Protected pointers to specify access privileges in distributed systems

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

Antonella Santone
Dipartimento di Bioscienze e Territorio, Università del Molise, Contrada Fonte Lappone, 86090 Pesche, Isernia, Italy. Email: antonella.santone@unimol.it

Abstract—With reference to a distributed environment consisting of nodes connected in an arbitrary network topology, we propose the organization of a protection system in which a set of subjects, e.g. processes, generates access attempts to memory segments. One or more primary passwords are associated with each node. An access to a given segment can be accomplished successfully only if the subject attempting the access holds an access privilege, certified by possession of a valid protected pointer (p-pointer) referencing that segment. Each p-pointer includes a local password; the p-pointer is valid if the local password descends from a primary password by application of a universally known, parametric one-way generation function. A set of protection primitives makes it possible to manage the primary passwords, to reduce p-pointers to include less access rights, to allocate new segments, to delete existing segments, to read the segment contents and to overwrite these contents. The resulting protection environment is evaluated from a number of viewpoints, which include p-pointer forging and revocation, the network traffic generated by the execution of the protection primitives, the memory requirements for p-pointer storage, and the relation of our work to previous work. An indication of the flexibility of the p-pointer concept is given by applying p-pointers to the solution of a variety of protection problems.

Keywords: access privilege; distributed system; parametric one-way function; password; protection; segment.

1 INTRODUCTION

Let us consider a protection system in which a set of active entities, the *subjects* $S_0, S_1, \ldots$, generates access attempts to a set of protected, passive entities, the *objects* $B_0, B_1, \ldots$ [23], [26], [40]. A subject can be a scheduled computation (a process), or, in an event-driven environment, a processing activity caused by the occurrence of an event, e.g. a hardware interrupt [30]. The system associates a set of *access rights* with each object; each access right makes it possible to access the object in a specific mode. Thus, a subject is a unit of computation that may possess access rights, and an object is a unit to which specific access rights may be applied [26]. In a classical model, the protection system takes the

form of an *access matrix* AM, featuring a row for each subject and a column for each object [33], [36], [40]. Element $AM_{i,j}$ of the access matrix specifies the *access privilege*, i.e. the set of access rights, held by subject S_i on object B_j.

An important problem in the implementation of a protection system is how to represent the access matrix in memory. A solution is to associate a set of *passwords* with each given object. Each password corresponds to an access privilege for that object. A subject that holds a given password can access the object to carry out the actions permitted by the access rights in the access privilege associated with this password.

1.1 Password proliferation

Passwords tend to proliferate. For each given object, we have one password for each significant access privilege. For instance, for two access rights, we may have up to three passwords, corresponding to each access right separately, and the two access rights in conjunction. These passwords will be stored as part of the internal representation of the object; for small objects, the memory area reserved for password storage can be a significant fraction of the total. Alternatively, we may define only two passwords, separately for the two access rights. In this case, a subject that is granted both access rights owns the two passwords, and an action requiring full access privileges will be permitted by presenting both these passwords. This is an undue complication in access privilege management. Of course, the problem is exacerbated for objects supporting more access rights.

1.2 Password reduction

A further important problem is that of access privilege *reduction*. Let us consider a subject S_0 that holds a password p corresponding to a given access privilege for object B. S_0 can grant this access privilege to a different subject S_1 simply by transmitting p to S_1. So doing, S_1 acquires all the access rights associated with p. Let us now suppose that S_0 is aimed at transmitting only a subset of these access rights. In this case, S_0 sends p to a component of object B, which we shall call the *password manager* PM_B. The password manager returns a password for the reduced access privilege to S_0. If this password does not exist, the entire procedure must be supported by an *ad hoc* ability of PM_B to forge new passwords. If this is indeed impossible, PM_B returns a negative acknowledgement to S_0, and the access right reduction request fails. Of course, this procedure is an undue complication of the whole password management process. In a distributed system, network costs and delays are associated with the necessity to communicate between S_0 and PM_B, if they reside in different nodes.

We may conclude that a mechanism is desirable, allowing a subject that holds a password for a given object to forge passwords for reduced privileges autonomously,

without incurring the costs and complications connected with requests to a password manager.

In a password-based system, the access rights held by a given subject are restricted to the passwords it holds. Therefore, passwords are well suited to the support of the *principle of least privilege* [7], [26], [32]: at any given time, each subject should be granted the minimum privilege that is necessary for that subject at that time to carry out its job. In a least privilege view of access control, each subject is granted access to least possible objects, and we grant this access to least possible subjects [27]. In traditional protection systems, protection is coarse-grained at the application level; different virtual spaces correspond to different applications. In contrast, password systems can support forms of fine-grained memory protection, which can be exercised at the level of a single subject.

1.3 Password review and revocation

With reference to the access matrix model, revocation of an access privilege means to eliminate this access privilege from one or more elements of the matrix. Revocation can be carried out *by column*, i.e. it applies to all, or part of, the subjects that hold a privilege for a given object, or *by rows*, i.e. we revoke the access privileges held by a given subject, for all, or part of, the objects to which these access privileges apply. Revocation by row is especially interesting in a distributed system, for instance, to limit revocation to the access privileges held by a subject in a specific node.

A characteristic of password environments is the ease of access privilege distribution [17], [21]. A subject that receives a copy of a password acquires the same access privilege of the subject that grants this password; in fact, the copy is indistinguishable from the original. The recipient subject is free to transmit the password further. This means that copies of the same password tend to spread throughout the system, and it is hard, if not impossible, to keep track of their position. Even worse, in a distributed environment, the copies can be stored in different nodes. A related problem is that of password *revocation* [10]. After a password has been revoked, it is no longer be possible to use that password for successful object access.

If we modify the internal representation of an object to replace a given password with a new password, we revoke the corresponding access privilege from all the subjects that hold the old password. In a distributed system, revocation is independent of the network location of these subjects. A revocation can be followed by the distribution of the new password. Suppose that we are aimed at revoking an access privilege from a subset of the subjects, e.g. the subjects being executed in a given node. We can change the password, and distribute the new password to the subjects in the other nodes. An approach of

this type has high costs in terms of network traffic, it induces considerable delays due to network propagation, and is an undesirable complication of the whole process of access privilege management.

1.4 Protected pointers

In this paper, we present solutions to the problems, outlined above. We refer to a distributed system consisting of nodes connected by a local area network. The network topology is inessential. We make no hypothesis on the internal architecture of the nodes, the only exception being the provision for the two traditional modes, a kernel mode, and a user mode with memory access limitations. In each node, the primary memory is partitioned into a *private* memory area, which hosts the protection system and can be accessed only from within the node, and a *shared* memory area, which can also be accessed from the other nodes, albeit in a strictly controlled fashion.

The shared memory is segmented. A *segment* is a contiguous memory area completely defined by an *identifier*, a *base* and a *limit*. Identifiers are local to the given node. They are assigned to segments in the order of their creation. The base of a given segment is the address of the first storage unit of this segment. The limit expresses the segment size. Segments can overlap, partially or totally. This means that a memory cell can be part of two or more segments. Segments can have subsegments. A subsegment of a given segment occupies a contiguous memory area, contained within the boundaries of that segment. The subsegment is completely defined by an identifier, a base within the original segment, and a limit that expresses the subsegment size. Subsegment identifers are relative to segments. This means that, for every given segment, its first subsegment is identified by 1 (as will be shown later, subsegment 0 is reserved).

Segments are the basic unit of information protection and sharing between the nodes. Four access rights are defined for a segment, the *read* access right that makes it possible to read the segment contents, the *write* access right that makes it possible to overwrite these contents, the *new* access right that makes it possible to create subsegments within the segment, and the *delete* access right that makes it possible to delete the segment.

An access privilege can be expressed in terms of any combination of the four access rights. A subject can access a given segment only if it owns an access privilege certified by possession of a *protected pointer* for this segment (*p-pointer* from now on, for short). A p-pointer for a segment in the shared memory of a given node includes the node name, the segment identifier, an optional specification of an access privilege, and a *local password*. The p-pointer is valid if the local password is valid. If this is the case, the p-pointer grants the specified access privilege for that segment. If the access privilege specification is lacking, the p-pointer grants a full access privilege, i.e. all the four access rights.

