Password-based protection of clustered segments in distributed memory systems

Lanfranco Lopriore

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

Abstract — With reference to a distributed system consisting of nodes connected by a local area network, we consider the problems related to the distribution, verification, review and revocation of access permissions. We propose the organization of a protection system that takes advantage of a form of protected pointer, the handle, to reference clusters of segments allocated in the same node. A handle is expressed in terms of a selector and a password. The selector specifies the segments, the password specifies an access right, read or write. Two primary passwords are associated with each cluster, corresponding to an access permission for all the segments in that cluster. A handle weakening algorithm takes advantage of a parametric one-way function to generate secondary passwords corresponding to less segments. A small set of protection primitives makes it possible to allocate and delete segments in active clusters, and to use handles to access remote segments both to read and to write. The resulting protection environment is evaluated from a number of viewpoints, which include handle forging, review and revocation, the memory costs for handle storage, the execution times for handle validation and the network traffic generated by the execution of the protection primitives. An indication of the flexibility of the handle concept is given by applying handles to the solution of a variety of protection problems.

Keywords: access right; distributed system; parametric one-way function; protection; revocation; segment.

1 INTRODUCTION

We refer to a distributed architecture consisting of nodes connected by a local area network. The network topology is inessential. We hypothesize that, in each node, the primary memory is partitioned into two regions, *private* and *shared*. The private memory can only be accessed by software components being executed in that node. The shared memory, which is reserved to interprocess communication, can also be accessed by software components running in the other nodes, albeit in a strictly controlled fashion. We shall not consider the mechanisms for the control of private memory accesses; instead, we shall concentrate on shared memory protection, with special reference to the problems inherent in the distribution, verification, review and revocation of access permissions. These are major problems in the design of any protection system. Our solution complies with a segmented view of the shared memory.

A *segment* is a contiguous memory area completely defined by a *base* and a *length*. The

base is the address of the first storage unit reserved for the segment, the length expresses the segment size. Segments are the elementary unit of information transmission and sharing between the nodes. Two operations are possible on a segment, to read the segment contents and to replace these contents. Protection applies to *local* segments allocated in the shared memory of same node as the software component attempting the access, as well as to *remote* segments allocated in the shared memory of the other nodes.

In a classical protection paradigm, active entities, called *subjects*, generate access attempts to protected, passive entities, called *objects* [18], [24], [35]. The system associates a set of *access rights* with each object. A subject aimed at accessing a given object to execute one of the operations defined for that object must possess an access right permitting successful accomplishment of that operation; if this is not the case, the access attempt generates a protection violation, and fails. A *protection domain* is a set of access rights for correlated objects. A subject being executed in a given protection domain can access the objects, taking advantage of the access rights included in that domain.

We shall hypothesize that a subject can be a scheduled computation (a process), or, in an event driven environment, a software routine activated by a hardware interrupt [22]. Segments are the objects on which protection is exercised. Two access rights are defined for segments, *read* and *write*. A subject that holds access right *read* for a given segment is allowed copy the segment contents from the shared memory of the node where the segment is allocated into the private memory of the node where that subject is running. Similarly, a subject that holds access right *write* for a given segment can overwrite the segment contents with quantities taken from its own private memory.

We consider segments grouped in clusters. A *cluster* is a collection of correlated segments, all contained in the shared memory of the same network node. A subject that holds access right *read* for a given cluster can access the segments of that cluster to read their contents; this is similar to access right *write* for segment write accesses.

1.1 Capabilities

A major problem in the design of a protection system is how to represent the access rights held by each subject. A classical solution is based on the concept of a *capability* [15]. This is a pair (G, AR), where G is an object identifier, and AR is a set of access rights. A subject that holds capability (G, AR) can access object G to carry out the actions permitted by the access rights in AR.

Several aspects of a practical implementation of the capability concept deserve in-depth consideration for their impact on performance and usability. These include capability segregation, weakening, review and revocation, and the memory requirements for capability storage.

1.1.1 Segregation

Subjects must be prevented from modifying capabilities, for instance, to add access rights to an existing capability, or even to change the object identifier to forge a capability for a different object. Several solutions to this capability segregation problem have been proposed [5], [16], [32]. In a segmented memory system, special segments, which we shall call *capability segments*, can be reserved for capability storage (in contrast, the *data segments* contain ordinary information items) [6], [12]. A *capability list* is a collection of capabilities for correlated objects; a capability segment contains a capability list. The instruction set of the processor will be enlarged by the addition of special *capability instructions* for capability processing. Capability segments can only be accessed by using the capability instructions; if an ordinary data instruction is used, an exception of violated protection is raised.

In an alternative approach, a 1-bit *tag* is associated with each memory cell, which specifies whether this cell contains a capability or an ordinary information item [1], [11], [31]. A cell tagged to contain a capability can only be accessed by using the capability instructions. This approach requires memory banks specialized to contain the cell tags, and is contrary to the requisite of hardware standardization [20].

1.1.2 Weakening

A subject that holds a capability for a given object can transfer a copy of that capability to another subject. In this way, the recipient acquires the whole access privilege specified by that capability. In fact, a capability copy is indistinguishable from the original, and possession of the copy is equivalent to possession of the original. On the other hand, it may well be the case that a subject wishes to transfer only part of the access rights included in the original capability. In a classical capability environment, this means that the instruction set of the processor should include a capability instruction to modify the access right field in a strictly controlled fashion, excluding access right amplification. In a common approach, the access right field features one bit for each access right; if asserted, the given bit denotes the presence of the corresponding access right. The access right weakening instruction will permit to clear (but not to set) these bits.

1.1.3 Review and revocation

A subject that receives an access privilege in the form of a capability is in the position to transmit it further. It follows that capabilities tend to disperse throughout the system, and it is hard to keep track of all existing copies of the original capability. In a distributed system, this problem is exacerbated by the possibility that copies spread to different nodes. A relevant problem is access right revocation: a subject that created a given object

should be in the position to withdraw the access privileges distributed for this object [2], [8], [9]. An essential property is that the effects of a revocation should propagate to all the subjects that hold the access privilege being revoked (*transitive* revocation). In a distributed system, this means that revocation should extend across node boundaries. Other desirable properties are the abilities to limit revocation to a specific subset of the access rights (*partial* revocation), to revoke different access privileges for the same memory area independently of each other (*independent* revocation), and to restore the original privileges through the same mechanism as for revocation (*temporal* revocation).

Several solutions to the access right revocation problem have been devised [18]. Examples are a propagation graph associated with each capability, which keeps track of all successive transferrals of this capability between subjects [8]; temporary capabilities with short lifetimes, which must be renewed periodically to avoid implicit revocation [14]; and a centralized reference monitor associated with each object, which keeps track of the subjects that hold access permission for this object [28]. These solutions tend to impair a basic advantage of capability protection, i.e. simplicity in access right transmission between subjects.

1.1.4 Memory requirements

A subject that is granted access privileges for a number of distinct objects has to hold a capability for each of these objects. The resulting memory requirements tend to be high in percentage. This is especially the case if the system should support a large number of small-sized objects [7], [33], if we are aimed at exercising protection at a high level of granularity. Consider, for instance, a capability list that grants access permissions to a group of segments. We have to reserve a capability segment to contain the capability list. This capability segment is a memory waste, and a significant complication in access privilege management.

1.2 Password capabilities

Password capabilities are a remarkable improvement on the capability concept [2], [3], [10], [17]. A password capability is a pair (G, P), where G is an object identifier, and P is a password. A set of passwords is associated with each protected object, and each password corresponds to an access privilege expressed in terms of a set of access rights. If password P matches one of the passwords associated with object G, the password capability grants the access privilege corresponding to the matching password.

1.2.1 Segregation

If passwords are large and sparse, and a malevolent subject forges a password capability by

using a password chosen at random, the probability that this password capability is valid is virtually null [2]. It follows that password capabilities can be mixed in memory with ordinary information items, and can be treated by using standard machine instructions. Thus, password capabilities are an effective solution to the segregation problem.

1.2.2 Weakening

Access privilege weakening is arduous in password capability environments. Consider a subject that holds a password capability defined in terms of a password granting a given access privilege, and suppose that this subject wants to forge a new password capability for an access privilege defined in terms of less access rights. To this aim, the subject will have to ask for intervention of a password capability manager, associated with the object, and responsible for password capability weakening. The manager receives a password capability, and returns a new password capability defined in terms of the weaker password. This is indeed possible only if one such password exists. This means that more passwords should be stored in memory for each object, as part of the object internal representation. In a distributed system, network traffic is generated by the necessity to communicate with the password manager, if it is stored in a remote node.

