing*, **24**, 2, 202–207.

[24] Leung, A. W. and Miller, E. L. (2006) Scalable Security for Large, High Performance Storage Systems. In *Proceedings of the Second ACM Workshop on Storage Security and Survivability*, Alexandria, Virginia, USA, October, pp. 29–40. ACM, New York, NY, USA.

[25] Tanenbaum, A. S., van Renesse, R., van Staveren, H., Sharp, G. J. and Mullender, S. J. (1990) Experiences with the Amoeba distributed operating system. *Communications of the ACM*, **33**, 12, 46–63.

[26] Lopriore, L. (2012) Encrypted pointers in protection system design. *The Computer Journal*, **55**, 4, 497–507.

[27] Lopriore, L. (2013) Protection structures in multithreaded systems. *The Computer Journal*, **56**, 4, 478–496.

APPENDIX

This appendix illustrates the actions involved in the execution of each protection primitive. To simplify the presentation, we shall omit details concerning the protocols for inter-node communications (e.g. message encryption, and the message routing algorithms), as well as the usual security measures in these communications (e.g. prevention of forms of replay attack) [20]. Execution of primitives *newObject*(), *deleteObject*() and *convert*() is completely accomplished in the node where these primitives are issued, whereas network traffic is generated by execution of primitives *operation*(), *copyObject*() and *reduce*() for cooperation with a different node.

In the presentation, we shall hypothesize that each protection primitive is issued in node M by an application A whose key is k_A. C denotes a p-capability referencing an object G of type T; G_L is the local identifier of this object in the node where the object is stored.

$C \leftarrow newObject(T)$

1. A primary memory area is reserved in node M for the new object.
2. A new local object identifier G_L is generated.
3. Key k_A is used to convert pair (G_L, $w_{M,\text{OWN}}$) to ciphertext quantity V, where $w_{M,\text{OWN}}$ is the owner password granting access right OWN for type T in node M (see Figure 1). Quantities M and V are paired to obtain p-capability $C = (M, V)$. This p-capability is returned to the caller.

At point 2, let us hypothesize that the size of a local object identifier is so large that identifier reuse is never necessary. In this hypothesis, a simple method to generate a local object identifier is a sequential allocation, as follows. Each node maintains an object counter that, at any given time, contains the local identifier of the object to be allocated next at that time. The object counter is initialized to 0, and is incremented by 1 after creation of a new object.

deleteObject(C)

1. Application key k_A is used to convert the validation field V of p-capability $C = (M, V)$ to plaintext (see Figure 2). Let $(G_L, \underline{w})$ be the result of this conversion.
2. If quantity $\underline{w}$ does not match owner password $w_{M,\text{OWN}}$ of type T in node M, execution of *deleteObject*() terminates with failure; otherwise,
3. Object G is deallocated from node M, and the primary memory area reserved for this object is made free.

$S \leftarrow$ *operation*(C, i)

1. Node M uses application key k_A to convert the validation field V of p-capability $C = (N, V)$ to plaintext. Let $(G_L, \underline{w})$ be the result of this conversion.
2. Node M assembles a message m containing pair $(G_L, \underline{w})$. This message is sent to node N.
3. Node N extracts pair $(G_L, \underline{w})$ from message m. Quantity $\underline{w}$ is compared with the passwords associated with type T in node N. If no match is found, or the matching password does not grant the access rights that make it possible to execute operation R_i, a negative acknowledgement message is sent to node M, and execution of *operation*() terminates with failure; otherwise,
4. Node N executes operation R_i on object G. Let S be the result of this operation.
5. Node N assembles a message m' including quantity S. This message is sent to node M.
6. Node M extracts quantity S from message m'. This quantity is returned to the caller.

$C' \leftarrow$ *copyObject*(C)

1. Node M uses application key k_A to convert the validation field V of p-capability $C = (N, V)$ to plaintext. Let $(G_L, \underline{w})$ be the result of this conversion.
2. Node M assembles a message m containing pair $(G_L, \underline{w})$. This message is sent to node N.
3. Node N extracts pair $(G_L, \underline{w})$ from message m. Quantity $\underline{w}$ is compared with the passwords associated with type T in N. If no match is found, or the matching password does not grant access right COPY, a negative acknowledgement message is sent to node M, and execution of *copyObject*() terminates with failure; otherwise,
4. Node N assembles a message m' containing the value of object G. This message is sent to node M.

5. Node M extracts the value of object G from message m' and reserves an area in its own primary memory for a new object of type T. The value of G is copied into this area.
6. Node M generates a new local object identifier G'_L. Key k_A is used to convert pair (G'_L, $w_{M,\text{OWN}}$) to ciphertext quantity V', where $w_{M,\text{OWN}}$ is the owner password granting access right OWN for type T in node M. Node name M is paired with V' to obtain p-capability $C' = (M, V')$. This p-capability is returned to the caller.

$C' \leftarrow$ *reduce*(C, i)

1. Node M uses application key k_A to convert the validation field V of p-capability $C = (N, V)$ to plaintext. Let (G_L, $\underline{w}$) be the result of this conversion.
2. Node M assembles a message m containing pair (G_L, $\underline{w}$). This message is sent to node N.
3. Node N extracts pair (G_L, $\underline{w}$) from message m. If quantity $\underline{w}$ does not match owner password $w_{N,\text{OWN}}$ of type T in node N, a negative acknowledgement message is sent to node M, and execution of *reduce*() terminates with failure; otherwise,
4. Node N assembles a message m' containing pair (G_L, $w_{N,i}$). This message is sent to node M.
5. Node M extracts pair (G_L, $w_{N,i}$) from message m', and uses application key k_A to convert it to ciphertext quantity V'. Then, node name N is paired with V' to obtain p-capability $C' = (N, V')$. This p-capability is returned to the caller.

$C' \leftarrow$ *convert*(C, *app*)

1. Application key k_A is used to convert the validation field V of p-capability $C = (N, V)$ to plaintext. Let (G_L, $\underline{w}$) be the result of this conversion.
2. The key of application *app* is used to convert pair (G_L, $\underline{w}$) to ciphertext quantity V'. Node name N is paired with V' to obtain p-capability $C' = (N, V')$. This p-capability is returned to the caller.

At point 1, the validity of the password of the original capability C is not checked. In fact, conversion of an invalid capability to a different key produces an invalid capability, which will be rejected at the first subsequent attempt to access the object it references.