Of course, if we associate a password with each existing segment and each access privilege, the number of passwords grows unacceptably. This is a undesirable flaw that we are aimed at avoiding. Instead, we maintain a small number of passwords in each node, in the private memory area reserved in that node for the protection system. These passwords are called the *primary passwords.* Each primary password has an *identifier* (order number) and a *value.* Each p-pointer includes the identifier of a primary password. The p-pointer is valid if the local password results from the application of a password generation mechanism to that primary password. This mechanism is based on application of a universally-known *generation function.* The number of primary passwords in each given node is related to the possibility to revoke access privileges selectively. If a form of selective revocation is not required, a single primary password is sufficient.

The rest of this paper is organized as follows. Section 2 introduces our protection model, with special reference to p-pointer generation, validation, and revocation. Section 3 presents a set of primitives, the *protection primitives*, which form the subject interface of the protection system. The actions involved in the execution of each primitive are described. Primary password management, segment allocation and deletion, and remote segment access are considered in special depth. Section 4 presents a few examples of practical applications of p-pointers to the solution of a variety of protection problems. This section is especially aimed at giving an indication of the flexibility of the p-pointer concept. Section 5 discusses the proposed protection system from a number of viewpoints, which include p-pointer forging and revocation, the network traffic generated by the execution of the protection primitives, the memory requirements for p-pointer storage, and the relation of our work to previous work. Section 6 gives concluding remarks.

2 THE PROTECTION MODEL

2.1 Protected pointers

Function f is *one-way* if, given a value x, it is easy to compute $f(x)$, but given a value y, it is computationally unfeasible to find a value x such that $y = f(x)$ [2], [18]. One-way functions can be constructed starting from a good cryptosystem, to minimize the design and implementation efforts [25], [31]. In a common approach, a publicly know constant c is encrypted using x as the key, i.e. $f(x) = E_x(c)$ [34]. Function $f_c(x)$ is a *parametric one-way function* if, given a value y and a parameter c, it is computationally unfeasible to find a value x such that $y = f_c(x)$ [37]. Thus, a parametric one-way function is a family of one-way functions, one for each value of the parameter. It can be implemented starting from $f(x) = E_x(c)$, using c as a parameter [34].

As anticipated in Section 1.4, our mechanism for p-pointer generation takes advantage

Table 1: Protected pointers.

Simple pointer $P = (D, \overline{p}_{id}, s_0, p_0)$ D: node name $\overline{p}_{id}$: identifier of a primary password s_0: a segment in the shared memory of D $p_0 = f_{s_0}(\overline{p})$, where $\overline{p}$ is the value of $\overline{p}_{id}$ access privilege: full
Reduced pointer $RP = (D, \overline{p}_{id}, s_0, a_0, p'_0)$ $p'_0 = f_{a_0}(p_0) = f_{a_0}(f_{s_0}(\overline{p}))$ access privilege: a_0
Subpointer $SP = (D, \overline{p}_{id}, s_0, a_0, s_1, p_1)$ s_1: a subsegment of s_0 $p_1 = f_{s_1}(p'_0) = f_{s_1}(f_{a_0}(f_{s_0}(\overline{p})))$ access privilege: a_0
Reduced subpointer $RSP = (D, \overline{p}_{id}, s_0, a_0, s_1, a_1, p'_1)$ $p'_1 = f_{a_1}(p_1) = f_{a_1}(f_{s_1}(f_{a_0}(f_{s_0}(\overline{p}))))$ access privilege: $a_1 \wedge a_0$

of a parametric one-way function, the generation function, which we shall denote by f. A p-pointer that references a segment is called a *simple pointer* (Table 1). Let $s_0 : (b_0, t_0)$ denote a segment in the shared memory of node D, where s_0 is the segment identifier, b_0 is the base, and t_0 is the limit. A simple pointer P that references s_0 has the form $P = (D, \overline{p}_{id}, s_0, p_0)$, where $\overline{p}_{id}$ is the *identifier* (order number) of a primary password of node D, and p_0 is the local password. We have $p_0 = f_{s_0}(\overline{p})$, where argument $\overline{p}$ is the *value* of $\overline{p}_{id}$ (Figure 1a). P references s_0 with a full access privilege.

Let us now consider a subject S_0 that holds simple pointer P. S_0 is in the position to grant a full access privilege for segment s_0 to another subject S_1, being possibly executed in a different node, simply by transmitting a copy of P to S_1. Now suppose that S_0 is aimed at granting subject S_1 only a subset of the access rights for s_0. To this aim, S_0 preventively transforms P into a *reduced pointer* RP. We have $RP = (D, \overline{p}_{id}, s_0, a_0, p'_0)$,

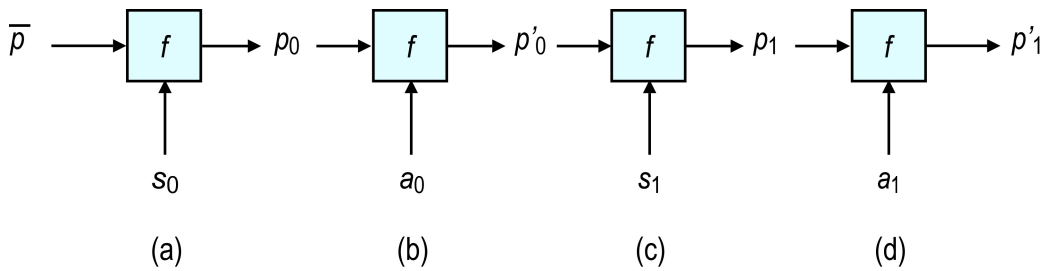

Figure 1: Generation of: (a) the local password p_0 of a simple pointer referencing segment s_0, with a full access privilege; (b) the local password p'_0 of a reduced pointer referencing segment s_0, with access privilege a_0; (c) the local password p_1 of a subpointer referencing subsegment s_1 of s_0, with access privilege a_0; and (d) the local password p'_1 of a reduced subpointer referencing subsegment s_1, with access privilege $a_1 \wedge a_0$.

where a_0 specifies the effective access privilege granted by RP, and password p'_0 is given by relation $p'_0 = f_{a_0}(p_0)$ (Figure 1b). Quantity a_0 is called the *access privilege specifier*. It consists of four bits, corresponding to the four access rights, in the order *new*, *delete*, *read*, and *write*; an asserted bit includes the corresponding access right. In the following, we shall use an abbreviated notation to specify access privileges enclosed in square brackets, e.g. $a_0 = [r]$ includes a single access right, *read*, and stands for the binary 0010, and $a_0 = [ndrw]$ includes all the four access rights, and stands for the binary 1111.

As anticipated in Section 1.4, in our protection model a segment can have subsegments. A subsegment of $s_0 : (b_0, t_0)$ is denoted by $s_1 : (b_1, t_1)$, where s_1 is the subsegment identifier, b_1 is the base of s_1 *within* s_0, and t_1 is the limit of s_1. Thus, the absolute addresses of the first and the last storage units of s_1 are given by $b_0 + b_1$ and $b_0 + b_1 + t_1 - 1$, respectively. The subsegment must be completely included within the boundaries of s_0, thus we have the *inclusion condition* $b_1 + t_1 \leq t_0$. Reduced pointer RP can be transformed into a *subpointer* SP that references s_1. We have $SP = (D, \overline{p}_{id}, s_0, a_0, s_1, p_1)$, where password p_1 is given by relation $p_1 = f_{s_1}(p'_0)$ (Figure 1c). The effective access privilege granted by SP is a_0.

In turn, subpointer SP can be transformed into a *reduced subpointer* RSP that specifies less access rights for the same subsegment s_1. We have $RSP = (D, \overline{p}_{id}, s_0, a_0, s_1, a_1, p'_1)$, where a_1 is an access privilege specifier, password p'_1 is given by relation $p'_1 = f_{a_1}(p_1)$, and the effective access privilege granted by RSP is $a_1 \wedge a_0$, i.e. the access rights in a_1 that are also included in a_0.