1.2.3 Review and revocation

Of course, in a password capability environment, by replacing one of the passwords associated with a given object we implement a form of transitive revocation that can be reversed at little effort by restoring the original password. However, this solution does not meet the requirement of partiality, as it cannot be limited to a subset of the access rights.

1.2.4 Memory requirements

The memory requirement problem is exacerbated in password capability environments by the necessity to store the passwords for each given object within the internal representation of that object. This will be especially the case if several combinations of access rights should be supported to specify distinct access privileges; a separate password will be necessary for each combination.

1.3 Handles

In this paper, we present a comprehensive solution to the problems, outlined above. We propose the *handle* as an extension of the password capability concept. A handle includes the name of a segment cluster, a password, and a segment selector that specifies a subset of, or all, the segments in the cluster. When a new cluster is activated in a given node, two passwords are generated in that node for the cluster. These passwords are called the

read primary password and the *write primary password*, and correspond to access rights *read* and *write*, respectively. Possession of a handle expressed in terms of the read primary password (the *read primary handle*) makes it possible to access all the segments in the cluster to read their contents; this is similar to the *write primary handle* for write accesses.

A subject that holds a handle for a given cluster can transmit a copy of this handle to another subject. An action of this type grants the recipient permission to access all the segments specified by the handle. The original subject can preventively weaken the handle copy to reference less segments, thereby reducing the access privilege of the recipient. The handle weakening algorithm is an application of parametric one-way functions. A *parametric one-way function* is a function $f_c(x)$ where, given a value y and a parameter c, it is computationally infeasible to find a value x such that $y = f_c(x)$ [30]. Thus, a parametric one-way function is a family of one-way functions [13], one function for each value of the parameter. We can take advantage of a good cryptosystem to reduce the design and implementation efforts, e.g., if E_x is a symmetric cypher, we have $f_c(x) = E_x(c)$ [25].

In our handle-based approach:

- Only two passwords, the primary passwords, are necessary for each cluster, independently of the number of segments that form the cluster.
- A handle corresponds to a list of password capabilities, which grants access right *read* or *write* for one or more segments in a segment cluster. Thus, a handle supports a whole protection domain. With respect to the classical password capability model, which associates passwords with single objects, significant advantages follow from the points of view of memory requirements and simplicity in access right management.
- The handle weakening algorithm does not require access to the primary passwords; it can even be executed remotely, in a different network node.
- A subject that created a given cluster is in the position to revoke the validity of the handles that reference this cluster. Revocation extends to the whole distributed system. The handle revocation mechanism, based on password replacement, is transitive, partial and temporal. Furthermore, two or more segments can be defined for the same memory area. In this case, we can revoke the access permissions by deleting one of these segments. The resulting revocation is transitive, temporal and independent.

The rest of this paper is organized as follows. Section 2 introduces our protection model, with special reference to clusters and handles. Handle generation, weakening, revocation, and reduction (to express handles in a compact form to save memory) are analysed in special depth. Section 3 presents the process interface of the protection system. It consists

of a set of primitives, the *protection primitives*, which make it possible to activate new clusters, to allocate segments in existing clusters, and to use handles to access segments. The actions involved in the execution of each protection primitive will be described with special reference to access privilege checking, and the interactions between the network nodes. Section 4 gives an indication of the flexibility of the handle concept. Handles are used to solve a variety of protection problems, which correspond to different meanings associated with the concept of a segment. Section 5 discusses the proposed protection environment from a number of viewpoints that include handle forging and revocation, the memory costs for handle storage, the execution times for handle validation, the network traffic generated by the execution of the protection primitives, and the relation of our work to previous work. Section 6 gives concluding remarks.

2 THE PROTECTION MODEL

As anticipated in part in Section 1, a cluster is a collection of up to n correlated segments that are allocated in the shared memory of the same given node, where quantity n is protection system specific. Each cluster is assigned a unique identifier C consisting of the name N of the node where the cluster is allocated, and the local name L of the cluster in that node, i.e. $C = (N, L)$. The segments that form cluster C are numbered starting from 0, and are denoted by $c_i, i = 0, 1, \ldots, n-1$.

A handle H is a triple (C, P, S), where C is a cluster name, P is a password, and S is a segment *selector*, aimed at identifying one or more segments in C. If all the bits of S are asserted, and P is a primary password of cluster C, then H is *valid.* In a situation of this type, the handle is a primary handle, and it references all the segments in the cluster. A process can take advantage of H to access each of these segments. Let RP_C and WP_C denote the read primary password and the write primary password of cluster C. If $P = RP_C$, H is the read primary handle, which is denoted by RH_C and can be used to access the segments to read their contents. If $P = WP_C$, H is the write primary handle, which is denoted by WH_C; in this case, the permitted accesses are to write.

A handle for a given cluster can also be expressed in terms of a *secondary password* granting access permission to a subset of the segments in that cluster. In detail, if one or more bits of segment selector S are cleared, then the handle is valid if password P is a secondary password derived from a either RP_C or WP_C by an iterative password conversion procedure, which will be detailed shortly, and is based on the configuration of S.

Specifically, in a given handle $H = (C, P, S)$, segment selector S is partitioned into m subfields, called *subselectors*, and denoted by $s_i, i = 0, 1, \ldots, m-1$, where quantity m is protection system specific. Thus, $S = (s_{m-1}, s_{m-2}, \ldots, s_0)$. The size of each subselector is n

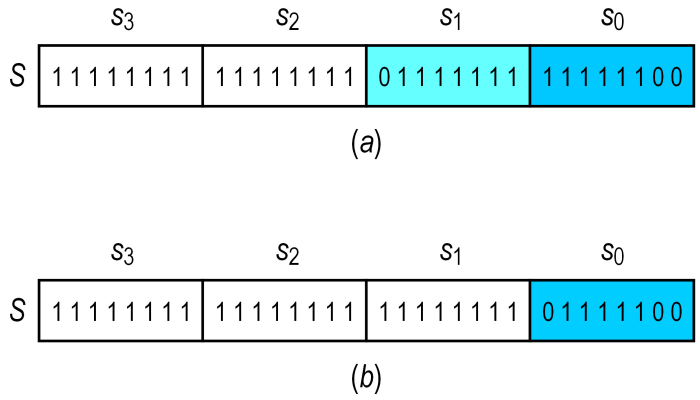

Figure 1: Two different configurations of segment selector S. In both cases, the cluster is supposed to include up to eight segments. The segment selector is partitioned into four subselectors, s_0 to s_3, and the size of each subselector is eight bits.

bits, one bit for each segment in the cluster (the least significant bit, bit 0, corresponds to the first segment, c_0). For each bit that is cleared in the generic subselector, the corresponding segment is eliminated from the handle. This means that the handle references the segments corresponding to the bits that are asserted in quantity $s_0 \wedge s_1 \wedge \ldots \wedge s_{m-1}$.

A subselector whose bits are all asserted is called *flat*. In a given selector, all flat subselectors are always placed in the most significant positions, at the highest order numbers. This means that if subselector s_i is flat, then subselector s_j is flat, $j = i+1, i+2, \ldots, m-1$. Selector S is flat if all its subselectors are flat, i.e. S is all 1's. As anticipated previously, in a situation of this type the handle is valid if the password is a primary password, and in this case the handle references all the segments in the cluster.

Figure 1 shows two different configurations of segment selector S. In these examples, $n = 8$, that is, the cluster consists of up to 8 segments, and $m = 4$, that is, segment selector S is partitioned into four subselectors. In the configuration of Figure 1a, bits 0 and 1 of subselector s_0 are cleared. This means that segments c_0 and c_1 of the cluster are excluded from the handle. Subselector s_1 features a single bit cleared, bit 7. This subselector excludes segment c_7. The other subselectors are flat. We may conclude that the handle including S references segments c_2 to c_6. In the configuration of Figure 1b, subselector s_0 features three bits cleared, bits 0, 1 and 7. The other subselectors are flat. In this case, too, the handle references segments c_2 to c_6. However, as will be made clear shortly, the two cases correspond to different passwords. In fact, in a given cluster we may have different passwords for the same set of segments.