Now suppose that a subject received a reduced pointer RP for segment s_0, and is aimed at transmitting this pointer with less access rights. This is indeed possible by taking advantage of the fact that, in a subpointer, subsegment 0, called the *null subsegment*, indicates the original segment. Thus, both reduced pointer RP and reduced subpointer $RSP = (D, \overline{p}_{id}, s_0, a_0, 0, a_1, p''_1)$ reference s_0, but the access privilege in RSP is restricted by access privilege specifier a_1. We have $p''_1 = f_{a_1}(f_0(f_{a_0}(p_0)))$, where f_0 corresponds to the null subsegment, and the effective access privilege is $a_1 \wedge a_0$.

We wish to point out that p-pointers granting the same access privilege may have different passwords. For instance, consider subpointers $RSP_A = (D, \overline{p}_{id}, s_0, a_A, s_1, a_B, p'_{1,A})$ and $RSP_B = (D, \overline{p}_{id}, s_0, a_B, s_1, a_A, p'_{1,B})$. These subpointers reference the same subsegment, s_1, and the access privilege is $a_A \wedge a_B$ in both cases, but the passwords are different. We have $p_{1,A} = f_{a_B}(f_{s_1}(f_{a_A}(f_{s_0}(\overline{p}))))$ and $p_{1,B} = f_{a_A}(f_{s_1}(f_{a_B}(f_{s_0}(\overline{p}))))$.

2.2 Access validation

Let us now consider a subject B that holds simple pointer $P = (D, \overline{p}_{id}, s_0, p_0)$ referencing segment $s_0 : (b_0, t_0)$. When B issues an access attempt to s_0 by using P, e.g. to read the contents of this segment, or to overwrite these contents, the access terminates successfully

only if P is valid, that is, password $\overline{p}_{id}$ exists, and $p_0 = f_{s_0}(\overline{p})$. For a reduced pointer $RP = (D, \overline{p}_{id}, s_0, a_0, p'_0)$, an access attempt to s_0 terminates successfully only if a_0 includes the access right that is necessary to accomplish the access, and RP is valid, that is, $\overline{p}_{id}$ exists, and $p'_0 = f_{a_0}(f_{s_0}(\overline{p}))$.

For a subpointer $SP = (D, \overline{p}_{id}, s_0, a_0, s_1, p_1)$ referencing subsegment $s_1 : (b_1, t_1)$, an access attempt to s_1 terminates successfully only if a_0 includes the access right that is necessary to accomplish the access, and SP is valid, that is, $\overline{p}_{id}$ exists, and $p_1 = f_{s_1}(f_{a_0}(f_{s_0}(\overline{p})))$. For a reduced subpointer $RSP = (D, \overline{p}_{id}, s_0, a_0, s_1, a_1, p'_1)$, an access attempt to s_1 terminates successfully only if quantity $a_1 \wedge a_0$ includes the access right that is necessary to accomplish the access, and RSP is valid, that is, $\overline{p}_{id}$ exists, and $p'_1 = f_{a_1}(f_{s_1}(f_{a_0}(f_{s_0}(\overline{p}))))$.

Finally, for a reduced subpointer $RSP = (D, \overline{p}_{id}, s_0, a_0, 0, a_1, p''_1)$ defined in terms of the null subsegment, an access attempt to segment s_0 terminates successfully only if quantity $a_1 \wedge a_0$ includes the access right that is necessary to accomplish the access, and RSP is valid, that is, $\overline{p}_{id}$ exists, and $p''_1 = f_{a_1}(f_0(f_{a_0}(f_{s_0}(\overline{p}))))$.

2.3 Access privilege revocation

If we delete a segment, all the p-pointers referencing this segment are revoked; it will be no longer possible to use these p-pointers to access memory. As seen in Section 1.4, two or more segments can overlap in memory. If we delete one of these segments, the validity of the p-pointers referencing the other segments is unaffected by the deletion. Similar considerations can be made for subsegments. If we delete a subsegment of a given segment, all the subpointers referencing this subsegment are revoked, but the validity of all the subpointers referencing any overlapped subsegment is unaffected by the deletion.

P-pointers can also be revoked by replacing the value of a primary password with a new value, or by deleting a primary password. Let $\overline{p}_{id}$ denote a primary password of node D. If we change the value of $\overline{p}_{id}$, we revoke all the p-pointers defined in terms of the old value, independently of the node where these p-pointers are stored. In fact, the validation of these p-pointers is destined to fail (see Section 2.2).

Consider two p-pointers referencing the same segment s_0, and defined in terms of different primary passwords, e.g. $P_A = (D, \overline{p}_{id,A}, s_0, p_A)$, and $P_B = (D, \overline{p}_{id,B}, s_0, p_B)$, where $p_A = f_{s_0}(\overline{p}_A)$, $\overline{p}_A$ is the value of $\overline{p}_{id,A}$, $p_B = f_{s_0}(\overline{p}_B)$, and $\overline{p}_B$ is the value of $\overline{p}_{id,B}$. In a situation of this type, if we replace the value of $\overline{p}_{id,A}$ with a new value, we revoke P_A, which is defined in terms of $\overline{p}_{id,A}$; however, the validity of P_B, defined in terms of $\overline{p}_{id,B}$, is not affected by the replacement. After revocation, it will be possible to access segment s_0 by using P_B, but this is no longer true for P_A.

3 THE PROTECTION SYSTEM

3.1 Protection tables

Each given node D contains a *password table* PT_D in the private memory region reserved for the protection system. This table features an entry for each primary password generated in that node. The entry for a given primary password contains the identifier of that password and the password value. A simple method for the generation of primary password identifiers is a sequential generation. Each node maintains a password counter, which is initialized to 0 when the node becomes part of the system, and is incremented by 1 when a new primary password is generated. The identifier of the new primary password is given by the contents of the password counter. Primary password values will be generated at random. They will be sparse and large, according to the security requirements of the system.

As will be shown shortly, when a new segment is allocated, a primary password is used to generate a simple pointer for that segment. We say that the segment is *linked* to this primary password. In node D, a *segment table* ST_D features an entry for each segment in the shared memory of that node. The entry for a given segment contains the identifier s_0, the base b_0, and the limit t_0 of that segment, together with the identifier $\overline{p}_{id}$ of the primary password to which that segment is linked. For each segment, a *subsegment table* is reserved to contain the identifier s_1, the base b_1, and the limit t_1 of each subsegment of that segment.

3.2 Access rights

When a node D is added to the system, a primary password, the *root password* $\overline{p}_{id,R}$, a segment, the *root segment* s_R, and a simple pointer, the *root pointer* P_R, are created in that node as part of the node initialization procedure. The identifier, the base and the limit of s_R are all equal to 0. Memory space is not reserved for s_R. The root pointer P_R references s_R with full access privileges. It has the form $P_R = (D, \overline{p}_{id,R}, 0, p_R)$, where p_R denotes quantity $f_0(\overline{p}_R)$, and $\overline{p}_R$ is the value of $\overline{p}_{id,R}$. Of course, P_R can be transformed into a reduced root pointer to contain less access rights; a result of this type will be obtained by taking advantage of the usual procedure for simple pointer reduction (see Section 2.1).

Access right *read* for the root segment s_R of a given node D allows us to create new primary passwords in D (Table 2). Access right *write* allows us to replace the value of the primary passwords with new values. Access right *delete* is necessary to delete the primary passwords. Access right *new* makes it possible to create new segments in D. For a segment, access right *new* makes it possible to create subsegments in that segment. Access right *delete* allows us to delete the segment. Access rights *read* and *write* make it possible to access the segment to read its contents, and to overwrite these contents, respectively. For subsegments, access right *new* is undefined.

Table 2: Access rights.

Root segment s_R:
new: to create new segments
delete: to delete the primary passwords
read: to create new primary passwords
write: to change the values of the primary passwords
Segment s_0:
new: to create new subsegments
delete: to delete the segment
read: to read the segment contents
write: to overwrite the segment contents
Subsegment s_1:
new: <undefined>
delete: to delete the subsegment
read: to read the subsegment contents
write: to overwrite the subsegment contents

3.3 Protection primitives

The subject interface of the protection system consists of a set of primitives, the *protection primitives.* Table 3 summarizes the actions involved in the execution of each primitive; the rest of this section describes these actions in more detail. The protection primitives are intended to be executed in the kernel mode. This is required to access the protection tables, which are stored in the primary memory region reserved for the protection system.