2.1 Password conversion

Of course, for large clusters, the high number of potential secondary passwords corresponds to unacceptable storage requirements. We shall present a mechanism for password con-

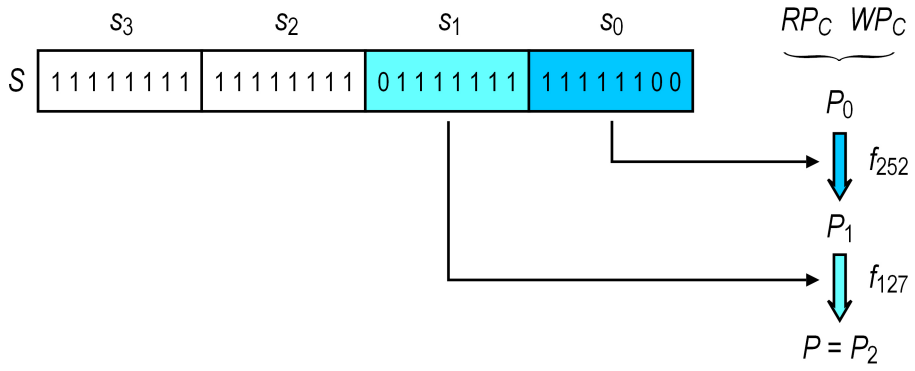

Figure 2: Conversion of a primary password into a secondary password, in a cluster of up to eight segments, for a segment selector partitioned into four subselectors. The password in the resulting handle derives from primary password RP_C, or, if the handle is for write, from primary password WP_C, by iterated application of password conversion function f, taking the values of the subselectors as parameters.

version that allows us to limit the memory requirements for password storage to the two primary passwords. This mechanism takes advantage of a universally-known parametric one-way function, the *password conversion function*. This function is used to convert a primary password into the corresponding secondary password, according to the value of the selector. It has the form $f_s(P)$, where argument P is a password, and parameter s is a subselector.

In detail, let $H = (C, P, S)$ be a handle that references a subset of the segments in cluster C, as is specified by selector S. The handle is valid if P is a secondary password derived from a primary password associated with C by a conversion process using S. If H is valid, a subject that possesses the handle is entitled to access the segments specified by S. If P derives from primary password RP_C, then the access right for these segments is *read*; if the primary password is WP_C, then the access right is *write*.

Conversion of a primary password into a secondary password is an iterative procedure that uses password conversion function f and the subselectors $s_0, s_1, \ldots, s_{m-1}$ of S. We have $P_{i+1} = f_{s_i}(P_i), i = 0, 1, \ldots, k-1$, where k is the index of the first flat subselector s_k, P_0 denotes a primary password (RP_C or WP_C), and $P = P_k$. If no subselector is flat, then $k = m$.

Figure 2 shows the evaluation of password P in handle $H = (C, P, S)$ for a specific configuration of the segment selector S. In this example, cluster C includes up to eight segments ($n = 8$). Selector S is partitioned into four subselectors ($m = 4$), thus we have $S = (s_3, s_2, s_1, s_0)$. Bits 2 to 6 are set in all subselectors; it follows that H references segments c_2 to c_6. In our iterative procedure for evaluation of password P, we have $P_0 = RP_C$, or, if the handle is for write, $P_0 = WP_C$. Subselector s_0 is 11111100 (252 in decimal notation), thus we have $P_1 = f_{252}(P_0)$. Subselector s_1 is 01111111, thus we have

$P_2 = f_{127}(P_1)$. Subselectors s_2 and s_3 are flat; this terminates the procedure, and $P = P_2$.

Let us now consider a subject that holds handle $H = (C, P, S)$ referencing cluster C, with password P and selector S. When the subject presents the handle to the protection system to access a segment of C to read or to write, say segment c_i, the access is validated as follows:

1. Subselectors $s_0, s_1, \ldots, s_{m-1}$ of segment selector S are considered, and quantity $s_0 \wedge s_1 \wedge \ldots \wedge s_{m-1}$ is evaluated. If the i-th bit of this quantity is asserted, then handle H references segment c_i.
2. If the access is to read, the iterative password conversion algorithm, outlined above, is applied to primary password RP_C. If the access is to write, the algorithm is applied to primary password WP_C. If the result is equal to P, then the handle is valid.

The access is validated, and can be accomplished successfully, if the handle is valid and it references segment c_i, i.e. both steps above terminate successfully.

We may conclude that handle $H = (C, P, S)$ grants an access privilege for cluster C. If S is flat (i.e. it is all 1's) and $P = RP_C$, or (if S is not flat) P derives from primary password RP_C by conversion, then the handle grants access right *read* for the segments identified by segment selector S. In a situation of this type, we say that the handle references these segments in read mode. If the primary password is WP_C, the mode is to write.

Two or more handles are *equivalent* if they reference the same set of segments in the same mode (to read or to write), but they specify these segments in terms of different configurations of the segment selector. In a situation of this type, the passwords will be different. In fact, we may have different passwords for the same access privilege.

2.2 Handle weakening

Weakening a handle that references two or more segments in a given clusters means to transform it into a different handle that references a subset of these segments, in the same access mode. For instance, let us consider a subject B_1 that holds handle $H_1 = (C, P_1, S_1)$ referencing the segments in cluster C that are identified by segment selector S_1. The access right granted by H_1 on these segments is that corresponding to the primary password generating P_1. Suppose that subject B_1 is aimed at transferring this access right for a subset of these segments to another subject B_2. In a situation of this type, B_1 weakens handle H_1 into a new handle $H_2 = (C, P_2, S_2)$ for the same cluster C and a subset of the segments, as identified by segment selector S_2. To this aim:

1. Selector S_1 is transformed into selector S_2 by modifying the first (least significant)

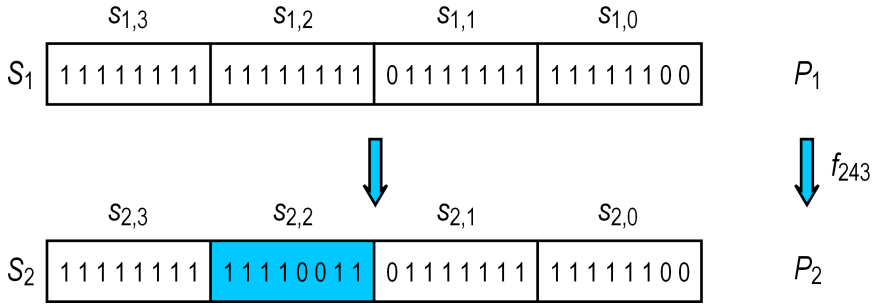

Figure 3: Weakening of handle $H_1 = (C, P_1, S_1)$ referencing a cluster C of up to eight segments. The segment selector S_1 is partitioned into four subselectors. The transformation is aimed at excluding segments c_2 and c_3 from the resulting handle $H_2 = (C, P_2, S_2)$.

subselector of S_1 that is flat, say the k-th subselector $s_{1,k}$, to clear the bits corresponding to the segments referenced by H_1 that should *not* be referenced by H_2. The result is $s_{2,k}$.

2. Password P_1 is transformed into password P_2 by applying password conversion function f. We have $P_2 = f_{s_{2,k}}(P_1)$.

It should be noted that we can take advantage of handle weakening even within the boundaries of a single subject, if this subject is formed by software components supporting different functionalities. An example is a process consisting of several concurrent threads that operate on different subsets of the segments in the same given cluster. In accordance with the *principle of least privilege* [4], [23], [27], each of these threads should be granted the smallest set of access rights that is necessary for that thread to carry out its job. This means that the original handle will be weakened into different handles for the different threads.

Figure 3 shows an example of a handle weakening. In this example, cluster C consists of up to eight segments ($n = 8$), and the original selector S_1 is partitioned into four subselectors ($m = 4$). Bits 2 to 6 are set in all subselectors; it follows that handle $H_1 = (C, P_1, S_1)$ references segments c_2 to c_6. We are aimed at weakening H_1 to exclude segments c_2 and c_3, so that the resulting handle $H_2 = (C, P_2, S_2)$ will reference segments c_4 to c_6. The conversion of S_1 into S_2 clears bits 2 and 3 of the first flat subselector, $s_{1,2}$, to obtain the corresponding subselector, $s_{2,2}$. Furthermore, the original password P_1 is transformed into the new password P_2 by using password conversion function f. The configuration of $s_{2,2}$ is 11110011 (243 in decimal notation), thus we have $P_2 = f_{243}(P_1)$.

2.3 Handle reduction

Handle weakening may well be iterated. For instance, consider a subject that received a handle, and is aimed at transmitting this handle in a weakened form. If the handle is

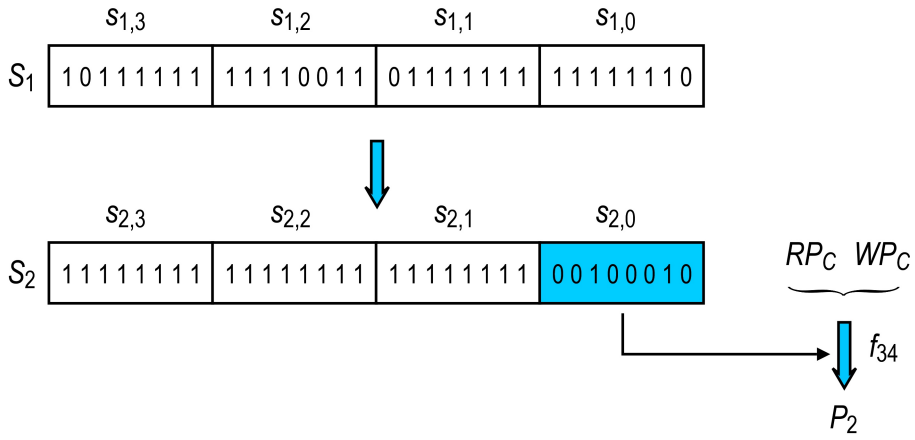

Figure 4: Reduction of handle $H_1 = (C, P_1, S_1)$ into handle $H_2 = (C, P_2, S_2)$. All subselectors of S_2 are flat except the first subselector, $s_{2,0}$. Password P_2 derives from primary password RP_C, or, if the original handle H_1 is to write, from primary password WP_C, by application of password conversion function f.

the result of a previous weakening action, we are in the presence of two weakening steps. The handle weakening algorithm implies a modification of the first subselector that is flat. If no flat subselector is available, the handle weakening should be preceded by a handle *reduction*. This is the action of transforming a handle into an equivalent handle in which all subselectors are flat except the first subselector.

Let us denote the original handle by $H_1 = (C, P_1, S_1)$ and the reduced handle by $H_2 = (C, P_2, S_2)$. The reduction procedure is as follows:

1. The new segment selector S_2 is assembled by using the subselectors of the original segment selector S_1. We have $s_{2,0} = s_{1,0} \wedge s_{1,1} \wedge \ldots \wedge s_{1,m-1}$. All the other subselectors of S_2 are flat.
2. The new password P_2 is evaluated by using password conversion function f. As S_2 has a single non-flat subselector, we have $P_2 = f_{s_{2,0}}(RP_C)$ or, if the original handle H_1 is to write, $P_2 = f_{s_{2,0}}(WP_C)$.

Figure 4 shows an example of a handle reduction. In this example, bits 1 and 5 are asserted in all the subselectors of the original handle $H_1 = (C, P_1, S_1)$. It follows that the configuration of first subselector $s_{2,0}$ of the reduced handle $H_2 = (C, P_2, S_2)$ is 00100010 (34 in decimal notation). All the other subselectors are flat. The new password P_2 of H_2 is evaluated by application of password conversion function f, starting from the primary password used in H_1. We have $P_2 = f_{34}(RP_C)$, or, if H_1 is to write, $P_2 = f_{34}(WP_C)$.

2.4 Handle revocation

Handle revocation is based on the replacement of a primary password. Let RP_C and WP_C

be the primary passwords of cluster C. If we replace one of these password, say RP_C, we revoke primary handle $RH_C = (C, RP_C, 11\ldots1)$ defined in terms of this primary password, and all the copies of RH_C. Furthermore, we revoke all the handles that have been derived from RH_C by weakening, and their copies, independently of the present locations of these handles in the distributed memory. In fact, after password replacement, validation of any handle defined in terms of the old password is destined to fail.

Replacement of a primary password is a general revocation mechanism, which involves all the handles for a given cluster in the same access mode. A different mechanism, for a more limited revocation, takes advantage of the possibility to define two or more segments that overlap in memory. For instance, let A denote an area in the shared memory of node N, and suppose that two segments c_1 and c_2 have been defined in cluster C in terms of this memory area. Let $H_1 = (C, P_1, S_1)$ be a handle for cluster C, whose selector indicates segment c_1, and similarly for handle $H_2 = (C, P_2, S_2)$ and segment c_2. In a situation of this type, by deleting c_2 we revoke H_2, whereas the validity of H_1 remains intact. Subsequently, it will be possible to access memory area A by using H_1, but this is no longer true for H_2.

3 THE PROTECTION SYSTEM

We shall now describe a conceptual scheme for the implementation of a distributed protection system based on clusters and handles. We make no hypothesis on the configuration of the underlying hardware. Instead, our system is designed to be fully implemented at software level, supported by a conventional processor architecture. In particular, no modification of the processor instruction set is necessary, and no specialized hardware is required for memory protection, a single exception being the support for the two traditional processor modes, a privileged mode and a user mode with memory access limitations.

3.1 The protection tables

In each given node N, a *password table* PT_N contains the primary passwords of all the clusters hosted in that node. This table is stored in the memory area reserved for the protection system. It features one entry for each cluster. The entry for cluster C contains the two primary passwords RP_C and WP_C associated with this cluster.

Furthermore, a *cluster table* CT_C is associated with cluster C in N. This table features n entries, one entry for each potential segment in the cluster, reserved to contain the segment base and length. Initially, when C is activated in N, all the entries of CT_C are cleared (the value of the length field is 0). When segment c_i is allocated in C, the base and length of the new segment are inserted into the i-th entry of CT_C.

Table 1: The protection primitives.

$(RH_C, WH_C) \leftarrow newCluster(RH_{C_0})$
In the current node, activates a new cluster C, and returns the primary handles RH_C and WH_C of this cluster. RH_{C_0} is the read primary handle of cluster C_0 of the current node.

$deleteCluster(WH_{C_0}, L)$
In the current node, deletes the cluster whose local name is L. WH_{C_0} is the write primary handle of cluster C_0 of the current node.

$newSegment(RH_C, i, b, t)$
In the current node, allocates segment c_i of the cluster C referenced by read primary handle RH_C. Arguments b and t are the base and the length of the new segment.

$deleteSegment(WH_C, i)$
In the current node, deletes segment c_i of the cluster C referenced by write primary handle WH_C.

$readSegment(H_C, i, addr)$
Copies the contents of segment c_i of the cluster C referenced by handle H_C into an area starting at address $addr$ of the private memory of the current node. H_C should permit read access to c_i.

$writeSegment(H_C, i, addr)$
Replaces the contents of segment c_i of the cluster C referenced by handle H_C with quantities taken from an area starting at address $addr$ of the private memory of the current node. H_C should permit write access to c_i.

$H_2 \leftarrow weakenHandle(H_1, msk)$
Returns a handle H_2 that references the subset specified by msk of the segments included in handle H_1, in the same access mode.

$H_2 \leftarrow reduceHandle(H_1)$
Returns a handle H_2 derived from handle H_1 by reduction.

$PH_2 \leftarrow newPassword(PH_1)$
Changes the primary password of the cluster C specified by primary handle PH_1. Returns a primary handle PH_2 defined in terms of the new primary password, in the same access mode.

3.2 The protection primitives

The process interface of the protection system consists of a set of primitives, the *protection primitives*. Table 1 summarises the effects of the execution of each primitive. For a few primitives, execution is completely accomplished within the boundaries of the node where the primitive is issued. Other primitives imply forms of cooperation between nodes, by message exchanges. A message can be a *control message* or a *data message*. A control message can be a *request message* specifying actions that must be accomplished in the destination node, or a *reply message* that contains information concerning the result of the execution of these actions. A data message includes the contents of a segment. A message of this type is generated when a segment is accessed to read or to write.

Protection primitives are intended to be implemented in the form of system routines, which are executed in the privileged mode. This is necessary, in particular, to access the protection tables, which are stored in the memory area reserved for the protection system.

In the rest of this section, we shall present the actions caused by the execution of each protection primitive, with special reference to access right checks, and to the messages

that are exchanged across the network. To simplify the presentation, we shall not mention the actions involved in handle validation, which have been illustrated in Section 2.1. Furthermore, we shall omit details concerning interprocess communications and their protocols, e.g. message encryption and message routing, as well as the usual security issues affecting these communications, e.g. prevention of forms of replay attack [29].