To simplify the presentation, we shall omit details concerning the communication protocols between the network nodes, e.g. message routing and message encryption. Furthermore, we shall not consider the security issues that are relevant to these communications, e.g. the prevention of forms of replay attack. In the presentation, node D is the *current node*, i.e. the node where the given protection primitive is executed.

3.3.1 Primary password management

Protection primitive $\overline{p}_{id} \leftarrow newPrimaryPassword(G_R)$ generates a new primary password in the current node D, and returns the identifier $\overline{p}_{id}$ of this primary password. Execution is as follows:

1. Argument G_R is validated; it should be a root pointer, or a reduced root pointer specifying access right *read.* If the validation is unsuccessful, execution fails.
2. The identifier $\overline{p}_{id}$ and the value $\overline{p}$ of a new primary password are generated, and are inserted into a free entry of the password table PT_D of node D. Quantity $\overline{p}_{id}$ is returned to the caller.

In step 1, the local password in G_R is compared with quantity $f_0(\overline{p}_R)$, or, if G_R is a reduced root pointer and a_0 is the access privilege specifier, with quantity $f_{a_0}(f_0(\overline{p}_R))$, where 0 is

Table 3: The protection primitives.

$\overline{p}_{id} \leftarrow newPrimaryPassword(G_R)$
In the current node, generates a new primary password, and returns the identifier $\overline{p}_{id}$ of this primary password. Argument G_R should be a root pointer, or a reduced root pointer specifying access right *read*.

$changePrimaryPassword(G_R, \overline{p}_{id})$
In the current node, replaces the value of primary password $\overline{p}_{id}$ with a new value. Argument G_R should be a root pointer, or a reduced root pointer specifying access right *write*.

$deletePrimaryPassword(G_R, \overline{p}_{id})$
In the current node, deletes primary password $\overline{p}_{id}$, and all the segments linked to this password. Argument G_R should be a root pointer, or a reduced root pointer specifying access right *delete*.

$P \leftarrow newSegment(G_R, \overline{p}_{id}, b_0, t_0)$
In the current node, allocates a segment having base b_0 and limit t_0. An identifier s_0 is assigned to the new segment. The segment is linked to primary password $\overline{p}_{id}$. Returns a simple pointer P referencing s_0. Argument G_R should be a root pointer, or a reduced root pointer specifying access right *new*.

$RP \leftarrow reduceSimplePointer(P, a_0)$
Returns a reduced pointer RP derived from simple pointer P by using access privilege specifier a_0.

$SP \leftarrow newSubsegment(G, b_1, t_1)$
In the current node, allocates a subsegment having base b_1 and limit t_1 in the segment s_0 referenced by p-pointer G, which should be a simple pointer, or a reduced pointer specifying access right *new*. An identifier s_1 is assigned to the new subsegment. Returns a subpointer SP referencing s_1 and including all the access rights in G.

$RSP \leftarrow reduceSubpointer(SP, a_1)$
Returns a reduced subpointer RSP derived from subpointer SP by using access privilege specifier a_1.

$deleteSegment(G)$
In the current node, deletes the segment referenced by p-pointer G, which should be a simple pointer, or a reduced pointer specifying access right *delete*.

$deleteSubsegment(G)$
In the current node, deletes the subsegment referenced by p-pointer G, which should be a subpointer, or a reduced subpointer specifying access right *delete*.

$readSegment(G, addr)$
Copies the contents of the segment or subsegment referenced by p-pointer G into an area starting at address *addr* of the private memory of the current node. G should specify access right *read*.

$writeSegment(G, addr)$
Replaces the contents of the segment or subsegment referenced by p-pointer G with quantities taken from an area starting at address *addr* of the private memory of the current node. G should specify access right *write*.

---

the identifer of the root segment. The actions involved in this validation process have been illustrated in Section 2.2. To simplify the presentation, from now on these actions will be simply referred to as a p-pointer validation.

Protection primitive $changePrimaryPassword(G_R, \overline{p}_{id})$ changes the value of primary password $\overline{p}_{id}$ in the current node D. Execution is as follows:

1. Argument G_R is validated; it should be a root pointer, or a reduced root pointer specifying access right *write*. If the validation is unsuccessful, execution fails.

2. A new primary password value is generated, and is inserted into the entry reserved for primary password $\overline{p}_{id}$ in the password table PT_D of node D.

Execution in node D of protection primitive $deletePrimaryPassword(G_R, \overline{p}_{id})$ deletes both the primary password whose identifier is $\overline{p}_{id}$, and all the segments linked to $\overline{p}_{id}$. Execution is as follows:

1. Argument G_R is validated; it should be a root pointer, or a reduced root pointer specifying access right *delete*. If the validation is unsuccessful, execution fails.
2. Segment table ST_D is accessed to delete the table entries reserved for the segments linked to primary password $\overline{p}_{id}$.
3. Password table PT_D is accessed to delete the table entry reserved for $\overline{p}_{id}$.

3.3.2 Allocating new segments

Execution in node D of primitive $P \leftarrow newSegment(G_R, \overline{p}_{id}, b_0, t_0)$ allocates a new segment in D, and returns a simple pointer P referencing this segment. Arguments b_0 and t_0 are the base and the limit of the new segment. An identifier s_0 is assigned to the new segment, and the segment is linked to primary password $\overline{p}_{id}$. Execution is as follows:

1. P-pointer G_R is validated; it should be a root pointer, or a reduced root pointer specifying access right *new*. If the validation is unsuccessful, execution fails.
2. Quantities b_0 and t_0 are considered. If the new segment cannot be completely contained in the shared memory of node D, execution fails.
3. The entry reserved for primary password $\overline{p}_{id}$ in the password table PT_D of node D is accessed to extract the value $\overline{p}$ of this primary password. If no such entry exists, execution fails.
4. The identifier s_0 of the new segment is generated, and quantities s_0, b_0, t_0, and $\overline{p}_{id}$ are inserted into a free entry of segment table ST_D.
5. Quantity $\overline{p}$ and relation $p_0 = f_{s_0}(\overline{p})$ are used to forge simple pointer $P = (D, \overline{p}_{id}, s_0, p_0)$ referencing the new segment. P is returned to the caller.

In step 4, a simple strategy for the generation of segment identifiers is a sequential generation, supported by a segment counter in each node. When node D is initialized, its segment counter is set to 1 (as seen in Section 2.1, segment identifier 0 is reserved). When a new segment is generated, the segment identifier is taken from the segment counter, and then the value of the counter is incremented by 1.

The $RP \leftarrow reduceSimplePointer(P, a_0)$ primitive returns a reduced pointer $RP = (D, \overline{p}_{id}, s_0, a_0, p'_0)$ derived from simple pointer $P = (D, \overline{p}_{id}, s_0, p_0)$ by using access privilege

specifier a_0. Execution of this primitive uses generation function f to evaluate quantity $p'_0 = f_{a_0}(p_0)$ (see Section 2.1).

A subsegment of a given segment s_0 can be allocated by using primitive $SP \leftarrow newSubsegment(G, b_1, t_1)$. Arguments b_1 and t_1 are the base and the limit of the new subsegment. An identifier s_1 is assigned to the new subsegment. The primitive returns a subpointer SP referencing s_1. Execution is as follows:

1. Argument G is validated; it should be a simple pointer referencing segment s_0, or a reduced pointer referencing s_0 with access right *new*. If the validation is unsuccessful, execution fails.
2. Quantities b_1 and t_1 are considered. The new subsegment should be completely contained within the memory area reserved for s_0, i.e. the inclusion condition $b_1 + t_1 \leq t_0$ should be satisfied. If this is not the case, execution fails.
3. The identifier s_1 of the new subsegment is generated, and quantities s_1, b_1, and t_1 are inserted into a free entry of the subsegment table of s_0.
4. If G is a reduced pointer $RP = (D, \overline{p}_{id}, s_0, a_0, p'_0)$, subpointer $SP = (D, \overline{p}_{id}, s_0, a_0, s_1, p_1)$ referencing the new subsegment is generated by using relation $p_1 = f_{s_1}(p'_0)$. If G is a simple pointer $P = (D, \overline{p}_{id}, s_0, p_0)$, then we have $a_0 = [ndrw]$, and $p_1 = f_{s_1}(f_{a_0}(p_0))$. In both cases, SP is returned to the caller.

In step 3, a simple strategy for the generation of the subsegment identifiers is a sequential generation, supported by a subsegment counter for each existing segment.

The $RSP \leftarrow reduceSubpointer(SP, a_1)$ primitive returns a reduced subpointer derived from subpointer SP by using access privilege specifier a_1. Let $SP = (D, \overline{p}_{id}, s_0, a_0, s_1, p_1)$ and $RSP = (D, \overline{p}_{id}, s_0, a_0, s_1, a_1, p'_1)$. Execution of this primitive uses generation function f to evaluate quantity $p'_1 = f_{a_1}(p_1)$.

Protection primitives *newSegment* and *newSubsegment* need to access the protection table, and can only be used to allocate memory locally, in the current node. This is not the case for primitives *reduceSimplePointer* and *reduceSubpointer*. In fact, a subject is always in the position to carry out a p-pointer reduction autonomously. No assistance is needed of the node where the referenced segment is stored, and the p-pointer transformation generates no network traffic. This important result has been obtained by taking advantage of generation function f, which is universally known.

3.3.3 Deleting segments

The $deleteSegment(G)$ primitive allows a subject running in node D to delete the segment s_0 referenced by p-pointer G in D, and all the subsegments of this segment. Execution accesses segment table ST_D to eliminate the table entry reserved for s_0. The subsegment

table associated with s_0 is deleted. Execution terminates successfully only if G is valid, and is a simple pointer, or a reduced pointer including access right *delete*.

Similarly, the *deleteSubsegment*(G) primitive makes it possible to delete the subsegment s_1 of segment s_0, which is referenced by p-pointer G in D. Execution accesses the subsegment table of s_0 to eliminate the table entry reserved for s_1. Execution terminates successfully only if G is valid, and is a subpointer, or a reduced subpointer including access right *delete*.

When a new segment is allocated, or an existing segment is deleted, the contents of the corresponding memory area are not modified. This means, for instance, that if two or more segments are defined for the same memory area, and we delete one of them, the other segments are not affected by the deletion. Segment creation and deletion are restricted to the current node; the protection primitives do not allow us to create or delete segments in the shared memory of a remote node. Thus, memory management activities are confined within the node boundaries. Remote memory accesses are only permitted to read the contents of a remote segment, or to overwrite these contents.

3.3.4 Accessing segments

Every given segment can be accessed, to read or to write, only by presenting a p-pointer specifying the corresponding access right, *read* or *write*. To this aim, the protection system includes two *communication primitives*, called *readSegment* and *writeSegment*. If used to access a segment in a remote node, both these primitives cause the exchange of messages with that node. A message can be a *request message*, a *reply message*, or a *data message*. A request message specifies actions to be accomplished in the remote node, a reply message is used to return the results of these actions, a data message is used to transmit the contents of a segment.

In the rest of this section, we shall describe the actions involved in the execution of the communication primitives. We shall refer to the case of an access to a remote segment. The activities resulting from an access to a segment in the local shared memory can be easily imagined, and will not be described in detail.

The *readSegment*($G, addr$) communication primitive copies the contents of the segment or subsegment s referenced by p-pointer G into an area starting at address *addr* of the private memory of the current node D. Let R denote the remote node where s is stored. Execution is as follows:

1. Node D validates p-pointer G; it should specify access right *read*. If this is not the case, execution fails.
2. Node D sends a request message to node R. On receipt of this message, R accesses segment table ST_R or, if s is a subsegment, the subsegment table of the segment

including s, as specified by p-pointer G, to find the table entry reserved for s and extract the base b and the limit t of s. If no such entry exists, s has been deleted; a negative reply message is sent to D, and execution of *readSegment* fails. Otherwise,

3. Node R uses quantities b and t to assemble a data message d including t and the contents of s. This data message is returned to D.
4. Node D copies the contents of s from data message d into a local private memory area of size t, which starts at address *addr*.

The $writeSegment(G, addr)$ communication primitive copies the contents of an area starting at address *addr* of the private memory of the current node D into the segment or subsegment s referenced by p-pointer G. Let R denote the remote node where s is stored. Execution is as follows:

1. Node D validates p-pointer G; it should specify access right *write*. If this is not the case, execution fails.
2. Node D sends a request message to node R. On receipt of this message, R accesses segment table ST_R or, if s is a subsegment, the subsegment table of the segment including s, as specified by p-pointer G, to find the table entry reserved for s and extract the limit t of s. If no such entry exists, s has been deleted; a negative reply message is sent to D, and execution of *writeSegment* fails. Otherwise,
3. Node R assembles a reply message including quantity t. This message is returned to D.
4. Node D assembles a data message d including the contents of an area of size t, which starts at address *addr* of the local private memory. This data message is sent to R.
5. Node R copies the contents of data message d into s.

4 EXAMPLES OF APPLICATIONS

This section presents a few examples of practical applications of p-pointers to the solution of a variety of protection problems. In the first example, segments are used to form containers aimed storing of both p-pointers and ordinary information items. Then, we consider the implementation of hierarchical organizations of security classes. Finally, we take advantage of subsegments to support a protection paradigm based on access control lists. These examples are by no means exhaustive; they are only aimed at giving an indication of the flexibility of the p-pointer concept.

4.1 Containers

A *container* is a segment partitioned into two subsegments, which we shall call the p-pointer subsegment (*p-subsegment*, for short) and the data subsegment (*d-subsegment*).

The p-subsegment is aimed at storing p-pointers, the d-subsegment contains ordinary information items. The p-pointers in the p-subsegment may reference other containers, which can even be stored remotely, in different nodes. In this way, containers can be organized into arbitrary structures, according to the specific requirements of the intended application. An example is given below.

We wish to point out that possession of a p-pointer for a given container may grant access privileges stronger than that included in the p-pointer itself. For instance, let us consider a container C whose p-subsegment contains simple pointers. A subject S that owns a reduced subpointer RSP referencing the p-subsegment of C with access right *read* can acquire these simple pointers to access the segments they reference, both to read and to write. In contrast, if the access right in RSP is *write*, S can access the p-subsegment to overwrite the existing p-pointers, and to delete these p-pointers. However, S is not allowed to read these p-pointers and take advantage of them to access the corresponding segments.

4.2 Hierarchical classes

Let us consider a hierarchical tree structure defined in terms of security classes. Each class can have only one parent, and many children. Each subject is assigned to a class. A subject in a given class can access this class, and all the classes that descend from this class, hierarchically. Thus, a subject in the class that is the root of the hierarchy can access all the classes, and a subject in a leaf class, at the lowest hierarchical level, can access only this leaf class.

A hierarchical structure of this type can be implemented by reserving a container for each class. The p-subsegment of the container associated with a given class stores reduced pointers with access right *read*, referencing the containers associated with the children of that class. The d-subsegment will store the information items relevant to the class. No container is reserved for a leaf class, at the lowest hierarchical level, whereas the container for the class which is the parent of one or more leaf classes will include a d-subsegment for each of these leaf classes.

In this implementation, a subject S in a given class C owns a reduced pointer for the container associated with C, with access right *read*. This reduced pointer allows S to access the d-subsegment in this container, to read the information items for C. Furthermore, S is in the position to access the p-subsegment, to acquire the p-pointers it contains. S can use these p-pointers to access the containers reserved for the direct and indirect children of C, recursively. If S is in a class at the penultimate hierarchical level, it can use the reduced pointer for the container of this class to access the d-subsegment for this class, and also the d-subsegments for the children of this class. Finally, if S is in a leaf class, at the lowest hierarchical level, it owns a reduced subpointer, with access right *read*, for the

d-subsegment reserved for this class in the container of the parent class.