We shall use the term *current node* to denote the node where the given protection primitive has been issued. A *local segment* is a segment allocated in the shared memory the current node, a *remote segment* is allocated in the shared memory of a different node (a *remote node*).

3.2.1 Cluster activation and deletion

In each given node N, the cluster whose local name is 0, which we shall denote by C_0, is a fictitious cluster that is always active. Memory space is never reserved for the segments in this cluster. A subject that holds the read primary handle $RH_{C_0} = (C_0, RP_{C_0}, 11\ldots1)$ of C_0 is entitled to activate new clusters in N. If the subject holds the write primary handle $WH_{C_0} = (C_0, WP_{C_0}, 11\ldots1)$, it can delete clusters from N.

In detail, if executed in node N, the $(RH_C, WH_C) \leftarrow newCluster(RH_{C_0})$ protection primitive activates a new cluster C in N. This primitive returns the two primary handles of the new cluster, the read primary handle RH_C and the write primary handle WH_C. Execution of *newCluster* is as follows:

1. Validity of handle RH_{C_0} is verified; it should be the read primary handle of cluster C_0 of node N. If this is not the case, an exception of violated protection is raised, and execution of *newCluster* fails.
2. The local name L of the new cluster $C = (N, L)$ is generated, and a new cluster table is allocated for C, namely CT_C, in the memory area reserved in node N for the protection system. CT_C features n entries, one entry for each potential segment in C. In each entry, the length field is cleared, to indicate that the corresponding segment has not been allocated.
3. The two primary passwords of cluster C, RP_C and WP_C, are generated, and are inserted into password table PT_N.
4. The two primary handles of cluster C, $RH_C = (C, RP_C, 11\ldots1)$ and $WH_C = (C, WP_C, 11\ldots1)$, are constructed, and are returned to the caller.

In step 2, a simple strategy for generation of local cluster names is a sequential generation. In each node, a cluster counter contains the local identifier of the next cluster to be activated in that node. This counter is initialized to 1 when the protection system is initialized. After activation of a new cluster, the value of the counter is incremented by 1.

The *deleteCluster*(WH_{C_0}, L) protection primitive allows a subject running in node N to delete cluster $C = (N, L)$. Execution deletes the cluster table CT_C associated with C, and eliminates the entry relevant to C from password table PT_N. Execution terminates successfully only if WH_{C_0} is the write primary handle of cluster C_0 of node N.

We wish to point out that the aims of primitives *newCluster* and *deleteCluster* are restricted to cluster activation and deletion within the boundaries of the node where these primitives are issued. In fact, a subject running in a given node cannot activate a cluster in a remote node or delete an existing cluster from a remote node. *newCluster* allocates no segment in the new cluster; segment allocation will be carried out by using the *newSegment* primitive, which is introduced below. In contrast, by deleting the cluster table, *deleteCluster* deletes all the segments in the cluster.

3.2.2 Segment allocation and deletion

Protection primitive *newSegment*(RH_C, i, b, t) allocates a new segment in the current node, in the cluster C referenced by read primary handle RH_C. Argument i is the index of the new segment c_i in the cluster, arguments b and t are the segment base and length. Execution is as follows:

1. Validity of handle RH_C is verified; if it is not a read primary handle, an exception of violated protection is raised, and execution of *newSegment* fails.
2. Quantities b and t are considered to verify that the new segment can be completely contained within the boundaries of the shared memory of the current node. If this is not the case, execution fails.
3. The i-th entry of segment table ST_C is accessed. If the length field of this entry is not cleared, then segment c_i has already been allocated, and execution fails. Otherwise, quantities b and t are inserted into the base and length fields, respectively.

As anticipated in Section 2.4, the same storage unit can be part of two or more segments at the same time, and in fact, *newSegment* does not prevent segments to overlap. After execution of this primitive, it will be possible to access the new segment to read or to write, by taking advantage of a handle referencing this segment in the corresponding access mode, e.g. a primary handle referencing cluster C, or a handle generated from a primary handle by weakening.

Protection primitive *deleteSegment*(WH_C, i) deletes the i-th segment c_i from the cluster C of the current node that is referenced by handle WH_C. Execution clears the length field of the i-th entry of segment table ST_C. Execution terminates successfully only if WH_C is a write primary handle.

It should be noted that *deleteSegment* does not alter the contents of the memory area

which was reserved for the deleted segment. This means that if we have two or more overlapping segments, and we delete one of them, the contents of the other segments are not affected.

Furthermore, *newSegment* cannot be used to change the base or the length of an existing segment. An effect of this type can be obtained by using *deleteSegment* to delete the segment, and then taking advantage of *newSegment* to create a new segment with the desired characteristics. It follows that a modification of a segment in a given cluster requires possession of both the read primary handle and the write primary handle of this cluster.

Finally, the aims of *newSegment* and *deleteSegment* are restricted to segment allocation and deletion in the shared memory of the node where these primitives are issued. In fact, as will be illustrated shortly, a subject running in a given node that holds adequate access permissions can access the remote segments to read or to write only; it cannot create a new remote segment or delete an existing remote segment. Thus, memory management activities are confined within the node boundaries.

3.2.3 Accessing segments

A subject can access the contents of a segment, be it local or remote, only by presenting a handle referencing this segment. We shall now introduce two protection primitives, namely *readSegment* and *writeSegment*, which can be used to read the contents of a segment and to replace these contents, respectively. The arguments of these primitives include a handle referencing the segment involved in the access. The actions caused by the execution of these primitives will be described with reference to remote segment accesses. The activities entailed by an access to a local segment can be easily imagined, and will not be discussed.

Let $H_C = (C, P, S)$ be a handle referencing cluster C in the read mode, and suppose that C is allocated in node M. If executed in node N, protection primitive $readSegment(H_C, i, addr)$ copies the contents of the i-th segment c_i of cluster C from node M into an area starting at address $addr$ of the private memory of node N. Execution of this primitive is as follows:

1. Validity of handle H_C is verified; it should permit read access to segment c_i. If this is not the case, an exception of violated protection is raised, and execution of *readSegment* fails.
2. A request message is sent to node M. On receipt of this message, M accesses the i-th entry of the cluster table CT_C of cluster C. If the length field of this entry is cleared, then segment c_i is not allocated; a negative reply message is returned to N, and *readSegment* fails. Otherwise, a data message d is assembled including the contents of segment c_i, and the specification of the length t of this segment. This

data message is sent to N.

3. Node N copies the contents of segment c_i from data message d into a local private memory area of size t, which starts at address *addr*.

Let $H_C = (C, P, S)$ be a handle referencing cluster C of node M in the write mode. If executed in node N, protection primitive $writeSegment(H_C, i, addr)$ copies the contents of a memory area starting at address *addr* of the local private memory into the i-th segment c_i of cluster C. Execution is as follows:

1. Validity of handle H_C is verified; it should permit write access to segment c_i. If this is not the case, an exception of violated protection is raised, and execution of *writeSegment* fails.
2. A request message is sent to node M. On receipt of this message, M accesses the i-th entry of the cluster table CT_C of cluster C. If the length field of this entry is cleared, then segment c_i is not allocated; a negative reply message is returned to N, and *writeSegment* fails. Otherwise, a reply message is assembled including the specification of the length t of c_i. This message is returned to N.
3. Node N assembles a data message d including the contents A of an area of size t starting at address *addr* of the local private memory. This message is sent to M.
4. Node M copies quantity A from data message d into segment c_i.

3.2.4 Handle management

We shall now introduce a set of protection primitives for handle management. A first example is the $H_2 \leftarrow weakenHandle(H_1, msk)$ primitive, which returns a handle $H_2 = (C, P_2, S_2)$ derived from handle $H_1 = (C, P_1, S_1)$ by weakening. Argument msk, of size n bits, specifies the extent of the weakening action; for each bit that is cleared in msk, the corresponding segment is excluded from H_2. The actions caused by execution of this primitive correspond to the handle weakening algorithm of Section 2.2, in particular as specified by step 1 for the transformation of selector S_1 into selector S_2, and by step 2 for the transformation of password P_1 into password P_2.