Thus, a subject in a given class holds a *single p-pointer*. This approach has significant advantages over the alternative, *multiple p-pointer* approach, whereby each class corresponds to a data segment. In this case, a subject in a given class possesses a simple pointer for the data segment associated with this class, and a simple pointer for the data segment of each descendant class. The multiple p-pointer approach penalizes the subjects in the most privileged classes, which have to handle more p-pointers [8]. Significant complications follow, for instance, in a dynamic access control, i.e. the ability to add new classes to the class hierarchy, and to eliminate existing classes from the hierarchy [6]. For instance, the addition of a new class implies a new p-pointer distribution, which involves all the subjects in the ancestor classes, whereas, in the single p-pointer approach, we simply add a new p-pointer to the p-subsegment in the container of the parent class.

4.3 Access control lists

The access matrix, introduced in Section 1, can be represented in memory by columns. To this aim, in a classical implementation, an *access control list* ACL_j is associated with each given object B_j [24], [32]. ACL_j consists of pairs (S_i, ar), where ar specifies the set of access rights owned by subject S_i for B_j.

In our p-pointer system, ACL_j can be simply implemented by a data segment d_j. This data segment is partitioned into subsegments, a subsegment for each subject. The i-th subsegment, corresponding to subject S_i, consists of a single memory cell, which encodes the access rights owned by S_i for B_j. If the k-th bit of this cell is asserted, then S_i owns the k-th access right, ar_k.

In this approach, subject S_i holds a reduced subpointer for the i-th subsegment of d_j, with access right *read*. The internal representation of object B_j includes a simple pointer for d_j. This simple pointer is used to manage the access rights of each subject, by modifying the corresponding subsegment, to add new access rights, or to eliminate the existing access rights. When S_i attempts to access B_j to execute a given operation, it presents the subpointer it owns for d_j. The operation will use this subpointer to read the contents of the memory cell that forms the corresponding subsegment to check whether the bits encoding the required access rights are asserted. If this is not the case, the operation terminates with failure.

5 DISCUSSION

As seen in Section 1, in a classical password-based solution of the memory protection problem, one or more passwords are associated with each object, and each password corresponds to an access privilege. In contrast, in our system, the primary passwords

are intended to be associated with subjects. In a possible organization, when a node is initialized, a *root subject* S_R is created. This subject receives a root pointer P_R referencing the root segment s_R of that node. When a new subject S is created in the same node, e.g. a new process, the root subject takes advantage of the *newPrimaryPassword* protection primitive to generate one or more primary passwords for S. Then, the primary passwords and the *newSegment* primitive are used to create the segments that are necessary for S, for instance, to communicate with the other subjects of the system across the network. S will distribute p-pointers for these segments to these remote subjects. S will preventively reduce these p-pointers to eliminate the unnecessary access rights, e.g. only access right *read* is required by a remote subject using a given segment to receive data. Alternatively, subject S can create subsegments, and then transmit subpointers for these subsegments.

5.1 Access privilege revocation

As seen in Section 1.3, in a password-based protection system, a simple method to revoke an access privilege for a given object is to replace the password associated with this access privilege. An action of this type affects all the subjects that own this password, independently of the nodes where these subjects are running. In the access matrix model, a revocation approach of this type is *by columns*: the revocation involves all the matrix elements in the column corresponding to the object with which the revoked password is associated.

In contrast, in our protection system based on p-pointers, an access right revocation corresponds to the replacement of the value of a primary password with a new value. An action of this type revokes all the p-pointers defined in terms of the old value. If we associate primary passwords with subjects, in the access matrix model this revocation approach is *by rows*: the effect of a revocation is to eliminate access privileges from the elements of the matrix in the row corresponding to the subject with which the revoked password is associated. The elements involved are those of the objects referenced by the p-pointers expressed in terms of that primary password. As seen in Section 1.3, revocation by rows is especially interesting in a distributed system, to limit its effects to a specific node. Consider a subject running in a given node, and aimed at communicating with a few other nodes. For this subject, we associate a primary password with each of these nodes. If the value of a primary password is changed, the revocation is restricted to the corresponding node.

In a different approach, the access privileges for a given memory area are revoked by deleting a segment defined in terms of that area. As seen in Section 1.4, we can allocate overlapping segments corresponding to the same memory area, and deletion of one of these segments has no effect on the others. This means that the revocation is limited to a subset

of all the subjects that hold access privileges for that memory area (*independent* revocation [11]). A subsequent creation of a new segment in the same memory area has no effect on the p-pointers referencing the deleted segment; these p-pointers will not be renewed. This is a consequence of the mechanism for password creation, based on generation function f. As seen in Section 2.1, this mechanism considers the segment name rather than its base and limit. Segment names are never reused, so the p-pointer for the new segment will have a different local password.

Several mechanisms for access privilege revocation have been proposed in the past, with special reference to capability systems ([20]; see also subsequent Section 5.5). Examples are a propagation graph associated with each access privilege, which records the propagation of this access privilege throughout the system [11], [12]; a centralized reference monitor that, for each given object, keeps track of all the subjects that hold access privileges for that object [35]; and temporary access privileges, whose validity must be renewed periodically to avoid implicit revocation [19]. In a distributed system, these mechanisms are prone to significant network traffic: messages must be exchanged between the nodes, to update the propagation graph, to interact with the centralized monitor, or to renew the access privileges. In contrast, in our system, the activities connected with access privilege revocation are confined within the boundaries of a single network node. This is true for revocations based on the deletion of a primary password, as well as for revocations based on the deletion of a segment.

5.2 Network costs

As seen in Section 1.2, in traditional password systems, a password reduction implies the intervention of a password manager, which receives the original password and returns the reduced password. An action of this type is prone to generate network traffic. For instance, consider a subject that holds a password for an object stored in a remote node. Messages will be exchanged with the remote node, to send the original password and receive the reduced password. In contrast, in our system, a subject running in a given node and owning a p-pointer for a remote object is in the position to reduce the p-pointer autonomously. In fact, execution of the *reduceSimplePointer* and the *reduceSubpointer* protection primitives generates no network traffic. We have obtained this important result by taking advantage of the generation function, which is universally known. The generation function can be used in a given node to reduce a p-pointer, independently of the node storing the segment referenced by this p-pointer.

In fact, the actions involved in the execution of each protection primitive are confined within the boundaries of the node where the call to this primitive has been issued; these actions generate no network traffic. The only exceptions are the communication primitives

Table 4: Memory requirements for p-pointer storage (in bits).

	simple pointer	reduced pointer	subpointer	reduced subpointer
Format specifier	2	2	2	2
Node name D	10	10	10	10
Primary password identifier $\overline{p}_{id}$	16	16	16	16
Segment identifier s_0	28	28	28	28
Access privilege specifier a_0	–	4	4	4
Subsegment identifier s_1	–	–	32	32
Access privilege specifier a_1	–	–	–	4
Local password $\overline{p}$	128	128	128	128

readSegment and *writeSegment*, which produce message exchanges if they are used to access a remote segment.

It is worth to note that the primary passwords of a given node are confined within the boundaries of that node. These passwords are never transmitted across the network; instead, they are only used in the creation and the deletion of local segments.

5.3 Memory requirements and execution times

In a p-pointer, a 10-bit node name D supports a large network of up to 1024 nodes. If we associate primary passwords with subjects, 16-bit primary password identifiers are suitable for a large number of subjects, and can support repeated actions of access privilege revocation obtained by primary password deletion (see Section 5.1). 28-bit segment identifers permit iterated actions of segment creation and deletion, as is the case if access privileges are revoked at segment level. Four-bits are required in the access privilege specifier to encode the four access rights. Finally, the local password size is a function of the overall security requirements, e.g. 128 bits.

As shown in Table 4, the resulting p-pointer size is in the range from the 182 bits of a simple pointer to the 222 bits of a reduced subpointer. If a single, 28-byte size is used for all p-pointers, a two-bit format specifier will select the actual p-pointer format.

We can now compare these results with the memory requirements for password storage in a traditional, password-based protection system. Here, each password is associated with the identifier of the corresponding segment, i.e. a node name and a local segment identifier. For 10-bit node names, 28-bit local segment identifiers, and 128-bit passwords, we have a total memory requirement of 166 bits. The size increase we pay in our system for p-pointer storage is compensated by the necessity to store less passwords. In fact, in a traditional password system we have several passwords for each object, one password for each access privilege defined for that object. In contrast, in our system, only the primary passwords need to be stored in each node. The local passwords, corresponding to a specific object and a specific access permission, are generated dynamically, starting from the primary

passwords, taking advantage of the generation function.

As for execution times, the number of applications of the generation function required to validate a given p-pointer varies, according to the p-pointer type, from the single application that is sufficient for a simple pointer, up to the four applications that are necessary for a reduced subpointer. In a given node, let us now consider a subject aimed at distributing an access privilege for an area in the shared memory of that node. If the subject holds a simple pointer for a segment that includes this memory area, a solution is to use the *newSubsegment* protection primitive to generate a subsegment, and a subpointer for this subsegment. If the subject is a root subject, an alternative is to reserve a segment for the area; the validity of a simple pointer for this segment can be verified efficiently.

5.4 Forging p-pointers

Let us now consider a malevolent subject that holds a reduced pointer for a given memory segment, and is aimed at amplifying the access rights in this reduced pointer, e.g. by forging a simple pointer for the same segment. The subject can take advantage of node name D, primary password $\overline{p}_{id}$, and segment identifier s_0 in the reduced pointer to include them into the simple pointer. The next step is to transform local password p'_0 in the reduced pointer into local password p_0 in the simple pointer. In fact, p_0 *precedes* p'_0 in the password conversion procedure, illustrated in Section 2.1, which starts from a primary password to generate the password corresponding to the given p-pointer type (see Figure 1). This procedure takes advantage of generation function f, which is one-way. It follows that is computationally infeasible to invert f to evaluate p_0 starting from p'_0. An alternative is to use a password chosen at random. If passwords are large and sparse, the probability of a casual match is virtually null, and the simple pointer forging attempt is destined to fail.

Similar considerations can be made for the transformation of a subpointer into the corresponding simple pointer. In this case, too, quantities D, $\overline{p}_{id}$, and s_0 can be extracted from the subpointer, but it will be computationally infeasible to evaluate local password p_0 in the simple pointer starting from local password p_1 in the subpointer.

5.5 Related work

5.5.1 Capabilities

In a classical approach, the access privilege held by a subject for a given object is expressed in terms of a *capability* [20]. This is pair (B, ar), where ar is a set of access rights for object B. In this approach, an important problem is capability segregation: we should prevent a subject that holds a given capability from modifying this capability, for instance, to add new access rights, or to change the object identifier to forge a capability for a different object.

Solutions to the capability segregation problem have been conceived, and actually implemented in existing systems [9]. In a segmented memory environment, special segments, which we shall call *capability segments*, can be reserved for capability storage [16]. In this approach, the instruction set of the processor is augmented with a set of special instructions, the *capability instructions*, aimed at capability processing. An access to a capability segment terminates successfully only if it uses a capability instruction. This approach is prone to segment proliferation, and is an undue complication to object representation. Let us consider a simple data object consisting of two data segments, for instance. A capability segment will be necessary to store the capabilities for the data segments.

A different approach takes advantage of a tagged memory [3], [14], [39]. A 1-bit tag associated with each memory cell specifies whether this cell contains a capability, or an ordinary information item. A cell tagged to contain a capability can be accessed only by using the capability instructions. If an ordinary instruction is used, execution fails. This approach requires *ad hoc* memory devices aimed at containing the cell tags; this is contrary to hardware standardization. Undue complications may follow in the management of the large storage units, for instance, in the swap operations connected with memory virtualization, owing to the necessity to save, and then to restore, the tags.

PSOS [29] is an example of a capability-based operating system using tags for capability segregation. In PSOS, the processor includes two capability operations, to create a new capability and to restrict the access rights in a given capability. The tagging system prevents any other processor operation to be used successfully to alterate an existing capability. Tags are preserved throughout the system, within the processor as well as in the primary and the secondary memories.

The CHERI capability system [38], [39] extends the 64-bit MIPS IV architecture to include a capability coprocessor. The coprocessor interacts with the processor pipeline by receiving instructions, exchanging operands and sending exceptions. A capability is partitioned into a base field and a limit field that describe a memory segment, and an access right field that specifies access rights for this segment. Capabilities and ordinary data items can safely coexist in the same data structure owing to a form of tagged memory protection. A tag bit is associated with each 256-bit memory location. If asserted, the tag bit specifies that the corresponding location contains a capability. Any non-capability store clears the tag. The coprocessor includes a set of special registers, called *capability registers*. A capability must be preventively loaded from memory into a capability register to access the corresponding memory segment. To this aim, an *ad hoc* capability instruction is provided.

5.5.2 Password capabilities

Passwords are a significant alternative to capabilities, which does not suffer from the segregation problem. As seen in Section 1, in a password system, a set of passwords is associated with each object, one password for each access privilege defined for that object. If passwords are large, sparse and chosen at random, the probability that a malevolent subject guesses a valid password to obtain illegitimate access privileges is vanishingly low. A further requirement is that there should be no computable relation between the value of a given password and the access privilege granted by that password, otherwise this relation could be inverted to create new passwords corresponding to amplified access privileges.

Password capabilities are a practical implementation of the password paradigm that received much attention in the past [1], [5], [13], [15]. A password capability is a pair (B, p), where p is a password for object B. A subject that holds a password capability referencing a given object is granted the access privileges for this object that are associated with the password [22].

Walnut [4], [28] is an example of a tightly-coupled multiprocessor using password capabilities for object protection. In Walnut, objects are stored in a virtual address space partitioned into volumes. Each volume has an unique 32-bit identifier, which is permanently associated with a specific fixed or removable storage device. Objects can only exist within the boundaries of a single volume; objects splitted across different volumes are not allowed. Each object is associated with a 32-bit serial number, which is combined with the identifier of a volume to form the unique object identifier. A password capability is a 128-bit value including an object identifier and a 64-bit password. An arbitrary number of capabilities can be associated with the same given object, corresponding to specific access rights and different passwords. The association of passwords with access rights is recorded in a capability table within the boundaries of the protection system. No computable relation exists between a password and the access rights. When an object is created, a master capability is associated with that object. A capability derivation mechanism makes it possible to create new capabilities with restricted access rights. The resulting capability structure takes the form of an inverted tree that describes the interdependencies between the master capability and its derived capabilities. When a capability is destroyed, all its derived capabilities are also destroyed. If we destroy the master capability, the corresponding object is deleted, as it can no longer be referenced.

In the Annex system [12], [30], password capabilities can only reside within the kernel boundaries, to limit undue propagation. Outside the kernel, a password capability can only be referenced by using a *handle*, mapped to that password capability by the kernel. A password capability consists of a 64-bit device name, a 48-bit object name that univocally identifies an object on the target device, a 16-bit capability name that univocally identifies

the capability, and a 256-bit password, assigned at random to prevent forging. Capability revocation is based on a propagation graph, associated with the given capability, and similar to that proposed in [11]. The propagation graph is maintained by the kernel, and records the propagation of the corresponding capability across the devices.

6 CONCLUDING REMARKS

With reference to a distributed environment consisting of nodes connected in an arbitrary network topology, we have proposed the organization of a protection system in which subjects generate access attempts to memory segments. In our approach:

- Segments are the basic unit of information protection and sharing between the nodes. A subject can access a given segment only if it owns an access privilege certified by possession of a p-pointer referencing this segment. Segments can have subsegments.
- One or more primary passwords are associated with each node. Each p-pointer includes a local password, which is valid if it descends from a primary password by application of a universally known, parametric one-way generation function. The p-pointer may also include an optional access privilege specifier, corresponding to less access rights.
- A set of protection primitives forms the subject interface of the protection system. These primitives make it possible to generate new primary passwords, to delete existing primary passwords, and to change their value. Furthermore, they allow subjects to reduce p-pointers to include less access rights, to allocate new segments, to delete the existing segments, and to access the segments to read their contents or to overwrite these contents.