Primitive $H_2 \leftarrow reduceHandle(H_1)$ returns a handle $H_2 = (C, P_2, S_2)$ derived from handle $H_1 = (C, P_1, S_1)$ by reduction (see Section 2.3). Suppose that cluster C is allocated in node M. The reduction procedure uses a primary password of C. Consequently, execution of this primitive in node N causes handle H_1 to be sent from N to M, where the primary passwords are stored. The actions corresponding to the reduction algorithm of Section 2.3 will take place in node M, in particular as specified by step 1 for the transformation of S_1 into S_2, and by step 2 for the transformation of P_1 into P_2.

Finally, for handle revocation (see Section 2.4), let PH_1 be a primary handle of cluster C, i.e. $PH_1 = (C, RP, 11\ldots1)$ or $PH_1 = (C, WP, 11\ldots1)$. If C is allocated in node M, execution of protection primitive $PH_2 \leftarrow newPassword(PH_1)$ accesses the entry reserved for C in password table PT_M to change the primary password specified by PH_1, RP or WP. Execution returns a primary handle PH_2 defined in terms of the new primary password. This means that replacement of a given primary password requires possession of a handle expressed in terms of this primary password.

4 EXAMPLES OF APPLICATIONS

Handles can be used effectively to solve a variety of protection problems, which correspond to different meanings associated with the concept of a segment. This section presents a few examples of applications, concerning the implementation of communication ports with priority, tree-shaped node hierarchies, and access control lists in handle-based environments. These are by no means exhaustive, but give an indication of the flexibility of the handle concept. A few considerations about segments containing handles to form segment hierarchies are presented, too.

4.1 Communication ports

Let us consider a system featuring up to n communication ports connecting a set of clients with a server. Each port is unidirectional; it permits data transmission from the server to the clients, or, if the port is for write, in the opposite direction, from the clients to the server. A priority is assigned to each port and each client, in the range from 0 (the highest priority) to $n-1$. Port priorities are unique; it is never the case that two ports exhibit the same priority. A limitation on port usage is that a client at a given priority can access only the ports at the same or a lower priority. This means that a client at priority i can only access the ports at priorities $i, i+1, \ldots, n-1$.

In our protection model, a system of this type can be implemented by reserving a segment for each port. These segments are organized into a cluster contained in the server. The server distributes a handle to each client. This handle is obtained by weakening the read primary handle of the cluster, or, if the ports are for write, the write primary handle. For instance, in the write case, a client at priority i will be granted a handle obtained by weakening the write primary handle to specify segments $i, i+1, \ldots, n-1$. To this aim, the first subselector of this handle will have bits $i, i+1, \ldots, n-1$ asserted, and the other bits cleared. The password will be generated starting from the write primary handle by the handle weakening procedure, illustrated in Section 2.2.

A client that holds a handle for given ports can distribute this handle to other clients, thereby transmitting the right to use these ports. The handle can be weakened before

distribution. A client that receives a weakened handle can use only those ports that have not been eliminated by the handle weakening procedure.

4.2 Tree-shaped node hierarchies

Let us now refer to a distributed system for environmental observations. The system consists of nodes connected to form a network whose physical topology is inessential. At the logical level, the nodes are organized into a three-level hierarchy. The nodes at the lower hierarchical level are in contact with the external environment. They are grouped into applications according to functionality criteria. A node in a given application is called a *member* of this application. For each application, a node at the intermediate level, called the *application server*, is responsible for transforming the data collected by the application members into a form suitable for transmission to the *base station*, at the root of the hierarchy. The base station is aimed at the final presentation and delivery of the observational results of the entire network.

In an organization of this type, the server of each given application maintains a segment cluster, the *application cluster*, featuring a segment for each member of that application. The application server distributes a handle to each application member. This handle permits write accesses to the segment reserved for that member. The member takes advantage of the *writeSegment* protection primitive to communicate with the application server. In turn, the base station maintains a segment cluster with a segment for each application. The base station distributes a handle to each application server. This handle permits write accesses to the segment reserved for that application server. The application server uses this handle to communicate with the base station.

Suppose that a member of a given application should be eliminated from that application. It is necessary to revoke the access privilege that this member holds for the corresponding segment in the application cluster, to prevent it from taking advantage of these privileges any longer. To this aim, the application server simply issues the *deleteSegment* protection primitive to delete the segment. Now suppose that a new node should be added to an application. It is necessary to reserve a new segment for this node, in the cluster of that application. To this aim, the application server uses the *newSegment* protection primitive for a segment that has never been allocated. Thus, repeated actions of node deletion and addition in the same given application may well lead to exhaustion of available segments in the application cluster. In a situation of this type, a solution is to replace the primary passwords of the cluster. Consequently, all the handles derived from the old passwords are revoked, and all the free segments can be used again. A new distribution of handles to the application members will take place, to replace the revoked handles with new handles.

4.3 Handle segments

A *handle segment* is a segment reserved to contain a collection of handles (in contrast, a *data segment* only contains ordinary information items). Of course, a handle in a handle segment may well reference segment clusters contained in different nodes. In turn, these clusters may include other handle segments. In this way, handle segments and data segments form a hierarchical structure, which is distributed across the network. In this structure, the terminal nodes are data segments, the intermediate nodes and the root are handle segments. In Section 1.1.1, with reference to capability environments, we introduced the similar concept of a capability segment. In a handle-based system, less handle segments are necessary, as a single handle references an entire segment cluster.

It should be noted that a handle referencing a handle segment may indirectly grant access permissions significantly stronger than the access right included in the handle itself. Consider a subject that holds a handle granting access right *read* for a given handle segment. This subject is in the position to read the handles in the handle segment, to access the segments in the clusters referenced by these handles. On the other hand, access right *write* for a given handle segment makes it only possible to access the handle segment to modify its contents, by adding new handles, overwriting the existing handles, or deleting them. *Write* does not permit any form of access to the clusters referenced by these handles.

4.4 Access control lists

In a well-known approach, the state of a protection system is expressed by associating an *access control list* with each object [19], [24], [26]. The access control list ACL_G associated with object G consists of a set of entries having the form (D, AR), where D denotes a protection domain, and AR specifies the access rights for G, which are contained in D. In an access control list environment, it is straightforward to determine the access rights associated with domain D for object G. To this aim, in ACL_G, we shall inspect the entry that corresponds to D.

In a handle-based system, we can implement an access control list at little effort, as follows. We reserve a segment cluster C_G for ACL_G. In this cluster, we associate a segment S with each given domain D. The contents of this segment specify the access rights included in D for G. This segment consists of a single memory cell with one bit for each access right; if asserted, the given bit denotes that D includes the corresponding access right.

Let H be a handle for cluster C_G. The selector of this handle specifies one or more segments in C_G. If H specifies a single segment, e.g. segment S corresponding to domain D, then the subject that possesses H holds the access rights for G, which are specified by S. If H specifies more than a single segment, then the subject holds the *union* of the

access rights for G, which are specified by each of these segments.

Cluster C_G is part of the internal representation of object G, and consequently, it is contained in the network node storing G. A subject that holds handle H referencing C_G can issue the operations defined by the type of G. The arguments of each of these operations include a handle for C_G. The operation will use H to access C_G to verify the access rights.

5 DISCUSSION

5.1 Memory requirements for handle storage

Handles with different compositions may well reference the same set of segments. For instance, let us consider handles $H_1 = (C, P_1, S_1)$ and $H_2 = (C, P_2, S_2)$, where S_2 has been obtained by inverting the positions of two given subselectors in S_1. In a situation of this type, the segments referenced by H_1 are the same referenced by H_2, as the logical AND of the subselectors of S_1 produces the same result as for the subselectors of S_2. However, passwords P_1 and P_2 will be different, as follows from the iterative password conversion algorithm of Section 2.1 applied to S_1 and S_2. As a further example, let us consider the handle reduction process illustrated in Section 2.3. A reduced handle references the same set of segments as the original handle, but the passwords are different.

We wish to remark that the memory requirements for password storage are not increased by this apparent password proliferation. In fact, only two passwords must be kept in memory for each cluster, the primary passwords. The validity of each given secondary password will be assessed by using the password conversion algorithm. This is in sharp contrast with password capability environments, where validation is based on comparisons with pre-existing password sets.