The following is a summary of the main results we have obtained:

- A subject that holds a simple pointer referencing a given segment is in the position to reduce the access privilege specified by that simple pointer autonomously. An action of this type can be completely accomplished locally, and generates no network traffic, even if the segment is stored in a different node. We have obtained this important result by taking advantage of the generation function, which is universally known.
- Taking advantage of null subsegments, a reduced pointer can be reduced further, to specify less access rights.
- A reduced pointer can be transformed into a subpointer referencing a subsegment of the original segment. In this way, a subject that holds an access privilege for a given memory area can distribute an access privilege for a fraction of this area.

- A single primary password is sufficient in each node for all the segments and subsegments allocated in that node. Local passwords within p-pointers are evaluated dynamically, taking advantage of the generation function. This is in sharp contrast with the traditional view of several passwords associated with each protected object, one password for each access privilege defined for that object.
- If passwords are large, sparse and chosen at random, it is impossible for a malevolent subject to forge valid p-pointers. The non-invertibility property of the generation function guarantees that any attempt to amplify a given p-pointer to include more access rights is destined to fail. Similarly, it is impossible to convert a subpointer for a subsegment of a given segment into a simple pointer referencing that segment.
- Two different mechanisms support the review and revocation of access privileges. By replacing the value of a given primary password with a new value, we revoke all the p-pointers defined in terms of the old value. If a primary password is associated with a given subject, an action of this type revokes all the access privileges held by the subject in terms of that primary password. Alternatively, two or more segments can be defined for the same memory area. If we delete one of these segments, we revoke all the access privileges for that memory area, which are expressed in terms of that segment.

REFERENCES

[1] M. Anderson, R. D. Pose, and C. S. Wallace. A password-capability system. *The Computer Journal*, 29(1):1–8, February 1986.

[2] S. Bakhtiari, R. Safavi-Naini, and J. Pieprzyk. Cryptographic hash functions: a survey. Technical report, Centre for Computer Security Research, Department of Computer Science, University of Wollongong, Australia, 1995.

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

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

[5] J. S. Chase, H. M. Levy, E. D. Lazowska, and M. Baker-Harvey. Lightweight shared objects in a 64-bit operating system. *ACM SIGPLAN Notices*, 27(10):397–413, October 1992.

[6] T.-S. Chen and J.-Y. Huang. A novel key management scheme for dynamic access control in a user hierarchy. *Applied Mathematics and Computation*, 162(1):339–351, 2005.

[7] S. De Capitani di Vimercati, S. Paraboschi, and P. Samarati. Access control: principles and solutions. *Software – Practice and Experience*, 33(5):397–421, 2003.

[8] A. De Santis, A. L. Ferrara, and B. Masucci. Cryptographic key assignment schemes for any access control policy. *Information Processing Letters*, 92(4):199–205, 2004.

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

[10] G. Dini and L. Lopriore. Distributed storage protection in wireless sensor networks. *Journal of Systems Architecture*, 61(5–6):256–266, May–June 2015.

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

[12] D. A. Grove, T. C. Murray, C. A. Owen, C. J. North, J. A. Jones, M. R. Beaumont, and B. D. Hopkin. An overview of the Annex system. In *Proceedings of the Twenty-Third Annual Computer Security Applications Conference*, pages 341–352, Miami Beach, Florida, USA, December 2007. IEEE.

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

[14] M. E. Houdek, F. G. Soltis, and R. L. Hoffman. IBM System/38 support for capability-based addressing. In *Proceedings of the 8th Annual Symposium on Computer Architecture*, pages 341–348, Minneapolis, Minnesota, USA, May 1981. IEEE Computer Society Press.

[15] J. King-Lacroix and A. Martin. BottleCap: a credential manager for capability systems. In *Proceedings of the Seventh ACM Workshop on Scalable Trusted Computing*, pages 45–54, Raleigh, NC, USA, October 2012. ACM.

[16] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al. seL4: formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, pages 207–220, Big Sky, MT, USA, October 2009. ACM.

[17] I. Kuz, G. Klein, C. Lewis, and A. Walker. capDL: a language for describing capability-based systems. In *Proceedings of the First ACM Asia-Pacific Workshop on Systems*, pages 31–36, New Delhi, India, August 2010. ACM.

[18] L. Lamport. Password authentication with insecure communication. *Communications of the ACM*, 24(11):770–772, November 1981.

[19] A. W. Leung, E. L. Miller, and S. Jones. Scalable security for petascale parallel file systems. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, pages 1–12, Reno, NV, USA, November 2007. IEEE.

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

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

[22] L. Lopriore. Password capabilities revisited. *The Computer Journal*, 58(4):782–791, April 2015.

[23] L. Lopriore. Password management: distribution, review and revocation. *The Computer Journal*, 58(10):2557–2566, October 2015.

[24] L. Lopriore. Access control lists in password capability environments. *Computers & Security*, 62:317–327, September 2016.

[25] R. C. Merkle. One way hash functions and DES. In *Proceedings of the 9th Annual International Cryptology Conference – Advances in Cryptology*, pages 428–446, Santa Barbara, California, USA, August 1989. Springer.

[26] M. S. Miller and J. S. Shapiro. Paradigm regained: abstraction mechanisms for access control. In *Proceedings of the 8th Asian Computing Science Conference*, pages 224–242, Mumbai, India, December 2003. Springer.

[27] M. S. Miller, K.-P. Yee, and J. Shapiro. Capability myths demolished. Technical report, Systems Research Laboratory, Johns Hopkins University. http://srl.cs.jhu.edu/pubs/SRL2003-02.pdf, 2003.

[28] D. Mossop and R. Pose. Information leakage and capability forgery in a capability-based operating system kernel. In *Proceedings of the OTM Confederated International Conferences "On the Move to Meaningful Internet Systems"*, pages 517–526, Montpellier, France, October 2006. Springer.

[29] P. G. Neumann and R. J. Feiertag. PSOS revisited. In *Proceedings of the 19th Annual Computer Security Applications Conference*, pages 208–216, Las Vegas, NV, USA, December 2003. IEEE.

[30] T. Newby, D. A. Grove, A. P. Murray, C. A. Owen, J. McCarthy, and C. J. North. Annex: a middleware for constructing high-assurance software systems. In *Proceedings of the 13th Australasian Information Security Conference*, pages 25–34, Sydney, Australia, January 2015. ACS.

[31] B. Preneel, R. Govaerts, and J. Vandewalle. Hash functions based on block ciphers: a synthetic approach. In *Proceedings of the 13th Annual International Cryptology Conference*, pages 368–378, Santa Barbara, California, USA, August 1993. Springer.

[32] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[33] P. Samarati and S. De Capitani Di Vimercati. Access control: policies, models, and mechanisms. In R. Focardi and R. Gorrieri, editors, *Foundations of Security Analysis and Design*, pages 137–196. Springer, Berlin, Heidelberg, 2001.

[34] R. S. Sandhu. Cryptographic implementation of a tree hierarchy for access control. *Information Processing Letters*, 27(2):95–98, 1988.

[35] J. S. Shapiro and S. Weber. Verifying the EROS confinement mechanism. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pages 166–176, Berkeley, California, USA, May 2000. IEEE.

[36] W. Tolone, G.-J. Ahn, T. Pai, and S.-P. Hong. Access control in collaborative systems. *ACM Computing Surveys*, 37(1):29–41, 2005.

[37] W. Trappe, J. Song, R. Poovendran, and K. J. Liu. Key management and distribution for secure multimedia multicast. *IEEE Transactions on Multimedia*, 5(4):544–557, 2003.

[38] R. N. Watson, R. M. Norton, J. Woodruff, S. W. Moore, P. G. Neumann, J. Anderson, D. Chisnall, B. Davis, B. Laurie, M. Roe, et al. Fast protection-domain crossing in the CHERI capability-system architecture. *IEEE Micro*, 36(5):38–49, 2016.

[39] R. N. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, et al. CHERI: a hybrid capability-system architecture for scalable software compartmentalization. In *Proceedings of the 36th IEEE Symposium on Security and Privacy*, San Jose, California, USA, May 2015. IEEE.

[40] X. Zhang, Y. Li, and D. Nalla. An attribute-based access matrix model. In *Proceedings of the 2005 ACM Symposium on Applied Computing*, pages 359–363, Santa Fe, New Mexico, USA, March 2005. ACM.