In a capability system, a protection domain corresponds to a collection of capabilities, one for each object in that domain, which are grouped to form a capability list supported by a capability segment (see Section 1.1.1). In our system, a single handle is sufficient to specify an access right, *read* or *write*, for an entire collection of segments within the boundaries of the same cluster. This compact domain representation leads to significant memory space savings, and also to simplicity in access right management. For instance, consider a data structure consisting of two or more data segments. A single handle, instead of a capability list, is required to reference these segments. The transmission of this handle will be sufficient to transfer an access permission for the whole data structure, whereas, in a capability environment, the copy of the whole capability list is necessary. A related problem is connected with domain weakening, to eliminate the access rights for a subset of the segments. In a capability system, an action of this type corresponds to the elimination of capabilities in a capability list. In our handle-based environment, the same result is

Table 2: Memory requirements for handle storage (in bytes).

Cluster size	small ($n = 4$)	standard ($n = 8$)	large ($n = 16$)
Cluster name	3	3	3
Password	16	16	16
Subselector	0.5	1	2
Selector ($m = 4$)	-	4	8
Handle ($m = 4$)	-	23	27
Selector ($m = n - 1$)	2	7	30
Handle ($m = n - 1$)	21	26	49

obtained by a handle weakening action, as is supported by the *weakenHandle* protection primitive.

As seen in Section 2, the number m of the subselectors is defined at system level. For a cluster consisting of n segments, if $m = n - 1$, then handle reduction is never necessary, even in the case of a handle weakened to reference a single segment. In a practical implementation, supporting as many as $n - 1$ subselectors tends to be a wastage of memory resources. In fact, a handle is usually weakened before being transmitted to a different subject, to reduce the extent of the access permissions of the recipient, but it is rarely the case that a handle undergoes many transmission steps with weakening. This suggests us to limit quantity m. Of course, if a handle should be weakened and no flat subselector is available, we shall take advantage of the *reduceHandle* protection primitive to reduce the handle.

In a practical implementation, we shall support clusters of different sizes, corresponding to different values of n. For instance, we may have small clusters of up to four segments, standard clusters of up to eight segments, and large clusters of up to 16 segments (Table 2). A cluster name of three bytes (two bytes for the node name and one byte for the local cluster name) will be sufficient to support a high connectivity between nodes in a large network. The password size derives from the overall protection requirements, e.g. 128 bits. For small clusters, the selector fits into two bytes, and the resulting handle size is 21 bytes. If $m = 4$, for standard clusters the selector size is four bytes, and the handle size is 23 bytes; for large clusters, the selector size is eight bytes, and the handle size is 27 bytes. In contrast, if $m = n - 1$, for standard clusters we have a selector size of seven bytes and a handle size of 26 bytes; for large clusters we have a selector size of 30 bytes, and a handle size of 49 bytes.

Let us now compare these results with the analogous memory requirements in a segment-oriented password capability environment. A password capability is a pair (G, P), where G is a segment identifier and P is a password. If the size of a segment identifier is 4 bytes (two bites for the name of the node where the segment is allocated, and two

bytes for the local name of the segment in that node), for 128-bit passwords the size of a password capability referencing a single segment is 20 bytes. In contrast, in our handle-based environment, if $m = 4$, the size of a large handle is 27 bytes (see Table 2), and the handle can specify access permissions for up to 16 segments. This important result has been obtained by encoding information concerning the segments actually addressed by a given handle at the password level, taking advantage of the parametric one-way password conversion function.

5.2 Execution times for handle validation

In the iterative algorithm for handle validation, introduced in Section 2.1, let T_f denote the average time necessary for the execution of a single iteration, including the execution time of password conversion function f, and let T_v denote the total handle validation time. We have $T_v = k \cdot T_f$, where k is the number of the subselectors that are not flat in the handle to be validated. For a primary handle, we have $k = 0$, and the validation time $T_{v,min}$ is marginal. For a handle that has been reduced by using the reduction procedure of Section 2.3, we have $k = 1$ and $T_v = T_f$. The maximum time cost corresponds to the case of no flat subselector, and in this case $T_{v,max} = m \cdot T_f$.

5.3 Network costs

As seen in Section 3.2, the actions caused by the execution of several protection primitives are completely confined within the boundaries of the current node, where the given primitive is issued, at no network cost. This is the case, for instance, for primitives *newCluster* and *deleteCluster*, which make it possibile to activate and delete clusters locally in the current node, and for primitives *newSegment* and *deleteSegment*, which allow us to allocate and delete segments in clusters of the current node. In fact, the procedure for validation of a given handle is entirely confined within the node of the cluster referenced by that handle. Of course, network traffic is generated by the execution of primitives *readSegment* and *writeSegment*, when they are used to access a remote segment.

Handle reduction requires a knowledge of the primary password corresponding to the access mode (to read or to write) permitted by the given handle. It follows that a subject that holds a handle cannot reduce this handle autonomously. Instead, intervention of the protection system is required, as is supported by the *reduceHandle* primitive, and the handle transformation will take place remotely, in the node storing the corresponding primary password. On the other hand, execution of primitive *weakenHandle* is completely confined in the node where this primitive is issued. No network traffic is produced. This important result follows from our approach to handle weakening, based on utilization of the password conversion function.

5.4 Forging handles

Let us suppose that a malevolent subject B is aimed at forging a handle $H = (C, P, S)$ for cluster C. The configuration of selector S will be set in relation to the segments of C that the handle should reference, e.g. S will be flat (all 1's) if H should reference all the segments. Subject B does not know the password (a primary password, in the case of a flat S), and will use a password chosen at random. If passwords are large and sparse, the probability of a casual match is virtually null.

Let us now suppose that subject B possesses a valid handle $H_1 = (C, P_1, S_1)$ that references a few segments of cluster C, and is aimed at modifying this handle into handle $H_2 = (C, P_2, S_2)$ that references more segments. Transforming S_1 into S_2 is an easy task. For instance, B will replace the most significant non-flat subselector of S_1 with a flat subselector in S_2. The next step is to transform P_1 into P_2. In fact, P_2 *precedes* P_1 in the iterative password conversion procedure, introduced in Section 2.1, which starts from a primary password to obtain the password corresponding to the selector. But password conversion function f is one-way, and it is computationally unfeasible to invert it to evaluate P_2. In this case, too, a solution is to use a password chosen at random, but the probability of success is vanishingly low.

5.5 Handle revocation

In Section 2.4, we introduced a handle revocation mechanism based on the replacement of a primary password, and supported by the *newPassword* protection primitive. Despite its simplicity, this mechanism is characterized the important properties introduced in Section 1.1.3 [8]. Revocation is transitive, as the effects of a revocation propagate to all the subjects that hold a handle derived from the password being replaced. In fact, by changing a given primary password, we revoke the primary handle defined in terms of this password, all its copies, all the handles derived from this primary handle by weakening, and all their copies. Thus, revocation extends to the whole distributed system, across the node boundaries. Revocation is partial, as it can be limited to a single access right, *read* or *write*; and in fact, the replacement of a given primary password does not impair the validity of the handles defined in terms of the other primary password. Revocation is temporal, as it can be reversed by restoring the original primary password.

As seen in Section 2.4, a further revocation mechanism is based on two or more segments defined in terms of the same memory area. By issuing the *deleteSegment* protection primitive to delete one of these segments, we revoke all the handles referencing the memory area in terms of this segment. In this case, too, revocation is transitive (it extends to all the copies of these handles in the whole distributed system), and temporal (it can be reversed by restoring the deleted segment). Furthermore, revocation is independent, that

is, handles for the same memory area can be revoked independently of each other.

5.6 Relation to previous work

Capability protection is a multi-decade idea [15]. Several capability-based systems were designed and actually implemented in the past, and solutions to the related protection problems were conceived at both the hardware and the software levels. This section takes a few recent systems into consideration, namely Walnut, Annex and CHERI. For each of them, the protection environment will be briefly discussed. The aim of this presentation is a comparison with our system; the accent is on the protection of the distributed information at a high level of granularity.

5.6.1 Walnut

Walnut [2], [21] is a design and implementation effort aimed at demonstrating the feasibility of using off-the-shelf microprocessors in a tightly-coupled multiprocessor based on password capabilities. In the Walnut protection model, a volume name is associated with each physical storage device. An object stored in a given volume is univocally identified by the name of this volume and a serial number, local to the volume, and valid for the entire object lifetime. A capability consists of an object identifier and a password. Passwords are sparse and generated at random by a physically random number generator. Each capability is associated with a fixed access privilege expressed in terms of a set of access rights. No computable relation exists between the password and the access rights. Instead, in each volume, a capability table contains the mapping from passwords to access rights for all the objects stored in that volume.

When an object is created, a master capability is assigned to this object, with the access rights specified by the creator. New capabilities, with reduced access rights, can be derived from the master capability by a call to the kernel. Capability derivation is recursive. The resulting interdependencies can be modeled as an inverted tree whose root is the master capability, and the other nodes are the derived capabilities. If a capability is destroyed, all its descendants are destroyed, too. When the master capability is destroyed, the object becomes inaccessible, and is deleted.

In our protection model, with reference to distributed systems, we concentrate on controlled communication, as is supported by the protection primitives. A single handle is sufficient to reference an entire cluster of segments. This is important from the points of view of handle manageability and the memory requirements for handle storage, especially for small sized segments (an example of application to single-cell segments has been given in Section 4.4). A subject holding a handle for a given cluster can weaken this handle autonomously, to produce a new handle with less access permission. No intervention is

required by the kernel, and no network traffic is generated to communicate with the node storing the cluster.

5.6.2 Annex

In the Annex object capability system [9], [22], computer security is pursued by using password capabilities and message passing primitives to control the interactions between otherwise isolated components. Capabilities are combined with object-oriented programming. Complex systems are decomposed into component objects, and capabilities are used to address each of these objects to implement fine-grained security policies. Objects are the fundamental, self-contained units of strong isolation, unable to access any information item outside of their boundaries. However, an object may hold capabilities to communicate with external objects by message passing. An object resides in its own address space, where the object code and state are stored.

Annex is an event-driven distributed system. An event can be a hardware interrupt, or a message from an object. Each object is idle until it receives a message. Object methods operate according to the call-by-copy semantics. The only way to transmit a reference between objects is by passing a capability. Method calls are asynchronous. An object that issues a method call is not forced to wait for the result; instead, it can continue to operate while the call is processed.

In Annex, a password capability for a given object consists of the identifier of the device storing the object, the identifier of the object in that device, the identifer of a capability in the set of capabilities for that object, and a password that prevents the forging of valid capabilities. Each device contains a catalogue of all the capabilities that reference the objects stored in that device. Capabilities can only reside within the boundaries of the kernel. The kernel associates a capability list with each object. Objects do not possess capabilities, to prevent capability alteration or forging; instead, outside the kernel, capabilities can only be referenced by using handles. A handle is an index into a capability list.

In contrast, in our system, we support a single object type, the segment. Segments are grouped into clusters to the aim of referencing, which is obtained by using handles. In Section 4, we have demonstrated the flexibility of the handle and cluster concepts in the solution of a variety of protection problems. Handles can be weakened to include less privileges. No indirection is necessary to reference a handle from outside the kernel. Instead, as seen in Section 5.4, handles are protected from alteration and forging by the one-way property of the password conversion function.

5.6.3 CHERI

CHERI [31], [34] is a hybrid capability system aimed at extending the 64-bit MIPS IV instruction set architecture. Emphasis is given to the architectural support for fine-grained memory protection, in contrast with the coarse-grained protection at the application level that characterizes traditional virtual memory systems. Safety is guaranteed by the fact that every memory access occurs through a capability. Key features are a capability coprocessor and a tagged memory. The coprocessor includes a set of capability registers aimed at address translation, from capabilities to virtual addresses. To access a memory segment, a capability for this segment must be preventively loaded into a capability register.

A capability consists of a base field and a length field that describe a memory segment, and a permission field that describes the access rights. Possible access rights are load and store capability, load and store data, and execute. The coprocessor interacts with the MIPS pipeline by receiving instructions, exchanging operands and sending exceptions. The load and store instructions address memory via the capability registers. The contents of a capability register can be accessed to modify the permission field of the capability contained in that register to reduce (but not to amplify) the access rights. The tagged memory guarantees capability integrity. The tag of a memory cell, if asserted, indicates that this cell contains a capability. A traditional store clears the tag.

In contrast, in our protection system, we do not rely on ad-hoc hardware for the support of memory addressing and protection. Instead, handles are fully implemented at software level. Handle integrity is guaranteed by passwords and the one-way property of the password conversion function. With respect to tagged memory solutions, significant advantages follow from the points of view of hardware compatibility with existing memory systems and hardware standardization [20]. Furthermore, we avoid the complications inherent in the necessity to propagate the tags across the cache hierarchy and to the secondary memory. Fine-grained memory protection is supported to the limit of a segment size of a single memory cell. The memory requirements for handle storage are mitigated by the fact that a handle can reference a whole segment cluster.

6 CONCLUDING REMARKS

With reference to a distributed system consisting of nodes connected by a local area network, we have considered the problems inherent in the distribution, verification, review and revocation of access permissions. We have proposed the organization of a protection system that takes advantage of a form of protected pointer, the handle. In our approach:

- A handle references one or more segments in the same given cluster in terms of a selector and a password. The selector specifies the segments, the password specifies

an access right, *read* or *write*. Segments are the basic unit of information protection and sharing between the nodes.

- Two primary passwords are associated with each cluster, corresponding to an access permission for all the segments in that cluster. A password conversion algorithm takes advantage of a parametric one-way function to generate secondary passwords corresponding to less segments.
- A small set of protection primitives makes it possible to allocate and delete segments within active clusters, and to access remote segments to read and to write.

We have obtained the following results:

- A subject that holds a handle for a given cluster can weaken this handle to reference less segments. The handle weakening algorithm can be iterated up to the limit of a handle referencing a single segment. The number of weakening steps is limited, to save memory space for handle storage. A handle reduction procedure makes it possible to transform a handle into an equivalent handle permitting application of more weakening steps.
- A single handle is sufficient to specify an access permission for an entire collection of segments, within the boundaries of the same cluster. This is in sharp contrast with capability environments, where a protection domain corresponds to a capability list including a capability for each object in that domain. Our compact domain representation leads to simplicity in access right management, and also to significant memory space savings.
- When a segment is accessed to read its contents or to replace these contents, the handle presented to certify possession of the corresponding access permission is validated in the node storing the segment. No network traffic is generated by this validation activity. Furthermore, when a handle is weakened to reference less segments, the handle weakening procedure does not require access to the primary passwords, and can be accomplished locally. We have obtained these important results by taking advantage of a parametric one-way function for password conversion.
- If primary passwords are large and sparse, the probability that a malevolent subject guesses a password to forge a valid handle is virtually null. Transformation of a handle into a stronger handle referencing more segments is prevented by the non-invertibility property of the password conversion function.
- Handle review and revocation takes advantage of the possibility to change a primary password to revoke the corresponding primary handle, and all the handles derived from this primary handle. We can also revoke access permissions for a given memory area by deleting a segment defined in terms of this area.

ACKNOWLEDGEMENT

The author thanks the anonymous reviewers for their insightful comments and constructive suggestions.

REFERENCES

[1] 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.

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

[3] 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.

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

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

[6] D. M. England. Capability concept mechanism and structure in System 250. In *Proceedings of the International Workshop on Protection in Operating Systems*, pages 63–82, Paris, France, 1974. IRIA.

[7] E. F. Gehringer. Variable-length capabilities as a solution to the small-object problem. In *Proceedings of the Seventh ACM Symposium on Operating Systems Principles*, pages 131–142, Asilomar, CA, USA, December 1979. ACM.

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

[14] A. W. Leung and E. L. Miller. Scalable security for large, high performance storage systems. In *Proceedings of the Second ACM Workshop on Storage Security and Survivability*, pages 29–40, Alexandria, Virginia, USA, October 2006. ACM.

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

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

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

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

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

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

[21] 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.

[22] 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.

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

[24] 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.

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

[26] R. S. Sandhu and P. Samarati. Access control: principles and practice. *IEEE Communications Magazine*, 32(9):40–48, September 1994.

[27] F. B. Schneider. Least privilege and more. *IEEE Security & Privacy*, 1(5):55–59, September 2003.

[28] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: a fast capability system. *ACM SIGOPS Operating Systems Review*, 34(2):170–185, 2000.

[29] M. Stamp. *Information Security: Principles and Practice.* John Wiley & Sons, Hoboken, NJ, USA, second edition, 2011.

[30] 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.

[31] 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.

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

[33] E. Witchel, J. Cates, and K. Asanović. Mondrian memory protection. *Operating Systems Review*, 36(5):304–316, 2002.

[34] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe. The CHERI capability model: revisiting RISC in an age of risk. In *Proceedings of the 41st ACM/IEEE International Symposium on Computer Architecture*, pages 457–468, Minneapolis, MN, USA, June 2014. IEEE.

[35] 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.